Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs. Sémantiques, implantations et certification.

Frédéric Gava

Université Paris 12 – Val-de-Marne U.F.R. de sciences

Numéro attribué par la bibliothèque : . . . . . .

Thèse

pour obtenir le grade de
Docteur de l'université Paris 12 – Val-de-Marne
discipline : Informatique
présentée et soutenue publiquement par

Frédéric GAVA

le 12 décembre 2005

Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs.

Sémantiques, implantations et certification.

Composition du jury

Président :	Thierry PRIOL	IRISA-INRIA Rennes
Rapporteurs :	Murray COLE	Univ. of Edinburgh
	Walter DOSCH	Univ. of Lübeck
Examinateurs :	Jocelyn SÉROT	Univ. de Clermont-Ferrand
	Anatol SLISSENKO	Univ. de Paris 12 – Val-de-Marne
Directeur :	Frédéric LOULERGUE	Univ. d'Orléans

Remerciements

Je voudrais remercier Thierry PRIOL de m'avoir fait l'honneur de présider mon jury. Je veux également exprimer toute ma gratitude à Murray COLE et Walter DOSCH pour avoir accepté d'être rapporteurs de cette thèse. Je remercie aussi vivement Jocelyn SÉROT et Anatol SLISSENKO qui me font le plaisir de participer au jury.

Frédéric LOULERGUE m'a encadré durant toutes ces années de travail. Je lui suis très reconnaissant des conseils qu'il m'a prodigués, de son soutien sans faille et de sa patience lors de mes moments de découragement. Ces années de recherches «acharnées» resteront indissociables de ses nombreux méls.

Je ne voudrais pas oublier tous les membres du projet CARAML dont l'aide m'a été très précieuse.

Je tiens également à remercier tous les membres du Département d'Informatique et particulièrement Flore TSILA, Franck POMMEREAU, Elisabeth PELZ, Fabrice MOURLIN ainsi que tous ceux avec qui j'ai participé à des enseignements.

Merci à Mathieu for its help. J'espère le lui rendre. Merci à Papa pour son abnégation dans la correction aurthografik de ce mémoire.

Merci à tout les copains, copines, compères et comparses : Antonin et Marion, Manu, Guillaume et Hélène, Anthony et Flavie, Manue, Christophe et Raluca, Julien, Louis, Steeve et Magalie, Pierre, June, Matthieu et Laurence, Alexandre, Cécile, Pierre, Aurélien et tous les autres que j'aurais pu oublier (ne soyez pas vexés). Même si cette énumeration ressemble plus à un catalogue de prénoms, ils (ou elles) se reconnaîtront.

Je remercie aussi toute ma famille pour son soutien indéfectible (merci Tonton pour l'hôtel pas trop cher dans le 13ième !). Merci Môman pour tes petits plats et ton pain d'épices...

Je tiens à remercier tout particulièrement Pazzo qui m'a accompagné durant ces trois années de «labeur» et qui ne pourra pas ronronner du résultat final. Tu me manques... Merci à Lulu pour nos prochaines aventures.

Merci à tous ceux que j'ai pu oublier et que je ne saurais citer ici. Je tiens à saluer la bonne idée de la première cellule vivante de se diviser et qui, d'une certaine manière, a permis que j'en sois à écrire ces quelques lignes aujourd'hui.

Ce travail ne serait rien sans les «allez zou, au boulot !» d'une personne dont je ne citerais point le nom. Je lui suis gré de ne pas m'en vouloir si je lui dis...

Sommaire

1 Introduction

CERTAINS problèmes, comme la simulation de phénomènes physiques ou chimiques de grande taille nécessitent des performances que seules les machines massivement parallèles peuvent offrir. L'écriture d'algorithmes pour ce type de machines demeure plus difficile que pour celles strictement séquentielles et la conception de langages adaptés est un sujet de recherche actif nonobstant la fréquente utilisation de la programmation concurrente.

En effet, la conception d'un langage de programmation est le résultat d'un compromis qui détermine l'équilibre entre les différentes qualités du langage telles que l'expressivité, la sûreté, la prédiction des performances, l'efficacité ou bien la simplicité de la sémantique. Dans le cas de langages pour le parallélisme, les propriétés ainsi obtenues diffèrent considérablement selon les choix effectués.

Afin de préciser les termes de ces choix, nous allons tout d'abord faire un rapide parcours des solutions existantes pour la programmation de machines parallèles. Puis, nous préciserons quels ont été ces choix dans le contexte des recherches qui ont précédé le présent travail. Le cadre ayant été posé, nous aborderons alors les motivations qui ont présidé aux recherches présentées dans ce tapuscrit.

1.1 Langages pour la programmation parallèle

Nous examinons dans un premier temps, deux approches extrêmes : la programmation concurrente et la parallélisation automatique ; puis, dans un second temps, des approches intermédiaires : le data-parallélisme, les patrons algorithmiques et enfin les extensions non-concurrentes de langages fonctionnels.

1.1.1 Approches extrêmes

Programmation concurrente. L'approche la plus répandue pour la programmation parallèles est la programmation concurrente. Elle implique la combinaison d'un langage séquentiel (usuellement C ou Fortran) avec une bibliothèque de communications par passage de messages (à la base avec deux opérations, l'envoi d'un message et sa réception) comme MPI [245] (*Message Passing Interface*) ou PVM [114] (*Parallel Virtual Machine*).

Ce style de programmation est évidemment très expressif car le programmeur définit non seulement l'algorithme parallèle mais aussi les détails de la réalisation des communications *via* des protocoles. Cette liberté va de pair avec une difficulté de mise au point des programmes. La complexité des programmes devient telle, qu'il est souvent impossible de prévoir correctement leurs temps de calcul. De plus, les risques d'indéterminismes et d'interblocages rendent la validation formelle des programmes trop complexe [7] et gêne leur portabilité. Notons que ces problèmes demeurent néanmoins dans les langages concurrents de plus haut niveau comme Jocaml [72, 120] ou Concurrent ML [217] où, en plus, des problèmes d'implantation des langages se présentent (le π-calcul synchrone [210] est, par exemple, considéré comme impossible à implanter efficacement).

Parallélisation automatique. Dans le spectre des solutions possibles aux problèmes de grandes tailles, la programmation séquentielle, *via* une parallélisation automatique du code, a été longuement étudiée. Bien qu'idéale pour sa facilité d'écriture, elle est surtout limitée à l'utilisation de tableaux et de boucles [55], ce qui oblige le compilateur à découvrir le parallélisme implicite et à produire le meilleur programme parallèle possible à partir d'une spécification qui est parfois, dans le pire des cas, strictement séquentielle. Le problème étant, en général, indécidable, des heuristiques sont utilisées. Mais vu la subtilité de certains algorithmes parallèles [156], cette méthode ne peut être utilisée de façon générale et satisfaisante.

On retrouve aussi cette approche dans la programmation fonctionelle avec notamment la réduction en parallèle de graphes [2]. Ces systèmes n'expriment pas directement les algorithmes parallèles et ne permettent pas la prévision des temps d'exécution car les processus physiques y sont implicites et la stratégie d'évaluation parallèle n'est pas décrite par la sémantique. De plus, leur extensibilité à un grand nombre de processeurs est limitée par la structure de graphe qui est partagée et par l'utilisation d'un glaneur de cellules parallèle (aussi appelé «ramasse miettes» ; *Garbage Collector* en anglais). Enfin, et cela quel que soit le langage séquentiel employé, le coût de l'administration des tâches (qui sont créées dynamiquement) peut être largement supérieur à celui de l'exécution proprement dite des tâches lorsque celles-ci sont trop petites (problème dit de *granularité*).

Il paraît donc indispensable de mettre au point des méthodes de programmation parallèle plus générales que la parallélisation automatique du code (ou que de la réduction parallèles de graphes) mais aussi plus simples que la programmation concurrente. La conception d'un langage parallèle est alors un compromis entre la possibilité pour le programmeur de contrôler les aspects nécessaires à une prédiction précise des performances (mais sans rendre les programmes trop difficiles à écrire) et l'abstraction de ces fonctionnalités qui sont nécessaires à la simplification de l'écriture des programmes parallèles (mais sans les rendre inefficaces).

1.1.2 Approches intermédiaires

Parallélisme de données. Dans ce paradigme, un programme décrit une séquence d'actions sur des tableaux à accès parallèles [37, 44, 168]. Le modèle de programmation est synchrone. Il offre une plus grande facilité de conception des programmes que dans le cas concurrent : le non blocage est garanti. De plus, les possibilités d'indéterminisme sont assez réduites pour que les systèmes de preuves de programmes [45] aient une complexité proche des systèmes traitant de programmes séquentiels. Il est aussi prévu que l'exécution d'un programme data-parallèle puisse être désynchronisée par le compilateur ou le système dynamique en vue de maximiser ses performances [168].

Dans ce domaine, le modèle BSP [34, 242, 263] (*Bulk-Synchronous Parallelism*) vise à maximiser la portabilité des performances en ajoutant une notion de processus explicites au parallélisme de données. Un programme BSP est écrit en fonction du nombre de processeurs de l'architecture sur laquelle il s'exécute. Le modèle d'exécution BSP sépare synchronisation et communication et oblige les deux à être des opérations collectives. Il propose un modèle de coût fiable (et simple) permettant de prévoir les performances de façon réaliste et portable (le modèle BSP a été utilisé avec succès pour une large variété de problèmes).

Squelettes algorithmiques. Une autre approche, intermédiaire entre la programmation concurrente et séquentielle, est celle des langages à patrons aussi appelés squelettes algorithmiques [69, 70, 86, 99, 100, 221, 226, 249, 250, 251] (*algorithmic skeletons* dans la littérature anglo-saxonne). Un ensemble *fixé* d'opérations sont exécutées en parallèle. Leurs sémantiques fonctionnelles sont explicites mais leurs sémantiques opérationnelles parallèles sont implicites. Du point de vue du programmeur, le style de programmation associé revient à l'utilisation de combinateurs dont l'effet sur la parallélisation est défini extérieurement.

Les patrons ne sont pas, en général, n'importe quelles opérations parallèles, mais essaient de capturer l'essence éprouvée de techniques de la programmation parallèle telles que les algorithmes de *pipeline* parallèles, les algorithmes maîtres-esclaves ou l'application d'une fonction à des collections de données réparties *etc.* Les avantages par rapport aux extensions concurrentes sont bien sûr que le déterminisme et le non blocage sont garantis.

Cependant, le concepteur de bibliothèques de patrons doit faire face à deux problèmes majeurs qui sont opposés. D'une part, il doit fournir un ensemble de patrons le plus complet et le plus expressif possible. Trop restreint, celui-ci serait inutilisable en pratique. Par exemple, la bibliothèque de [69] ne proposait que quatre patrons, non combinables entre eux. D'autre part, il faut que cet ensemble soit le plus petit possible car il faut implanter efficacement chaque patron sur chaque type ciblé de machine parallèle. De plus, s'il y a pléthore de patrons, le programmeur qui en jouit, aurait les plus grandes difficultés à choisir celui qui conviendrait le mieux à son problème.

Pour aider le programmeur, il existe des méthodologies de transformation de programmes qui permettent de remplacer un patron par un autre plus efficace ou une composition de patrons par un seul patron. Ces méthodologies reposent sur des modèles de performances mais souvent :

- Celui-ci est trop abstrait (comme par exemple le modèle PRAM, *Parallel Random Access Machine*, [110]) et donc inapplicable à certaines architectures cibles ;

- Il est trop détaillé et donc en pratique différent pour chaque architecture cible. La transformation faite pour une architecture peut ne plus être valable pour une autre architecture ce qui, bien entendu, ne facilite pas la portabilité.

Par exemple, une méthodologie [3] de transformation de programmes a été proposée pour les patrons de la bibliothèque P3L [15]. Celle-ci s'appuie sur un modèle de coûts dérivé de logP [79]. Mais les combinaisons que l'on peut en faire sont cependant trop complexes pour que des coûts réalistes soient donnés. Un autre exemple est le travail de [87] qui propose des patrons permettant de coder directement le graphe des processus mais dont les combinaisons posent des problèmes de portabilité des performances.

Plus récemment, une proposition vise à intégrer les approches des patrons algorithmiques avec les pratiques plus répandues de programmation parallèle comme MPI [70]. Ceci permet à l'utilisateur de pouvoir programmer à l'aide de MPI les algorithmes qu'il n'est pas possible d'implanter à l'aide des patrons qui ont été fournis. Bien entendu, on retrouve alors les problèmes que la programmation MPI peut apporter. Il paraît notamment difficile de pouvoir donner un modèle des performances simple et portable. D'autres travaux proposent des modèles des performances simples et portables basés sur le modèle BSP [71, 141, 142, 241] mais en ayant complètement reconsidéré les ensembles de patrons à employer.

Extensions explicitement parallèles non-concurrentes. Les langages de programmation fonctionnelle offrent des mécanismes d'abstraction tels que les fonctions d'ordre supérieur, le polymorphisme, les types ou le filtrage qui facilitent grandement l'écriture des programmes parallèles. D'autres approches étendent des langages fonctionnels par des primitives ou des annotations pour le parallélisme, tout en évitant de tomber dans des sémantiques concurrentes.

Par exemple, les I-structures de [9], NESL [38], Eden [167], Gph [260] (Glasgow Parallel Haskell) ou Caml-Flight [108] étendent un langage fonctionnel avec des opérations de communication tout en préservant le déterminisme du langage.

Cependant, ces extensions ne suivent pas réellement un modèle de coût et leurs sémantiques sont parfois non fonctionnelles. Eden et Gph mettent en œuvre des opérations de créations de tâches parallèles et leur placement implicite. Le choix de la stratégie d'évaluation de ces tâches est laissé au compilateur et les performances ne sont pas toujours à la hauteur des espérances. L'avantage de ces langages par rapport à ceux à patrons est une plus grande souplesse pour la gestion du parallèlisme (permettant, entre autres, une écriture des patrons dans ces langages [29]) et une facilité d'implantation. *A contrario*, les langages à patrons ont des sémantiques plus simples et compositionelles.

Bulk Synchronous Parallel ML et BSλ-calcul. Dans des travaux antérieurs à cette thèse, il a été entrepris d'approfondir la position intermédiaire que le paradigme des patrons occupe. Toutefois il ne s'agissait pas de concevoir un ensemble *a priori* fixé d'opérations puis de concevoir des méthodologies pour la prédiction des performances, mais de fixer un modèle de parallélisme structuré (avec son modèle de coûts) puis de concevoir un ensemble universel d'opérations permettant de programmer n'importe quel algorithme de ce modèle. L'objectif est donc le suivant : parvenir à la conception de langages universels dans lesquels le programmeur peut se faire une idée du coût à partir du code source. Cette dernière exigence nécessite que soient explicites, dans les programmes, les lieux du réseau statique des processeurs de la machine. En d'autres termes, les programmes seront écrits en fonction du nombre de processeurs et l'on pourrait déterminer «immédiatement» à la lecture du code source sur quel processeur se trouve telle partie du programme et des données.

Le modèle choisi dans [196] est le modèle BSP. Une approche opérationnelle a amené à définir une extension du λ-calcul par des opérations parallèles BSP. Ce calcul permet d'écrire n'importe quel algorithme BSP, c'est en ce sens qu'il est universel. Une bibliothèque basée sur ce calcul, la BSMLlib [189] (nous utiliserons aussi l'acronyme BSML) a été développée pour le langage Objective Caml [62, 229, 276] (OCaml). Elle permet la programmation data-parallèle basée sur une structure de données parallèle qui est polymorphe. Les programmes sont des fonctions (séquentielles) que l'on peut programmer en OCaml mais qui manipulent cette structure de données parallèle à l'aide d'opérations dédiées. Elle suit le modèle d'exécution BSP. Les inter-blocages sont donc impossibles. La prévisibilité des performances y a été vérifiée [135] et, étant basée sur un calcul confluent, elle est déterministe. Enfin, contrairement à l'approche des langages à patrons une sémantique parallèle explicite a été proposée [187].

L'implantation parallèle de BSML (il existe aussi une implantation séquentielle qui donne les mêmes résultats que celle parallèle) est écrite sous la forme d'un programme SPMD (*Single Program Multiple*

Data) en OCaml. Il a été conçu dans [187] une sémantique proche de cette implantation et son équivalence avec une variante du BSλ-calcul a aussi été prouvée [186].

1.2 Le projet CARAML et les travaux menés

Cette thèse s'inscrit dans le cadre du projet «CoordinAtion et Répartition des Applications Multiprocesseurs en objective camL» (CARAML[1]) de l'ACI (Actions Concertées Initiatives) Globalisation des ressources informatiques et des données (ACI GRID) dont l'objectif était le développement de bibliothèques pour le calcul haute-performance et globalisé autour du langage OCaml. Il a été fourni des bibliothèques de primitives parallèles et globalisées ainsi que des bibliothèques applicatives orientées bases de données et calcul numérique. Ce projet était organisé en trois phases successives :

Phase 1 : Sûreté et opérations data-parallèles irrégulières mono-utilisateur ;

Phase 2 : Opérations de multitraitement data-parallèle ;

Phase 3 : Opérations globalisées pour la programmation de grilles de calcul.

Les contributions de l'auteur dans chacune des phases de projet sont les suivantes :

Phase 1, sémantiques et certification : une étude sémantique d'un langage fonctionnel pour la programmation BSP et la certification des programmes écrits dans ce langage ;

Phase 2, extensions et applications : l'ajout d'une primitive de composition parallèle (et qui permet la programmation d'algorithmes «diviser-pour-régner» parallèles[2]), de bibliothèques applicatives de structures de données et d'une extension pour les entrées/sorties parallèles en BSML ;

Phase 3, opérations globalisées : la conception d'un langage à deux niveaux pour le méta-calcul, un niveau est basé sur BSML et un autre sur un langage dérivé de BSML sans barrières de synchronisation.

1.2.1 Sémantiques et certification

La participation de l'auteur à la première phase du projet Caraml a débuté (lors d'un stage d'initiation à la recherche de Master) par la conception d'un système de typage polymorphe [R3] qui empêche l'emboîtement des structures de données parallèles. Ce travail n'est pas présenté dans ce mémoire. Cet aspect sûreté a ensuite été travaillé selon deux axes complémentaires :

1. La conception de sémantiques de BSML prouvées équivalentes afin de garantir l'équivalence du modèle de programmation et du modèle d'exécution de BSML ;
2. La correction de programmes BSML.

Sémantiques de BSML. Le premier axe relève d'une préoccupation classique en programmation data-parallèle [246]. Un modèle de programmation est présenté au programmeur dans lequel un programme est une séquence d'opérations sur une structure de données parallèle. Le modèle d'exécution, lui, est plus complexe puisque qu'un programme est alors vu comme une composition parallèle de p copies d'un même programme SPMD échangeant des valeurs par passage de messages. Il faut alors garantir que le modèle d'exécution et le modèle de programmation sont équivalents. La formalisation du modèle d'exécution reste néanmoins éloignée de l'implantation du langage. Dans le cas de BSML le modèle d'exécution est formalisé par une machine abstraite parallèle, très proche de l'implantation du langage.

Correction de programmes BSML. Le seconde axe est encore plus classique dans son objectif. En effet, le souci de prouver que des programmes vérifient leur spécification est ancien [149]. Les programmes *critiques* (où par exemple des vies humaines sont en jeu) demandent des preuves formelles de leurs algorithmes afin de garantir la sécurité des matériaux gérés par les logiciels. Les bibliothèques standards des

[1] page web à `http://www.caraml.org`

[2] Le principe général de cette technique est de partitionner l'ensemble des données du problème en un nombre fini de sous-ensembles de «plus petites» tailles, chaque sous-ensemble définissant un problème indépendant. Ce partitionnement est appliqué récursivement aux sous-problèmes jusqu'à ce que chaque sous-problème soit suffisamment petit pour être directement résolu.

langages se doivent alors d'être certifiées, c'est-à-dire ne comporter que des programmes qui calculent réellement ce que l'on attend d'eux. Tester les programmes sur une grande variété d'entrées permet de détecter un certain nombre d'erreurs. Malgré tout, seules les méthodes formelles peuvent en garantir la correction.

Il existe de nombreux outils d'aide à la preuve de programmes. Certains vérifient (semi-)automatiquement les propriétés demandées aux programmes (principe des «modèle-checkers»). D'autres, au contraire, permettent, dans un modèle logique, une construction «pas-à-pas» des programmes à l'aide de commandes spécifiques (principe des assistants de preuves).

Les systèmes basés sur le λ-calcul typé[3], plus exactement sur l'isomorphisme de Curry-Howard (qui fait correspondre preuves et programmes, c'est-à-dire les λ-termes), ont l'avantage d'utiliser le même langage pour exprimer les propriétés attendues et les programmes. Le calcul des constructions [75] (qui est à la base de l'assistant de preuve **Coq** [254]), par exemple, a été enrichi de constructions permettant la définition de structures de données usuelles sous une forme proche de celle qui est utilisée par les langages de programmation fonctionnelle de la famille ML (dont fait partie OCaml). De plus, il est possible d'extraire un programme d'une preuve [180, 220]. Ce programme est alors certifié par l'assistant de preuves, d'être correct vis-à-vis de la spécification.

Les preuves de programmes concurrents sont complexes [7]. La complexité de celles des programmes data-parallèles [45] est plus faible. La preuve de programmes BSP est encore plus simple [160]. La preuve de programme BSML est donc possible et relativement simple car elle mixe le modèle BSP à la programmation fonctionnelle.

1.2.2 Extensions et bibliothèque de structures de données

Compositions parallèles. L'ordre supérieur des langages fonctionnels et la composition de fonctions permettent de composer facilement des programmes fonctionnels. Ils facilitent ainsi la programmation, rendent le code plus aisément réutilisable et diminuent le risque d'erreurs.

Au niveau du parallélisme, la composition parallèle de programmes est essentiellement réservé :

1. Soit à partager le réseau de la machine parallèle en autant de sous-réseaux indépendants, l'un pour chaque programme ;
2. Soit à partager le temps de calcul de chaque processeur, chaque programme parallèle étant exécuté sur ces temps.

La sous-synchronisation, c'est-à-dire la synchronisation d'une partie seulement des processeurs de la machine, est une fonctionnalité d'une grande variété de modèles de programmation parallèle (indépendants ou dérivés du modèle BSP [41, 89]). La présence et l'utilisation de la sous-synchronisation sont justifiées par la nécessité de pouvoir décomposer récursivement un problème en sous-problèmes indépendants les uns des autres (technique algorithmique souvent appelée «diviser-pour-régner»).

Pour évaluer deux programmes parallèles sur une même machine on peut la partitionner en deux et évaluer indépendamment chacun des programmes sur chacune des partitions. Toutefois, en procédant ainsi, le modèle BSP est perdu puisqu'alors la synchronisation globale de chaque sous-machine ne sera plus effectuée en un temps constant mais en un temps dépendant du nombre de processeurs de la sous-machine. Les auteurs de la BSPlib [147] font remarquer que le désavantage principal du partionnement par groupe de processeurs (c'est-à-dire la sous-synchronisation) est alors une perte de la prévision des performances du modèle de coût.

Pour conserver le modèle BSP, ce qui est souhaitable [133], il faut donc que les barrières de synchronisation concernent toute la machine. Le modèle BSP n'autorise donc pas la sous-synchronisation : les barrières sont collectives et cela est souvent considéré comme un obstacle pour la programmation parallèle d'algorihmes «diviser-pour-régner» en BSP (et dans notre cas, en BSML).

Pourtant, la technique dite «diviser-pour-régner», est une méthode naturelle pour la conception et l'implantation d'algorithmes pour de nombreux problèmes. La transformation manuelle d'un programme utilisant des techniques «diviser-pour-régner» en un programme sans ces techniques est un travail fastidieux où il est facile de se tromper.

Pour exprimer les algorithmes «diviser-pour-régner» en BSML, une nouvelle primitive appelée *composition parallèle* a tout d'abord été proposée [188]. Mais celle-ci était limitée à la composition de deux programmes BSML dont les évaluations nécessitaient le même nombre de super-étapes. C'était évidemment restrictif et le respect de cette contrainte échoyait au programmeur.

[3] Pour le lecteur ne connaissant pas ce paradigme, une présentation formelle du λ-calcul non typé est donnée au chapitre 3.

Cette restriction a été levée avec la *juxtaposition parallèle* [190]. Le principal problème avec cette approche est que la nature purement fonctionnelle de BSML est perdue. En effet cette primitive effectue un effet de bord sur le nombre de processeurs rendant très difficile la preuve assistée par ordinateur des programmes BSML utilisant la juxtaposition.

Dans l'article [258], l'auteur avance l'hypothèse que le paradigme «diviser-pour-régner» peut s'insérer naturellement dans le modèle BSP sans avoir besoin de la sous-synchronisation. Il propose une méhode pour la programmation «diviser-pour-régner» qui est en adéquation avec le modèle des barrières de synchronisations globales du modèle BSP. Cette méthode est basée sur un entrelacement de processus légers, appelés *super-threads*[4], chacun d'entre eux étant un processus de calcul BSP.

Une nouvelle primitive BSML, appelée *superposition parallèle* et permettant d'écrire de manière fonctionnelle ce type d'algorithmes, a été présentée dans [191]. Dans cet article, la superposition est simplement décrite en terme d'une sémantique de haut niveau (sémantique à «petits pas»). Dans ce cas, la superposition est équivalente à la création d'une paire. Nous avons approfondi l'étude de cette nouvelle primitive, contribuant ainsi à la phase 2 du projet CARAML.

Structures de données parallèles. Cette primitive a ensuite été utilisée pour le développement de bibliothèques de structures parallèles. En effet, depuis longtemps, dans la communauté informatique [275], il est connu que la bonne conception des structures de données est tout aussi importante que l'utilisation d'un bon algorithme sachant que beaucoup d'applications scientifiques sont en fait la simulation de phénomènes physiques, c'est-à-dire du calcul symbolique dans lequel les structures de données représentent le domaine physique. La notion de type de données est alors un des paradigmes centraux de la programmation moderne. La possibilité de définir de telles structures de manière primitive est un des traits caractéristiques d'un langage de programmations de haut niveau.

Il est facile de constater qu'un grand nombre d'applications pour le calcul symbolique (dans le sens algorithmique du terme) n'ont pas leur(s) version(s) parallèle(s). Pour constater ce fait, il suffit juste de compter le nombre d'applications parallèles qui existent, et ceci pour n'importe quel langage de programmation. Ainsi, les chercheurs (non informaticiens) qui écrivent des programmes ne veulent pas, essentiellement par manque de temps, avoir à faire avec la conception des algorithmes parallèles et l'utilisation de langages pour la programmation parallèle. Pourtant, ces applications scientifiques entraînent la résolution de problèmes manipulant un grand ensemble de données. Utiliser le parallélisme pourrait réduire le temps de calcul et augmenter la mémoire physique disponible pour les contenir.

Dans le but d'avoir, au final, des programmes pour le calcul symbolique fonctionnant sur des machines parallèles, les chercheurs utilisent régulièrement des compilateurs tels que «High Performance Fortran» [168] qui permettent la parallélisation automatique du code. Mais comme nous l'avions expliqué, ce genre de langages (et leurs compilateurs associés) ne fournit pas les caractéristiques réellement nécessaires au calcul symbolique, comme, par exemple, le polymorphisme et les types de données algébriques. De plus, la prédiction des performances y est impossible (ou presque). Les chercheurs peuvent aussi se tourner vers l'utilisation des langages à patrons. Malheureusement, l'ensemble des squelettes nécessaires dépend fréquemment du domaine d'application et il est souvent difficile d'augmenter cet ensemble sans avoir à tout recoder. La prédiction des performances à partir du code source n'y est pas chose aisée et la mauvaise combinaison de certains squelettes peut impliquer trop de communications, ce qui peut faire décroître fortement les performances de l'application. De plus, il est difficile de trouver des patrons pour des structures de données pré-définies.

BSML, avec la superposition parallèle, va nous permettre de programmer diverses structures de données parallèles tout en autorisant le recours aux primitives si un patron nécessaire à une application donnée est manquant.

Mémoires externes : entrées/sorties parallèles. Dans le cadre des travaux sur la sûreté de BSML, nous nous sommes intéressés à l'interaction des traits impératifs de OCaml avec les opérations parallèles BSML [C4]. Ce travail, plus orienté sûreté qu'extension des primitives BSML, a naturellement débouché sur l'étude de l'interaction des opérations d'entrées/sorties de OCaml avec les opérations parallèles de BSML. Or, cette étude, bien qu'intéressante au niveau sûreté, trouve tout son intérêt dans la définition de nouvelles primitives BSML pour les entrées/sorties parallèles sûres qui sont extrêmement importantes en pratique.

En effet, un nombre important d'applications informatiques impliquent de résoudre des problèmes de

[4] Nous préférons le terme de *super-thread* et non sa traduction littérale «super processus légers».

très grande taille [234]. De telles applications sont généralement appelées dans la littérature scientifique anglo-saxonne des «applications *out-of-core*». On trouve ce genre d'applications par exemple[5], dans la simulation de phénomènes astronomiques [255], météorologiques [266] (prévision du temps), de crashes de véhicules [68], dans le domaine des systèmes d'informations géographiques [8, 170] (cartographie) ou bien encore dans la bio-informatique [98], la manipulation de gros graphes [154, 172, 212, 261] ou l'informatique appliquée à la géométrie [78, 121].

Employer le parallélisme permet de réduire le temps de calcul d'une application mais aussi, d'augmenter la capacité de mémoire disponible. Mais pour ce genre d'application, les mémoires principales se sont pas toujours de taille suffisante pour contenir l'ensemble des données du problème. Prenons le cas du «Large Hadron Collider», l'accélérateur de particules en construction au CERN[6]. Lors de sa mise en fonction, cet instrument pourra produire environ 10 Pentabytes de données diverses par mois. Le «earth-simulator», la machine parallèle japonaise classée parmis les premières dans le top 500 des machines parallèles les plus puissantes, possède un total d'environ 1 Pentabyte de mémoire principale et de 100 Pentabytes dans ses différentes mémoires externes. N'employer que la mémoire principale n'est donc pas satisfaisant pour stocker toutes les données d'une expérience (celle-ci pouvant durer plusieurs jours). L'utilisation des disques de la mémoire externe devient alors une nécessité. Notre extension permet de le faire de façon sûre et prévisible avec BSML.

1.2.3 Opérations globalisées

Certains problèmes requièrent des performances qu'une seule machine parallèle ne peut fournir. On souhaite alors en utiliser plusieurs, comme on emploie plusieurs machines dans une grappe. On parle alors de méta-calcul (*meta-computing* [244]) pour l'utilisation d'une telle architecture que nous appellerons *méta-ordinateur*. La programmation de telles architectures est encore plus difficile que celle des machines parallèles classiques, puisque l'on a souvent à faire à un ensemble hétérogène d'architectures parallèles, aussi bien au niveau des processeurs que des réseaux.

Un méta-ordinateur unifie donc l'utilisation de plusieurs machines parallèles, grappes de PC ou de machines massivement parallèles, *via* un réseau, pour résoudre un problème nécessitant une grande quantité de calculs. La présence de différentes architectures rend difficile leur modélisation.

Une solution peut consister à programmer une grappe de machines parallèles comme si c'était une seule machine parallèle. Toutefois, ceci est peu convaincant, notamment en suivant le modèle BSP puisque d'une part, en cas d'hétérogénéité, les algorithmes BSP usuellement équilibrés iront, pour le calcul local, à la vitesse des processeurs les plus lents, et d'autre part, en général, le réseau reliant les différentes machines parallèles est beaucoup moins efficace que les réseaux internes, rendant la réalisation d'une barrières de synchronisation coûteuse (et parfois inutilisable en pratique).

Une autre solution est de ne pas ignorer la structure hiérarchique des méta-ordinateurs et d'utiliser un langage de programmation à deux niveaux l'un pour programmer chaque noeud parallèle et l'autre pour assurer la coordination des noeuds parallèles. BSML aurait pu être utilisé à chaque niveau, toutefois dans ce contexte il est préférable que le niveau de coordination n'impose pas de barrières de synchronisation.

C'est pourquoi, plutôt que de passer directement à la conception d'un modèle et d'un langage à deux niveaux pour l'utilisation de grappes de machines parallèles, nous avons considéré la conception d'un langage proche de BSML mais sans les barrières de synchronisation.

1.3 Plan du mémoire

Ce tapuscrit est organisé en 3 parties correspondant chacune aux contributions de l'auteur dans chacune des phases du projet CARAML :

1. Une étude sémantique d'un langage fonctionnel pour la programmation BSP et la certification des programmes écrits dans ce langage ;

2. Une présentation d'une primitive de composition parallèle (et qui permet aussi la programmation d'algorithmes «diviser-pour-régner» parallèles), un exemple d'application et une extension pour les entrées/sorties parallèles en BSML ;

[5] Les articles [238, 268] présentent d'autres sujets, moins connus, d'applications relatives à cette catégorie d'algorithmes.
[6] page web à `lhc-new-homepage.web.cern.ch`

3. L'adaption du langage pour le méta-calcul.

Ces trois parties sont précédées d'un second chapitre introductif qui présente le langage fonctionnel parallèle initial et son modèle de parallélisme intrinsèque. Elles sont suivies d'un chapitre présentant des travaux connexes et d'un chapitre de conclusions et de perspectives.

1.3.1 Sémantiques et certification

Le chapitre 3 expose des sémantiques formelles du langage BSML. Celles-ci sont avec substitutions explicites afin de mieux faire apparaître les étapes du calcul et les coûts. Le chapitre 4 définit une machine abstraite pour notre langage. Cette machine est alors prouvée correcte vis-à-vis de la sémantique. Le chapitre 5 définit la conception d'une bibliothèque standard de fonctions utilisant les primitives de notre langage et le chapitre 6 est dédié à la preuve de correction de ces fonctions dans un assistant de preuves.

1.3.2 Extensions et bibliothèque de structures de données

Le chapitre 7 aborde le problème de la multi-programmation de machines parallèles (c'est-à-dire la possibilité d'avoir plusieurs programmes parallèles en cours d'exécution sur une même machine parallèle) sous l'angle de nouvelles opérations de composition parallèle pour BSML. Une nouvelle sémantique de BSML, avec une nouvelle primitive, est présentée, ainsi qu'une nouvelle implantation réalisée en suivant les spécifications fournies par la sémantique. Le chapitre 8 est dédié à l'implantation de structures de données parallèles. Ces implantations utilisent massivement cette nouvelle primitive de composition parallèle afin de calculer et de rebalancer les données entre les processeurs. Des expériences sont effectuées sur une application scientifique pour mesurer expérimentalement les performances des implantations. Le chapitres 9 présente une extension non fonctionnelle pure de BSML. Celle-ci ajoute des entrées/sorties parallèles afin de pouvoir programmer des algorithmes manipulant une grande quantité de données (quantité supérieure au total des mémoires internes de la machine parallèle). Ceci implique une modification du modèle BSP. Une sémantique formelle est étudiée.

1.3.3 Opérations globalisées

Le chapitre 10 décrit l'étude de modifications à apporter à BSML afin d'y supprimer les barrières de synchronisation tout en gardant une structuration (modèle) des programmes parallèles. Cela donne lieu à la conception d'un nouveau langage appelé MSPML. Une nouvelle sémantique y est donnée ainsi qu'une implantation. Le chapitre 11 traite de la globalisation de notre approche pour les grilles de calcul, vues ici dans le cadre plus restreint du méta-calcul dit départemental, en ce sens que nous considérons que les machines parallèles font partie d'une même organisation et que le réseau les reliant, même si ses performances sont faibles, est fiable (le trafic y est à peu près constant). MSPML est alors mixé avec BSML afin d'obtenir un langage, appelé DMML, dédié au méta-calcul. Ce chapitre présente ce dernier langage ainsi qu'un nouveau modèle de coût, une sémantique et une implantation.

2 Programmation fonctionnelle BSP

Sommaire

BSP (*Bulk-Synchronous Parallelism*) est un modèle de programmation introduit dans [263] et qui offre à la fois un haut degré d'abstraction tout en étant portable et permettant la prévision réaliste des performances sur une grande variété d'architectures. Un algorithme BSP est en *mode direct* quand la structure des processus physiques est explicite, c'est-à-dire lorsque la fonction qui associe les données et les calculs aux processeurs est explicite. Il est alors plus difficile d'exprimer cet algorithme mais l'efficacité en est meilleure dans de nombreux cas [117].

Dans les langages fonctionnels comme NESL [38, 63] le parallélisme imbriqué est autorisé et la structure des processus physiques est implicite, au prix de l'efficacité ou de la prévisibilité réaliste des performances.

Dans un souci de généralité et en tirant la conclusion, inspirée par BSλ (un λ-calcul étendu pour la programmation BSP), que le parallélisme imbriqué se combine mal avec les processus explicites, les auteurs de BSλ [196] ont choisi d'exclure le parallélisme imbriqué et de favoriser le mode direct dans la conception d'un langage fonctionnel basé sur BSλ : BSML. Ce chapitre donne un aperçu d'un tel langage de programmation et des exemples de programmes fonctionnels BSP en mode direct.

2.1 Bulk-Synchronous Parallelism

Le modèle de programmation parallèle BSP (*Bulk Synchronous Parallelism*) [34, 201, 202, 203, 204, 263] décrit une architecture parallèle (abstraite), un modèle d'exécution et un modèle de coût. L'objectif d'un tel modèle est de fournir un niveau d'abstraction facilitant, d'une part la portabilité des programmes sur une grande variété d'architectures parallèles, et d'autre part de permettre la prévision des performances d'un même programme sur différentes machines parallèles réelles. Ainsi, le modèle BSP fournit une machine abstraite dont les opérations sont de plus haut niveau que celles de la majorité des modèles de programmation parallèle [35, 162, 242] mais autorise aussi de prévoir les temps d'exécution de manière réaliste.

2.1.1 Architecture parallèle BSP

Un ordinateur parallèle BSP possède trois ensembles de composants :

1. Un ensemble homogène de paires processeur-mémoire (généralement des processeurs séquentiels avec des blocs locaux de mémoire),

2. Un réseau de communication permettant l'échange de messages entre ses paires,

3. Une unité de synchronisation globale qui exécute des demandes collectives de barrières de synchronisation.

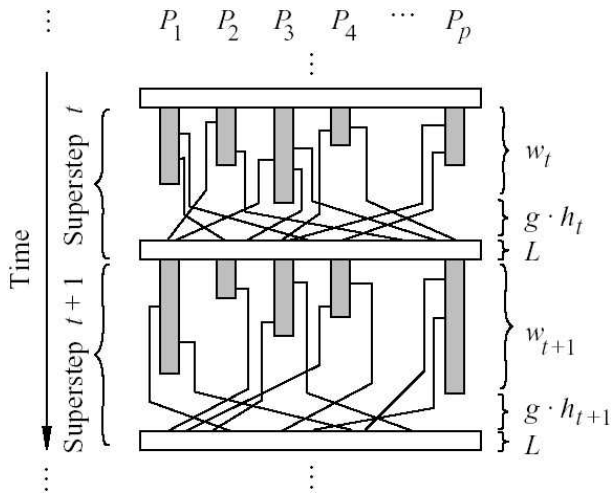


Figure 2.1 — Une super-étape BSP

L'objectif des barrières de synchronisation est de garantir la cohérence des données communiquées entre les processeurs et cela sans que l'ordre des séquences de communication n'affecte le traitement des différents processeurs.

De nombreuses architectures réelles peuvent être vues comme des ordinateurs parallèles BSP. Les machines à mémoire partagée peuvent, par exemple, être utilisées de telle sorte que chaque processeur n'accède qu'à une partie (qui sera alors «privée») de la mémoire partagée et les communications peuvent être faites en utilisant des zones de la mémoire partagée réservées à cet usage. De plus, l'unité de synchronisation est rarement physique mais plutôt logicielle (l'article [148] présente plusieurs algorithmes à cet effet).

Les performances d'un ordinateur BSP sont caractérisées par trois paramètres exprimés en multiples de la vitesse des processeurs (sinon, la vitesse des processeurs s, est donnée en tant que quatrième paramètre) :

1. Le nombre de paires processeur-mémoire p ;

2. Le temps l nécessaire à la réalisation d'une barrière de synchronisation ;

3. Le temps g pour un échange collectif de messages entre les différentes paires processeur-mémoire, appelé 1-relation et dans laquelle chaque processeur envoie et/ou reçoit au plus un mot ; le réseau peut réaliser un échange, appelé h-relation (chaque processeur envoie et/ou reçoit au plus h mots) en un temps $h \times g$. Ainsi, une h-relation est une communication globale où chaque processeur peut envoyer ou recevoir au plus h mots.

En pratique, ces paramètres peuvent être facilement obtenus en utilisant des tests [147]. En général, les paramètres l et g dépendent du nombre de processeurs p et de la topologie du réseau. Par exemple, dans une machine parallèle dite «en hypercube», la valeur de l est de l'ordre de $O(\log p)$ alors qu'elle est de l'ordre de $O(p)$ dans un réseau linéaire.

2.1.2 Modèle d'exécution

L'exécution d'un programme BSP est une séquence de *super-étapes*. Chaque super-étape est divisée en trois phases successives et logiquement disjointes (voir à la figure 2.1) :

1. Chaque processeur utilise les données qu'il détient localement pour faire des calculs de façon séquentielle et pour demander des transferts depuis ou vers d'autres processeurs ;

2. Le réseau réalise les échanges de données demandés à la phase précédente ;

3. Une barrière de synchronisation globale termine la super-étape. À l'issue de cette barrière de synchronisation globale, les données échangées sont effectivement disponibles pour la nouvelle super-étape qui commence alors.

2.1.3 Modèle de coût

Le temps nécessaire à l'exécution d'une super-étape s est la somme :

- Du maximum des temps de calculs locaux ;
- Du temps de la réalisation des échanges entre les processeurs ;
- Du temps de la réalisation d'une barrière de synchronisation globale.

On l'exprime par la formule suivante :

$$\text{Time}(s) = \max_{0 \leq i < p} w_i^{(s)} + \max_{0 \leq i < p} h_i^{(s)} \times g + l$$

où $w_i^{(s)}$ est le temps de calcul local sur le processeur i pendant la super-étape s et $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ où $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) est le nombre de mots envoyés (resp. reçus) par le processeur i durant la super-étape s.

Le temps d'exécution $\sum_s \text{Time}(s)$ d'un programme BSP composé de S super-étapes est donc la somme de trois termes :

$$W + H \times g + S \times l \qquad \text{où} \begin{cases} W &= \sum_s \max_{0 \leq i < p} (w_i^s) \\ H &= \sum_s \max_{0 \leq i < p} (h_i^s). \end{cases}$$

En général W, H et S sont fonctions de p et de la taille des données n, ou de paramètres plus complexes tels que le déséquilibre des données (*data skew*). Afin de minimiser le temps d'exécution, un algorithme BSP doit minimiser à la fois le nombre de super-étapes, le volume total H et le déséquilibre des communications, le volume total W et le déséquilibre des calculs locaux.

Comme il est remarqué dans [90] : «A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures between the late eighties and the time from the mid-nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain».

Le modèle BSP a été utilisé avec succès pour une large variété de problèmes : le calcul scientifique [34, 35, 150], les algorithmes génétiques [51] et la programmation génétique [102], les réseaux de neurones [233], les bases de données parallèles [18, 19], les solveurs de contraintes [124], *etc.*

2.2 Processus explicites + BSP = mode direct ≠ SPMD

Chez les chercheurs intéressés par la programmation parallèle déclarative, il y a un intérêt croissant pour les modèles de coûts d'exécution prenant en compte des paramètres globaux du matériel comme le nombre de processeurs ou la bande passante. De ce point de vue, le principal avantage d'un langage à *processus explicites* est qu'il existe une correspondance explicite entre les processeurs et les données qui n'a pas à être retrouvée en inversant la sémantique des directives de placement (comme c'est le cas par exemple dans le langage HPF [168]).

Dans le langage BSML, une valeur parallèle est construite à partir d'une fonction OCaml des numéros de processeurs vers des données locales. Une restriction cruciale sur les constructions du langage est que les valeurs parallèles ne peuvent être imbriquées. Une telle imbrication impliquerait soit des créations dynamiques de processus, soit des coûts dynamiques non-constants pour le placement des valeurs parallèles sur le réseau de processeurs, les deux contredisant l'objectif de programmation BSP en mode direct (nous nous référons au chapitre 3 pour de plus amples informations).

Le style de programmation SPMD (*Single Program Multiple Data*), par la combinaison d'un langage séquentiel et d'une bibliothèque de communication par passage de messages comme MPI [245] (*Message Passing Interface*), a des avantages grâce à ses processus et messages explicites. Le programmeur peut écrire dans ce style des algorithmes BSP et maîtriser les paramètres qui définissent le temps d'exécution dans le modèle de coûts. Toutefois, les programmes ainsi écrits sont loin d'être aisés à comprendre (et à «débogger» en cas de problèmes) car ils sont possiblement non-déterministes (et peuvent même contenir des inter-blocages). De plus la variable `pid` (identifiant de processeur, c'est-à-dire le numéro de processeur) est liée à *l'extérieur du programme source*.

Considérons par exemple p processeurs et un programme SPMD P utilisant la variable `pid`. La «sémantique» de P est alors :

$$[\![P]\!]_{SPMD} = [\![E@0]\!] \,||\, \ldots \,||\, [\![E@(p-1)]\!]$$

où $P@i = P[\texttt{pid} \leftarrow i]$ et $||$ fait référence à la sémantique concurrente définie par la bibliothèque de communications. Ce schéma a deux inconvénients majeurs :

1. Il utilise une sémantique concurrente pour exprimer des algorithmes parallèles qui servent par définition à concevoir des fonctions [156] et non des relations générales,

2. La variable `pid` est utilisée sans un «lieur» explicite. Il n'y a donc pas de construction syntaxique permettant de sortir du contexte d'un processeur particulier dans le but d'orienter le contrôle global du programme.

Les parties globales d'un programme SPMD sont celles qui ne dépendent d'aucune conditionnelle utilisant la variable `pid`. On donne à cette propriété dynamique le rôle de définir l'aspect le plus élémentaire d'un algorithme parallèle, c'est-à-dire la distinction entre ses parties locales et ses parties globales. Ce manque de contrôle est à l'origine des problèmes de non-déterminisme et d'inter-blocage, comme dans l'exemple ci-dessous :

if pid=0 **then** reception_message_du_proc_1 **else** calculs_asynchrones;échange_total

Doit-on lire que le processeur 0 se bloque à la réception d'un message envoyé explicitement par le processeur 1 ? Dans ce cas, nous sommes en présence d'un blocage du processeur 0 qui ne recevra jamais de message du processeur 1. Ou le message provenant de l'échange total effectué par le processeur 1 suffira-t-il à débloquer le processeur 0 ? En ayant mixé le calcul local et le contrôle global, la sémantique des programmes SPMD est dépendante de celle de la bibliothèque de communications entraînant des résultats non-déterministes car dépendant de l'ordre d'envoi et réception des messages[1].

Le langage BSML propose d'éliminer ces deux problèmes en utilisant un ensemble minimal d'opérations algorithmiques ayant une interprétation BSP. La structure de contrôle parallèle est analogue au PAR de Occam [163] mais sans possibilité d'imbrication. La variable `pid` est remplacée par un argument d'une fonction dans un constructeur parallèle. La propriété d'être une expression locale est donc visible dans la syntaxe ; la variable correspondant à `pid` n'est liée qu'à l'intérieur des blocs parallèles et les parties globales du programme sont précisément celles qui sont hors des blocs parallèles.

2.3 Bulk-Synchronous Parallel ML

BSML n'est donc pas un langage de programmation SPMD. Les programmes BSML sont des programmes OCaml (et donc séquentiels) manipulant une structure de données parallèle. Les principaux avantages sont une sémantique (confère chapitre 3) simplifiée par rapport à celle des langages de programmation SPMD et une compréhension plus aisée des codes sources : les résultats des exécutions parallèles du code sont identiques à ceux d'une exécution séquentielle.

Il n'y a pas, pour l'instant, d'implantation complète du langage Bulk-Synchronous Parallel ML, mais une implantation partielle en tant que bibliothèque pour Objective Caml (OCaml), appelée BSMLlib[2].

Cette bibliothèque est basée sur les primitives données à la figure 2.2. En premier lieu, cette bibliothèque donne accès aux paramètres BSP de l'architecture sur laquelle sont évalués les programmes BSML. La valeur de **bsp_p**() est p, le nombre statique de processeurs de la machine parallèle. Cette valeur est constante durant l'exécution (ce n'est plus vrai en présence d'une opération de composition parallèle [188] qui sera simulée dans le chapitre 7). **bsp_g**() est g, le temps pour effectuer collectivement une 1-relation. **bsp_l**() est l, le temps nécessaire pour une barrière de synchronisation.

Les valeurs parallèles de largeur p contenant en chaque processeur une valeur de type α , appelées vecteurs parallèles, sont représentées par le type abstrait α **par**. L'imbrication de vecteurs parallèles est interdite. Jusqu'à présent, le programmeur était responsable de l'absence d'imbrication. Le système de types présenté dans [R3] remédie à ce défaut. Ceci constitue une amélioration par rapport à Caml-Flight [108, 134] dans lequel l'imbrication de la structure parallèle globale de contrôle `sync` était interdite *dynamiquement*.

[1] Notons que l'emploi des fonctions collectives de MPI à de nombreux avantages par rapport à l'utilisation des fonctions de communication point-à-point dont celui de donner une vue plus globale de l'algorithme parallèle et est donc préférable [123]. Il existe d'ailleurs des approches visant à remplacer automatiquement des appels de fonctions de communication point-à-point par des appels à des fonctions collectives [199]. De plus, seules les fonctions collectives peuvent être implantées de façon plus sûre permettant la vérification des arguments à l'exécution [259].

[2] page web à `http://bsmllib.free.fr/`

bsp_p: unit→int **bsp_l**: unit→float **bsp_g**: unit→float
mkpar: (int→α)→α **par**
apply: (α →β)**par**→α **par**→β **par**
put: (int→α option)**par**→(int→α option)**par**
proj: α option **par**→int→α option
avec **type** α option = None | Some **of** α

Figure 2.2 — Les primitives de la BSMLlib

Les vecteurs parallèles sont créés par la primitive **mkpar** tel que (**mkpar** f) s'évalue en un vecteur parallèle qui possède au processeur i la valeur de (f i), pour tout i compris entre 0 et $(p-1)$:

$$\textbf{mkpar f} = \boxed{(\text{f } 0) \mid (\text{f } 1) \mid \cdots \mid (\text{f } i) \mid \cdots \mid (\text{f } (p-1))}$$

Nous écrivons souvent **fun** pid→e pour f afin de montrer que l'expression e peut être différente en chaque processeur. Cette expression est dite *locale*. L'expression (**mkpar** f) est un objet parallèle et est dite *globale*. Une expression OCaml usuelle qui n'est pas dans un vecteur parallèle est dite *répliquée*, c'est-à-dire identique (ou plutôt dupliquée) sur chaque processeur.

Un algorithme BSP est exprimé comme une combinaison de calculs locaux asynchrones (première phase d'une super-étape), de communications globales (seconde phase d'une super-étape) et d'une synchronisation (troisième phase d'une super-étape). Les calculs asynchrones sont programmés avec les primitives **mkpar** et **apply** tels que (**apply** (**mkpar** f) (**mkpar** e) calcule ((f i) (e i)) au processeur i :

$$\textbf{apply} \boxed{f_0 \mid f_1 \mid \cdots \mid f_i \mid \cdots \mid f_{p-1}} \boxed{v_0 \mid v_1 \mid \cdots \mid v_i \mid \cdots \mid v_{p-1}}$$
$$= \boxed{(f_0\ v_0) \mid (f_1\ v_1) \mid \cdots \mid (f_i\ v_i) \mid \cdots \mid (f_{p-1}\ v_{p-1})}$$

Ni l'implantation de BSMLlib, ni sa sémantique [187] (voir aussi au chapitre 3) ne préconisent de synchronisation entre deux appels successifs à l'une ou l'autre de ces deux primitives. Prenons par exemple l'expression suivante :

let vf=**mkpar**(**fun** pid x→x+pid) **and** vv=**mkpar**(**fun** pid→2∗pid+1) **in apply** vf vv

Les deux vecteurs parallèles sont alors respectivement équivalents à :

$$\boxed{\textbf{fun x}\rightarrow \text{x}+0 \mid \textbf{fun x}\rightarrow \text{x}+1 \mid \cdots \mid \textbf{fun x}\rightarrow \text{x}+i \mid \cdots \mid \textbf{fun x}\rightarrow \text{x}+(p-1)}$$

et

$$\boxed{1 \mid 3 \mid \cdots \mid 2\times i+1 \mid \cdots \mid 2\times(p-1)+1}$$

L'expression **apply** vf vv est alors évaluée en :

$$\boxed{1+0 \mid 3+1 \mid \cdots \mid 2\times i+1+i \mid \cdots \mid 2\times(p-1)+1+(p-1)}$$

c'est-à-dire :

$$\boxed{1 \mid 4 \mid \cdots \mid 3\times i+1 \mid \cdots \mid 3\times(p-1)+1}$$

Un programmeur habitué à la BSPlib [147] (pour la programmation BSP en C) remarquera que nous ignorons la distinction entre la phase de demande de communication et sa réalisation à la barrière de synchronisation. Les phases de communication et de synchronisation sont exprimées à l'aide des primitives **put** et **proj**.

Considérons l'expression suivante : **put**(**mkpar**(**fun** i→fs$_i$)). Pour envoyer une valeur v d'un processeur j vers un processeur i, la fonction fs$_j$ du processeur j doit être telle que (fs$_j$ i) s'évalue en Some v. Pour ne pas envoyer de message de j à i, fs$_j$ doit s'évaluer en None.

L'expression s'évalue en un vecteur parallèle contenant en chaque processeur une fonction fd$_i$ des messages transmis. Au processeur i, (fd$_i$ j) s'évalue en None si le processeur j n'a pas envoyé de message à i ou s'évalue en Some v si le processeur j a envoyé la valeur v au processeur i.

La bibliothèque contient également une primitive de projection globale synchrone appelée **proj**. L'expression (**proj** vec) calcule une fonction f telle que (f n) retourne la $n$ième valeur du processeur vec. Si cette valeur est la valeur vide None alors le processeur n n'a rien transmis aux autres processeurs. Autrement,

celle-ci est Some v et v est diffusée aux autres processeurs. C'est donc une primitive de multi-diffusion permettant de «sortir» des valeurs d'un vecteur parallèle, c'est-à-dire de les passer du contexte local à celui global. Ainsi, sans cette primitive, le contrôle global ne pourrait pas tenir compte des données calculées localement. Cette projection est nécessaire pour exprimer des algorithmes ayant la forme suivante :

Repeat Iteration Parallèle **Until** Max des erreurs locales $< \epsilon$.

Notons que les premières versions de BSMLlib, étant basées directement sur le BSλ-calcul, n'avaient comme projection qu'une conditionnelle globale : ifat: (bool **par**)$*$int$*\alpha*\alpha \rightarrow \alpha$ telle que ifat (v,i,v1,v2) s'évaluait en v1 ou v2 selon que la valeur de v au processeur i était **true** ou **false**. OCaml étant un langage strict avec appel par valeur, cette conditionnelle ne pouvait être définie comme une fonction. C'est la raison pour laquelle les anciennes BSMLlib contenaient une primitive **at**:bool **par**→int→bool qui ne devait être utilisée que dans la construction suivante : **if** (**at** vec pid) **then**... **else**..., où (vec:bool **par**) et (pid:int) et dont la sémantique était celle de ifat.

Avec la primitive de projection, nous pouvons facilement simuler cette primitive de la manière suivante :

```
(* at: int→α par→α *)
let at n v =
 if (0<=n) && (n<(bsp_p())) then
   noSome ((proj (apply (mkpar (fun pid v→ if pid=n then Some v else None)) v)) n)
 else raise At_Failure
```

La primitive **proj** permet aussi d'exprimer le filtrage des vecteurs parallèles de [C6] qui nécessitait l'emploi (dans l'implantation) d'une projection non-sûre et d'un échange total des données. Maintenant, cet échange total peut être directement écrit avec **proj** et ensuite, le filtrage classique de OCaml peut être utilisé : nul besoin d'une primitive non-sûre pour le filtrage des vecteurs.

Les primitives décrites dans la section précédente constituent le noyau de la bibliothèque BSMLlib. Le chapitre 3 décrit des sémantiques de ce langage. Ces primitives sont suffisantes pour exprimer tous les algorithmes BSP. Malgré leur caractère universel, il est souhaitable d'avoir un ensemble de fonctions dites utilitaires pour simplifier la programmation de ces algorithmes et rendre le code plus lisible. Le chapitre 5 décrit un tel ensemble et le chapitre 6 leurs certification dans un assistant de preuves.

Première partie

Sémantiques et certification

3 Sémantiques opérationnelles de BSML

Sommaire

DANS ce chapitre, nous allons présenter des sémantiques dynamiques d'un noyau de notre langage, afin d'en faire apparaître des propriétés. Les définitions présentées ici seront largement utilisées par la suite dans les différentes sémantiques des extensions de BSML ou de nos nouveaux langages. Nous supposons que le lecteur est familier avec les notions de sémantique des langages fonctionnels. Si ce n'est pas le cas, il en trouvera une présentation dans [274].

3.1 Introduction au λ-calcul

Le λ-calcul a été développé en 1930 par A. Church pour fonder un formalisme de représentation des fonctions. Il espérait également pouvoir lui adjoindre un formalisme pour un fondement logique des mathématiques. Ce calcul s'est montré suffisant pour y exprimer toutes les fonctions calculables (les fonctions récursives) mais Kleene et Rosser ont démontré en 1935 l'inconsistance logique du λ-calcul.

3.1.1 Définition du λ-calcul

Initialement, nous nous donnons un ensemble infini dénombrable X dont les éléments sont appelés les variables. Les variables sont des termes du λ-calcul. Si t_1 et t_2 sont deux termes du λ-calcul alors $(t_1\ t_2)$ est un terme du λ-calcul qui représente l'application de t_1 à t_2. Enfin si $x \in X$ et t est un terme du λ-calcul alors $\lambda x.t$ est un terme du λ-calcul représentant la fonction qui à x associe t.

Définition 1 (λ-*calcul*).
La syntaxe des termes du λ-calcul est le plus petit ensemble défini par la grammaire suivante :

$$t ::= x \mid \lambda x.t \mid (t\ t) \qquad \textit{avec } x \in X$$

Exprimer formellement la notion de substitution d'un paramètre d'une fonction par son argument effectif n'est pas une chose aisée. La difficulté provient de la notion de variables : libres, liées ou muettes. Intuitivement, nous voulons que les expressions $\lambda x.(x\ x)$ et $\lambda y.(y\ y)$ soient considérées comme égales. Elles le sont à renommage près des variables : ces expressions sont α-convertibles. Cette égalité pose alors le problème que les termes $\lambda y.(x\ y)$ et $\lambda z.(x\ z)$, où x est une variable libre, sont α-convertibles, tandis que ces mêmes termes, où nous avons substitué x par la variable z, ne le sont plus : nous obtenons $\lambda y.(z\ y)$ et $\lambda z.(z\ z)$ où les variables y et z sont toujours liées. Dans le deuxième cas, la variable muette z a été capturée par la fonction. Réciproquement, si nous substituons x par t dans $\lambda x.x$ et $\lambda y.x$, nous obtenons des termes non équivalents : $\lambda x.x$ et $\lambda y.t$.

La substitution doit donc prendre en compte le problème de la capture des variables libres (non liées) et donc ne substituer que les variables qui ne sont pas liées par un λ dans le terme où nous effectuons la substitution. Nous définissons formellement l'ensemble des variables libre d'un terme du λ-calcul par :

Définition 2 (*Variables libres*).
L'ensemble des variables libres $\mathcal{FV}(t)$ d'un λ-terme t est récursivement défini par

$$\begin{array}{rcl} \mathcal{FV}(x) & = & x \\ \mathcal{FV}(\lambda x.t) & = & \mathcal{FV}(t) \setminus \{x\} \\ \mathcal{FV}((t_1\ t_2)) & = & \mathcal{FV}(t_1) \cup \mathcal{FV}(t_2) \end{array}$$

Nous pouvons alors formellement définir la substitution :

Définition 3 (*Opération de substitution*).
L'opération de substitution d'une variable x par un λ-terme t dans un terme t', notée $t'[x \leftarrow t]$ est récursivement définie par :

$$\begin{array}{rcl} y[x \leftarrow t] & = & \begin{cases} t & \text{si } y = x \\ y & \text{si } y \neq x \end{cases} \\ (\lambda x.t'')[x \leftarrow t] & = & (\lambda x.t'') \\ (\lambda y.t'')[x \leftarrow t] & = & (\lambda z.t''[y \leftarrow z][x \leftarrow t]) \quad \text{si } x \neq y \text{ et } z \notin \mathcal{FV}(t) \cup (\lambda y.t'') \\ (t_1\ t_2)[x \leftarrow t] & = & (t_1[x \leftarrow t]\ t_2[x \leftarrow t]) \end{array}$$

Pour une variable, la substitution consiste simplement à remplacer littéralement la variable par le terme à substituer. Pour une fonction, deux cas se présentent : si la variable est liée par l'abstraction, la substitution n'est pas effectuée pour éviter les problèmes de capture des variables liées, sinon nous commençons par remplacer dans t'' la variable liée par une nouvelle variable (dite fraîche) z qui n'apparaît ni dans t ni dans t'', puis nous remplaçons x par t. Nous évitons ainsi les problèmes de capture d'une variable libre. Pour une application, la substitution s'effectue récursivement dans les sous-termes.

La notion de réduction dans le λ-calcul est réduite à sa plus simple expression : calculer la valeur d'une fonction appliquée à un argument revient à substituer la variable qu'elle lie par son argument dans son corps. Cette réduction, notée $\rightarrow$, est couramment appelée β-réduction. Nous notons $\xrightarrow{*}$ la fermeture réflexive et transitive de $\rightarrow$. Nous pouvons appliquer cette règle dans tous les sous-termes du terme que nous cherchons à réduire.

Définition 4 (*Réduction forte*).
La réduction forte du λ-calcul est définie par :

$$\frac{}{(\lambda x.t_1\ t_2) \rightarrow t_1[x \leftarrow t_2]}$$

$$\frac{t \rightarrow t'}{\Gamma(t) \rightarrow \Gamma(t')} \qquad \text{avec } \Gamma ::= [] \mid \lambda.x.\Gamma \mid (t\ \Gamma) \mid (\Gamma\ t) \quad \forall x \in X$$

où $[]$ est un trou qui peut être «rempli» par n'importe quel terme. Ceci nous permet de différencier la règle de tête et la règle de contexte Γ. Avec ces contextes, nous pouvons réduire en profondeur dans les termes et ainsi réduire leurs sous-termes.

Il suffit de modifier la grammaire des contextes Γ pour définir une nouvelle forme de réduction. Par exemple, nous pouvons définir la réduction faible du λ-calcul avec les contextes suivants :

$$\Gamma ::= [] \mid (t\ \Gamma) \mid (\Gamma\ t)$$

Dans ce cas, la règle de contexte ne permet pas de réduire (effectuer un calcul) sous une abstraction : elle est considérée comme complètement évaluée. C'est donc une valeur.

3.1.2 Propriétés

Les définitions et les propriétés que nous rappelons ici ne sont pas spécifiques au λ-calcul. Nous les donnons dans un cadre général et indiquerons si le λ-calcul les vérifie. Dans cette section, un calcul sera la donnée d'un ensemble de termes $\mathcal{T}$ et d'une notion de réduction $\mathcal{R}$ (une relation binaire sur les termes).

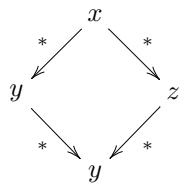
Définition 5 (*Formes normales (valeurs)*).
Un terme est en forme normale pour une relation $\mathcal{R}$ s'il n'a pas de réduit pour $\mathcal{R}$. Nous noterons $\mathcal{NF}_{\mathcal{R}}$ l'ensemble des termes en forme normale pour $\mathcal{R}$ (les valeurs) :

$$\mathcal{NF}_{\mathcal{R}} \underset{def}{\equiv} \{t \in \mathcal{T} \mid \forall t'.\ \neg(t\ \mathcal{R}\ t')\}$$

Définition 6 (*Confluence*).
Nous disons qu'une relation $\mathcal{R}$ est confluente si elle vérifie la propriété suivante :

$$\forall x, y, z \text{ tel que } x\mathcal{R}^* y \text{ et } x\mathcal{R}^* z \Rightarrow \exists u\ y\mathcal{R}^* u \text{ et } z\mathcal{R}^* u$$

schématiquement :



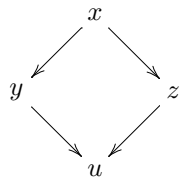
Théorème 1 (*Confluence du λ-calcul*)
Le λ-calcul est confluent pour $\mathcal{R} = \longrightarrow$

Preuve. voir [21] ■

Définition 7 (*Confluence forte*).
Une relation est fortement confluente si elle vérifie la propriété suivante :

$$\forall x, y, z \text{ tel que } x\mathcal{R}y \text{ et } x\mathcal{R}z \Rightarrow \exists u\ y\mathcal{R}u \text{ et } z\mathcal{R}u$$

schématiquement :



Lemme 1
Toute relation fortement confluente est confluente.

Preuve. Une preuve intuitive et graphique est donnée dans [223]. ■

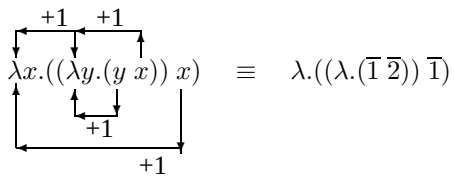


Figure 3.1 — Codage d'un λ-terme avec les indices de De Bruijn

3.1.3 Substitutions explicites

Le λ-calcul ne définit qu'une seule forme de transformation, la β-réduction. Celle-ci remplace l'application d'une fonction sur un argument par la valeur en résultant. Cette transformation est basée sur la substitution de l'argument aux variables le référençant. Elle n'est d'ailleurs pas définie dans le λ-calcul mais plutôt représentée comme une *méta-opération* du langage dont la complexité n'est pas constante : la complexité dépend du terme auquel la substitution est appliquée. En conséquence, sa transposition dans le code objet d'un programme (code machine) ne se résume pas à une simple séquence d'instructions à effectuer, mais à tout un mécanisme (difficile à prouver) que le code objet doit implanter.

Pour cette raison, et bien qu'il décrive le code source des programmes fonctionnels, le λ-calcul n'est pas adapté parce que trop imprécis. Les λ-calculs avec substitutions explicites vont fournir une réponse formelle à ce problème. Dans ces calculs, la substitution n'est plus une méta-opération mais une réduction (voire plusieurs en pratique) au même titre que la β-réduction. Ceci a pour conséquence que la substitution devient une opération *atomique*. Outre une atomicité des opérations, les λ-calculs avec substitutions explicites amènent à se poser des questions sur la nature des composantes du calcul. En particulier, ils nous obligent à préciser ce qu'est une variable (son instanciation), une substitution et une réduction (parallèle ou non).

Les noms de variables posent problème dès qu'il s'agit d'implanter le λ-calcul : à chaque β-réduction, le système doit gérer le renommage des variables. Alors que pour un être humain, le renommage des variables n'est qu'un détail (une α-conversion), une machine passerait une trop grande partie de son temps de calcul à changer ces noms. C'est pourquoi, durant le développement de l'assistant de preuves AUTOMATH, De Bruijn [88] a été amené à décrire un λ-calcul où les variables sont remplacées par des entiers et où le renommage est une incrémentation (ou une décrémentation) de ces entiers.

Le principe des *indices* de De Bruijn est de remplacer une variable par le nombre de λ qui la sépare du λ qui la lie. Il n'est alors plus besoin d'étiqueter le λ par une variable, puisque cette information est déjà contenue dans chaque variable. Par exemple le λ-terme $\lambda x.((\lambda y.(y\ x))\ x)$ devient $\lambda.((\lambda.(\overline{1}\ \overline{2}))\ \overline{1})$ (voir à la figure 3.1). Notons que les entiers (en tant que variables) sont notés avec une barre afin de ne pas être confondus avec des entiers d'un langage de programmation. De Bruijn présente alors un λ-calcul appelé λ_{DB}-calcul, confluent et isomorphe au λ-calcul [200]. Malgré tout, la substitution y reste une opération extérieure (et insécable) au calcul, et cela quelle que soit la complexité des termes manipulés.

Pour remédier à ce défaut, de nombreux auteurs ont proposé des λ-calculs où les substitutions font partie intégrante des termes. Ces calculs fournissent une atomicité des opérations de substitution à base d'environnements qui sont adjoints aux indices de De Bruijn. Les présenter tous serait fastidieux[1], et nous allons simplement définir le $\lambda\sigma_w$-calcul [216] muni d'une stratégie faible d'appel par valeur.

Définition 8 ($\lambda\sigma_w$-*calcul*).
Le $\lambda\sigma_w$-calcul est défini par la grammaire suivante :

$$\begin{array}{lcl} e & ::= & \overline{n} \mid \lambda.e \mid (e\ e) \mid e[s] \\ s & ::= & \bullet \mid v \circ s \\ v & ::= & \lambda.e[s] \end{array}$$

Les valeurs $\lambda.e[s]$ correspondent exactement à la fameuse notion de *fermeture* (*closure* dans la littérature anglo-saxonne) des langages fonctionnels, à savoir un programme e dans un environnement (substitution) s. Cette substitution lie les variables de e avec leurs valeurs. Les règles de réduction du $\lambda\sigma_w$-calcul sont les

[1] Le lecteur intéressé peut consulter [179, 223] pour des taxinomies des λ-calculs avec substitutions explicites.

suivantes :

$$\begin{array}{rcl} \overline{1}[v \circ s] & \rightarrow & v \\ \overline{n+1}[v \circ s] & \rightarrow & \overline{n}[s] \\ ((\lambda.e)[s]\ v) & \rightarrow & e[v \circ s] \\ (e_1\ e_2)[s] & \rightarrow & (e_1[s]\ e_2[s]) \end{array}$$

La stratégie est munie des contextes $\Gamma ::= [] \mid (v\ \Gamma) \mid (\Gamma\ e)$ et de la règle de contexte suivante :

$$\frac{e \rightarrow e'}{\Gamma[e] \rightarrow \Gamma[e']}$$

On a alors le résultat suivant :

Théorème 2 (*Déterminisme*)

Si e est une expression sans variable libre ni substitutions et si $e[\bullet] \xrightarrow{*} v_1$ et $e[\bullet] \xrightarrow{*} v_2$ alors $v_1 = v_2$.

Preuve . Chacune des règles est déterministe. Le contexte ne permet la réduction que d'une unique sous-expression. La relation $\rightarrow$ est donc fortement confluente et par le lemme 1, confluente. ■

Ce théorème peut se lire comme, «étant donné un terme e sans variables libres ni substitutions alors son évaluation dans un environnement (substitution) vide est déterministe, c'est-à-dire qu'elle conduit toujours au même résultat».

3.1.4 Langages fonctionnels et sémantiques opérationnelles

Le λ-calcul présente l'avantage d'avoir une sémantique propre ce qui représente un atout considérable pour le développement (et la certification) de programmes. En plus de sa simplicité, il offre une grande expressivité calculatoire (il est Turing-complet). Malgré tout, il est inefficace algorithmiquement parlant et inutilisable pour le commun des programmeurs : écrire une fonction «utile» en λ-calcul est un travail titanesque.

Les langages fonctionnels sont des extensions du λ-calcul permettant une programmation simple et efficace des fonctions. Ils sont donc des langages réalistes pour la programmation[2]. Ils conservent une sémantique propre et l'introduction de structures de données permet la facilité d'écriture des programmes. La clarté de la sémantique des langages fonctionnels et leurs ressemblances avec les mathématiques ont fait qu'ils se sont imposés comme langages de base pour l'enseignement, la recherche et, à moindre mesure, l'industrie (bien que leur utilisation dans ce domaine s'accroît). Les intérêts de définir la sémantique d'un langage de programmation sont :

1. De spécifier le sens des programmes écrits (ce qu'ils calculent) ;
2. De définir formellement le fonctionnement (le comment) de leurs calculs ;
3. De préciser quels sont les programmes qui sont syntaxiquement corrects mais sémantiquement absurdes : $1 + (\lambda x.x)$;
4. De permettre des études formelles sur les programmes ; par exemple, une étude des coûts des programmes : quel programme sera alors le plus efficace ?
5. De permettre des optimisations (au niveau des algorithmes ou de la modularité du code).

Dans la littérature, nous trouvons trois méthodes pour spécifier la notion de programme absurde. La première est la sémantique à grands pas (aussi appelée sémantique naturelle) où les programmes absurdes sont ceux qui ne s'évaluent pas. Mais cette sémantique ne permet pas de différencier les programmes absurdes des programmes qui bouclent (qui ne se terminent pas).

La deuxième consiste à rajouter un terme *err* au langage et des règles de réduction pour tous les cas absurdes (qui s'évaluent en *err*). Nous avons alors trois *scenarii* possibles :

1. Le programme s'évalue vers *err*, c'est un programme absurde ;

[2] Dans le cadre de cette thèse, il serait trop long de présenter un langage fonctionnel dans son intégralité (par exemple OCaml). Nous nous réferons à [62, 77] pour une description complète d'un langage fonctionnel comme OCaml : syntaxe, sémantique (in)formelle, bibliothèques, exemples etc.

2. Le programme s'évalue vers un terme qui n'est pas *err*, c'est un programme valide ;

3. Le programme s'évalue à l'infini : il boucle.

Toutefois, cette approche n'est pas satisfaisante, car elle oblige à rajouter beaucoup de règles au risque d'en oublier. De plus, ces approches ne s'intéressent pas aux étapes de calcul, mais seulement au résultat final, ce qui ne prête pas bien à la description des machines abstraites (voir au chapitre 4).

La dernière possibilité est d'utiliser une sémantique à «petits pas» (sémantique à réduction), qui décrit toutes les étapes de calcul, et de définir une notion de valeur, c'est-à-dire un programme valide en forme normale. Ceci permet aussi d'étudier l'exécution des programmes qui ne terminent pas (par exemple les serveurs). Nous avons alors trois *scenarii* possibles pour l'exécution d'un terme e :

1. e s'évalue vers une valeur en un nombre fini d'étapes : $e \to e_1 \to \cdots \to e_n \to v$, c'est-à-dire $e \xrightarrow{*} v$;

2. e se réduit à l'infini : $e \to e_1 \to \cdots \to e_n \to \cdots$;

3. e se réduit en une forme normale qui n'est pas une valeur : $e \to e_1 \to \cdots \to e_n \not\to$. C'est un programme absurde.

3.2 Sémantiques de BSML

Comme nous venons de le voir, les sémantiques opérationnelles apportent un formalisme aux calculs. Dans cette section, nous allons décrire trois sémantiques pour un mini-langage BSML, qui auront chacune leurs avantages et inconvénients.

3.2.1 Syntaxe d'un mini-langage parallèle applicatif

Raisonner sur une définition complète et exhaustive d'un langage fonctionnel et parallèle comme BSML serait trop complexe. Pour notre propos et afin de simplifier la présentation, cette section introduit un mini-langage applicatif qui peut être considéré comme un noyau de BSML. Ce mini-langage a la forme d'un calcul mais n'en est pas un, à proprement parler, puisqu'une stratégie d'évaluation sera donnée. Nous ferons donc un abus de langage en l'appelant calcul par opposition à BSML. Par la suite, nos termes seront notés avec éventuellement des indices[3].

Définition 9 (*Langage source*).
Les expressions initiales sont définies par la grammaire suivante :

e^p	::=	$\lambda x.e^p$	*abstraction (fonction)*
	|	$(e^p\ e^p)$	*application*
	|	(e^p, e^p)	*paire*
	|	$\mathbf{mkpar}\ e^p$	*création d'un vecteur parallèle*
	|	$\mathbf{apply}\ e^p\ e^p$	*application parallèle*
	|	$\mathbf{put}\ e^p$	*communication*
	|	$\mathbf{proj}\ e^p$	*projection globale*
	|	$\mathbf{c}$	*constante*
	|	$\mathbf{op}$	*opération prédéfinie*
	|	$\mathbf{x}$	*variable*
	|	$\mu x.e^p$	*définition d'une fonction récursive*
	|	$\mathbf{if}\ e^p\ \mathbf{then}\ e^p\ \mathbf{else}\ e^p$	*conditionnelle*

Ces expressions sont celles données par le «programmeur». Ce sont donc les termes de notre mini-langage de base. L'ensemble des constantes contient par exemple les entiers, les booléens etc. Nous utilisons aussi la constante $\mathbf{nc}$ (pour «no-communication») qui correspondra au None de OCaml. Les opérateurs peuvent être les opérations arithmétiques, logiques etc. Cet ensemble contient aussi deux opérations d'identité un peu particulières : $\mathbf{GloId}$ et $\mathbf{LocId}$. La première ne peut être exécutée que dans le contexte global (en dehors d'un vecteur parallèle) et *a contratrio* la deuxième uniquement dans un contexte local (c'est-à-dire dans un vecteur parallèle). Le chapitre 9 sur les entrées/sorties en BSML donnera des exemples

[3] Nous utiliserons aussi un parenthèsage dit prioritaire à gauche (comme en OCaml) afin d'améliorer la lisibilité de nos règles.

plus concrets[4] de ce genre d'opérations. Notons que $\lambda x.e^p$ se traduit en OCaml par **fun** x →e et $\mu f.e$ par (**let rec** f = e **in** f).

Comme pour le λ-calcul (et pour les mêmes raisons), nous utilisons des substitutions explicites dans nos sémantiques.

Définition 10 (*Expressions des sémantiques*).
Nos expressions ont donc la forme suivante :

e	::=	$(e)[s]$	*expression munie d'une substitution*
	|	$\lambda.e$	*abstraction*
	|	$\overline{(\lambda.e)[s]}$	*fermeture*
	|	$(e\ e)$	*application*
	|	(e, e)	*paire*
	|	$\mathbf{mkpar}\ e$	*création d'un vecteur parallèle*
	|	$\mathbf{apply}\ e\ e$	*application parallèle*
	|	$\mathbf{put}\ e$	*communication*
	|	$\mathbf{proj}\ e$	*projection globale*
	|	$\mathbf{c}$	*constante*
	|	$\mathbf{op}'$	*opérateur*
	|	$\overline{\mathbf{n}}$	*variables de substitution (indices)*
	|	$\mu.e$	*définition d'une fonction récursive*
	|	$\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$	*conditionnelle*
	|	$\langle e, \ldots, e\rangle$	*vecteur parallèle de taille p*
	|	$[e, \ldots, e]$	*tableau fonctionnel*

où $\mathbf{op}' ::= \mathbf{op} \cup \{\mathbf{delpar}, \mathbf{init}, \mathbf{access}, \mathbf{send}\}$.

Notons que nous avons ajouté les vecteurs parallèles de taille p fixe, ainsi que des tableaux (qui seront purement fonctionnels, c'est-à-dire sans effets de bord). Nous aurons donc une sémantique par valeur de p. L'ensemble des opérateurs est étendu avec des opérations internes au calcul : la suppression du constructeur de vecteur parallèle $\langle\rangle$, la création d'un tableau purement fonctionnel, l'accès à ces valeurs et l'échange de valeurs entre les processeurs. Notons que nous différencions une abstraction (avec sa substitution) $(\lambda.e)[s]$ et une fermeture. Bien qu'au niveau sémantique, il n'y ait pas de différence, la création d'une fermeture a néanmoins un coût. En les différenciant, nous allons pouvoir ajouter une règle de création de la fermeture et lui donner un coût[5].

Définition 11 (*Substitutions et valeurs*).
Les substitutions et les valeurs (sous-ensemble des expressions) sont classiquement définies par :

s	::=	$\bullet$	*substitution vide*
	|	$v \circ s$	*valeur suivie de la suite de la substitution*

$$v ::= \mathbf{op} \mid \mathbf{c} \mid \overline{(\lambda.e)[s]} \mid (v, v) \mid \langle v, \ldots, v\rangle$$

La traduction des «expressions du programmeur» en nos expressions, c'est-à-dire le remplacement des variables par des indices de De Brujin, se définit inductivement (figure 3.2) où $\mathcal{E}$ est un environnement de substitution des variables (un dictionnaire entre les variables et les indices) défini par :

$$\mathcal{E} ::= \bullet \mid \{x \mapsto \overline{n}, \mathcal{E}\}$$

avec la fonction suivante de mise à jour de l'environnement :

$$\begin{array}{rcl} \mathcal{R}_x(\bullet) & = \bullet & \\ \mathcal{R}_x(\{x \mapsto \overline{n}, \mathcal{E}\}) & = & \mathcal{R}_x(\mathcal{E}) \\ \mathcal{R}_x(\{y \mapsto \overline{n}, \mathcal{E}\}) & = & \{y \mapsto \overline{n+1}, \mathcal{R}_x(\mathcal{E})\} \quad \text{si } y \neq x \end{array}$$

[4] Un exemple évident d'opération qui ne peut pas être exécutée globalement est une fonction rand:int→int qui retourne un entier «aléatoire». Si cette opération était exécutée globalement, nous n'aurions plus la même valeur dans le contexte global, ce qui entraînerait des problèmes comme le montre le chapitre 9.

[5] Dans de futurs travaux, nous voudrions comparer formellement les coûts prédits par une analyse statique avec ceux d'une sémantique opérationnelle et les temps d'exécutions réelles.

$$
\begin{array}{rcl|rcl}
\mathcal{T}_{\mathcal{E}}(\lambda x.e^p) & = & \lambda.T_{\{x\mapsto\overline{1},R_x(\mathcal{E})\}}(e^p) & \mathcal{T}_{\mathcal{E}}(\mathbf{apply}\ e_1^p\ e_2^p) & = & \mathbf{apply}\mathcal{T}_{\mathcal{E}}(e_1^p)\ \mathcal{T}_{\mathcal{E}}(e_2^p) \\
\mathcal{T}_{\mathcal{E}}(\mu x.e^p) & = & \mu.T_{\{x\mapsto\overline{1},R_x(\mathcal{E})\}}(e^p) & \mathcal{T}_{\mathcal{E}}(\mathbf{put}\ e^p) & = & \mathbf{put}\ \mathcal{T}_{\mathcal{E}}(e^p) \\
\mathcal{T}_{\mathcal{E}}((e_1^p\ e_2^p)) & = & (\mathcal{T}_{\mathcal{E}}(e_1^p)\ \mathcal{T}_{\mathcal{E}}(e_2^p)) & \mathcal{T}_{\mathcal{E}}(\mathbf{proj}\ e^p) & = & \mathbf{proj}\ \mathcal{T}_{\mathcal{E}}(e^p) \\
\mathcal{T}_{\mathcal{E}}((e_1^p, e_2^p)) & = & (\mathcal{T}_{\mathcal{E}}(e_1^p), \mathcal{T}_{\mathcal{E}}(e_2^p)) & \mathcal{T}_{\mathcal{E}}(\mathbf{c}) & = & \mathbf{c} \\
\mathcal{T}_{\mathcal{E}}(\mathbf{mkpar}\ e^p) & = & \mathbf{mkpar}\ \mathcal{T}_{\mathcal{E}}(e^p) & \mathcal{T}_{\mathcal{E}}(\mathbf{op}) & = & \mathbf{op}
\end{array}
$$

$$\mathcal{T}_{\mathcal{E}}(\mathbf{if}\ e_1^p\ \mathbf{then}\ e_2^p\ \mathbf{else}\ e_3^p) = \mathbf{if}\ \mathcal{T}_{\mathcal{E}}(e_1)\ \mathbf{then}\ \mathcal{T}_{\mathcal{E}}(e_2)\ \mathbf{else}\ \mathcal{T}_{\mathcal{E}}(e_3)$$

$$\mathcal{T}_{\mathcal{E}}(x) = \overline{n} \text{ si } \mathcal{E} = \{\cdots, x \mapsto \overline{n}, \cdots, \bullet\}$$

Figure 3.2 — Instanciation des variables en des indices de De Brujin

Cette mise à jour de l'environnement est effectuée lors de la traduction d'une expression de la forme $\lambda x.e$ ou $\mu x.e$. En effet, ce sont les lieurs de notre calcul, et dans ce cas, la variable x sera maintenant l'indice 1 et toutes les autres variables verront leurs indices incrémentés. C'est ce que fait $\mathcal{R}$ quand elle modifie l'environnement de substitution des variables.

Propriété 1

Soit l'expression e^p telle que $e = \mathcal{T}_\bullet(e^p)$, alors e^p est sans variables libres (ainsi que e par conséquent).

Preuve. Par induction triviale sur la traduction de l'expression e^p. ■

Notons que, considérant notre calcul comme le noyau de notre langage BSML, le fait de ne pas avoir de variables libres n'est pas un souci du point de vue de l'expressivité : elles seront de toutes façons systématiquement rejetées par un compilateur lors de l'analyse statique du terme (ou lors de sa compilation).

3.2.2 Sémantique naturelle

Nous pouvons maintenant définir la sémantique naturelle de notre calcul ; elle est exprimée à l'aide d'une relation d'induction.

Nous définissons deux types de relations : $\rhd_g$ pour la réduction globale de l'expression et $\rhd_i$ pour la réduction locale au processeur i. Comme un grand nombre de règles sont communes, nous avons utilisé un ensemble de règles $\rhd$ génériques aux deux réductions. Nous avons aussi les règles $\rhd_{\bowtie}$ des primitives parallèles. La figure 3.3 donne ces ensembles de règles. Les règles sont toutes de la forme $s, e \rhd v$, qui peut se lire comme «dans l'environnement s (défini comme une substitution), l'expression e s'évalue en la valeur v».

Définition 12 (*Sémantique naturelle*).
La sémantique naturelle est définie par : $\rhd_i = \rhd$ et $\rhd_g = \rhd \cup \rhd_{\bowtie}$.

Remarquons que les primitives parallèles introduisent l'utilisation d'opérations internes au calcul comme par exemple **send** qui effectue les échanges entre les processeurs.

Les règles 3.7 définissent le fonctionnement des opérations. Ces dernières sont distinguées entre les locales, les globales et les génériques. Par exemple, pour les génériques nous avons les opérations suivantes :

$$
\begin{array}{rcl}
s, \mathbf{fst}\ (v_1, v_2) & \rhd & v_1 \\
s, \mathbf{snd}\ (v_1, v_2) & \rhd & v_2 \\
s, +\ (n_1, n_2) & \rhd & n_1 + n_2 \\
s, \mathbf{isnc\ nc} & \rhd & \mathbf{true} \\
s, \mathbf{isnc}\ v & \rhd & \mathbf{false} \text{ si } v \neq \mathbf{nc} \\
s, \mathbf{init}\ (n, f) & \rhd & [(f\ 0), \ldots, (f\ n-1)] \\
s, \mathbf{access}\ ([v_0, \ldots, v_i, \ldots, v_{p-1}], i) & \rhd & v_i
\end{array}
$$

init permet de créer un tableau (purement fonctionnel) et **access** permet de lire les valeurs dans ce tableau. Notons que ces tableaux seront purement fonctionnels car nous ne donnons pas un opérateur d'affectation

d'une composante d'un tableau. Les autres règles sont évidentes. Nous avons $s, \mathbf{LocId}\ v \rhd_i v$ en tant qu'opération uniquement locale. Pour les globales, nous avons :

$$\begin{array}{rcl} s, \mathbf{delpar}\ \langle f, \ldots, f\rangle & \rhd_g & f \\ s, \mathbf{send}\ \langle [v_0^0, \ldots, v_{p-1}^0], \ldots, [v_0^{p-1}, \ldots, v_{p-1}^{p-1}]\rangle & \rhd_g & \langle [v_0^0, \ldots, v_0^{p-1}], \ldots, [v_{p-1}^0, \ldots, v_{p-1}^{p-1}]\rangle \\ s, \mathbf{GloId}\ v & \rhd_g & v \text{ si } \mathcal{V}_\bullet(v) = \mathbf{true} \end{array}$$

où la fonction $\mathcal{V}$ est inductivement définie à la figure 3.4. Celle-ci indique **false** dès qu'un vecteur parallèle (ou une primitive parallèle) est rencontré. Cette fonction permet de vérifier qu'une valeur ne contient pas de vecteurs parallèles (ou qu'elle puisse en contenir). Dans ce cas, cela permet de savoir si la valeur est répliquée (valeur OCaml mais dupliquée sur les processeurs) ou non. Notons que l'environnement permet de tester les valeurs des indices (variables) : les fermetures changent donc cet environnement.

send permet d'échanger les valeurs contenues dans les tableaux d'un vecteur parallèle. Comme nous allons le constater dans les prochaines sections, c'est l'opérateur de communication et de synchronisation qui n'est exécuté que par les primitives de communications. L'opérateur **delpar** ne permet de détruire un vecteur que si toutes les composantes sont identiques et cela sans communications. Ceci est possible car cette opération n'est utilisée que dans la primitive **proj**. Le vecteur résultant aura alors toutes ses composantes identiques afin de restituer une fonction répliquée. Notons que ces deux opérations ne peuvent pas être utilisées par le programmeur : elles sont internes aux primitives. **GloId** est un opérateur global (confère chapitre 9 pour des exemples concrets de telles opérations).

Détaillons maintenant les règles de la figure 3.3 :

- Les règles 3.1 et 3.2 permettent d'accéder à la $n$ème valeur de l'environnement. C'est le remplacement de la variable par sa valeur lors d'une substitution ;
- La règle 3.3 rend possible la transformation d'une fonction (dans un environnement) en une fermeture ;
- La règle 3.5 calcule inductivement une paire d'expressions ;
- La règle 3.6 calcule l'application d'une fermeture (provenant d'une fonction) à un argument. Pour cela l'environnement de la fermeture est utilisé et complété avec l'argument afin de calculer le corps de la fonction ;
- Les règles 3.7 permettent de calculer l'application d'un opérateur à un argument ;
- La règle 3.8 permet d'évaluer les composantes d'un tableau. Les tableaux seront employés pour l'échange des données de la primitive de communication **put** ;
- La règle 3.9 est dédiée à la définition récursive des programmes ;
- Les règles 3.10 et 3.11 sont pour la conditionnelle ;
- La règle 3.12 est dédiée à la primitive **mkpar** qui permet la création des vecteurs. Nous testons alors son paramètre (qui sera mis dans le vecteur) afin d'empêcher la présence d'un vecteur parallèle ou d'une primitive parallèle. En effet, nous avons vu que nous voulions éviter l'emboîtement des vecteurs parallèles. Ce test, noté $\mathcal{V}$, s'effectue donc sur le paramètre f de **mkpar** (en pratique une fonction donc une fermeture) dans un environnement vide. Ce test est inductivement défini à la figure 3.4.
- La règle 3.13 permet l'évaluation des composantes d'un vecteur parallèle. Les évaluations deviennent locales ;
- La règle 3.14 est pour l'application parallèle point-à-point en créant un nouveau vecteur à partir des deux autres ;
- La règle 3.15 est dédiée à la primitive **put**, celle des communications. Nous créons tout d'abord un vecteur de tableaux des valeurs à transmettre *via* le réseau (tableaux de tailles p). Ensuite, ces valeurs sont échangées par l'opérateur **send** et nous reconstruisons le vecteur final de fonctions en appliquant la fonction **F** en chaque composante du vecteur de tableaux ; Cette fonction permet de construire une nouvelle fonction à partir d'un tableau (des valeurs reçues) et d'un indice. Cet indice correspond au pid du processeur émetteur de la valeur reçue.
- La règle 3.16 permet l'évaluation de la projection globale synchrone. Pour ce faire, nous utilisons la primitive **put** afin de transmettre les valeurs à projeter. Ces valeurs sont donc transmises à tous les processeurs. Ensuite, comme chaque composante du vecteur contient la même fonction (les valeurs multi-diffusées), nous supprimons le constructeur de vecteur afin d'avoir le résultat final (une fonction répliquée qui indique la valeur de la $i$ème composante du vecteur donné en paramètre).

Notons que les primitives parallèles introduisent des environnements vides. Ils sont vides pour la preuve d'équivalence avec la sémantique à «petits pas» et parce que les vecteurs ne contiennent à leurs créations que des valeurs qui sont soit des constantes, soit des fermetures. Elles n'ont donc pas besoin de l'environnement. Nous avons alors le résultat suivant :

Théorème 3 (*Déterminisme*)

Soit une expression «programmeur» e^p telle que $e = \mathcal{T}_\bullet(e^p)$. Si $\bullet, e \rhd_g v_1$ et $\bullet, e \rhd_g v_2$ alors $v_1 = v_2$.

Preuve. Voir en section 3.A.1 de l'annexe de ce chapitre. ■

Prenons pour exemple, l'expression e^p (programmeur) de diffusion (depuis processeur 0) suivante :

```
(apply (mkpar (fun i→fun f→ (f 0)))
       (put (apply (mkpar (fun i→if i=0 then (fun v→fun j→v) else (fun v→fun j→nc)))
                   (mkpar (fun i→i)))))
```

Nous obtenons pas $\mathcal{T}_\bullet$ l'expression e suivante :

$$(\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1}, 0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1})))))$$

La figure 3.5 donne l'évaluation de cette expression avec la sémantique naturelle (avec deux processeurs) de notre mini-langage.

Cette sémantique, bien que très simple, n'est pas suffisante, et ce pour deux raisons. La première, générale aux sémantiques naturelles, est que l'expression est évaluée «du début à la fin» et qu'il est donc impossible de raisonner sur des programmes qui ne terminent pas. La deuxième est que toutes les opérations (même les parallèles) semblent synchrones. Le parallélisme (l'entrelacement des calculs) n'est pas du tout présent, puisque tout est relations. Il nous faut donc une autre sémantique.

3.2.3 Sémantique à «petits pas»

La sémantique à «petits pas» consiste en une relation entre deux expressions, définie à l'aide d'axiomes et de règles appelées *pas*. Elle décrit tous les pas (étapes) de calculs de l'expression jusqu'à sa valeur.

Depuis le début de ce manuscrit, nous parlons de l'avantage de la programmation BSP (et de la programmation parallèle structurée en général) qu'est la possibilité d'y modéliser les coûts (on parle de prédiction des performances). Grâce à la sémantique à «petits pas», nous allons définir «précisément» les coûts de ces réductions en les ajoutant aux étapes de calculs.

Définition 13 (*Coûts*).
L'algèbre de coûts que nous utilisons est celle des entiers munie de l'addition $\oplus$, de la multiplication $\otimes$ usuelles (donc toutes deux communicatives et associatives) et de deux constantes g et l.

Nous nous contentons de compter le nombre de β-réductions et de δ-réductions des expressions[6]. Nous y ajoutons aussi les communications et synchronisations BSP (paramètres g et l) qui seront donc des entiers (ce qui n'est pas un problème en soit puisque nous donnerons des coûts symboliques et non des coûts réels). Nous notons c_g le coût correspondant aux réductions globales et $\langle c_0, \ldots, c_{p-1}\rangle$ les coûts locaux. Nous notons aussi $\langle \mathbf{c}\rangle$ les coûts locaux quand ceux-ci n'interviennent pas dans la règle.

Comme nous utilisons une stratégie stricte d'appel par valeur, les arguments des fonctions (comme ceux des opérateurs) sont d'abord évalués en valeurs. Un programme est donc toujours réduit de la «même façon». Comme il est écrit dans [215], «Each evaluation order has its advantages and disanvatages, but strict evaluation is clearly superior in at least one area : ease of reasoning about asymptotic complexity».

Notre sémantique à «petits pas» a la forme suivante : $e/c_g/\langle c_0, \ldots, c_{p-1}\rangle \rightharpoonup e/c'_g/\langle c'_0, \ldots, c'_{p-1}\rangle$ où e est une expression, c_g est le coût global et $\langle c_0, \ldots, c_{p-1}\rangle$ sont les coûts locaux.

Nous notons $\stackrel{*}{\rightharpoonup}$ la fermeture transitive et réflexive de $\rightharpoonup$, c'est-à-dire que nous notons :

$$e/c_g/\langle c_0, \ldots, c_{p-1}\rangle \stackrel{*}{\rightharpoonup} e'/c'_g/\langle c'_0, \ldots, c'_{p-1}\rangle$$

[6] Notre modèle de coûts peut paraître «grossier», mais il sera amplement suffisant pour comparer les coûts de cette sémantique avec ceux de la sémantique définie dans le chapitre 7 (où une nouvelle primitive est ajoutée au calcul) et ceux d'un autre calcul défini dans le chapitre 10.

Gestion des substitutions :

$$\frac{s, \overline{n} \rhd v}{v' \circ s, \overline{n+1} \rhd v} \tag{3.1}$$

$$\frac{}{\overline{v \circ s, \overline{1}} \rhd v} \tag{3.2}$$

$$\frac{}{\overline{s, \lambda.e \rhd \overline{(\lambda.e)[s]}}} \tag{3.3}$$

$$\frac{\text{si } v \neq \langle \cdots \rangle \text{ et } v \neq (v_0, v_1)}{s, v \rhd v} \tag{3.4}$$

Noyau fonctionnel :

$$\frac{s, e_1 \rhd v_1 \qquad s, e_2 \rhd v_2}{s, (e_1, e_2) \rhd (v_1, v_2)} \tag{3.5}$$

$$\frac{s, e_1 \rhd \overline{(\lambda.e)[s']} \qquad s, e_2 \rhd v' \qquad v' \circ s', e \rhd v}{s, (e_1\ e_2) \rhd v} \tag{3.6}$$

$$\frac{s, e_1 \rhd_l \mathbf{op} \quad s, e_2 \rhd_l v \quad s, \overline{\mathbf{op}(v)} \rhd_l v'}{s, (e_1\ e_2) \rhd_l v'} \qquad \frac{s, e_1 \rhd_g \mathbf{op} \quad s, e_2 \rhd_g v \quad s, \overline{\mathbf{op}(v)} \rhd_g v'}{s, (e_1\ e_2) \rhd_g v'} \tag{3.7}$$

$$\frac{\forall i \in \{0, \ldots, p-1\} \quad s, e_i \rhd v_i}{s, [e_0, \ldots, e_{p-1}] \rhd [v_0, \ldots, v_{p-1}]} \tag{3.8}$$

$$\frac{(\mu.e) \circ s, e \rhd v}{s, \mu.e \rhd v} \tag{3.9}$$

$$\frac{s, e_1 \rhd \mathbf{true} \qquad s, e_2 \rhd v}{s, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd v} \tag{3.10}$$

$$\frac{s, e_1 \rhd \mathbf{false} \qquad s, e_3 \rhd v}{s, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd v} \tag{3.11}$$

Primitives parallèles :

$$\frac{s, e \rhd_{\bowtie} f \qquad \bullet, \langle (f\ 0), \ldots, (f\ (p-1)) \rangle [\bullet] \rhd_{\bowtie} \langle v_0, \ldots, v_{p-1} \rangle \quad \text{si } \mathcal{V}_\bullet(f) = \mathbf{true}}{s, \mathbf{mkpar}\ e \rhd_{\bowtie} \langle v_0, \ldots, v_{p-1} \rangle} \tag{3.12}$$

$$\frac{\forall i \in \{0, \ldots, p-1\} \quad \bullet, e_i \rhd_i v_i}{\bullet, \langle e_0, \ldots, e_{p-1} \rangle \rhd_{\bowtie} \langle v_0, \ldots, v_{p-1} \rangle} \tag{3.13}$$

$$\frac{s, e_1 \rhd_{\bowtie} \langle f_0, \ldots, f_{p-1} \rangle \quad s, e_2 \rhd_{\bowtie} \langle v_0, \ldots, v_{p-1} \rangle \quad \bullet, \langle (f_0\ v_0), \ldots, (f_{p-1}\ v_{p-1}) \rangle \rhd_{\bowtie} \langle v'_0, \ldots, v'_{p-1} \rangle}{s, \mathbf{apply}\ e_1\ e_2 \rhd_{\bowtie} \langle v'_0, \ldots, v'_{p-1} \rangle} \tag{3.14}$$

$$\frac{s, e \rhd_{\bowtie} \langle f_0, \ldots, f_{p-1} \rangle \quad \bullet, (\mathbf{apply}\ (\mathbf{mkpar}(\lambda.\mathbf{F}))\ (\mathbf{send}\ \langle (\mathbf{init}\ (p, f_0)), \ldots, (\mathbf{init}\ (p, f_{p-1})) \rangle)) \rhd_{\bowtie} \langle f'_0, \ldots, f'_{p-1} \rangle}{s, \mathbf{put}\ e \rhd_{\bowtie} \langle f'_0, \ldots, f'_{p-1} \rangle} \tag{3.15}$$

$$\frac{s, e \rhd_{\bowtie} \langle v_0, \ldots, v_{p-1} \rangle \quad \bullet, (\mathbf{delpar}\ (\mathbf{put}\ \langle (\lambda.v_0), \ldots, (\lambda.v_{p-1}) \rangle)) \rhd f}{s, \mathbf{proj}\ e \rhd_{\bowtie} f} \tag{3.16}$$

avec $\mathbf{F} = \lambda.\lambda.\mathbf{if}\ (0 \leq \overline{1})\mathbf{and}(\overline{1} < \mathbf{p})\ \mathbf{then}\ (\mathbf{access}\ (\overline{2}, \overline{1}))\ \mathbf{else}\ \mathbf{nc}$

Figure 3.3 — Règles de la sémantique naturelle de mini-BSML

$$
\begin{array}{rcl|rcl}
\mathcal{V}_s(\mathbf{mkpar}\ e) & = & \mathbf{false} & \mathcal{V}_s(\mathbf{op}) & = & \mathbf{true} \\
\mathcal{V}_s(\mathbf{proj}\ e) & = & \mathbf{false} & \mathcal{V}_s(\mathbf{c}) & = & \mathbf{true} \\
\mathcal{V}_s(\mathbf{put}\ e) & = & \mathbf{false} & \mathcal{V}_s(\lambda.e) & = & \mathcal{V}_{(x\circ s)}(e) \\
\mathcal{V}_s(\mathbf{apply}\ e\ e) & = & \mathbf{false} & \mathcal{V}_s(\overline{(\lambda.e)[s']}) & = & \mathcal{V}_{(x\circ s')}(e) \\
\mathcal{V}_s(\langle \ldots \rangle) & = & \mathbf{false} & \mathcal{V}_s(\overline{\mu.e}) & = & \mathcal{V}_{(x\circ s)}(e) \\
\mathcal{V}_s(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & = & \mathcal{V}_s(e_2) \wedge \mathcal{V}_s(e_2) \wedge \mathcal{V}_s(e_3) & \mathcal{V}_{(v\circ s)}(\overline{n+1}) & = & \mathcal{V}_s(\overline{n}) \\
\mathcal{V}_s(e_1\ e_2) & = & \mathcal{V}_s(e_1) \wedge \mathcal{V}_s(e_2)) & \mathcal{V}_{(v\circ s)}(\overline{1}) & = & \mathcal{V}_s(v) \\
\mathcal{V}_s(e_1, e_2) & = & \mathcal{V}_s(e_1) \wedge \mathcal{V}_s(e_2) & \mathcal{V}_s(x) & = & \mathbf{true}
\end{array}
$$

$$\mathcal{V}_s([e_0, \ldots, e_{p-1}]) = \bigwedge_{i=0}^{p-1} \mathcal{V}_s(e_i)$$

Figure 3.4 — Vérification de l'expression pour la création d'un vecteur parallèle

pour $e/c_g/\langle c_0, \ldots, c_{p-1}\rangle \rightharpoonup e^0/c_g^0/\langle c_0^0, \ldots, c_{p-1}^0\rangle \cdots \rightharpoonup e'/c'_g/\langle c'_0, \ldots, c'_{p-1}\rangle$.

L'évaluation complète d'une expression e donnée par le programmeur sera :

$$(\mathcal{T}_\bullet(e^p))[\bullet]/0/\langle 0, \ldots, 0\rangle \stackrel{*}{\rightharpoonup} v/c_g/\langle c_0, \ldots, c_{p-1}\rangle$$

Cela peut être lu comme «dans un environnement (de «traduction») vide, une substitution vide et un coût zéro, l'expression traduite de e^p s'évalue en v pour un coût global c_g et des coûts locaux $\langle c_0, \ldots, c_{p-1}\rangle$». Le temps d'exécution final BSP du programme e^p sera alors de $c_g + \max_{i=0}^{p-1}(c_i)$.

Définition 14 (*Relations de la sémantique à «petits pas»*).
Pour définir la relation $\rightharpoonup$ nous définissons préalablement deux types de réduction :

1. $\stackrel{i}{\rightharpoonup}$ *est la réduction locale d'une expression (dans une composante d'un vecteur parallèle) ;*

2. $\stackrel{\bowtie}{\rightharpoonup}$ *est la réduction globale d'une expression (en dehors d'un vecteur parallèle).*

avec :

$$\stackrel{i}{\rightharpoonup} = \underset{i}{\stackrel{\varepsilon}{\rightharpoonup}} \cup \underset{i}{\stackrel{\delta}{\rightharpoonup}} \qquad \stackrel{\bowtie}{\rightharpoonup} = \underset{\bowtie}{\stackrel{\varepsilon}{\rightharpoonup}} \cup \underset{\bowtie}{\stackrel{\delta}{\rightharpoonup}}$$

Nous commençons d'abord par une série d'axiomes (règles) qui sont communs aux deux relations. Celles-ci sont de la forme $e \rightharpoonup e', c$. Nous avons tout d'abord les relations de réduction de la β-réduction (avec substitution explicite sur une fermeture) et les règles de propagation de cette substitution. Notons que nous ne donnons pas de coûts à ces dernières car, dans une machine abstraite (ou dans le code machine), la substitution (sous la forme d'un environnement) est toujours directement accessible par toutes les sous-expressions où la substitution est propagée. Une exception est la règle de construction de la fermeture qui a un coût constant[7].

Les règles sont les suivantes. Pour commencer nous avons les règles classiques pour la β-réduction (avec substitution explicite) :

$$
\begin{array}{rcl}
(\overline{(\lambda.e)[s]}\ v) & \stackrel{\varepsilon}{\rightharpoonup} & e[v \circ s], 1 \\
\overline{n+1}[v \circ s] & \stackrel{\varepsilon}{\rightharpoonup} & \overline{n}[s], 0 \\
\overline{1}[v \circ s] & \stackrel{\varepsilon}{\rightharpoonup} & v[\bullet], 1 \\
(\mu.e)[s] & \stackrel{\varepsilon}{\rightharpoonup} & e[\mu.e \circ s], 1
\end{array}
$$

[7] Dans une étude de coût plus fine, le coût serait plutôt proportionnel à la taille de la substitution.

$$\dfrac{A \qquad B \qquad C}{\bullet, (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1}, 0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))) \rhd \langle 0, 0\rangle}$$

où A est

$$\dfrac{\dfrac{}{\bullet, (\lambda.\lambda.(\overline{1}\ 0)) \rhd \overline{(\lambda.\lambda.(\overline{1}\ 0))[\bullet]} \quad \overline{\bullet, \langle (f^1\ 0), (f^1\ 1)\rangle \rhd \langle \overline{\lambda.(\overline{1}\ 0)}[0 \circ \bullet], \overline{\lambda.(\overline{1}\ 0)}[1 \circ \bullet]\rangle}\ \ldots}{\bullet, (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0))) \rhd \langle \overline{\lambda.(\overline{1}\ 0)}[0 \circ \bullet], \overline{\lambda.(\overline{1}\ 0)}[1 \circ \bullet]\rangle}$$

où B est

$$\dfrac{\dfrac{\dfrac{}{\bullet, (\mathbf{mkpar}\ (\cdots)), \rhd\langle \overline{(\lambda.\lambda.\overline{2})}[0 \circ \bullet], \overline{(\lambda.\lambda.\mathbf{nc})}[1 \circ \bullet]\rangle}\ \ldots \quad \dfrac{}{\bullet, (\mathbf{mkpar}\ (\lambda.\overline{1})) \rhd \langle 0, 1\rangle}\ \ldots \quad \dfrac{}{\bullet, \langle (\overline{(\lambda.\lambda.\overline{2})}[0 \circ \bullet]\ 0), (\overline{(\lambda.\lambda.\mathbf{nc})}[1 \circ \bullet]\ 1)\rangle \rhd \langle f_0^2, f_1^2\rangle}\ \ldots}{\bullet, (\mathbf{apply}\ (\cdots)\ (\cdots)) \rhd \langle f_0^2, f_1^2\rangle} \quad \dfrac{}{\bullet, (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{F}))\ (\mathbf{send}\ \langle (\mathbf{init}\ f_0^2), (\mathbf{init}\ f_1^2)\rangle)) \rhd \langle f^3, f^3\rangle}\ \ldots}{\bullet, (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1}, 0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1})))) \rhd \langle f^3, f^3\rangle}$$

où C est

$$\dfrac{\dfrac{\dfrac{}{\bullet, (\overline{(\lambda.(\overline{1}\ 0)[0 \circ \bullet]} \rhd \overline{(\lambda.(\overline{1}\ 0)[0 \circ \bullet]}} \quad \dfrac{}{f3 \circ 0 \circ \bullet, \overline{1} \rhd 0}\ \ldots}{\bullet, (\overline{\lambda.(\overline{1}\ 0)[0 \circ \bullet]}\ f^3) \rhd 0} \quad \dfrac{\dfrac{}{\bullet, (\overline{\lambda.(\overline{1}\ 0)[0 \circ \bullet]} \rhd \overline{(\lambda.(\overline{1}\ 0)[0 \circ \bullet]}} \quad \dfrac{}{f3 \circ 0 \circ \bullet, \overline{1} \rhd 0}\ \ldots}{\bullet, (\overline{\lambda.(\overline{1}\ 0)[0 \circ \bullet]}\ f^3) \rhd 0}}{\bullet, \langle (\overline{\lambda.(\overline{1}\ 0)[0 \circ \bullet]}\ f^3), (\overline{\lambda.(\overline{1}\ 0)[1 \circ \bullet]}\ f^3)\rangle \rhd \langle 0, 0\rangle}$$

avec

$$\begin{array}{lcl} f^1 & = & \overline{(\lambda.\lambda.(\overline{1}\ 0))[\bullet]} \\ f_0^2 & = & \overline{(\lambda.\overline{2})[0 \circ 0 \circ \bullet]} \\ f_1^2 & = & \overline{(\lambda.\mathbf{nc})[1 \circ 1 \circ \bullet]} \\ f^3 & = & \overline{(\lambda.\mathbf{if}\ (0 \leq \overline{1})\mathbf{and}(\overline{<}\mathbf{p})\ \mathbf{then}\ (\mathbf{access}\ (\overline{2}, \overline{1}))\ \mathbf{else}\ \mathbf{nc})[[0, \mathbf{nc}] \circ \bullet]} \end{array}$$

Figure 3.5 — Exemple de la sémantique naturelle de BSML

ensuite nous avons celles pour la propagation de la substitution (la propager aux sous-termes) :

$$\begin{array}{rcl}
(\lambda.e)[s] & \xrightarrow{\varepsilon} & \overline{(\lambda.e)[s]}, 1 \\
v[s] & \xrightarrow{\varepsilon} & v, 0 \quad \text{si } v \neq \langle \cdots \rangle \text{ et } v \neq (v_0, v_1) \\
(e_1\ e_2)[s] & \xrightarrow{\varepsilon} & (e_1[s]\ e_2[s]), 0 \\
(e_1, e_2)[s] & \xrightarrow{\varepsilon} & (e_1[s], e_2[s]), 0 \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)[s] & \xrightarrow{\varepsilon} & \textbf{if } e_1[s] \textbf{ then } e_2[s] \textbf{ else } e_3[s], 0 \\
(\textbf{proj } e)[s] & \xrightarrow{\varepsilon} & (\textbf{proj } e[s]), 0 \\
(\textbf{put } e)[s] & \xrightarrow{\varepsilon} & (\textbf{put } e[s]), 0 \\
(\textbf{apply } e_1\ e_2)[s] & \xrightarrow{\varepsilon} & (\textbf{apply } e_1[s]\ e_2[s]), 0 \\
(\textbf{mkpar } e)[s] & \xrightarrow{\varepsilon} & (\textbf{mkpar } e[s]), 0 \\
\langle e_0, \ldots, e_{p-1} \rangle[s] & \xrightarrow{\varepsilon} & \langle e_0[s], \ldots, e_{p-1}[s] \rangle, 0 \\
[e_0, \ldots, e_{p-1}][s] & \xrightarrow{\varepsilon} & [e_0[s], \ldots, e_{p-1}[s]], 0
\end{array}$$

Enfin, nous avons les règles correspondant aux opérateurs du langage, les δ-réductions. Pour les règles qui sont génériques (locales comme globales), nous avons par exemple :

$$\begin{array}{rcl}
\textbf{access } ([v_0, \ldots, v_i, \ldots, v_{p-1}], i) & \xrightarrow{\delta} & v_i, 1 \\
\textbf{init } (n, f) & \xrightarrow{\delta} & [(f\ 0), \ldots, (f\ (n-1))], 1 \\
\textbf{fst } (v_1, v_2) & \xrightarrow{\delta} & v_1, 1 \\
\textbf{snd } (v_1, v_2) & \xrightarrow{\delta} & v_2, 1 \\
+\ (n_1, n_2) & \xrightarrow{\delta} & n_1 + n_2, 1 \\
\textbf{isnc nc} & \xrightarrow{\delta} & \textbf{true}, 1 \\
\textbf{isnc } v & \xrightarrow{\delta} & \textbf{false}, 1 \text{ si } v \neq \textbf{nc} \\
\textbf{if true then } e_2 \textbf{ else } e_3 & \xrightarrow{\delta} & e_2, 1 \\
\textbf{if false then } e_2 \textbf{ else } e_3 & \xrightarrow{\delta} & e_3, 1
\end{array}$$

Elles sont donc similaires à celles de la section précédente mais chacune avec un coût constant 1. Puis nous avons l'opérateur local :

$$\textbf{LocId } v \xrightarrow[i]{\delta} v, 1$$

et les opérateurs globaux :

$$\begin{array}{rcl}
\textbf{delpar } \langle f, \ldots, f \rangle / c_g / \langle c \rangle & \xrightarrow[\bowtie]{\delta} & f / c_g \oplus 1 / \langle c \rangle \\
\textbf{GloId } v / c_g / \langle c \rangle & \xrightarrow[\bowtie]{\delta} & v / c_g \oplus 1 / \langle c \rangle \text{ si } \mathcal{V}_\bullet(v) = \textbf{true}
\end{array}$$

$$\begin{array}{rcl}
\textbf{send } \langle [v_0^0, \ldots, v_{p-1}^0], \ldots, [v_0^{p-1}, \ldots, v_{p-1}^{p-1}] \rangle / c_g / \langle c_0, \ldots, c_{p-1} \rangle & \xrightarrow[\bowtie]{\delta} & \\
\langle [v_0^0, \ldots, v_0^{p-1}], \ldots, [v_{p-1}^0, \ldots, v_{p-1}^{p-1}] \rangle / c'_g / \langle 0, \ldots, 0 \rangle & &
\end{array}$$

tel que $c'_g = c_g \oplus 1 \oplus \max_{i=0}^{p-1}(c_i) \oplus \max_{i=0}^{p-1}(\max(\sum_{j=0}^{p-1} \mathcal{S}_\bullet(v_j^i), \sum_{j=0}^{p-1} \mathcal{S}_\bullet(v_i^j))) \otimes g \oplus l$

Dans cette dernière règle, nous prenons le maximum des coûts locaux puis le maximum des communications en un processeur et enfin une barrière. Ceci correspond à la fin de la super-étape courante. Au niveau des coûts, la première composante (resp. la deuxième) correspond au h^+ (resp. h^-) de la définition du modèle de coûts BSP. La taille d'une valeur est donnée par la fonction $\mathcal{S}$ (avec un environnement au départ vide) définie[8] à la figure 3.7. L'opération **send** étant notre opération de communication et de synchronisation, il rend à zéro les coûts locaux et ajoute les communications au coût global (ainsi que le maximal des coûts locaux).

La figure 3.6 donne les δ-règles des primitives parallèles. Celles-ci sont très proches de la définition des règles 3.14, 3.15, 3.16 de la sémantique naturelle. Notons la création dans ces règles de substitutions vides.

[8]Notons que nous ne donnons pas la taille pour les primitives parallèles ou les vecteurs parallèles car ceux-ci ne sont pas présents dans les vecteurs parallèles et ne sont donc pas transmissibles.

$$\mathbf{mkpar}\ f/c_g/\langle \mathbf{c}\rangle \xrightarrow[\bowtie]{\delta} \langle (f\ 0),\ldots,(f\ (p-1))\rangle[\bullet]/c_g \oplus 1/\langle \mathbf{c}\rangle \quad \text{si } \mathcal{V}_\bullet(f) = \mathbf{true} \tag{3.17}$$

$$\mathbf{apply}\langle f_0,\ldots,f_{p-1}\rangle\ \langle v_0,\ldots,v_{p-1}\rangle/c_g/\langle \mathbf{c}\rangle \xrightarrow[\bowtie]{\delta} \langle (f_0\ v_0),\ldots,(f_{p-1}\ v_{p-1})\rangle[\bullet]/c_g \oplus 1/\langle \mathbf{c}\rangle \tag{3.18}$$

$$\mathbf{put}\ \langle f_0,\ldots,f_{p-1}\rangle/c_g/\langle \mathbf{c}\rangle \xrightarrow[\bowtie]{\delta} (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{F}))\ (\mathbf{send}\ \langle (\mathbf{init}\ (p, f_0)),\ldots,(\mathbf{init}\ (p, f_{p-1}))\rangle))[\bullet]/c_g \oplus 1/\langle \mathbf{c}\rangle \tag{3.19}$$

$$\mathbf{proj}\ \langle v_0,\ldots,v_{p-1}\rangle/c_g/\langle \mathbf{c}\rangle \xrightarrow[\bowtie]{\delta} (\mathbf{delpar}\ (\mathbf{put}\ \langle (\lambda.v_0),\ldots,(\lambda.v_{p-1})\rangle))[\bullet]/c_g \oplus 1/\langle \mathbf{c}\rangle \tag{3.20}$$

Figure 3.6 — Règles des primitives parallèles

$$\begin{array}{rcl|rcl}
\mathcal{S}_s(e_1\ e_2) & = & \mathcal{S}_s(e_1) \oplus \mathcal{S}_s(e_2) \oplus 1 & \mathcal{S}_s(\lambda.e) & = & \mathcal{S}_{(x\circ s)}(e) \oplus 1 \\
\mathcal{S}_s(e_1, e_2) & = & \mathcal{S}_s(e_1) \oplus \mathcal{S}_s(e_2) \oplus 1 & \mathcal{S}_s(\overline{(\lambda.e)[s']}) & = & \mathcal{S}_{(x\circ s')}(e) \oplus 1 \\
\mathcal{S}(\mathbf{op}) & = & 1 & \mathcal{S}_s(\mu.e) & = & \mathcal{S}_{(x\circ s)}(e) \oplus 1 \\
\mathcal{S}(\mathbf{c}) & = & 1 & \mathcal{S}_{(v\circ s)}(\overline{n+1}) & = & \mathcal{S}_s(\overline{n}) \\
\mathcal{S}_s([e_0,\ldots,e_{p-1}]) & = & \sum_{i=0}^{p-1} \mathcal{S}_s(e_i) & \mathcal{S}_{(v\circ s)}(\overline{1}) & = & \mathcal{S}_s(v) \\
 & & & \mathcal{S}_s(x) & = & 0
\end{array}$$

$$\mathcal{S}_s(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) = \mathcal{S}_s(e_2) \oplus \mathcal{S}_s(e_2) \oplus \mathcal{S}_s(e_3) \oplus 1$$

Figure 3.7 — Fonction définissant la taille des données

Celles-ci seront immédiatement éliminées mais elles sont nécessaires pour la preuve d'équivalence avec la sémantique naturelle du langage. La primitive **put** se décompose en deux étapes, correspondant fidèlement à l'implantation actuelle de BSML (voir aussi au chapitre 4 et 5). En premier lieu, chaque processeur crée un tableau purement fonctionnel de valeurs en appliquant la fonction qu'il détient à tous les numéros de processeurs possibles. Notons que l'opération de création d'un tableau prend un argument entier définissant la taille du tableau (ici le nombre de processeurs). Ensuite, l'opération de plus bas niveau **send** fait les échanges et retourne un vecteur parallèle de tableaux. Cette règle 3.17 est une formalisation du mécanisme décrit dans le chapitre 5. La valeur à l'indice j du tableau au processeur i est envoyée, si ce n'est pas **nc**, au processeur j qui la stocke à l'indice i du tableau résultant. La fonction $\mathbf{F}$ construit le vecteur parallèle final de fonctions à partir des tableaux du vecteur

On constate aisément qu'il n'est pas toujours possible de faire des «réductions de tête». Il faut donc ajouter des règles de contexte. Ceci se fait à l'aide des règles suivantes, accompagnées des définitions de contextes de la figure 3.8. Nous avons des règles de «passage» permettant d'appliquer les règles génériques :

$$\frac{e \xrightarrow{\varepsilon} e', c}{e \xrightarrow[i]{\varepsilon} e', c} \qquad \frac{e \xrightarrow{\delta} e', c}{e \xrightarrow[i]{\delta} e', c}$$

$$\frac{e \xrightarrow{\varepsilon} e', c}{e/c_g/\langle \mathbf{c}\rangle \xrightarrow[\bowtie]{\varepsilon} e'/c_g \oplus c/\langle \mathbf{c}\rangle} \qquad \frac{e \xrightarrow{\delta} e', c}{e/c_g/\langle \mathbf{c}\rangle \xrightarrow[\bowtie]{\delta} e'/c_g \oplus c/\langle \mathbf{c}\rangle}$$

Définition 15 (*Sémantique à «petits pas»*).
La sémantique à «petits pas» $\rightharpoonup$ est définie par les deux règles de contexte (l'une locale, l'autre globale)

$$
\begin{array}{rcl}
\Gamma & ::= & [] \\
& | & \Gamma\ e \\
& | & v\ \Gamma \\
& | & (\Gamma, e) \\
& | & (v, \Gamma) \\
& | & \textbf{if}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{mkpar}\ \Gamma) \\
& | & (\textbf{apply}\ \Gamma\ e) \\
& | & (\textbf{apply}\ v\ \Gamma) \\
& | & (\textbf{put}\ \Gamma) \\
& | & (\textbf{proj}\ \Gamma)
\end{array}
\qquad
\begin{array}{rcl}
\Delta_i & ::= & \Delta_i\ e \\
& | & v\ \Delta_i \\
& | & (\Delta_i, e) \\
& | & (v, \Delta_i) \\
& | & \textbf{if}\ \Delta_i\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{mkpar}\ \Delta_i) \\
& | & (\textbf{apply}\ \Delta_i\ e) \\
& | & (\textbf{apply}\ v\ \Delta_i) \\
& | & (\textbf{put}\ \Delta_i) \\
& | & (\textbf{proj}\ \Delta_i) \\
& | & \langle e, \ldots, \overbrace{\Gamma^l[e]}^{i}, \ldots, e \rangle
\end{array}
\qquad
\begin{array}{rcl}
\Gamma^l & ::= & [] \\
& | & \Gamma^l\ e \\
& | & v\ \Gamma^l \\
& | & (\Gamma^l, e) \\
& | & (v, \Gamma^l) \\
& | & \textbf{if}\ \Gamma^l\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & [\Gamma^l, e_1, \ldots, e_n] \\
& | & [v_0, \Gamma^l, \ldots, e_n] \\
& | & \ldots \\
& | & [v_0, v_1, \ldots, \Gamma^l]
\end{array}
$$

Figure 3.8 — Contextes d'évaluation

suivantes :

$$\frac{e/c_g/\langle c_0, \ldots, c_{p-1}\rangle \overset{\bowtie}{\rightharpoonup} e'/c'_g/\langle c'_0, \ldots, c'_{p-1}\rangle}{\Gamma[e]/c_g/\langle c_0, \ldots, c_{p-1}\rangle \rightharpoonup \Gamma[e']/c'_g/\langle c'_0, \ldots, c'_{p-1}\rangle}$$

$$\frac{e \overset{i}{\rightharpoonup} e', c}{\Delta^i[e]/c_g/\langle \ldots, c_i, \ldots\rangle \rightharpoonup \Delta^i[e']/c_g/\langle \ldots, c_i \oplus c, \ldots\rangle}$$

Le contexte Γ (resp. Δ^i) permet l'évaluation (application d'une règle) de l'expression en dehors d'un vecteur parallèle (resp. dans la $i$ème composante d'un vecteur, c'est-à-dire au $i$ème processeur).

Notons que la réduction locale ne modifie qu'une composante d'un vecteur parallèle, et donc ne modifie qu'une composante du vecteur de coût. Les réductions globales ne modifient que le coût global, sauf la règle **send** (fin de la super-étape) qui prend le maximal des coûts locaux pour l'ajouter au coût global (les coûts locaux sont ensuite réduits à zéro).

La figure 3.9 donne l'évaluation de l'exemple donné dans la section précédente mais cette fois-ci avec notre sémantique à «petits pas» (avec toujours deux processeurs). Nous avons alors les résultats suivants.

Lemme 2 (*Confluence forte*)

Soit une expression e.

Si	$e/c_g/\langle c_0, \cdots, c_{p-1}\rangle \rightharpoonup e^1/c^1_g/\langle c^1_0, \cdots, c^1_{p-1}\rangle$
et	$e/c_g/\langle c_0, \cdots, c_{p-1}\rangle \rightharpoonup e^2/c^2_g/\langle c^2_0, \cdots, c^2_{p-1}\rangle$
alors il existe	*une expression* e_3 *et des coûts* c^3_g *et* $\langle c^3_0, \cdots, c^3_{p-1}\rangle$
tels que	$e^1/c^1_g/\langle c^1_0, \cdots, c^1_{p-1}\rangle \rightharpoonup e^3/c^3_g/\langle c^3_0, \cdots, c^3_{p-1}\rangle$
et	$e^2/c^2_g/\langle c^2_0, \cdots, c^2_{p-1}\rangle \rightharpoonup e^3/c^3_g/\langle c^3_0, \cdots, c^3_{p-1}\rangle$.

Preuve . Voir en section 4 de l'annexe de ce chapitre. ■

Théorème 4 (*Confluence*)

Soit une expression «programmeur» e^p.

Si	$e = \mathcal{T}_\bullet(e^p)$
alors si	$e[\bullet]/0/\langle 0, \cdots, 0\rangle \overset{*}{\rightharpoonup} v/c_g/\langle c_0, \cdots, c_{p-1}\rangle$
et	$e[\bullet]/0/\langle 0, \cdots, 0\rangle \overset{*}{\rightharpoonup} v'/c'_g/\langle c'_0, \cdots, c'_{p-1}\rangle$
alors	$v = v'$, $c_g = c'_g$ *et* $\forall i \in \{0, \ldots, p-1\}\ c_i = c'_i$.

Preuve . La relation $\rightharpoonup$ est fortement confluente, donc d'après le lemme 1, elle est confluente. ■

Nous avons donc le fait que si un programme s'évalue avec un coût c alors ce même programme s'évaluera toujours avec le coût c.

$$
\begin{array}{rcl}
 & & e[\bullet]/0/\langle 0,0\rangle \\
= & & (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet]/0/\langle 0,0\rangle \\
\rightharpoonup & & (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))[\bullet]\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet])/0/\langle 0,0\rangle \\
\rightharpoonup & & (\mathbf{apply}\ (\mathbf{mkpar}\ \underline{(\lambda.\lambda.(\overline{1}\ 0))[\bullet]})\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet])/0/\langle 0,0\rangle \\
\rightharpoonup & & (\mathbf{apply}\ (\mathbf{mkpar}\ \overline{(\lambda.\lambda.(\overline{1}\ 0))[\bullet]})\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet])/1/\langle 0,0\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{((\lambda.\lambda.(\overline{1}\ 0))[\bullet]\ 0)}, \overline{((\lambda.\lambda.(\overline{1}\ 0))[\bullet]\ 1)}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet])/2/\langle 0,0\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{((\lambda.\lambda.(\overline{1}\ 0))[\bullet]\ 1)}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet])/2/\langle 1,0\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet])/2/\langle 1,1\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1})))[\bullet]))/2/\langle 1,1\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))[\bullet]\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet]))/2/\langle 1,1\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ \underline{(\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc}))[\bullet]})\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet]))/2/\langle 1,1\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ \overline{(\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc}))[\bullet]})\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet]))/3/\langle 1,1\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle\overline{((\lambda.\mathbf{if}\cdots)[\bullet]\ 0)}, \overline{((\lambda.\mathbf{if}\cdots)[\bullet]\ 1)}\rangle\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet]))))/4/\langle 1,1\rangle \\
\cdots & & \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\lambda.\mathbf{nc}[1\circ\bullet]}\rangle\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet])))/4/\langle 6,6\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\lambda.\mathbf{nc}[1\circ\bullet]}\rangle\ \underline{(\mathbf{mkpar}\ (\lambda.\overline{1})[\bullet])}))/4/\langle 6,6\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\lambda.\mathbf{nc}[1\circ\bullet]}\rangle\ \overline{(\mathbf{mkpar}\ (\lambda.\overline{1})[\bullet])}))/5/\langle 6,6\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\lambda.\mathbf{nc}[1\circ\bullet]}\rangle\ \langle\overline{((\lambda.\overline{1})[\bullet]\ 0)}, \overline{((\lambda.\overline{1})[\bullet]\ 1)}\rangle)))/6/\langle 6,6\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\lambda.\mathbf{nc}[1\circ\bullet]}\rangle\ \langle\overline{1}[0\circ\bullet], \overline{((\lambda.\overline{1})[\bullet]\ 1)}\rangle)))/6/\langle 7,6\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\lambda.\mathbf{nc}[1\circ\bullet]}\rangle\ \langle\overline{1}[0\circ\bullet], \overline{1}[1\circ\bullet]\rangle)))/6/\langle 7,7\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\lambda.\mathbf{nc}[1\circ\bullet]}\rangle\ \langle\overline{1}[0\circ\bullet], 1\rangle)))/6/\langle 7,8\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ (\underline{\mathbf{apply}}\ \langle\overline{(\lambda.\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\lambda.\mathbf{nc}[1\circ\bullet]}\rangle\ \langle 0,1\rangle)))/6/\langle 8,8\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle(\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ \langle\overline{((\lambda.\lambda.\overline{2})[0\circ\bullet]\ 0)}, \overline{((\lambda.\lambda.\mathbf{nc}[1\circ\bullet]\ 1)}\rangle))/7/\langle 8,8\rangle \\
\cdots & & \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{put}\ \langle\overline{(\lambda.\overline{2})[0\circ 0\circ\bullet]}, \overline{(\lambda.\mathbf{nc}[1\circ 1\circ\bullet]}\rangle))/7/\langle 10,10\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{apply}\ (\mathbf{mkpar}\ \overline{(\lambda.\mathbf{F})[\bullet]})\ (\mathbf{send}\ \langle(\mathbf{init}\ \overline{(\lambda.\overline{2})[0\circ 0\circ\bullet]}), (\mathbf{init}\ \overline{(\lambda.\mathbf{nc}}[1\circ 1\circ\bullet])\rangle)))/8/\langle 10,10\rangle \\
\cdots & & \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{apply}\ \langle(\overline{\mathbf{F}})[0\circ\bullet], (\overline{\mathbf{F}})[1\circ\bullet]\rangle\ (\mathbf{send}\ \langle[0,0], [\mathbf{nc},\mathbf{nc}]\rangle)))/9/\langle 14,14\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{apply}\ \langle(\overline{\mathbf{F}})[0\circ\bullet], (\overline{\mathbf{F}})[1\circ\bullet]\rangle\ \langle[0,\mathbf{nc}], [0,\mathbf{nc}]\rangle)))/9\oplus 1\oplus\max(14,14)\oplus\max(\max(2,1),\max(0,1))\otimes g\oplus l/\langle 0,0\rangle \\
= & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ (\mathbf{apply}\ \langle(\overline{\mathbf{F}})[0\circ\bullet], (\overline{\mathbf{F}})[1\circ\bullet]\rangle\ \langle[0,\mathbf{nc}], [0,\mathbf{nc}]\rangle)))/24\oplus(2\otimes g)\oplus l/\langle 0,0\rangle \\
\rightharpoonup & & (\mathbf{apply}\ \langle(\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ \langle((\overline{\mathbf{F}})[0\circ\bullet]\ [0,\mathbf{nc}]), ((\overline{\mathbf{F}})[1\circ\bullet]\ [0,\mathbf{nc}])\rangle)/25\oplus(2\otimes g)\oplus l/\langle 0,0\rangle \\
\cdots & & \\
\rightharpoonup & & (\mathbf{apply}\ \langle\overline{(\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.\lambda.(\overline{1}\ 0))[1\circ\bullet]}\rangle\ \langle f_0^1, f_1^1\rangle)/25\oplus(2\otimes g)\oplus l/\langle 2,2\rangle \\
\rightharpoonup & & \langle\overline{((\lambda.\lambda.(\overline{1}\ 0))[0\circ\bullet]}\ f_0^1), \overline{((\lambda.(\overline{1}\ 0))[1\circ\bullet]}\ f_1^1)\rangle/27\oplus(2\otimes g)\oplus l/\langle 2,2\rangle \\
\cdots & & \\
\rightharpoonup & & \langle 0,0\rangle/27\oplus(2\otimes g)\oplus l/\langle 15,15\rangle
\end{array}
$$

avec

$$
\begin{array}{rcl}
f_0^1 & = & (\overline{\lambda.\mathbf{if}\ (0\leq\overline{1})\mathbf{and}(\overline{1}<\mathbf{p})\ \mathbf{then}\ (\mathbf{access}\ (\overline{2},\overline{1}))\ \mathbf{else}\ \mathbf{nc}})[[0,\mathbf{nc}]\circ 0\circ\bullet] \\
f_1^1 & = & (\lambda.\mathbf{if}\ (0\leq\overline{1})\mathbf{and}(\overline{1}<\mathbf{p})\ \mathbf{then}\ (\mathbf{access}\ (\overline{2},\overline{1}))\ \mathbf{else}\ \mathbf{nc})[[0,\mathbf{nc}]\circ 1\circ\bullet]
\end{array}
$$

$$
\begin{array}{rcl}
\text{coût BSP final} & = & 27\oplus(2\otimes g)\oplus l\oplus\max(15,15) \\
 & = & 42\oplus(2\otimes g)\oplus l
\end{array}
$$

Figure 3.9 — Exemple de la sémantique à «petits pas» de BSML

$$
\begin{array}{rcl}
\mathcal{P}_i((e)[s]) & = & \mathcal{P}_i(e)[\mathcal{P}_i(s)] \\
\mathcal{P}_i(\lambda.e) & = & \lambda.\mathcal{P}_i(e) \\
\mathcal{P}_i(\overline{(\lambda.e)[s]}) & = & \overline{(\lambda.\mathcal{P}_i(e))[\mathcal{P}_i(s)]} \\
\mathcal{P}_i(e_1\ e_2) & = & (\mathcal{P}_i(e_1)\ \mathcal{P}_i(e_2)) \\
\mathcal{P}_i(e_1, e_2) & = & (\mathcal{P}_i(e_1), \mathcal{P}_i(e_2)) \\
\mathcal{P}_i(\mathbf{apply}\ e_1\ e_2) & = & (\mathbf{apply}\ \mathcal{P}_i(e_1)\ \mathcal{P}_i(e_2)) \\
\mathcal{P}_i([e_0, \ldots, e_{p-1}]) & = & [\mathcal{P}_i(e), \ldots, \mathcal{P}_i(e)] \\
\mathcal{P}_i(\langle \ldots, e_i, \ldots \rangle) & = & \langle \mathcal{P}_i(e_i) \rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{P}_i(\mu.e) & = & (\mu.\mathcal{P}_i(e)) \\
\mathcal{P}_i(\mathbf{mkpar}\ e) & = & (\mathbf{mkpar}\ \mathcal{P}_i(e)) \\
\mathcal{P}_i(\mathbf{put}\ e) & = & (\mathbf{put}\ \mathcal{P}_i(e)) \\
\mathcal{P}_i(\mathbf{proj}\ e) & = & (\mathbf{proj}\ \mathcal{P}_i(e)) \\
\mathcal{P}_i(\mathbf{init}\ e) & = & (\mathbf{init}\ \mathcal{P}_i(e)) \\
\mathcal{P}_i(\mathbf{c}) & = & \mathbf{c} \\
\mathcal{P}_i(\mathbf{op}) & = & \mathbf{op} \\
\mathcal{P}_i(\overline{\mathbf{n}}) & = & \overline{\mathbf{n}}
\end{array}
$$

$$\mathcal{P}_i(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) = \mathbf{if}\ \mathcal{P}_i(e_1)\ \mathbf{then}\ \mathcal{P}_i(e_2)\ \mathbf{else}\ \mathcal{P}_i(e_3)$$

Figure 3.10 — Projection d'une expression sur un processeur i

Théorème 5 (*Équivalence*)

Soit une expression programmeur e^p telle que $e = \mathcal{T}_\bullet(e^p)$. Alors :

1. *Si $e[\bullet]/0/\langle 0, \ldots, 0\rangle \stackrel{*}{\rightharpoonup} v/c_g/\langle c_0, \ldots, c_{p-1}\rangle$ alors $\bullet, e \rhd_g v$;*
2. *Si $\bullet, e \rhd_g v$ alors $e[\bullet]/0/\langle 0, \ldots, 0\rangle \stackrel{*}{\rightharpoonup} v/c_g/\langle c_0, \ldots, c_{p-1}\rangle$*

Preuve. On ignore les coûts de la sémantique de $\rightharpoonup$ afin de simplifier la lecture de la preuve (les coûts n'apparaissant pas dans la sémantique $\rhd$). Notons qu'avec la propriété 1, l'expression e est sans variables libres. Nous avons donc :

1. Si l'on a $e[\bullet] \rightharpoonup e_1 \rightharpoonup \cdots \rightharpoonup e_n \rightharpoonup v$, il s'en suit par le lemme 12 que $\bullet, v \rhd_g v$, puis il existe une substitution s telle que $s, e_n \rhd_g v$ par le lemme 13, et ainsi de suite jusqu'à $\bullet, e \rhd_g v$ par applications répétées du lemme 13
2. Par application du lemme 11.

Les preuves des lemmes 11, 12 et 13 se trouvent dans la section 3.A.3 (en annexe de ce chapitre). ■

S'il paraît normal que tous les processeurs effectuent en même temps le **send**, il semble que les autres opérations globales puissent être evaluées de manière asynchrone. Dans un paradigme SPMD, le programme est identique en chaque processeur. Tout ce qui ne nécessite pas de communications peut donc être exécuté de manière asynchrone, chose qui n'est pas visible dans cette sémantique. C'est pourquoi nous proposons maintenant une évaluation distribuée où les termes sont des vecteurs de termes.

3.2.4 Sémantique distribuée

Les termes de la sémantique distribuée, appelés termes projetés, sont ceux présentés précédemment mais avec des vecteurs parallèles de taille 1, c'est-à-dire :

$$e ::= \cdots \mid \langle e \rangle$$

Les valeurs sont modifiées de la même manière.

Pour passer de la sémantique à «petits pas» dont les termes sont des termes de vecteurs parallèles, à une évaluation distribuée (plus proche de l'exécution BSP), dont les termes sont des vecteurs de termes, nous allons définir une *projection* de nos termes. Cette projection, pour le processeur i, est une fonction de nos expressions usuelles vers les expressions distribuées. Elle est présentée à la figure 3.10 où cette projection est définie comme suit sur les substitutions :

$$
\begin{array}{rcl}
\mathcal{P}_i(\bullet) & = & \bullet \\
\mathcal{P}_i(v \circ s) & = & \mathcal{P}_i(v) \circ \mathcal{P}_i(s)
\end{array}
$$

En fait les expressions ne sont pas modifiées, sauf les vecteurs parallèles, dont nous ne prenons que la $i$ème composante. La transformation d'une expression e, en vue de son évaluation distribuée, procède alors en deux étapes :

$$
\begin{array}{rcl}
(\overline{(\lambda.e)[s]}\ v) & \overset{\varepsilon}{\rightsquigarrow} & e[v \circ s] \\
\overline{n+1}[v \circ s] & \overset{\varepsilon}{\rightsquigarrow} & \overline{n}[s] \\
\overline{1}[v \circ s] & \overset{\varepsilon}{\rightsquigarrow} & v[\bullet] \\
(\mu.e)[s] & \overset{\varepsilon}{\rightsquigarrow} & \overline{e[\mu.e \circ s]} \\
(\lambda.e)[s] & \overset{\varepsilon}{\rightsquigarrow} & \overline{(\lambda.e)[s]} \\
v[s] & \overset{\varepsilon}{\rightsquigarrow} & v \quad \text{si } v \neq \langle \cdots \rangle \text{ et } v \neq (v_0, v_1) \\
(e_1\ e_2)[s] & \overset{\varepsilon}{\rightsquigarrow} & (e_1[s]\ e_2[s]) \\
(e_1, e_2)[s] & \overset{\varepsilon}{\rightsquigarrow} & (e_1[s], e_2[s]) \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)[s] & \overset{\varepsilon}{\rightsquigarrow} & \textbf{if } e_1[s] \textbf{ then } e_2[s] \textbf{ else } e_3[s] \\
(\textbf{proj}\ e)[s] & \overset{\varepsilon}{\rightsquigarrow} & (\textbf{proj}\ e[s]) \\
(\textbf{put}\ e)[s] & \overset{\varepsilon}{\rightsquigarrow} & (\textbf{put}\ e[s]) \\
(\textbf{apply}\ e_1\ e_2)[s] & \overset{\varepsilon}{\rightsquigarrow} & (\textbf{apply}\ e_1[s]\ e_2[s]) \\
(\textbf{mkpar}\ e)[s] & \overset{\varepsilon}{\rightsquigarrow} & (\textbf{mkpar}\ e[s]) \\
\langle e \rangle[s] & \overset{\varepsilon}{\rightsquigarrow} & \langle e[s] \rangle \\
[e_0, \ldots, e_{p-1}][s] & \overset{\varepsilon}{\rightsquigarrow} & [e_0[s], \ldots, e_{p-1}[s]]
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{access}\ ([v_0, \ldots, v_i, \ldots, v_{p-1}], i) & \overset{\delta}{\rightsquigarrow} & v_i \\
\textbf{init}\ (n, f) & \overset{\delta}{\rightsquigarrow} & [(f\,0), \ldots, (f\,(n-1))] \\
\textbf{fst}\ (v_1, v_2) & \overset{\delta}{\rightsquigarrow} & v_1 \\
\textbf{snd}\ (v_1, v_2) & \overset{\delta}{\rightsquigarrow} & v_2 \\
+\ (n_1, n_2) & \overset{\delta}{\rightsquigarrow} & n_1 + n_2 \\
\textbf{isnc nc} & \overset{\delta}{\rightsquigarrow} & \textbf{true} \\
\textbf{isnc}\ v & \overset{\delta}{\rightsquigarrow} & \textbf{false si } v \neq \textbf{nc} \\
\textbf{if true then } e_2 \textbf{ else } e_3 & \overset{\delta}{\rightsquigarrow} & e_2 \\
\textbf{if false then } e_2 \textbf{ else } e_3 & \overset{\delta}{\rightsquigarrow} & e_3
\end{array}
$$

Figure 3.11 — Réductions fonctionnelles et opérations génériques

1. L'expression est projetée sur chaque processeur donnant p termes projetés ;
2. Ces termes sont rassemblés en un terme distribué.

Nous avons pour une expression e le terme distribué suivant :

$$\langle\!\langle \mathcal{P}_0(e), \ldots, \mathcal{P}_{p-1}(e) \rangle\!\rangle$$

La sémantique distribuée a donc la forme suivante :

$$\langle\!\langle e_0, \ldots, e_{p-1} \rangle\!\rangle \rightsquigarrow \langle\!\langle e'_0, \ldots, e'_{p-1} \rangle\!\rangle$$

Nous notons $\overset{*}{\rightsquigarrow}$ pour la fermeture transitive et réflexive de $\rightsquigarrow$.

Définition 16 (*Relations de la sémantique distribuée*).
L'évaluation distribuée (notée $\rightsquigarrow$) peut-être définie en deux parties :

1. *L'évaluation locale ($\overset{i}{\rightsquigarrow}$ et $\overset{\bowtie_i}{\rightsquigarrow}$) d'un terme projeté ;*
2. *L'évaluation globale $\rightsquigarrow$ de* **send** *(un par terme projeté), ce qui met en jeu des communications (et une synchronisation) entre les processeurs.*

avec :

$$\overset{i}{\rightsquigarrow} = \underset{i}{\overset{\varepsilon}{\rightsquigarrow}} \cup \underset{i}{\overset{\delta}{\rightsquigarrow}} \qquad \overset{\bowtie_i}{\rightsquigarrow} = \underset{\bowtie_i}{\overset{\varepsilon}{\rightsquigarrow}} \cup \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}}$$

L'évaluation distribuée d'un terme distribué ne peut se faire que si les termes projetés proviennent tous d'une même expression. Nous notons $\overset{\bowtie_i}{\rightsquigarrow}$ l'évaluation au processeur i d'une règle globale et $\overset{i}{\rightsquigarrow}$ l'évaluation locale (dans un vecteur projeté) au processeur i.

Nous définissons les règles de substitution, propagation de la substitution (dans les sous-expressions), δ-règles génériques (règles classiques de la programmation fonctionnelle) de la même manière que dans la sémantique à petits pas (figure 3.11). Les δ-règles des opérateurs locaux et globaux sont :

$$\textbf{LocId}\ v \quad \underset{i}{\overset{\delta}{\rightsquigarrow}} \quad v$$

$$
\begin{array}{rcl}
\textbf{delpar}\ \langle f \rangle & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & f \\
\textbf{GloId}\ v & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & v \text{ si } \mathcal{V}_\bullet(v) = \textbf{true}
\end{array}
$$

Les règles des primitives parallèles sont alors les suivantes :

$$\begin{aligned}
\mathbf{apply}\ \langle f\rangle\ \langle v\rangle &\overset{\delta}{\underset{\bowtie_i}{\rightsquigarrow}} \langle (f\ v)\rangle[\bullet] \\
\mathbf{proj}\ \langle v\rangle &\overset{\delta}{\underset{\bowtie_i}{\rightsquigarrow}} (\mathbf{delpar}\ (\mathbf{put}\ \langle \lambda.v\rangle))[\bullet] \\
\mathbf{put}\ \langle f\rangle &\overset{\delta}{\underset{\bowtie_i}{\rightsquigarrow}} (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{F}))\ (\mathbf{send}\ \langle(\mathbf{init}\ f)\rangle))[\bullet] \\
\mathbf{mkpar}\ f &\overset{\delta}{\underset{\bowtie_i}{\rightsquigarrow}} \langle (f\ i)\rangle[\bullet] \quad \text{si } \mathcal{V}_\bullet(f) = \mathbf{true}
\end{aligned}$$

Notons que les primitives parallèles fonctionnent de la même manière qu'auparavant mais avec des vecteurs parallèles de taille fixe 1. En effet, chaque processeur a une composante du vecteur (un «bout»). Seule la règle de l'opération de communication **send** va réellement être modifiée et ce comme suit :

Définition 17 (*Sémantique distribuée*).
Nous avons les trois règles de contexte suivantes pour définir l'évaluation distribuée $\rightsquigarrow$:

$$\frac{e_i \overset{\bowtie_i}{\rightsquigarrow} e_i'}{\langle\!\langle e_0, \ldots, \Gamma[e_i], \ldots, e_{p-1}\rangle\!\rangle \rightsquigarrow \langle\!\langle e_0, \ldots, \Gamma[e_i'], \ldots, e_{p-1}\rangle\!\rangle}$$

$$\frac{e_i \overset{i}{\rightsquigarrow} e_i'}{\langle\!\langle e_0, \ldots, \Delta[e_i], \ldots, e_{p-1}\rangle\!\rangle \rightsquigarrow \langle\!\langle e_0, \ldots, \Delta[e_i'], \ldots, e_{p-1}\rangle\!\rangle}$$

$$\frac{\forall i \in \{0,\ldots,p-1\} \quad \textit{si } e_i = \Gamma[\mathbf{send}\ \langle[v_0^i, \ldots, v_{p-1}^i]\rangle] \quad \textit{alors } e_i' = \Gamma[\langle[v_i^0, \ldots, v_i^{p-1}]\rangle]}{\langle\!\langle e_0, \ldots, e_{p-1}\rangle\!\rangle \rightsquigarrow \langle\!\langle e_0', \ldots, e_{p-1}'\rangle\!\rangle}$$

Ces règles utilisent les règles de passage suivantes :

$$\frac{e \overset{\varepsilon}{\rightsquigarrow} e'}{e \overset{\varepsilon}{\underset{i}{\rightsquigarrow}} e'} \qquad \frac{e \overset{\delta}{\rightsquigarrow} e'}{e \overset{\delta}{\underset{i}{\rightsquigarrow}} e'} \qquad \frac{e \overset{\varepsilon}{\rightsquigarrow} e'}{e \overset{\varepsilon}{\underset{\bowtie_i}{\rightsquigarrow}} e'} \qquad \frac{e \overset{\delta}{\rightsquigarrow} e'}{e \overset{\delta}{\underset{\bowtie_i}{\rightsquigarrow}} e'}$$

La figure 3.1 donne l'évaluation de l'exemple donné dans la section précédente avec notre sémantique distribuée (avec toujours deux processeurs). Notons que l'entrelacement des calculs des processeurs, donné dans cet exemple, n'est qu'une combinaison parmis tant d'autres. Nous avons alors les résultats suivants.

Lemme 3 (*Confluence forte*)

Soit $\langle\!\langle e_0, \ldots, e_{p-1}\rangle\!\rangle$ un terme distribué.
Si $\langle\!\langle e_0, \ldots, e_{p-1}\rangle\!\rangle \rightsquigarrow \langle\!\langle e_0^1, \ldots, e_{p-1}^1\rangle\!\rangle$
et $\langle\!\langle e_0, \ldots, e_{p-1}\rangle\!\rangle \rightsquigarrow \langle\!\langle e_0^2, \ldots, e_{p-1}^2\rangle\!\rangle$
alors il existe $\langle\!\langle e_0^3, \ldots, e_{p-1}^3\rangle\!\rangle$
tel que $\langle\!\langle e_0^1, \ldots, e_{p-1}^1\rangle\!\rangle \rightsquigarrow \langle\!\langle e_0^3, \ldots, e_{p-1}^3\rangle\!\rangle$
et $\langle\!\langle e_0^2, \ldots, e_{p-1}^2\rangle\!\rangle \rightsquigarrow \langle\!\langle e_0^3, \ldots, e_{p-1}^3\rangle\!\rangle$.

Preuve. Voir en section 3.A.4 de l'annexe de ce chapitre. ■

Théorème 6 (*Confluence*)

Soit e^p une expression «programmeur» telle que $e = \mathcal{T}_\bullet(e^p)$.
Si $\forall i \in \{0,\ldots,p-1\}\ e_i = \mathcal{P}_i(e)$
alors si $\langle\!\langle e_0[\bullet], \ldots, e_{p-1}[\bullet]\rangle\!\rangle \rightsquigarrow \langle\!\langle v_0, \ldots, v_{p-1}\rangle\!\rangle$
et $\langle\!\langle e_0[\bullet], \ldots, e_{p-1}[\bullet]\rangle\!\rangle \rightsquigarrow \langle\!\langle v_0', \ldots, v_{p-1}'\rangle\!\rangle$
alors $\forall i \in \{0,\ldots,p-1\}\ v_i = v_i'$.

Preuve. La relation $\rightsquigarrow$ est fortement confluente, donc d'après le lemme 1, elle est confluente. ■

$$\langle\!\langle e[\bullet], e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1})))))[\bullet], e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))[\bullet]\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet]), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ (\mathbf{mkpar}\ \underline{(\lambda.\lambda.(\overline{1}\ 0))}[\bullet])\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet]), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ (\underline{\mathbf{mkpar}\ (}\overline{\lambda.\lambda.(\overline{1}\ 0)})[\bullet])\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet]), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle\underline{((\overline{\lambda.\lambda.(\overline{1}\ 0)}}))[\bullet]\ 0)\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet]), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle\underline{(\lambda.(\overline{1}\ 0))}[0\circ\bullet]\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet]), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet]), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))\ (\mathbf{mkpar}\ (\lambda.\overline{1})))[\bullet])), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})))[\bullet]\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet]))), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ \underline{(\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc}))}[\bullet])\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet]))), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{put}\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\overline{\lambda.\mathbf{if}\ =(\overline{1},0)\ \mathbf{then}\ (\lambda.\lambda.\overline{2})\ \mathbf{else}\ (\lambda.\lambda.\mathbf{nc})})[\bullet])\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet]))), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\lambda.(\overline{1}\ 0))[0\circ\bullet]\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle((\overline{\lambda.\mathbf{if}\cdots})[\bullet]\ 0)\rangle\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet]))), e[\bullet]\rangle\!\rangle$$
$$\cdots$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle(\overline{\lambda.\lambda.\overline{2}})[0\circ\bullet]\rangle\ (\mathbf{mkpar}\ (\lambda.\overline{1}))[\bullet]))), e[\bullet]\rangle\!\rangle$$
$$\cdots$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{put}\ (\mathbf{apply}\ \langle(\overline{\lambda.\lambda.\overline{2}})[0\circ\bullet]\rangle\ \langle 0\rangle))), e[\bullet]\rangle\!\rangle$$
$$\cdots$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{put}\ \langle(\overline{\lambda.\overline{2}})[0\circ 0\circ\bullet]\rangle)), e[\bullet]\rangle\!\rangle$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{apply}\ (\mathbf{mkpar}\ (\overline{\lambda.\mathbf{F}})[\bullet])\ (\mathbf{send}\ \langle(\mathbf{init}\ (\overline{\lambda.\overline{2}})[0\circ 0\circ\bullet])\rangle)))), e[\bullet]\rangle\!\rangle$$
$$\cdots$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{apply}\ \langle(\overline{\lambda.\mathbf{F}})[0\circ\bullet]\rangle\ (\mathbf{send}\ \langle(\mathbf{init}\ (\overline{\lambda.\overline{2}})[0\circ 0\circ\bullet])\rangle))), e[\bullet]\rangle\!\rangle$$
$$\cdots$$
$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{apply}\ \langle(\overline{\lambda.\mathbf{F}})[0\circ\bullet]\rangle\ (\mathbf{send}\ \langle[0,0]\rangle))), e[\bullet]\rangle\!\rangle$$

$\cdots$ calcul similaire pour le processeur 1

$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{apply}\ \langle(\overline{\lambda.\mathbf{F}})[0\circ\bullet]\rangle\ (\mathbf{send}\ \langle[0,0]\rangle))), (\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[1\circ\bullet]\rangle\ (\mathbf{apply}\ \langle(\overline{\lambda.\mathbf{F}})[1\circ\bullet]\rangle\ (\mathbf{send}\ \langle[\mathbf{nc},\mathbf{nc}]\rangle)))\rangle\!\rangle$$

synchronisation au niveau des deux **send**

$$\rightsquigarrow \langle\!\langle(\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[0\circ\bullet]\rangle\ (\mathbf{apply}\ \langle(\overline{\lambda.\mathbf{F}})[0\circ\bullet]\rangle\ (\mathbf{send}\ \langle[0,\mathbf{nc}]\rangle))), (\mathbf{apply}\ \langle(\overline{\lambda.(\overline{1}\ 0)})[1\circ\bullet]\rangle\ (\mathbf{apply}\ \langle(\overline{\lambda.\mathbf{F}})[1\circ\bullet]\rangle\ (\mathbf{send}\ \langle[0,\mathbf{nc}]\rangle)))\rangle\!\rangle$$

$\cdots$ calculs asynchrones sur les deux processeurs pour terminer l'évaluation

$$\rightsquigarrow \langle\!\langle\langle 0\rangle, \langle 0\rangle\rangle\!\rangle$$

Table 3.1 — Exemple de la sémantique distribuée de BSML

Théorème 7 (*Équivalence*)

Soit e^p une expression «programmeur» telle que $e = \mathcal{T}_\bullet(e^p)$. Alors :

1. *Si $e[\bullet]/0/\langle 0,\ldots,0\rangle \stackrel{*}{\rightarrow} v/c_g/\langle c_0,\ldots,c_{p-1}\rangle$ alors*
   $\langle\langle \mathcal{P}_0(e[\bullet]),\ldots,\mathcal{P}_{p-1}(e[\bullet])\rangle\rangle \stackrel{*}{\rightsquigarrow} \langle\langle v_0,\ldots,v_{p-1}\rangle\rangle$ tel que $\forall i \in \{0,\ldots,p-1\}\ \mathcal{P}_i(v) = v_i$;
2. *Si $\langle\langle \mathcal{P}_0(e[\bullet]),\ldots,\mathcal{P}_{p-1}(e[\bullet])\rangle\rangle \stackrel{*}{\rightsquigarrow} \langle\langle v_0,\ldots,v_{p-1}\rangle\rangle$ alors*
   $e[\bullet]/0/\langle 0,\ldots,0\rangle \stackrel{*}{\rightarrow} v/c_g/\langle c_0,\ldots,c_{p-1}\rangle$ tel que $\forall i \in \{0,\ldots,p-1\}\ \mathcal{P}_i(v) = v_i$.

Preuve . Par application de la proposition 1 (voir en section 3.A.5 de l'annexe de ce chapitre). ■

3.2.5 Imbrication de vecteurs parallèles

Nous présentons dans cette section les raisons pour lesquelles nous refusons l'imbrication de vecteurs parallèles en BSML. Considérons le programme BSML suivant :

```
(* bcast: int→α  par→α  par *)
let bcast n vec =
 let tosend=mkpar (fun pid→fun v→fun dst→
                        if pid=n then Some v else None) in
   let recv=put(apply tosend vec) in
     apply noSome (apply recv (replicate n))
```

(bcast n vec) diffuse la valeur provenant du vecteur parallèle vec et qui se trouve au processeur n, à tous les autres processeurs. Le coût BSP d'un appel à cette fonction est : $p + (p-1) \times s \times g + l$ où s est la taille de la valeur de vec se trouvant sur le processeur n. Considérons maintenant l'expression suivante :

```
let example1 = mkpar(fun pid→bcast pid vec)
```

La difficulté est de donner un sens (en terme de parallélisme) à cette expression. Dans le chapitre 2, nous avions indiqué que **mkpar** f s'évaluait en un vecteur parallèle qui contenait la valeur de f i au processeur i. Dans cet exemple, le processeur 0 devrait donc contenir la valeur de l'expression (bcast 0 vec) qui est une valeur parallèle. Notre langage étant basé sur un calcul confluent, les expressions peuvent aussi bien être évaluées en parallèle que de façon séquentielle. Il est ainsi tout à fait possible que l'expression (bcast 0 vec) soit évaluée séquentiellement par le processeur 0. Toutefois, dans ce cas, le temps d'exécution ne serait pas du tout conforme à la formule de coût BSP de notre fonction. Le coût d'évaluation d'une expression serait alors dépendant du contexte, rendant le modèle de coût non compositionnel, ce que nous voulons à tout prix éviter.

Une autre possibilité serait d'évaluer l'expression (bcast 0 vec) possédée par le processeur 0 en parallèle. Il faudrait pour cela une diffusion préalable de cette expression sur tous les autres processeurs. Une fois cette diffusion réalisée l'évaluation de l'expression suivrait la formule de coût BSP. Toutefois, cette phase préalable de diffusion nécessite évidemment des communications et une synchronisation. Ce coût additionnel rend à nouveau le modèle de coût non compositionnel puisque l'expression (bcast 0 vec) évaluée en dehors d'un **mkpar** ne nécessite pas cette phase préalable. De plus, ce choix impliquerait l'usage d'un ordonnanceur (réceptions des communications implicites nécessaires à l'évaluation des vecteurs imbriqués) ce qui rendrait les formules de coûts particulièrement malaisées à écrire.

D'un point de vue purement sémantique, l'imbrication des vecteurs pose directement un autre problème. Prenons, par exemple, la fonction BSML suivante :

```
let a=proj (mkpar (fun i →if i=0 then mkpar(fun i →i) else nc))
 in (a 0)
```

Si cette expression avait été évaluée avec notre sémantique à «petits pas» (et avec trois processeurs), nous aurions le vecteur $\langle 0,1,2\rangle$. Par contre, avec la sémantique distribuée, nous aurions le vecteur distribué $\langle\langle\langle 0\rangle,\langle 0\rangle,\langle 0\rangle\rangle\rangle$. Nous n'obtenons donc pas les mêmes résultats. Cela s'explique par le fait que le vecteur a été «projeté». Les vecteurs représentent les données en chaque processeur. Si un vecteur contient un autre vecteur, alors, naturellement, il lui manque les données des autres processeurs : il lui faudrait alors récupérer ces valeurs, ce qui entraînerait des communications implicites, ce que nous ne voulons absolument pas (compositionalité du modèle de coûts).

Afin d'éviter ces problèmes, l'imbrication de vecteurs parallèles n'est pas autorisée. L'article [R3] a proposé une analyse statique empêchant cette emboîtement. Malheureusement, le langage a depuis fort évolué (nouvelles primitives et extensions), rendant cette analyse obsolète. Un travail futur serait une nouvelle analyse statique assurant la sûreté de notre langage.

3.3 Comparaison avec les anciens calculs

La première sémantique d'un calcul pour BSML a été définie dans l'article [196] : le $BS\lambda$-calcul. Pour éviter l'emboîtement des vecteurs, les auteurs proposaient l'emploi de deux types de termes : les globaux et les locaux. Ce calcul se voulait être la base de la programmation fonctionnelle BSP, comme le λ-calcul l'était pour la programmation séquentielle. Malheureusement, la limitation au niveau de la syntaxe des termes rendait impossible l'écriture de programmes tels que **let** f=(**fun** x→x) **in** f (**mkpar** (**fun** i→(f i))) car la fonction f devait être soit locale (et donc inapplicable au **mkpar**) soit globale (et donc inutilisable dans le **mkpar**). Tous les programmes (notamment BSML) ne pouvaient donc pas être encodés en $BS\lambda$-calcul, ce qui est contraire à l'esprit d'un calcul. De plus, l'opérateur de projection était une «simple» conditionnelle globale.

Le $BS\lambda_p$-calcul [187] a été présenté pour décrire la sémantique distribuée. Comme pour le calcul précédent, deux syntaxes de termes ont été utilisées.

Dans [195], l'auteur propose un calcul avec projection, mais cette fois-ci avec trois syntaxes de termes. Un premier calcul avec substitution explicite a été défini dans [192]. Une seule syntaxe des termes a été utilisée, mais au prix de la possibilité de l'imbrication des vecteurs.

Notre mini-langage a, lui, une seule syntaxe pour les termes, des substitutions explicites et un non-emboîtement dynamique des vecteurs parallèles. Tous les programmes BSML peuvent donc être encodés dans ce mini-langage, mais c'est au prix d'une stratégie d'évaluation. Notons que le test sur l'imbrication des vecteurs n'est pas un problème en soi. Ce test pourra être supprimé si une analyse statique interdit l'évaluation des programmes pouvant la déclencher.

3.A Annexe : preuves des lemmes

3.A.1 Déterminisme de $\triangleright$

Notons qu'avec la propriété 1, l'expression e est sans variables libres. La preuve peut donc se faire par induction sur les arbres de dérivations :

- Les cas des règles 3.2, 3.4 et 3.3 sont évidents car celles-ci sont déterministes.
- Cas de la règle 3.12. Nous avons donc :

$$\frac{s, e \triangleright_{\bowtie} f^1 \qquad s, \langle (f^1\ 0), \ldots, (f^1\ (p-1)) \rangle \triangleright_{\bowtie} \langle v_0, \ldots, v_{p-1} \rangle \quad \text{si } \mathcal{V}_\bullet(f^1) = \mathbf{true}}{s, \mathbf{mkpar}\ e \triangleright_{\bowtie} \langle v_0^1, \ldots, v_{p-1}^1 \rangle}$$

$$\frac{s, e \triangleright_{\bowtie} f^2 \qquad s, \langle (f^2\ 0), \ldots, (f^2\ (p-1)) \rangle \triangleright_{\bowtie} \langle v_0, \ldots, v_{p-1} \rangle \quad \text{si } \mathcal{V}_\bullet(f^2) = \mathbf{true}}{s, \mathbf{mkpar}\ e \triangleright_{\bowtie} \langle v_0^2, \ldots, v_{p-1}^2 \rangle}$$

  Par hypothèse d'induction, $f^1 = f^2$ et les deux dérivations sont donc équivalentes ($\forall i\ v_i^1 = v_i^2$).

- Les preuves des autres cas sont similaires (par hypothèses d'inductions).

3.A.2 Confluence forte de $\rightharpoonup$

Lemme 4 (*Déterminisme des règles fonctionnelles*)

Soit e une expression. Alors :

1. *Si $e \xrightarrow{\varepsilon} e^1, c^1$ et $e \xrightarrow{\varepsilon} e^2, c^2$ alors $e^1 = e^2$ et $c^1 = c^2$;*
2. *Si $e \xrightarrow{\delta} e^1, c^1$ et $e \xrightarrow{\delta} e^2, c^2$ alors $e^1 = e^2$ et $c^1 = c^2$.*

Preuve. Par cas sur les règles. ■

Lemme 5 (*Déterminisme des règles globales*)

Soit e une expression. Alors :

1. *Si $e/c_g/\langle \mathbf{c} \rangle \xrightarrow[\bowtie]{\varepsilon} e^1/c_g \oplus c^1/\langle \mathbf{c} \rangle$ et $e/c_g/\langle \mathbf{c} \rangle \xrightarrow[\bowtie]{\varepsilon} e^2/c_g \oplus c^2/\langle \mathbf{c} \rangle$ alors $e^1 = e^2$ et $c^1 = c^2$;*
2. *Si $e/c_g/\langle \mathbf{c} \rangle \xrightarrow[\bowtie]{\delta} e^1/c_g \oplus c^1/\langle \mathbf{c} \rangle$ et $e/c_g/\langle \mathbf{c} \rangle \xrightarrow[\bowtie]{\delta} e^2/c_g \oplus c^2/\langle \mathbf{c} \rangle$ alors $e^1 = e^2$ et $c^1 = c^2$.*

Preuve. Par application du lemme 4 et par cas sur les règles des primitives et des opérations globales. ■

Lemme 6 (*Déterminisme des règles*)

Soit e une expression. Alors :

1. *Si $e/c_g/\langle c_0, \ldots, c_{p-1} \rangle \xrightarrow{\bowtie} e^1/c_g^1/\langle c_0^1, \ldots, c_{p-1}^1 \rangle$ et $e/c_g/\langle c_0, \ldots, c_{p-1} \rangle \xrightarrow{\bowtie} e^2/c_g^2/\langle c_0^2, \ldots, c_{p-1}^2 \rangle$ alors $e^1 = e^2$, $c_g^1 = c_g^2$ et $\forall i\ c_i^1 = c_i^2$;*
2. *Si $e \xrightarrow{i} e^1, c^1$ et $e \xrightarrow{i} e^2, c^2$ alors $e^1 = e^2$ et $c^1 = c^2$.*

Preuve. Par application du lemme 5 pour (1) et du lemme 4 pour (2). ■

Lemme 7 (*Déterminisme des contextes globaux*)

Soit e une expression. Si $e = \Gamma^1[e^1]$ et $e = \Gamma^2[e^2]$ alors $\Gamma^1 = \Gamma^2$ et $e^1 = e^2$.

Lemme 8 (*Confluence des contextes locaux*)

Soit e une expression. Alors :

1. $\forall i$, *si* $e = \Delta^i[e^1]$ *et* $e = \Delta^i[e^2]$ *alors* $e^1 = e^2$;
2. $\forall i, j\ i \neq j$ *si* $e = \Delta^i[e^1]$ *et* $e = \Delta^j[e^2]$ *alors* $e = \Gamma[\langle \cdots, \Gamma_{l^1}[e^1], \cdots, \Gamma_{l^1}[e^2], \cdots \rangle]$;
3. *Si* $e = \Gamma_{l^1}[e^1]$ *et* $e = \Gamma_{l^2}[e^2]$ *alors* $\Gamma_{l^1} = \Gamma_{l^2}$ *et* $e^1 = e^2$.

Lemme 9 (*Unicité du choix d'un type de contexte*)

Soit e une expression. Si $e = \Gamma[e']$ alors $\nexists \Delta^i$ tel que $e = \Delta^i[e'']$.

Preuve. Par construction de nos contextes. En effet, ceux-ci ne permettent l'application d'une règle :

- Globale que dans une unique sous-expression ;
- Locale que dans un unique vecteur parallèle mais possiblement dans deux composantes différentes de ce vecteur.

Nous savons aussi, par construction, que les contextes Γ_l sont déterministes (dans le sens qu'une seule sous-expression à la fois peut être réduite). ■

Lemme 10 (*Confluence forte*)

Soit une expression e.

Si	$e/c_g/\langle c_0, \cdots, c_{p-1}\rangle \rightharpoonup e^1/c_g^1/\langle c_0^1, \cdots, c_{p-1}^1\rangle$
et	$e/c_g/\langle c_0, \cdots, c_{p-1}\rangle \rightharpoonup e^2/c_g^2/\langle c_0^2, \cdots, c_{p-1}^2\rangle$
alors il existe	*une expression* e_3 *et des coûts* c_g^3 *et* $\langle c_0^3, \cdots, c_{p-1}^3\rangle$
tels que	$e^1/c_g^1/\langle c_0^1, \cdots, c_{p-1}^1\rangle \rightharpoonup e^3/c_g^3/\langle c_0^3, \cdots, c_{p-1}^3\rangle$
et	$e^2/c_g^2/\langle c_0^2, \cdots, c_{p-1}^2\rangle \rightharpoonup e^3/c_g^3/\langle c_0^3, \cdots, c_{p-1}^3\rangle$.

Preuve. Par le lemme 9, nous avons deux types de réductions bien distincts.

Si $\rightharpoonup$ est une réduction globale, alors par le lemme 7 il n'existe qu'un contexte global et par le lemme 6, la réduction est déterministe.

Si $\rightharpoonup$ est une réduction locale alors nous avons $e = \Delta^i[e^i]$ et $e = \Delta^j[e^j]$. Nous avons alors deux cas :

1. Si $i = j$ alors par le lemme 8.1 il n'existe qu'un contexte et par le lemme 6, la réduction est déterministe
2. Si $i \neq j$ alors par le lemme 8.2, il n'existe qu'un seul vecteur parallèle à réduire. Dans ce cas, nous avons $e = \Gamma[\langle \ldots, \Gamma_{l^i}[e^i], \ldots, \Gamma_{l^j}[e^j], \ldots\rangle]$. Par les lemmes 8.3 et 6, les deux réductions sont déterministes. Donc par j, $e \rightharpoonup \Gamma[\langle \ldots, \Gamma_{l^i}[e^i], \ldots, \Gamma_{l^j}[e'^j], \ldots\rangle]$ et par i, $e \rightharpoonup \Gamma[\langle \ldots, \Gamma_{l^i}[e'^i], \ldots, \Gamma_{l^j}[e^j], \ldots\rangle]$. Il est alors facile de constater que l'ordre dans lequel ces deux réductions sont appliquées, indiffère le résultat final : les réductions interviennent dans deux composantes différentes d'un même vecteur. Les règles peuvent donc s'entrelacer. Nous pouvons donc avoir $\Gamma[\langle \ldots, \Gamma_{l^i}[e'^i], \ldots, \Gamma_{l^j}[e'^j], \ldots\rangle]$.

Les réductions des deux types sont donc fortement confluentes. ■

3.A.3 Équivalence entre $\triangleright$ et $\rightharpoonup$

Lemme 11 (*De $\triangleright$ à $\rightharpoonup$*)

Soit une expression e et s un environnement (une substitution). Si $s, e \triangleright v$ alors $e[s] \overset{*}{\rightharpoonup} v$.

Preuve. Par $\triangleright$, nous entendons $\triangleright_g$ ou $\triangleright_i$. En effet, si la dérivation est de la forme $s, e \triangleright_i v$ alors cette évaluation est dans un vecteur parallèle et donc nous construisons la preuve dans un contexte Δ^i et ceux $\forall i$. Nous ignorons les coûts de la sémantique de $\rightharpoonup$ afin de simplifier la lecture de la preuve (les coûts n'apparaissant pas dans la sémantique $\triangleright$). La preuve se fait par induction sur la dérivation $s, e \triangleright v$.

- Cas règle (3.1). Nous avons :
$$\frac{s, \overline{n} \triangleright v}{v' \circ s, \overline{n+1} \triangleright v}$$
Avec l'hypothèse de récurrence (HR) nous construisons $\overline{n+1}[v' \circ s] \rightharpoonup \overline{n}[s] \overset{*}{\rightharpoonup} v$. D'où le résultat.
- Cas règle (3.2). Nous avons :
$$\frac{}{v \circ s, \overline{1} \triangleright v}$$
et nous avons $\overline{1}[v \circ s] \rightharpoonup v[\bullet] \rightharpoonup v$. D'où le résultat.

- Cas règle (3.4). Nous avons :

$$\frac{\text{si } v \neq \langle \cdots \rangle \text{ et } v \neq (v_0, v_1)}{s, v \rhd v}$$

et nous avons $v[s] \rightharpoonup v$ si $v \neq \langle \cdots \rangle$ et $v \neq (v_0, v_1)$. D'où le résultat.

- Cas règle (3.3). Nous avons :

$$\frac{}{s, \lambda.e \rhd \overline{(\lambda.e)[s]}}$$

et nous avons $\lambda.e[s] \rightharpoonup \overline{(\lambda.e)[s]}$. D'où le résultat.

- Cas règle (3.5). Nous avons :

$$\frac{s, e_1 \rhd v_1 \qquad s, e_2 \rhd v_2}{s, (e_1, e_2) \rhd (v_1, v_2)}$$

Avec les HR et les contextes $([], e_2[s])$ et $(v_1, [])$, nous construisons :

$$(e_1, e_2)[s] \rightharpoonup (e_1[s], e_2[s]) \xrightarrow{*} (v_1, e_2[s]) \xrightarrow{*} (v_1, v_2)$$

D'où le résultat.

- Cas règle (3.6). Nous avons :

$$\frac{s, e_1 \rhd \overline{(\lambda.e)[s']} \qquad s, e_2 \rhd v' \qquad v' \circ s', e \rhd v}{s, (e_1\ e_2) \rhd v}$$

Avec les HR et les contextes $([]\ e_2[s])$ et $(v_1\ [])$, nous construisons :

$$(e_1\ e_2)[s] \rightharpoonup (e_1[s]\ e_2[s]) \xrightarrow{*} (\overline{(\lambda.e)[s']}\ e_2[s]) \xrightarrow{*} (\overline{(\lambda.e)[s']}\ v_2) \rightharpoonup e[v_2 \circ s'] \xrightarrow{*} v$$

D'où le résultat. Les règles (3.7) sont prouvées de manière similaire avec le fait que les axiomes $\xrightarrow{\delta}$ et $\xrightarrow[\bowtie]{\delta}$ correspondent aux définitions de $\rhd_l$ et de $\rhd_g$.

- Cas règle (3.9). Nous avons :

$$\frac{(\mu.e) \circ s, e \rhd v}{s, \mu.e \rhd v}$$

Avec l'HR, nous construisons $(\mu e)[s] \rightharpoonup e[\mu e \circ s] \xrightarrow{*} v$. D'où le résultat.

- Cas règle (3.10). Nous avons :

$$\frac{s, e_1 \rhd \mathbf{true} \qquad s, e_2 \rhd v}{s, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd v}$$

Avec les HR et les contextes $\mathbf{if}\ []\ \mathbf{then}\ e_2[s]\ \mathbf{else}\ e_3[s]$ et $\mathbf{if\ true\ then}\ []\ \mathbf{else}\ e_3[s]$, nous construisons $(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)[s] \rightharpoonup \mathbf{if}\ e_1[s]\ \mathbf{then}\ e_2[s]\ \mathbf{else}\ e_3[s] \xrightarrow{*} \mathbf{if\ true\ then}\ e_2[s]\ \mathbf{else}\ e_3[s] \rightharpoonup e_2[s] \xrightarrow{*} v$. D'où le résultat. Le cas de la règle (3.11) est similaire.

- Cas règle (3.14). Nous avons :

$$\frac{s, e_1 \rhd_\bowtie \langle f_0, \ldots, f_{p-1} \rangle \quad s, e_2 \rhd_\bowtie \langle v_0, \ldots, v_{p-1} \rangle \quad \bullet, \langle (f_0\ v_0), \ldots, (f_{p-1}\ v_{p-1}) \rangle \rhd_\bowtie \langle v'_0, \ldots, v'_{p-1} \rangle}{s, \mathbf{apply}\ e_1\ e_2 \rhd_\bowtie \langle v'_0, \ldots, v'_{p-1} \rangle}$$

Avec les HR et les contextes $(\mathbf{apply}\ []\ e_2[s])$ et $(\mathbf{apply}\ \langle f_0, \ldots, f_{p-1} \rangle\ [])$, nous construisons $(\mathbf{apply}\ e_1\ e_2)[s] \rightharpoonup \mathbf{apply}\ e_1[s]\ e_2[s] \xrightarrow{*} \mathbf{apply}\ \langle f_0, \ldots, f_{p-1} \rangle\ e_2[s] \xrightarrow{*} \mathbf{apply}\ \langle f_0, \ldots, f_{p-1} \rangle\ \langle v_0, \ldots, v_{p-1} \rangle \rightharpoonup \langle (f_0\ v_0), \ldots, (f_{p-1}\ v_{p-1}) \rangle [\bullet] \xrightarrow{*} \langle v'_0, \ldots, v'_{p-1} \rangle$. D'où le résultat.

- Cas règle (3.16). Nous avons :

$$\frac{s, e \rhd_\bowtie \langle v_0, \ldots, v_{p-1} \rangle \quad \bullet, (\mathbf{delpar}\ (\mathbf{put}\ \langle \lambda.v_0, \ldots, \lambda.v_{p-1} \rangle))[\bullet] \rhd f}{s, \mathbf{proj}\ e \rhd_\bowtie f}$$

Avec les HR et le contexte $\mathbf{proj}\ []$, nous construisons :

$$(\mathbf{proj}\ e)[s] \rightharpoonup \mathbf{proj}\ e[s] \xrightarrow{*} \mathbf{proj}\ \langle v_0, \ldots, v_{p-1} \rangle \rightharpoonup (\mathbf{delpar}\ (\mathbf{put}\ \langle \lambda.v_0, \ldots, \lambda.v_{p-1} \rangle))[\bullet] \xrightarrow{*} f$$

D'où le résultat.

- Cas règle (3.15). Nous avons :

$$\frac{s, e \rhd_\bowtie \langle f_0, \ldots, f_{p-1} \rangle \quad \bullet, (\mathbf{apply}\ (\mathbf{mkpar}(\lambda.\mathbf{F}))\ (\mathbf{send}\ \langle (\mathbf{init}\ f_0), \ldots, (\mathbf{init}\ f_{p-1}) \rangle))[\bullet] \rhd_\bowtie \langle f'_0, \ldots, f'_{p-1} \rangle}{s, \mathbf{put}\ e \rhd_\bowtie \langle f'_0, \ldots, f'_{p-1} \rangle}$$

Avec les HR et le contexte $\mathbf{put}\ []$, nous construisons
$(\mathbf{put}\ e)[s] \rightharpoonup \mathbf{put}\ e[s] \xrightarrow{*} \mathbf{put}\ \langle f_0, \ldots, f_{p-1} \rangle \rightharpoonup$
$(\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{F}))\ (\mathbf{send}\ \langle (\mathbf{init}\ f_0), \ldots, (\mathbf{init}\ f_{p-1}) \rangle))[\bullet] \xrightarrow{*} \langle f'_0, \ldots, f'_{p-1} \rangle$. D'où le résultat.

- Cas règle (3.13). Nous avons :

$$\frac{\forall i \in \{0,\ldots,p-1\} \quad \bullet, e_i \rhd_l v_i}{\bullet, \langle e_0,\ldots,e_{p-1}\rangle \rhd_{\bowtie} \langle v_0,\ldots,v_{p-1}\rangle}$$

  Avec les HR et les contextes $\forall i\ \langle \ldots, \overbrace{[]}^{i}, \ldots\rangle$, nous construisons :

$$\langle e_0,\ldots,e_{p-1}\rangle[\bullet] \rightharpoonup \langle e_0[\bullet],\ldots,e_{p-1}[\bullet]\rangle \stackrel{*}{\rightharpoonup} \langle v_0,\ldots,v_{p-1}\rangle$$

  D'où le résultat.
- Cas règle (3.8). Nous avons :

$$\frac{\forall i \in \{0,\ldots,p-1\} \quad s, e_i \rhd v_i}{s, [e_0,\ldots,e_{p-1}] \rhd [v_0,\ldots,v_{p-1}]}$$

  Avec les HR et les contextes $\forall i\ [\ldots, \overbrace{[]}^{i}, \ldots]$, nous construisons :

$$[e_0,\ldots,e_{p-1}][\bullet] \rightharpoonup [e_0[\bullet],\ldots,e_{p-1}[\bullet]] \stackrel{*}{\rightharpoonup} [v_0,\ldots,v_{p-1}]$$

  D'où le résultat.
- Cas règle (3.12). Nous avons :

$$\frac{s, e \rhd_{\bowtie} f \qquad \bullet, \langle (f\ 0),\ldots,(f\ (p-1))\rangle[\bullet] \rhd_{\bowtie} \langle v_0,\ldots,v_{p-1}\rangle \quad \text{si } \mathcal{V}_\bullet(f) = \mathbf{true}}{s, \mathbf{mkpar}\ e \rhd_{\bowtie} \langle v_0,\ldots,v_{p-1}\rangle}$$

  Avec les HR et le contexte $\mathbf{mkpar}\ []$, nous construisons :

$$(\mathbf{mkpar}\ e)[s] \rightharpoonup \mathbf{mkpar}\ e[s] \stackrel{*}{\rightharpoonup} \mathbf{mkpar}\ f \rightharpoonup \langle (f\ 0),\ldots,(f\ (p-1))\rangle[\bullet] \stackrel{*}{\rightharpoonup} \langle v_0,\ldots,v_{p-1}\rangle$$

  D'où le résultat.
- Les cas de $\rhd_i$ sont similaires dans un contexte Δ^i. ■

Lemme 12

Soient une valeur v et une substitution s, alors $s, v \rhd v$.

Preuve. Immédiate par récurrence structurelle sur v. ■

Lemme 13 (*De $\rightharpoonup$ à $\rhd$*)

Soit e une expression et s une substitution. Alors :

- *Si $e \rightharpoonup e'[s]$ et si $s, e' \rhd v$ alors $\exists s'$ tel que $s', e \rhd v$;*
- *Si $e[s] \rightharpoonup (\mathbf{CSTR}\ e_1[s] \cdots e_n[s])$ (avec $\mathbf{CSTR}$=application, pair ou ifthenelse etc.) et si $\forall i \in \{1,\cdots,n\}\ s, e_i \rhd v_i$ alors $s, e \rhd (\mathbf{CSTR}\ v_0 \cdots v_i)$;*
- *Si $(\mathbf{OP}\ v') \rightharpoonup e$ et si $s, e \rhd v$ alors $s, (\mathbf{OP}\ v') \rhd v$.*

Preuve. Nous montrons d'abord le résultat pour une réduction de tête, en examinant les axiomes de réduction :

- Si $(\overline{\lambda.e'[s'']}\ v') \rightharpoonup e'[v' \circ s'']$, alors nous construisons la dérivation suivante :

$$\frac{s', \overline{(\lambda.e')[s'']} \rhd \overline{(\lambda.e')[s'']} \qquad s', v' \rhd v' \qquad v' \circ s'', e' \rhd v}{s', (\overline{(\lambda.e')[s'']}\ v') \rhd v}$$

  La première prémisse provient de la règle 3.4, la deuxième prémisse du lemme 12 et la dernière de l'hypothèse. Nous avons bien le résultat $\forall s'$ et avec $s = v' \circ s''$.
  Si $(\mathbf{mkpar}\ f) \rightharpoonup \langle (f\ 0),\ldots,(f\ (p-1))\rangle[\bullet]$ si $\mathcal{V}_\bullet(f) = \mathbf{true}$, alors nous construisons la dérivation suivante :

$$\frac{s', f \rhd_{\bowtie} f \qquad \bullet, \langle (f\ 0),\ldots,(f\ (p-1))\rangle[\bullet] \rhd_{\bowtie} \langle v_0,\ldots,v_{p-1}\rangle \quad \text{si } \mathcal{V}_\bullet(f) = \mathbf{true}}{s', \mathbf{mkpar}\ f \rhd \langle v_0,\ldots,v_{p-1}\rangle}$$

  Nous avons la première prémisse par le lemme 12 et la seconde prémisse par hypothèse. D'où le résultat avec $s = \bullet$ et $\forall s'$. Les autres cas sont similaires.

- Si $(e_1, e_2)[s] \rightharpoonup (e_1[s], e_2[s])$, alors nous construisons la dérivation suivante :

$$\frac{s, e_1 \rhd v_1 \qquad s, e_2 \rhd v_2}{s, (e_1, e_2) \rhd (v_1, v_2)}$$

  Les deux prémisses provenant de l'hypothèse. D'où le résultat. Les autres cas sont similaires.
- Si $(\mathbf{fst}\ (v_1, v_2)) \rightharpoonup v_1$, alors nous construisons la dérivation suivante :

$$\frac{s, \mathbf{fst} \rhd \mathbf{fst} \qquad s, (v_1, v_2) \rhd (v_1, v_2) \qquad (\mathbf{fst}\ (v_1, v_2)) = v_1}{s, (\mathbf{fst}\ (v_1, v_2)) \rhd v_1}$$

  Les prémisses proviennent du lemme 12 et de la définition de la règle de l'opérateur $\mathbf{fst}$. Les autres cas sont similaires.

Pour finir la preuve, il faut montrer que cela passe bien au contexte. Nous faisons la preuve par récurrence structurelle sur les contextes Γ et Δ^i :

- Le cas de base $\Gamma = []$ est immédiat avec le résultat précédent sur les réductions de tête.
- Cas d'un contexte global, le cas $\Gamma = \Gamma'\ e_2$. Supposons $s, \Gamma[e'] \rhd v$. Comme $\Gamma[e'] = \Gamma'[e']\ e_2$, nous avons, par exemple, une dérivation de la forme :

$$\frac{s, \Gamma'[e'] \rhd \overline{\lambda.e_3[s_3]} \quad s, e_2 \rhd v_2 \quad v_2 \circ s_3, e_3 \rhd v}{s, \Gamma'[e']\ e_2 \rhd v}$$

  Par hypothèse de récurrence, nous avons $s, \Gamma'[e] \rhd \overline{\lambda.e_3[s_3]}$. Nous pouvons donc construire la dérivation :

$$\frac{s, \Gamma'[e] \rhd \overline{\lambda.e_3[s_3]} \quad s, e_2 \rhd v_2 \quad v_2 \circ s_3, e_3 \rhd v}{s, \Gamma'[e]\ e_2 \rhd v}$$

  qui conclut $s, \Gamma[e] \rhd v$. La preuve pour les autres formes de contextes est similaire.
- Cas d'un contexte local, le cas Δ^i. Supposons que $s, \Delta^i[e'] \rhd v$. Nous avons une dérivation de la forme

$$\frac{\bullet, \Gamma_l[e'] \rhd v_i \quad \forall j\ j \neq i\ \bullet, e_j \rhd v_j}{s, \langle \cdots, \overbrace{\Gamma_l[e']}^{i}, \cdots \rangle \rhd v}$$

  avec $v = \langle \cdots, v_i, \cdots \rangle$. Par hypothèse de récurrence, nous avons $\bullet, \Gamma_l[e] \rhd v_i$. Nous pouvons donc construire la dérivation :

$$\frac{\bullet, \Gamma_l[e] \rhd v_i \quad \forall j\ j \neq i\ \bullet, e_j \rhd v_j}{s, \langle \cdots, \overbrace{\Gamma_l[e]}^{i}, \cdots \rangle \rhd v}$$

  d'où le résultat. ■

3.A.4 Confluence forte de $\rightsquigarrow$

Lemme 14 (*Déterminisme des règles fonctionnelles*)

Soit e une expression. Alors :

1. *Si $e \overset{\varepsilon}{\rightsquigarrow} e^1$ et $e \overset{\varepsilon}{\rightsquigarrow} e^2$ alors $e^1 = e^2$;*
2. *Si $e \overset{\delta}{\rightsquigarrow} e^1$ et $e \overset{\delta}{\rightsquigarrow} e^2$ alors $e^1 = e^2$.*

Preuve. Par cas sur les règles. ■

Lemme 15 (*Déterminisme des règles globales*)

Soit e une expression. Alors :

1. *Si $e \underset{\bowtie_i}{\overset{\varepsilon}{\rightsquigarrow}} e^1$ et $e \underset{\bowtie_i}{\overset{\varepsilon}{\rightsquigarrow}} e^2$ alors $e^1 = e^2$;*
2. *Si $e \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} e^1$ et $e \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} e^2$ alors $e^1 = e^2$.*

Preuve. Par application du lemme 14 et par cas sur les règles des primitives et des opérations globales. ■

Lemme 16 (*Déterminisme des règles*)

Soit e une expression. Alors :

1. *Si $e \overset{\bowtie_i}{\rightsquigarrow} e^1$ et $e \overset{\bowtie_i}{\rightsquigarrow} e^2$ alors $e^1 = e^2$;*
2. *Si $e \overset{i}{\rightsquigarrow} e^1$ et $e \overset{i}{\rightsquigarrow} e^2$ alors $e^1 = e^2$.*

Preuve. Par application du lemme 15 pour (1) et du lemme 14 pour (2). ■

Lemme 17 (*Déterminisme des contextes*)

Soit e une expression. Alors :

1. *Si $e = \Gamma^1[e^1]$ et $e = \Gamma^2[e^2]$ alors $\Gamma^1 = \Gamma^2$ et $e^1 = e^2$;*
2. *Si $e = \Delta^1[e^1]$ et $e = \Delta^2[e^2]$ alors $\Delta^1 = \Delta^2$ et $e^1 = e^2$.*

Lemme 18 (*Unicité des contextes*)

Soit e une expression. Si $e = \Gamma[e']$ alors $\nexists \Delta$ tel que $e = \Delta[e'']$.

Preuve. Par construction de nos contextes. En effet, ceux-ci ne permettent l'application d'une règle que dans une unique sous-expression (en global comme en local). ■

Lemme 19 (*Confluence forte*)

Soit $\langle\langle e_0, \ldots, e_{p-1}\rangle\rangle$ un terme distribué.

Si	$\langle\langle e_0, \ldots, e_{p-1}\rangle\rangle \rightsquigarrow \langle\langle e_0^1, \ldots, e_{p-1}^1\rangle\rangle$
et	$\langle\langle e_0, \ldots, e_{p-1}\rangle\rangle \rightsquigarrow \langle\langle e_0^2, \ldots, e_{p-1}^2\rangle\rangle$
alors il existe	$\langle\langle e_0^3, \ldots, e_{p-1}^3\rangle\rangle$
tel que	$\langle\langle e_0^1, \ldots, e_{p-1}^1\rangle\rangle \rightsquigarrow \langle\langle e_0^3, \ldots, e_{p-1}^3\rangle\rangle$
et	$\langle\langle e_0^2, \ldots, e_{p-1}^2\rangle\rangle \rightsquigarrow \langle\langle e_0^3, \ldots, e_{p-1}^3\rangle\rangle$.

Preuve. Par le lemme 18, nous avons trois types de réductions distincts. Les deux premières réductions se font respectivement au processeur i et j. La dernière est la réduction d'un $\mathbf{send}$. Nous avons alors trois cas :

1. Si $i = j$ alors $\rightsquigarrow$ est une réduction globale ou locale (les deux au processeur i). Par les lemmes 17.1 et 17.2, il n'existe alors qu'un contexte et par le lemme 16, la réduction est déterministe ;
2. Si $i \neq j$ il est alors facile (comme précédemment) de constater que les règles peuvent s'entrelacer : les réductions interviennent dans deux composantes différentes d'un même vecteur distribué ;
3. Si $\rightsquigarrow$ est une réduction d'un $\mathbf{send}$ alors $\forall i\ e_i = \Gamma[\mathbf{send}\ \langle[v_0^i, \ldots, v_{p-1}^i]\rangle]$ et $e_i' = \Gamma[\langle[v_i^0, \ldots, v_i^{p-1}]\rangle]$. La règle est déterministe.

Les réductions des trois types sont donc fortement confluentes. ■

3.A.5 Équivalence entre $\rightsquigarrow$ et $\rightharpoonup$

Dans cette section, nous faisons fi des coûts de la sémantique de $\rightharpoonup$ afin de simplifier la lecture de la preuve (les coûts n'apparaissant pas dans la sémantique $\rightsquigarrow$).

Lemme 20 (*La projection d'une valeur est une valeur projetée*)

Soient une valeur v et un indice $i \in \{0, \cdots p-1\}$, alors $\mathcal{P}_i(v) = v_i$ (v_i est une valeur projetée).

Preuve. Immédiate par récurrence structurelle sur v. ■

Lemme 21 (*Test d'une valeur*)

Soit v une valeur. Si $\mathcal{V}_\bullet(v) = \mathbf{true}$ alors $\forall i \in \{0, \cdots p-1\}\ \mathcal{V}_\bullet(\mathcal{P}_i(v)) = \mathbf{true}$.

Preuve. Immédiate par récurrence structurelle sur v. ■

Nous notons $\overset{+}{\rightsquigarrow}$ la fermeture transitive de $\rightsquigarrow$.

Lemme 22 (*Un $\rightharpoonup$ vaut plusieurs $\rightsquigarrow$*)

Soit e une expression. Si $e \rightharpoonup e'$ alors $\langle\!\langle \mathcal{P}_0(e), \ldots, \mathcal{P}_{p-1}(e) \rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{P}_0(e'), \ldots, \mathcal{P}_{p-1}(e') \rangle\!\rangle$.

Preuve. Nous montrons d'abord le résultat pour une réduction de tête, en examinant les axiomes de réduction :

- Si $(\overline{\lambda.e[s]}\ v) \rightharpoonup e[v \circ s]$ alors

$$\begin{array}{rl} & \langle\!\langle \mathcal{P}_0((\overline{\lambda.e[s]}\ v)), \ldots, \mathcal{P}_{p-1}((\overline{\lambda.e[s]}\ v)) \rangle\!\rangle \\ = & \langle\!\langle (\overline{\lambda.\mathcal{P}_0(e)[\mathcal{P}_0(s)]}\ \mathcal{P}_0(v)), \ldots, (\overline{\lambda.\mathcal{P}_{p-1}(e)[\mathcal{P}_{p-1}(s)]}\ \mathcal{P}_{p-1}(v)) \rangle\!\rangle \\ = & \langle\!\langle (\overline{\lambda.e_0[s_0]}\ v_0), \ldots, (\overline{\lambda.e_{p-1}[s_{p-1}]}\ v_{p-1}) \rangle\!\rangle \\ \overset{*}{\rightsquigarrow} & \langle\!\langle e_0[v_0 \circ s_0], \ldots, e_{p-1}[v_{p-1} \circ s_{p-1}] \rangle\!\rangle \qquad \text{par } p \text{ applications} \\ = & \langle\!\langle \mathcal{P}_0(e)[\mathcal{P}_0(v) \circ \mathcal{P}_0(s)], \ldots, \mathcal{P}_{p-1}(e)[\mathcal{P}_{p-1}(v) \circ \mathcal{P}_{p-1}(s)] \rangle\!\rangle \\ = & \langle\!\langle \mathcal{P}_0(e[v \circ s]), \ldots, \mathcal{P}_{p-1}(e[v \circ s]) \rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu.
- Si $(e^1\ e^2)[s] \rightharpoonup (e^1[s]\ e^2[s])$ alors

$$\begin{array}{rl} & \langle\!\langle \mathcal{P}_0((e^1\ e^2)[s]), \ldots, \mathcal{P}_{p-1}((e^1\ e^2)[s]) \rangle\!\rangle \\ = & \langle\!\langle (\mathcal{P}_0(e^1)\ \mathcal{P}_0(e^2))[\mathcal{P}_0(s)], \ldots, (\mathcal{P}_{p-1}(e^1)\ \mathcal{P}_{p-1}(e^2))[\mathcal{P}_{p-1}(s)] \rangle\!\rangle \qquad \text{alors par } p \text{ applications} \\ \overset{*}{\rightsquigarrow} & \langle\!\langle (\mathcal{P}_0(e^1)[\mathcal{P}_0(s)]\ \mathcal{P}_0(e^2)[\mathcal{P}_0(s)]), \ldots, (\mathcal{P}_{p-1}(e^1)[\mathcal{P}_{p-1}(s)]\ \mathcal{P}_{p-1}(e^2)[\mathcal{P}_{p-1}(s)]) \rangle\!\rangle \\ = & \langle\!\langle \mathcal{P}_0((e^1[s]\ e^2[s])), \ldots, \mathcal{P}_{p-1}((e^1[s]\ e^2[s])) \rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu.
- Si $(\mathbf{mkpar}\ f) \rightharpoonup \langle (f\ 0), \ldots, (f\ (p-1)) \rangle$ alors (en utilisant p fois le lemme 21 sur f) :

$$\begin{array}{rl} & \langle\!\langle \mathcal{P}_0(\mathbf{mkpar}\ f), \ldots, \mathcal{P}_{p-1}(\mathbf{mkpar}\ f) \rangle\!\rangle \\ = & \langle\!\langle \mathbf{mkpar}\ \mathcal{P}_0(f), \ldots, \mathbf{mkpar}\ \mathcal{P}_{p-1}(f) \rangle\!\rangle \qquad \text{alors par } p \text{ applications} \\ \overset{*}{\rightsquigarrow} & \langle\!\langle \langle (\mathcal{P}_0(f)\ 0) \rangle, \ldots, \langle (\mathcal{P}_{p-1}(f)\ (p-1)) \rangle \rangle\!\rangle \\ = & \langle\!\langle \mathcal{P}_0(\langle (f\ 0) \rangle), \ldots, \mathcal{P}_{p-1}(\langle (f\ (p-1)) \rangle) \rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu.
- Tous les autres cas sont similaires.

Pour finir la preuve, il faut montrer que tout cela passe bien au contexte, c'est-à-dire :

- Si $\Gamma[e] \rightharpoonup \Gamma[e']$ alors $\langle\!\langle \mathcal{P}_0(\Gamma[e]), \ldots, \mathcal{P}_{p-1}(\Gamma[e]) \rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{P}_0(\Gamma[e']), \ldots, \mathcal{P}_{p-1}(\Gamma[e']) \rangle\!\rangle$;
- Si $\Delta^i[e] \rightharpoonup \Delta^i[e']$ alors $\langle\!\langle \mathcal{P}_0(\Delta[e]), \ldots, \mathcal{P}_{p-1}(\Delta[e]) \rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{P}_0(\Delta[e']), \ldots, \mathcal{P}_{p-1}(\Delta[e']) \rangle\!\rangle$.

Cela se fait par récurrence structurelle sur les contextes Γ et Δ^i :

- Le cas de base $\Gamma = []$ est immédiat avec le résultat précédent sur les réductions de tête.
- Cas d'un contexte global, le cas $\Gamma = (\Gamma'\ e_2)$. Nous avons alors $(\Gamma'[e]\ e^2) \rightharpoonup (\Gamma'[e']\ e^2)$ et :

$$\begin{array}{rl} & \langle\!\langle \mathcal{P}_0(\Gamma'[e]\ e^2), \ldots, \mathcal{P}_{p-1}(\Gamma'[e]\ e^2) \rangle\!\rangle \\ = & \langle\!\langle (\mathcal{P}_0(\Gamma'[e])\ \mathcal{P}_0(e^2)), \ldots, (\mathcal{P}_{p-1}(\Gamma'[e])\ \mathcal{P}_{p-1}(e^2)) \rangle\!\rangle \\ \overset{+}{\rightsquigarrow} & \langle\!\langle (\mathcal{P}_0(\Gamma'[e'])\ \mathcal{P}_0(e^2)), \ldots, (\mathcal{P}_{p-1}(\Gamma'[e'])\ \mathcal{P}_{p-1}(e^2)) \rangle\!\rangle \quad \text{par hypothèse de récurrence} \\ = & \langle\!\langle \mathcal{P}_0(\Gamma'[e']\ e^2), \ldots, \mathcal{P}_{p-1}(\Gamma'[e']\ e^2) \rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu. Les autres cas sont similaires.
- Cas d'un contexte local, le cas Δ^i. Par le lemme 8.2, nous avons $\Gamma[\langle \ldots, \overbrace{\Gamma^l[e]}^{i}, \ldots \rangle] \rightharpoonup \Gamma[\langle \ldots, \overbrace{\Gamma^l[e']}^{i}, \ldots \rangle]$. Donc nous avons :

$$\begin{array}{rl} & \langle\!\langle \ldots, \mathcal{P}_i(\Gamma[\langle \Gamma^l[e] \rangle]), \ldots \rangle\!\rangle \\ = & \langle\!\langle \ldots, (\Gamma[\langle \mathcal{P}_i(\Gamma^l[e]) \rangle]), \ldots \rangle\!\rangle \\ \rightsquigarrow & \langle\!\langle \ldots, (\Gamma[\langle \mathcal{P}_i(\Gamma^l[e']) \rangle]), \ldots \rangle\!\rangle \text{ par hypothèse} \\ = & \langle\!\langle \ldots, \mathcal{P}_i(\Gamma[\langle \Gamma^l[e'] \rangle]), \ldots \rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu. ■

Lemme 23 (*Les valeurs projetées proviennent d'une valeur*)

Soit e une expression. Si $\forall i \in \{0, \ldots, p-1\}\ \mathcal{P}_i(e) = v_i$ alors $e = v$ et v est une valeur.

Preuve. Immédiate par récurrence structurelle sur e. ■

Lemme 24 (*Les expressions projetées ne proviennent pas d'une valeur*)

Soit e une expression. Si $\forall i \in \{0,\dots,p-1\}\ \mathcal{P}_i(e) \neq v_i$ alors $e \neq v$ (e n'est pas une valeur).

Preuve. Par contraposition immédiate du lemme précédent. ■

Lemme 25 (*Test d'une valeur projetée*)

Soit v une valeur. Si $\forall i \in \{0,\dots,p-1\}\ \mathcal{V}_\bullet(\mathcal{P}_i(v)) = \mathbf{true}$ alors $\mathcal{V}_\bullet(v) = \mathbf{true}$.

Preuve. Immédiate par induction structurelle sur v. ■

Lemme 26 ($\overset{i}{\rightsquigarrow}$ *implique* $\overset{i}{\rightharpoonup}$)

Soit e une expression. Si $e \overset{i}{\rightsquigarrow} e'$ alors $e \overset{i}{\rightharpoonup} e'$.

Preuve. Par cas sur les axiomes. ■

Lemme 27 (*Un $\rightsquigarrow$ implique l'existence d'un $\rightharpoonup$*)

Soit e une expression. Si $\langle\!\langle \mathcal{P}_0(e),\dots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \rightsquigarrow \langle\!\langle e'_0,\dots,e'_{p-1}\rangle\!\rangle$ alors $\exists e'$ tel que $e \rightharpoonup e'$.

Preuve. Par induction sur e.

- Cas $e = c$, $e = \mathbf{op}$, $e = \overline{n}$, $e = \lambda.e'$, $e = \overline{\lambda.e'[s]}$ et $e = \mu.e'$. $\langle\!\langle \mathcal{P}_0(e),\dots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \not\rightsquigarrow$
- Cas $e = e'[s]$. Nous avons donc $\langle\!\langle \mathcal{P}_0(e),\dots,\mathcal{P}_{p-1}(e)\rangle\!\rangle = \langle\!\langle \mathcal{P}_0(e')[\mathcal{P}_0(s)],\dots,\mathcal{P}_{p-1}(e')[\mathcal{P}_{p-1}(s)]\rangle\!\rangle$. Si $\langle\!\langle \mathcal{P}_0(e')[\mathcal{P}_0(s)],\dots,\mathcal{P}_{p-1}(e')[\mathcal{P}_{p-1}(s)]\rangle\!\rangle \rightsquigarrow \langle\!\langle e''_0,\dots,e''_{p-1}\rangle\!\rangle$ alors nous avons plusieurs cas pour chaque e'_i :
  - $e'_i = (e^1_i\ e^2_i)$ alors $e = (e^1\ e^2)[s] \rightharpoonup \underline{(e^1[s]}\ e^2[s])$
  - $e'_i = \lambda.e^1_i$ alors $e = (\lambda.e^1)[s] \rightharpoonup \underline{\overline{(\lambda.e^1)[s]}}$
  - Les cas $e'_i = \mathbf{op}$, $\mathbf{c}$, $\mu.e^1_i$, $\mathbf{mkpar}\ e^1$, $\overline{n}$ etc. sont similaires
- Cas $e = (e^1, e^2)$. Nous avons donc $\langle\!\langle \mathcal{P}_0(e),\dots,\mathcal{P}_{p-1}(e)\rangle\!\rangle = \langle\!\langle (\mathcal{P}_0(e^1),\mathcal{P}_0(e^2)),\dots,(\mathcal{P}_{p-1}(e^1),\mathcal{P}_{p-1}(e^2))\rangle\!\rangle$. Nous avons alors deux cas :
  - $\forall i\ \mathcal{P}_i(e^1) \neq v^1_i$. Par le lemme 24, nous avons $e^1 \neq v^1$ et donc par induction $e = (e^1, e^2) \rightharpoonup (e'^1, e^2) = e'$
  - $\forall i\ \mathcal{P}_i(e^1) = v^1_i$ et $\mathcal{P}_i(e^2) \neq v^2_i$. Par le lemme 24, nous avons $e^2 \neq v^2$ et $e^1 = v^1$ par le lemme 23. Nous avons donc par induction $e = (e^1, e^2) = (v^1, e^2) \rightharpoonup (v^1, e'^2) = e'$
- Cas $e = \mathbf{mkpar}\ e^1$. Nous avons donc $\langle\!\langle \mathcal{P}_0(e),\dots,\mathcal{P}_{p-1}(e)\rangle\!\rangle = \langle\!\langle \mathbf{mkpar}\ \mathcal{P}_0(e^1),\dots,\mathbf{mkpar}\ \mathcal{P}_{p-1}(e^1)\rangle\!\rangle$. Or $\langle\!\langle \mathbf{mkpar}\ \mathcal{P}_0(e^1),\dots,\mathbf{mkpar}\ \mathcal{P}_{p-1}(e^1)\rangle\!\rangle \rightsquigarrow \langle\!\langle e'_0,\dots,e'_{p-1}\rangle\!\rangle$ donc $e^1_i = f_i$ (une valeur) et ceux $\forall i$. Donc $e^1 = f$ et par le lemme 25, $\mathcal{V}_\bullet(f) = \mathbf{true}$. Par conséquent $\mathbf{mkpar}\ f \rightharpoonup \langle (f\ 0),\dots,(f\ (p-1))\rangle[\bullet]$
- Les cas $e = \mathbf{apply}\ e^1\ e^2$, $\mathbf{proj}\ e^1$, $\mathbf{put}\ e^1$, et $\mathbf{if}\ e^1\ \mathbf{then}\ e^2\ \mathbf{e^3}$ sont similaires
- Cas $e = \langle e^0,\dots,e^{p-1}\rangle$. Nous avons donc $\langle\!\langle \mathcal{P}_0(e),\dots,\mathcal{P}_{p-1}(e)\rangle\!\rangle = \langle\!\langle \langle e^0\rangle,\dots,\langle e^{p-1}\rangle\rangle\!\rangle$ avec $\langle\!\langle \langle e^0\rangle,\dots,\langle e^{p-1}\rangle\rangle\!\rangle \rightsquigarrow \langle\!\langle \langle e^0\rangle,\dots,\langle e^i\rangle,\dots,\langle e^{p-1}\rangle\rangle\!\rangle$. Par le lemme 26, $e^i \overset{i}{\rightharpoonup} e'^i$. Nous construisons $e' = \langle e^0,\dots,e'^i,\dots,e^{p-1}\rangle$ et nous avons $e \rightharpoonup e'$ car $e = \Delta^i[e_i]$ et $e' = \Delta^i[e'_i]$
- Cas $e = (e^1 e^2)$. Nous avons donc $\langle\!\langle \mathcal{P}_0(e),\dots,\mathcal{P}_{p-1}(e)\rangle\!\rangle = \langle\!\langle (\mathcal{P}_0(e^1)\ \mathcal{P}_0(e^2)),\dots,(\mathcal{P}_{p-1}(e^1)\ \mathcal{P}_{p-1}(e^2))\rangle\!\rangle$. Nous avons alors trois cas :
  - $\forall i\ \mathcal{P}_i(e^1) \neq v^1_i$. Par le lemme 24, nous avons $e^1 \neq v^1$ et donc par induction $e = (e^1\ e^2) \rightharpoonup (e'^1\ e^2) = e'$
  - $\forall i\ \mathcal{P}_i(e^1) = v^1_i$ et $\mathcal{P}_i(e^2) \neq v^2_i$. Par le lemme 24, nous avons $e^2 \neq v^2$ et $e^1 = v^1$ par le lemme 23. Nous avons donc par induction $e = (e^1\ e^2) = (v^1\ e^2) \rightharpoonup (v^1\ e'^2) = e'$
  - $\forall i\ \mathcal{P}_i(e^1) = v^1_i$ et $\mathcal{P}_i(e^2) = v^2_i$. Par le lemme 23 nous avons trois cas :
    - $\forall i\ v^1_i = \underline{\mathbf{op}}$ donc $v^1 = \mathbf{op}$ et donc $(\mathbf{op}\ v^2) \rightharpoonup \overline{\mathbf{op}\ v^2}$
    - $\forall i\ v^1_i = \overline{\lambda.e''_i[s_i]}$ alors $e = (\overline{\lambda.e''[s]}\ v^2) \rightharpoonup e''[v^2 \circ s] = e'$
    - $\forall i\ v^1_i = \mathbf{send}$ et donc $v^2_i = [v^0_i,\dots,v^{p-1}_i]$. Alors $\langle\!\langle (\mathbf{send}\ v^2_0),\dots,(\mathbf{send}\ v^2_{p-1})\rangle\!\rangle \rightsquigarrow \langle\!\langle v'^2_0),\dots,v'^2_{p-1})\rangle\!\rangle$ et donc $e = (\mathbf{send}\langle v^2_0,\dots,v^2_{p-1}\rangle) \rightharpoonup \langle v'^2_0,\dots,v'^2_{p-1}\rangle = e'$. ■

Proposition 1 (*Équivalence*)

Soit e^p une expression «programmeur» telle que $e = \mathcal{T}_\bullet(e^p)$. Alors :

1. *Si* $e[\bullet]/0/\langle 0,\ldots,0\rangle \xrightarrow{*} v/c_g/\langle c_0,\ldots,c_{p-1}\rangle$ *alors*
$\langle\!\langle \mathcal{P}_0(e[\bullet]),\ldots,\mathcal{P}_{p-1}(e[\bullet])\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$ *tel que* $\forall i \in \{0,\ldots,p-1\}\ \mathcal{P}_i(v) = v_i$ *;*

2. *Si* $\langle\!\langle \mathcal{P}_0(e[\bullet]),\ldots,\mathcal{P}_{p-1}(e[\bullet])\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$ *alors*
$e[\bullet]/0/\langle 0,\ldots,0\rangle \xrightarrow{*} v/c_g/\langle c_0,\ldots,c_{p-1}\rangle$ *tel que* $\forall i \in \{0,\ldots,p-1\}\ \mathcal{P}_i(v) = v_i$.

Preuve . Notons qu'avec la propriété 1, l'expression e est sans variables libres. Nous avons donc :

1. Soit $e \xrightarrow{*} v$. Nous montrons la conclusion par induction sur la longueur de cette réduction :
   - Si $e = v$ alors par le lemme 20, $\mathcal{P}_i(v) = v_i$ et donc $\langle\!\langle \mathcal{P}_0(v),\ldots,\mathcal{P}_{p-1}(v)\rangle\!\rangle = \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$ avec $\forall i\ \mathcal{P}_i(v) = v_i$. Ce qui est bien le résultat attendu.
   - Si $e \rightharpoonup e' \xrightarrow{*} v$. Par hypothèse d'induction, $\langle\!\langle \mathcal{P}_0(e'),\ldots,\mathcal{P}_{p-1}(e')\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$ avec $\forall i\ \mathcal{P}_i(v) = v_i$. Par le lemme 22, $\langle\!\langle \mathcal{P}_0(e),\ldots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{P}_0(e'),\ldots,\mathcal{P}_{p-1}(e')\rangle\!\rangle$.
     Donc $\langle\!\langle \mathcal{P}_0(e),\ldots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$ avec $\forall i\ \mathcal{P}_i(v) = v_i$.

2. Soit $\langle\!\langle \mathcal{P}_0(e),\ldots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$. Nous montrons la conclusion par induction sur la longueur de cette réduction :
   - Si $\langle\!\langle \mathcal{P}_0(e),\ldots,\mathcal{P}_{p-1}(e)\rangle\!\rangle = \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$ alors par le lemme 23, $e = v$ et donc $v \xrightarrow{*} v$. Ceux-ci est bien le résultat attendu.
   - Si $\langle\!\langle \mathcal{P}_0(e),\ldots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \rightsquigarrow \langle\!\langle e'_0,\ldots,e'_{p-1}\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$ alors par le lemme 23, $\forall i\ \mathcal{P}_i(v) = v_i$ alors v est une valeur. Par le lemme 27, nous avons $\exists e'$ tel que $e \rightharpoonup e'$. Par lemme 22, nous avons $\langle\!\langle \mathcal{P}_0(e),\ldots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{P}_0(e'),\ldots,\mathcal{P}_{p-1}(e')\rangle\!\rangle$. Or par le théorème 6, la relation $\rightharpoonup$ est confluente, donc $\langle\!\langle \mathcal{P}_0(e'),\ldots,\mathcal{P}_{p-1}(e')\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$. La longeur de cette dérivation est égale à celle de $\langle\!\langle \mathcal{P}_0(e),\ldots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0,\ldots,v_{p-1}\rangle\!\rangle$ moins la longeur de la dérivation $\langle\!\langle \mathcal{P}_0(e),\ldots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{P}_0(e'),\ldots,\mathcal{P}_{p-1}(e')\rangle\!\rangle$ qui est non nul. Nous pouvons donc appliquer l'hypothèse d'induction pour conclure que $e' \xrightarrow{*} v$. Par conséquence $e \xrightarrow{*} v$ avec $\forall i\ \mathcal{P}_i(v) = v_i$, ce qui est bien le résultat attendu. ■

4 Une machine abstraite BSP pour BSML

Des versions préliminaires d'une partie de ce chapitre ont fait l'objet des articles [C9] et [C5]. Ces articles ont été écrits en collaboration avec Frédéric Loulergue.

Sommaire

COMPILER un programme consiste à traduire ce programme, dit *code-source*, en un autre programme, dit *code-objet* écrit dans un langage propre à être exécuté sur une machine (et dans notre cas, sur une machine parallèle). Le but de ce chapitre est de prouver la correction des mécanismes mis en jeu par un compilateur BSML.

4.1 Introduction

Les sémantiques données au précédent chapitre donnaient une *vision* sur le fonctionnement des programmes BSML. Néanmoins, le *comment* de l'exécution de ces programmes sur une machine BSP n'était pas précisé. Pour cela il nous faut *compiler* nos termes pour qu'ils soient exécutables.

Notre démarche est celle d'un premier pas vers la preuve d'un compilateur : nous ne voulons pas, pour l'instant, établir de propriété sur les mécanismes «bas niveau» du code-machine tels que la gestion de la mémoire et des registres. Nous allons choisir comme langage objet, un langage suffisamment formel pour que nous puissions prouver la correction de la compilation de nos termes. Ce choix s'est porté vers une *machine abstraite* (encore appelée *machine virtuelle* ou à *environnement*) car celle-ci est simple de compréhension, suffisamment complexe pour être proche d'une machine abstraite réelle (utilisée par un compilateur) et le code des machines abstraites peut être traduit vers le code machine (dit natif) des ordinateurs modernes (le compilateur OCaml fournit un tel mécanisme).

4.2 Définition et correction d'une machine abstraite

La correction d'une machine exprime une relation entre le code-source (provenant du terme de départ) avec lequel on initialise la machine, et la valeur qu'on extrait de l'état terminal. La correction d'une machine

abstraite peut se traduire par : «la valeur résultante de l'exécution est la forme normale du programme source». Pour comprendre les mécanismes mis en œuvre par les instructions d'une machine abstraite et en interpréter les états intermédiaires, nous allons nous donner un formalisme générique pour le code-objet d'une machine quelconque. Pour ce faire, il nous faut expliquer ce qu'est une machine abstraite et ce que veut dire «compiler un programme fonctionnel» (et parallèle).

4.2.1 Définition d'une machine abstraite

Une machine abstraite est définie comme un automate déterministe dont les états décrivent la mémoire d'une pseudo-machine. Les transitions, elles, simulent l'exécution d'un programme sur cette machine abstraite.

Définition 18 (*Machine abstraite [216]*).
Une machine est une paire $(E, \underset{E}{\rightarrow})$ où E est un ensemble d'états et $\underset{E}{\rightarrow}$ un fonction de transition (fonction partielle de $E \times E$ dans E). Nous notons $s' = (\underset{E}{\rightarrow}(s))$ par $s \underset{E}{\rightarrow} s'$ si $c \in E$.

Il existe dans la littérature beaucoup de machines absraites différentes ; leur représentation varie d'une machine à une autre et d'un auteur à un autre. Toutefois, [216] a unifié leur présentation de la manière suivante :

1. Les instructions diffèrent d'une machine à une autre, mais pour chacune d'entre elles, le code est une liste (possiblement vide) d'instructions ;
2. Une fermeture est un morceau de code associé à un environnement (une substitution) et un environnement est une liste de fermetures ou de valeurs ;
3. Certaines machines utilisent une ou plusieurs piles de fermetures et de valeurs ;
4. La structure exacte des blocs varie suivant la machine considérée ; mais un bloc contient au moins toujours un morceau de code et un environnement ;
5. Un état de la machine abstraite est une liste de blocs.

La fonction de transition d'une machine abstraite est définie par cas sur le (ou les) bloc(s) courant(s) d'un état. Notons que, dans la plupart des cas, la transition ne dépend que de l'instruction de tête du code du bloc courant.

Nous ne décrirons, dans ce chapitre, qu'une seule machine abstraite : une CAM. Nous n'utilisons pas la ZAM [176], la machine abstraite de OCaml, car elle est bien trop compliquée pour notre propos. Les lecteurs intéressés peuvent, par une description formelle de cette machine (et sa preuve de correction dans l'assistant de preuve **Coq**), se référer à [127] et à [216] pour un inventaire de machines abstraites prouvées correctes (à la main) et une taxinomie des différentes machines abstraites existantes.

A cette définition de machine abstraite, nous devons ajouter un processus de compilation pour transformer un terme en un état d'une machine afin de démarrer l'exécution.

Définition 19 (*Compilation [216]*).
Soit une machine $(E, \underset{E}{\rightarrow})$. La compilation est une fonction $\mathcal{C} : \Lambda \rightarrow E$. Un état s est dit initial s'il existe un terme e tel que $s = \mathcal{C}(e)$. Un état s' est dit accessible s'il existe un état initial s tel que $s \overset{*}{\underset{E}{\rightarrow}} s'$.

De manière réciproque, il faut définir une fonction de «décompilation» pour nous permettre d'extraire de l'état final le résultat de l'exécution d'un programme. Pour se faire, il nous faut définir la notion de valeur d'une machine abstraite. Nous allons voir pourquoi, le λ-calcul n'est pas un langage suffisant pour exprimer les valeurs des machines abstraites.

4.2.2 Retour sur les substitutions

Une fonction est transformée en un morceau de code fixe, accomplissant les actions inscrites dans son corps. Ce code reste inchangé pour toutes les invocations de la fonction au cours de l'exécution du programme. Cette fonction contient des variables qui peuvent être de deux sortes : ce sont des paramètres ou des variables libres (mais liées à l'extérieur du corps de la fonction). Les paramètres formels changent à chaque invocation de la fonction, tandis que les variables gardent la même valeur. Ceci amène à considérer les fermetures

comme des objets de «bas niveau» décrivant les fonctions. Une fermeture est donc un couple composé du code de la fonction et de l'ensemble des valeurs que prennent les variables libres de la fonction.

Le λ-calcul est insuffisant pour exprimer le code objet. Prenons par exemple, le terme suivant :

$$(\lambda x.\lambda y.(x\ y))(\lambda z.z)$$

Ce terme se réduit en $(\lambda y.((\lambda z.z)\ y))$. D'un point de vie sémantique, rien d'anormal. La variable liée a bien été substituée par le terme donné en paramètre lors de l'application. Mais du point de vue de l'implantation, cela revient à dire que le code a été modifié pour créer cette nouvelle fonction, contredisant notre invariant qui stipule que le code est fixé[1] pendant l'exécution (seules les valeurs de la mémoire changent). Si sur ce même exemple, nous considérons maintenant comme code-objet l'association du λ-terme représentant le code de la fonction et un environnement englobant les valeurs des variables libres de cette fonction, nous obtenons :

$$\boxed{\begin{array}{|c|}\hline (\lambda x.\lambda y.(x\ y))(\lambda z.z) \\ \hline \text{environnement vide} \\ \hline \end{array}} \quad \rightarrow \quad \boxed{\begin{array}{|c|}\hline (\lambda y.(x\ y)) \\ \hline \text{avec } x = (\lambda z.z) \\ \hline \end{array}}$$

où ici, le code contient une variable libre dont la valeur est connue dans l'environnement (encore appelé substitution). Le code source d'un langage fonctionnel est bien le λ-calcul, car il exprime naturellement les fonctions. Mais ce formalisme n'est pas suffisant pour exprimer les valeurs calculées par les programmes. Les fermetures et les substitutions remédient à ce défaut.

4.2.3 Correction d'une machine abstraite

Nous pouvons maintenant préciser la notion de correction d'une machine abstraite puisque nous savons, d'une part, dans quel langage exprimer les valeurs d'une machine et, d'autre part, quel est le calcul censé s'appliquer sur un programme.

Définition 20 (*Décompilation [216]*).
La décompilation $\mathcal{D}$ d'une machine abstraite est une fonction qui à un état de la machine associe un terme du langage.

Le résulat d'un programme est l'image par $\mathcal{D}$ de l'état terminal de l'exécution de ce programme.

Définition 21 (*Correction d'une machine abstraite [216]*).
Soit $(E, \underset{E}{\rightarrow})$ une machine abstraite munie de ses fonctions de compilation $\mathcal{C}$ et de décompilation $\mathcal{D}$. La machine est dite correcte si pour tout code $\mathcal{C}(e)$ qui a pour résultat s', alors v est un réduit de $e[\bullet]$ (par $\rightsquigarrow$) :

$$\textit{Si } \mathcal{C}(e) \overset{*}{\underset{E}{\rightarrow}} s' \textit{ et } v = \mathcal{D}(s') \quad \textit{alors } e[\bullet] \overset{*}{\rightsquigarrow} v$$

Notons que nous ne préciserons pas qu'il existe un résultat de l'évaluation de e. L'évaluation peut être infinie. Mais nous assurerons alors que si cette évaluation de e ne se termine pas, alors la machine abstraite ne s'arrêtera pas pour donner un résultat incohérent.

Les bi-simulations établissent des équivalences entres des systèmes de relations, par exemple, entre deux systèmes de réécriture. [216] a utilisé des bi-simulations pour établir la correction de machines abstraites avec la proposition suivante.

Proposition 2 (*Correction d'une machine abstraite*)

Soit $(E, \underset{E}{\rightarrow})$ une machine abstraite munie de ses fonctions de compilation $\mathcal{C}$ et de décompilation $\mathcal{D}$. Si la fonction de décompilation satisfait les conditions suivantes :

1. *état initial : pour tout terme initial e, $\mathcal{D}(\mathcal{C}(e)) = e[\bullet]$;*
2. *état accessible : soient S_1 et S_2 deux états de la machine tels que $S_1 \underset{E}{\rightarrow} S_2$. Si $\mathcal{D}(S_1)$ est défini alors $\mathcal{D}(S_2)$ est aussi défini aussi et, $\mathcal{D}(S_1) \overset{*}{\rightsquigarrow} \mathcal{D}(S_2)$; Si $\mathcal{D}(S_1) = \mathcal{D}(S_2)$, nous parlons d'une action silencieuse. Nous avons donc ces deux cas de figure :*

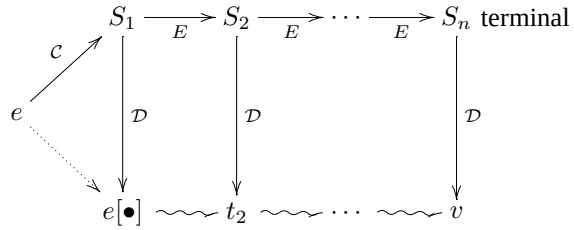
[1] Les programmes modifiant leurs propres codes, comme les virus informatiques, sont ici hors de propos.

3. *progrès : il ne peut y avoir une infinité d'actions silencieuses ;*
4. *état terminal : si S est un état terminal et $\mathcal{D}(S)$ est défini, alors $\mathcal{D}(S)$ est soit une valeur soit $\mathcal{D}(S) \not\rightsquigarrow$(et $\mathcal{D}(S)$ n'est pas une valeur)*

alors la machine abstraite est correcte.

Preuve. Par induction sur $\underset{E}{\rightarrow}$ (voir [216] pour les détails). ■

Le schéma qui suit, résume cette proposition dans un cas idéal :



où chaque état de la machine abstraite correspond à un terme et chaque transition simule la réduction d'un et un seul radical.

Corollaire 1

Si S_1 est un état de la machine tel que $\mathcal{D}(S_1)$ est défini et $\mathcal{D}(S_1) \rightsquigarrow e$, alors il existe S_2 tel que $\mathcal{D}(S_2) = e$ et $S_1 \underset{E}{\overset{*}{\rightarrow}} S_2$.

Preuve. Par application de la proposition 2 et du fait que $\rightsquigarrow$ soit confluent (voir [139] pour les détails). ■

Propriété 2 (*Complétude d'une machine abstraite*)

Soit e une expression

1. *si $e[\bullet] \overset{*}{\rightsquigarrow} v$ alors $\mathcal{C}(e) \underset{E}{\overset{*}{\rightarrow}} S$ tel que $\mathcal{D}(S) = v$*
2. *si $e[\bullet] \overset{*}{\rightsquigarrow} \cdots$ alors $\mathcal{C}(e) \underset{E}{\overset{*}{\rightarrow}} \cdots$ (réductions infinies)*

Preuve. Par induction sur $\rightsquigarrow$ et par application du corollaire 1. ■

4.3 Définition d'une BSP-CAM

4.3.1 Machine abstraite CAM

Pour commencer, nous allons tout d'abord décrire la machine séquentielle : la CAM (*Categoritical Abstract Machine*) [76, 80]. La CAM était la machine abstraite de feu CAML[2]. Nous n'avons pas choisi la ZAM qui est la machine abstraite de OCaml pour des raisons de simplicité. En effet, cette machine est bien plus compliquée et les difficultés de simulation qu'elle entraîne sont hors propos dans ce travail.

La CAM utilise un jeu d'instructions qui est issu des combinateurs catégoriques [80]. Les instructions sont les suivantes :

Cst Op Fst Snd Push Swap Cons App Closure CloseRec Case

Le code est donc une liste de telles instructions terminée par l'«instruction vide» (fin de la liste). Les environnements sont dénotés par des arbres binaires de valeurs (fermetures, constantes, opérateurs, vecteurs). Une fermeture est constituée d'un code et d'un environnement. La pile utilisée par cette machine est une

[2]CAML était l'ancêtre de Caml-light qui a ensuite donné OCaml. Le lecteur archéologue peut consulter `http://www.pps.jussieu.fr/~cousinea/Caml/caml_history.html` pour un historique plus précis.

$$\{e \circ P, \mathbf{Cst}(\mathbf{c}); C, t\} \underset{\mathbf{Cst}}{\rightarrow} \{\mathbf{c} \circ P, C, t\} \quad (4.1)$$

$$\{e \circ P, \mathbf{Op}(\mathbf{op}); C, t\} \underset{\mathbf{Op}}{\rightarrow} \{[\overline{\mathbf{op}}, e] \circ P, C, t\} \quad (4.2)$$

$$\{(e_1, e_2) \circ P, \mathbf{Fst}; C, t\} \underset{\mathbf{Fst}}{\rightarrow} \{e_1 \circ P, C, t\} \quad (4.3)$$

$$\{(e_1, e_2) \circ P, \mathbf{Snd}; C, t\} \underset{\mathbf{Snd}}{\rightarrow} \{e_2 \circ P, C, t\} \quad (4.4)$$

$$\{e \circ P, \mathbf{Push}; C, t\} \underset{\mathbf{Push}}{\rightarrow} \{e \circ e \circ P, C, t\} \quad (4.5)$$

$$\{e_1 \circ e_2 \circ P, \mathbf{Swap}; C, t\} \underset{\mathbf{Swap}}{\rightarrow} \{e_2 \circ e_1 \circ P, C, t\} \quad (4.6)$$

$$\{e_1 \circ e_2 \circ P, \mathbf{Cons}; C, t\} \underset{\mathbf{Cons}}{\rightarrow} \{(e_2, e_1) \circ P, C, t\} \quad (4.7)$$

$$\{([C', e_1], e_2) \circ P, \mathbf{App}; C, t\} \underset{\mathbf{App}}{\rightarrow} \{(e_1, e_2) \circ P, C'; C, t\} \quad (4.8)$$

$$\{e \circ P, \mathbf{Closure}(C'); C, t\} \underset{\mathbf{Closure}}{\rightarrow} \{[C', e] \circ P, C, t\} \quad (4.9)$$

$$\{e \circ P, \mathbf{CloseRec}(C'); C, t\} \underset{\mathbf{CloseRec}}{\rightarrow} \{[C', e] \circ P, C'; C, t\} \quad (4.10)$$

$$\{\mathbf{true} \circ P, \mathbf{Case}(C_1, C_2); C, t\} \underset{\mathbf{Case}}{\rightarrow} \{P, C_1; C, t\} \quad (4.11)$$

$$\{\mathbf{false} \circ P, \mathbf{Case}(C_1, C_2); C, t\} \underset{\mathbf{Case}}{\rightarrow} \{P, C_2; C, t\} \quad (4.12)$$

$$\{(n_1, n_2) \circ P, \mathbf{Add}; C, t\} \underset{\mathbf{Add}}{\rightarrow} \{n_1 + n_2 \circ P, C, t\} \quad (4.13)$$

$$\{e \circ P, \mathbf{Isnc}; C, t\} \underset{\mathbf{Isnc}}{\rightarrow} \{e' \circ P, C, t\} \text{ avec} \begin{cases} e' = \mathbf{true}\ si\ e = \mathbf{nc} \\ e' = \mathbf{false}\ \text{sinon} \end{cases} \quad (4.14)$$

$$\{([e_0, \ldots, e_i, \ldots, e_n], i) \circ P, \mathbf{Access}; C, t\} \underset{\mathbf{Access}}{\rightarrow} \{e_i \circ P, C, t\} \quad (4.15)$$

$$\{(f, n) \circ P, \mathbf{Init}; C, t\} \underset{\mathbf{Init}}{\rightarrow} \{(f, 0) \circ \cdots \circ (f, n-1) \circ P, C, t\} \quad (4.16)$$

$$\{e_0 \circ \cdots \circ e_{n-1} \circ P, \mathbf{Tab}^n; C, t\} \underset{\mathbf{Tab}}{\rightarrow} \{[e_0, \ldots, e_{n-1}] \circ P, C, t\} \quad (4.17)$$

Figure 4.1 — Règles de transition de la CAM

liste d'environnements et non de valeurs. Une CAM est l'association du code courant, d'une pile et d'un type de contexte, loc ou glo (ces types seront utilisés pour différencier l'exécution en dehors ou en dedans d'un vecteur). Nous avons donc :

Code	::=	Instruction; Code	|	ε	
Environnement	::=	()	|	v	| (Environnement, Environnement)
Fermeture	::=	[Environnement, Code]			
Pile	::=	Environnement $\circ$ Pile	|	$\bullet$	
CAM	::=	{Pile, Code, Type}			

Nous noterons la pile P, le code C et le type t.

4.3.2 Transition de la CAM

A chaque instruction correspond une règle de transition particulière. Avant de détailler les règles de transition, remarquons que chacune d'entre elles suppose que la tête de la pile soit d'une forme précise. Le fonctionnement des instructions de la CAM est le suivant :

- **Cst** : cette instruction modifie la tête de la pile en une constante ;
- **Op** : cette instruction modifie la tête de la pile en une fermeture associant le code $\overline{\mathbf{op}}$ de l'opérateur et l'environnement en tête de pile.
- **Fst** et **Snd** : ces instructions sont respectivement la première et la deuxième projection des couples. Nous verrons par la suite que les environnements sont en fait des listes de fermetures stockées sous la forme d'arbres binaires[3]. Sous cette hypothèse, **Fst** remplace l'environnement par lui-même

[3]Nous gardons tout de même cette représentation afin d'avoir gratuitement les couples comme nous le verrons par la suite.

privé de sa première valeur et **Snd** accède à la nème valeur de l'environnement. Ces instructions apparaissent toujours dans une série d'instructions de la forme $\mathbf{Fst}^n; \mathbf{Snd}$. Leur exécution a pour effet d'accéder à la $(n+1)$ème valeur de l'environnement en tête de pile (l'environnement est détruit pour être remplacé par une valeur) ;

- **Push** : cette instruction a pour effet de dupliquer l'environnement courant afin que deux séries d'instructions aient chacune une copie de cet environnement (cela partage donc l'environnement courant) ;
- **Swap** : cette instruction a pour objet d'échanger les têtes de pile pour restaurer l'environnement courant avant d'entreprendre l'évaluation d'une autre série d'instructions ;
- **Cons** : construit une paire avec deux valeurs ;
- **App** : cette instruction s'exécute sur une paire (valeur d'une fonction ou d'un opérateur, valeur d'un argument). Elle introduit un nouveau calcul en reprenant l'exécution sur le code de la fermeture avec pour environnement courant son environnement augmenté par la valeur de l'argument. Une fois ce code entièrement exécuté, le calcul reprendra là où il avait été laissé mais avec le résultat du calcul en tête de pile.
- $\mathbf{Closure}(C')$: cette instruction crée une fermeture associant le code C' et l'environnement en tête de pile. Cette instruction a aussi pour effet de détruire l'environnement courant. L'instruction **CloseRec** fait de même mais exécute préalablement C' (récursion) ;
- $\mathbf{Case}(C_1, C_2)$: cette instruction permet de choisir l'exécution du code C_1 ou du code C_2 suivant la tête de la pile et en le consommant ;

La figure 4.1 donne formellement les règles de transition $\xrightarrow[E]{i,p}$ de ces instructions. Notons que cette fonction de transition est dépendante de deux entiers i et p qui seront le «pid» du la CAM et le nombre de processeurs de la BSP-CAM.

4.3.3 De la CAM à la BSP-CAM

Dans un paradigme SPMD, le programme est identique en chaque processeur. Comme nous l'avons vu dans le chapitre 3, les programmes BSML peuvent être évalués comme des programmes SPMD (sémantique distribuée). Pour évaluer nos programmes BSML sur une machine abstraite BSP, nous reprenons ce principe pour définir une BSP-CAM.

Définition 22 (*BSP-CAM*).
Une BSP-CAM est définie en tant qu'adjonction de p CAM, c'est-à-dire :

$$\langle\!\langle (P_0, C_0, t_0), \cdots, (P_{p-1}, C_{p-1}, t_{p-1}) \rangle\!\rangle$$

La fonction de transition $\underset{E}{\rightarrow}$ de notre machine abstraite (la BSP-CAM) se définit par :

1. Évaluation asynchrone $\xrightarrow[E]{i,p}$ d'une instruction d'une CAM ;
2. Évaluation de l'instruction parallèle globale et synchrone **SEND**.

L'évaluation asynchrone d'une instruction d'une CAM de la BSP-CAM se définit par :

$$\langle\!\langle \cdots, (P_i, \mathbf{INSTR}; C_i, t_i), \cdots,) \rangle\!\rangle \xrightarrow{E} \langle\!\langle \cdots, (P_i', C_i', t_i'), \cdots,) \rangle\!\rangle$$

$$\text{si } (P_i, \mathbf{INSTR}; C_i, t_i) \xrightarrow[E]{i,p} (P_i', C_i', t_i')$$

Notons que dans cette évaluation asynchrone nous avons i et p qui sont liés, comme dans un programme SPMD avec les variables «pid» et «nprocs».

Maintenant, il nous faut donner les instructions qui sont spécifiquement «parallèles». Nous complétons alors les fonctions de transition $\underset{E}{\rightarrow}$ (instruction synchrone) et $\xrightarrow[E]{i,p}$ (instruction asynchrone) de la BSP-CAM avec ces nouvelles instructions. Ces instructions sont naturellement dépendantes de p, le nombre de processeurs, mais elles ne le sont qu'à l'exécution (et non à la compilation comme dans [208, 207]). Les trois instructions fonctionnent ainsi :

1. L'instruction $\mathbf{NPROC}$ (à la manière de l'instruction $\mathbf{Cst}$) retourne le nombre de processeurs sur la pile de la CAM de ce processeur :

$$(e \circ P, \mathbf{NPROC}; C_i, t_i) \underset{\mathbf{NPROC}}{\overset{i,p}{\rightarrow}} (p \circ P, C_i, t_i)$$

2. L'instruction $\mathbf{PID}$ (à la manière de l'instruction $\mathbf{Cst}$) retourne le «pid» du processeur sur la pile de la CAM de ce processeur :

$$(e \circ P, \mathbf{PID}; C_i, loc) \underset{\mathbf{PID}}{\overset{i,p}{\rightarrow}} (i \circ P, C_i, loc)$$

3. L'instruction $\mathbf{SEND}$ permet l'échange des valeurs entre les processeurs. Elle est donc globale (et synchrone) et modifient les têtes de pile de toutes les CAM :

$$\langle\langle([v_0^0, \ldots, v_0^{p-1}] \circ P_0, \mathbf{SEND}; C_0, glo), \cdots, ([v_{p-1}^0, \ldots, v_{p-1}^{p-1}] \circ P_{p-1}, \mathbf{SEND}; C_{p-1}, e_{p-1}, glo)\rangle\rangle$$
$$\underset{E}{\rightarrow}$$
$$\langle\langle([v_0^0, \ldots, v_{p-1}^0] \circ P_0, \mathbf{SEND}; C_0, glo), \cdots, ([v_0^{p-1}, \ldots, v_{p-1}^{p-1}] \circ P_{p-1}, \mathbf{SEND}; C_{p-1}, e_{p-1}, glo)\rangle\rangle$$

Notons que l'instruction $\mathbf{SEND}$ n'a de sens que dans un contexte global puisqu'elle est globale à toutes les CAM. *A contrario*, l'instruction $\mathbf{PID}$ n'a de sens que dans un contexte local (dans un vecteur parallèle).

4.4 Compilation de BSML

4.4.1 Termes séquentiels

La transformation des expressions (sans les primitives parallèles) en des instructions CAM se définit par induction comme suit :

$$\begin{aligned}
[\![1]\!] &= \mathbf{Snd} \\
[\![n+1]\!] &= \mathbf{Fst}; [\![n]\!] \\
[\![\mathbf{c}]\!] &= \mathbf{Cst}(\mathbf{c}) \\
[\![\mathbf{op}]\!] &= \mathbf{Op}(\mathbf{op}) \\
[\![\lambda.e]\!] &= \mathbf{Closure}([\![e]\!]) \\
[\![\mu.e]\!] &= \mathbf{CloseRec}([\![e]\!]) \\
[\![(e_1\ e_2)]\!] &= \mathbf{Push}; [\![e_1]\!]; \mathbf{Swap}; [\![e_2]\!]; \mathbf{Cons}; \mathbf{App} \\
[\![(e_1, e_2)]\!] &= \mathbf{Push}; [\![e_1]\!]; \mathbf{Swap}; [\![e_2]\!]; \mathbf{Cons} \\
[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] &= \mathbf{Push}; [\![e_1]\!]; \mathbf{Case}([\![e_2]\!], [\![e_3]\!])
\end{aligned}$$

Remarquons tout de suite que dans les suites d'instructions générées par cette fonction, les $\mathbf{Push}$, $\mathbf{Cons}$, $\mathbf{App}$ et $\mathbf{Case}$ forment un parenthésage équilibré. Cela nous permettra de caractériser la forme du code lors de la définition de la fonction de décompilation.

4.4.2 Primitives parallèles

Plutôt que de donner une fonction de compilation $\mathcal{C}$ de nos termes dans une BSP-CAM, nous allons procéder en deux étapes. La première consiste à transformer les termes qui sont spécifiques au parallèlisme, c'est-à-dire les primitives parallèles. Ensuite la fonction de transformation sur ces termes permet d'obtenir le code parallèle. Ceci nous permet de différencier ce qui est propre à la CAM de ce qui est générique à BSML.

La figure 4.2 donne le schéma de compilation des primitives parallèles. Notons tout de suite que celles-ci sont «supprimées» afin de fournir un code générique dépendant d'opérations parallèles de plus bas niveau. Détaillons ce schéma de compilation :

- La primitive $\mathbf{mkpar}$ est traduite en une fonction qui attend un argument, puis crée un vecteur avec cet argument et le pid du processeur ;
- La primitve $\mathbf{apply}$ est traduite en une fonction à deux arguments (les vecteurs) et qui applique le premier argument au deuxième. Un vecteur est alors construit à partir du résultat du calcul local ;

$$
\begin{array}{rcl}
[\![\mathbf{mkpar}\ e]\!] & = & [\![((\lambda.\mathbf{par}\ ((\overline{1}\ \mathbf{pid})))\ e)]\!] \\
[\![\mathbf{apply}\ e_1\ e_2]\!] & = & [\![((\lambda.\lambda.\mathbf{par}\ ((\overline{2}\ \overline{1})))\ e_1\ e_2)]\!] \\
[\![\mathbf{proj}\ e]\!] & = & [\![((\lambda.(\mathbf{delpar}\ (\mathbf{put}\ (\mathbf{par}\ \lambda.\overline{2}))))\ e)]\!] \\
[\![\mathbf{put}\ e]\!] & = & [\![((\lambda.(\mathbf{apply}\ (\mathbf{mkpar}\ \lambda.\mathbf{F})\ (\mathbf{send}\ (\mathbf{par}\ (\mathbf{init}\ (\overline{1}, p))))))\ e)]\!]
\end{array}
$$

Figure 4.2 — Compilation des primitives

$[\![\mathbf{mkpar}\ e]\!] =$

$$\mathbf{Push}; \mathbf{Closure}(\mathbf{Loc}; \mathbf{Push}; \overline{1}; \mathbf{Swap}; \mathbf{PID}; \mathbf{Cons}; \mathbf{App}; \mathbf{Par}); \mathbf{Swap}; [\![e]\!]; \mathbf{Cons}; \mathbf{App}$$

$[\![\mathbf{apply}\ e_1\ e_2]\!] =$

$$\mathbf{Push}; \mathbf{Push}; \mathbf{Closure}(\mathbf{Closure}(\mathbf{Loc}; \mathbf{Push}; \overline{2}; \mathbf{Swap}; \overline{1}; \mathbf{Cons}; \mathbf{App}; \mathbf{Par})); \mathbf{Swap}; [\![e1]\!]; \mathbf{Cons}; \mathbf{App}; \mathbf{Swap}; [\![e1]\!]; \mathbf{Cons}; \mathbf{App}$$

$[\![\mathbf{put}\ e]\!] =$

$$\mathbf{Push}; \mathbf{Closure}(\mathbf{Push}; \mathbf{Push}; \mathbf{Closure}(\mathbf{Closure}(\mathbf{Push}; \overline{2}; \mathbf{Swap}; \overline{1}; \mathbf{Cons}; \mathbf{App})); \mathbf{Swap}; \mathbf{Push}; \mathbf{Closure}(\mathbf{Loc}; \mathbf{Push}; \overline{1}; \mathbf{Swap}; \mathbf{PID}; \mathbf{Cons}; \mathbf{App}; \mathbf{Par}); \mathbf{Swap}; \mathbf{Closure}([\![\mathbf{F}]\!]); \mathbf{Cons}; \mathbf{App}; \mathbf{Cons}; \mathbf{App}; \mathbf{Swap}; \mathbf{Push}; \mathbf{Op}(send); \mathbf{Swap}; \mathbf{Loc}; \mathbf{Push}; \mathbf{Op}(init); \mathbf{Swap}; \mathbf{Push}; \overline{1}; \mathbf{Swap}; \mathbf{Cst}(p); \mathbf{Cons}; \mathbf{Cons}; \mathbf{App}; \mathbf{Par}; \mathbf{Cons}; \mathbf{App}; \mathbf{Cons}; \mathbf{App}); \mathbf{Swap}; [\![e]\!]; \mathbf{Cons}; \mathbf{App}$$

$[\![\mathbf{proj}\ e =]\!]$

$$\begin{array}{l}
\mathbf{Push}; \mathbf{Closure}(\mathbf{Push}; \mathbf{Op}(delpar); \mathbf{Swap}; \mathbf{Push}; \mathbf{Closure}(\mathbf{Push}; \mathbf{Push}; \\
\mathbf{Closure}(\mathbf{Closure}(\mathbf{Loc}; \mathbf{Push}; \overline{2}; \mathbf{Swap}; \overline{1}; \mathbf{Cons}; \mathbf{App}; \mathbf{Par})); \\
\mathbf{Swap}; \mathbf{Push}; \mathbf{Closure}(\mathbf{Loc}; \mathbf{Push}; \overline{1}; \mathbf{Swap}; \mathbf{PID}; \mathbf{Cons}; \mathbf{App}; \mathbf{Par}); \\
\mathbf{Swap}; \mathbf{Closure}([\![\mathbf{F}]\!]); \mathbf{Cons}; \mathbf{App}; \mathbf{Cons}; \mathbf{App}; \mathbf{Swap}; \mathbf{Push}; \mathbf{Op}(send); \\
\mathbf{Swap}; \mathbf{Loc}; \mathbf{Push}; \mathbf{Op}(init); \mathbf{Swap}; \mathbf{Push}; \overline{1}; \mathbf{Swap}; \mathbf{Cst}(p); \mathbf{Cons}; \mathbf{Cons}; \\
\mathbf{App}; \mathbf{Par}; \mathbf{Cons}; \mathbf{App}; \mathbf{Cons}; \mathbf{App}); \mathbf{Swap}; [\![e]\!]; \mathbf{Cons}; \mathbf{App}; \mathbf{Cons}; \mathbf{App}); \\
\mathbf{Swap}; [\![e]\!]; \mathbf{Cons}; \mathbf{App}
\end{array}$$

Figure 4.3 — Instructions des primitives parallèles

- La primitive **proj** est traduite en une fonction qui attend un argument et construit avec celui-ci une fonction qui sera donnée à la primitive **put**. Le constructeur de vecteur retourné sera ensuite détruit afin d'avoir le résultat final ;
- La primitive **put** est traduite en une fonction qui attend un argument et qui construit un vecteur de tableau puis échange les données ainsi calculées et retourne le résultat sous la forme d'un vecteur

La figure 4.3 donne les ensembles complets d'instructions de ce schéma de compilation de nos primitives parallèles.
Ce schéma de compilation correspond à l'implantation suivante de nos primitives en OCaml (nous verrons ensuite comment optimiser cette implantation) :

```
type α par = Par of α
let delpar (Par a) = a
let mkpar = fun f →Par(f pid)
let apply = function (Par e1) (Par e2) →Par (e1 e2)
let put = function (Par e) →apply
      (mkpar (fun _ t i →if ((0<=i) && (i<nprocs)) then t.(i) else None))
      (send (Par (Array.init nprocs e)))
let proj = function (Par e) →delpar (put (Par (fun _ →e)))
```

La compilation de nos primitives parallèles introduit un nouveau constructeur, celui des vecteurs parallèles. La fonction de transformation est étendue par :

$$[\![\mathbf{par}\ e]\!] = \mathbf{Loc}; [\![e]\!]; \mathbf{Par}$$

$$
\begin{array}{rcl}
\mathcal{V}_e(\mathbf{c}) & = & \mathbf{true} \\
\mathcal{V}_e(\mathbf{op}) & = & \mathbf{true} \\
\mathcal{V}_e((v_1, v_2)) & = & \mathcal{V}_e(v_1) \wedge \mathcal{V}_e(v_2) \\
\mathcal{V}_e([v_1, \ldots, v_n]) & = & \bigwedge_{i=0}^{n} \mathcal{V}_e(v_i) \\
\mathcal{V}_e(\langle v \rangle) & = & \mathbf{false} \\
\mathcal{V}_e([C, e']) & = & \mathcal{V}_{(x,e')}(C) \\
\mathcal{V}_{(e1,e2)}(\mathbf{Fst}; C) & = & \mathcal{V}_{e_1}(C) \\
\mathcal{V}_{(e1,v)}(\mathbf{Snd}; C) & = & \mathcal{V}_{()}(v) \wedge \mathcal{V}_{v_2}(C) \\
\mathcal{V}_e(\mathbf{Closure}(C'); C) & = & \mathcal{V}_{(x,e)}(C') \wedge \mathcal{V}_e(C) \\
\mathcal{V}_e(\mathbf{CloseRec}(C'); C) & = & \mathcal{V}_{(x,e)}(C') \wedge \mathcal{V}_e(C) \\
\mathcal{V}_e(\mathbf{Loc}; C) & = & \mathbf{false} \\
\mathcal{V}_e(\mathbf{SEND}; C) & = & \mathbf{false} \\
\mathcal{V}_e(\mathbf{Par}; C) & = & \mathbf{false} \\
\mathcal{V}_e(\mathbf{Delpar}; C) & = & \mathbf{false} \\
\mathcal{V}_e(\mathbf{Case}(C_1, C_2); C) & = & \mathcal{V}_e(C_1) \wedge \mathcal{V}_e(C_2) \wedge \mathcal{V}_e(C) \\
\mathcal{V}_e(\mathbf{INSTR}; C) & = & \mathcal{V}_e(C) \text{ si } \mathbf{INSTR} = \text{autres instructions}
\end{array}
$$

Figure 4.4 — Vérification d'un paramètre de la BSP-CAM

Nous avons alors deux nouvelles instructions. Les règles sont alors les suivantes :

$$\{\langle e \rangle \circ P, \mathbf{Loc}; C, glo\} \underset{\mathbf{Loc}}{\rightarrow} \{e \circ P, C, loc\} \tag{4.18}$$

$$\{(\langle e_1 \rangle, \langle v_2 \rangle) \circ P, \mathbf{Loc}; C, glo\} \underset{\mathbf{Loc}}{\rightarrow} \{(e_1, e_2) \circ P, C, loc\} \tag{4.19}$$

$$\{e \circ P, \mathbf{Loc}; C, glo\} \underset{\mathbf{Loc}}{\rightarrow} \{e \circ P, C, loc\} \text{ si } \mathcal{V}_{()}(e) = \mathbf{true} \tag{4.20}$$

$$\{e \circ P, \mathbf{Par}; C, loc\} \underset{\mathbf{Par}}{\rightarrow} \{\langle e \rangle \circ P, C, glo\} \tag{4.21}$$

$$\{\langle e \rangle \circ P, \mathbf{Delpar}; C, glo\} \underset{\mathbf{Delpar}}{\rightarrow} \{e \circ P, C, glo\} \tag{4.22}$$

L'instruction $\mathbf{Loc}$ permet de passer du contexte global (en dehors du vecteur) au contexte local. Cette instruction testera aussi la validité des paramètres (pouvons-nous évaluer cette valeur localement, comme dans le cas d'un $\mathbf{mkpar}$?) ou détruira le constructeur de vecteur afin d'évaluer localement l'expression (cas d'un $\mathbf{put}$ ou d'un $\mathbf{proj}$) ou pour un couple de vecteur (cas d'un $\mathbf{apply}$). L'instruction $\mathbf{Par}$ construit le vecteur parallèle et fait repasser au contexte global. L'instruction $\mathbf{Delpar}$ correspond à l'opérateur $\mathbf{delpar}$ qui supprime le constructeur des vecteurs. Le test $\mathcal{V}_{()}$ est défini à la figure 4.4. Celui-ci parcourt inductivement la valeur. Si la valeur est une fermeture, alors le test parcourt les instructions pour vérifier qu'elles ne font pas référence à des valeurs parallèles ou qu'elles ne sont pas elles-mêmes des instructions parallèles.

Définition 23 (*Compilation de BSML dans la BSP-CAM*).
Soit e un terme. La fonction de compilation $\mathcal{C}$ est définie par :

$$\mathcal{C}(e) = \langle\!\langle \{\bullet, [\![e]\!], glo\}, \cdots, \{\bullet, [\![e]\!], glo\} \rangle\!\rangle$$

4.4.3 Correction de la BSP-CAM

Pour prouver la correction de notre machine nous utilisons la fonction de décompilation $\mathcal{D}$ définie à la figure 4.5 où la décompilation du code se fait suivant la fonction inverse de la compilation. Les fermetures se traduisent en des fermetures de notre calcul.

La décompilation d'une CAM (Pile et Code) est un peu plus difficile à définir et cela pour trois raisons connexes :

1. La pile est hétérogène, elle peut contenir soit des valeurs, soit des paires d'environnements. Cette situation interdit de définir de manière générique la traduction de la pile ;

2. Le code obtenu dans les états intermédiaires du calcul (après exécution d'une partie des instructions) n'est pas décompilable par la simulation directe du code

Décompilation de la BSP-CAM :

$$\mathcal{D}(\langle\!\langle (P_0, C_0, t_0), \cdots, (P_{p-1}, C_{p-1}, t_{p-1}) \rangle\!\rangle) = \langle\!\langle \mathcal{D}(P_0, C_0, t_0), \cdots, \mathcal{D}(P_{p-1}, C_{p-1}, t_{p-1}) \rangle\!\rangle$$

Décompilation du code, des fermetures et des environnements :

$$\begin{array}{rcl}
 & & \text{Code} \\
\overline{\overline{\mathbf{Fst}^n; \mathbf{Snd}}} & = & n+1 \\
\overline{\overline{\mathbf{Cst}(\mathbf{c})}} & = & \mathbf{c} \\
\overline{\overline{\mathbf{Op}(\mathbf{op})}} & = & \mathbf{op} \\
\overline{\overline{\mathbf{Closure}(C)}} & = & \lambda.\overline{\overline{C}} \\
\overline{\overline{\mathbf{Push}; C_1; \mathbf{Swap}; C_2; \mathbf{Cons}; \mathbf{App}}} & = & (\overline{\overline{C_1}}\ \overline{\overline{C_2}}) \\
\overline{\overline{\mathbf{Push}; C_1; \mathbf{Swap}; C_2; \mathbf{Cons}}} & = & (\overline{\overline{C_1}}, \overline{\overline{C_2}}) \\
\overline{\overline{\mathbf{Push}; C_1; \mathbf{Case}(C_2, C_3)}} & = & \mathbf{if}\ \overline{\overline{C_1}}\ \mathbf{then}\ \overline{\overline{C_2}}\ \mathbf{else}\ \overline{\overline{C_3}} \\
Code(mkpar); C & = (\mathbf{mkpar}\ \overline{\overline{C}}) & \\
Code(apply); C_1; C_2 & = (\mathbf{apply}\ \overline{\overline{C_1}}\ \overline{\overline{C_2}}) & \\
Code(proj); C & = (\mathbf{proj}\ \overline{\overline{C}}) & \\
Code(put); C & = (\mathbf{put}\ \overline{\overline{C}}) & \\
 & & \text{Environnement} \\
\overline{\overline{\bullet}} & = & \bullet \\
\overline{\overline{(e, f)}} & = & \overline{\overline{f}} \circ \overline{\overline{e}} \\
 & & \text{Fermeture} \\
\overline{\overline{[C, e]}} & = & \overline{\overline{C}}[\overline{\overline{e}}]
\end{array}$$

Décompilation d'une CAM :

$$\mathcal{D}((P_i, C_i, t_i)) = \Psi(\Phi((P_i, C_i, t_i)))$$

Début Φ de décompilation d'une CAM :

$$\begin{array}{rcl}
\Phi(\{\bullet, C, t\}) & = & (\overline{\overline{C}}, \{() \circ \bullet, \varepsilon, t\} \\
\Phi(\{e \circ P, C_1; C, t\}) & = & (\overline{\overline{C_1}}[\overline{\overline{e}}], \{P, C, t\}) \\
\Phi(\{f \circ e \circ P, \mathbf{Swap}; C_2; \mathbf{Cons}; \mathbf{App}; C, t\}) & = & (\overline{\overline{f}}, \{e \circ P, \mathbf{Swap}; C_2; \mathbf{Cons}; \mathbf{App}; C, t\}) \\
\Phi(\{f_1 \circ f_2 \circ P, \mathbf{Cons}; \mathbf{App}; C, t\}) & = & (\overline{\overline{f_1}}, \{\overline{\overline{f_2}} \circ P, \mathbf{Cons}; \mathbf{App}; C, t\}) \\
\Phi(\{(f_1, f_2) \circ P, C, t\}) & = & ((\overline{\overline{f_1}}\ \overline{\overline{f_2}}), \{\overline{\overline{f_2}} \circ P, C, t\}) \\
\Phi(\{f \circ e \circ P, \mathbf{Swap}; C_2; \mathbf{Cons}; C, t\}) & = & (\overline{\overline{f}}, \{e \circ P, \mathbf{Swap}; C_2; \mathbf{Cons}; C, t\}) \\
\Phi(\{f_1 \circ f_2 \circ P, \mathbf{Cons}; C, t\}) & = & (\overline{\overline{f_1}}, \{\overline{\overline{f_2}} \circ P, \mathbf{Cons}; C, t\}) \\
\Phi(\{(f_1, f_2) \circ P, C, t\}) & = & ((\overline{\overline{f_1}}, \overline{\overline{f_2}}), \{\overline{\overline{f_2}} \circ P, C, t\})
\end{array}$$

Fin Ψ de décompilation d'une CAM :

$$\begin{array}{rcl}
\Psi(f, \{\bullet, \varepsilon, t\}) & = & f \\
\Psi(f, \{e \circ P, \mathbf{Swap}; C_2; \mathbf{Cons}; \mathbf{App}; C, t\}) & = & \Psi((f\overline{\overline{C_2}}[\overline{\overline{e}}], \{P, C, t\})) \\
\Psi(f_1, \{f_2 \circ P, \mathbf{Cons}; \mathbf{App}; C, t\}) & = & \Psi((\overline{\overline{f_2}}\ f_2), \{P, C, t\})
\end{array}$$

Figure 4.5 — Fonctions de décompilation

3. Les instructions provenant des primitives parallèles ne doivent pas être confondues avec celles provenant de la compilation des termes séquentiels.

Pour remédier à ces problèmes, nous définissons la fonction de reconstruction par cas sur la tête de code et en sachant que la suite des instructions aura forcément une forme bien définie (les instructions forment

un parenthésage équilibré). La définition d'un état quelconque d'une CAM se définit par induction sur la tête du code. Nous procédons par étapes : en premier lieu nous définissons une fonction Φ qui a une CAM associe une expression et une CAM. Cette fonction est définie à la figure 4.5. Ensuite, nous poursuivons la décompilation en utilisant le fait que si le code de départ provient de la compilation (bon parenthésage) alors son image par Φ débute forcément par l'instruction $\mathbf{Cons}$, soit par l'instruction $\mathbf{Swap}$ ou est vide. La seconde fonction de décompilation Ψ est définie à la figure 4.5.

La simulation (fonction de décompilation) telle qu'elle est définie est bien entendu partielle : un état quelconque n'a pas forcément de traduction. Cependant, si nous décompilons un état accessible, le code a une forme telle que la simulation est définie (en faisant aussi l'hypothèse sur le caractère décompilable des éléments de la pile). D'autre part, la simulation est apparemment ambiguë. En effet, pour un code quelconque, il y a a priori plusieurs façons de le découper. Là encore, si on ne considère que des états accessibles, cette ambiguïté est levée.

Avec cette fonction de décompilation, nous pouvons alors prouver les quatre conditions de la proposition 2 pour obtenir la correction de notre machine. Les preuves de ces conditions se trouve à l'annexe 4.A. Nous avons alors le résultat suivant :

Proposition 3 (*Correction de la BSP-CAM*)

La BSP-CAM munie de $\underset{E}{\rightarrow}$, de $\mathcal{C}$ et de $\mathcal{D}$ est correcte.

Preuve . Par application de la proposition 2 et des lemmes 29, 30, 31 et 32. ■

Notre BSP-CAM est donc correcte. On peut donc compiler avec cette machine afin d'obtenir des résultats valides. De plus, si l'évaluation du programme ne se termine pas alors la machine boucle. Nous allons maintenant voir comment optimiser cette machine de manière sûre.

4.4.4 Optimisation

Il est facile de voir que le constructeur Par des vecteurs parallèles n'est pas réellement utile pour la sémantique distribuée tout comme pour la machine abstraite BSP. Ce constructeur permet juste de distinguer, pour les preuves, les valeurs qui sont répliquées sur les processeurs, des vecteurs parallèles. Nous pouvons donc les supprimer afin d'optimiser le code. En supposant aussi une analyse statique $\mathcal{A}$ qui n'admet que des programmes pouvant s'exécuter correctement[4], c'est-à-dire si $\mathcal{A}(e)$ alors $e \rightsquigarrow e'$ et $\mathcal{A}(e')$, alors nous pouvons aussi supprimer le test $\mathcal{V}$ qui interdit l'emboîtement des vecteurs parallèles.

Cette machine abstraite est donc non sûre en général, mais reste correcte dans le cas des programmes qui s'exécuteront bien. Cette machine se définit comme la précédente mais avec les instructions $\mathbf{Loc}$, $\mathbf{Par}$ qui sont des instructions ne faisant rien[5], c'est-à-dire :

$$\begin{array}{rcl} \{e \circ P, \mathbf{Loc}; C, t\} & \underset{\mathbf{Loc}}{\rightarrow} & \{e \circ P, C, t\} \\ \{e \circ P, \mathbf{Par}; C, t\} & \underset{\mathbf{Par}}{\rightarrow} & \{e \circ P, C, t\} \\ \{e \circ P, \mathbf{Delpar}; C, t\} & \underset{\mathbf{Delpar}}{\rightarrow} & \{e \circ P, C, t\} \end{array}$$

Nous notons $\underset{E^{op}}{\rightarrow}$ cette fonction de transition et nous avons le résultat suivant :

Proposition 4 (*Correction de la BSP-CAM optimisée*)

Si $S_1 \overset{*}{\underset{E}{\rightarrow}} S_n$ alors $S_1 \overset{*}{\underset{E^{op}}{\rightarrow}} S_n$.

Preuve . Immédiate par induction sur la dérivation et par cas sur les instructions. ■

Notons que le résultat contraire n'est pas vrai car la BSP-CAM optimisée peut réduire des termes non valides comme l'imbrication de vecteurs. Par contre nous avons :

[4] Le papier [R3] propose une telle analyse statique.

[5] qui pourraient être ensuite supprimées par le compilatateur OCaml lors de la phase d'optimisation de code.

Proposition 5 (*Complétude de la mini BSP-CAM optimisée*)

Si $\mathcal{A}(e)$ *et si* $S_1 = \mathcal{C}(e)$ *et si* $S_1 \underset{E^{op}}{\overset{*}{\rightarrow}} S_n$ *alors* $S_1 \underset{E}{\overset{*}{\rightarrow}} S_n$.

Preuve . Triviale par induction sur la dérivation, par cas sur les instructions et utilisant la proposition 3 qui stipule que l'évaluation des instructions de la BSP-CAM est correcte par rapport à l'évaluation de e. ■

Ce schéma de compilation (après suppression des fonctions d'identité et du constructeur **Par** devenu inutile) correspond à l'implantation de nos primitives en OCaml qui sera définie dans le prochain chapitre.

4.A Annexe, preuves des conditions

Pour la preuve de correction, nous allons utiliser des termes distribués et non des simples termes. La sémantique à «petits pas» étant équivalente à la sémantique distribuée (théorème 7), cela ne pose pas de problème.

Lemme 28

Soit un terme projeté e, alors $\overline{\overline{[\![e]\!]}} = e$

Preuve. Par induction sur le terme projeté e.

- $e = \overline{n}$; $[\![e]\!] = \mathbf{Fst}$; $[\![\overline{n-1}]\!] = \cdots = \mathbf{Fst}^{n-1}$; $\mathbf{Snd}$. Donc $\overline{\overline{[\![\overline{n}]\!]}} = \overline{\overline{\mathbf{Fst}^{n-1}; \mathbf{Snd}}} = \overline{n}$
- $e = \mathbf{c}$; $\overline{\overline{[\![e]\!]}} = \overline{\overline{\mathbf{Cst}(c)}} = \mathbf{c}$
- $e = \mathbf{op}$; $\overline{\overline{[\![e]\!]}} = \overline{\overline{\mathbf{Cst}(op)}} = \mathbf{op}$
- $e = \lambda.e'$; $\overline{\overline{[\![\lambda.e']\!]}} = \overline{\overline{\mathbf{Closure}([\![t']\!])}} = \lambda.\overline{\overline{[\![t']\!]}} = \lambda.t'$
- $e = (e_1\ e_2)$; $\overline{\overline{[\![(e_1\ e_2)]\!]}} = \overline{\overline{\mathbf{Push}; [\![e_1]\!]; \mathbf{Swap}; [\![e_2]\!]; \mathbf{Cons}; \mathbf{App}}} = (\overline{\overline{[\![e_2]\!]}}\ \overline{\overline{[\![e_1]\!]}}) = (e_1\ e_2)$
- $e = (e_1, e_2)$; $\overline{\overline{[\![(e_1, e_2)]\!]}} = \overline{\overline{\mathbf{Push}; [\![e_1]\!]; \mathbf{Swap}; [\![e_2]\!]; \mathbf{Cons}}} = (\overline{\overline{[\![e_2]\!]}}, \overline{\overline{[\![e_1]\!]}}) = (e_1, e_2)$
- $e = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$; $\overline{\overline{[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]}} = \overline{\overline{\mathbf{Push}; [\![e_1]\!]; \mathbf{Case}([\![e_2]\!], [\![e_3]\!])}}$
  $= \mathbf{if}\ \overline{\overline{[\![e_1]\!]}}\ \mathbf{then}\ \overline{\overline{[\![e_2]\!]}}\ \mathbf{else}\ \overline{\overline{[\![e_3]\!]}} = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$
- $e = (\mathbf{mkpar}\ e')$; $\overline{\overline{[\![(\mathbf{mkpar}\ e')]\!]}} = \overline{\overline{\mathbf{Push}; \mathbf{Closure}(\mathbf{Loc}; \cdots; \mathbf{Par}); \mathbf{Swap}; [\![e]\!]; \mathbf{Cons}; \mathbf{App}}} =$
  $(\mathbf{mkpar}\ \overline{\overline{[\![e]\!]}}) = (\mathbf{mkpar}\ e')$
- $e = (\mathbf{apply}\ e_1\ e_2)$; $\overline{\overline{[\![\mathbf{apply}\ e_1\ e_2]\!]}}$
  $= \overline{\overline{\mathbf{Push}; \mathbf{Push}; \mathbf{Closure}(\mathbf{Closure}(\mathbf{Loc}; \cdots \mathbf{App}; \mathbf{Par})); \cdots; [\![e1]\!]; \mathbf{Cons}; \mathbf{App}}}$
  $= (\mathbf{apply}\ \overline{\overline{[\![e_1]\!]}}\ \overline{\overline{[\![e_2]\!]}}) = (\mathbf{apply}\ e_1\ e_2)$
- cas $e = (\mathbf{put}\ e')$ et $(\mathbf{proj}\ e')$; de manière similaire. ■

Lemme 29 (*Condition de l'état initial*)

La BSP-CAM munie de $\underset{E}{\rightarrow}$, de $\mathcal{C}$ et de $\mathcal{D}$ vérifie la condition de l'état initial.

Preuve.

$\mathcal{D}(\mathcal{C}(e)) = \mathcal{D}(\langle\!\langle(\bullet, [\![e]\!], glo), \cdots, (\bullet, [\![e]\!], glo)\rangle\!\rangle) = \langle\!\langle\mathcal{D}(\{\bullet, [\![e]\!], glo\}, \cdots, \mathcal{D}(\{\bullet, [\![e]\!], glo\})\rangle\!\rangle =$
$\langle\!\langle\overline{\overline{[\![e]\!]}}[\overline{\bullet}], \cdots, \overline{\overline{[\![e]\!]}}[\overline{\bullet}]\rangle\!\rangle = \langle\!\langle e[\bullet], \cdots, e[\bullet]\rangle\!\rangle$ ■

Lemme 30 (*Condition des états accessibles*)

La BSP-CAM munie de $\underset{E}{\rightarrow}$, de $\mathcal{C}$ et de $\mathcal{D}$ vérifie la condition des états accessibles.

Preuve. Il y a deux sortes de réduction de la BSP-CAM : une réduction asynchrone et une réduction synchrone.

Le cas d'une réduction asynchrone est la réduction d'une CAM de la BSP-CAM. Nous avons à faire la preuve pas cas sur l'instruction de tête de cette CAM. Nous avons, par exemple, les diagrammes de transition suivant :

$$\begin{array}{ccc}
\{(e,f)\circ P, \mathbf{Snd}; C, t\} & \xrightarrow[\mathbf{Snd}]{} & \{f\circ P, C, t\} \\
\downarrow \Phi & & \downarrow \Phi \\
(\mathbf{Snd}[\overline{\overline{(e,f)}}], \{P, C, t\}) & & (\overline{\overline{f}}, \{P, C, t\}) \\
\downarrow \Psi & & \downarrow \Psi \\
\overline{1}[\overline{\overline{f}}\circ\overline{\overline{e}}] & \rightsquigarrow & \overline{\overline{f}}
\end{array}$$

$$\begin{array}{ccc}
\{(e,f)\circ P, \mathbf{Fst}; \mathbf{Fst}^{n-1}; \mathbf{Snd}; C, t\} & \xrightarrow[\mathbf{Fst}]{} & \{e\circ P, \mathbf{Fst}^{n-1}; \mathbf{Snd}; C, t\} \\
\downarrow \Phi & & \downarrow \Phi \\
(\overline{\overline{\mathbf{Fst}^n; \mathbf{Snd}}}[\overline{\overline{(e,f)}}], \{P, C, t\}) & & (\overline{\overline{\mathbf{Fst}^n; \mathbf{Snd}}}[\overline{\overline{e}}] \\
\downarrow \Psi & & \downarrow \Psi \\
\overline{n+1}[\overline{\overline{f}}\circ\overline{\overline{e}}] & \rightsquigarrow & \overline{n}[\overline{\overline{e}}]
\end{array}$$

$$\begin{array}{ccc}
\{e\circ P, \mathbf{Closure}(C'); C, t\} & \xrightarrow[\mathbf{Closure}]{} & \{[C', e]\circ P, C, t\} \\
\downarrow \Phi & & \downarrow \Phi \\
([\overline{\overline{\mathbf{Closure}(C')[e]}}, \{P, C, t\}) & & (\overline{\overline{C'[e]}}, \{P, C, t\}) \\
\downarrow \Psi & & \downarrow \Psi \\
\lambda.\overline{\overline{C'}}[\overline{\overline{e}}] & \rightsquigarrow & \overline{\lambda.\overline{\overline{C'}}[\overline{\overline{e}}]}
\end{array}$$

Les autres cas sont similaires.

Dans le cas d'une réduction synchrone, chaque CAM i de la BSP-CAM est de la forme

$$\langle\!\langle([v_i^0, \ldots, v_i^{p-1}]\circ P_i, \mathbf{SEND}; C_i, glo)\rangle\!\rangle$$

La décompilation de cette BSP-CAM donne le terme : $\langle\!\langle e_0, \ldots, e_{p-1}\rangle\!\rangle$ tel que $e_i = \Gamma[\mathbf{send}\ \langle[v_0^i, \ldots, v_{p-1}^i]\rangle]$. Le résulat de l'évaluation de la BSP-CAM est :

$$\langle\!\langle([v_0^i, \ldots, v_{p-1}^i]\circ P_i, C_i, glo)\rangle\!\rangle$$

qui se décompile en : $\langle\!\langle e'_0, \ldots, e'_{p-1}\rangle\!\rangle$ tel que $e_i = \Gamma[\mathbf{send}\ \langle[v_i^0, \ldots, v_i^{p-1}]\rangle]$. Ce qui est bien le résultat attendu. ■

Lemme 31 (*Condition de progression*)

La BSP-CAM munie de $\underset{E}{\rightarrow}$, de $\mathcal{C}$ et de $\mathcal{D}$ vérifie la condition de progrès.

Preuve. Il est facile de constater qu'une suite d'instructions ne comportant pas l'instruction $\mathbf{App}$ se termine toujours (la transition consumant l'instruction). Ainsi, comme la transition de l'instruction $\mathbf{App}$ n'est pas silencieuse, il ne peut y avoir une suite infinie d'instructions silencieuses. ■

Lemme 32 (*Condition de l'état terminal*)

La BSP-CAM munie de $\underset{E}{\rightarrow}$, de $\mathcal{C}$ et de $\mathcal{D}$ vérifie la condition de l'état terminal.

Preuve. Il y a quatre sortes d'états susceptibles de provoquer l'arrêt de la machine :

1. Les codes sont vides. C'était donc des états accessibles et d'après le lemme 30, ces états sont simulables. Ils se traduisent donc de la forme $\{v \circ \bullet, \varepsilon, t\}$. Or dans ce cas, ils se traduisent par $\overline{\overline{v}} = v$ qui sont donc des valeurs.
2. Une instruction courante est $\mathbf{Fst}$ ou $\mathbf{Snd}$. L'expression n'ayant pas de variables libre, ce cas est impossible.
3. Le test, d'une instruction courante $\mathbf{Loc}$ est faux. Dans ce cas, la décompilation donne un terme qui n'est pas réductible car son test sera aussi faux et qui n'est donc pas une valeur.
4. Un des états de la BSP-CAM est aussi terminal quand la forme de la tête de pile ne correspond pas à ce qui est attendu pour une des instructions courantes. Cette état est donc accessible, donc d'après le lemme 30, il doit être décompilable. Or par définition de la décompilation, cette état n'est pas décompilable. Ces cas ne se produisent donc pas pour des états accessibles. ■

5 Bibliothèque de programmes BSML

Une version préliminaire d'une partie de ce chapitre a fait l'objet d'un article [C1] écrit en collaboration avec Frédéric Loulergue et David Billiet.

Sommaire

DANS ce chapitre nous présentons un ensemble de fonctions BSML pour des opérations BSP classiques et apparaissant régulièrement dans les algorithmes BSP. Cet ensemble peut être vu comme une bibliothèque standard de la BSMLlib. Nous présentons aussi l'implantation modulaire de la BSMLlib[1] et, des permiers tests de performances de nos fonctions comparées aux prédictions BSP.

5.1 Fonctions BSP classiques avec la BSMLlib

Les primitives décrites dans la section précédente constituent le noyau de la bibliothèque BSMLlib. Elles sont suffisantes pour exprimer tous les algorithmes BSP. Malgré leur caractère universel, il est souhaitable d'avoir un ensemble de fonctions dites utilitaires pour simplifier la programmation de ces algorithmes et rendre le code plus lisible. Toutes les fonctions décrites dans cette section sont ainsi implantées en utilisant uniquement ces primitives.

5.1.1 Exemples de fonctions asynchrones

La fonction replicate crée un vecteur parallèle contenant en chaque composante la même valeur :

```
(* replicate: α →α par *)
let replicate v = mkpar(fun _ →v)
```

où plus schématiquement : replicate v = | v | ··· | v |

La primitive **apply** est utilisée pour un vecteur parallèle de fonctions d'arité unaire. Pour facilement manipuler des vecteurs de fonctions dont les arités ne sont pas unaires, nous définissons une série de fonctions d'applications parallèles et asynchrones :

```
(* apply2: (α →β →γ )→α par→β par→γ par *)
let apply2 vf v1 v2 = apply (apply vf v1) v2
```

[1] provenant du travail sur les sémantiques du chapitre 3 et en partie du travail sur les instructions BSP de la machine abstraite du chapitre 4.

```
(* apply3: (α →β →γ → δ )→α par→β par→γ par→ δ par *)
let apply3 vf v1 v2 v3 = apply (apply (apply vf v1) v2) v3
```

Par exemple :

apply2 $\boxed{f_0 \mid \cdots \mid f_{p-1}}$ $\boxed{v_0 \mid \cdots \mid v_{p-1}}$ $\boxed{v'_0 \mid \cdots \mid v'_{p-1}} = \boxed{(f_0\ v_0\ v'_0) \mid \cdots \mid (f_{p-1}\ v_{p-1}\ v'_{p-1})}$

Nous utiliserons aussi couramment l'application d'une même fonction séquentielle en chaque composante d'un vecteur parallèle, c'est-à-dire l'application de la même fonction en chaque processeur. Pour cela, nous définissons une série de fonctions parfun d'applications parallèles d'une même fonction dont seules les arités diffèrent :

```
(* parfun: (α →β )→α par→β par *)
let parfun f v = apply (replicate f) v

(* parfun2: (α →β →γ )→α par→β par→γ par *)
let parfun2 f v1 v2 = apply (parfun f v1) v2

(* parfun3: (α →β →γ → δ )→α par→β par→γ par→ δ par *)
let parfun3 f v1 v2 v3 = apply (parfun2 f v1 v2) v3
```

Par exemple : parfun f $\boxed{v_0 \mid \cdots \mid v_{p-1}} = \boxed{(\mathsf{f}\ v_0) \mid \cdots \mid (\mathsf{f}\ v_{p-1})}$

Il est aussi courant d'appliquer une fonction différente en un processeur donné. applyat $n\ f_1\ f_2\ v$ applique la fonction f_1 au processeur n et la fonction f_2 aux autres processeurs :

```
(* applyat: int→(α →β )→(α →β )→α par→β par *)
let applyat n f1 f2 v = apply (mkpar (fun i→if i=n then f1 else f2)) v
```

De manière schématique :

applyat n f_1 f_2 $\boxed{v_0 \mid \cdots \mid v_n \mid \cdots \mid v_{p-1}} = \boxed{(\mathsf{f}_2\ v_0) \mid \cdots \mid (\mathsf{f}_1\ v_n) \mid \cdots \mid (\mathsf{f}_2\ v_{p-1})}$

Les fonctions décrites ci-dessous sont des fonctions couramment utilisées pour exprimer les phases de communication et de synchronisation d'un algorithme BSP. Il paraît alors nécessaire de donner aussi leurs formules de coût.

Dans les sections qui vont suivre, tous les tests de performances ont été effectués sur une grappe de 6 nœuds chacun doté de 1Go de RAM. Les nœuds sont des Intel pentium IV 2.8 Ghz avec des cartes Gigabit Ethernet et interconnectés par un réseau Gigabit Ethernet (10/100/1000). Une Mandrake Clic 2.0 a été utilisée comme système d'exploitation et les programme ont été compilés avec OCaml 3.08.2 en mode natif. Chacun des programmes a été exécuté 100 fois de suite et la moyenne des exécutions a été prise pour les graphiques. Les versions de la BSMLlib utilisant la PUB [40], MPI ou directement les fonctionnalités TCP/IP d'OCaml ont été utilisées. Nous nous sommes servis de la version TCP/IP de la PUB ainsi que de la version MPICH de MPI.

5.1.2 Fonctions d'échange total

Notre premier exemple est un échange total *répliqué*, c'est-à-dire que le résultat sera une liste répliquée, en chaque processeur, des valeurs des composantes du vecteur parallèle. Ainsi, chaque processeur contient une valeur et le résultat de (rpl_total $\boxed{v_0 \mid \cdots \mid v_{p-1}}$) est $[v_0, \ldots, v_{p-1}]$ la liste de ces valeurs :

```
(* rpl_total: α par→α list *)
let rpl_total vec =
 let rpl_totex vec = compose noSome (proj (parfun (fun v →Some v) vec)) in
  List.map (rpl_totex vec) (procs())
```

$$\text{avec} \begin{cases} \text{compose: } (\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta \\ \text{compose } f\ g\ x = (f\ (g\ x)) \\ \text{noSome : } \alpha\ \text{option} \to \alpha \\ \text{noSome (Some } v) = v \\ \text{List.map: } (\alpha \to \beta) \to \alpha\ \text{list} \to \beta\ \text{list} \\ \text{List.map } f\ [v_0; \ldots; v_n] = [(f\ v_0); \ldots; (f\ v_n)] \\ \text{procs:unit} \to \text{int list} \\ \text{procs()} = [0; 1; \ldots; \textbf{bsp\_p}()\text{-}1]. \end{cases}$$

Le coût BSP est $(p-1) \times s \times g + l$ où s est la taille en octet de la plus grande des valeurs contenues par les processeurs.

rpl_total est une fonction qui peut être utilisée pour définir d'autres fonctions (que nous utiliserons massivement pour l'implantation des structures de données parallèles, cf chapitre 8) telles que parfun_total, qui permet d'appliquer une fonction séquentielle à chaque composante d'un vecteur parallèle, fait un échange total de ces valeurs et enfin applique une autre fonction sur ce résultat :

$$\text{parfun\_total } f_1\ f_2 \boxed{v_0 \mid \cdots \mid v_{p-1}} \Rightarrow (f_2\ [v'_0; \ldots; v'_{p-1}])$$

avec $(f_1\ v_i) = v'_i$. Cette fonction est codée comme suit :

```
(* parfun_total : (α →β )→(β list→γ )→α par→γ  *)
let parfun_total f1 f2 par_v = f2 (rpl_total (parfun f1 par_v))
```

Nous pouvons aussi programmer un échange total avec un **put**, mais avec comme résultat un vecteur de listes contenant les valeurs initiales :

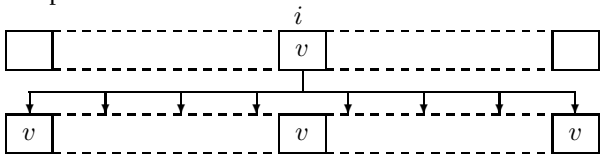
```
(* totex: α par→(int→α )par *)
let totex vv = parfun (compose noSome) (put (parfun (fun v dst→Some v) vv))

(* total_exchange: α par→α list par *)
let total_exchange vec = parfun2 List.map (totex vec) (replicate (procs()))
```

C'est-à-dire : total_exchange $\boxed{v_0 \mid \cdots \mid v_{p-1}} = \boxed{[v_0, \ldots, v_{p-1}] \mid \cdots \mid [v_0, \ldots, v_{p-1}]}$.

La figure 5.1 montre les performances mesurées avec en chaque processeur une liste d'entiers (chacune de la même longueur). Sur les deux graphiques, les longueurs de ces listes vont croissant.

5.1.3 Fonctions pour la diffusion d'une valeur

Notre second exemple est la programmation avec la BSMLlib de la diffusion d'une valeur d'un processeur aux autres processeurs. Ceci peut être schématisé comme suit :



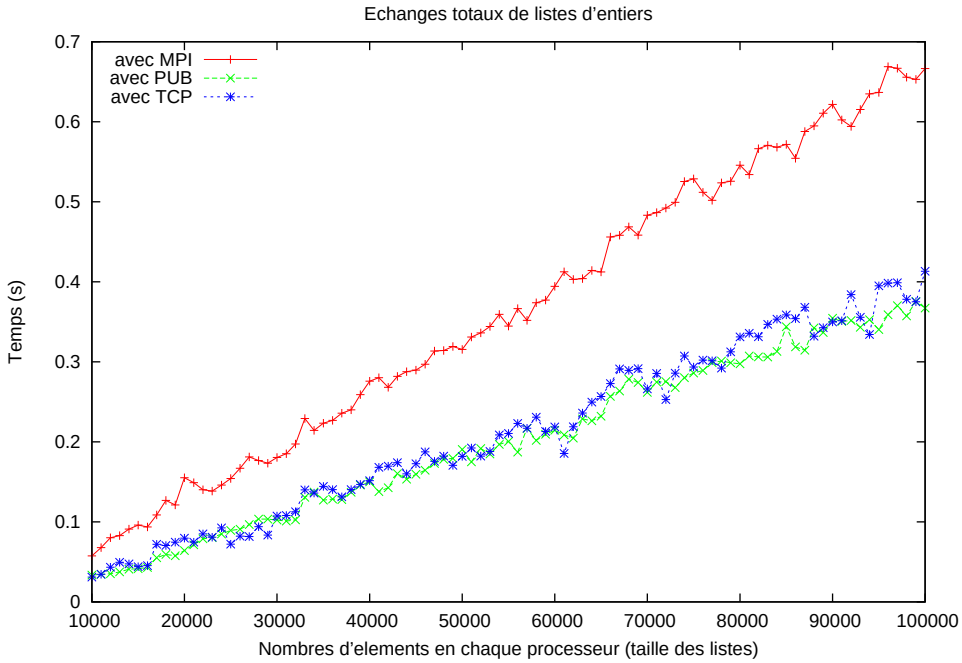
où le processeur i envoie sa valeur v aux autres processeurs. Ce travail peut être effectué en une seule super-étape avec le code suivant :

```
(* bcast_direct: int→α par→α par *)
let bcast_direct root vv =
 if not (within_bounds root) then raise Bcast else
  let mkmsg = applyat root (fun v dst→Some v) (fun _ dst →None) vv in
    parfun noSome (apply (put mkmsg) (replicate root))
```

auquel nous pouvons donner la formule de coût suivante :

$$(p-1) \times \mathcal{S}(v_i) \times g + l$$

où $\mathcal{S}(v_i)$ est la taille en octets de v_i, la valeur diffusée. Notons qu'une diffusion directe peut aussi se coder en utilisant la primitive de projection globale :



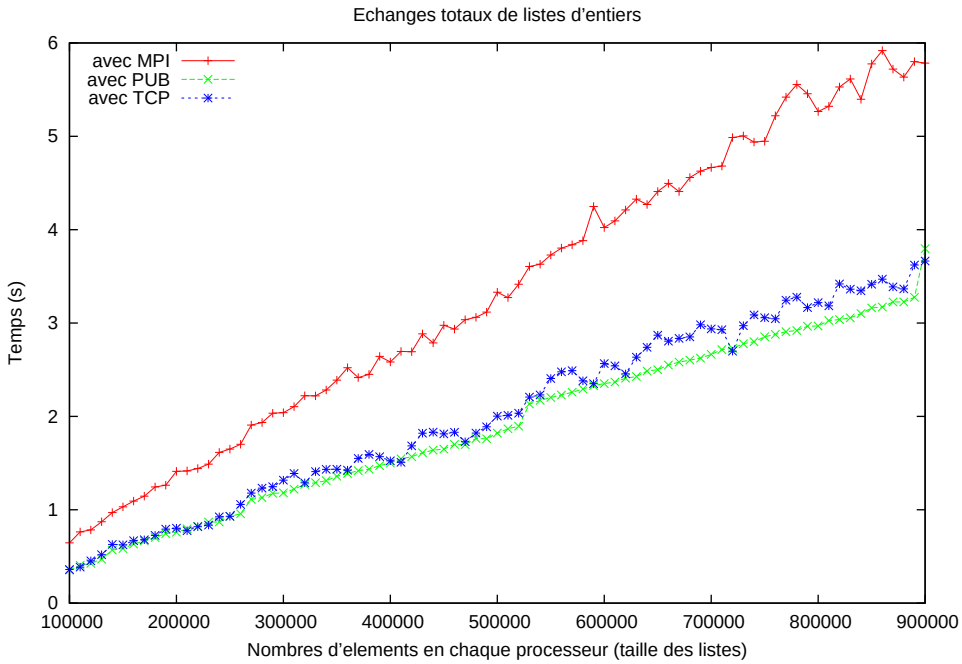


Figure 5.1 — Performances des échanges totaux

```
(* bcast_direct_rpl: int→α par→α *)
let bcast_direct_rpl root vv =
 if not (within_bounds root) then raise Bcast else
  let fmsg=proj (applyat root (fun v →Some v) (fun _ →None) vv) in
   (noSome (fmsg root))
```

Plus schématiquement : bcast_direct_rpl i $\boxed{v_0 \mid \cdots \mid v_i \mid \cdots \mid v_{p-1}} = v_i$

La figure 5.2 montre les performances mesurées avec au processeur 0 (processeur émetteur) une liste d'entiers. Sur les deux graphiques, la longueur de cette liste va de manière croissante.

Quand la taille de la valeur à diffuser est importante, et suivant les paramètres BSP de la machine, l'algorithme de diffusion en deux phases décrit dans [34, 117] peut s'avérer plus efficace. La figure 5.3 illustre la méthode utilisée. L'émission en deux super-étapes procède commme suit : le processeur de départ «découpe» son message en p messages et envoie chacun d'eux aux $p-1$ autres processeurs (première super-étape). Ensuite, chaque processeur envoie son bout de message aux autres processeurs (échange total) et pour finir chaque processeur «recolle» les morceaux reçus. La formule de coût BSP est alors :

$$2 \times \frac{(p-1) \times \mathcal{S}(v_i)}{p} \times g + 2 \times l + d(v_i) + r(v_i)$$

où $d(v_i)$ est le temps pour découper la valeur v_i en p morceaux et $r(v_i)$ le temps pour les recoller.

Dans un premier temps, nous définissons la première super-étape, celle qui éparpille la valeur du processeur initial. Nous la codons avec la fonction ci-dessous, qui prend en paramètre une fonction définissant le découpage de la valeur à diffuser :

```
(* scatter :(α →int→β option)→int→α par→β par *)
let scatter partition root v =
 if not (within_bounds root) then raise Scatter else
  let mkmsg = applyat root partition (fun _ _ →None) in
   parfun noSome (apply (put (mkmsg v)) (replicate root))
```

La figure 5.4 montre les performances mesurées avec au processeur 0 (processeur émetteur) une liste d'entiers. Sur les deux graphiques, la taille de cette liste va de manière croissante.

Ensuite, nous pouvons implanter une version générique de cet algorithme de diffusion, générique car la fonction de «découpage» et de «recollage» sont les premiers paramètres de la fonction :

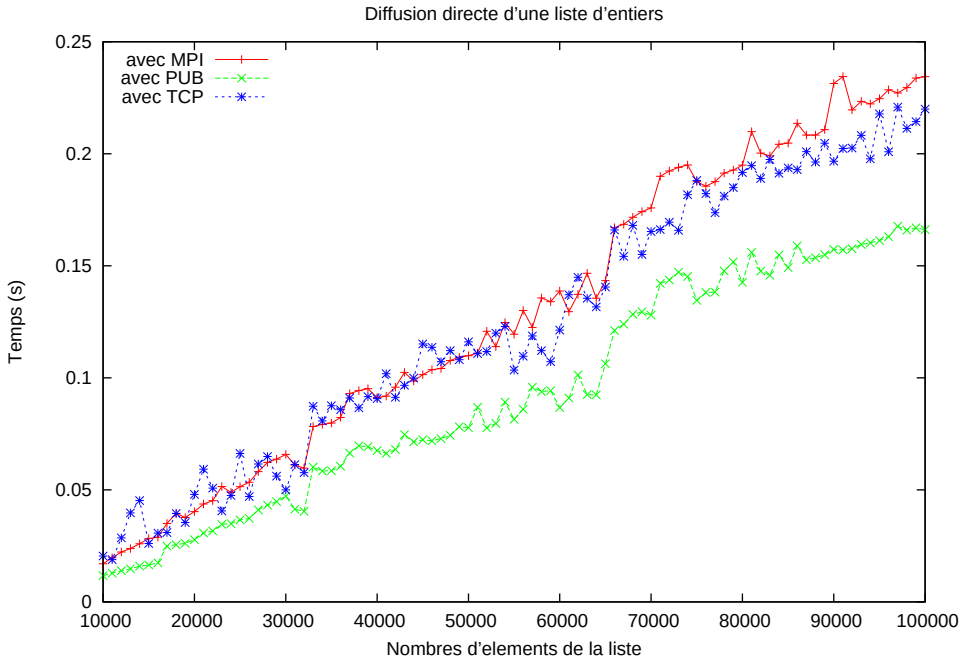
```
(* bcast_totex_gen: (α →int→β option)→((int→β )→γ )→int→α par→γ par *)
let bcast_totex_gen partition paste root vv =
 if not (within_bounds root) then raise Bcast else
  let phase1 = scatter partition root vv in
  let phase2 = totex phase1 in
   parfun paste phase2
```

Celle-ci pouvant aisément être spécialisée comme par exemple pour les listes :

```
let bcast_totex_list root vl =
  let paste f = List.flatten (List.map f (procs())) in
    bcast_totex_gen cut_list paste root vl
```

avec
- List.flatten:α list list $\rightarrow \alpha$ list
- List.flatten $[l_0; \cdots; l_n] = [l_0 @ \cdots @ l_n]$
- cut_list: α list→int→α list option
- cut_list $[e_0; \cdots; e_n]\ i$ = (Some $[e_{i-j}; \cdots; e_{i+j}]$) où $j = n \bmod p$

La figure 5.5 montre les performances mesurées avec au processeur 0 (processeur émetteur) une liste d'entiers. Sur les deux graphiques, la longueur de cette liste va de manière croissante. Sur notre cluster de test, les performances de la diffusion en deux phases sont moins bonnes que celles de la diffusion directe. Cela est dû au temps de «recollage» des listes qui est proportionel à la longueur de la liste du processeur émetteur. Pour contourner ce problème, nous pourrions utiliser des tableaux, permettant ainsi un «découpage» (ainsi qu'un «recollage») en un temps proportionel à p.



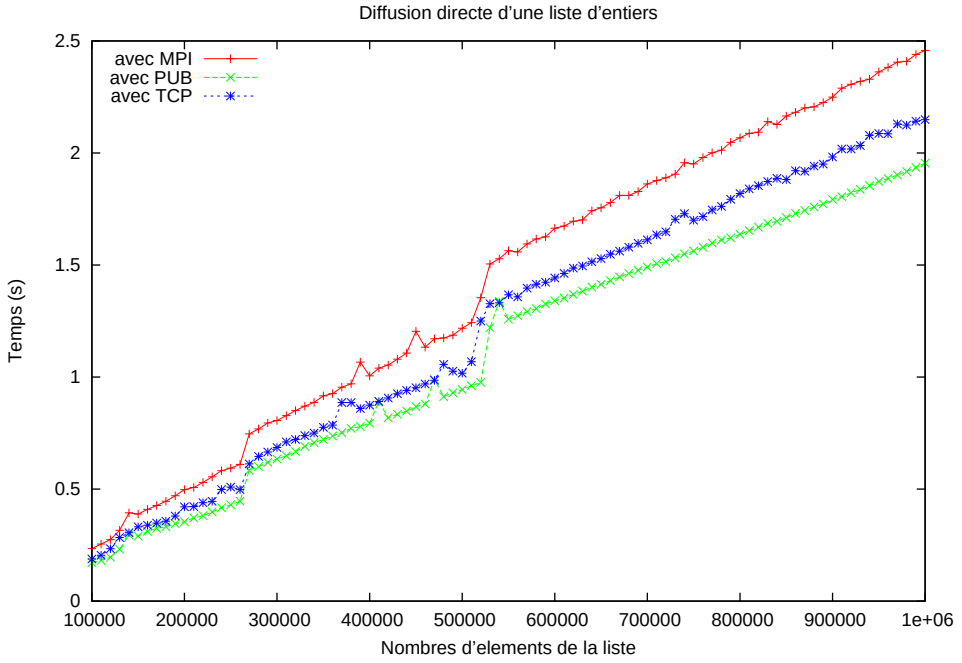


Figure 5.2 — Performances de la diffusion directe

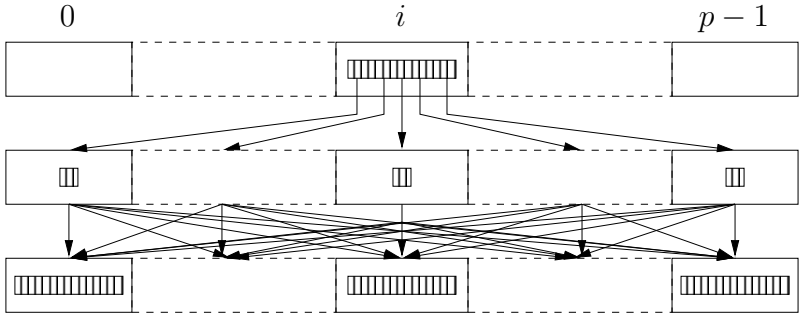


Figure 5.3 — Diffusion en 2 phases (aussi appelée avec échange total)

5.1.4 Fonctions pour le calcul des préfixes

Notre troisième exemple est le calcul parallèle classique des préfixes d'un ensemble de valeurs avec un opérateur associatif admettant un élément neutre. Pour représenter cet ensemble, nous faisons l'hypothèse que chaque processeur en mémorise une partie sous la forme d'une liste d'éléments. Nous avons donc un vecteur parallèle de listes d'éléments. Prenons pour exemple, l'expression suivante pour une machine à 3 processeurs (avec $e = 1$ comme élément neutre) permettant de calculer les factorielles de 1 à 5 :

scan_list e ($\times$) | $[1;2]$ | $[3;4]$ | $[5]$ |

Cette expression sera évaluée en :

| $[e \times 1; e \times 1 \times 2]$ | $[e \times 1 \times 2 \times 3; e \times 1 \times 2 \times 3 \times 4]$ | $[e \times 1 \times 2 \times 3 \times 4 \times 5]$ |

$\Rightarrow$ | $[1;2]$ | $[6;24]$ | $[120]$ |

Chaque processeur effectue une réduction locale avec les éléments de sa propre liste, puis envoie son résultat partiel aux processeurs qui le suivent (dans l'ordre des *pid* de la machine) et finalement réduit ce résultat avec les valeurs envoyées par les processeurs qui le précèdent.

Pour ce calcul, nous avons donc besoin, tout d'abord, d'une fonction pour la réduction parallèle, c'est-à-dire réduire, *via* un opérateur, les valeurs d'un vecteur parallèle. Ceci équivaut à calculer au processeur i, la réduction des valeurs contenue par les processeurs j tel que $j < i$. Prenons par exemple, l'expression suivante :

scan e ($\times$) | 1 | 2 | 3 |

Celle-ci sera évaluée en :

| e | 1×2 | $1 \times 2 \times 3$ |

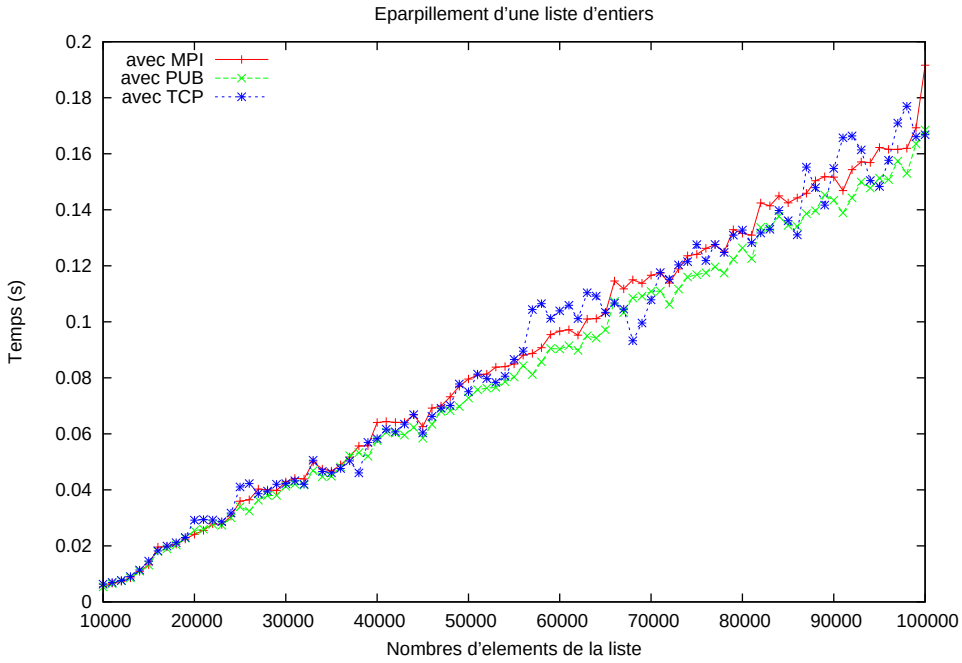
$\Rightarrow$ | 1 | 2 | 6 |

La figure 5.6 donne le code BSML pour une telle fonction en utilisant l'algorithme BSP direct [34]. Nous employons les fonctions suivantes :

$$\begin{cases} \text{List.fold\_left: } (\alpha \to \beta \to \alpha) \to \alpha \to \beta \text{ list} \to \alpha \\ \text{List.fold\_left } f\ e\ [v_0; \ldots; v_n] = f\ (\cdots (f\ (f\ e\ v_0)\ v_1) \cdots)\ v_n \\ \text{from\_to:int} \to \text{int} \to \text{int list} \\ \text{from\_to } n\ m = [n; n+1; n+2; \ldots; m]. \end{cases}$$

Ensuite, nous pouvons obtenir une version générique pour le calcul des préfixes en utilisant les fonctions suivantes :

```
(* scan_wide :((α →β →β )→γ → δ par→α par)→((α →β →β )→γ → ϵ → δ ∗ ζ )→
              ((β →β )→ ζ → η )→(α →β →β )→γ → ϵ par→ η par ∗)
let scan_wide scan seq_scan_last map op e vl =
 let local_scan=parfun (seq_scan_last op e) vl in
 let last_elements=parfun fst local_scan in
 let values_to_add=(scan op e last_elements) in
```



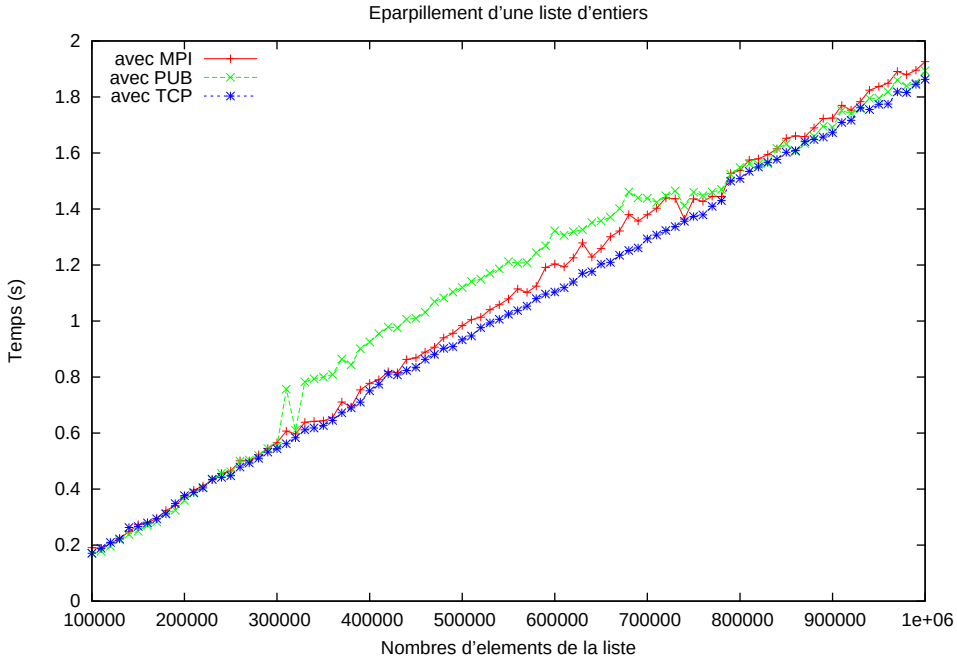
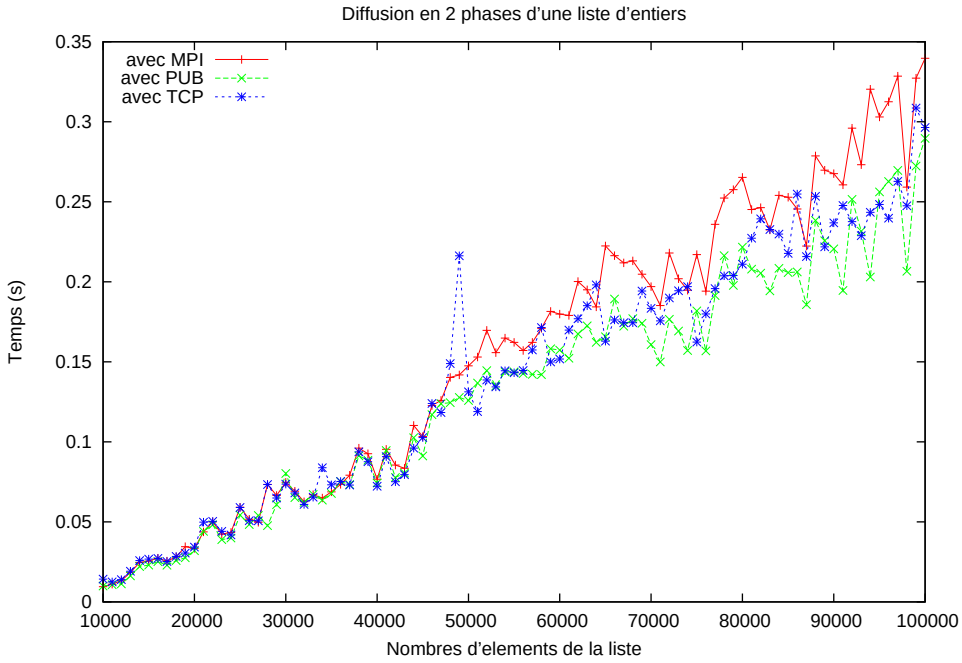


Figure 5.4 — Performances de l'éparpillement d'une liste d'entiers



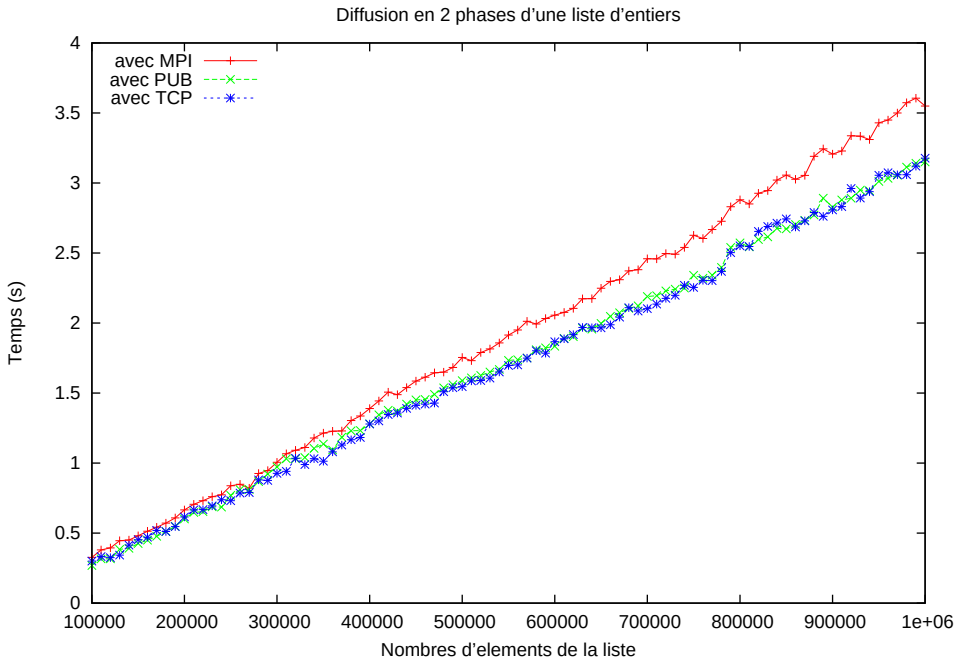


Figure 5.5 — Performances de la diffusion en 2 phases d'une liste d'entiers

```
(* scan_direct: (α →β →α )→α →β par→α par *)
let scan_direct op e vv =
 let mkmsg pid v dst=if dst<pid then None else Some v in
 let procs_lists=mkpar (fun pid→from_to 0 pid) in
 let rcv_msgs=put (apply (mkpar mkmsg) vv) in
 let values_lists= parfun2 List.map (parfun (compose noSome) rcv_msgs) procs_lists in
  applyat 0 (fun _ →e) (List.fold_left op e) values_lists
```

Figure 5.6 — Code de l'algorithme direct pour la réduction parallèle

```
 let pop=applyat 0 (fun _ y→y) op in
  parfun2 map (pop values_to_add) (parfun snd local_scan)

(* seq_scan_last: (α →β →α )→α →β list→α *α list *)
let seq_scan_last op e l =
  let rec seq_scan' last l accu =
      match l with
            [] →(last,[last])
    | [hd] →(last,(op last hd)::accu)
    | hd::tl →(let new_last = (op last hd)
                  in seq_scan' new_last tl (new_last::accu)) in
  seq_scan' e l []

(* scan_list : ((α →α →α )→α →α par→α par)→(α →α →α )→α →α list par→α list par *)
let scan_list scan op e vl = scan_wide scan seq_scan_last List.rev_map op e vl
```

tel que seq_scan_last $f\ e\ [v_0; v_1; \ldots; v_n] = (\text{last}, [(f\ \text{last}\ v_n); \cdots; (f(f\ e\ v_0)\ v_1); (f\ e\ v_0)])$ avec last $= (f\ (\cdots (f\ (f\ e\ v_0)\ v_1)\cdots)\ v_{n-1})$. Nous pouvons alors composer les fonctions définies ci-dessus pour obtenir notre fonction :

```
(* scan_list_direct:(α →α →α )→α →α list par→α list par *)
let scan_list_direct op e vl = scan_list scan_direct op e vl
```

La formule de coût BSP d'une telle fonction (en faisant l'hypothèse que op a un coût constant c_{op}) est :

$$2 \times N \times c_{op} \times r\ + (p-1) \times s \times g\ +\ l$$

telle que s est la taille en octets d'un élément et N est la longueur de la plus longue liste contenue par un processeur. Nous avons donc la somme des coûts pour calculer la réduction partielle, échanger ces résultats et finir la réduction avec les éléments échangés.

La figure 5.7 montre les performances mesurées avec en chaque processeur une liste de flottants (chacune de même longueur). La somme de flottants a été utilisée comme opérateur. Sur les deux graphiques, les longueurs de ces listes vont croissant.

Il est aussi possible de calculer les préfixes en $\log_2(p) + 1$ super-étapes [34]. La figure 5.8 donne le code BSML pour une telle fonction. Nous pouvons alors recomposer les fonctions définies ci-dessus pour avoir une nouvelle version du calcul des préfixes :

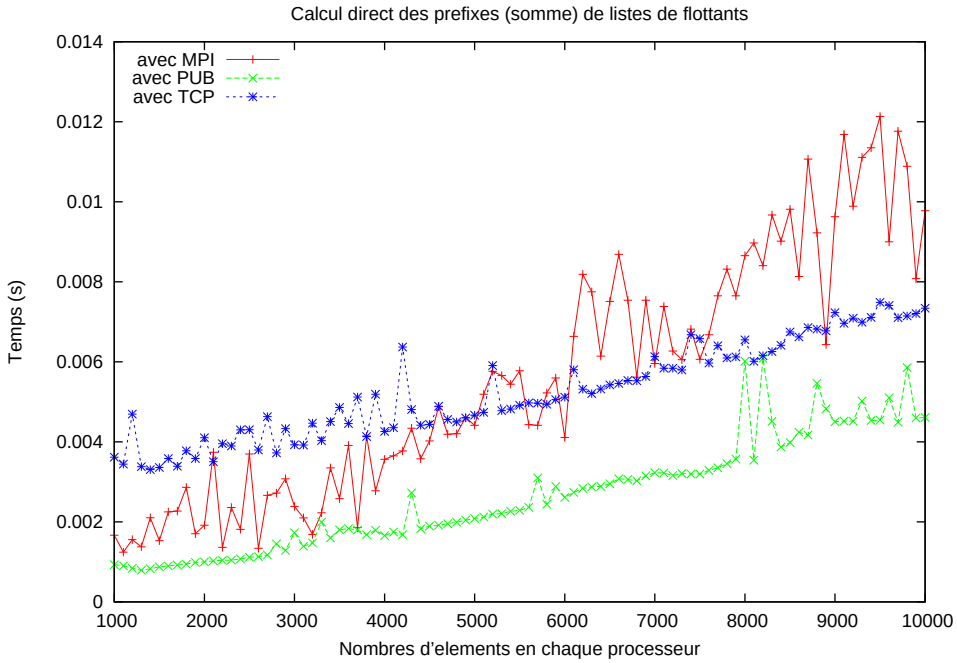
```
(* scan_list_direct:(α →α →α )→α →α list par→α list par *)
let scan_list_logp op e vl = scan_list scan_logp op e vl
```

La formule de coût BSP d'une telle fonction (en faisant l'hypothèse que op a un coût constant c_{op}) est :

$$2 \times N \times c_{op} \times r\ + (\log_2(p) + 1) \times (g + l)$$

avec s qui est la taille en octets d'un élément et N qui est la longueur de la plus longue liste contenue par un processeur. Nous avons donc la somme des coûts pour calculer la réduction partielle, échanger ces résultats en $\log_2(p) + 1$ super-étapes et finir la réduction avec les éléments échangés.

La figure 5.9 montre les performances mesurées avec en chaque processeur une liste de flottants (chacune de même longueur). La somme de flottants a été utilisée comme opérateur. Sur les deux graphiques, les longueurs de ces listes vont croissant.



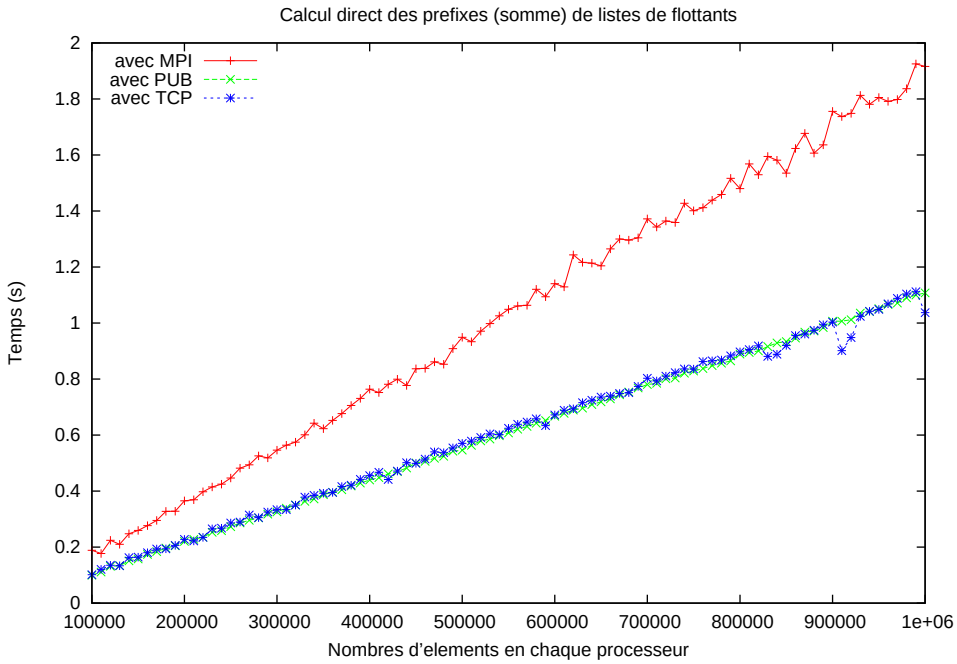


Figure 5.7 — Performances de la réduction parallèle directe de listes

```
(* scanlogp: (α →α →α )→α →α par→α par *)
let scan_logp op e vec =
 let rec scan_aux n vec =
  if n >= (bsp_p()) then (applyat 0 (fun _ →e) (fun x →x) vec) else
   let msg = mkpar (fun pid v dst→
     if ((dst=pid+n)or(pid mod (2*n)=0))&&(within_bounds (dst−n))
      then Some v else None)
   and senders = mkpar (fun pid→natmod (pid−n) (bsp_p()))
   and op' = fun x y→match y with Some y'→op y' x | None →x in
    let vec' = apply (put (apply msg vec)) senders in
    let vec''= parfun2 op' vec vec' in
     scan_aux (n*2) vec'' in
 scan_aux 1 vec
```

Figure 5.8 — Code de l'algorithme binaire pour la réduction parallèle

5.1.5 Exemple de fonctions destructurant un vecteur de listes

Comme expliqué précédemment, la BSMLlib contient une primitive synchrone de projection qui est nécessaire pour tous les algorithmes dont la séquence d'exécution parallèle (c'est-à-dire le comportement global) dépend d'une valeur locale (d'un processeur). Considérons comme quatrième et dernier exemple le problème de convertir une valeur de type α list **par** (un vecteur de listes, c'est-à-dire une liste par processeur) en une valeur de type α **par** list (liste de vecteurs parallèles). Ceci peut être effectué en supprimant un élément en chaque liste jusqu'à ce que l'une d'entre elles soit une liste vide :

```
(* list_of_par: α list par→α par list *)
let rec list_of_par lpar =
 let test=parfun_total (function [] →true | _ →false) (List.fold_left (or) false) lpar in
  if test then []
   else (parfun List.hd lpar)::(list_of_par (parfun List.tl lpar))
```

Dans ce cas, la projection est utilisée comme test de terminaison et nous avons la formule de coût suivante :

$$n \times (4 + p + p \times g + l)$$

où n est la longueur de la plus petite liste contenue par un des processeur. Nous avons bien n super-étapes, avec à chaque fois un *pattern matching* pour tester si la liste est vide ou non, un échange total de ces tests, un «ou» général de ces tests, un List.hd, un List.tl et enfin une concaténation d'un vecteur parallèle dans la liste finale.

Nous pouvons, bien sûr, faire le même travail avec une fonction beaucoup moins coûteuse en nombre de super-étapes, celle-ci calculant d'abord la longueur minimale de ces listes puis construisant directement la liste finale :

```
(* list_of_par': α list par→α par list *)
let list_of_par' lpar =
 let minl=parfun_total List.length (List.fold_left min max_int) lpar in
 let rec take n vl accu =
  if n=0 then (List.rev accu) else
   take (n−1) (parfun List.tl vl) ((parfun List.hd vl)::accu) in
  take minl lpar []
```

et nous avons alors la formule de coût suivante :

$$n_1 + p \times g + l + p + n_2 \times 4$$

où n_1 (resp. n_2) est la plus grande (resp. petite) des longueurs des listes. Nous avons le temps pour calculer ces longueurs, de les échanger, de calculer cette plus petite longueur et enfin, comme ci-dessus, de construire le résultat final.

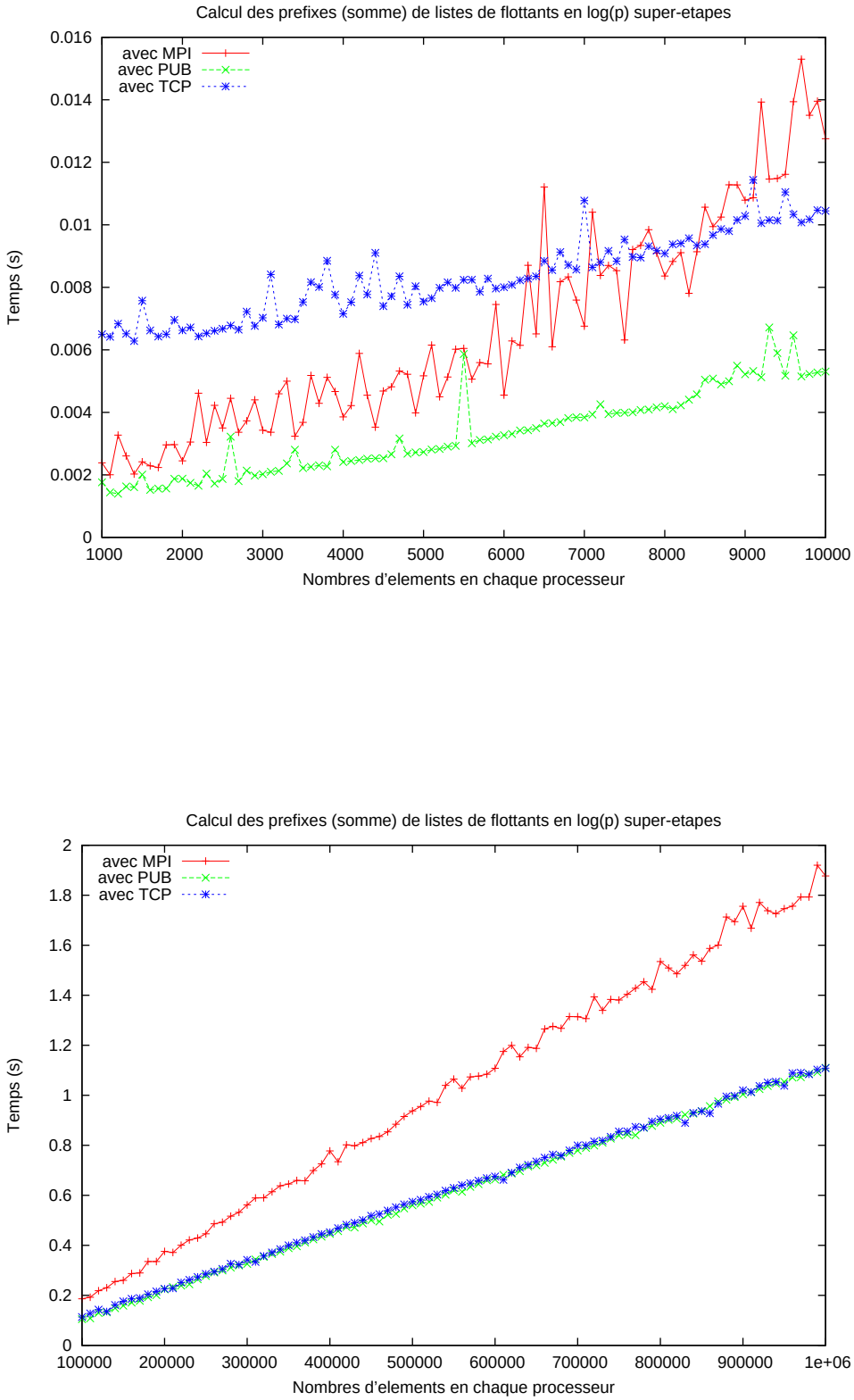


Figure 5.9 — Performances de la réduction parallèle binaire de listes

5.2 Implantation de la bibliothèque BSMLlib

5.2.1 Historique

La première version de la BSMLlib a précédé les calculs [184], et il ne s'agissait alors que d'une simple version séquentielle des primitives où les vecteurs parallèles étaient simulés par des tableaux de taille p. Dans cette première version il n'y avait pas de conditionnelle globale et les primitives de communications étaient un **get** généralisé et un **put**, ces deux primitives opérant sur des vecteurs parallèles de tables de hachage pour éviter que l'ordre des messages ne soit imposé par la structure de données résultante :

get: α **par**→int list **par**→(int $*$ α) Hashtbl.t **par**
put: (int $*$ α) list **par**→(int $*$ α) Hashtbl.t **par**

La première version distribuée de la BSMLlib, la version 0.1 (2000), décrite dans [16] et utilisée dans [135], comprenait en outre la primitive **at** pour laquelle il était précisé que sa seule utilisation valide était d'en faire la condition d'une expression conditionnelle. Cette version 0.1 comprenait une version séquentielle et une version parallèle basée sur la bibliothèque BSPlib (BSP pour C [147]), ainsi qu'un début de bibliothèque standard, comprenant par exemple des diffusions, réductions, *etc.*

Le **put** et le **get** généralisés ont par la suite été formalisés dans [185]. Cette formalisation a suggéré la forme actuelle de la primitive **put**. La fonction **get** généralisée a également été abandonnée en tant que primitive, puisqu'exprimable avec la nouvelle primitive **put**, sans communication et synchronisation supplémentaires. À partir de cette époque, la bibliothèque BSPlib n'a plus été maintenue et la bibliothèque PUB [40] n'était pas encore assez mature. JoCaml [72][2] n'était pas non plus disponible. On a donc opté en faveur d'une implantation utilisant MPI [245], pour des raisons, importantes dans le cadre du projet CARAML, de portabilité.

Cette nouvelle version 0.2 [189] a été distribuée à partir de fin 2002. Suite aux travaux sur la certification de programmes BSML (voir au chapitre 6), une version 0.25 corrigeant un certain nombre d'erreurs de la bibliothèque standard et améliorant l'implantation a été proposée début 2004. Suite aux travaux sur les entrées/sorties (confère chapitre 9), les compositions parallèles (voir au chapitre 7) et les structures de données parallèles (cf. chapitre 8), une nouvelle version 0.3 sera distribuée fin 2005. Celle-ci comporte un noyau modulaire décrit dans cette section.

5.2.2 Une implantation modulaire

La version 0.25 de la bibliothèque BSMLlib n'était pas modulaire dans le sens où l'implantation du module de primitives utilisait directement un module de lien avec MPI pour C et que cette implantation était donc très dépendante de MPI.

Or, pour plusieurs raisons, il était souhaitable de pouvoir s'appuyer sur d'autres bibliothèques de communication que MPI, tout d'abord pour des raisons de portabilité, mais aussi pour des raisons d'efficacité. La structuration de l'exécution permet des optimisations. Par exemple, l'implantation d'une bibliothèque BSPlib avec VIA [164] a permis des optimisations qu'il n'était pas possible de mettre en œuvre pour une implantation de MPI au-dessus de VIA, du fait de l'absence de structuration du modèle sur lequel repose MPI. Enfin, dans le cadre du projet CARAML, il était prévu de traiter de la tolérance aux pannes. Toutefois, cette caractéristique entraîne toujours une baisse des performances. Aussi est-il tout à fait concevable d'avoir plusieurs implantations basées sur la même bibliothèque de communication mais offrant ou non une tolérance aux pannes [43, 46, 47, 48, 49].

C'est pour faciliter la mise en œuvre de cet objectif que le module qui contient les primitives présentées dans la section 2.3 est un *foncteur* [177] prenant en argument un module de communication de plus «bas niveau». Le style de programmation est ici SPMD. Ce module, appelé Comm, est basé sur les principaux éléments donnés ci-dessous (des formalisations, justifiant l'utilisation de ces fonctions, ont été présentées dans les chapitres 3 et 4) :

val pid : unit →int
val nprocs : unit →int
val send : α option array →α option array

[2] téléchargeable à `http://http://pauillac.inria.fr/jocaml`

La signification de pid et de nprocs est évidente : ils donnent respectivement l'identifiant du processeur et le nombre p de processeurs. La fonction send prend sur chaque processeur un tableau de taille p. Ces tableaux contiennent des valeurs optionnelles. Ces valeurs sont obtenues en appliquant sur chaque processeur i la fonction f_i (argument de la primitive **put**) sur les entiers allant de 0 à $p-1$.

Si au processeur j la valeur contenue à l'index i est (Some v), alors la valeur v sera envoyée du processeur j au processeur i. Si la valeur est None, rien ne sera envoyé. Dans le résultat, qui est aussi un tableau, None à l'index j sur le processeur i signifie que le processeur j n'a rien envoyé au processeur i et la valeur (Some v) signifie que le processeur j a envoyé la valeur v à i. Une synchronisation globale s'effectue à l'intérieur de cette fonction de communication. **put** et **proj** sont implantés en utilisant send.

L'implantation des types abstraits, **mkpar** (règle 3.17) et **apply** (règle 3.18) est la suivante :

```
type α par = α
let mkpar f = f (Comm.pid())
let apply f v = f v
let bsp_p = Comm.nprocs
```

L'implantation de la primitive **put** peut être décrite rapidement sur un exemple. Pour cela, on considère que l'on travaille sur une machine parallèle à 4 processeurs. Soit la fonction f_i de type int→α **par** telle que $(f_i\ (i+1)) =$ Some v_i pour $i = 0, 1, 2$ et $(f_i\ j) =$ None sinon. L'expression **mkpar**(**fun** i →f_i) sera évaluée comme suit :

1. D'abord, sur chaque processeur, la fonction est appliquée à tous les identifiants des processeurs. On produit ainsi p valeurs, les messages à envoyer pour chaque processeur. Dans la matrice qui suit, chaque colonne représente les valeurs produites par processeur et chaque ligne correspond à une destination (la première ligne représente les messages envoyés au processeur 0, *etc.*) :

None	None	None	None
Some v_0	None	None	None
None	Some v_1	None	None
None	None	Some v_2	None

2. Ensuite, l'échange s'effectue et on obtient une nouvelle matrice, qui est en fait la transposée de la matrice précédente. C'est send qui réalise ceci, ce qu'on représente ici comme matrice étant simplement les p tableaux des p processeurs de la machine parallèle.

None	Some v_0	None	None
None	None	Some v_1	None
None	None	None	Some v_2
None	None	None	None

3. Finalement, le vecteur parallèle de fonctions est produit. Chaque processeur i a un tableau a_i de taille p (une colonne de la matrice précédente) et la fonction est **fun** x →a_i.(x). Dans notre exemple, au processeur 3, $(f_3\ 0) =$ None, ce qui signifie que le processeur 3 n'a pas reçu de message du processeur 0 et $(f_3\ 2) =$ Some v_2, le processeur 2 a envoyé un message au processeur 3.

L'implantation des primitives **put** (règle 3.19) et **proj** (règle 3.20) sont alors les suivantes :

```
let mkfun = (fun res i →if ((0<=i) && (i<(Comm.nprocs()))) then res.(i) else None)

let put f = mkfun (Comm.send (Array.init (Comm.nprocs()) f))

let proj v = put (fun _ →v)
```

En effet, pour le **proj**, il suffit d'envoyer la même valeur à tous les autres processeurs, d'où l'utilisation possible du **put**. Notons que **proj** est bien une primitive et non une fonction dépendant de **put** car la primitive **proj** enlève le constructeur **par** (des vecteurs parallèles) dans le résultat, ce que la primitive **put** ne peut faire seule.

Il existe actuellement plusieurs implantations du module Comm basées sur MPI [245], PVM [114], PUB [40] et TCP/IP (dans ce dernier cas l'implantation ne repose que sur OCaml, en utilisant un carré latin pour les communications [148]) et il est tout à fait possible que de nouveaux modules Comm puissent être implantés par des programmeurs extérieurs.

```
(* fold_direct: (α →β →α )→α →β par→α par *)
let fold_direct op e vec = parfun (List.fold_left op e) (total_exchange vec)

(* inprod_array: float array→float array→float *)
let inprod_array v1 v2 = let s = ref 0. in
 for i = 0 to (Array.length v1)−1 do
  s:=!s+.(v1.(i)∗.v2.(i));
 done; !s

(* inprod_list: float list→float list→float *)
let inprod_list v1 v2 = List.fold_left2 (fun s x y→s+.x∗.y) 0. v1 v2

(* inprod: (α →β →float)→α par→β par→float par *)
let inprod seqinprod v1 v2 =
  let local_inprod = parfun2 seqinprod v1 v2 in
    fold_direct (+.) 0. local_inprod
```

Figure 5.10 — Produit scalaire

Il faut préciser également que le module des primitives prend en argument un module dédié à l'implantation des nouvelles primitives dédiées aux entrées/sorties. Toutefois, pour l'instant, il n'en existe qu'une seule implantation. Cette modularité permet d'envisager de nouvelles implantations basées sur des bibliothèques spécialisées dans les entrées/sorties parallèles. Notons que le chapitre 7, qui présente la sémantique et l'implantation d'une nouvelle primitive de composition parallèle, donne les modifications «mineures» ajoutées par cette primitive.

5.2.3 Prévision des performances

L'un des principaux avantages du modèle BSP est son modèle de coût : il est simple mais suffisamment précis. Cela a par exemple été montré dans l'article [157] en utilisant la bibliothèque BSPlib [147]. Ce constat a été de nouveau démontré expérimentalement [171] avec la BSPlib, mais aussi avec la PUB, MPI et cela sur différentes architectures parallèles.

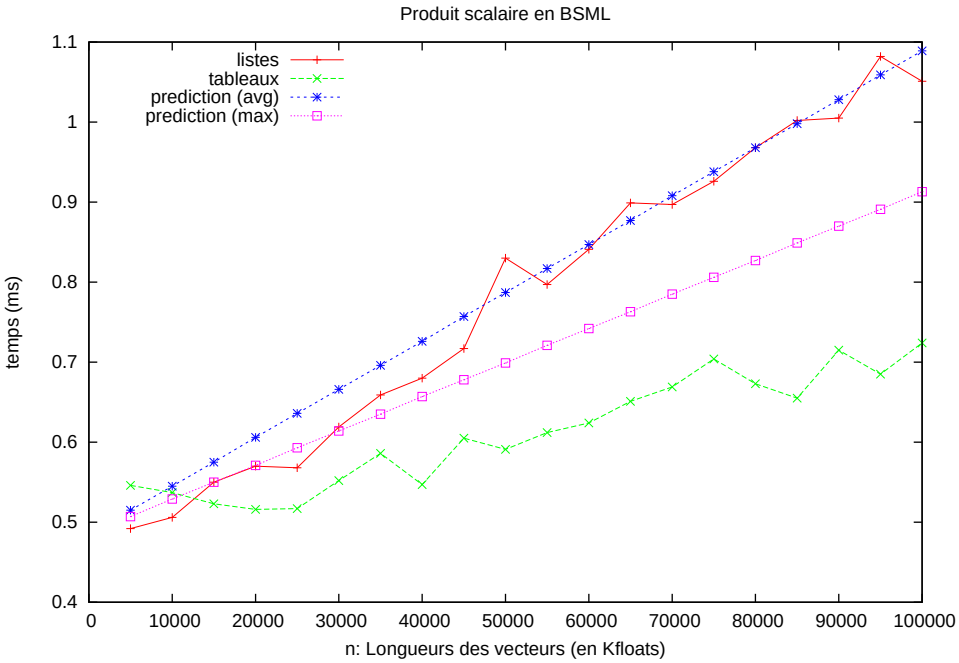
La prévision des performances nécessite que l'on puisse déterminer les paramètres BSP des machines que l'on utilise. C'est ce que fait par exemple le programme «probe» présenté dans [34]. Nous l'avons adapté pour la BSMLlib [C1]. La figure 5.12 donnée à la fin de ce chapitre présente la partie principale du programme.

Le tableau de la figure 5.11 présente les résultats obtenus en exécutant le programme «probe» de [34] (C+MPI) et notre «bsmlprobe» (BSMLlib avec l'implantation MPI du module Comm) sur notre grappe constituée de 6 nœuds.

Quelques expérimentations ont été réalisées sur un programme (voir à la figure 5.11) qui calcule le produit scalaire de deux vecteurs, soit en utilisant des tableaux pour stocker les vecteurs, soit en utilisant des listes. Le code est donné à la figure 5.10. Ce programme utilise la fonction fold_direct qui est une réduction en une étape de calcul et une étape de communication/synchronisation (une super-étape BSP).

La formule de coût BSP de inprod est : $n + 2 \times p + (p-1) \times g + l$. Les résultats expérimentaux et la courbe des performances prévues sont présentés à la figure 5.11.

Il n'existe pas d'applications «réelles» écrites en BSML, mais quelques exemples de taille modeste, l'implantation de structures de données parallèles (voir au chapitre 8) et surtout les fonctions disponibles dans la bibliothèque standard BSMLlib. Nous reviendrons sur ce point dans le chapitre 13.



C+MPI			BSMLlib (MPI)		
r	g	l	r	g	l
478	25.2	623141	469	28.0	227512
r en Mflops/s, g en flops/mots et l en flops					

Figure 5.11 — Performances du produit scalaire

```
let determine_one_r niters n =
  let a = 1./.3. and b = 4./.9. and z = Array.init n foi
  and y = Array.init n foi and x = Array.init n foi in
    begin
      start_timing();
      for iter=1 to niters do
        for i=0 to n−1 do y.(i)<− a∗.x.(i) +. y.(i); done;
        for i=0 to n−1 do z.(i)<−z.(i)−.b∗.x.(i); done;
      done;
      stop_timing();
      get_cost();
    end

type mode = Max | Average

let determine_r niters maxn mode =
  let rec mklist n1 n2 = if n1>n2 then [] else n1::(mklist (2∗n1) n2) in
  let ns = mklist 16 maxn in
  let mflops =
    let f n = parfun (toMflops niters n) (determine_one_r niters n) in
      List.map f ns in
  let rs = List.map avgtime mflops in
    match mode with
      | Average →(List.fold_left (+.) 0. rs)/.(float)(List.length ns)
      | Max →(List.fold_left max 0. rs)

let determine_one_g_and_l niters h =
  let rest = h mod (bsp_p()−1) in
  let size1 = h/(bsp_p()−1) in
  let size2 = if rest=0 then size1 else size1+1
  and create size = if size>0 then Some(Array.create size foi) else None in
  let v1 = create size1 and v2 = create size2 in
  let msg pid dst =
    let d = if pid<rest then 1 else 0 in
      if pid=dst then None else if dst<rest+d then v2 else v1 in
    begin
      start_timing();
      for iter=1 to niters do
        put(mkpar msg);
      done;
      stop_timing();
      get_cost()
    end

let determine_g_and_l niters maxh r r'=
  let rec mklist n1 n2 = if n1>n2 then [] else n1::(mklist (n1+1) n2) in
  let hs = mklist (bsp_p()) maxh in
  let timings = List.map (compose avgtime (determine_one_g_and_l niters)) hs in
  let g,l = leastsquares hs timings in
   (g/.(float) niters)∗.r, (l/.(float) niters)∗.r, (g/.(float) niters)∗.r', (l/.(float) niters)∗.r'
```

Figure 5.12 — Programme déterminant les paramètres BSP de la machine parallèle

6 Certification de programmes BSML

Le contenu de ce chapitre étend le travail qui a été effectué dans les articles [R5] et [N2].

Sommaire

La programmation fonctionnelle est un bon cadre de travail pour l'écriture et la preuve de programmes car, sans effet de bord, il est plus simple de prouver la correction des programmes et de les réutiliser en tenant compte de leurs propriétés formelles : la sémantique est dite *compositionnelle*. Le langage BSML (sans les traits impératifs ni les entrées/sorties) permet de concevoir des programmes parallèles purement fonctionnels. Sa sémantique est aussi compositionnelle. Se pose alors la question de la réalisation de programmes BSML certifiés, c'est-à-dire dont le résultat est toujours celui qui a été spécifié. Ce chapitre traite ce problème et nous utiliserons l'assistant de preuves **Coq** afin de founir une bibliothèque de fonctions BSML certifiées qui sera un large sous-ensemble de la bibliothèque standard de la BSMLlib.

6.1 Introduction

Le système **Coq** [30, 254][1] est un environnement (et un langage logique) pour le développement de preuves et qui est basé sur le Calcul des Constructions Inductives [75], un λ-calcul typé étendu avec des *définitions*

[1] Téléchargeable à l'adresse suivante : `http://coq.inria.fr/`.

inductives. La théorie des types est un bon cadre pour le développement de preuves de programmes (notamment ceux fonctionnels [218]) car elle fournit une grande expressivité logique. Dans l'assistant de preuves **Coq**, il existe une interprétation constructive des preuves, c'est-à-dire la possibilité de construire automatiquement une λ-terme à partir de la preuve d'une formule logique. Ainsi, dans un tel formalisme et grâce à l'isomorphisme de Curry-Howard, une preuve de la formule $\forall x.P(x) \Rightarrow \exists y.(Q\ y\ x)$, appelée une spécification, permet d'obtenir un programme correct qui vérifie en entrée la propriété P (une pré-condition) et fournit un résultat qui vérifie Q (une post-condition). Le programme extrait de la preuve (en oubliant les parties logiques de la preuve [220]) est donc garanti par **Coq** pour bien réaliser la spécification donnée. On parle alors d'un programme certifié.

Nous allons maintenant nous intéresser à la spécification de programmes BSML et à leur développement (réalisation) en **Coq**. Tout d'abord, nous présenterons rapidement le système **Coq** (section 6.2). Nous pourrons alors expliquer comment intégrer les primitives parallèles dans ce système (section 6.3) pour certifier des éléments de la bibliothèque BSMLlib et ainsi obtenir une bibliothèque certifiée (section 6.5).

6.2 Présentation succinte de Coq

6.2.1 Généralités

Coq est un système d'aide à la preuve basé sur le Calcul des Constructions Inductives (CCI), qui est un λ-calcul typé dans lequel des types sont des termes comme les autres. Il fournit des mécanismes pour écrire des définitions et pour faire des preuves formelles.

Le système **Coq** est un programme interactif permettant à l'utilisateur de construire des preuves. Cela permet de :

1. Définir des termes, c'est-à-dire les objets (mathématiques) du discours, comme les entiers, les listes etc., et des propriétés sur ces objets puisque celles-ci sont des types et que types et termes ne sont pas différenciés ;

2. Construire des démonstrations de ces propriétés (qui sont également des termes) à l'aide d'un langage de *tactiques* permettant de fabriquer interactivement un terme d'un type donné ;

3. Vérifier les preuves ainsi construites, en demandant au programme de les typer.

Le mécanisme de construction des preuves n'a pas besoin d'être certifié puisque le noyau du système **Coq**, c'est-à-dire la fonction de typage vérifie les preuves. Le noyau fait en revanche l'objet de certifications et de vérifications [22]. Ajoutons que la représentation concrète des preuves a d'autres avantages : par exemple, elle rend possible l'écriture d'un vérificateur de type indépendant de **Coq**, et elle autorise également l'emploie de toutes sortes d'outils pour engendrer des preuves que **Coq** pourra vérifier *a posteriori* [107].

Tous les exemples donnés dans la syntaxe **Coq** seront dans une police *verbatim* et seront en général suivis de la réponse du système. Les différents exemples de ce chapitre peuvent êtres tapés directement dans **Coq**.

Il n'y a pas d'inférence de type en **Coq**, celle-ci n'étant pas décidable[2]. Il est cependant laborieux de devoir annoter toutes les définitions par des informations de type quand celles-ci semblent pouvoir être aisément déduites du contexte. **Coq** propose donc un mécanisme simple d'inférence, qui, bien sûr, peut échouer. **Coq** fournit également un mécanisme d'arguments implicites et il s'efforce aussi de les inférer.

Nous nous intéressons donc, dans cette section, au système d'aide à la preuve **Coq**. Cette section a pour objectif de faire découvrir au lecteur ce système à l'aide d'une rapide description. Bien sûr, cette section n'a pas pour vocation de remplacer la documentation accompagnant le système. Le lecteur pourra donc consulter en complément : le tutoriel et le livre [30] pour une introduction plus graduelle ainsi que le manuel de référence[3] pour une description formelle et exhaustive de **Coq**.

6.2.2 Assistant de preuves

L'objectif premier de **Coq** est d'être un assistant de preuves, et donc de permettre de formaliser le raisonnement mathématique. Prenons un énoncé basique : $\forall A, A \Rightarrow A$. Cet énoncé, qui se lit «pour toute proposition `A` (c'est-à-dire objet `A` de type `Prop`), `A` implique `A`», peut être directement transcrit dans le système **Coq** comme suit :

[2] Cela correspondrait à un «prouveur» automatique «universel», ce qui est rigoureusement impossible.

[3] Le manuel de référence ainsi que le tutoriel sont librement téléchargeables à `http://coq.inria.fr`.

```
Lemma easy: ∀ A:Prop, A→A.
Proof.
```

Cette portion de script commence par le nom et l'énoncé de notre lemme, où `Prop` est l'ensemble des propositions logiques. Le système entre alors dans sa nature *interactive* qui permet de bâtir cette preuve étape par étape (mot clés **Proof**). Pour cela, le système utilise des directives appelées *tactiques*. Celles-ci reflètent la structure de la preuve en déduction naturelle. Nous avons donc :

```
1 subgoal

  ============================
   ∀ A:Prop, A→A

easy < intros.
```

En entrant la tactique **intros** (introduire les hypothèses), le système affiche l'état courant du ou des buts à prouver :

```
1 subgoal

  A : Prop
  H : A
  ============================
   A

easy < apply H.
Proof completed.
```

Il s'agit alors de trouver une preuve de `A` sous l'hypothèse `A` (cette hypothèse étant nommée `H` tel que `A` soit de type `Prop`). La tactique **apply** `H` permet d'appliquer cette hypothèse, ce qui termine la preuve. Il ne reste plus qu'à faire vérifier l'ensemble de la preuve au système **Coq** (vérification des types de tous les sous-termes) à l'aide de la tactique **Qed.**

De nombreuses interfaces existent pour faciliter cette interaction, comme par exemple **CoqIDE**[4]. Il faut également noter que **Coq** met à la disposition de l'utilisateur une grande variété de tactiques. Ici, par exemple, pour un énoncé aussi simple, les tactiques de recherche automatique telles que **auto** ou **trivial** auraient été suffisantes.

La représentation interne d'une preuve est un λ-terme. Le système logique sous-jacent à **Coq** est le CCI (Calcul des Constructions Inductives) qui est un λ-calcul typé où les énoncés exprimables en **Coq** sont des types du CCI. En application de l'isomorphisme de Curry-Howard [119], vérifier si `t` est bien une preuve valide de l'énoncé `T` consiste à vérifier que le type `T` est bien un type légal du λ-terme `t`. C'est ce qui est effectué par le système avec la tactique **Qed.** Par exemple, si on demande à **Coq** quelle est la représentation interne du lemme `easy` à l'aide d'un `Print`, nous obtenons :

```
easy = fun (A:Prop) (H:A) ⇒ H
     : ∀ A:Prop, A→A
```

où **fun** `x:X` ⇒ `e` est la notation **Coq** pour l'abstraction typée. Celle-ci est l'effet (d'après l'isomorphisme de Curry-Howard) de la tactique **intro**. Quand à **apply**, son effet est l'utilisation de la variable de contexte `H`. De façon générale, une tactique contribue à construire petit à petit le λ-terme CCI de la preuve.

Notons que si l'on connaît à l'avance le λ-terme complet, on peut le donner directement sous la forme par exemple d'une **Definition**, ce qui donne :

```
Definition easy: ∀ A:Prop, A→A := fun (A:Prop) (H:A) ⇒ H
```

La quantification universelle $\forall x : X, T$ est appelé *produit* ou *type dépendant*, puisque, le plus souvent, le corps `T` du produit dépend de la variable `x` de la tête du produit. Cette quantification n'étant pas restreinte, le CCI (le système **Coq**), est donc une logique d'ordre supérieur. Notons que la syntaxe `A→B` est un sucre syntaxique de **Coq** pour `∀ _:A, B`.

6.2.3 Langage de programmation

On peut également aborder **Coq** (et c'est ce qui nous intéressera le plus ici) depuis l'autre versant de l'isomorphisme de Curry-Howard : considérer **Coq** non pas comme un système logique mais plutôt commme un

[4] librement téléchargeables à `http://coq.inria.fr`

λ-calcul, c'est-à-dire un langage de programmation purement fonctionnel. Par exemple, notre lemme `easy` devient une fonction d'identité sur les propositions logiques.

En tant que langage de programmation, il nous faut pouvoir définir des types de données que l'on rencontre dans les langages de programmation. Cette définition se fait en **Coq** à l'aide de types inductifs. Le type des booléens s'obtient par la déclaration :

```
Inductive bool:Set := true:bool | false:bool
```

Cette déclaration crée un nouveau type, nommé `bool`, ainsi que deux nouveaux constructeurs. L'annotation `Set` permet de désigner quel va être le type du type `bool`. C'est le type des objets calculatoires (à la différence de `Prop` qui est le type des objets logiques). De même on peut définir les entiers de Peano de la manière suivante :

```
Inductive nat:Set := 0:nat | S:nat→nat.
```

où `0` est le zéro et `S` le successeur d'un `nat`. Un dernier exemple usuel est celui des listes paramétriques :

```
Inductive list (A:Set): Set := nil:list A | cons:A→list A→list A
```

où la liste dépend d'un paramètre `A`. Ce paramètre sera fourni en arguments lors de l'utilisation de cette structure de données. Par exemple une liste d'entiers naturels peut être définie par :

```
Inductive list_nat:Set := nil_nat:list_nat
                        | cons_nat:nat→list_nat→list_nat
```

mais plus simplement par `Definition list_nat:Set := (list nat).`

Il faut noter que le système refuse certains inductifs (on parle d'une condition de positivité) dont la définition est syntaxiquement valide. Ces restrictions ont pour but d'assurer la cohérence du système du point de vue logique[5], c'est-à-dire l'impossibilité de prouver P et $\neg P$. Par exemple :

```
Inductive absurde: Set := C:(absurde→absurde)→absurde.

Error: Non strictly positive occurrence of "absurde" in
 "(absurde → absurde) → absurde"
```

L'entrelacement des parties logiques (des propositions) et des parties purement calculatoires est possible dans le système **Coq**. Cela permet notamment d'enrichir un terme calculatoire avec des pré- et post-conditions, ou encore d'utiliser une récursion bien fondée en justifiant la décroissance d'une mesure à chaque appel récursif.

Outre le besoin d'exprimer la spécification d'une fonction sous la forme de pré- et de post-conditions, les pré-conditions logiques vont également apporter une solution au problème de la définition des fonctions partielles. Considérons, par exemple, une fonction de prédécesseur sur les entiers naturels `pred:nat→nat`, qui n'est pas définie quand son argument est nul. On peut alors exprimer, par une pré-condition logique, le fait que cet argument doit être non nul si l'on veut utiliser cette fonction. Le type de `pred` devient alors `∀ n:nat, n<>0 → nat`. On notera que le type de l'argument n'est plus un type flèche (un produit anonyme) mais un produit nommé par `n` afin de s'y référer dans l'assertion logique. La fonction `pred` n'est alors plus définie en dehors du domaine de validité de l'assertion logique. La contre-partie est que l'on doit toujours fournir une preuve logique de non-nullité à chaque appel à `pred`.

Combiner pré- et post-conditions spécificie une fonction dans un style à la Hoare [149]. Ainsi, une fonction de type `A→B` de pré-condition `P` et de post-condition `Q` correspond à la preuve constructive : $\forall x : A,\ (P\ x) \rightarrow \exists y : B,\ (Q\ x\ y)$. Ceci est exprimable en **Coq** à l'aide d'un type inductif `sig` :

```
Inductif sig (A:Set)(P:A→Prop): Set :=
    exist: ∀ x:A, (P x)→(sig A P).
```

qui s'écrit également avec un sucre syntaxique `{x:A | (P x)}`. Une spécification complète en **Coq** de la fonction de prédécesseur entière s'écrit alors :

```
Definition pred: ∀ n:nat, n<>0 → {q:nat | (S q)=n}
```

Il ne reste alors qu'à donner au système un λ-terme répondant à cette spécification (un λ-terme ayant ce type). On peut le construire par une interaction de tactiques notamment avec une de filtrage (comme **case**) pour prouver tous les cas des constructeurs.

Une solution plus simple est l'utilisation de la tactique **refine** qui permet de donner directement le terme **Coq** correspondant à la spécification mais avec des fragments vides. Ces fragments (généralement la

[5] Le lecteur logicien intéressé par plus de détails peut consulter [271].

preuve logique que le terme correspond à la spécification) vont générer des buts qui devront être prouvés afin d'accepter le terme. Dans notre cas :

```
refine (fun n ⇒ match n return n<>0 → {q:nat | (S q)=n} with
   0 ⇒ fun h:(0<>0) ⇒ _
 | (S p) ⇒ fun h:((S p)<>0) ⇒ (exist _ p _) end).
```

où _ indique le fragment (trou) du terme à compléter avec des tactiques [218] et où `return` est une construction syntaxique utilisée lors du filtrage d'un terme provenant d'un type dépendant. Avec le **match**, nous distinguons les deux cas possibles du calcul et faisons ressortir la preuve `h` de la propriété de l'argument. Le retour est l'existence (avec `exist`) d'une valeur `p` qui vérifie la spécification. Nous avons alors ici deux buts à résoudre :

```
2 subgoals

  n : nat
  h : 0<>0
  ============================
   {q:nat | S q = 0}

subgoal 2 is:
 S p = S p
```

Le premier but se résout facilement par l'absurde : **absurd** `(0<>0); auto.` et le second est une trivialité. Nous pouvons maintenant afficher le λ-terme CCI complet de notre fonction :

```
Print pred.
pred =
fun n:nat ⇒
match n as n0 return (n0<>0 → {q:nat | S q = n0}) with
| O ⇒
    fun h:0<>0 ⇒
    False_rec {q:nat | S q = 0}
      (let H := h in
       (let H0 := fun H:0=0 → False ⇒ H (refl_equal 0) in
        fun H1:0<>0 ⇒ H0 H1) H)
| S p ⇒
    fun _ :S p<>0 ⇒ exist (fun q:nat ⇒ S q = S p) p (refl_equal (S p))
end
     : ∀ n:nat, n<>0 → {q:nat | S q = n}
```

Il est aussi possible de définir en **Coq** un terme par récurrence structurelle sur un objet inductif (une liste par exemple). Une définition récursive n'est acceptée que si tout appel récursif interne se fait sur un argument récursif qui est structurellement plus petit que l'argument récursif initial[6]. La préoccupation est, là encore, celle de la cohérence logique. Sans ces conditions sur les appels récursifs, il serait en effet aisé de construire des termes de n'importe quel type `A` :

```
Fixpoint loop (n:nat) : A := loop n.
Definition impossible : A := loop 0.
```

Le système logique serait alors incohérent.

6.2.4 Coq et BSML

L'originalité du travail présenté dans ce chapitre est le «plongement» de BSML dans un système d'aide à la preuve. De précédents travaux visant à l'axiomatisation d'un langage BSP (voir travaux connexes au chapitre 12) ont montré qu'il était très délicat de tenir compte de tous ces aspects. Dans ce travail, le choix des langages utilisés, **Coq** et BSML, a été important car cela permet une axiomatisation simple et naturelle. Celle-ci pourra donc être employée pour la preuve de programmes BSP.

BSML apparaît comme un bon candidat pour une telle axiomatisation car il s'agit d'un langage purement fonctionnel dont la sémantique est *a priori* assez bien adaptée à une description formelle. **Coq** est apparu comme un système d'aide à la preuve approprié à cette axiomatisation. C'est un système pour la programmation fonctionnelle et les types dépendants se sont avérés (pour l'instant) suffisamment puissants pour exprimer les propriétés formelles de nos fonctions BSML. Nous allons donc exprimer BSML dans ce calcul de types.

[6] On trouve une définition formelle de «plus petit» dans [219].

6.3 Formalisation des primitives BSML

Dans cette section, nous nous intéresserons à la certification de fonctions BSML. Pour cela, nous allons exprimer la sémantique naturelle de BSML (présentée dans le chapitre 3) dans le système de preuves **Coq**. Ce codage nous fournit directement un vérificateur des propriétés des fonctions de la bibliothèque standard de la BSMLlib.

Pour représenter nos primitives parallèles et pour avoir une spécification-réalisation des programmes BSML, nous avons choisi une approche classique : une *axiomatisation* de nos opérateurs (figure 6.1). Les axiomes sont basés sur la sémantique naturelle de BSML et les primitives de BSML sont données *via* des *paramètres*. Elles ne dépendent donc pas de leurs implantations (séquentielles ou parallèles). Les programmes certifiés sont donc corrects quelle que soit l'implantation des primitives (tant qu'elle valide les définitions formelles des primitives).

6.3.1 Axiomes et paramètres

Le nombre de processus, `bsp_p()`, est naturellement un nombre entier supposé supérieur à 0 (axiome `good_bsp_p`). Le constructeur `tt` est le constructeur **Coq** de `()` d'OCaml. Les vecteurs parallèles sont indexés sur le type `Z` (les entiers relatifs de **Coq**[7]). Ils sont représentés dans le monde logique par un type dépendant : **`Vector`** `T`, où `T` est le type des éléments du vecteur. Ce type abstrait est, bien entendu, uniquement manipulé par les primitives données en paramètres. Notre axiomatisation des primitives parallèles BSML est donnée à la figure 6.1. Celle-ci est simple mais suffisante pour notre approche (l'extraction des preuves de nos spécifications pour avoir une bibliothèque certifiée). **`att`** est une *fonction abstraite* d'«accès» (qui ne doit pas être utilisée dans les programmes) pour les vecteurs parallèles. Elle donne la valeur locale contenue dans un processus et sera employée par la suite uniquement dans les spécifications des programmes pour donner les valeurs contenues dans les vecteurs parallèles. Elle a un type dépendant pour vérifier que l'entier `i` est bel et bien un nom de processus valide.

Les primitives asynchrones, la primitive **`mkpar`** et l'application globale **`apply`** sont axiomatisés avec **`mkpar_def`** et **`apply_def`**. Pour une fonction `f`, **`mkpar_def`** applique `(f i)` sur le processus `i` en employant la fonction d'accès **`att`**. De la même manière, **`apply_def`** permet l'application des composantes au processus `i`. Le vecteur parallèle résultant est alors décrit avec une égalité qui est donc une proposition logique. Ce résultat est donné pour un paramètre `i` qui doit être prouvé comme un nom valide de processus.

La primitive **`put`** est axiomatisée avec **`put_def`**. Celui-ci transforme un vecteur fonctionnel en un autre vecteur fonctionnel qui permet la communication en utilisant le paramètre `j` pour lire les valeurs des processus distants (dans le vecteur `Vf`). Le paramètre `j` est testé avec la fonction `within_bounds` de type :

```
∀ (i:Z), {0<=i<(bsp_p tt)}+{¬(0<=i<(bsp_p tt))}
```

qui indique si un entier est un nom de processus valide ou non et en fournit une preuve (*via* `H1` dans la règle). Ceci est nécessaire car le système **Coq** est un système avec type dépendant. En effet, pour des preuves de programmes, il est plus aisé de manipuler directement des preuves que des constantes primitives. Si `i` est bien un nom de processus valide, la valeur sur le processus `i` est lue sur le processus `j` avec la fonction d'accès **`att`** sur la `j`ème composante du vecteur `Vf` (j est un nom valide de processus ; la preuve est donnée par `within_bounds`). Autrement, une constante vide est retournée. Dans une véritable implantation, et pour des raisons évidentes d'optimisation, les valeurs à émettre sont tout d'abord calculées et ensuite échangées. L'axiomatisation complète contient également la projection globale synchrone. Elle est facilement exprimable dans le système **Coq** en utilisant la fonction d'«accès» **`att`** pour «lire» la valeur au processeur `j` (dans la `j`ème composante du vecteur parallèle `V`). Ce paramètre `j` doit être un nom valide de processus et cette preuve est donnée à la fonction d'«accès» **`att`** par un type dépendant.

Nous allons maintenant illustrer cette axiomatisation en employant nos axiomes pour obtenir une bibliothèque certifiée par l'assistant de preuve **Coq**. Nous pouvons voir l'ajout de ces axiomes comme le passage de **Coq** à un «BS-Coq» (comme le passage de ML à BSML par l'utilisation de nos primitives parallèles) avec des $BS\lambda$-termes CCI quand les primitives parallèles sont utilisées.

[7] nous les préférons aux entiers de peano `nat` pour des raisons d'efficacité et afin de profiter de la tactique `Omega` qui autorise la résolution automatique et certifiée d'équations dans l'arithmétique de Presburger

```
Parameters bsp_p: unit→Z.
           Vector: Set→Set.
           mkpar: ∀ T:Set, (Z→T)→(Vector T).
           apply: ∀ T1 T2:Set, (Vector (T1→T2))
                               → (Vector T1)→(Vector T2).
           put: ∀ T:Set, (Vector (Z→(option T)))
                                 → (Vector (Z→(option T))).
           proj: ∀ T:Set, (Vector (option T))→Z→(option T).


Parameter att: ∀ T:Set, (Vector T)→∀ (i:Z), 0<=i<(bsp_p tt)→T.


Definition within_bound: ∀ i:Z, {0<=i<(bsp_p tt)}+{¬(0<=i<(bsp_p tt))}.


Axiom good_bsp_p: 0<(bsp_p tt).


Axiom mkpar_def: ∀ (T:Set) (f:Z→T) (i:Z) (H:0<=i<(bsp_p tt)),
  (att T (mkpar T f) i H)=(f i).


Axiom apply_def: ∀ (T1 T2:Set) (V1:(Vector (T1→T2)))
                 (V2:(Vector T1)) (i:Z) (H:0<=i<(bsp_p tt)),
  (att T2 (appply T1 T2 V1 V2) i H)=((att (T1→T2) V1 i H) (att T1 V2 i H)).


Axiom put_def: ∀ (T:Set) (Vf:(Vector (Z→(option T))))
               (i:Z) (H:0<=i<(bsp_p tt)),
 ((att (Z→(option T)) (put T Vf) i H)=(fun j:Z ⇒ match (within_bound j) with
      left H1 ⇒ ((att (Z→(option T)) Vf j H1) i)
    | right _ ⇒ None end)).


Axiom proj_def: ∀ (T:Set) (Vv:(Vector (option T))),
 (proj T Vv)=(fun j:Z ⇒ match (within_bound j) with
    left H1 ⇒ (att (option T) Vv j H1)
  | right _ ⇒ None end).


Axiom at_H: ∀ (T:Set) (v:(Vector T)) (i:Z) (H1 H2:0<=i<(bsp_p tt)),
  (att T v i H1)=(att T v i H2).
```

Figure 6.1 — Axiomatisation des primitives BSML

6.3.2 Cohérence et acceptation de l'ajout des axiomes BSML

Pour prouver la cohérence de nos axiomes dans le système **Coq** (impossibilité de prouver $\bot$), nous pouvons implanter les primitives en **Coq** à l'aide de listes (chacune de taille «fixe»). Nos précédents axiomes sont alors des lemmes et la cohérence vient de celle de **Coq**. Ainsi, nos axiomes et nos paramètres (dont on peut donner une implantation certifiée) correspondent à l'exécution séquentielle, mais au moment de la compilation (des fonctions extraites), rien n'empêche d'utiliser l'implantation parallèle (qui a été prouvée équivalente, cf chapitre 3).

Une autre question se pose : est-ce que nos axiomes (sémantiques des primitives) correspondent bien à l'exécution de nos programmes ? En effet, le chapitre 3 propose deux exécutions possibles et équivalentes de nos termes : une version séquentielle et la version distribuée (c'est-à-dire parallèle) associée. Dans les deux cas, les primitives emploient des opérations parallèles, notemment pour l'échange des valeurs et il nous faut prouver qu'ils correspondent bien à la sémantique de «haut niveau» du système de preuves. Pour cela nous allons correspondre les axiomes à des évaluations de termes symboliques. Nous notons att v i (ième composante d'un vecteur v) par $v[i]$.

Notre premier axiome est celui de la création d'un vecteur parallèle, **mkpar_def**. Avec cet axiome, nous avons : $\forall i \in \{0,\ldots,p-1\}$ $(\textbf{mkpar}\ \mathsf{f}[i]) = (\mathsf{f}\ i)$ et nous avons la règle d'évaluation suivante : $(\textbf{mkpar}\ f) \stackrel{*}{\rightharpoonup} \langle (f\ 0),\ldots,(f\ (p-1))\rangle$. L'axiome et la règle d'évaluation se correspondent bien. Nous pouvons aussi appliquer le même procédé à l'axiome **apply_def** afin de faire correspondre les résulats.

Pour l'axiome **put_def**, nous avons : $\forall i \in \{0,\ldots,p-1\}$ $(\textbf{put}\ \mathsf{vec}) = \mathsf{g}_i$ tel que $\mathsf{g}_i = \textbf{fun}\ j \rightarrow \textbf{if}\ (0 \leq j < p)\ \textbf{then}\ (vec[j]\ i)\ \textbf{else None}$. Avec les régles d'évaluation du chapitre 3, nous avons :

$$
\begin{array}{rl}
 & \textbf{put}\ \langle f_0,\ldots,f_{p-1}\rangle \\
\rightharpoonup & (\textbf{apply}\ (\textbf{mkpar}\ (\lambda.\mathbf{F})[\bullet])[\bullet]\ (\textbf{send}\ \langle(\textbf{init}\ f_0),\ldots,(\textbf{init}\ f_{p-1})\rangle)) \\
\cdots & \\
\rightharpoonup & (\textbf{apply}\ \langle(\overline{(\mathbf{F})[0\circ\bullet]},\ldots,\overline{(\mathbf{F})[p-1\circ\bullet]}\rangle\ (\textbf{send}\ \langle(\textbf{init}\ f_0),\ldots,(\textbf{init}\ f_{p-1})\rangle)) \\
\cdots & \\
\rightharpoonup & (\textbf{apply}\ \langle(\overline{(\mathbf{F})[0\circ\bullet]},\ldots,\overline{(\mathbf{F})[p-1\circ\bullet]}\rangle\ (\textbf{send}\ \langle[v_0^0,\ldots v_0^{p-1}]\ldots,[v_{p-1}^0,\ldots v_{p-1}^{p-1}]\rangle)) \\
\rightharpoonup & (\textbf{apply}\ \langle(\overline{(\mathbf{F})[0\circ\bullet]},\ldots,\overline{(\mathbf{F})[p-1\circ\bullet]}\rangle\ \langle[v_0^0,\ldots v_{p-1}^0]\ldots,[v_0^{p-1},\ldots v_{p-1}^{p-1}]\rangle) \\
\cdots & \\
\rightharpoonup & \langle\cdots,\overline{(\lambda.\textbf{if}\ (0\leq\overline{1}<p)\ \textbf{then}\ (\textbf{access}\ (\overline{2},\overline{1}))\ \textbf{else nc})[[v_0^i,\ldots,v_{p-1}^i]\circ i\circ\bullet]},\cdots\rangle \\
= & \langle\cdots,g_i',\cdots\rangle
\end{array}
$$

Maintenant, nous pouvons comparer, les deux résultats. Prenons un entier a. Si $\neg(0 \leq a < p)$ alors $(g_i\ a) = \textbf{None}$ et $(g_i'\ a) \stackrel{*}{\rightharpoonup} \textbf{nc}$. Si $(0 \leq a < p)$ alors $(g_i\ a) = (vec[a]\ i) = (f_a\ i) = v_a^i$ et $(g_i'\ a) \stackrel{*}{\rightharpoonup} v_a^i$. Dans les deux cas, les valeurs se correspondent. Nous pouvont aussi appliquer le même procédé à l'axiome **proj_def** afin de faire correspondre les résultats.

Nos axiomes de nos primitives vérifient bien les règles d'évaluation. Nous pouvons donc bien les accepter et les utiliser afin de certifier les fonctions BSML.

6.4 Développements de fonctions certifiées

Nous présentons ici le développement complet de fonctions parallèles de base de la bibliothèque BSMLlib dans le système **Coq**.

6.4.1 Fonction certifiée utilisant un mkpar

Notre premier exemple d'un développement complet d'une fonction BSML certifiée est la réplication d'une valeur en chaque composante d'un vecteur parallèle : replicate. Cette fonction permet de mettre en parallèle un élément. Par la suite, nous ne donnons pas le type d'un objet quand celui-ci est évident et $vec[.]$ est un sucre syntaxique pour l'«accès» aux éléments d'un vecteur parallèle vec. Nous avons alors la spécification logique suivante pour replicate :

$$\forall \text{T} : \text{Set}\ \forall a : \text{T} \Rightarrow \exists res\ \text{tel que}\ \forall i\ (0 \leq i < p)\ (\text{res}[i] = a)$$

Nous traduisons cette spécification en **Coq** de la manière suivante :

```
Definition replicate: ∀ (T:Set) (a:T), {res:(Vector T) |
      ∀ (i:Z) (Hyp_i:0<=i<(bsp_p tt)), (att T res i Hyp_i)=a}.
Proof.
```

Coq entre alors dans un mode interactif pour la preuve et nous donne le but suivant (qui n'est alors que la spécification) :

```
1 subgoal

  ============================
   ∀ (T:Set) (a:T),
    {res:Vector T | ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt), att T res i Hyp_i = a}

replicate < intros T a.
```

Nous commençons alors la preuve en introduisant les paramètres en tant qu'hypothèses dans le contexte. Ce qui donne le but suivant :

```
  T : Set
  a : T
  ============================
   {res:Vector T | ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt), att T res i Hyp_i = a}

replicate < exists (mkpar T (fun pid:Z ⇒ a)).
```

Nous pouvons alors donner l'existence d'un terme `res` répondant aux critères de la spécification et nous obtenons :

```
  T : Set
  a : T
  ============================
   ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt), att T (mkpar T (fun _ :Z ⇒ a)) i Hyp_i = a

replicate < intros i Hyp_i.
```

Là encore, il nous faut introduire les hypothèses, ce qui donne :

```
  T : Set
  a : T
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  ============================
   att T (mkpar T (fun _ :Z ⇒ a)) i Hyp_i = a

replicate < rewrite mkpar_def.
```

Nous pouvons alors appliquer (réécrire) l'axiome de **mkpar_def** qui définit sa sémantique :

```
  T : Set
  a : T
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  ============================
   a = a

replicate < trivial.
Proof completed.
```

La fin de la preuve est alors **trivial**e. A l'aide d'un `Print`, nous pouvons afficher quel est le terme $BS\lambda$-terme CCI qui correspond à la preuve de cette spécification. Ce terme est donc une fonction BSML certifiée dont le type est la spécification donnée au début :

```
replicate =
fun (T:Set) (a:T) ⇒
exist
  (fun res:Vector T ⇒
   ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt), att T res i Hyp_i = a)
  (mkpar T (fun _ :Z ⇒ a))
  (fun (i:Z) (Hyp_i:0<=i<bsp_p tt) ⇒
   eq_ind_r (fun t:T ⇒ t = a) (refl_equal a)
     (mkpar_def T (fun _ :Z ⇒ a) i Hyp_i))
     : ∀ (T:Set) (a:T),
       {res:Vector T |
       ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt), att T res i Hyp_i = a}
```

6.4.2 Fonction certifiée utilisant un apply

Notre second exemple est une fonction permettant l'application d'une même fonction en chaque composante d'un vecteur parallèle (parfun). Sa spécification logique est la suivante :

$$\forall \text{T1,T2} : \text{Set}\ \forall f : \text{T1} \rightarrow \text{T2}\ \forall vec \Rightarrow \exists res \text{ tel que } \forall i\ (0 \leq i < p)\ (\text{res}[i] = (f\ vec[i]))$$

ce qui ce traduit en **Coq** par :

```
Definition parfun: ∀ (T1 T2:Set) (f:T1→T2) (v:(Vector T1)),
  {res:(Vector T2) | ∀ (i:Z) (Hyp_i:0<=i<(bsp_p tt)),
                       (att T2 res i Hyp_i)=(f (att T1 v i Hyp_i))}.
Proof.
```

Nous avons alors le but suivant :

```
1 subgoal

  ============================
   ∀ (T1 T2 :Set) (f:T1→T2) (v:Vector T1),
   {res:Vector T2 |
   ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt), att T2 res i Hyp_i = f (att T1 v i Hyp_i)}

parfun < intros T1 T2 f v.
```

Pour commencer nous introduisons les hypothèses et nous obtenons :

```
  T1 : Set
  T2 : Set
  f : T1→T2
  v : Vector T1
  ============================
   {res:Vector T2 |
   ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt), att T2 res i Hyp_i = f (att T1 v i Hyp_i)}

parfun < refine match (replicate _ f) with
          (exist p h) ⇒ (exist _ (apply _ _ p v) _) end.
```

Nous donnons alors la fonction répondant à la spécification, mais avec des trous correspondant aux preuves logiques. Le système **Coq** nous donne alors le but suivant à prouver pour accepter le terme :

```
  T1 : Set
  T2 : Set
  f : T1→T2
  v : Vector T1
  p : Vector (T1→T2)
  h : ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
      att (T1→T2) p i Hyp_i = f
  ============================
   ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
   att T2 (apply T1 T2 p v) i Hyp_i = f (att T1 v i Hyp_i)

parfun < intros i Hyp_i.
```

où encore une fois, nous introduisons les hypothèses :

```
  T1 : Set
  T2 : Set
  f : T1→T2
  v : Vector T1
  p : Vector (T1→T2)
  h : ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
      att (T1→T2) p i Hyp_i = f
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  ============================
   att T2 (apply T1 T2 p v) i Hyp_i = f (att T1 v i Hyp_i)

parfun < rewrite apply_def.
```

Nous pouvons alors réécrire la propriété formelle de **apply** avec l'axiome **apply_def** :

```
  T1 : Set
  T2 : Set
  f : T1→T2
  v : Vector T1
  p : Vector (T1→T2)
  h : ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
      att (T1→T2) p i Hyp_i = f
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  ============================
   att (T1→T2) p i Hyp_i (att T1 v i Hyp_i) = f (att T1 v i Hyp_i)

parfun < rewrite (h i Hyp_i).
```

Nous utilisons alors la spécification formelle de `replicate` qui est ici appelée `h`, afin de réécrire le terme de gauche. Nous obtenons :

```
  T1 : Set
  T2 : Set
  f : T1→T2
  v : Vector T1
  p : Vector (T1→T2)
  h : ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
      att (T1→T2) p i Hyp_i = f
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  ============================
   f (att T1 v i Hyp_i) = f (att T1 v i Hyp_i)

parfun < trivial.
Proof completed.
```

qui est prouvé **trivial**ement. Nous pouvons afficher quel terme correspond à la preuve de cette spécification :

```
parfun =
fun (T1 T2:Set) (f:T1→T2) (v:Vector T1) ⇒
let (p, h) := replicate (T1→T2) f in
exist
  (fun res:Vector T2 ⇒
   ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
   att T2 res i Hyp_i = f (att T1 v i Hyp_i)) (apply T1 T2 p v)
  (fun (i:Z) (Hyp_i:0<=i<bsp_p tt) ⇒
   eq_ind_r (fun t:T2 ⇒ t = f (att T1 v i Hyp_i))
     (eq_ind_r
        (fun t:T1 → T2 ⇒ t (att T1 v i Hyp_i) = f (att T1 v i Hyp_i))
        (refl_equal (f (att T1 v i Hyp_i))) (h i Hyp_i))
     (apply_def T1 T2 p v i Hyp_i))
     : ∀ (T1 T2:Set) (f:T1→T2) (v:Vector T1),
       {res:Vector T2 |
       ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
       att T2 res i Hyp_i = f (att T1 v i Hyp_i)}
```

6.4.3 Fonction de communication certifiée utilisant un proj

Notre dernier exemple d'un développement complet d'une fonction BSML en **Coq** est celui d'une diffusion directe répliquée. Celle-ci se fait à l'aide d'un **proj**. Pour commencer nous avons besoin d'une fonction qui permet l'application d'une fonction à toutes les composantes d'un vecteur parallèle et qui applique une autre fonction sur une composante donnée. Sa spécification logique est alors :

$$\forall \text{T1,T2 : Set } \forall n \in \mathbb{Z}\ (0 \leq n < p)\ \forall f1, f2 : \text{T1} \to \text{T2}\ \forall vec \Rightarrow$$
$$\exists res \text{ tel que } \forall i\ (0 \leq i < p) \begin{cases} (\text{res}[i] = (f1\ vec[i])) & \text{si } i = n \\ (\text{res}[i] = (f2\ vec[i])) & \text{si } i \neq n \end{cases}$$

Celle-ci se traduit en **Coq** (notons l'utilisation d'un $\wedge$ pour différencier les deux cas) de la manière suivante :

```
Definition applyat : ∀ (T1 T2:Set) (root:Z) (Hyp_root:0<=root<(bsp_p tt))
                       (f1:T1→T2) (f2:T1→T2) (vec:(Vector T1)),
  {res:(Vector T2) | ∀ (i:Z) (Hyp_i:0<=i<(bsp_p tt)),
```

```
          ((i=root)→((att T2 res i Hyp_i)=(f1 (att T1 vec i Hyp_i)))) ∧
          ((i<>root)→((att T2 res i Hyp_i)= (f2 (att T1 vec i Hyp_i))))}.
Proof.
intros T1 T2 root Hyp_root f1 f2 vec.
refine (exist _ (apply _ _ (mkpar _ (fun i ⇒ fun v ⇒ match (Z_eq_dec i root) with
       left H1 ⇒ (f1 v)
    | right H2 ⇒ (f2 v) end)) vec) _).
```

Pour prouver cette spécification, nous introduisons les hypothèses puis nous donnons un terme BSML qui pourrait convenir (les trous _ sont les types et les fragments de preuves manquants). Le système **Coq** demande alors la preuve du but suivant ·

```
  T1 : Set
  T2 : Set
  root : Z
  Hyp_root : 0<=root<bsp_p tt
  f1 : T1→T2
  f2 : T1→T2
  vec : Vector T1
  ============================
   ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
   (i=root →
    att T2
      (apply T1 T2
         (mkpar (T1→T2)
            (fun (i0:Z) (v:T1) ⇒ if Z_eq_dec i0 root then f1 v else f2 v))
         vec) i Hyp_i = f1 (att T1 vec i Hyp_i)) ∧
   (i<>root →
    att T2
      (apply T1 T2
         (mkpar (T1→T2)
            (fun (i0:Z) (v:T1) ⇒ if Z_eq_dec i0 root then f1 v else f2 v))
         vec) i Hyp_i = f2 (att T1 vec i Hyp_i))
```

```
applyat < intros i hyp_i;split;intro h.
```

Nous pouvons alors de nouveau introduire les hypothèses et avec **split** générer les buts des deux cas de la spécification. Les deux cas sont similaires et nous ne donnons que le premier :

```
  T1 : Set
  T2 : Set
  root : Z
  Hyp_root : 0<=root<bsp_p tt
  f1 : T1→T2
  f2 : T1→T2
  vec : Vector T1
  i : Z
  hyp_i : 0<=i<bsp_p tt
  h : i=root
  ============================
   att T2
     (apply T1 T2
        (mkpar (T1→T2)
           (fun (i0:Z) (v:T1) ⇒ if Z_eq_dec i0 root then f1 v else f2 v))
        vec) i hyp_i = f1 (att T1 vec i hyp_i)
```

```
applyat < rewrite apply_def;rewrite mkpar_def.
```

Pour prouver ce cas, nous réécrivons préalablement les primitives **mkpar** et **apply**, ce qui nous donne :

```
  T1 : Set
  T2 : Set
  root : Z
  Hyp_root : 0<=root<bsp_p tt
  f1 : T1→T2
  f2 : T1→T2
  vec : Vector T1
  i : Z
  hyp_i : 0<=i<bsp_p tt
  h : i=root
  ============================
   (if Z_eq_dec i root
```

```
    then f1 (att T1 vec i hyp_i)
    else f2 (att T1 vec i hyp_i)) = f1 (att T1 vec i hyp_i)

applyat < case (Z_eq_dec i root);intro h';intuition.
```

Nous avons à nouveau deux cas qui sont discriminés par la tactique **case**. Ceux-ci sont alors basiques et résolus automatiquement par la tactique **intuition**.

Maintenant, nous pouvons commencer l'écriture de la diffusion directe. Nous commençons par la spécification logique suivante :

$$\forall \text{T} : \text{Set}\ \forall r \in \mathbb{Z}\ (0 \leq r < p)\ \forall vec : (\text{option T}) \Rightarrow \exists sv \text{ tel que } sv = (\text{Some } res[r])$$

Notons que dans cette spécification, nous avons encore le Some nécessaire à la primitive **proj**. Cette spécification se traduit ainsi :

```
Definition rpl_bcast_direct_tmp: ∀ (T:Set) (root:Z)
                                    (Hyp_root:0<=root<(bsp_p tt)) (vec:(Vector T)),
  {v:(option T) | ∀ (i:Z) (Hyp_i:0<=i<(bsp_p tt)),
                          v=(Some (att T vec root Hyp_root)) }.
Proof.
intros T root Hyp_root vec.
refine match (applyat _ _ root Hyp_root (fun v ⇒ (Some v)) (fun _ ⇒ None) vec) with
   (exist p h) ⇒ (exist _ ( (proj _ p) root) _) end.
```

Comme à l'accoutumée, nous introduisons les hypothèses de départ puis nous donnons le terme avec trous qui, nous l'espérons, doit satisfaire la spécification. Nous avons alors le but suivant :

```
  T : Set
  root : Z
  Hyp_root : 0<=root<bsp_p tt
  vec : Vector T
  p : Vector (option T)
  h : ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
      (i=root → att (option T) p i Hyp_i = Some (att T vec i Hyp_i)) ∧
      (i<>root → att (option T) p i Hyp_i = None)
  ============================
   ∀ i:Z,
   0<=i<bsp_p tt → proj T p root = Some (att T vec root Hyp_root)

rpl_bcast_direct_tmp < intros i Hyp_i; rewrite proj_def.
```

Nous introduisons les nouvelles hypothèses dans le contexte puis nous réécrivons la primitive **proj** et nous obtenons :

```
  T : Set
  root : Z
  Hyp_root : 0<=root<bsp_p tt
  vec : Vector T
  p : Vector (option T)
  h : ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
      (i=root → att (option T) p i Hyp_i = Some (att T vec i Hyp_i)) ∧
      (i<>root → att (option T) p i Hyp_i = None)
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  ============================
   match within_bound root with
   | left H1 ⇒ att (option T) p root H1
   | right _ ⇒ None (A:=T)
   end = Some (att T vec root Hyp_root)

rpl_bcast_direct_tmp < case (within_bound root);[ intro h' | intuition].
                       elim (h root h');intros H1 H2.
```

Nous faisons alors une preuve par cas sur $(0 \leq r < p)$. Le cas $\neg(0 \leq r < p)$ est résolu automatiquement par **Coq** avec `intuititon`. Pour l'autre cas, nous utilisons l'hypothèse `h` avec `root` et la preuve `Hyp_root` afin d'introduire dans le contexte deux nouveaux faits. Nous obtenons :

```
  T : Set
  root : Z
  Hyp_root : 0<=root<bsp_p tt
```

```
  vec : Vector T
  p : Vector (option T)
  h : ∀ (i:Z) (Hyp_i : 0<=i<bsp_p tt),
      (i=root → att (option T) p i Hyp_i = Some (att T vec i Hyp_i)) ∧
      (i<>root → att (option T) p i Hyp_i = None)
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  h' : 0<=root<bsp_p tt
  H1 : root=root → att (option T) p root h' = Some (att T vec root h')
  H2 : root<>root → att (option T) p root h' = None
  ============================
   att (option T) p root h' = Some (att T vec root Hyp_root)

rpl_bcast_direct_tmp < generalize (H1 (refl_equal root)); intro H.
```

Nous pouvons utiliser H1 pour introduire un nouveau fait H. Nous utilisons pour cela le lemme **Coq** :

```
refl_equal: ∀ (A : Type) (x : A), x = x
```

Nous avons alors :

```
  T : Set
  root : Z
  Hyp_root : 0<=root<bsp_p tt
  vec : Vector T
  p : Vector (option T)
  h : ∀ (i:Z) (Hyp_i:0<=i<bsp_p tt),
      (i=root → att (option T) p i Hyp_i = Some (att T vec i Hyp_i)) ∧
      (i<>root → att (option T) p i Hyp_i = None)
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  h' : 0<=root<bsp_p tt
  H1 : root=root → att (option T) p root h' = Some (att T vec root h')
  H2 : root<>root → att (option T) p root h' = None
  H : att (option T) p root h' = Some (att T vec root h')
  ============================
   att (option T) p root h' = Some (att T vec root Hyp_root)

rpl_bcast_direct_tmp < rewrite (at_H T vec root Hyp_root h').
                       apply H.
Proof completed.
```

A ce moment nous ne pouvons pas appliquer directement l'hypothèse H car dans celle-ci, l'hypothèse Hyp_root est utilisée, alors que dans le but c'est l'hypothèse h' qui est partout utilisée. Les termes ne sont donc pas unifiables car **Coq** ne permet pas l'unification de termes où les sous-termes dans Prop ne sont pas identiques (*proof revelant* dans la littérature anglo-saxonne). Nous utilisons alors l'axiome at_H, qui dit que l'accès à une composante d'un vecteur n'est pas dépendant de la preuve (dans Prop) tant que l'accès se fait bien dans les bornes du vecteur. Le terme (en tant que but) qui a été réécrit est alors unifiable avec H. Nous appliquons alors directement H sur le but afin de terminer la preuve.

Pour terminer la diffusion, il nous faut supprimer le constructeur Some du résultat. Pour cela, nous allons utiliser une fonction noSome qui est partielle car elle ne pourra être utilisée que si l'on fournit une preuve que son argument n'est pas None. Sa spécification est tout simplement :

$$\forall T : \text{Set}\ \forall sv : (\text{option T})\ (sv \neq \text{None}) \Rightarrow \exists v \text{ tel que } sv = (\text{Some } v)$$

Sa définition en **Coq** est alors (nous donnons aussi deux lemmes techniques triviaux qui seront utilisés par la suite) :

```
Definition noSome: ∀ (T:Set)(sv:(option T)), (sv<>None)→{v:T | sv=(Some v)}.

Lemma egal_some: ∀ (T:Set)(a b:T), ((Some a)=(Some b)) → (a=b).

Lemma Some_donc_pas_None: ∀ (T:Set)(sv:(option T))(v:T), (sv=(Some v))→(sv<>None).
```

Avec cette nouvelle fonction, nous pouvons alors supprimer le constructeur Some, du résultat de rpl_bcast_direct_tmp et avoir la spécification suivante :

$$\forall \text{T} : \text{Set}\ \forall r \in \mathbb{Z}\ (0 \leq r < p)\ \forall vec : (\text{option T}) \Rightarrow \exists sv \text{ tel que } sv = res[r]$$

ce qui ce traduit ainsi en **Coq** :

```
Definition rpl_bcast_direct: ∀ (T:Set) (root:Z)
                                (Hyp_root:0<=root<(bsp_p tt)) (vec:(Vector T)),
  {v:T | ∀ (i:Z) (Hyp_i:0<=i<(bsp_p tt)), v=(att T vec root Hyp_root)}.
Proof.
intros T root Hyp_root vec.
refine match (rpl_bcast_direct_tmp _ root Hyp_root vec) with
 (exist sv h) ⇒ match (noSome _ sv  (Some_donc_pas_None _ sv _ (h root Hyp_root))) with
  (exist v h') ⇒ (exist _ v _) end end.
intros i Hyp_i.
```

Nous introduisons nos hypothèses et donnons le terme (avec trous) qui est la composition de `rpl_bcast_direct_tmp` et `noSome`. Notons que nous utilisons `Some_donc_pas_None` afin de donner la preuve que le résultat n'est pas `None`. Le système **Coq** nous demande alors de prouver le but suivant :

```
  T : Set
  root : Z
  Hyp_root : 0<=root<bsp_p tt
  vec : Vector T
  sv : option T
  h : ∀ i:Z, 0<=i<bsp_p tt → sv = Some (att T vec root Hyp_root)
  v : T
  h' : sv = Some v
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  ============================
   v = att T vec root Hyp_root
```

```
rpl_bcast_direct < generalize (h i Hyp_i); rewrite h'; intro H.
```

Pour cela, nous utilisons `h` avec `i` et `Hyp_i` afin d'avoir un fait sur le résultat de `rpl_bcast_direct_tmp`. Puis nous insérons `h'` dans cette généralisation afin de remplacer `sv` par sa valeur. Nous obtenons :

```
  T : Set
  root : Z
  Hyp_root : 0<=root<bsp_p tt
  vec : Vector T
  sv : option T
  h : ∀ i:Z, 0<=i<bsp_p tt → sv = Some (att T vec root Hyp_root)
  v : T
  h' : sv = Some v
  i : Z
  Hyp_i : 0<=i<bsp_p tt
  H : Some v = Some (att T vec root Hyp_root)
  ============================
   v = att T vec root Hyp_root
```

```
rpl_bcast_direct < apply (egal_some T v (att T vec root Hyp_root) H).
Proof completed.
```

dont le but peut être facilement unifié avec une application de `egal_some` sur l'hypothèse `H`.

6.4.4 Composition de vecteurs parallèles

Nonobstant l'implantation de fonctions certifiées, l'assistant de preuves **Coq** peut aussi être utilisé pour prouver des propriétés de fonctions BSML.

Pour illustrer cette possibilité, nous donnons, dans cette section, les preuves en **Coq** de propriétés d'une fonction qui est une forme faible de la composition parallèle ([188]) analogue à une conditionnelle data-parallèle pour couvrir le réseau entier (une condition suffisante pour une barrière de synchronisation implicite). Cette fonction a été utilisée dans l'article [196] afin d'illustrer la possibilité de vérification des propriétés de programme BSML en utilisant les règles de la théorie équationnelle du BSλ-calcul. Mais sans l'aide d'un assistant de preuves, les auteurs avaient fait quelques erreurs. La fonction est implantée en BSML par :

```
(* mask :(int →bool)→α par→α par→α par *)
let mask c x y = apply (apply (mkpar (fun j →
 if (c j) then (fun x0 y0 →x0) else (fun x0 y0 →y0))) x ) y
```

Nous pouvons l'écrire de la manière suivante en **Coq** :

```
Implicit Arguments mkpar.
Implicit Arguments apply.

Definition mask := fun (T:Set) (c:Z→bool) (x y: (Vector T)) ⇒
 apply (apply (mkpar (fun (j:Z) ⇒ match (c j) with
                                  true ⇒ (fun (x0:T) (y0:T) ⇒x0)
                                | false ⇒ (fun (x0:T) (y0:T) ⇒y0) end)) x) y.
```

[196] propose alors le lemme[8] suivant :

Lemme 33

$$\mathit{mask}\ c\ (\pi f)\ (\pi g) = \pi(\lambda j.\ \mathit{if}\ (c\ j)\ \mathit{then}\ f\ j\ \mathit{else}\ g\ j).$$

qu'ils prouvent alors «à la main», à l'aide des règles du BSλ-calcul : « Substitution de (πf) et (πg) dans la définition de mask, suivie par l'application de la règle de $(\mathbf{apply})$, (β)-contraction locale, une autre application de la règle $(\mathbf{apply})$ (correspondant à la variable Y dans la définition) et finalement une (β)-contraction locale. » *dixit* [196].

Avec l'aide du système **Coq**, nous pouvons aussi prouver cette propriété, mais de façon plus sûre, car le système ne nous permettra pas d'oublier un cas ou une application d'une règle :

```
Lemma mask_is_composition : ∀ (T:Set) (c:(Z→bool)) (f g:Z→T)
                                    (i:Z) (H_i:0<=i<(bsp_p tt)),
 (att T (mask T c (mkpar f) (mkpar g)) i H_i)
=
 (att T (mkpar (fun j:Z ⇒ match (c j) with
                                  true ⇒ (f j)
                                | false ⇒ (g j) end)) i H_i).
Proof.
intros T c f g i H_i.
unfold mask.
autorewrite with BSML_rules.
case (c i);trivial.
Qed.
```

Remarquons que nous nous intéressons à toutes les composantes des vecteurs parallèles avec le `(i:Z)` et le **att**. Nous donnons par la suite des explications sur ces tactiques.

Les auteurs de [196] proposent ensuite le lemme suivant :

Lemme 34

$$\mathit{mask}\ c\ (\pi f \# \pi f')\ (\pi g \# \pi g') = (\mathit{mask}\ c\,(\pi f)\,(\pi g)) \# (\mathit{mask}\ c\,(\pi f')\,(\pi g')).$$

qu'ils prouvent toujours «à la main» à l'aide des règles du BSλ-calcul :

«

mask $c\ (\pi f \# \pi f')\ (\pi g \# \pi g')$

$$\begin{array}{rl}
= & \pi(\lambda j.\ \text{if } (c\ j) \text{ then } (\pi f \# \pi f')\ j \text{ else } (\pi g \# \pi g')\ j) \text{ par le lemme 33} \\
\overset{(\mathbf{apply})}{=} & \pi(\lambda j.\ \text{if } (c\ j) \text{ then } (f\ j)(f'\ j) \text{ else } (g\ j)(g'\ j)) \\
= & \pi(\lambda j.\ (\text{if } (c\ j) \text{ then } (f\ j) \text{ else } (g\ j))(\text{if } (c\ j) \text{ then } (f'\ j) \text{ else } (g'\ j)) \\
 & \text{par énumération sur } (c\ j) \\
\overset{(\mathbf{apply})}{=} & [\pi(\lambda j.\ \text{if } (c\ j) \text{ then } (f\ j) \text{ else } (g\ j))] \#[\pi(\lambda j.\ \text{if } (c\ j) \text{ then } (f'\ j) \text{ else } (g'\ j))] \\
= & (\text{mask } c\,(\pi f)\,(\pi g)) \# (\text{mask } c\,(\pi f')\,(\pi g')) \text{ par le lemme 33, deux fois.}
\end{array}$$

»

Même si le lemme est correct, la preuve, elle, est incorrecte dès la première égalité entre mask et le BSλ-terme. Les auteurs ont donc commis une erreur difficilement détectable sans l'aide d'un assistant de preuves. En **Coq**, nous pouvons formaliser le lemme de la manière suivante :

```
Lemma mask_is_composition_apply : ∀ (T:Set) (c:(Z→bool)) (f g:Z→T→T)
                                        (f' g':Z→T) (i:Z) (H_i:0<=i<(bsp_p tt)),
 ((att T (mask T c (apply (mkpar f) (mkpar f'))
           (apply (mkpar g) (mkpar g'))) i H_i)
```

[8] En BSλ-calcul, le terme (πf) correspond à notre $(\mathbf{mkpar}\ f)$ et $(e_1 \# e_2)$ à $(\mathbf{apply}\ e_1\ e_2)$.

```
=
 (att T (apply (mask (T→T) c (mkpar f) (mkpar g))
               (mask T c (mkpar f') (mkpar g'))) i H_i).
Proof.
intros T c f g f' g' i H_i.
```

Nous commençons d'abord par introduire les hypothèses dans le contexte. Nous obtenons :

```
  T : Set
  c : Z→bool
  f : Z→T→T
  g : Z→T→T
  f' : Z→T
  g' : Z→T
  i : Z
  H_i : 0<=i<bsp_p tt
  ============================
   att T (mask T c (apply (mkpar f) (mkpar f')) (apply (mkpar g) (mkpar g')))
     i H_i =
   att T
     (apply (mask (T → T) c (mkpar f) (mkpar g))
        (mask T c (mkpar f') (mkpar g'))) i H_i

mask_is_composition_apply < unfold mask.
                            autorewrite with BSML_rules.
```

puis nous réduisons mask. Cela fait apparaître un grand nombre de **mkpar** et de **apply**. Les réécrire un par un avec les axiomes serait laborieux. Nous utilisons alors la tactique **autorewrite** qui va réécrire toutes les primitives avec nos axiomes. Pour cela nous avons préalablement entré les axiomes dans un «schéma de règles de réécriture» :

```
Hint Rewrite mkpar_def apply_def put_def proj_def super_def : BSML_rules.
```

Nous obtenons alors :

```
  T : Set
  c : Z → bool
  f : Z → T → T
  g : Z → T → T
  f' : Z → T
  g' : Z → T
  i : Z
  H_i : 0<=i<bsp_p tt
  ============================
   (if c i then fun x _ : T ⇒ x else fun _ y : T ⇒ y)
     (f i (f' i)) (g i (g' i)) =
   (if c i then fun x _ : T → T ⇒ x else fun _ y : T → T ⇒ y)
     (f i) (g i)
     ((if c i then fun x _ : T ⇒ x else fun _ y : T ⇒ y) (f' i) (g' i))

mask_is_composition_apply < case (c i); trivial.
Proof completed.
```

Pour terminer la preuve, il nous suffit de prouver les deux cas pour `(c i)` (`true` ou `false`) qui seront générés à l'aide de la tactique **case**. Ces deux preuves sont alors **trivial**es et **Coq** peut les construire automatiquement. Notons que, comme dans la preuve faite à la main, nous devons prouver les deux cas de `(c i)` et que nous n'avons pas eu besoin du premier lemme. Nous avons donc pu prouver le lemme et cela sans erreur (assuré par **Coq**).

6.5 Création d'une bibliothèque BSMLlib certifiée

Nous présentons uniquement ici, par souci de clarté et de concision, le développement de sept fonctions parallèles de la bibliothèque BSMLlib dans le système **Coq** qui apparaissent fréquemment dans un algorithme BSP. Cette étude de cas est un exemple de la pertinence de l'utilisation du système **Coq** dans la preuve de programmes parallèles (les fonctions n'ont pas toutes été spécifiées dans un autre formalisme et encore moins complètement spécifiées dans un assistant de preuves).

```
Definition totex : ∀ (T:Set) (vec:(Vector T)),
 {res:(Vector (Z→(option T))) | ∀ (i:Z) (Hyp_i:0<=i<(bsp_p tt)) (src:Z),
   (∀ (Hyp_src:0<=src<(bsp_p tt)),
     ((att (Z→(option T)) res i Hyp_i) src)=(Some (att T vec src Hyp_src)))
∧
   (∀ (Hyp_src:¬(0<=src<(bsp_p tt))),
     ((att (Z→(option T)) res i Hyp_i) src) = None)}.

Proof.
intros T vec. (* introduction des hypothèses *)
refine match (parfun _ _ (fun (data:T) (dst:Z) ⇒ (Some data)) vec) with
  (exist p h) ⇒ (exist _ (put _ p) _) end.    (* donner le programme *)
intros i Hyp_i src. (* introduire les nouvelles hypothèses *)
(* résoudre les deux cas en ayant préalablement ré-écrit les primitives *)
split;intro Hyp_src;autorewrite with BSML_rules.
(* premier cas     0<=src<(bsp_p tt) *)
case (within_bound src);[ intro H_src | intuition ].
  (* premier sous-cas *)
  rewrite (h src H_src).
  rewrite (at_H T vec src H_src Hyp_src);auto.
  (* second sous-cas résolu par intuition *)
(* second cas      ¬(0<=src<(bsp_p tt)) *)
case (within_bound src);[ intro H_src;intuition | trivial ].
Defined.
```

Figure 6.2 — Réalisation d'un échange total

6.5.1 Echange total

Notre premier exemple est un échange total des valeurs d'un vecteur parallèle. Cela peut être représenté schématiquement par :

$$\text{total\_exchange} \boxed{v_0 \mid \cdots \mid v_{p-1}} = \boxed{[v_0,\ldots,v_{p-1}] \mid \cdots \mid [v_0,\ldots,v_{p-1}]}$$

Pour cela nous commençons avec une fonction `totex` qui retourne un vecteur de fonctions renvoyant toutes les valeurs du vecteur initial. Sa spécification est la suivante :

$$\forall \text{Data}\, \forall vect \Rightarrow \exists res \text{ tels que } \forall i\ (0 \leq i < p)\, \forall src \begin{cases} (\text{res}[i]\ src) = (\text{Some vec}[src]) & \text{si } (0 \leq src < p) \\ (\text{res}[i]\ src) = \text{None} & \text{si } \neg(0 \leq src < p) \end{cases}$$

La figure 6.2 donne la spécification en **Coq** et les tactiques nécessaires pour une réalisation de la spécification. Notons l'utilisation du $\wedge$ pour discriminer les deux cas.

Nous pouvons alors donner la spécification de la fonction d'échange total :

$$\begin{aligned} &\forall \text{Data}\, \forall vect \Rightarrow \exists res \text{ tels que } \forall i\ (0 \leq i < p) \\ &\qquad (\text{length } src[i] = p) \wedge \forall src\ (0 \leq src < p)\ (\text{nth res}[src] = \text{vec}[src]) \end{aligned}$$

Cette spécification a été prouvée en utilisant les propriétés formelles des listes (la fonction nth nécessite la preuve que l'entier donné en paramètre soit compris entre 1 et la taille de la liste) et en utilisant la spécification de `totex`.

6.5.2 Rassemblement d'un vecteur parallèle

La fonction `gather_list` est une fonction qui rassemble les valeurs $v_0,\ldots,v_{p-1}$ (une par processeur) dans une liste sur un processeur $root$ donné en paramètre :

$$\texttt{gather\_list}\ root \boxed{v_0 \mid \cdots \mid v_{root} \mid \cdots \mid v_{p-1}} = \boxed{[] \mid \cdots \mid [v_0;\cdots;v_{root};\cdots;v_{p-1}] \mid \cdots \mid []}$$

Pour cela, nous utilisons une fonction intermédiaire, `gather` (dont la spécification et la réalisation sont données à la figure 6.3), qui donne un vecteur parallèle de fonctions. Celle au processeur $root$ renvoie la valeur du processeur $dest$ passé en paramètre, si celui-ci est un numéro de processeur valide (autrement, elle retourne None), et les fonctions aux processeurs $i \neq root$ retournent toujours None pour toute valeur de

```
Definition gather: ∀ (T:Set) (root:Z) (vv:(Vector T)),
 { res:(Vector (Z→(option T))) | ∀ (i:Z) (Hyp_i:0<=i<(bsp_p tt)),

 ( (i=root)→ ∀ (dest:Z) (H_dest:0<=dest<(bsp_p tt)),
              ((att _ res i Hyp_i) dest)=(Some (att _ vv dest H_dest)))
  ∧
 ((i<>root) → ∀ (dest:Z), ((att _ res i Hyp_i) dest)=None)}.

Proof.
intros T root vv.
refine (exist _ (put _ (apply _ _ (mkpar _
        (fun (pid:Z) (v:T) (dest:Z) ⇒ match (Z_eq_dec dest root) with
           left H1 ⇒ (Some v)
         | right H2 ⇒ None end)) vv)) _).
intros i Hyp_i.
split; [intros H_i dest H_dest | intros H_i dest];rewrite put_def.

(* premier cas i=root*)
case (within_bound dest); [intro H_dest' | intuition].
autorewrite with BSML_rules.
case (Z_eq_dec i root); [intro H_root | intuition].
rewrite (at_H T vv dest H_dest H_dest').
trivial.

(* second cas i<>root *)
case (within_bound dest); [intro H_dest' | intuition].
autorewrite with BSML_rules.
case (Z_eq_dec i root); intuition.
Defined.
```

Figure 6.3 — Réalisation d'un rassemblement

$dest$. Ainsi, au processeur $root$, nous pouvons appliquer cette fonction à une liste contenant les numéros de processeur dans le cas du processeur $root$ et une liste vide dans le cas des autres processeurs. Nous avons alors le vecteur parallèle désiré. La spécification formelle de `gather_list` est :

$$\begin{gathered}\forall \text{Data}\ \forall vect\ \forall root\ (0 \leq root < p) \Rightarrow \exists res \text{ tels que} \\ \exists l{:}list \text{ tels que } l = res[root]\ \wedge\ H{:}(length\ l) = p \wedge \\ \forall j\ (0 \leq j < p)\ (nth\ l\ j\ H) = vect[j] \wedge\ \forall i(\ 0 \leq i < p)\ (i \neq root)\ res[i] = nil\end{gathered}$$

où `nth` est une fonction qui renvoie la n$^{\text{ième}}$ valeur de la liste l (utilisant la preuve que $j \leq (length\ l)$) et l est la liste au processeur $root$ de longueur p (le nombre de processeurs) et contenant les éléments désirés (sur les autres processeurs, la liste est vide). Cette spécification a été prouvée en utilisant la propriété de `gather`, par cas sur $i = root$ et avec un lemme technique qui donne une propriété sur la longueur des listes de chaque processus (celle-ci est de longueur p au processeur $root$, 0 autrement).

6.5.3 Demander et recevoir des valeurs d'autres processeurs

Notre troisième cas est le développement d'un opérateur `get_list`, dual de l'opérateur **put**. En effet, échanger des valeurs est l'un des points critiques d'un algorithme BSP. Cette fonction permet à un processus de recevoir les valeurs d'autres processus, ces valeurs étant dans les composantes d'un vecteur parallèle (chaque composante correspond à un processeur) et peut être représentée par le schéma suivant :

$$\texttt{get\_list}\ \boxed{v_0 \mid \cdots \mid v_{p-1}}\ \boxed{l_0 \mid \cdots \mid l_{p-1}} = \boxed{l'_0 \mid \cdots \mid l'_{p-1}}$$

tel que si $(\texttt{nth}\ l_i) = j$ alors $(\texttt{nth}\ l'_i) = v_j$ pour $0 \leq i < p$. Pour une simple raison, cette fonction est implantée avec deux **put** : les processus, à la première super-étape, envoient des requêtes aux processus désirés pour obtenir leurs valeurs puis, à la seconde super-étape, ceux-ci envoient leurs valeurs aux processus

qui leur ont envoyé une requête. Nous avons alors la spécification logique suivante :

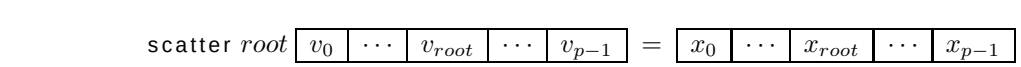
$$\begin{array}{c}\forall \text{Data}\ \forall vec{:}(Vector\ Data)\ \forall lpids{:}(Vector\ (list\ Z))\\ (\forall i\ H{:}(0{\leq}\, i{<}p)(lpids[i] = nil)\ \vee\\ ((\forall n\ H_n{:}(1{\leq}\, n{<}length(lpids[i])) \rightarrow (0{\leq}(nth\ lpids[i]\ n\ H_n){<}p)))\\ \Rightarrow \exists res \text{ tels que } \forall i\ H{:}(0{\leq}i{<}p)\ length(res[i]) = length(lpids[i])\ \wedge\\ (\forall n\ H_n{:}(1{\leq}\, n{<}length(lpids[i])) \rightarrow\\ (nth\ res[i]\ n\ H_n) = vec[(nth\ lpids[i]\ n\ H_n)]\end{array}$$

où nous donnons le vecteur des éléments $\forall vec : (Vector\ Data)$ puis le vecteur des listes de noms de processus $\forall lpids : (Vector\ (list\ Z))$. Ces listes, en chaque processus, sont vides ($lpids[i] = nil$) ou contiennent des noms valides de processeurs, c'est-à-dire compris en 0 et $p-1$ ($(0{\leq}(nth\ lpids[i]\ n\ H_n){<}p)$ où H_n est la preuve que n est un entier compris entre 1 et la taille de la liste). Le résultat est un vecteur de listes (d'éléments) qui sont de même longueur que la liste de départ ($length(res[i]) = length(lpids[i])$) et qui contiennent les éléments désirés. Cette spécification a été prouvée en utilisant les propriétés formelles des listes (nth nécessite une preuve que l'entier soit compris entre 1 et la taille de la liste) et par cas sur les numéros de processus des listes données en paramètres. On peut remarquer que la fonction est *partielle* puisqu'elle ne peut être appliquée (utilisée) que si les numéros de processus sont valides. Ceci permet de toujours fournir une preuve pour l'«accès» aux éléments du vecteur car chaque processeur lit les valeurs d'autres processeurs. Ces numéros de processeurs étant dans une liste, il est donc nécessaire que ces numéros soient compris entre 0 et $p-1$ (c'est-à-dire un pid de processeur valide)

6.5.4 Diffusion en 2 phases

Pour les données de grande taille, [117] propose un algorithme plus fin (plus efficace pour certains paramètres BSP), l'émission en deux super-étapes (illustrée par la figure 5.3) qui procède ainsi : le processeur de départ «découpe» son message en p messages et envoie chacun d'eux aux $p-1$ autres processeurs (première super-étape). L'algorithme dépose ainsi un morceau de la valeur initiale sur chaque processeur. Ensuite, chaque processeur envoie son bout de message aux autres processeurs (un échange total) et pour finir chaque processeur «recolle» les morceaux reçus.

L'envoi des messages est donc un «éparpillement» (`scatter`) de la valeur. Cette fonction utilise une fonction de `partition` pour définir où doivent être envoyées les valeurs. Schématiquement nous avons :



tel que pour tout i, (`partition` $v_{root}\ i) = y_i$. La spécification formelle est :

$$\begin{array}{c}\forall \text{Data1,Data2}\ \forall partition\ \forall root\ (0 \leq root < p)\ \forall vect \Rightarrow \exists res \text{ as}\\ \forall i(\ 0 \leq i < p)\ (Some\ \text{res}[i]) = (partition\ \text{vect}[root]\ i)\end{array}$$

La preuve de cette spécification est faite par cas sur l'égalité entre $root$ et le numéro de processeur. Avec les types dépendant de **Coq**, il est facile de prouver que les «accès» au vecteur sont valides. La figure 6.4 donne la spécification en **Coq** ainsi que la réalisation.

Revenons à notre diffusion. Celle-ci a donc la spécification suivante :

$$\begin{array}{c}\forall \text{Data1,Data2}\ \forall decoupe{:}(Data1 \rightarrow z \rightarrow (Data2\ option))\\ \forall recolle{:}(z \rightarrow Data2) \rightarrow Data1\ \ (H1{:}(\forall x\ \forall i\ \neg((partition\ x\ i) = None)))\\ (\forall x\ (recolle\ (\lambda i.(noSome\ (H_1\ x\ i))\ (decoupe\ x\ i))) = x)\\ \forall root\ (0{\leq}root{<}\ p)\ \forall vect \Rightarrow \exists res \text{ tels que } \forall i(\ 0{\leq}i{<}\ p)\ (\text{res}[i] = \text{vect}[root])\end{array}$$

où nous avons les fonctions $decoupe$ et $recolle$ telle que $decoupe$ donne toujours un élément (éventuellement vide) à envoyer à un processeur ($(\forall x\ \forall i\ \neg((partition\ x\ i) = None))$) et $recolle$, pour chaque morceau du message, retourne le message d'origine ($\forall x\ (recolle\ (\lambda i.(noSome\ (H_1\ x\ i))\ (decoupe\ x\ i))) = x$). L'utilisation du $noSome$ et donc de $H1$ vient du fait que l'on manipule des objets de type `option`. Pour construire notre réalisation de cette spécification, nous avons utilisé la fonction d'«éparpillement» qui effectue la première super-étape puis cette spécification est prouvée par cas sur les processeurs (s'ils font l'éparpillement ou non) et en utilisant les propriétés des paramètres pour reconstruire le message (après le «total-exchange», tous les bouts de message sont disponibles pour pouvoir appliquer la fonction $recolle$).

```
Definition scatter : ∀ (T1 T2:Set) (partition:(T1→Z→(option T2))) (root:Z)
                          (Hyp_root:0<=root<(bsp_p tt)) (v:(Vector T1))
  (Hyp:(∀ (i:Z) (H_i:0<=i<(bsp_p tt)), ¬(partition (att T1 v root Hyp_root) i)=None)),
 { res :(Vector (option T2)) | ∀ (i:Z) (Hyp_i:0<=i<(bsp_p tt)),
                (att _ res i Hyp_i)=(partition (att T1 v root Hyp_root) i)}.

Proof.
intros T1 T2 partition root Hyp_root v Hy_part.
refine match (replicate _ root) with
 (exist p h) ⇒ (exist _
 (apply _ _ (put _ (apply _ _ (mkpar _ (fun (pid:Z) ⇒
    match (Z_eq_dec pid root) with
          left H1 ⇒ partition
       | right H2 ⇒(fun (a:T1) (i:Z) ⇒ None) end)) v)) p) _) end.
intros i Hyp_i.
(* ré-écrire les primitives *)
autorewrite with BSML_rules.
(* tester les cas "nom de processeur" valide ou non *)
case (within_bound (att Z p i Hyp_i)); intro H_at;autorewrite with BSML_rules.
(* premier cas *)
(* cas sur root *)
case (Z_eq_dec (att Z p i Hyp_i)); intro H_at'.
  (* premier sous-cas *)
  (* utiliser l'hypothèse sur root *)
  induction H_at'.
  rewrite (at_H T1 v (att Z p i Hyp_i) H_at Hyp_root);trivial.
  (* second sous-cas, absurde*)
  absurd ((att Z p i Hyp_i) <> root);auto.
(* second cas *)
(* on introduit l'hypothèse sur root *)
generalize (h i Hyp_i);intro H_root.
(* cas absurde *)
absurd (¬(0 <= (att Z p i Hyp_i) < (bsp_p tt)));intuition.
Defined.
```

Figure 6.4 — Réalisation d'un éparpillement

Cet algorithme est plus efficace que l'algorithme naïf suivant les paramètres BSP de la machine utilisée (voir [117] pour plus de détails).

Cette version de la diffusion peut être spécialisée, par exemple pour les listes, en donnant les fonctions `decoupe` et `recolle` adéquates. Dans la BSMLlib, une version polymorphe a été donnée en utilisant le module Marshall afin de transformer (resp décoder) n'importe quelle valeur en (resp depuis) une chaîne de caractères. Les fonctions `decoupe` et `recolle` sont alors définies sur les String. Malheureusement, ces fonctions du module Marshall ne sont pas sûres et n'existent pas en **Coq** : on pourrait utiliser n'importe quelle valeur avec n'importe quelle autre valeur. Il faudra donc en **Coq** définir autant de `bcast_2phases` que de types de valeurs à diffuser.

6.5.5 Réduction directe

La fonction `scan` est un algorithme parallèle classique. La fonction utilise un opérateur binaire `R` et informellement :

$$\text{scan}\ \boxed{v_0}\boxed{\cdots}\boxed{v_i}\boxed{\cdots}\boxed{v_{p-1}} = \boxed{s_0}\boxed{\cdots}\boxed{s_i}\boxed{\cdots}\boxed{s_{p-1}}$$

tel que $s_i = R_{0 \leq k < p} v_k$, c'est-à-dire la réduction de l'opérateur binaire R sur tous les éléments v_k, $0 \leq k < p$. On peut donc en déduire la spécification logique suivante :

$$\begin{array}{c} \forall \text{Data}\ \forall vec\ \forall R : Data \rightarrow Data \rightarrow Data \Rightarrow \\ \exists res \text{ tels que } \forall i\ H : (0 \leq i < p) \\ (k\_first\_R\ Data\ R\ vec\ (p-1)\ (0 \leq (p-1) < p)\ res[i]) \end{array}$$

qui utilise le type inductif suivant :

```
Inductive k_first_R (T:Set) (R:T→T→T) (v:(Vector T))
          : ∀ (k:Z) (H_r:0<=k<(bsp_p tt)) (res:T), Prop :=
```

```
(* cas de départ *)
   k_zero: (k_first_R T R v 0 H_0 (att T v 0 H_0))
(* cas inductif *)
 | k_rec : (k:Z) (H_r:(0<=k<(bsp_p tt))) (a:T) (H_sub:(k>0))
                 (k_first_R T R v (k-1) (p_sub k H_r H_sub)  a)
             → (k_first_R T R v k H_r (R a (att T v k H_r))).
```

qui donne l'application de l'opérateur R aux k premiers éléments du vecteur parallèle avec p_sub qui est un lemme technique qui transforme une preuve de $0 \leq k < p$ et $k > 0$ en $0 \leq (k-1) < p$). Étant un inductif logique, il n'apparaît donc pas dans le programme ML extrait. Il est seulement employé ici pour vérifier les propriétés formelles. Ainsi nous pouvons utiliser la fonction d'«accès» sans problème. Le cas de départ décrit que si $k = 0$ alors on a juste l'accès à la valeur contenue dans la 0ième composante du vecteur. Le cas inductif décrit que pour tout $k > 0$, si le cas $k-1$ est possible alors le cas k est l'application de l'opérateur binaire sur le résultat du cas $k-1$ et sur la $k$ième composante du vecteur parallèle (valeur du $k$ième processeur). Pour construire notre réalisation de cette spécification, nous avons dû faire une démonstration par cas sur le nom du processus : le premier processus n'effectue pas de calcul tandis que les autres vont utiliser une fonction séquentielle (fold_left) sur les listes. Nous utilisons alors les propriétés (données par la bibliothèque standard PolyList) de cette fonction (interactions entre les éléments de la liste des valeurs du vecteur et les k premiers éléments donnés par le type inductif) pour terminer la preuve. Cette preuve se fait donc par induction sur le nombre de processeurs et en utilisant le type inductif pour prouver chaque cas.

6.5.6 Décalage

Le décalage des valeurs des processeurs est une fonction de communication qui est parfois utilisée dans certains algorithmes, notamment ceux sur les multiplications de matrices [257]. Chaque processeur lit la valeur du processeur qui le précède (*modulo* le nombre de processeur) :

$$\text{shift} \begin{array}{|c|c|c|c|c|c|} \hline v_0 & \cdots & v_{i-1} & v_i & \cdots & v_{p-1} \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|c|} \hline v_{p-1} & \cdots & v_{i-2} & v_{i-1} & \cdots & v_{p-2} \\ \hline \end{array}$$

ce qui être formellement décrit par :

$$\forall Data\ vec : (VectorData) \Rightarrow \exists res \text{ tels que } \forall i (0 \leq i < p)$$
$$(i = p-1 \Rightarrow (res[i] = (vec[0]))) \wedge (i \neq p-1 \Rightarrow (res[i] = (vec[i-1])))$$

ce qui être prouvé encore par cas sur i. Notons que nous pourrions utiliser la fonction get_list afin que chaque processeur récupère la valeur du processeur qui le précède. Mais get_list est une fonction qui utilise deux super-étapes, alors que dans notre cas, une seule super-étape suffit.

6.5.7 Fonction de tri parallèle

Trier des données est un problème classique des algorithmes parallèles. C'est un problème difficile et qui nécessite un grand nombre de constructions. Il n'est pas évident d'écrire un algorithme de tri parallèle correct car, même sans de subtiles optimisations, la moindre erreur a tout de suite des conséquences irrémédiables. Notre dernier exemple est donc le tri parallèle par échantillonnage (*parallel sampling sort algorithm*, PSRS, dans la litératture anglo-saxonne) de Schaeffer dans sa version BSP [257].

Prenons un ensemble d'éléments $\mathcal{X}$ (par exemple les entiers ou les chaînes de caractères). Nous assumons que tous les éléments de $\mathcal{X}$ soient différents. Nous notons $\langle a, b \rangle$ un intervalle ouvert, c'est-à-dire l'ensemble de tous les éléments $c \in \mathcal{X}$ tels que $a < c < b$. Nous supposons que la liste initiale x (à trier) a été partitionnée en p sous-listes $x^1, \ldots, x^p$ de taille n/p, une sous-liste par processeur (dans un vecteur parallèle). L'algorithme par échantillonnage procède comme suit.

À la première super-étape, toutes les sous-listes x^q sont triées indépendamment en chaque processeur q. Le problème consiste maintenant à réunir ces sous-listes pour avoir une liste triée. En chaque processeur, $p+1$ éléments régulièrement espacés de la sous-liste sont sélectionnés et constituent l'échantillon primaire (le premier et le dernier éléments de la sous-liste sont inclus dans cet échantillon). Nous notons cet échantillon primaire de la sous-liste x^q (au processeur q) par $\bar{x}_0^q, \ldots, \bar{x}_p^q$. Celui-ci découpe la sous-liste x^q en p blocs primaires (de taille n/p^2) que nous noterons $[\bar{x}_0^q, \bar{x}_1^q], \ldots, [\bar{x}_{p-1}^q, \bar{x}_p^q]$. Ensuite, un échange total des p échantillons primaires est effectué (donc $p \times (p+1)$ éléments échangés) et chaque processeur trie l'ensemble des éléments des échantillons primaires. Nous noterons ces sous-listes triées y^q.

Après ceci, à la seconde super-étape, de nouveau en chaque processeur, $p+1$ éléments de y^q sont sélectionnés de la même manière (le premier et le dernier élements sont de nouveau inclus) et constituent l'échantillon secondaire. Nous notons cet échantillons secondaire $\bar{\bar{x}}_0, \ldots, \bar{\bar{x}}_p$ (on peut remarquer que cet échantillon est le même sur tous les processeurs). L'échantillon secondaire partitionne les éléments de x (et pas de x^q) en p blocs (échantillons) secondaires qui correspondent aux intervalles ouverts $\langle \bar{\bar{x}}_0, \bar{\bar{x}}_1 \rangle, \ldots, \langle \bar{\bar{x}}_{p-1}, \bar{\bar{x}}_p \rangle$. Les blocs secondaires sont distribués sur les processeurs.

La troisième et dernière super-étape consiste à ce que chaque processeur q récupère les éléments (des autres processeurs) de l'intervalle ouvert $\langle \bar{\bar{x}}_q, \bar{\bar{x}}_{q+1} \rangle$ (avec $0 \leq q < p$).

Comme dans un tri séquentiel, l'algorithme PSRS nécessite une *relation* (`inf`) qui est décidable, transitive et anti-symétrique. L'algorithme nécessite une fonction permettant le tri séquentiel d'une liste, une autre fonction permettant de «fusionner» deux listes triées (nous pouvons utiliser les versions certifiées **Coq** [218])

Pour notre tri parallèle nous devons exprimer deux faits :

1. les listes finales de l'algorithme en chaque processeur sont triées,
2. toutes les valeurs contenues dans les listes initiales apparaissent dans les listes finales (l'algorithme peut permuter des valeurs mais ni en enlever ni en rajouter).

Pour les listes, ce fait est décrit dans **Coq** par `sorted` and `permutation`. Nous avons besoin d'exprimer ces relations entre le vecteur de listes initiales et le vecteur de listes finales. La figure 6.5 donne ces relations : `PSRS_sorted` est un prédicat qui peut être lu comme «chaque liste du vecteur parallèle est triée et le maximal de la $i$ème liste est inférieur au minimal de la $(i+1)$ème liste» (où `inf_list` est un prédicat logique d'infériorité : chaque élément de la première liste est inférieur aux éléments de la seconde liste). `PSRS_exchange` permet d'exprimer que tous les élémentes des listes du premier vecteur parallèle apparaissent dans le second vecteur (où `list_in_list` est aussi un prédicat logique qui permet d'exprimer l'inclusion de listes). Maintenant nous pouvons définir `PSRS_permut` comme la plus petite relation d'équivalence qui contient les transpositions (c'est-à-dire l'échange des éléments). Remarquons que lors de la première super-étape, l'algorithme ne modifie pas le vecteur de listes et que donc la réflexivité de la permutation est nécessaire. De la même manière, la transitivité de la relation permet de conserver le prédicat logique tout au long de la séquence de super-étapes.

La spécification du tri parallèle est donc l'existence d'un vecteur parallèle de listes qui sont triées et qui est une permutation du vecteur initial :

$$\forall \text{Data}\ \forall vec : ((listData)\ Vector)\ \forall inf : Data \to Data \to Data \Rightarrow \exists res : ((listData)\ Vector)$$
$$\text{tels que } (\text{PSRS\_sorted}\ res) \wedge (\text{PSRS\_exchange}\ vec\ res) \wedge (\text{PSRS\_permut}\ vec\ res)$$

La réalisation de cette spécification demande plusieurs lemmes techniques : un qui permet de prouver que la première super-étape conserve le prédicat `PSRS_permut` et un autre qui dit que la troisième super-étape conserve aussi ce prédicat et apporte le prédicat logique `PSRS_sorted`. Les démonstrations sont faites à l'aide des propriétés des fonctions de communication précédentes (notamment l'échange total) et par cas sur la longueur des listes échangées.

6.5.8 Extraction de code

Comme nous l'avons expliqué précédemment, des preuves de spécification peuvent être développées (avec un contenu algorithmique qui peut différer) et des programmes peuvent alors être extraits de ces preuves. Un tel programme s'appelle *réalisation* d'une spécification. L'extraction permet l'élimination des parties non algorithmiques des preuves, y compris les types dépendants. **Coq** est donc un bon cadre pour établir une bibliothèque de programmes certifiés provenant de preuves de spécification. Ces programmes peuvent être obtenus dans deux langages de programmation différents : OCaml et Haskell[9] (ces extracteurs font partie de la distribution standard de **Coq**). Ainsi, dans le premier cas, nous pourrons employer la bibliothèque BSMLlib pour compiler notre bibliothèque de programmes extraits et dans le second cas, le développement de [209]. Le choix du système **Coq** comme assistant de preuves devient alors évident : la BSMLlib étant implantée pour OCaml, nous pouvons utiliser **Coq** pour vérifier les fonctions classiques de la programmation BSP implantées dans la bibliothèque standard de la BSMLlib.

Pour obtenir notre bibliothèque BSMLlib certifiée, il suffit donc d'utiliser l'extracteur du système **Coq** sur les preuves constructives de nos spécifications. Naturellement, les fonctions données en paramètres

[9] Téléchargeable à l'adresse suivante : `http://www.haskell.org`.

```
Inductive PSRS_sorted [T:Set; v:(Vector (list T))] : Prop :=
 PSRS_sor: (i:Z)(H:0<=i<(bsp_p tt))((sorted T inf (at ? v i H)) ∧
           ((H':i<>(bsp_p tt)-1) (inf_list T inf (at T v i H)
                        (at T v i+1 H'')))) → (PSRS_sorted T v).

Inductive PSRS_exchange [T:Set; v1,v2:(Vector (list T))] : Prop :=
    PSRS_exch : (i:Z) (H_i:0<=i<(bsp_p tt)) (l:(list T))
(list_in_list T l (at T v1 i H_i))→(Ex [j:Z](H_j:0<=j<(bsp_p tt))
     (list_in_list T l (at T v1 j H_j))) → (PSRS_exchange T v1 v2).

Inductive PSRS_permut: (Vector (list T))→(Vector (list T))→Prop :=
 PSRS_exch_is_permut : (v,v':(Vector(list T)))(PSRS_exchange T v v')
                              →(PSRS_permut T v v')
| PSRS_permut_refl :  (v:(Vector (list T)))(PSRS_permut T v v)
| PSRS_permut_sym : (v,v':(Vector (list T)))(PSRS_permut T v v')
                         → (PSRS_permut T v' v)
| PSRS_permut_trans: (v,v',v'':(Vector(list T)))(PSRS_permut T v v')
       → (PSRS_permut T v' v'') → (PSRS_permut T v v'').
```

Figure 6.5 — Spécification inductive pour le tri parallèle par échantillonnage

devant être extraites vers les opérateurs pré-définis du langage BSML, nous devons préalablement préciser ce choix à l'extracteur du système. Par exemple, pour la primitive **mkpar** :

```
Extract Constant mkpar ⇒ "Bsmllib.mkpar"
```

Les développements formels décrits ci-dessus (ainsi que les nombreux autres éléments de la bibliothèque BSMLlib certifiée) sont librement disponibles à la page web de l'auteur[10]. Ils ont été effectués avec la version 7.3 de **Coq**. La traduction dans la version 8.0 est en cours. Ainsi, on peut programmer d'autres algorithmes BSP plus complexes mais dont les éléments standards (notamment les fonctions de communication) sont sûrs, et par la suite, exécuter ces algorithmes sur de véritables machines parallèles avec les opérateurs parallèles de la bibliothèque BSMLlib.

Pour illustrer ce travail, nous donnons en annexe 6.A le code OCaml obtenu par l'extraction des réalisations de nos précédentes spécifications (avec quelques mises en forme). Notons qu'en utilisant l'arithmétique (certifiée) de **Coq**, les programmes extraits des précédentes spécifications sont bien plus lents que ceux écrits «à la main» avec l'arithmétique natif des machines. Mais pour les parties, à proprement parler, parallèles, les performances des fonctions extraites (et donc certifiées) sont équivalentes à celles «écrites main».

6.5.9 Autres applications possibles

[111] propose un langage fonctionnel parallèle : ML étendu avec des squelettes de programmation parallèle sur des vecteurs qui peuvent être imbriqués et qui peuvent être regroupés en quatre classes : calcul, réorganisation, communication et regroupement. La compilation est décrite en une série de transformations (prouvées correctes) vers un langage fonctionnel de type SPMD. Celui-ci utilise des opérations parallèles qui sont en réalité celles que nous venons de décrire (et certifier). Nous pouvons de ce fait envisager l'utilisation de notre bibliothèque pour une implantation certifiée de ce langage.

L'article [97] propose un calcul de distribution dont le but est de décrire des stratégies pour la distribution de tableaux (de multiples dimensions) sur un ensemble de processeurs. Les auteurs donnent une sémantique formelle permettant de prouver différentes équations de distribution, montrent comment associer à ces équations le modèle de coûts BSP autorisant ainsi de choisir une stratégie appropriée et distinguent un certain nombre d'opérateurs pour ces distributions. Ces opérateurs sont encore ceux que nous venons de certifier. En utilisant notre bibliothèque, les auteurs envisagent de valider leurs systèmes d'équations de stratégies dans un environnement certifié.

[10] http://www.univ-paris12.fr/lacl/gava/

6.A Code de l'extraction

Quelques fonctions utilitaires.

```
let noSome = function
  | Some x →x
  | None →assert false

let replicate a = mkpar (fun pid →a)
let parfun f v = apply (replicate f) v
let parfun2 f v1 v2 = apply (parfun f v1) v2

let applyat root f1 f2 vec =
  apply
    (mkpar (fun i v →
      match z_eq_dec i root with
        | Left →f1 v
        | Right →f2 v)) vec
```

Code de `get_list`.

```
let get_list datas lpids = parfun2 (fun x x0 →map x x0)
   (put (parfun (fun l dst →match (in_dec z_eq_dec dst l) with
                            | Left →Some ()
                            | Right →None) lpids)) datas)) lpids
```

Code de la réduction directe.

```
let totex_tmp vec = put (parfun (fun data dst →Some data) vec)
let totex vec = parfun (fun f src →noSome (f src)) (totex_tmp vec)

let procs h = from_to (last ())
let total_exchange vec = parfun2 map (totex vec) (replicate (procs ()))

let tl_hd_vect v = parfun (fun l →Pair ((hd l), (tl l))) v

let fold_left_direct op0 vec =
  parfun (fun hd_tl →fold_left op0 (snd hd_tl) (fst hd_tl))
         (tl_hd_vect (total_exchange vec))
```

Code du rassemblement.

```
let procs_lists_gather root = match within_bound root with
  | Left →mkpar (fun pid →match z_eq_dec root pid with
                       | Left →from_to (zminus (bsp_p ()) (POS XH))
                       | Right →Nil)
  | Right →assert false


let gather root vv =
  put
    (apply
      (mkpar (fun pid v dest →
        match z_eq_dec dest root with
          | Left →Some v
          | Right →None)) vv)
```

```
let gather_list vec root = match within_bound root with
  | Left →parfun2 (fun x x0 →map x x0)
              (parfun (fun x x0 →(fun x1 →noSome x1) (x x0))
              (gather root vec)) (procs_lists_gather root)
  | Right →assert false
```

Code de la diffusion en une ou deux phases.

```
let rpl_bcast_direct_tmp root vec =
  proj (applyat root (fun v →Some v) (fun x →None) vec) root

let rpl_bcast_direct root vec =
  noSome (rpl_bcast_direct_tmp root vec)

let scatter_tmp partition root v =
  apply
    (put
      (apply
        (mkpar (fun pid →
          match z_eq_dec pid root with
            | Left →partition
            | Right →(fun a i →None))) v)) (replicate root)


let scatter partition root v = match within_bound root with
   | Left →parfun noSome (scatter_tmp partition root v)
   | Right →assert false

let bcast_twophases partition paste root vv =
 match within_bound root with
  | Left →parfun paste (totex (scatter partition root vv))
  | Right →assert false
```

Code de la composition parallèle.

```
let mask c x y =
  apply
    (apply
      (mkpar (fun j x0 y0 →match c j with
                                  | True →x0
                                  | False →y0)) x) y
```

Code du tri parallèle.

```
let regular_sample_sort v =
  let local_sort = apply (replicate (Sort.list (<))) in
  let list_max = List.fold_left max 0 in
  let locally_sorted = local_sort v in
  let primary_samples =
    apply (replicate (sample (nprocs+1) )) locally_sorted in
  let sorted_primary_samples =
   local_sort (apply (replicate List.flatten) (total_exchange primary_samples)) in
  let secondary_samples =
    apply (replicate (sample (nprocs+1) )) sorted_primary_samples in
  let message_lists =
```

```
    apply2
      (replicate (fun sec_s xs →List.map (fun x →(dest sec_s x, x)) xs))
      sec_sampls_array
      v in
  let received_values =
    apply (replicate (List.map snd)) (put_list message_lists)
  in local_sort received_values
```

Deuxième partie

Extensions et bibliothèque de structures de données

7 Compositions parallèles

Une partie de ce chapitre a fait l'objet de l'article [N1].

Sommaire

NOUS avons vu que BSML est un langage parallèle permettant la programmation fonctionnelle BSP. Basé sur BSP, il permet une estimation du temps d'exécution du programme et est confluent, donc sans inter-blocage. Malheureusement, les primitives de BSML sont plates, dans le sens que les processus explicites du langage (les vecteurs parallèles), partitionnent de manière directe les processeurs d'une unité BSP. Il est souvant trop difficile d'exprimer des algorithmes «diviser-pour-régner» (*«divide-and-conquer»* dans la littérature anglo-saxonne) sans avoir à traduire ces algorithmes en des algorithmes en mode direct. Ce chapitre présente la sémantique et l'implantation d'une nouvelle primitive pour BSML qui facilite l'écrire de ces algorithmes diviser-pour-régner parallèles. Cette primitive est appelée **superposition** car elle peut être vue comme la composition parallèle de deux processus BSP utilisant chacun l'ensemble des processeurs de la machine. Nous donnerons aussi un exemple d'application et le modèle de coût BSP de cette nouvelle primitive.

7.1 Introduction

La sous-synchronisation, c'est-à-dire la synchronisation d'une partie seulement des processeurs de la machine, est une fonctionnalité d'une grande variété de modèles de programmation parallèle (indépendants ou dérivés du modèle BSP [89, 41]). La présence et l'utilisation de la sous-synchronisation sont justifiées par la nécessité de pouvoir décomposer récursivement un problème en sous-problèmes indépendants les uns des autres. Cette technique algorithmique est appelée «diviser-pour-régner», et dans le cas de la programmation parallèle, simplifie l'écriture de certains programmes.

Pour évaluer deux programmes parallèles sur une même machine on peut la partitionner en deux et évaluer indépendamment chacun des programmes sur chacune des partitions. Toutefois, en procédant ainsi, le modèle BSP est perdu puisqu'alors la synchronisation globale de chaque sous-machine ne coûtera plus l. Pour conserver le modèle BSP, ce qui est souhaitable [133], il faut donc que les barrières de synchronisation concernent toute la machine.

Le modèle BSP n'autorise pas la sous-synchronisation : les barrières sont collectives. C'est souvent considéré comme un obstacle pour la programmation parallèle d'algorihmes diviser-pour-régner en BSP (et dans notre cas, en BSML). Dans l'article [243], les auteurs ont noté que pour de grandes applications, la perte d'efficacité et d'expressivité due à la contrainte des barrières globales, est compensée par les avantages des communications BSP (notamment l'efficacité accrue des transferts de données sur certaines architectures) qui rendent la programmation parallèle plus simple.

Pourtant, la technique dite diviser-pour-régner est une méthode naturelle pour la conception et l'implantation de nombreux algorithmes. La transformation manuelle d'un programme utilisant des techniques diviser-pour-régner en un programme sans ces techniques est un travail fastidieux où il est facile de se tromper. Les auteurs de la BSPlib [147] font remarquer que le désavantage principal du partionnement par groupe de processeurs (c'est-à-dire la sous-synchronisation) est une perte de la prévision des performances du modèle de coût ([133] donne des arguments, sur des cas pratiques, en défaveur de la sous-synchronisation).

Pour exprimer les algorithmes diviser-pour-régner en BSML, [188] a tout d'abord proposé une nouvelle primitive appelée «composition parallèle». Mais celle-ci était limitée à la composition de deux programmes BSML dont les évaluations nécessitaient le même nombre de super-étapes. C'était évidemment restrictif et le respect de cette contrainte échoyait au programmeur.

Cette restriction a été levée avec la «juxtaposition parallèle» de [190]. Le principal problème avec cette approche est que la nature purement fonctionnelle de BSML est perdue. En effet cette primitive effectue un effet de bord sur le nombre de processeurs rendant impossible la preuve assistée par ordinateur des programmes BSML utilisant la juxtaposition, comme il est fait au chapitre 6 avec des programmes BSML «plats», ou alors il faut passer les programmes par une phase préalable de transformation [183] (mais avec une perte des coûts BSP initiaux).

Dans [258], l'auteur avance l'hypothèse que le paradigme diviser-pour-régner peut s'insérer naturellement dans le modèle BSP sans avoir besoin de la sous-synchronisation. Il propose une méhode pour la programmation diviser-pour-régner qui est en adéquation avec le modèle des barrières de synchronisations globales du modèle BSP. Cette méthode est basée sur un entrelacement de processus légers, appelés *super-threads*[1], chacun d'entre eux étant un processus de calcul BSP.

L'article [191] présente une nouvelle primitive BSML, appelée *superposition parallèle* permettant d'écrire de manière fonctionnelle ce type d'algorithmes. Dans ce papier, la superposition est simplement décrite en terme d'une sémantique de haut niveau. Dans ce cas, la superposition est équivalente à la création d'une paire. Dans ce chapitre, nous allons détailler une sémantique à «petits pas» qui donne plus précisément le fonctionnement de cette nouvelle primitive. Cela nous aidera pour concevoir une nouvelle implantation de la BSMLlib et pour comparer les coûts BSP des programmes avec ou sans super-threads.

Premièrement, nous décrirons informellement la nouvelle primitive BSML et les conséquences de son fonctionnement sur les traits non-fonctionnels de BSML. Ensuite, nous donnerons deux sémantiques à petits pas, afin de comparer les coûts BSP de cette primitive. Nous donnerons ensuite quelques détails sur la nouvelle implantation de BSML. Pour finir, nous donnerons deux exemples d'application de cette primitive.

7.2 La superposition parallèle

Le principe des super-threads est le suivant : au lieu de diviser la machine en sous-machines effectuant des sous-synchronisations, on peut diviser la puissance de la machine parallèle en des processus BSP. Les res-

[1] Nous préférons le terme de *super-thread* et non son anglicisme «super processus léger» qui est un peu pompeux.

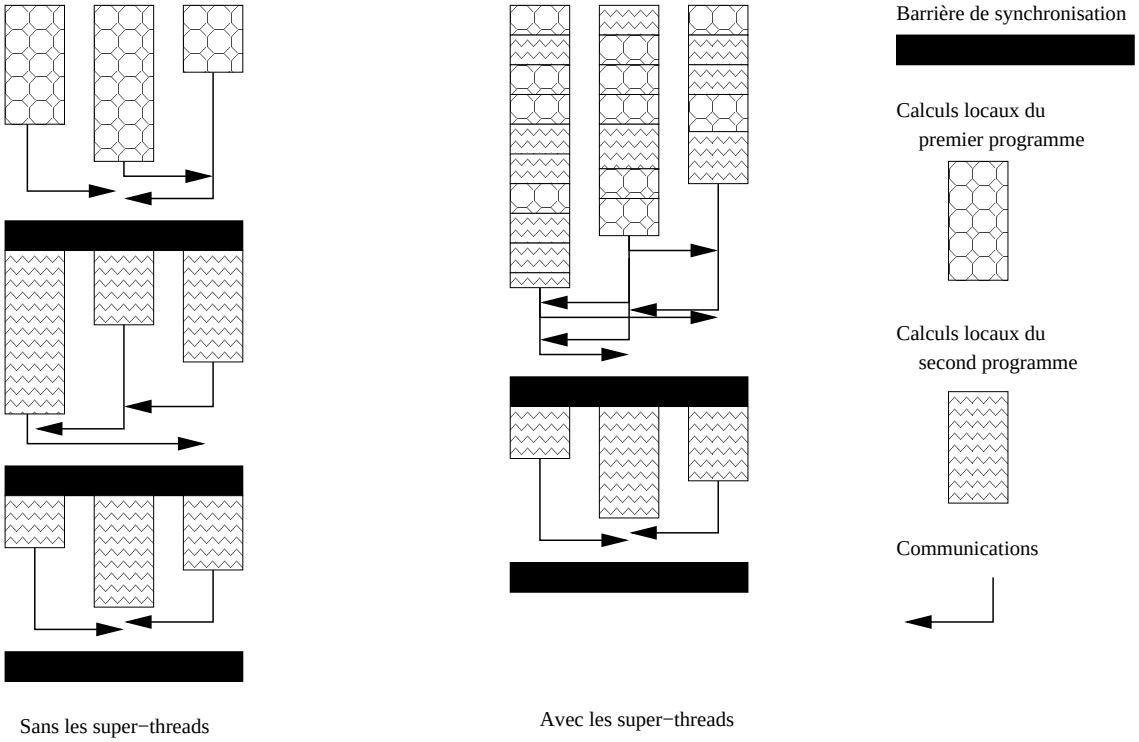


Figure 7.1 — Deux programmes avec super-threads

sources de la machine BSP sont partagées par les processus. Ces processus sont appelés *super-threads* car chacun de ces processus légers utilise l'ensemble des processeurs. Les communications entre les processus légers sont fusionnées et les synchronisations globales du modèle BSP sont alors partagées par les super-threads : on n'a qu'une seule synchronisation pour plusieurs super-threads effectuant des communications. Les méthodes optimisant les transferts de données BSP peuvent alors être conservées. Cette nouvelle primitive appelée *superposition* permet de cette manière l'évaluation de deux expressions BSML. La figure 7.1 illustre ce propos avec deux super-threads qui sont le calcul de deux expressions BSML indépendantes. Sans la superposition et les super-threads, les deux expressions sont évaluées séquentiellement. Dans le cas d'une exécution avec super-threads, les communications et les synchronisations sont fusionnées, ce qui diminue le temps d'exécution globale (voir la section 7.3 pour une analyse plus formelle). Notons que le calcul local, lui, n'est pas affecté. Seul l'ordre d'évaluation change.

7.2.1 Présentation informelle

De point de vue du programmeur, la sémantique parallèle de la superposition est la même que la paire. Dans la BSMLlib, la primitive a le type suivant :

super: (unit→α)→(unit→β)→α $*$ β

Bien évidemment l'évaluation de $\mathbf{super}\ E_1\ E_2$ est différente de celle de $((E_1\ ()), (E_2\ ()))$. Elle se déroule ainsi (voir section 7.3 pour une sémantique formelle) :

- La première phase de calcul asynchrone de E_1 et celle de E_2 sont exécutées ;
- Ensuite, la première phase de communication de E_1 est mise en commun avec celle de E_2. Les messages sont la «concaténation» des messages de la première phase de communication de E_1 et de E_2 ;
- Enfin, une seule barrière de synchronisation termine la première super-étape de E_1 *et* celle de E_2. Les super-étapes suivantes des deux *super-threads* peuvent alors être exécutées.

Si l'évaluation de E_1, par exemple, nécessite moins de super-étapes que celle de E_2 alors l'évaluation de E_2 se termine comme s'il n'y avait pas de superposition de deux *super-threads* (et vice-versa).

7.2.2 Modèle de coût informel

Pour déterminer le coût de l'évaluation de **super** E_1 E_2, il n'est pas suffisant, ni de considérer le coût d'évaluation de E_1 et E_2 de la forme $W + H \times g + l$, ni de considérer les coûts de chaque super-étape de la même forme. Il faut considérer le coût de chaque super-étape dans les détails, donné sous la forme de trois vecteurs :

- Le coût du calcul local sur chaque processeur : $\langle\ w_0\ , \ldots,\ w_{p-1}\ \rangle$;
- La taille des messages envoyés par chaque processeur : $\langle\ h_0^+\ , \ldots,\ h_{p-1}^+\ \rangle$;
- La taille des messages reçus par chaque processeur :$\langle\ h_0^-\ , \ldots,\ h_{p-1}^-\ \rangle$.

On note $(\bar{w}, \bar{h}^+, \bar{h}^-)$ le coût d'une super-étape. Le coût de l'évaluation d'une expression est une liste de ces triplets de vecteurs.
Si les coûts d'évaluation de E_1 (avec $k1$ super-threads) et E_2 (avec $k2$ super-threads) sont respectivement :

$$\left\{\begin{array}{l}(\bar{w}^0, \bar{h}^{0+}, \bar{h}^{0-}), \ldots, (\bar{w}^{k_1}, \bar{h}^{k_1+}, \bar{h}^{k_1-}) \\ (\bar{w}'^0, \bar{h}'^{0+}, \bar{h}'^{0-}), \ldots, (\bar{w}'^{k_2}, \bar{h}'^{k_2+}, \bar{h}'^{k_2-})\end{array}\right.$$

alors le coût d'évaluation de **super** E_1 E_2 est :

$$\left(\begin{array}{c}\langle\ w_0^0 + w'^0_0\ , \ldots,\ w_{p-1}^0 + w'^0_{p-1}\ \rangle \\ \langle\ h_0^{0+} + h'^{0+}_0\ , \ldots,\ h_{p-1}^{0+} + h'^{0+}_{p-1}\ \rangle \\ \langle\ h_0^{0-} + h'^{0-}_0\ , \ldots,\ h_{p-1}^{0-} + h'^{0-}_{p-1}\ \rangle\end{array}\right), \ldots, \left(\begin{array}{c}\langle\ w_0^k + w'^k_0\ , \ldots,\ w_{p-1}^k + w'^k_{p-1}\ \rangle \\ \langle\ h_0^{k+} + h'^{k+}_0\ , \ldots,\ h_{p-1}^{k+} + h'^{k+}_{p-1}\ \rangle \\ \langle\ h_0^{k-} + h'^{k-}_0\ , \ldots,\ h_{p-1}^{k-} + h'^{k-}_{p-1}\ \rangle\end{array}\right)$$

où $k = \max\{k_1, k_2\}$ et où w_i^n, h_i^{n+} et h_i^{n-} (resp. $w_i'^n$, $h_i'^{n+}$ et $h_i'^{n-}$) sont considérés égaux à 0 si $n > k_1$ (resp. $n > k_2$). Le coût BSP habituel est alors :

$$(\sum_{n=0}^{k} \max\{W^n\}) \ + \ (\sum_{n=0}^{k} h^n) \times g \ + \ k \times l \quad \text{où} \quad \left\{\begin{array}{lcl} W^n & = & \max\limits_{i=0}^{p-1}\{w_i^n + w_i'^n\} \\ h^n & = & \max\limits_{i=0}^{p-1}\{(h_i^{n+} + h_i'^{n+})\ ,\ (h_i^{n-} + h_i'^{n-})\}\end{array}\right.$$

Là encore, le modèle de coût est compositionnel. La superposition parallèle de E_1 et E_2 est moins coûteuse que l'évaluation de E_1 suivie de l'évaluation de E_2. Par exemple le h résultant de la superposition peut être égal au plus grand des deux et donc inférieur à la somme. C'est bien sûr toujours le cas pour le nombre de barrières de synchronisation.

7.2.3 Choix d'une stratégie pour l'évaluation des super-threads

L'ordre d'évaluation (la stratégie) d'expressions fonctionnelles n'a pas d'importance : la sémantique est confluente. Mais, dans la familles des langages ML, il est facile d'ajouter des fonctionnalités impératives (voir [C4] ou au chapitre 9 pour BSML). Dans ce cas, l'ordre d'évaluation est nécessaire pour garder une sémantique déterministe (ou confluente dans le cas de BSML). La présence d'expressions impératives dans les super-threads et de données partagées forcent à l'utilisation d'une stratégie pour les super-threads. Prenons par exemple, l'expression suivante :

```
let a=ref 0 in
let _ = super (fun () →a:=1;replicate 1)
              (fun () →a:=2;replicate 1) in
 if (!a)=1 then (replicate 1) else (scan_direct (+) 0 (replicate 1))
```

Dans cet exemple, chaque processeur crée une référence globale, appelée a et qui contient l'entier 0. Ensuite, deux super-threads sont créés à l'aide de la primitive de superposition. Chacun de ces super-threads affecte une valeur différente à la variable a. Sans une stratégie bien prédifinie pour les super-threads (en tant que processus légers), il est impossible de prévoir lequel des deux fera en dernier l'affectation. Le résultat de l'expression (!a) est alors non déterministe. Pire même, nous pouvons avoir une erreur d'exécution car cet ordre sera généralement différent en chaque processeur, ce qui donnera des valeurs différentes à a. Des processeurs exécuterons repliacte 1 et d'autres (scan_direct (+) 0 (replicate 1)) : la barrière globale du modèle BSP ne l'est plus. Notons que nous pouvons avoir les mêmes problèmes avec les entrées/sorties (cf. chapitre 9) et les traits impératifs [C4]).

Le même problème de non déterminisme peut aussi survenir avec l'utilisation uniquement de références locales. Prenons par exemple l'expression suivante :

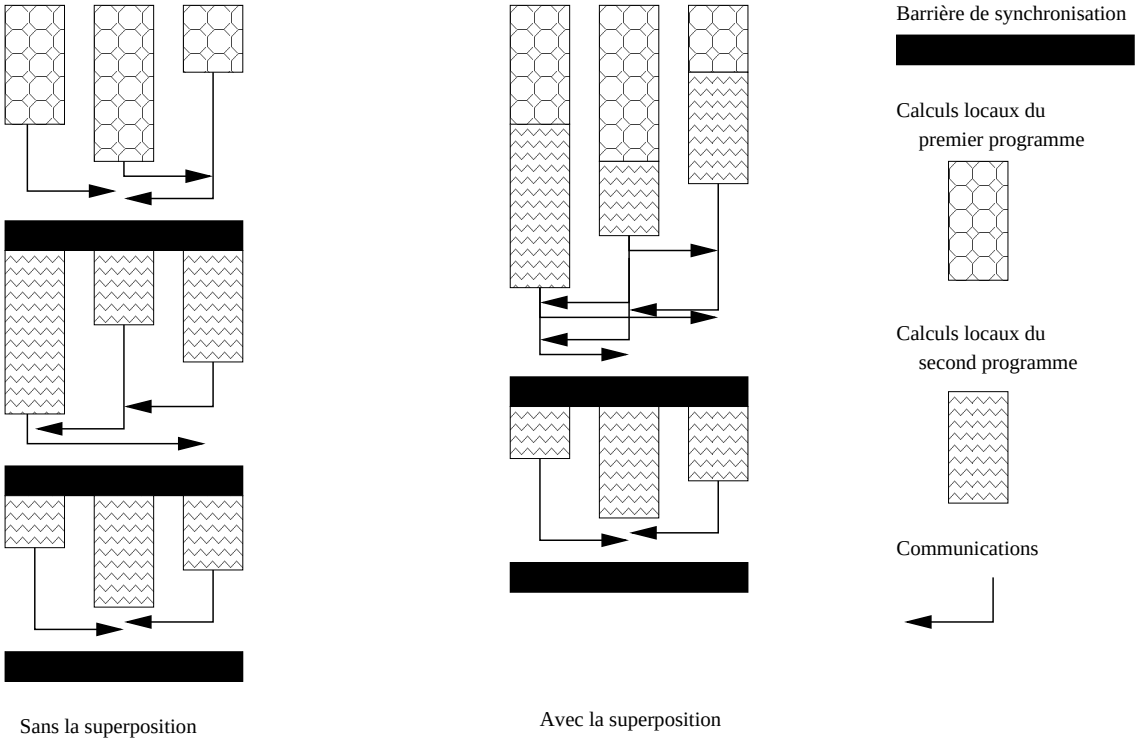


Figure 7.2 — Superposition de deux programmes

```
let a=mkpar (fun pid →ref pid) in
let _ = super (fun () →parfun (fun r →r:=!r+1) a)
              (fun () →parfun (fun r →r:=!r∗2) a) in
 parfun (fun r →!r) a
```

avec a qui est un vecteur parallèle de références locales et où les deux super-threads peuvent affecter des valeurs différentes à ces références.

Le problème vient du fait que plusieurs processus légers sont actifs lors des phases de calculs asynchrones : l'ordre d'évaluation des processus légers est non-déterministe et est géré par l'ordonnanceur (*scheduler* en anglais) du système d'exploitation. Ces processus légers peuvent donc modifier l'état de valeurs partagées.

La présence d'un seul processus léger actif permettrait d'éviter ces problèmes. Cela impose une stratégie pour déterminer, quel super-thread est actif ou non. Sans cette stratégie, l'entrelacement des processus légers peut donner des résultats non déterministes. De plus, l'utilisation d'un seul super-thread actif permet d'obtenir de meilleurs performances : le coût (engendré par l'ordonnanceur) de changer «régulièrement» de super-thread actif est supprimé.

OCaml ayant une statégie stricte d'appel par valeur, nous avons opté pour la même stratégie dans le cas des super-threads : un seul est actif et celui-ci est évalué jusqu'à la fin de ses calculs ou quand il entre dans la phase de communication, c'est-à-dire qu'il a terminé la phase de calculs asynchrones de la super-étape courante. Quand les communications ont été effectuées, le premier des super-threads qui a terminé sa phase de calcul locale est réactivé (il devient donc le nouveau super-thread actif). La figure 7.2 illustre ce propos.

7.3 Sémantiques pour la superposition

7.3.1 Sémantiques sans super-threads

Comme nous l'avons précisé, du point de vue du programmeur, la superposition n'est rien d'autre que la création d'une paire. Nous pouvons donc étendre la sémantique naturelle du chapitre 3 avec la règle suivante :

$$\frac{s, e_1 \rhd_{\bowtie} f_1 \quad s, e_2 \rhd_{\bowtie} f_2 \quad s, ((f_1\ ()), (f_2\ ())) \rhd_{\bowtie} (v_1, v_2)}{s, \mathbf{super}\ e_1\ e_2 \rhd_{\bowtie} (v_1, v_2)} \tag{7.1}$$

et nous avons le théorème suivant :

Théorème 8 (*Déterminisme*)

Soit une expression «programmeur» e^p telle que $e = \mathcal{T}_\bullet(e^p)$. Si $\bullet, e \rhd_g v_1$ et $\bullet, e \rhd_g v_2$ alors $v_1 = v_2$.

Preuve. Par induction sur les arbres d'évaluations. Preuve similaire à celle du chapitre 3. ■

Nous étendons aussi la sémantique à «petits pas» du chapitre 3 avec les règles suivantes :

$$\begin{aligned} (\mathbf{super}\ f_1\ f_2)/c_g/\langle \mathbf{c} \rangle &\xrightarrow[\bowtie]{\delta} (f_1(), f_2())/c_g \oplus 1/\langle \mathbf{c} \rangle \\ (\mathbf{super}\ e_1\ e_2)[s] &\xrightarrow{\varepsilon} (\mathbf{super}\ e_1[s]\ e_2[s]), 0 \end{aligned}$$

et nous avons les résultats suivants :

Lemme 35 (*Confluence forte*)

Soit une expression e.

Si	$e/c_g/\langle c_0, \cdots, c_{p-1} \rangle \rightharpoonup e^1/c_g^1/\langle c_0^1, \cdots, c_{p-1}^1 \rangle$
et	$e/c_g/\langle c_0, \cdots, c_{p-1} \rangle \rightharpoonup e^2/c_g^2/\langle c_0^2, \cdots, c_{p-1}^2 \rangle$
alors il existe	*une expression e_3 et des coûts c_g^3 et $\langle c_0^3, \cdots, c_{p-1}^3 \rangle$*
tels que	$e^1/c_g^1/\langle c_0^1, \cdots, c_{p-1}^1 \rangle \rightharpoonup e^3/c_g^3/\langle c_0^3, \cdots, c_{p-1}^3 \rangle$
et	$e^2/c_g^2/\langle c_0^2, \cdots, c_{p-1}^2 \rangle \rightharpoonup e^3/c_g^3/\langle c_0^3, \cdots, c_{p-1}^3 \rangle$.

Preuve. Preuve similaire à celle chapitre 3 ■

Théorème 9 (*Confluence*)

Soit une expression «programmeur» e^p.

Si	$e = \mathcal{T}_\bullet(e^p)$
alors si	$e[\bullet]/0/\langle 0, \cdots, 0 \rangle \xrightarrow{*} v/c_g/\langle c_0, \cdots, c_{p-1} \rangle$
et	$e[\bullet]/0/\langle 0, \cdots, 0 \rangle \xrightarrow{*} v'/c'_g/\langle c'_0, \cdots, c'_{p-1} \rangle$
alors	$v = v'$, $c_g = c'_g$ *et* $\forall i \in \{0, \ldots, p-1\}\ c_i = c'_i$.

Preuve. La relation $\rightharpoonup$ est fortement confluente donc, d'après le lemme 1, elle est confluente. ■

Théorème 10 (*Équivalence*)

Soit une expression programmeur e^p telle que $e = \mathcal{T}_\bullet(e^p)$. Alors :

1. *si $e[\bullet]/0/\langle 0, \ldots, 0 \rangle \xrightarrow{*} v/c_g/\langle c_0, \ldots, c_{p-1} \rangle$ alors $\bullet, e \rhd_g v$*
2. *si $\bullet, e \rhd_g v$ alors $e[\bullet]/0/\langle 0, \ldots, 0 \rangle \xrightarrow{*} v/c_g/\langle c_0, \ldots, c_{p-1} \rangle$*

Preuve. Preuve similaire à celle du chapitre 3 ■

La sémantique naturelle ne donne toujours pas les étapes de calculs. Elle ne donne pas non plus la création des super-threads. Ainsi, toutes les opérations parallèles semblent être synchrones.

La sémantique à «petits pas» fournit une «vision» plus asynchrone du langage mais ne dit toujours pas comment sont créés et gérés les super-threads. Cette sémantique n'aide donc pas à implanter la superposition. Il nous faut donc redéfinir cette sémantique afin de rendre apparente la création des super-threads.

De plus, le coût BSP de la superposition est toujours identique à celui de la création d'une paire, ce que nous ne voulons pas. En effet, ils nous faudrait avoir une règle du type :

$$(\mathbf{super}\ f_1\ f_2)/c_g/\langle \mathbf{c} \rangle \xrightarrow[\bowtie]{\delta} (f_1(), f_2())/c'_g/\langle \mathbf{c}' \rangle$$

Mais il paraît difficile de définir le c'_g et les c' tout en étant conforme aux coûts de la section 7.2.2. Un $\max$ sur le nombre de l (et avec une fusion des communications) n'est pas suffisant car il faut aussi prendre le $\max$ des calculs locaux des super-threads : le $\max$ d'une liste de coûts locaux avec les $\max$ des coûts asynchrones globaux. Ceci n'est pas naturel, difficilement vérifiable et comparable avec les coûts définis dans le chapitre 3.

Il nous faut donc redéfinir une sémantique à «petits pas», afin d'expliciter la gestion des super-threads.

7.3.2 Syntaxe des super-threads

La primitive de superposition ($\mathbf{super}$) va créer des super-threads sur la machine BSP. Chacun de ces super-threads contient une expression à évaluer. Nous organisons donc naturellement cet ensemble de super-threads comme un arbre binaire.

Définition 24 (*Organisation des super-threads*).
La syntaxe des super-threads est donnée par la grammaire suivante :

$$E \quad ::= \quad \phi e \flat_t \quad | \quad (E \bowtie E)$$

où les identifiants t sont définis par l'expression régulière suivante :

$$t ::= 1(.\{1,2\})^*$$

où e est une expression à évaluer par un super-thread de numéro (identifiant) t. Par exemple, 1, 1.1, 1.2, $1.1.2$ et $1.2.1$ sont des identifiants valides. Les super-threads sont organisés comme un arbre. Si l'on réduit un $\mathbf{super}$ contenu dans une expression d'un super-thread, alors ce super-thread donnera «naissance» à deux nouveaux super-threads. Nous parlerons alors de super-thread enfants.

Pour exprimer les communications BSP en BSML, nous utilisons les primitives $\mathbf{proj}$ et $\mathbf{put}$. Mais comme cela a été expliqué précédemment, en présence de super-threads, les communications sont «gelées» (*freeze* dans la littérature anglo-saxonne), c'est-à-dire retardées jusqu'à ce que tous les super-threads aient fini leurs phases de calculs (de la super-étape courante $\mathbf{s}$) afin de fusionner les communications et la barrière de synchronisation. Pour se faire, les valeurs (messages à envoyer) sont stockées dans un environnement de communications. Quand tous les *super-threads* ont terminé ce stockage, les communications sont effectuées, et la super-étape suivante peut commencer (les messages ont tous été reçus et sont accessibles par les processeurs).

Notre environnement de communications est noté :

$$\{\langle\, [\, v\, , \ldots ,\, v'\,]\, , \ldots ,\, [\, v''\, , \ldots ,\, v'''\,]\, \rangle_s^t, \ldots\}$$

C'est un ensemble de vecteurs parallèles de tableaux. Les vecteurs sont annotés avec l'identifiant t (resp. numéro $\mathbf{s}$) du super-thread (resp. de la super-étape) qui a stocké ces valeurs. Notons que ce numéro est soit celui de la super-étape courante, soit celui de la dernière super-étape si des *super-threads* ont recommencé leurs calculs (et donc lu les valeurs dans l'environnement) et d'autres non (ils n'ont pas encore eu le temps).

7.3.3 Nouvelle forme de sémantique

La nouvelle sémantique à «petits pas» a la forme suivante :

$$\mathcal{E}, \mathbf{s}, E/c_g/\langle c_0, \ldots, c_{p-1}\rangle \mapsto \mathcal{E}', \mathbf{s}', E'/c'_g/\langle c'_0, \ldots, c'_{p-1}\rangle$$

où

- $\mathcal{E}$ est l'environnement de communication courant,
- $\mathbf{s}$ est le numéro de la super-étape actuelle,
- E est l'arbre des super-threads,
- c_g représente les coûts globaux,
- $\langle c_0, \ldots, c_{p-1}\rangle$ sont les coûts locaux.

$$
\begin{array}{rcl}
(\overline{(\lambda.e)[s]}\ v) & \overset{\varepsilon}{\mapsto} & e[v \circ s], 1 \\
\overline{n+1}[v \circ s] & \overset{\varepsilon}{\mapsto} & \overline{n}[s], 0 \\
\overline{1}[v \circ s] & \overset{\varepsilon}{\mapsto} & v[\bullet], 1 \\
(\mu.e)[s] & \overset{\varepsilon}{\mapsto} & e[\mu.e \circ s], 1 \\
(\lambda.e)[s] & \overset{\varepsilon}{\mapsto} & \overline{(\lambda.e)[s]}, 1 \\
v[s] & \overset{\varepsilon}{\mapsto} & v, 0 \quad \text{si } v \neq \langle \cdots \rangle \text{ et } v \neq (v_0, v_1) \\
(e_1\ e_2)[s] & \overset{\varepsilon}{\mapsto} & (e_1[s]\ e_2[s]), 0 \\
(e_1, e_2)[s] & \overset{\varepsilon}{\mapsto} & (e_1[s], e_2[s]), 0 \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)[s] & \overset{\varepsilon}{\mapsto} & \textbf{if } e_1[s] \textbf{ then } e_2[s] \textbf{ else } e_3[s], 0 \\
(\textbf{proj}\ e)[s] & \overset{\varepsilon}{\mapsto} & (\textbf{proj}\ e[s]), 0 \\
(\textbf{put}\ e)[s] & \overset{\varepsilon}{\mapsto} & (\textbf{put}\ e[s]), 0 \\
(\textbf{apply}\ e_1\ e_2)[s] & \overset{\varepsilon}{\mapsto} & (\textbf{apply}\ e_1[s]\ e_2[s]), 0 \\
(\textbf{mkpar}\ e)[s] & \overset{\varepsilon}{\mapsto} & (\textbf{mkpar}\ e[s]), 0 \\
(\textbf{super}\ e_1\ e_2)[s] & \overset{\varepsilon}{\mapsto} & (\textbf{super}\ e_1[s]\ e_2[s]), 0 \\
\langle e_0, \ldots, e_{p-1} \rangle[s] & \overset{\varepsilon}{\mapsto} & \langle e_0[s], \ldots, e_{p-1}[s] \rangle, 0 \\
[e_0, \ldots, e_{p-1}][s] & \overset{\varepsilon}{\mapsto} & [e_0[s], \ldots, e_{p-1}[s]], 0
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{access}\ ([v_0, \ldots, v_i, \ldots, v_{p-1}], i) & \overset{\delta}{\mapsto} & v_i, 1 \\
\textbf{init}\ (n, f) & \overset{\delta}{\mapsto} & [(f\ 0), \ldots, (f\ (n-1))], 1 \\
\textbf{fst}\ (v_1, v_2) & \overset{\delta}{\mapsto} & v_1, 1 \\
\textbf{snd}\ (v_1, v_2) & \overset{\delta}{\mapsto} & v_2, 1 \\
+\ (n_1, n_2) & \overset{\delta}{\mapsto} & n_1 + n_2, 1 \\
\textbf{isnc nc} & \overset{\delta}{\mapsto} & \textbf{true}, 1 \\
\textbf{isnc}\ v & \overset{\delta}{\mapsto} & \textbf{false}, 1 \text{ si } v \neq \textbf{nc} \\
\textbf{LocId}\ v & \underset{i}{\overset{\delta}{\mapsto}} & v, 1 \\
\multicolumn{3}{l}{\textbf{if true then } e_2 \textbf{ else } e_3 \overset{\delta}{\mapsto} e_2, 1} \\
\multicolumn{3}{l}{\textbf{if false then } e_2 \textbf{ else } e_3 \overset{\delta}{\mapsto} e_3, 1}
\end{array}
$$

Figure 7.3 — Réductions fonctionnelles et opérations génériques (rappel)

Nous notons $\overset{*}{\mapsto}$ pour la fermeture transtive et réflexive de $\mapsto$, c'est-à-dire que nous notons :

$$\mathcal{E}, \mathbf{s}, E/c_g/\langle c_0, \ldots, c_{p-1} \rangle \overset{*}{\mapsto} \mathcal{E}', \mathbf{s}', E'/c'_g/\langle c'_0, \ldots, c'_{p-1} \rangle$$

pour :

$$\mathcal{E}, \mathbf{s}, E/c_g/\langle c_0, \ldots, c_{p-1} \rangle \mapsto \mathcal{E}^0, \mathbf{s}^0, E^0/c^0_g/\langle c^0_0, \ldots, c^0_{p-1} \rangle \mapsto \cdots \mapsto \mathcal{E}', \mathbf{s}', E'/c'_g/\langle c'_0, \ldots, c'_{p-1} \rangle$$

L'évaluation complète d'une expression e^p sera notée :

$$\{\}, 0, \phi\!\!\flat e \flat\!\!\phi_1/0/\langle 0, \ldots, 0 \rangle \overset{*}{\mapsto} \{\}, \mathbf{m}, \phi\!\!\flat v \flat\!\!\phi_1/c_g/\langle c_0, \ldots, c_{p-1} \rangle$$

si $e = \mathcal{T}_\bullet(e^p)$. Nous pouvons lire cette réduction ainsi : «dans un environnement de communications vide, l'expression e mise dans le super-thread initial à la première super-étape, sera évaluée en la valeur v, dans un environnement de communication vide et cela en $\mathbf{m}$ super-étapes et pour un coût final BSP de $c_g \oplus \max_{i=0}^{p-1}(c_i)$».

Définition 25 (*Relations de la nouvelle sémantique à «petits pas»*).
Pour définir la relation $\mapsto$ nous avons trois types de réductions :

1. $\underset{}{\overset{i}{\mapsto}}$ *est la réduction locale d'une expression ;*
2. $\underset{t}{\overset{\bowtie}{\mapsto}}$ *est la réduction globale d'une expression dans un super-thread t ;*
3. $\underset{t}{\mapsto}$ *est la réduction d'un super-thread t*

avec :

$$\underset{t}{\mapsto} = \underset{t}{\overset{\bowtie}{\mapsto}} \cup \overset{i}{\mapsto} \qquad \underset{t}{\overset{\bowtie}{\mapsto}} = \underset{\bowtie_t}{\overset{\delta}{\mapsto}} \cup \underset{\bowtie_t}{\overset{\varepsilon}{\mapsto}} \qquad \overset{i}{\mapsto} = \underset{i}{\overset{\delta}{\mapsto}} \cup \underset{i}{\overset{\varepsilon}{\mapsto}}$$

Nous allons maintenant détailler ces trois types de réduction pour définir la sémantique.

7.3.4 Règles génériques et parallèles

Nous redéfinissons les règles de β-réductions, propagation de la substitution, δ-règles génériques de la même manière que dans la sémantique à «petits pas» normale (sans les super-threads). La figure 7.3 les rappelle.

Les règles pour les primitives parallèles sont données à la figure 7.4. Celles-ci sont similaires à celles de la sémantique à «petits pas» sans les super-threads, mais avec les environnements de communications

$$\mathcal{E}, \mathbf{s}, \mathbf{mkpar}\ f/c_g/\langle \mathbf{c}\rangle \overset{\delta}{\underset{\bowtie_t}{\mapsto}} \mathcal{E}, \mathbf{s}, \langle\ (f\ v_0)\ , \ldots,\ (f\ v_{p-1})\ \rangle[\bullet]/c_g \oplus 1/\langle \mathbf{c}\rangle \quad \text{si } \mathcal{V}_\bullet(f) = \mathbf{true} \tag{7.2}$$

$$\mathcal{E}, \mathbf{s}, \mathbf{apply}\ \langle f_0, \ldots, f_{p-1}\rangle\ \langle v_0, \ldots, v_{p-1}\rangle/c_g/\langle \mathbf{c}\rangle \overset{\delta}{\underset{\bowtie_t}{\mapsto}} \mathcal{E}, \mathbf{s}, \langle (f_0\ v_0), \ldots, (f_{p-1}\ v_{p-1})\rangle[\bullet]/c_g \oplus 1/\langle \mathbf{c}\rangle \tag{7.3}$$

$$\mathcal{E}, \mathbf{s}, \mathbf{proj}\ \langle v_0, \ldots, v_{p-1}\rangle/c_g/\langle \mathbf{c}\rangle \overset{\delta}{\underset{\bowtie_t}{\mapsto}} \mathcal{E}, \mathbf{s}, (\mathbf{delpar}\ (\mathbf{put}\ \langle \lambda.v_0, \ldots, \lambda.v_{p-1}\rangle))[\bullet]/c_g \oplus 1/\langle \mathbf{c}\rangle \tag{7.4}$$

$$\mathcal{E}, \mathbf{s}, \mathbf{put}\ \langle f_0, \ldots, f_{p-1}\rangle/c_g/\langle \mathbf{c}\rangle \overset{\delta}{\underset{\bowtie_t}{\mapsto}} \mathcal{E}, \mathbf{s}, (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\mathbf{F}))\ (\mathbf{send}\ \langle (\mathbf{init}\ (f_0, p)), \ldots, (\mathbf{init}\ (f_{p-1}, p))\rangle))[\bullet]/c_g \oplus 1/\langle \mathbf{c}\rangle \tag{7.5}$$

avec $\mathbf{F} = \lambda.\lambda.\mathbf{if}\ (0 \leq \overline{1})\mathbf{and}(\overline{1} < \mathbf{p})\ \mathbf{then}\ (\mathbf{access}\ (\overline{2}, \overline{1}))\ \mathbf{else}\ \mathbf{nc}$

Figure 7.4 — Règles des primitives parallèles avec super-threads

(ceux-ci ne sont pas modifiés). Nous ne les redétaillons pas. En fait, les primitives se réécrivent avec des opérations de plus «bas niveau». Il est donc normal que leurs règles ne changent pas. Ce sont ces opérations de plus «bas niveau» qui vont voir leur fonctionnement modifié.

Nous avons alors les règles suivantes pour ces opérateurs parallèles :

$$\mathcal{E}, \mathbf{s}, \mathbf{delpar}\ \langle f, \ldots, f\rangle/c_g/\langle c\rangle \overset{\delta}{\underset{\bowtie_t}{\mapsto}} \mathcal{E}, \mathbf{s}, f/c_g \oplus 1/\langle c\rangle \tag{7.6}$$

$$\mathcal{E}, \mathbf{s}, \mathbf{GloId}\ v/c_g/\langle c\rangle \overset{\delta}{\underset{\bowtie_t}{\mapsto}} \mathcal{E}, \mathbf{s}, v/c_g \oplus 1/\langle c\rangle \text{ si } \mathcal{V}_\bullet(v) = \mathbf{true} \tag{7.7}$$

Les règles 7.6 et 7.7 sont similaires à celles de la sémantique à «petits pas» sans les super-threads, mais avec des environnements de communications qui ne sont pas modifiés. Nous ne les redétaillons pas. Nous avons aussi les règles suivantes pour ces opérateurs parallèles :

$$\mathcal{E}, \mathbf{s}, \mathbf{send}\ \langle \ldots, [\ v_0^i\ , \ldots,\ v_{p-1}^i\ ]\ , \ldots\rangle/c_g/\langle c\rangle \overset{\delta}{\underset{\bowtie_t}{\mapsto}} \mathcal{E} \cup \langle \ldots, [\ v_0^i\ , \ldots,\ v_{p-1}^i\ ]\ , \ldots\rangle_{\mathbf{s}}^t, \mathbf{s}, \mathbf{rcv_s}/c_g \oplus 1/\langle c\rangle \tag{7.8}$$

$$\mathcal{E} \cup \langle \ldots, [\ v_0^i\ , \ldots,\ v_{p-1}^i\ ]\ , \ldots\rangle_{\mathbf{s}}^t, \mathbf{s}+1, \mathbf{rcv_s}/c_g/\langle c\rangle \overset{\delta}{\underset{\bowtie_t}{\mapsto}} \mathcal{E}, \mathbf{s}+1, \langle \ldots, [\ v_0^i\ , \ldots,\ v_{p-1}^i\ ]\ , \ldots\rangle/c_g/\langle c\rangle \tag{7.9}$$

Remarquons que **send** (règle 7.8) ne fait plus l'échange des données. Il se contente de mettre les valeurs à envoyer dans l'environnement de communications. Il se transforme ensuite en **rcv** qui lui va lire dans l'environnement de communications les valeurs reçues. C'est donc **rcv** qui fait «l'échange» en lisant les valeurs échangées (lors de la phase des communications de la super-étapes) dans l'environnement de communications. C'est après cet échange qu'on passe à la super-étape suivante (passage que nous allons détailler par la suite). Notons que cet opérateur n'a de sens que si, les valeurs dans l'environnement sont marquées comme appartenant à la super-étape précédente. Cela permet de ne lire que les valeurs qui ont été échangées et non celles qui viennent juste d'être mises dans l'environnement. Notons aussi que le coût de $\mathbf{rcv_s}$ (règle (7.9)) est considéré comme nul car cette étape de calcul n'apparaît pas dans la sémantique à «petits pas» normale. Nous pouvons ainsi comparer plus facilement les coûts des deux sémantiques : nous n'avons pas à compter le nombre de **rcv** qui ont été réduits (il nous faudrait une algèbre de coûts plus fine).

$$\mathcal{E}, \mathbf{s}, \phi\Gamma[\mathbf{super}\ f_1\ f_2]\flat_t/c_g/\langle\mathbf{c}\rangle \underset{t}{\mapsto} \mathcal{E}, \mathbf{s}, (\phi\Gamma[((f_1\ ()), \mathbf{wait}_{t.2})]\flat_{t.1} \bowtie \phi(f_2\ ())\flat_{t.2})/c_g \oplus 1/\langle\mathbf{c}\rangle \tag{7.10}$$

$$\mathcal{E}, \mathbf{s}, (\phi\Gamma[\mathbf{wait}_{t.2}]\flat_{t.1} \bowtie \phi v_2\flat_{t.2})/c_g/\langle\mathbf{c}\rangle \underset{t}{\mapsto} \mathcal{E}, \mathbf{s}, \phi\Gamma[v_2]\flat_t/c_g/\langle\mathbf{c}\rangle \tag{7.11}$$

Figure 7.5 — Création et destruction des super-threads, la superposition

Nous avons alors les règles de «passage» permettant d'appliquer les règles génériques :

$$\frac{e \overset{\varepsilon}{\mapsto} e', c}{e \underset{i}{\overset{\varepsilon}{\mapsto}} e', c} \qquad \frac{e \overset{\delta}{\mapsto} e', c}{e \underset{i}{\overset{\delta}{\mapsto}} e', c}$$

$$\frac{e \overset{\varepsilon}{\mapsto} e', c}{\mathcal{E}, \mathbf{s}, e/c_g/\langle\mathbf{c}\rangle \underset{\bowtie_t}{\overset{\varepsilon}{\mapsto}} \mathcal{E}, \mathbf{s}, e'/c_g \oplus c/\langle\mathbf{c}\rangle} \qquad \frac{e \overset{\delta}{\mapsto} e', c}{\mathcal{E}, \mathbf{s}, e/c_g/\langle\mathbf{c}\rangle \underset{\bowtie_t}{\overset{\delta}{\mapsto}} \mathcal{E}, \mathbf{s}, e'/c_g \oplus c/\langle\mathbf{c}\rangle}$$

Celles-ci sont comme à l'accoutumée. Nous ne les redétaillons pas.

7.3.5 Règles de la superposition

La figure 7.5 donne les règles pour l'opérateur de création des super-threads, l'opérateur de superposition (**super**) ainsi que pour celui qui permet à un super-thread d'attendre la fin de l'évaluation de son «enfant».

Dans la règle 7.10, un nouveau super-thread est créé en tant qu'«enfant» du super-thread courant. Celui-ci est renommé afin de garder la structure des arbres des super-threads. Sur le super-thread courant, une nouvelle paire est créée. Dans la première composante de la paire, nous calculons le premier argument de la superposition. Dans la deuxième composante, nous plaçons un **wait** qui attendra la fin des calculs du super-thread «enfant».

La fin de cette attente est donnée par la règle 7.11. Dans cette règle, le super-thread «enfant» a terminé ses calculs (nous avons donc une valeur) et cette valeur est récupérée par le **wait** pour être inséreée dans la paire (cette paire est dans le contexte Γ). Notons aussi que le coût de cette règle est considéré comme nul car cette étape de calcul n'apparaît pas dans la sémantique à «petits pas» normale. Nous pouvons ainsi comparer plus facilement les coûts des deux sémantiques.

7.3.6 Contextes d'évaluation

On constate aisément qu'il n'est pas toujours possible de faire des réductions de tête. Il faut donc ajouter des règles de contexte qui permettront :

1. De choisir le super-thread qui va réduire son expression,
2. De réduire cette expression, c'est-à-dire d'appliquer soit une règle globale, soit une règle locale

Les contextes pour les expressions sont ceux habituels. Ils sont rappelés dans la figure 7.6.
Les contextes des super-threads sont les suivants :

$$\begin{array}{lcllcl}
\Gamma_{\mathbf{s}} & ::= & [] & | & (\Gamma_{\mathbf{s}} \bowtie E) & | \quad (\mathcal{V}_{\mathbf{s}} \bowtie \Gamma_{\mathbf{s}}) \\
\mathcal{V}_{\mathbf{s}} & ::= & (\phi\Gamma[\mathbf{wait}_{t'}]\flat_t \bowtie \mathcal{R}_{\mathbf{s}}) & | & (\mathcal{V}_{\mathbf{s}} \bowtie \mathcal{T}_{\mathbf{s}}) & \\
\mathcal{T}_{\mathbf{s}} & ::= & \phi v\flat_t & | & \phi\Gamma[\mathbf{rcv}_{\mathbf{s}}]\flat_t & | \quad \mathcal{V}_{\mathbf{s}} \\
\mathcal{R}_{\mathbf{s}} & ::= & \phi\Gamma[\mathbf{rcv}_{\mathbf{s}}]\flat_t & | & \mathcal{V}_{\mathbf{s}} &
\end{array}$$

Le contexte $\mathcal{V}_{\mathbf{s}}$ est un arbre de super-threads dont toutes les expressions ont été évaluées, en une valeur mais aussi dans l'attente des communications ou de la fin des calculs du super-thread «enfant». Notons qu'avec les contextes $\mathcal{R}_{\mathbf{s}}$ et $\mathcal{T}_{\mathbf{s}}$, il est impossible d'avoir un contexte de la forme : $(\phi\Gamma[\mathbf{wait}_{t.2}]\flat_{t.1} \bowtie \phi v\flat_{t.2})$ car celui-ci n'est pas un arbre de super-threads évalués, **wait** peut être réduit (règle (7.11)). Notons que le contexte $\mathcal{R}_{\mathbf{s}}$ ne peut être simplement un $\phi v\flat_t$, c'est-à-dire un super-thread entièrement évalué.

Avec ces contextes, nous pouvons réduire «dans nos super-threads» ainsi que dans les sous-expressions.

$$
\begin{array}{rcl}
\Gamma & ::= & [] \\
& | & \Gamma\ e \\
& | & v\ \Gamma \\
& | & (\Gamma, e) \\
& | & (v, \Gamma) \\
& | & \textbf{if } \Gamma \textbf{ then } e \textbf{ else } e \\
& | & (\textbf{mkpar } \Gamma) \\
& | & (\textbf{apply } \Gamma\ e) \\
& | & (\textbf{apply } v\ \Gamma) \\
& | & (\textbf{super } \Gamma\ e) \\
& | & (\textbf{super } v\ \Gamma) \\
& | & (\textbf{put } \Gamma) \\
& | & (\textbf{proj } \Gamma)
\end{array}
\qquad
\begin{array}{rcl}
\Delta_i & ::= & \Delta_i\ e \\
& | & v\ \Delta_i \\
& | & (\Delta_i, e) \\
& | & (v, \Delta_i) \\
& | & \textbf{if } \Delta_i \textbf{ then } e \textbf{ else } e \\
& | & (\textbf{mkpar } \Delta_i) \\
& | & (\textbf{apply } \Delta_i\ e) \\
& | & (\textbf{apply } v\ \Delta_i) \\
& | & (\textbf{super } \Delta_i\ e) \\
& | & (\textbf{super } v\ \Delta_i) \\
& | & (\textbf{put } \Delta_i) \\
& | & (\textbf{proj } \Delta_i) \\
& | & \langle e, \ldots, \overbrace{\Gamma^l[e]}^{i}, \ldots, e \rangle
\end{array}
\qquad
\begin{array}{rcl}
\Gamma^l & ::= & [] \\
& | & \Gamma^l\ e \\
& | & v\ \Gamma^l \\
& | & (\Gamma^l, e) \\
& | & (v, \Gamma^l) \\
& | & \textbf{if } \Gamma^l \textbf{ then } e \textbf{ else } e \\
& | & [\Gamma^l, e_1, \ldots, e_n] \\
& | & [v_0, \Gamma^l, \ldots, e_n] \\
& | & \ldots \\
& | & [v_0, v_1, \ldots, \Gamma^l]
\end{array}
$$

Figure 7.6 — Contextes d'évaluation

7.3.7 Communications

Dans cette nouvelle sémantique, les communications ne proviennent plus directement d'une règle d'un opérateur, mais elles ne sont effectuées que lorsque tous les super-threads sont soit des valeurs, soit dans l'attente d'un «enfant» (**wait**) ou de l'échange des valeurs (**rcv**). Cette règle ne s'applique donc que sur un arbre de super-threads de «valeurs» comme suit :

$$
\begin{array}{l}
\{\ldots, \langle [v_0^0, \ldots, v_{p-1}^0], \ldots, [v_0^{p-1}, \ldots, v_{p-1}^{p-1}] \rangle_{\mathbf{s}}^t, \ldots\}, \mathbf{s}, \mathcal{V}_{\mathbf{s}} / c_g / \langle c_0, \ldots, c_{p-1} \rangle \ \mapsto \\
\quad \{\ldots, \langle [v_0^0, \ldots, v_0^{p-1}], \ldots, [v_{p-1}^0, \ldots, v_{p-1}^{p-1}] \rangle_{\mathbf{s}}^t, \ldots\}, \mathbf{s}+1, \mathcal{V}_{\mathbf{s}} / c'_g / \langle 0, \ldots, 0 \rangle
\end{array}
\qquad (7.12)
$$

avec

$$
c'_g = c_g \oplus \max_{i=0}^{p-1}(c_i) \oplus \max_{i=0}^{p-1}(\max \left\{ \begin{array}{l} \sum_{\forall t}(\sum_{j=0}^{p-1} \mathcal{S}_\bullet(v_j^i)) \text{ tel que } v_j^i \in \langle \cdots \rangle_{\mathbf{s}}^t \\ \sum_{\forall t}(\sum_{j=0}^{p-1} \mathcal{S}_\bullet(v_i^j)) \text{ tel que } v_i^j \in \langle \cdots \rangle_{\mathbf{s}}^t \end{array} \right.) \otimes g \oplus l
$$

Les coûts sont bien ceux attendus pour la superposition : les communications et la barrière sont fusionnées. Nous avons donc bien qu'une fois l, le $\max$ des calculs locaux et les valeurs sauvegardées dans les environnements de communications sont fusionnés (en prenant les $\max$).

7.3.8 Nouvelle sémantique

Nous pouvons maintenant définir notre nouvelle sémantique avec les règles définies précédemment et les contextes des expressions et des super-threads.

Définition 26 (*Nouvelle sémantique à «petits pas»*).
La sémantique à petits pas est définie par les règles de contexte suivantes :

$$
\frac{\mathcal{E}, \mathbf{s}, \phi\langle e \rangle\flat_t / c_g / \langle c_0, \ldots, c_{p-1} \rangle \ \underset{t}{\mapsto}\ \mathcal{E}', \mathbf{s}, \phi\langle e' \rangle\flat_t / c'_g / \langle c'_0, \ldots, c'_{p-1} \rangle}{\mathcal{E}, \mathbf{s}, \Gamma_s[\phi\langle e \rangle\flat_t] / c_g / \langle c_0, \ldots, c_{p-1} \rangle \ \mapsto\ \mathcal{E}', \mathbf{s}, \Gamma_s[\phi\langle e' \rangle\flat_t] / c'_g / \langle c'_0, \ldots, c'_{p-1} \rangle}
$$

$$
\frac{\mathcal{E}, \mathbf{s}, e / c_g / \langle \mathbf{c} \rangle \ \overset{\bowtie}{\underset{t}{\mapsto}}\ \mathcal{E}', s, e' / c'_g / \langle \mathbf{c} \rangle}{\mathcal{E}, \mathbf{s}, \phi\langle \Gamma[e] \rangle\flat_t / c_g / \langle \mathbf{c} \rangle \ \underset{t}{\mapsto}\ \mathcal{E}', \mathbf{s}, \phi\langle \Gamma[e'] \rangle\flat_t / c'_g / \langle \mathbf{c} \rangle}
$$

$$
\frac{e \overset{i}{\mapsto} e', c}{\mathcal{E}, \mathbf{s}, \phi\langle \Delta_i[e] \rangle\flat_t / c_g / \langle \ldots, c_i, \ldots \rangle \ \underset{t}{\mapsto}\ \mathcal{E}', \mathbf{s}, \phi\langle \Delta_i[e'] \rangle\flat_t / c_g / \langle \ldots, c_i \oplus c, \ldots \rangle}
$$

La première permet de choisir quel super-thread effectuera son calcul et cela de manière déterministe (ce sera toujours ce super-thread tant qu'il n'aura pas fini son évaluation). Ensuite, les deux suivantes permettent de choisir soit une réduction globale, soit une réduction locale.

Nous avons alors les résultats suivants.

Lemme 36 (*Confluence forte*)

Si	$\mathcal{E}, \mathbf{s}, E, e/c_g/\langle c_0, \cdots, c_{p-1}\rangle \mapsto \mathcal{E}^1, \mathbf{s}^1, E^1, e^1/c_g^1/\langle c_0^1, \cdots, c_{p-1}^1\rangle$
et	$\mathcal{E}, \mathbf{s}, E, e/c_g/\langle c_0, \cdots, c_{p-1}\rangle \mapsto \mathcal{E}^2, \mathbf{s}^2, E, e^2/c_g^2/\langle c_0^2, \cdots, c_{p-1}^2\rangle$
alors il existe	$\mathcal{E}^1, \mathbf{s}^1, E^1, e^1/c_g^1/\langle c_0^1, \cdots, c_{p-1}^1\rangle \mapsto \mathcal{E}^3, \mathbf{s}^3, E^3, e^3/c_g^3/\langle c_0^3, \cdots, c_{p-1}^3\rangle$
et	$\mathcal{E}^2, \mathbf{s}^2, E^2, e^2/c_g^2/\langle c_0^2, \cdots, c_{p-1}^2\rangle \mapsto \mathcal{E}^3, \mathbf{s}^3, E^3, e^3/c_g^3/\langle c_0^3, \cdots, c_{p-1}^3\rangle$.

Preuve. Voir en section 7.A.1 de l'annexe de ce chapitre. ■

Théorème 11 (*Confluence*)

Si	$e = \mathcal{T}_\bullet(e^p)$
alors si	$\{\}, 0, \phi e[\bullet]\flat_1/0/\langle 0, \ldots, 0\rangle \overset{*}{\mapsto} \mathcal{E}, \mathbf{m}, \phi v\flat_1/c_g/\langle c_0, \cdots, c_{p-1}\rangle$
et	$\{\}, 0, \phi e[\bullet]\flat_1/0/\langle 0, \ldots, 0\rangle \overset{*}{\mapsto} \mathcal{E}', \mathbf{m}', \phi v'\flat_1/c'_g/\langle c'_0, \cdots, c'_{p-1}\rangle$
alors	$v = v'$, $c_g = c'_g$, $\mathcal{E} = \mathcal{E}'$, $\mathbf{m} = \mathbf{m}'$ *et* $\forall i \in \{0, \ldots, p-1\}\ c_i = c'_i$.

Preuve. La relation $\mapsto$ est fortement confluente, donc, d'après le lemme 1, elle est confluente. ■

Théorème 12 (*Équivalence*)

Si $e = \mathcal{T}_\bullet(e^p)$ *alors*

1. *si* $\{\}, 0, \phi e[\bullet]\flat_1/0/\langle 0, \ldots, 0\rangle \overset{*}{\mapsto} \{\}, \mathbf{m}, \phi v\flat_1/c_g/\langle c_0, \ldots, c_{p-1}\rangle$
   alors $e[\bullet]/0/\langle 0, \ldots, 0\rangle \overset{*}{\rightharpoonup} v/c'_g/\langle c'_0, \ldots, c'_{p-1}\rangle$
   avec $c_g \oplus \max_{i=0}^{p-1}(c_i) \leq c'_g \oplus \max_{i=0}^{p-1}(c'_i)$

2. *si* $e[\bullet]/0/\langle 0, \ldots, 0\rangle \overset{*}{\rightharpoonup} v/c'_g/\langle c'_0, \ldots, c'_{p-1}\rangle$
   alors $\{\}, 0, \phi e[\bullet]\flat_1/0/\langle 0, \ldots, 0\rangle \overset{*}{\mapsto} \{\}, \mathbf{m}, \phi v\flat_1/c_g/\langle c_0, \ldots, c_{p-1}\rangle$
   avec $c_g \oplus \max_{i=0}^{p-1}(c_i) \leq c'_g \oplus \max_{i=0}^{p-1}(c'_i)$

Preuve. Voir en section 7.A.2 de l'annexe de ce chapitre. ■

Prenons par exemple, l'expression suivante :

$$(\mathbf{super}\ (\lambda.\mathbf{bcast}\ 0\ (\mathbf{mkpar}\ (\lambda.\overline{1})))\ (\lambda.\mathbf{bcast}\ 1\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))$$

Nous pouvons évaluer cette expression avec la sémantique à petits pas normale et trois processeurs (figure 7.7). Nous pouvons aussi l'évaluer avec la sémantique à petits pas et les super-threads[2] (figure 7.8) et nous avons bien $17 \oplus g \oplus l \oplus 22 \leq 17 \oplus 2 \otimes (g \oplus l) \oplus 22$.

7.4 Implantation de la superposition parallèle

L'implantation des super-threads (qui sont nécessaires à la superposition) utilise les processus légers de OCaml. Chaque super-thread est défini comme un processus léger associé à un identificateur et un canal de commmunications à la Concurrent ML [217, 230] qui permet d'éveiller ou d'endormir le super-thread comme dans la stratégie décrite dans la section antépénultième. Un ordonnanceur spécifique a été implanté pour cette tâche. Nous avons aussi besoin d'un environnement de communication défini comme une table de hachage où les clés sont les identificateurs des super-threads et d'une implantation des deux opérateurs de «bas niveau» : wait et rcv.

[2] Quelques détails de la sémantiques sont omis.

$$
\begin{array}{rl}
 & (\mathbf{super}\ (\lambda.\mathbf{bcast}\ 0\ (\mathbf{mkpar}\ (\lambda.\overline{1})))\ (\lambda.\mathbf{bcast}\ 1\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet]/0/\langle 0,\ldots,0\rangle \\
\ldots & \\
\rightharpoonup & (\overline{((\lambda.\mathbf{bcast}\ 0\ (\mathbf{mkpar}\ (\lambda.\overline{1})))[\bullet]\ ())}, (\overline{(\lambda.\mathbf{bcast}\ 1\ (\mathbf{mkpar}\ (\lambda.\overline{1})))[\bullet]\ ())})/3/\langle 0,\ldots,0\rangle \\
\ldots & \\
\rightharpoonup & ((\mathbf{bcast}\ 0\ \langle 0,1,2\rangle), \overline{(\lambda.\mathbf{bcast}\ 1\ (\mathbf{mkpar}\ (\lambda.\overline{1})))[\bullet]\ ())})/5/\langle 1,\ldots,1\rangle \\
\ldots & \\
\rightharpoonup & ((\mathbf{bcast}\ 0\ \langle 0,1,2\rangle), (\mathbf{bcast}\ 1\ \langle 0,1,2\rangle))/7/\langle 2,\ldots,2\rangle \\
\ldots & \\
\rightharpoonup & (\langle 0,\ldots,0\rangle, (\mathbf{bcast}\ 1\ \langle 0,1,2\rangle))/12 \oplus 1\otimes g\otimes l/\langle 12,\ldots,12\rangle \\
\ldots & \\
\rightharpoonup & (\langle 0,\ldots,0\rangle, \langle 1,\ldots,1\rangle)/17 \oplus 1\otimes g\otimes l \oplus 1\otimes g\otimes l/\langle 22,\ldots,22\rangle \\
\equiv & (\langle 0,\ldots,0\rangle, \langle 1,\ldots,1\rangle)/17 \oplus 2\otimes (g\oplus l)/\langle 22,\ldots,22\rangle
\end{array}
$$

Figure 7.7 — Evaluation avec la sémantique à petits pas

$$
\begin{array}{rl}
 & \{\}, 0, \not\langle(\mathbf{super}\ (\lambda.\mathbf{bcast}\ 0\ (\mathbf{mkpar}\ (\lambda.\overline{1})))\ (\lambda.\mathbf{bcast}\ 1\ (\mathbf{mkpar}\ (\lambda.\overline{1}))))[\bullet]\not\rangle_1/0/\langle 0,\ldots,0\rangle \\
\ldots & \\
\mapsto & \{\}, 0, \not\langle(\overline{((\lambda.\mathbf{bcast}\ 0\ (\mathbf{mkpar}\ (\lambda.\overline{1})))[\bullet]\ ())}, \mathbf{wait}_{1.2})\not\rangle_{1.1} \bowtie \not\langle(\overline{(\lambda.\mathbf{bcast}\ 1\ (\mathbf{mkpar}\ (\lambda.\overline{1})))[\bullet]\ ())}\not\rangle_{1.2}/3/\langle 0,\ldots,0\rangle \\
\ldots & \\
\mapsto & \{\}, 0, \not\langle(, (\mathbf{bcast}\ 0\ \langle 0,1,2\rangle), \mathbf{wait}_{1.2})\not\rangle_{1.1} \bowtie \not\langle(\overline{(\lambda.\mathbf{bcast}\ 1\ (\mathbf{mkpar}\ (\lambda.\overline{1})))[\bullet]\ ())}\not\rangle_{1.2}/5/\langle 1,\ldots,1\rangle \\
\ldots & \\
\mapsto & \{\}, 0, \not\langle((\mathbf{bcast}\ 0\ \langle 0,1,2\rangle), \mathbf{wait}_{1.2})\not\rangle_{1.1} \bowtie \not\langle(\mathbf{bcast}\ 1\ \langle 0,1,2\rangle)\not\rangle_{1.2}/7/\langle 2,\ldots,2\rangle \\
\ldots & \\
\mapsto & \{\langle 0,1,2\rangle_0^{1.1}, \langle 0,1,2\rangle_0^{1.2}\}, 0, \not\langle((\mathbf{F}\ \mathbf{rcv}_0^{1.1}), \mathbf{wait}_{1.2})\not\rangle_{1.1} \bowtie \not\langle(\mathbf{F}\ \mathbf{rcv}_0^{1.2})\not\rangle_{1.2}/11/\langle 12,\ldots,12\rangle \\
\mapsto & \{\langle 0,0,0\rangle_0^{1.1}, \langle 1,1,1\rangle_0^{1.2}\}, 1, \not\langle((\mathbf{F}\ \mathbf{rcv}_0^{1.1}), \mathbf{wait}_{1.2})\not\rangle_{1.1} \bowtie \not\langle(\mathbf{F}\ \mathbf{rcv}_0^{1.2})\not\rangle_{1.2}/11 \oplus g \oplus l/\langle 12,\ldots,12\rangle \\
\mapsto & \{\langle 1,1,1\rangle_0^{1.2}\}, 1, \not\langle((\mathbf{F}\ \langle 0,0,0\rangle), \mathbf{wait}_{1.2})\not\rangle_{1.1} \bowtie \not\langle(\mathbf{F}\ \mathbf{rcv}_0^{1.2})\not\rangle_{1.2}/11/\langle 12,\ldots,12\rangle \\
\ldots & \\
\mapsto & \{\langle 1,1,1\rangle_0^{1.2}\}, 1, \not\langle(\langle 0,0,0\rangle, \mathbf{wait}_{1.2})\not\rangle_{1.1} \bowtie \not\langle(\mathbf{F}\ \mathbf{rcv}_0^{1.2})\not\rangle_{1.2}/14 \oplus g \oplus l/\langle 17,\ldots,17\rangle \\
\ldots & \\
\mapsto & \{\}, 1, \not\langle(\langle 0,0,0\rangle, \mathbf{wait}_{1.2})\not\rangle_{1.1} \bowtie \not\langle(\mathbf{F}\ \langle 1,1,1\rangle)\not\rangle_{1.2}/14 \oplus g \oplus l/\langle 17,\ldots,17\rangle \\
\mapsto & \{\}, 1, \not\langle(\langle 0,0,0\rangle, \mathbf{wait}_{1.2})\not\rangle_{1.1} \bowtie \not\langle\langle 1,1,1\rangle\not\rangle_{1.2}/17 \oplus g \oplus l/\langle 22,\ldots,22\rangle \\
\mapsto & \{\}, 1, \not\langle(\langle 0,0,0\rangle, \langle 1,1,1\rangle)\not\rangle_1/17 \oplus g \oplus l/\langle 22,\ldots,22\rangle
\end{array}
$$

Figure 7.8 — Evaluation avec la sémantique à petits pas et les super-threads

7.4.1 Nouvelle implantation des primitives BSML

Comme dans l'implantation précédente, les primitives sont implantées dans un style SPMD. Cette implantation dépend du module de communication, d'un module de l'ordonnanceur des super-threads appelé Scheduler, d'un module pour l'environnement de communications appelé EnvComm et d'un module générique d'opérations de «bas niveau» pour les super-threads appelé SuperThread.

L'implantation du type abstrait des vecteurs parallèles et des primitives BSML plates n'a pas changé. Elle respecte les règles 7.2, 7.3, 7.5 et 7.4 de la sémantique à petits pas. Notons que nous avons réduit le code comme le préconise le chapitre 4. Par exemple, l'opération **delpar**, n'étant rien d'autre que l'identité, a donc été supprimée afin de rendre le code plus lisible et efficace. Nous avons donc :

```
type α par = α
let mkpar f = f (Comm.pid())
let apply f v = (f v)
let mkfuns = (fun res i →if ((0<=i)&&(i<(!nprocs))) then res.(i) else None)
let put f = mkfuns (send (Array.init (!nprocs) f))
let proj v = put (fun _ →v)
```

Par contre, il n'est plus possible d'utiliser directement la fonction send du module de communication. En effet, celle-ci effectue les communications entre les processeurs et effectue une barrière de synchronisation.

```
module SuperThread : functor (How_Comm : sig val make_comm : (unit →unit) ref end) →sig
  val envComm : unit EnvComm.t
  val our_schedule : Scheduler.t
  val rcv : unit →α
  type β data_of_thread
  val create_child: (unit→β )→ β data_of_thread
  val wait: β data_of_thread →β
 end
```

Figure 7.9 — Le module des super-threads

Les communications ne doivent être effectuées que lorsque tous les super-threads ont terminé leurs phases de calculs asynchrones. Comme le précise la sémantique, les valeurs à communiquer sont stockées dans un environnement de communications. Ceci est effectué par la règle 7.8 de l'opérateur send, ce qui suggère l'implantation suivante :

```
let send v =
 let id=(Scheduler.pid_superthread_run SuperThread.our_schedule) in
  EnvComm.add SuperThread.envComm id v; (SuperThread.rcv())
```

Cet opérateur prend l'identificateur du super-thread actif (à partir de la fonction Scheduler.pid_superthread_run de l'ordonnanceur SuperThread.our_schedule) et met cette valeur dans l'environnement de communications (EnvComm). Ensuite, comme le précise la règle, la valeur recue est retournée par l'opérateur rcv après la phase de communication.

La primitive **super** est aussi implantée selon la sémantique (règle 7.10), c'est-à-dire que nous construisons une paire où la première composante est le calcul de f1 et la seconde l'opérateur **wait** qui attend le résultat d'un nouveau super-thread (enfant du super-thread actif) calculant f2. La fonction SuperThread.create_child fait exécuter ce nouveau super-thread et retourne immédiatement son identificateur qui sera l'argument de l'opérateur **wait** :

```
let super f1 f2 =
 let t=(SuperThread.create_child f2) and v=f1 ()
  in (v, SuperThread.wait t)
```

Notons que les calculs sont effectués en dehors de la paire car dans le code natif de OCaml, l'ordre d'évaluation des composantes d'une paire n'est pas spécifié et dépend des optimisations que peut apporter le compilateur (ce qui peut avoir comme conséquence de casser la stratégie des super-threads qui a été définie.

7.4.2 Fonctions nécessaires à l'implantation de la superposition

Le module de l'implantation des super-threads est un foncteur basé sur les éléments donnés à la figure 7.9. Nous avons un environnement de communications, un ordonnanceur, le type astrait des super-threads et les opérations nécessaires à l'implantation de la superposition.

Le foncteur est paramétré par un module qui contient une référence sur une fonction qui effectue les communications. Cette référence est initialement vide et est affectée au début du programme avec une fonction manipulant l'environnement de communications (itérations sur la table de hachage) et communiquant les valeurs avec la fonction send du module Comm.

La première opération de bas niveau, rcv, fonctionne comme dans la règle (7.9) de la sémantique à petits pas : l'opération retourne et supprime de l'environnement de communications, la valeur reçue par le super-thread lors de la phase de communication. rcv est implantée (figure 7.10) avec les fonctions de l'ordonnanceur comme suit : nous testons d'abord si le super-thread actif est le dernier des super-threads à faire le calcul asynchrone de la super-étape courante (nous utilisons la fonction am_i_the_last de l'ordonnanceur) ; si c'est le cas, les communications sont effectuées (il en existe car au moins ce super-thread effectue un **rcv**) car nous sommes à la fin de la super-étape ; ensuite, nous testons si le super-thread actif n'est pas le seul. Si c'est encore le cas, le premier des super-threads devient le super-thread actif (stratégie des super-threads) en endormant l'actuel super-thread actif (celui-ci reprendra la main ultérieusement) ; Si le super-thread actif est seul, nous retournons simplement la valeur lue dans l'environnement de communication.

```
let rcv () =
 if (Scheduler.am_i_the_last our_schedule) then ((!How_Comm.make_comm)());
 if not(Scheduler.am_i_the_only_one our_schedule) then
  begin
   let my_ch=Scheduler.ch_superthread_run our_schedule in
    Scheduler.to_next our_schedule;
    Scheduler.wake (Scheduler.ch_superthread_run our_schedule);
    Scheduler.to_sleep my_ch end;
 let id = Scheduler.pid_superthread_run our_schedule in
 let v = EnvComm.find envComm id in
  EnvComm.remove envComm id; v


let wait child =
  child.end_of_father<−true;
  if not(child.end_of_child) then begin
      Scheduler.remove_superthread_run our_schedule;
      Scheduler.wake (Scheduler.ch_superthread_run our_schedule);
      Scheduler.to_sleep child.ch_of_father end;
  child.value_of_the_child


let end_child child =
  child.end_of_child<−true;
 if child.end_of_father then
  begin
   Scheduler.child_add_father our_schedule child.id_of_father child.chan_of_father;
   Scheduler.wake child.chan_of_father
  end
   else begin
     (if Scheduler.am_i_the_last our_schedule then ((!How_Comm.make_comm)()));
     Scheduler.remove_superthread_run our_schedule;
     Scheduler.wake (Scheduler.ch_superthread_run our_schedule) end
```

Figure 7.10 — L'opération de bas niveau **rcv**

La seconde opération de «bas niveau», wait, retourne le résultat final du super-thread «enfant» (figure 7.10). Premièrement, nous testons si le super-thread «enfant» a terminé ses calculs. Si ce n'est pas le cas, le super-thread «père» attend ce résultat et est retiré de l'ordonnanceur avec la fonction remove_super thread. Nous réveillons alors le super-thread suivant de l'ordonnanceur. L'«enfant» sera réveillé par la suite par l'ordonnanceur. Pour terminer, le résulat de l'«enfant» est retourné.

La dernière opération create_child est la création d'un super-thread «enfant». Le nouveau super-thread est initialisé en tant que nouveau processus léger qui est tout de suite endormi (stratégie de la sémantique), effectue son calcul et enfin se termine avec le code de la figure 7.10 : premièrement, nous testons si le «père» a terminé ses calculs. Si c'est le cas, l'enfant réveille son père pour qu'il puisse terminer la superposition ; sinon, le super-thread est retiré de l'ordonnanceur et le super-thread suivant est réveillé ; nous testons ensuite si le super-thread «enfant» est le dernier des super-threads de la super-étape. Si c'est le cas, les communications sont effectuées (il en existe, car le «père» est endormi par l'ordonnanceur en attente d'une valeur par un **rcv**) car nous sommes à la fin de la super-étape courante.

L'ordonnanceur est implanté comme un tableau d'identifiants des super-threads associés avec leurs canaux de communication. Ce tableau décrit continuellement quels sont les super-threads endormis ou actifs. Les fonctions de l'ordonnanceur sont donc implantées comme des fonctions modifiant ce tableau selon la stratégie décrite par la sémantique à petits pas.

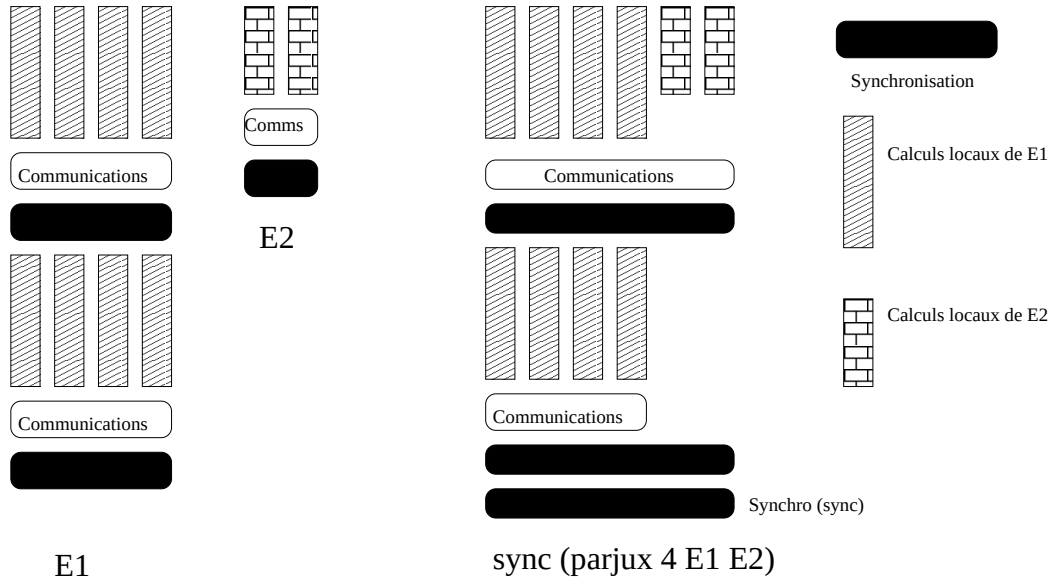


Figure 7.11 — Juxtaposition de deux programmes

7.5 Implantation de la juxtaposition parallèle

La juxtaposition parallèle est une primitive qui a été proposée dans [190] pour permettre de programmer des algorithmes diviser-pour-régner en BSML : deux programmes sont évalués sur la même machine découpée en deux sous-parties disjointes mais tout en conservant une exécution BSP (figure 7.11). Cette section propose une implantation de cette primitive avec la superposition. Pour comprendre cette implantation, nous allons d'abord décrire les modifications qu'ajoute la juxtaposition.

7.5.1 Présentation informelle

Pour évaluer deux programmes parallèles sur la même machine, on peut la partitionner en deux et évaluer indépendamment chacun des programmes sur chacune des partitions. Toutefois, en procédant ainsi le modèle BSP est perdu puisqu'alors la synchronisation globale de chaque sous-machine ne coûtera plus l. Pour conserver le modèle BSP, ce qui est souhaitable [133], il faut donc que les barrières de synchronisation concernent toute la machine.

Considérons le terme $\mathbf{juxta}\ m\ E_1\ E_2$. L'évaluation de ce terme se déroule comme suit : les m premiers processeurs évaluent le terme E_1 et les $p - m$ restant évaluent E_2. Ces $p - m$ processeurs sont toutefois renommés, le processeur m devenant 0 et le $(p-1)$ème devenant $(p-1)-m$. En dehors de ce renommage et du changement de la valeur **bsp_p()** sur chaque sous-machine, l'évaluation est la même que précédemment, à ceci près que l'évaluation de **put** ou de **proj** met en jeu tout le réseau au moment de la synchronisation globale. Un problème arrive toutefois si le nombre de super-étapes nécessaires à l'évaluation de E_1 et E_2 est différent. C'est pourquoi une autre primitive est nécessaire. Il s'agit de la primitive **sync**. Celle-ci appelle en boucle des barrières de synchronisation jusqu'à ce que l'un des appels concerne tout le réseau. La figure 7.11 illustre cette méthode.

Ainsi dans le cas où l'évaluation de E_1 nécessite une super-étape de plus que celle de E_2, l'évaluation de $\mathbf{sync}(\mathbf{juxta}\ m\ E_1\ E_2)$ peut être décrite ainsi :

- Au début, chaque appel de synchronisation globale pour l'évaluation de E_1 correspond à un appel de synchronisation globale pour l'évaluation de E_2
- Ensuite, l'évaluation de E_2 se termine. E_1 fait une demande de synchronisation globale supplémentaire pour sa dernière super-étape. La seconde sous-machine qui a fini d'évaluer E_2 évalue alors le **sync** : c'est un appel à une synchronisation globale qui va être en correspondance avec l'appel de la première sous-machine.
- Chaque sous-machine a fini d'évaluer son terme et les deux font une demande de synchronisation émanant d'un **sync**. Cet appel concernant tout le réseau, l'évaluation du **sync** se termine.

```
let rec scan_juxta op vec =
 if bsp_p'()=1 then vec else
  let mid = bsp_p'()/2 in
  let vec'=juxta' mid (fun ()→scan_juxta op vec) (fun ()→scan_juxta op vec) in
  let msg vec=apply'(mkpar'(fun i v→ if i<>mid−1 then (fun dst→None)
                                 else fun dst→if dst>=mid then Some v else None)) vec
  and parop=parfun2'(fun x y→match x with None→y|Some v→op v y) in
   parop (apply' (put' (msg vec')) (replicate' (mid−1))) vec'
```

Figure 7.12 — Code du calcul parallèle des préfixes avec la juxtaposition

Le résultat de l'évaluation d'une juxtaposition parallèle est un vecteur :

$$\mathbf{juxta}\ m\ \langle\ v_0\ ,\ldots,\ v_{m-1}\ \rangle\ \langle\ v'_0\ ,\ldots,\ v'_{p-1-m}\ \rangle\ =\ \langle v_0,\ldots,v_{m-1},v'_0,\ldots,v'_{p-1-m}\rangle$$

De point de vue fonctionnel, la fonction **sync** est l'identité. Au niveau de la bibliothèque BSML et sachant qu'OCaml est un langage dont la stratégie d'évaluation est une stratégie faible d'appel par valeur, il faut éviter que les deux derniers arguments de la fonction **juxta** et l'argument de la fonction **sync** ne soient évalués. Il faut qu'ils soient des fonctions :

juxta: int→(unit→α **par**)→(unit→α **par**)→α **par**
sync: (unit→α **par**)→α **par**

[190] décrit une sémantique à grands pas d'un BSλ-calcul avec juxtaposition parallèle. Si l'on ajoute naïvement la notion de juxtaposition parallèle au BSλ-calcul, la confluence est perdue. Plusieurs raisons en sont la cause. En particulier, le nombre de processeurs, qui est aussi la taille des vecteurs, n'est plus constant et dépend du contexte dans lequel est évalué le terme. Ainsi, selon la stratégie de réduction, on obtient des résultats différents. Comme dans les chapitres précédents, une stratégie faible d'appel par valeur a été utilisée afin de conserver la confluence du calcul.

7.5.2 Modèle de coût

Le coût de l'évaluation de $(\mathbf{sync}\ V)$ est simplement l si V est une valeur . Pour déterminer le coût de l'évaluation de $(\mathbf{juxta}\ m\ E_1\ E_2)$ il ne suffit pas de considérer les coûts BSP de l'évaluation de E_1 et E_2 de la forme $W + H \times g + l$ mais la liste des coûts de chaque super-étape.

Si $[(W_i^0, h_i^0); (W_i^1, h_i^1); \ldots; (W_i^{k_i}, h_i^{k_i})]$ sont les listes de coûts de chaque super-étape nécessaire à l'évaluation de E_i pour $i = 1$ et $i = 2$ alors le coût de l'évaluation de l'expression $\mathbf{juxta}\ m\ E_1\ E_2$ est :

$$(\sum_{n=0}^{k} \max\{W_1^n, W_2^n\})\ \ +\ \ (\sum_{n=0}^{k} \max\{h_1^n, h_2^n\}) \times g\ \ +\ \ k \times l$$

où $k = \max\{k_1, k_2\}$ et W_i^n (resp. h_i^n) sont considérés égaux à 0 si $n > k_i$.

7.5.3 Exemple

La figure 7.12 donne le code d'une version avec juxtaposition du calcul des préfixes. Le réseau est divisé en deux parties et la fonction scan_juxta y est appliquée récursivement. La valeur au dernier processeur de la première partie est diffusée à tous les processeurs de la seconde partie. Puis cette valeur et la valeur locale calculée par l'appel récursif sont combinées avec l'opération op sur chaque processeur de la seconde partie.

7.5.4 Implantation avec la superposition

Pour implanter la juxtaposition, nous pouvons utiliser la superposition (figure 7.13) afin de couper le réseau en deux. Chaque super-thread s'occupera d'une sous-partie du réseau. A cause de l'effet de bord sur le renommage des processeurs, nous redéfinissons les primitives BSML afin qu'elles ne fonctionnent que sur une sous-partie du vecteur parallèle.

Nous insérons une valeur nulle nc sur les processeurs qui ne prennent pas part au calcul. Ces sous-parties sont définies en utilisant les bornes (nom abstrait des processeurs de la sous-machines). La juxtaposition

```
let nc = Obj.magic None
let jux_p' = ref bsp_p and jux_f = ref 0
let inbound pid = (!jux_f<=pid)&&(pid<(!jux_f+(!jux_p'())))


let bsp_p' () = !jux_p'()
let mkpar' f = mkpar (fun pid→if (inbound pid) then (f (pid−(!jux_f))) else nc)
let apply' vf vv = apply2 (mkpar (fun pid f v→if (inbound pid) then (f v) else nc)) vf vv


let put' vf =
 let old_p' =(!jux_p'()) and old_f = (!jux_f) in
 let vf'=put (apply (mkpar (fun pid f→
            if (inbound pid) then (fun i→if (inbound i) then (f (i−(!jux_f))) else None)
            else (fun _→None))) vf) in
  jux_p':=(fun ()→old_p');
  jux_f:=old_f;
  apply (mkpar (fun pid f→if (inbound pid) then (fun i→(f (i+(!jux_f)))) else nc)) vf'


let proj' vv =
  let old_p' =(!jux_p'()) and old_f = (!jux_f) in
  let f=proj (apply (mkpar (fun pid v →if (inbound pid) then v else None)) vv) in
   jux_p':=(fun () →old_p');
   jux_f:=old_f;
   (fun pid →if (old_f<=pid)&&(pid<(old_f+(old_p'))) then (f pid) else None)


let juxta' m f1 f2 =
 if (0<m)&&(m<(!jux_p'())) then
  let old_p' =(!jux_p'()) and old_f = (!jux_f) in
  let (va,vb) = super (fun ()→jux_p':=(fun ()→m); f1())
                      (fun ()→jux_p':=(fun ()→old_p'−m); jux_f:=m+old_f; f2()) in
   jux_p':=(fun ()→old_p');jux_f:=old_f;
   apply2 (mkpar (fun pid a b →
       if ((!jux_f)<=pid)&&(pid<(!jux_f+m)) then a
        else if (!jux_f+m<=pid)&&(pid<(!jux_f+(!jux_p'()))) then b else nc)) va vb
 else raise (Parjux "m␣is␣not␣within␣bound")

let sync v = put (fun _ →None);v
```

Figure 7.13 — Code de l'implantation de la juxtaposition

```
(* scan_super: (α →α →α )→α →α par→α par *)
let scan_super op e vec =
  let rec scan' fst lst op vec =
    if fst>=lst then vec
    else let mid = (fst+lst)/2 in
    let vec'= mix mid (super (fun()→scan' fst mid op vec)
                             (fun()→scan'(mid+1) lst op vec)) in
    let msg vec = apply (mkpar(fun i v→
      if i=mid then fun dst→if inbounds (mid+1) lst dst then Some v else None
      else fun dst→ None)) vec
    and parop = parfun2(fun x y→match x with None→y|Some v→op v y) in
    parop (apply(put(msg vec'))(mkpar(fun i→mid))) vec' in
  applyat 0 (fun _ →e) (fun x→x) (scan' 0 (bsp_p()−1) op vec)
```

Figure 7.14 — Code de la «version superposition» du calcul des préfixes

parallèle est, dans ce cas, l'appel à la superposition parallèle avec chaque sous-machine sur chaque super-thread et où les bornes sont modifiées en chaque sous-machine (super-threads). Notons que la valeur nulle nc n'est présente que dans les parties non-utilisées d'un vecteur parallèle.

Les nouvelles primitives de communication sont implantées comme celles qui sont plates mais en redéfinissant leurs fonctions résultantes avec les nouvelles bornes de la sous-machine qui l'utilise. Notons que la primitive **sync** est juste la fonction d'identité puisque la gestion des barrières est affectée aux super-threads. Pour respecter le modèle de coûts de la juxtaposition, une barrière de synchronisation (**put** pour des fonctions retournant toujours None) est ajoutée.

7.6 Exemples et expériences

7.6.1 Exemple d'utilisation

Notre exemple utilisant directement la superposition est une version diviser-pour-régner du calcul des préfixes d'un opérateur sur un vecteur parallèle de valeurs. Dans cette version, le réseau est divisé en deux parties et le calcul des préfixes est récursivement appelé sur ces deux parties. La valeur calculée par le dernier processeur de la première partie est diffusée aux processeurs de la seconde partie. Ensuite, en chaque processeur de cette seconde partie, cette valeur et la valeur calculée localement sont combinées à l'aide de l'opérateur donné en paramètre op. La figure 7.14 donne le code de cette version du calcul des préfixes. Le code utilise les fonctions suivantes :

```
(* inbounds: int→int→int→bool *)
let inbounds first last n = (n>=first)&&(n<=last)

(* mix: int→α par * α par→α par *)
let mix m (v1,v2) = let f pid v1 v2 =
 if pid<=m then v1 else v2 in apply(apply(mkpar f) v1) v2
```

qui teste si un numéro de processeur est compris entre le premier et dernier des processeurs de la partie courante. La seconde fonction permet de combiner les résultats de la première et de la seconde partie (sous forme de vecteur parallèles) en un seul vecteur parallèle.

Dans nos expériences, nous allons faire des comparaisons de performances entre cette version du calcul des préfixes et celles présentées dans le chapitre 2. La première est directe (figure 5.6) et la seconde binaire (figure 5.8). Nous donnons ici à titre d'exemple et afin d'illustrer les différents types d'exécution de ces versions, l'exécution abstraite de la version binaire et celle utilisant la superposition. Pour faire la comparaison, nous utilisons comme opérateur la concaténation de chaînes de caractères (nous avons donc scan (^) (**mkpar** (**fun** i →(string_of_int i)))) et une machine de 10 processeurs. Notons que les 2 versions utilisent $\log_2(p)$ super-étapes.

La version avec superposition calcule de la manière suivante :

$$
\begin{array}{ll}
 & \langle 0,1,2,3,4,5,6,7,8,9\rangle \\
\rightarrow & \langle 0,01,2,3,34,5,56,7,8,89\rangle \\
\rightarrow & \langle 0,01,012,3,34,5,56,567,8,89\rangle \\
\rightarrow & \langle 0,01,012,0123,01234,5,56,567,5678,56789\rangle \\
\rightarrow & \langle 0,01,012,0123,01234,012345,0123456,01234567,012345678,0123456789\rangle
\end{array}
$$

et la version binaire comme suit :

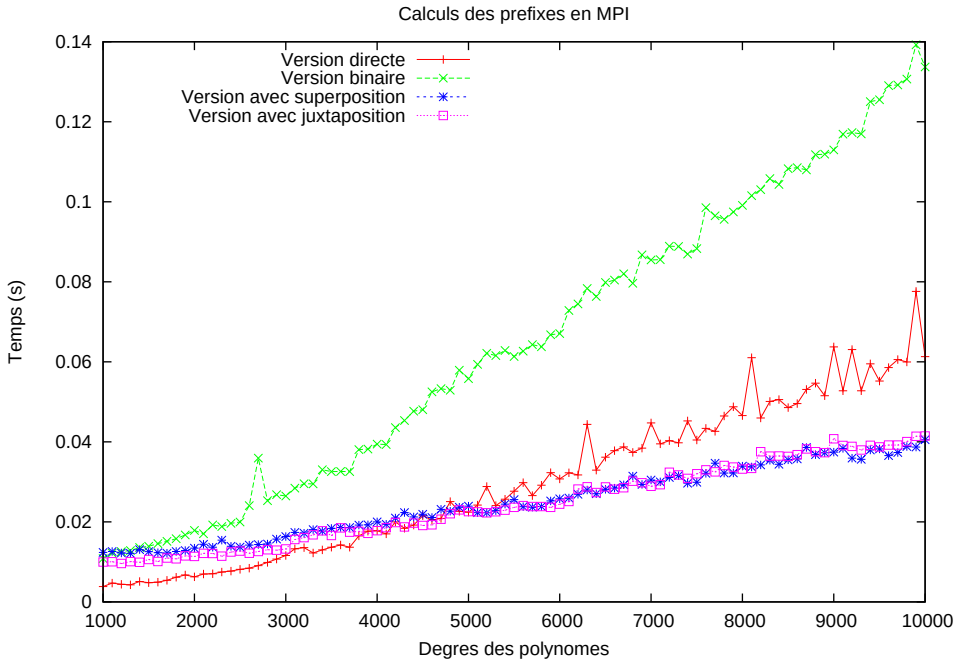
$$
\begin{array}{ll}
 & \langle 0,1,2,3,4,5,6,7,8,9\rangle \\
\rightarrow & \langle 0,01,12,23,34,45,56,67,78,89\rangle \\
\rightarrow & \langle 0,01,012,0123,1234,2345,3456,4567,5678,6789\rangle \\
\rightarrow & \langle 0,01,012,0123,01234,012345,0123456,01234567,12345678,23456789\rangle \\
\rightarrow & \langle 0,01,012,0123,01234,012345,0123456,01234567,012345678,0123456789\rangle
\end{array}
$$

7.6.2 Expériences du calcul des préfixes

Des tests de performances ont été effectués sur une grappe de 10 nœuds ayant chacun 1Go de RAM. Les nœuds sont des Intel pentium IV 2.8 Ghz avec des cartes Gigabit Ethernet et interconnectés par un réseau Gigabit Ethernet (10/100/1000). Une Mandrake clic 2.0 a été utilisée comme système d'exploitation et les programmes ont été compilés avec OCaml 3.08.02 en mode *natif*. Chacun des programmes comporte 100 exécutions du calcul des préfixes. Les programmes ont été exécutés 5 fois et la moyenne des exécutions a été prise pour les graphiques. Les versions de la BSMLlib utilisant MPI et TCP/IP ont été utilisées. L'opération employée pour le calcul des préfixes est la somme de deux polynômes. Les polynômes sont générés de manière aléatoire à chaque calcul des préfixes. Les degrés (identiques en chaque processeur) des polynômes vont croissant.

Sur les figures 7.15 et figures 7.15, la version MPI (graphiques du haut) et la version TCP/IP (graphiques du bas) de la BSMLlib ont été utilisées. Nous notons «avec superposition», la version du calcul des préfixes qui utilise la superposition (figure 7.14) et «avec juxtaposition», la version utilisant la juxtaposition (figure 7.12) simulée par la superposition (figure 7.11). La version directe utilise le code de la figure 5.6 et la version binaire, le code de la figure 5.8.

Nous constatons que la version directe est plus efficace dans le cas de petits polynômes. Par contre, dans le cas de polynômes plus importants, les versions avec superposition et juxtaposition sont plus rapides. La version TCP/IP est dans tous les cas, meilleure que celle MPI. La version «juxtaposition» est plus rapide que la version «superposition» dans le cas de petits polynômes mais plus lente quand leurs degrés augmentent.



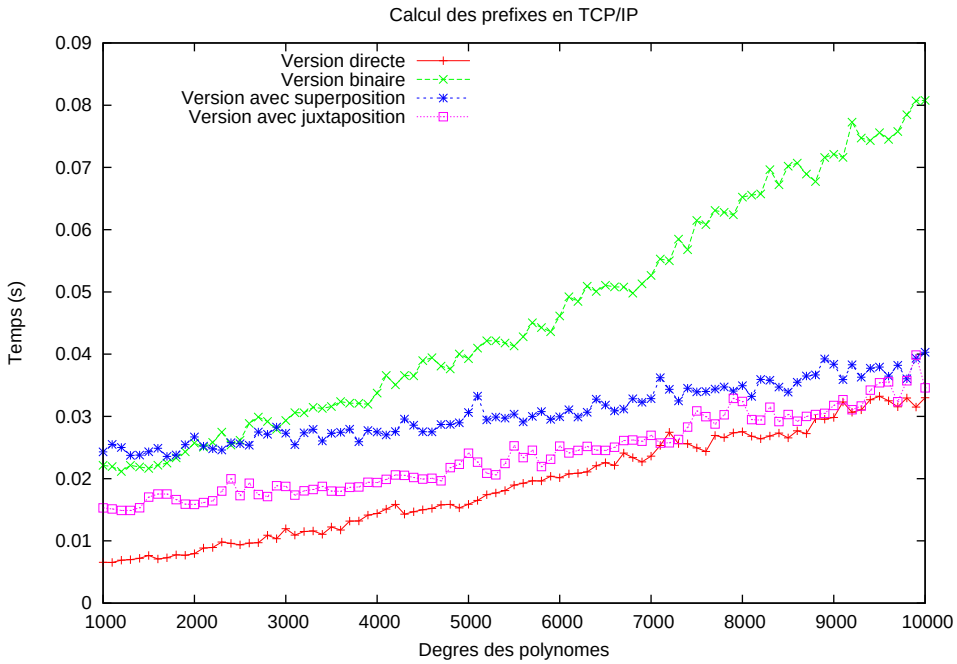
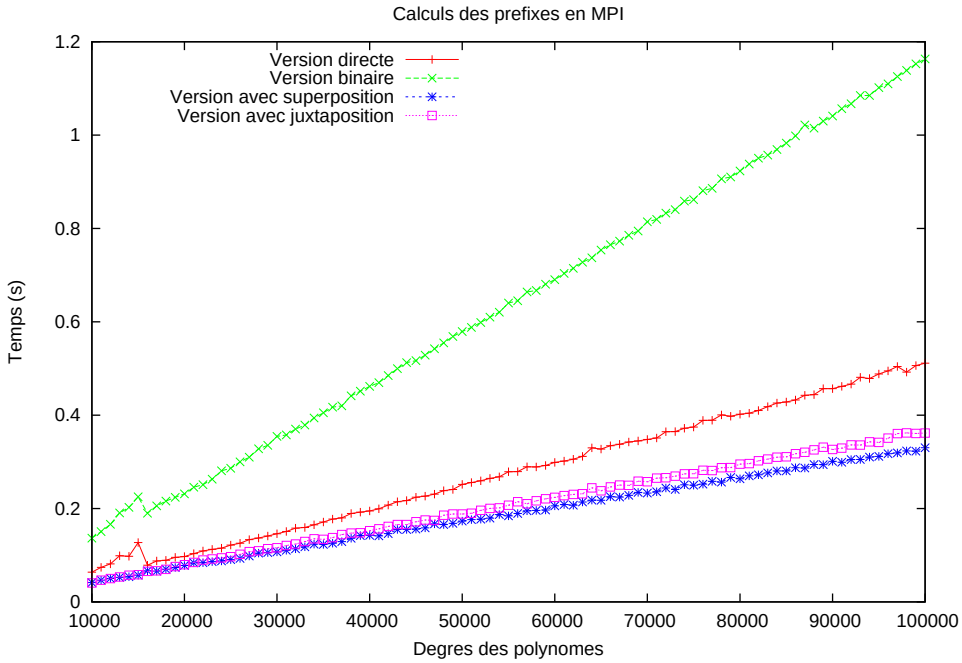


Figure 7.15 — Calcul des préfixes



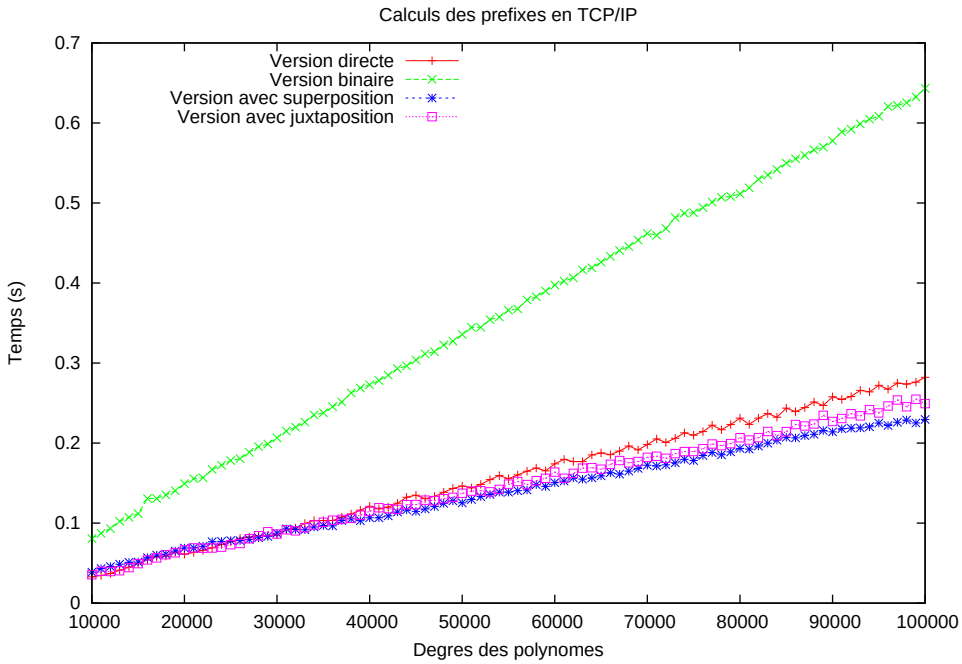


Figure 7.16 — Calculs des préfixes (suite)

7.A Preuves des théorèmes

7.A.1 Confluence forte de $\mapsto$

Lemme 37 (*Déterminisme des règles fonctionnelles*)

1. *Si* $e \overset{\varepsilon}{\mapsto} e^1, c^1$ *et* $e \overset{\varepsilon}{\mapsto} e^2, c^2$ *alors* $e^1 = e^2$ *et* $c^1 = c^2$ *;*
2. *Si* $e \overset{\delta}{\mapsto} e^1, c^1$ *et* $e \overset{\delta}{\mapsto} e^2, c^2$ *alors* $e^1 = e^2$ *et* $c^1 = c^2$.

Preuve. Par cas sur les règles. ■

Lemme 38 (*Déterminisme des règles globales*)

1. *Si* $\mathcal{E}, \mathbf{s}, e/c_g/\langle \mathbf{c} \rangle \underset{\bowtie_t}{\overset{\varepsilon}{\mapsto}} \mathcal{E}^1, \mathbf{s}^1, e^1/c_g \oplus c^1/\langle \mathbf{c} \rangle$ *et* $\mathcal{E}, \mathbf{s}, e/c_g/\langle \mathbf{c} \rangle \underset{\bowtie_t}{\overset{\varepsilon}{\mapsto}} \mathcal{E}^2, \mathbf{s}^2, e^2/c_g \oplus c^2/\langle \mathbf{c} \rangle$ *alors* $e^1 = e^2$, $c^1 = c^2$, $\mathcal{E}^1 = \mathcal{E}^2$ *et* $\mathbf{s}^1 = \mathbf{s}^2$ *;*
2. *Si* $\mathcal{E}, \mathbf{s}, e/c_g/\langle \mathbf{c} \rangle \underset{\bowtie_t}{\overset{\delta}{\mapsto}} \mathcal{E}^1, \mathbf{s}^1, e^1/c_g \oplus c^1/\langle \mathbf{c} \rangle$ *et* $\mathcal{E}, \mathbf{s}, e/c_g/\langle \mathbf{c} \rangle \underset{\bowtie_t}{\overset{\delta}{\mapsto}} \mathcal{E}^2, \mathbf{s}^2, e^2/c_g \oplus c^2/\langle \mathbf{c} \rangle$ *alors* $e^1 = e^2$, $c^1 = c^2$, $\mathcal{E}^1 = \mathcal{E}^2$ *et* $\mathbf{s}^1 = \mathbf{s}^2$.

Preuve. Par application du lemme 37 et par cas sur les règles des primitives et des opérations globales. ■

Lemme 39 (*Déterminisme des règles*)

1. *Si* $\mathcal{E}, \mathbf{s}, e/c_g/\langle c_0, \ldots, c_{p-1} \rangle \underset{t}{\overset{\bowtie}{\mapsto}} \mathcal{E}^1, \mathbf{s}^1, e^1/c_g^1/\langle c_0^1, \ldots, c_{p-1}^1 \rangle$
   et $\mathcal{E}, \mathbf{s}, e/c_g/\langle c_0, \ldots, c_{p-1} \rangle \underset{t}{\overset{\bowtie}{\mapsto}} \mathcal{E}^2, \mathbf{s}^2, e^2/c_g^2/\langle c_0^2, \ldots, c_{p-1}^2 \rangle$
   alors $e^1 = e^2$, $c_g^1 = c_g^2$, $\mathcal{E}^1 = \mathcal{E}^2$, $\mathbf{s}^1 = \mathbf{s}^2$ *et* $\forall i\ c_i^1 = c_i^2$ *;*
2. *Si* $e \overset{i}{\mapsto} e^1, c^1$ *et* $e \overset{i}{\mapsto} e^2, c^2$ *alors* $e^1 = e^2$ *et* $c^1 = c^2$.

Preuve. Par application du lemme 38 pour (1) et du lemme 37 pour (2). ■

Notons que les contextes des expressions sont identiques à ceux du chapitre 3. Nous utilisons donc les mêmes lemmes sur ces contextes.

Lemme 40 (*Déterminisme des contextes des super-threads*)

1. *Si* $E = \Gamma_{\mathbf{s}}^1[\!\langle e \rangle\!\rangle_t]$ *et* $E = \Gamma_{\mathbf{s}}^2[\!\langle e \rangle\!\rangle_t]$ *alors* $\Gamma_{\mathbf{s}}^1 = \Gamma_{\mathbf{s}}^2$ *;*
2. *Si* $E = \Gamma_{\mathbf{s}}^1[\!\langle e \rangle\!\rangle_t]$ *alors* $E \neq \mathcal{V}_{\mathbf{s}}$.

Preuve. Par construction de nos contextes. En effet, si un super-thread peut être réduit alors l'ensemble des super-threads ne peut pas être réduit par la règle de communication. ■

Lemme 41 (*Confluence forte*)

Si	$\mathcal{E}, \mathbf{s}, E, e/c_g/\langle c_0, \cdots, c_{p-1} \rangle \mapsto \mathcal{E}^1, \mathbf{s}^1, E^1, e^1/c_g^1/\langle c_0^1, \cdots, c_{p-1}^1 \rangle$
et	$\mathcal{E}, \mathbf{s}, E, e/c_g/\langle c_0, \cdots, c_{p-1} \rangle \mapsto \mathcal{E}^2, \mathbf{s}^2, E, e^2/c_g^2/\langle c_0^2, \cdots, c_{p-1}^2 \rangle$
alors il existe	$\mathcal{E}^1, \mathbf{s}^1, E^1, e^1/c_g^1/\langle c_0^1, \cdots, c_{p-1}^1 \rangle \mapsto \mathcal{E}^3, \mathbf{s}^3, E^3, e^3/c_g^3/\langle c_0^3, \cdots, c_{p-1}^3 \rangle$
et	$\mathcal{E}^2, \mathbf{s}^2, E^2, e^2/c_g^2/\langle c_0^2, \cdots, c_{p-1}^2 \rangle \mapsto \mathcal{E}^3, \mathbf{s}^3, E^3, e^3/c_g^3/\langle c_0^3, \cdots, c_{p-1}^3 \rangle$.

Preuve. Par le lemme 40, nous avons deux grand types de réductions. Celles qui effectuent les communications (règle (7.12)) et les règles de calculs asynchrones. La règle (7.12) de communication est clairement déterministique.

Par le lemme 40.1, nous savons que la règle de calcul asynchrone s'effectuera toujours dans le même super-thread. Par le lemme 9, nous avons deux types de réductions asynchrones distincts.

Si $\mapsto$ est une réduction globale, alors par le lemme 7 il n'existe qu'un contexte global et par le lemme 39, la réduction est déterministe. De même pour les règles 7.10 et 7.11.

Si $\mapsto$ est une réduction locale, alors nous avons $e = \Delta^i[e^i]$ et $e = \Delta^j[e^j]$. Nous avons alors deux cas :

1. Si $i = j$ alors par le lemme 8.1 il n'existe qu'un contexte et par le lemme 39, la réduction est déterministe ;
2. Si $i \neq j$, il est facile de constater (comme précédemment) que les règles peuvent donc s'entrelacer : les réductions interviennent dans deux composantes différentes d'un même vecteur et ces réductions sont déterministes.

Tous les types de réductions sont donc forteement confluentes. ■

7.A.2 Équivalence de $\mapsto$ et de $\rightharpoonup$

Lemme 42

Si $\forall i, j\ c_j^i \geq 0$ *alors* $\sum_{j=0}^{n} (\max_{i=0}^{p-1} \langle \ldots, c_i^j, \ldots \rangle) \geq \max_{i=0}^{p-1} \langle \ldots, \sum_{j=0}^{n} c_i^j, \ldots \rangle$

Preuve. Par induction sur n

- cas $n = 0$. On a : $\begin{cases} \sum_{j=0}^{n} (\max_{i=0}^{p-1} \langle \ldots, c_i^j, \ldots \rangle) = \sum_{j=0}^{0} (\max_{i=0}^{p-1} \langle \ldots, c_i^j, \ldots \rangle) = \max_{i=0}^{p-1} \langle \ldots, c_i^0, \ldots \rangle \\ \max_{i=0}^{p-1} \langle \ldots, \sum_{j=0}^{n} c_i^j, \ldots \rangle = \max_{i=0}^{p-1} \langle \ldots, \sum_{j=0}^{0} c_i^j, \ldots \rangle = \max_{i=0}^{p-1} \langle \ldots, c_i^0, \ldots \rangle \end{cases}$
- cas $n + 1$. On a :

$$\begin{aligned} & \sum_{j=0}^{n+1} (\max_{i=0}^{p-1} \langle \ldots, c_i^j, \ldots \rangle) \\ = \ & \max_{k=0}^{p-1} \langle \ldots, c_k^{n+1}, \ldots \rangle \oplus \sum_{j=0}^{n} (\max_{i=0}^{p-1} \langle \ldots, c_i^j, \ldots \rangle) \\ \geq \ & \max_{k=0}^{p-1} \langle \ldots, c_k^{n+1}, \ldots \rangle \oplus \max_{i=0}^{p-1} \langle \ldots, \sum_{j=0}^{n} c_i^j, \ldots \rangle \\ = \ & \max_{i=0}^{p-1} \langle \ldots, \max_{k=0}^{p-1} \langle \ldots, c_k^{n+1}, \ldots \rangle \oplus \sum_{j=0}^{n} c_i^j, \ldots \rangle \\ \geq \ & \max_{i=0}^{p-1} \langle \ldots, c_i^{n+1} \oplus \sum_{j=0}^{n} c_i^j, \ldots \rangle \\ = \ & \max_{i=0}^{p-1} \langle \ldots, \sum_{j=0}^{n+1} c_i^j, \ldots \rangle \end{aligned}$$

D'où le résultat. ■

Proposition 6 (*Équivalence*)

Si $e = \mathcal{T}_\bullet(e^p)$ *alors*

1. *si* $\{\}, n, \phi e[\bullet] \flat_1 / 0 / \langle 0, \ldots, 0 \rangle \overset{*}{\mapsto} \{\}, \mathbf{m}, \phi v \flat_1 / c_g / \langle c_0, \ldots, c_{p-1} \rangle$
   alors $e[\bullet]/0/\langle 0, \ldots, 0 \rangle \overset{*}{\rightharpoonup} v/c'_g/\langle c'_0, \ldots, c'_{p-1} \rangle$
   avec $c_g \oplus \max_{i=0}^{p-1}(c_i) \leq c'_g \oplus \max_{i=0}^{p-1}(c'_i)$
2. *si* $e[\bullet]/0/\langle 0, \ldots, 0 \rangle \overset{*}{\rightharpoonup} v/c'_g/\langle c'_0, \ldots, c'_{p-1} \rangle$
   alors $\{\}, 0, \phi e[\bullet] \flat_1 / 0 / \langle 0, \ldots, 0 \rangle \overset{*}{\mapsto} \{\}, \mathbf{m}, \phi v \flat_1 / c_g / \langle c_0, \ldots, c_{p-1} \rangle$
   avec $c_g \oplus \max_{i=0}^{p-1}(c_i) \leq c'_g \oplus \max_{i=0}^{p-1}(c'_i)$

Preuve. Les preuves se font par induction sur la longueur de $\mapsto$ et de $\rightharpoonup$. Ensuite, par cas sur ces réductions, il est facile de constater que les valeurs sont identiques et que les coûts dans les réductions de $\mapsto$ sont soit identiques, soit inférieurs (ou égales) dans le cas des communications (par le lemme 42). ■

8 Structures de données parallèles

Une version condensée de ce chapitre a été publiée dans l'article [W1].

Sommaire

COMME nous l'avons déjà fait remarquer, certains problèmes scientifiques nécessitent des performances que seules les machines parallèles peuvent offrir. Programmer ce genre d'architectures reste un travail difficile. La complexité des ordinateurs parallèles demande donc le développement de logiciels et de modèles de programmation afin d'aider les chercheurs (ou les ingénieurs) qui n'ont pas forcément les connaissances suffisantes pour le calcul parallèle et/ou qui n'ont pas le temps nécessaire pour écrire eux-mêmes du code efficace.

8.1 Introduction

Comme nous le faisions remarquer, les applications manipulent très fréquemment des structures de données. L'idée sous-jacente de ce chapitre est de remplacer l'utilisation de certaines structures de données séquentielles par leurs versions parallèles implantées dans un langage de programmation parallèle (ici, ce sera bien sûr BSML, mais un autre choix est envisageable) afin d'obtenir du code parallèle. On peut comparer cette idée à la définition de squelettes (patrons) de structures de données en BSML.

Le principal problème est donc de remplacer les structures initiales par des versions parallèles qui doivent rester efficaces surtout en présence d'un grand nombre de données. Dans les applications scientifiques, les

structures de données sont généralement une représentation du domaine physique, et donc une approximation du résultat final. La clé pour une bonne parallélisation est, ainsi, d'avoir un re-balancement des données sans détruire les propriétés de dépendance entre ces données : violer ces dépendances peut entraîner une sémantique incorrecte (par exemple, comment traiter un ensemble, dans le sens mathématique du terme, ayant des doublons ?) ou des calculs inutiles.

Ce chapitre traite de l'étude de cas d'implantation en BSML des structures de données classiques qui sont présentes dans la bibliothèque standard de OCaml. Le but est de faciliter le développement d'applications scientifiques effectuant du calcul symbolique sur des structures de données en utilisant les caractéristiques de haut niveau des langages fonctionnels. Pour cela, nos implantations utiliseront massivement le système de modules de OCaml, permettant de spécifier facilement une structure de données parallèle à partir de sa version séquentielle : la version parallèle sera ainsi indépendante de la version séquentielle utilisée. Ceci permettra une maintenance du code plus aisée. La complexité des implantations parallèles sera cachée par des interfaces de haut niveau que nous essaierons de rendre le plus proches possible de celles de OCaml, afin de simplifier leur emploi.

Un autre point important est l'application d'un calcul d'histogrammes pour le re-balancement des données (suivant un algorithme décrit dans [19]). Ce re-balancement pourra s'effectuer de manière automatique (ou, plus exactement, suivant des paramètres donnés par le programmeur) ou de manière directe, c'est-à-dire à l'endroit où le programmeur le souhaite (quand il l'estime nécessaire). Dans ce chapitre, nous donnerons aussi les résultats d'une première expérience de l'utilisation de ces structures dans une application scientifique et de son expérimentation sur un cluster. Nous verrons alors comment, à l'aide de ces structures parallèles de haut niveau, nous obtenons des gains de performance significatifs.

Ce chapitre est organisé comme suit : dans un premier temps, nous rappellerons les bases du système de modules de OCaml. Ensuite, nous décrirons l'implantation de plusieurs structures de données parallèles, puis nous donnerons les résultats de nos expériences d'une application scientifique. Enfin, nous terminerons avec quelques travaux relatifs.

8.2 Description des modules en OCaml

Dans cette section, nous rappelons brièvement les caractéristiques du système de modules d'OCaml pour le lecteur à qui cela n'est pas familier. Il s'agit d'un langage à part entière au dessus du langage de base d'OCaml (il est en réalité indépendant du langage de base [177]), remplissant uniquement des fonctions de génie logiciel : compilation séparée, structuration de l'espace de noms, encapsulation et généricité du code.

8.2.1 Définition des structures

Le langage des modules est un langage fonctionnel d'ordre supérieur fortement typé, dont les termes sont appelés *modules*. Les «briques» de base sont les *structures*, unités regroupant entre eux des types, des valeurs, des exceptions ou des définitions de modules. Par exemple, une implantation naïve des ensembles sous la forme de listes ordonnées pourrait être la suivante :

```
module Set =
 struct
  type α set = α list

  (* empty: α list *)
  let empty = []

  (* comp: α →α →int *)
  let comp = compare

  (* add: α →α list→α  list *)
  let rec add e s = match s with
     [] →[e]
   | hd::tl →match (comp e hd) with
                0 →s (* e est déjà dans s *)
              | x when x<0 →e :: s (* e est plus petit qu'un élément de l' ensemble s *)
```

```
            | x when x>0 →hd :: add e tl

  (* member: α →α list→bool *)
  let rec member e s = match s with
    [] →false
   | hd::tl →match (comp e hd) with
               0 →true (* x est inclu dans s *)
             | x when x<0 →false (* x est plus petit qu'un élément de l' ensemble s *)
             | x when x>0 →member e tl
 end
```

8.2.2 Définition des signatures

Les types des structures, appelés *signatures*, permettent de restreindre la visibilité des composantes d'une structure en lui adjoignant une *interface*[1] et de masquer la définition de certains types (on parle alors de *type abstrait*). Ainsi, une signature possible masquant l'implantation des ensembles serait :

```
module type SET =
 sig
  type α set
  val empty: α list
  val add: α →α list→α list
  val member: α →α list→bool
 end
```

Restreindre la structure Set avec la signature SET ainsi définie permet d'avoir un autre point de vue de la structure où la fonction comp n'est plus accessible (fonction utilitaire pour l'implantation et superfétatoire à l'utilisateur) et où la représentation des ensembles (c'est-à-dire comment ils ont été implantés) est cachée :

```
module Abstract_Set = (Set : SET)
```

Notons que l'on peut directement resteindre l'interface d'une structure durant sa définition :

```
module Set: SET = struct ... end
```

8.2.3 Abstraction des structures

Les fonctions de ce langage, appelées *foncteurs*, permettent d'écrire des modules paramétrés par d'autres modules et de les appliquer ensuite à des cas particuliers. L'apport des foncteurs en terme de génie logiciel est important lorsqu'il s'agit de paramétrer de manière cohérente un *ensemble* de types et de fonctions par un autre *ensemble* de types et de fonctions.

Par exemple, nous pouvons «fonctoriser» l'implantation des Set avec un module définissant le type des éléments de l'ensemble et comment comparer de tels éléments. Cette dernière fonction remplacera l'emploi de la fonction générique (et dans ce cas peu sûre) compare d'OCaml :

```
(* type de comparaison *)
type comparison = Less | Equal | Greater

(* module pour la comparaison des éléments *)
module type ORDERED_TYPE =
 sig
  type t
  val compare: t →t →comparison
 end

(* re−définition abstraite des ensembles *)
module type SET =
```

[1] Au sens de MODULA et non de JAVA.

```
 sig
  type elt (* type des éléments *)
  type t (* type des ensembles *)
  val empty: t
  val add: elt→t→t
  val member: elt→t→bool
 end

module Set (Elt: ORDERED_TYPE) : SET with type elt = Elt.t =
 struct
  type elt = Elt.t

  type t = elt list

  let empty = []

  let rec add e s = match s with
    [] →[e]
  | hd::tl →match Elt.compare e hd with
                Equal →s
              | Less →e :: s
              | Greater →hd :: add e tl

  let rec member e s = match s with
    [] →false
  | hd::tl →match Elt.compare e hd with
                Equal →true
              | Less →false
              | Greater →member e tl
 end
```

L'annotation **with type** permet d'unifier ici le type abstrait elt de la signature SET avec le type t de la signature ORDERED_TYPE. D'autre part, la signature ORDERED_TYPE exigée pour l'argument du foncteur ne contient que ce qui est nécessaire à l'implantation de l'algorithme. On pourra cependant appliquer ce foncteur à tout module dont la signature contient *au moins* ORDERED_TYPE, c'est-à-dire qui en est un *sous-type*. Enfin, il est possible de construire des agrégats de signatures ou de modules à l'aide de la construction **include**, qui peut être vue comme une inclusion textuelle.

La bibliothèque standard d'OCaml fournit de nombreuses structures de données. Chacune est définie par un module où sont décrites les opérations sur ces structures ainsi que, pour certaines, la complexité de telles opérations.

Les modules implantant les structures de données parallèles devront avoir les mêmes signatures que celles d'OCaml. La sémantique de haut niveau (décrivant le résultat de l'opérateur sur la structure) devra elle aussi être la même, afin de conserver la cohérence des résultats des programmes. Naturellement, les coûts seront différents, ainsi que le fonctionnement «bas niveau». Les signatures seront aussi dotées de quelques opérateurs permettant un meilleur emploi des structures parallèles. De cette manière, un programmeur pourra profiter pleinement des capacités de la machine parallèle en utilisant ces représentations parallèles des structures, et ceci sans avoir tous les inconvénients et difficultés d'un langage dédié à la programmation parallèle.

Pour l'instant, nous avons l'implantation de cinq de ces structures de données : Set (ensemble), Map (dictionnaire, parfois aussi appelé table d'association), Hashtable (table de hachage) , Queue (file, ou encore appelée FIFO, «first in, first out») et Stack (pile, également appelée LIFO, «last in, first out»). Dans les sections qui vont suivre, nous détaillerons ces implantations. Nous donnerons le coût BSP de chacune des opérations (ou squelettes de structures de données) et ceci de manière indépendante vis-à-vis de la complexité des opérations séquentielles. Nous notons $\mathcal{C}_t$ la complexité d'une opération séquentielle et $\mathcal{S}$ la taille en octets d'une donnée.

8.3 Implantation des dictionnaires

8.3.1 Définition

Le module des dictionnaires parallèles est un foncteur Make qui construit une implantation d'un dictionnaire parallèle polymorphe à partir :

1. D'un module contenant un type pour les clés, une fonction d'ordre total sur celles-ci et une fonction de hachage (telle que deux éléments égaux ont la même clé de hachage) ;

2. D'un module définissant la stratégie de re-balancement (celle-ci sera plus détaillée dans la section 8.4) ;

3. D'un dernier foncteur permettant la construction d'un dictionnaire séquentiel.

En OCaml, nous obtenons :

```
module Make (Ord : OrderedType)(Bal:BALANCE)
           (MakeLocMap:functor(Ord:OrderedType) →Map.S with type key=Ord.t)
    : S with type key = Ord.t and type α seq_t = α MakeLocMap(Ord).t
```

avec les signatures suivantes :

```
module type OrderedType =
 sig
  type t
  val compare:t→t→int
  val hash:t→int
 end
```

```
module type S = sig
 type key (* type des clés *)
 type α t (* type d'un dictionnaire parallèle *)
 type α seq_t (* type d'un dictionnaire séquentiel *)
 (* opérateurs classiques *)
 (* opérateurs parallèles *)
end
```

Nous définissons un dictionnaire parallèle comme une paire comportant un vecteur parallèle de dictionnaires séquentiels et une paire comportant un booléen et un entier :

```
module Make (Ord : OrderedType)(Bal:BALANCE)
           (MakeLocMap:functor(Ord:OrderedType) →Map.S with type key=Ord.t) =
 struct
  (* LocMap est le module des dictionnaires séquentiels *)
  module LocMap = MakeLocMap(Ord)
  type key = Ord.t
  type α t = α LocMap.t par * (int * bool)
  type α seq_t = α LocMap.t
  (* opérations sur α t, le dictionnaire parallèle *)
 end
```

Le booléen permet de savoir si le dictionnaire parallèle a été (oui ou non) re-balancé et l'entier définit le nombre d'opérations qui ont modifié le dictionnaire parallèle. Cet entier sera utilisé pour le re-balancement. Nous l'ignorons pour l'instant et nous nous reférons à la section 8.4 pour de plus amples informations.

Un dictionnaire parallèle enregistre un élément (c'est-à-dire une liaison entre une clé et une donnée) dans un unique dictionnaire séquentiel, et donc dans un seul processeur à la fois. Toutes les opérations se feront en supposant et en maintenant le fait qu'une liaison ne peut être présente que sur un seul processeur. Ce sera donc l'invariant de nos opérations. Le dictionnaire parallèle emploie le module des dictionnaires séquentiels qui a été donné en paramètre via un foncteur. Nous pouvons ainsi utiliser l'implantation OCaml des dictionnaires (arbres binaires équilibrés), mais nous pouvons tout aussi bien recourir à n'importe quelle autre implantation, comme celles qui ont été certifiées en **Coq** dans [106] (nous y trouvons des implantations certifiées se servant de listes triées, d'arbres binaires équilibrés ou bien encore d'arbres dits rouge-noir). Maintenant, nous allons décrire l'implantation générique des opérations sur ces dictionnaires parallèles. Celles-ci seront purement fonctionnelles, c'est-à-dire sans effets de bord. Par la suite, nous notons m_i pour un dictionnaire séquentiel au processeur i.

8.3.2 Implantation des opérations classiques

Nous avons besoin avant toute chose, de créer un dictionnaire parallèle. Celui-ci sera initialement vide, non re-balancé et non modifié :

```
(* empty: unit→t *)
let empty () = (replicate LocMap.empty,(0,true))
```

ce qui peut se représenter par le schéma suivant : $\boxed{\{\}\ |\ \cdots\ |\ \{\}}$.

La première opération importante consiste en l'ajout d'une nouvelle liaison au dictionnaire parallèle. L'opération add k e m retourne un dictionnaire contenant les mêmes liaisons que m, mais avec une nouvelle liaison entre la clé k et la valeur e. Si la clé k était déjà présente dans m, celle-ci est supprimée dans le résultat. Pour ajouter cette nouvelle liaison, nous choisissons en quel processeur celle-ci sera séquentiellement enregistrée (noté $m \cup \{k \to e\}$). Pour cela, nous utilisons la fonction de hachage donnée en paramètre du module, et qui est supposée être déterministe, c'est-à-dire retournant toujours le même entier quand elle est appliquée à la même valeur. Notons que si le dictionnaire parallèle a été re-balancé, les éléments ne sont plus forcement à leur place «originelle» (c'est-à-dire définie par la fonction de hachage). Nous devons alors supprimer la clé k sur les autres processeurs afin d'éviter l'apparition de doublons (invariant de la structure) :

```
(* destination: α →int and add: key→α →α t→α t *)
let dest x = (Ord.hash x) mod (bsp_p())

(* add: key→α →α t→α t *)
let add k e pmap = let (vect_map',(step',nbal')) = rebalance_if_needed pmap in
 if nbal' then ((applyat (dest k) (LocMap.add k e) id vect_pmap'),(step'+1,nbal'))
 else ((applyat (dest k) (LocMap.add k e) (LocMap.remove k) vect_pmap'),(step'+1,nbal'))
```

Ce qui correspond au schéma suivant :

add k e $\boxed{m_0\ |\ \cdots\ |\ m_i\ |\ \cdots\ |\ m_{p-1}} \Rightarrow$

$$\begin{cases} \boxed{m_0\backslash\{k\}\ |\ \cdots\ |\ m_i \cup \{k \to e\}\ |\ \cdots\ |\ m_{p-1}\backslash\{k\}} & \text{si le dictionnaire a été re-balancé} \\ \boxed{m_0\ |\ \cdots\ |\ m_i \cup \{k \to e\}\ |\ \cdots\ |\ m_{p-1}} & \text{autrement} \end{cases}$$

et le coût BSP de la fonction add est donc :

$$\begin{cases} \max(\max_{(j=0\ j\neq i)}^{\mathbf{P}-1} \mathcal{C}_{\mathbf{t}}(m_j \backslash \{k\}), \mathcal{C}_{\mathbf{t}}(m_i \cap \{k \to e\})) & \text{si le dictionnaire a été re-balancé} \\ \mathcal{C}_{\mathbf{t}}(m_i \cap \{k \to e\}) & \text{autrement} \end{cases}$$

Notons qu'avec des arbres binaires équilibrés (implantation de base de OCaml), le coût BSP est alors :

$$\begin{cases} \max_{j=0}^{\mathbf{P}-1}(\lg_2 |m_j|) & \text{si le dictionnaire a été re-balancé} \\ \lg_2 |m_i| & \text{autrement} \end{cases}$$

La seconde opération est de chercher la valeur associée à une clé. find k m retourne cette valeur pour une clé k du dictionnaire m. Si cette clé n'est pas présente, une exception est levée. La recherche de cette liaison dans un dictionnaire parallèle est effectuée de manière asynchrone par le processeur qui pourrait contenir la clé (suivant la fonction de hachage) ou en chaque processeur dans le cas d'un dictionnaire parallèle qui a été re-balancé. La recherche dans un dictionnaire séquentiel est notée $k \in_? m_i$. Le résultat de cette recherche est ensuite diffusé aux autres processeurs. Dans le cas d'un dictionnaire re-balancé, les résultats des recherches sont échangés entre tous les processeurs :

```
(* find: key→α t→α  *)
let find k pmap = let (pmap',_,nbal') = rebalance_if_needed pmap in
 if nbal' then let root = (dest k) in
  let loc_find= (applyat root (fun t →try Some (LocMap.find k t) with Not_found →None)
                              (fun _ →None) pmap') in
  let res = rpl_bcast root loc_find in match res with None →raise Not_found | Some x →x
 else parfun_total(fun t→try Some(LocMap.find k t) with Not_found→None)choose_l pmap'
```

avec choose_l:α option list$\rightarrow\alpha$ fonction qui sélectionne le premier (Some x) de la liste et lève Not_found sinon. parfun_total: ($\alpha \rightarrow\beta$)$\rightarrow$(β list$\rightarrow\gamma$)$\rightarrow\alpha$ **par**$\rightarrow\gamma$ permet d'appliquer une fonction à chaque composante d'un vecteur parallèle, de faire un échange total de ces résultats puis d'appliquer une fonction sur la liste de ces résultats. Ce code correspond au schéma suivant :

find k $\boxed{m_0 \mid \cdots \mid m_{p-1}} \Rightarrow$

$$\begin{cases} \boxed{k \in_? m_0 \mid \cdots \mid k \in_? m_{p-1}} & \Rightarrow b_? \quad \text{si le dictionnaire a été re-balancé} \\ \boxed{\cdots \mid k \in_? m_i \mid \cdots} & \Rightarrow b_i \quad \text{autrement} \end{cases}$$

avec $\forall j\ b_j = k \in_? m_j$. Le coût BSP d'une telle recherche est donc le temps maximal pour effectuer localement et séquentiellement cette recherche puis de la diffuser :

$$\begin{cases} \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(e \in_? m_j)) + (\mathbf{p}-1) \times \mathcal{S}(v_k) \times \mathbf{g} + \mathbf{l} & \text{si le dictionnaire a été re-balancé} \\ \mathcal{C}_{\mathbf{t}}(e \in_? m_i) + (\mathbf{p}-1) \times \mathcal{S}(v_k) \times \mathbf{g} + \mathbf{l} & \text{autrement} \end{cases}$$

avec v_k la valeur liée par la clé k. Notons qu'avec l'implantation OCaml des dictionnaires, la recherche se fait aussi en temps logarithmique.

Notons que l'on pourrait éventuellement améliorer le code en utilisant une diffusion en 2 phases. Mais comme on ne connaît pas la taille de la donnée, il faudrait préalablement diffuser cette taille ce qui coûterait une barrière supplémentaire. Pour éviter ce test, il faudrait un paramètre supplémentaire indiquant si il est préférable de toujour diffuser directement ou non (faire le test puis de choisir la diffusion la plus appropriée). En pratique, les structures de données parallèles ne sont vraiment intéressantes qu'en présence d'un grand nombre de données. Celles-ci sont alors de taille «raisonnable» et la différence entre la diffusion directe ou en 2 phases est alors difficilement décelable. De plus, l'ajout de cet énième paramètre n'est pas naturel.

La troisième et dernière opération importante est la suppresion d'une clé, c'est-à-dire supprimer une liaison entre une clé et sa valeur. Ceci peut être fait de manière asynchrone, en supprimant localement la clé à tous les processeurs si le dictionnaire a été re-balancé. Dans le cas contraire, on ne supprime la clé qu'au processeur originel :

```
let remove k pmap = let (pmap',step',nbal') = rebalance_if_needed pmap in
 if nbal' then (applyat (dest k) (LocMap.remove k) id pmap',step'+1,nbal')
  else (parfun (LocMap.remove k) pmap',step'+1,nbal')
```

Ce qui correspond au schéma suivant :

remove k $\boxed{m_0 \mid \cdots \mid m_i \mid \cdots \mid m_{p-1}} \Rightarrow$

$$\begin{cases} \boxed{m_0 \backslash \{k\} \mid \cdots \mid m_{p-1} \backslash \{k\}} & \text{si le dictionnaire a été re-balancé} \\ \boxed{\cdots \mid m_i \backslash \{k\} \mid \cdots} & \text{autrement} \end{cases}$$

et le coût BSP de remove est donc :

$$\begin{cases} \max_{j=0}^{\mathbf{p}-1} \mathcal{C}_{\mathbf{t}}(m_j \backslash \{k\}) & \text{si le dictionnaire a été re-balancé} \\ \mathcal{C}_{\mathbf{t}}(m_j \backslash \{k\}) & \text{autrement} \end{cases}$$

Notons qu'avec des arbres binaires équilibrés, la suppression s'effectue, là encore, en temps logarithmique.

8.3.3 Itérer sur un dictionnaire

Dans cette section, nous allons décrire un nouvel opérateur d'itérations sur les dictionnaires, permettant de profiter pleinement des capacités de la machine parallèle. Cet opérateur a été ajouté à l'interface OCaml classique pour la raison suivante : la sémantique de haut niveau du fold classique spécifie que les éléments sont réduits dans un ordre croissant (suivant la fonction de comparaison des clé du dictionnaire). Spécifiant un ordre de calcul strict, il est tout bonnement impossible de paralléliser cet opérateur : un calcul parallèle implique un ordre indéterminé des calculs. Ceci entraîne, afin exécuter correctement cet opérateur, la transformation préalable du dictionnaire parallèle en un dictionnaire séquentiel répliqué en chaque composante de la machine parallèle. Cette transformation implique naturellement un échange total des éléments, et est donc fort coûteuse. Par exemple, il est possible de calculer le cardinal d'un dictionnaire (le nombre de couples clé/valeur) de la manière suivante :

```
let cardinal pmap=ParMap.fold (fun key value i→i+1) 0 pmap
```

Ce code réduit tout d'abord le dictionnaire parallèle pmap en un dictionnaire séquentiel puis il itère (**fun** key value i→i+1) sur tous les éléments du dictionnaire afin de calculer le cardinal.

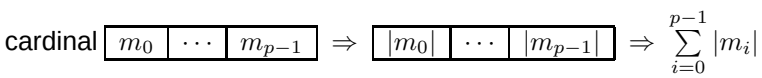
Pour éviter la transformation susmentionnée et pouvoir itérer en parallèle, nous avons ajouté un itérateur asynchrone qui permet une réduction de tous les éléments du dictionnaire parallèle, et ceci dans un ordre non-spécifié :

```
(* async_fold: (key→α →β →β )→α t→β →β par *)
let async_fold f pmap e = let pmap'= rebalance_if_needed pmap in
 parfun (fun s →LocMap.fold f s e) (fst pmap')
```

Le coût BSP de ce nouvel opérateur (en supposant que op a un coût constant c_{op}) est $c_{op} \times \max_{i=0}^{p-1} |m_i|$, c'est-à-dire le temps de calcul maximal d'un processeur pour réduire tous ces éléments locaux. En utilisant cet opérateur, nous pouvons réécrire la fonction de cardinalité en itérant en parallèle sur les éléments :

```
let cardinal pmap=List.fold_left (+) 0 (rpl_total (ParMap.async_fold (fun _ _ i→i+1) pmap 0))
```

Cette fonction correspond au schéma suivant :



avec $\max_{i=0}^{\mathbf{p}-1} |m_i| + (\mathbf{p}-1) \times \mathbf{g} + \mathbf{l} + \mathbf{p}$ comme coût BSP. Notons que nous avons ajouté, pour la même raison, un opérateur map asynchrone :

```
let async_map f pmap =
 let (vect_map',(step',nbal')) = rebalance_if_needed pmap in
  (parfun (LocMap.map f) vect_map',(step',nbal'))
```

8.4 Rebalancement de la localisation des données

Comme nous l'avons vu dans la section précédente, l'emplacement des données sur les processeurs, après une malencontreuse série d'opérations, peut être «imprévisible» et surtout discriminatoire : certains processeurs peuvent se voir attribuer le stockage de toutes les données. Ce phénomème peut dégrader les performances, en ralentissant une application parallèle à la vitesse du processeur le plus chargé en données. Avec les communications et les temps de synchronisation, un programme parallèle peut devenir moins efficace que sa version séquentielle. Un re-balancement des données entre les processeurs est alors nécessaire. Dans [19], une stratégie générale, utilisant deux phases de communications BSP, a été introduite pour permettre un re-balancement efficace. La première phase consiste en un échange total du nombre de données présentes en chaque processeur, donnant ce que l'on appelle couramment un *histogramme* de la structure. Ensuite, suivant ces tailles, chaque processeur sélectionne les données à envoyer aux autres processeurs. Ces données sont alors envoyées et supprimées localement. Enfin, les données reçues sont incorporées aux données locales.

Pour re-balancer sa structure de données, le programmeur peut demander explicitement ce re-balancement en appelant dans son code, une fonction dédiée à cette tache (re-balancement «à la main»), ou bien, s'il n'est pas sûr, avoir recours à un re-balancement «automatique», dont la stratégie est définie *via* un module donné en paramètre à la structure, et ayant la signature suivante :

```
module type BALANCE=sig
 val need_balance: unit→bool
 val unbalanced: unit→int
 val max_op_without_balance: unit→int end
```

need_balance définit si l'on veut ou non un re-balancement. max_op_without_balance est le nombre maximal d'opérations sur la structure avant d'éventuellement la re-balancer. On utilise pour cela l'entier mentionné dans la section précédente, qui précise le nombre d'opérations qui ont modifié la structure. unbalance fixe la différence maximale du nombre de données entre les structures locales.

Le code BSML pour le re-balancement a été tiré de [135]. Celui-ci ne fonctionnant alors que pour les ensembles. Nous l'avons modifié afin de le rendre générique. Ceci nous permettra de mettre ce re-balacement

```
(* union_set_list: LocSet.t→LocSet.elt list→LocSet.t *)
let union_set_list s l =
 let sl=List.fold_left (fun s e →Local_Set.add e s) Local_Set.empty l in
  Local_Set.union s sl

(* from_sets_to_parset: LocSet.t par→LocSet.t par *)
let from_sets_to_parset vect_sets =
 let sets_to_send = parfun (fun set →LocSet.fold (fun elt map →
      let pid=(destination elt) in
      let l_add=(try MapSet.find pid map with Not_found →[]) in
       MapSet.add pid (elt::l_add) map) set MapSet.empty) vect_sets in
 let to_send =parfun (fun map pid →try Some (MapSet.find pid map) with Not_found →None) sets_to_send in
 let delivery=put to_send in
  parfun (fun f →let list_sets = map_some_split f (procs ())
                 in List.fold_left union_set_list LocSet.empty list_sets) delivery
```

Figure 8.1 — Réorganisation d'un vecteur parallèle d'ensembles

en œuvre pour d'autres types de structures de données (notamment les ensembles qui seront décrits dans la prochaine section). Nous utilisons la fonction polymorphe (code à la figure 8.9, en fin du chapitre) suivante :

val rebalance_if_needed: ($\alpha \to \beta$ **par**)→(int→$\beta \to \gamma$ list $*$ β)→($\beta \to \beta \to \beta$)→(γ list→β)
(β **par**→α)→($\alpha \to$int **par**)→($\alpha \to$bool)→int→$\alpha \to \alpha$

où les paramètres de la fonction sont les suivants :

1. Une fonction qui transforme une structure de données parallèle en un vecteur parallèle de structures locales ;
2. Une fonction qui prend n éléments d'une structure et qui retourne une paire contenant la structure sans ces n éléments et une liste de ces n éléments extraits ;
3. Une fonction qui unifie deux structures de données locales ;
4. Une fonction qui transforme une liste d'éléments en un structure locale ;
5. Une fonction qui effectue le travail inverse du premier argument ;
6. Une fonction permettant de calculer l'histogramme de la structure parallèle ;
7. Une fonction de test pour savoir si le re-balancement est nécessaire ou non (suivant l'histogramme) ;
8. La différence maximale de taille entre 2 structures locales ;
9. La structure de données parallèle elle-même.

8.5 Implantation des ensembles

Nous appliquons les mêmes techniques de programmation modulaire pour l'implantation BSML des ensembles : chaque composante de la machine parallèle contient un sous-ensemble de l'ensemble originel. Chaque sous-ensemble est représenté par un ensemble séquentiel défini à l'aide d'un foncteur. Notons que l'implantation des ensembles de la bibliothèque standard d'OCaml utilise des arbres binaires re-balancés, mais toute autre implantation peut convenir (notamment les implantations certifiées décrites dans [106]).

Les opérateurs comme add, remove, mem, is_empty, async_fold etc. sont implantés d'une manière similaire à ceux des dictionnaires parallèles. Mais la structure des ensembles fournit en plus, des opérateurs binaires (c'est-à-dire utilisant deux ensembles) : l'union (union), l'intersection (inter) et la différence (diff) de deux ensembles. Nous décrivons dans les sections suivantes, ces opérateurs.

8.5.1 Transformations d'un vecteur parallèle d'ensembles

Pour commencer, nous avons besoin d'une fonction qui transforme un vecteur parallèle d'ensembles en un autre vecteur parallèle d'ensembles en y éliminant les doublons. Un algorithme naïf effectuerait un échange total des ces ensembles locaux pour ensuite faire leur union, ce qui supprimerait automatiquement les doublons. Un meilleur algorithme, minimisant les communications, réorganise le vecteur d'ensembles en envoyant chaque élément sur le processeur «indiqué» par la fonction de hachage (envoyer «à destination»). Les doublons sont ensuite éliminés par les processeurs avec l'union des ensembles reçus. La figure 8.1 donne le code d'une telle fonction, où MapSet est un dictionnaire de listes d'éléments (où les clés sont les pids des processeurs) et map_some_split:($\alpha \rightarrow \beta$ option)$\rightarrow\alpha$ list$\rightarrow\beta$ list applique une fonction f sur chaque élément de la liste l et si (f x)=(Some a) alors a est conservé dans la liste retournée. En utilisant la superposition parallèle, nous pouvons réorganiser 2, 3 etc. vecteurs parallèles d'ensembles en une et unique super-étape. Sans la superposition, 2 ou 3 ou etc. super-étapes seraient nécessaires. Par exemple :

```
(* distributed_two_sets: LocSet.t par→LocSet.t par→LocSet.t par * LocSet.t par *)
let distributed_two_sets vect_sets1 vect_sets2 =
  super(fun ()→from_sets_to_parset vect_sets1) (fun ()→from_sets_to_parset vect_sets2)
```

8.5.2 Définitions de base

Dans les sections qui vont suivre, nous allons prouver le bien-fondé de nos méthodes. Pour cela, nous avons besoin de quelques définitions et propriétés de base. La fonction de destination h a les propriétés suivantes :

$$\begin{array}{ll} \forall x \in \mathcal{D} & 0 \leq h(x) < p \\ \forall x, y \in \mathcal{D} & x = y \Rightarrow h(x) = h(y) \end{array}$$

où D est le domaine des éléments de nos ensembles. Un ensemble parallèle S est l'union de p ensembles (un par processeur). Nous avons donc l'invariant suivant : $S = \bigcup_{i=0}^{p-1} s_i$ tel que si $\forall i, j\ i \neq j$ alors $\forall x \in s_i \Rightarrow x \notin s_j$ (pas un même élément sur deux processeurs différents).

Quand l'ensemble parallèle S n'aura pas été re-balancé, c'est-à-dire que tous les éléments sont «à destination», nous aurons l'hypothèse suivante : $\forall x \in s_i \Rightarrow h(x) = i$.

Pour prouver le bien-fondé de nos méthodes, nous prouverons que l'ensemble résultant des calculs vérifie bien l'invariant. Nous noterons S^r cet ensemble et S l'ensemble attendu des calculs (la spécification). Pour prouver $S \equiv S'$, il nous suffira de prouver :

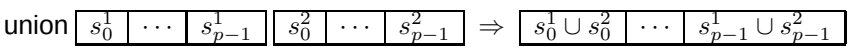
1. $\forall x \in S \Rightarrow x \in S^r$
2. $\forall x \in S^r \Rightarrow x \in S$
3. Si $S^r = \bigcup_{k=0}^{p-1} s_k$, alors $\forall i, j\ i \neq j$ et $\forall x \in s_i$ alors $x \notin s_j$

Les deux premières conditions sont celles traditionelles de l'équivalence ensembliste. La dernière propriété à prouver, est l'invariant des ensembles parallèles : pas de doublons, c'est-à-dire pas un élément sur deux processeurs différents.

8.5.3 Union de deux ensembles

Pour calculer l'union de deux ensembles parallèles, nous avons trois cas. Dans le premier cas, les deux ensembles parallèles n'ont pas été re-balancés et les éléments sont tous «à destination». Dans le deuxième cas, un des ensembles parallèles a été re-balancé. Dans le troisième, les deux ensembles ont été re-balancés. Par conséquent, les éléments peuvent être situés sur n'importe quel processeur. La figure 8.2 donne le code BSML de l'union de deux ensembles parallèles. Notons que l'ensemble parallèle résultant sera toujours non re-balancé, c'est-à-dire, avec tous les éléments «à destination».

Premier cas. Il suffit de calculer de manière asynchrone les unions locales, ce qui correspond au schéma suivant :



```
(* union: t→t→t *)
let union (psets1,(step1,nbal1)) (psets2,(step2,nbal2)) = match (nbal1, nbal2) with
    (true,true) (* deux ensembles non re−balancés *) →(parfun2 LocSet.union psets1 psets2,(step1+step2,true))
  | (true,false) (* le second a été re−balancé *) →
      let diff=parfun2 LocSet.diff psets2 psets1 in
      let diff'= from_sets_to_parset diff in
       (parfun2 LocSet.union psets1 diff',(step1+step2,true))
  | (false,true) (* le premier a été re−balancé *) →
      let diff=parfun2 LocSet.diff psets1 psets2 in
      let diff'= from_sets_to_parset diff in
       (parfun2 LocSet.union psets2 diff',(step1+step2,true))
  | (false,false) (* les deux ensembles ont été re−balancé *) →
       let redundant_sets = parfun2 LocSet.union psets1 psets2 in
        ((from_sets_to_parset redundant_sets),(step1+step2,true))
```

Figure 8.2 — Union de deux ensembles parallèles

Le coût BSP[2] est donc le temps maximal utilisé par un processeur pour faire l'union de ces 2 ensembles : $\max_{i=0}^{p-1} \mathcal{C}_{\mathbf{t}}(s_i^1 \cup s_i^2)$.

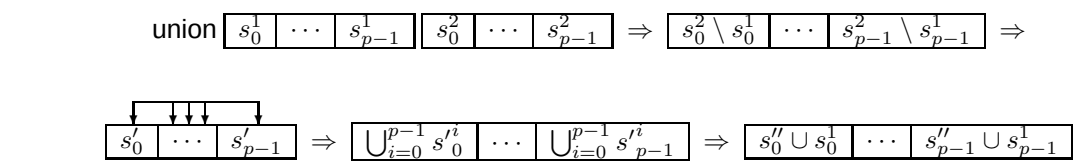
Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s_i^1$ et $S^2 = \bigcup_{i=0}^{p-1} s_i^2$ tels que S^1 et S^2 n'ont pas été re-balancés. Nous avons par définition $S = S^1 \cup S^2$ et donc $\forall x \in S^1$ ou $x \in S^2 \iff x \in S$. L'ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1} s_i^1 \cup s_i^2$.

Lemme 43

$S \equiv S'$

Preuve. Voir en section 8.A.1 de l'annexe de ce chapitre. ■

Deuxième cas. Nous sommes dans le cas où l'un des ensembles parallèles a été re-balancé. Supposons que ce soit le deuxième ensemble (le fonctionnement du cas dual est identique). Les éléments peuvent avoir été distribués sur n'importe quel processeur. Pour pouvoir pratiquer l'union, il nous faut alors remettre ces éléments «à destination». Pour cela, nous calculons la différence locale, éliminant ainsi les éléments déjà présents dans le premier ensemble parallèle. Puis nous redistribuons ces ensembles et nous pouvons ensuite finir par une union locale. Ceci correspond au schéma suivant :



avec $\forall j\ s'_j = s_j^2 \setminus s_j^1$ et $s''_j = \bigcup_{i=0}^{p-1} {s'}_j^i$ et ${s'}_j^i$ = ensemble des éléments envoyés par le processeur i au processeur j à partir des l'ensembles s'_j. Le coût BSP est alors :

$$\max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s_j^2 \setminus s_j^1)) + (\max_{j=0}^{\mathbf{p}-1}\mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1} {s'}_j^i)) \times \mathbf{g} + \mathbf{l} + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1} {s'}_j^i \cup s_j^1))$$

Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s_i^1$ et $S^2 = \bigcup_{i=0}^{p-1} s_i^2$ tel que S^1 n'a pas été re-balancé. Nous avons, par définition, $S = S^1 \cup S^2$. L' ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1} s_i^1 \cup s''_i$.

Lemme 44

$S \equiv S'$

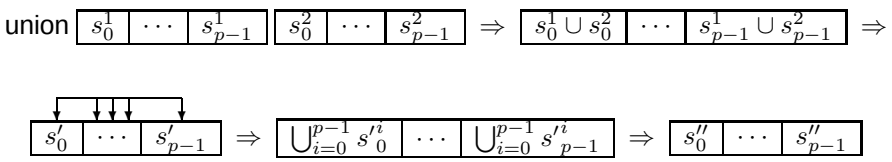
Preuve. Voir en section 8.A.1 de l'annexe de ce chapitre. ■

[2] Au moment de l'écriture de ce manuscrit, les complexités de $\cup, \cap, \setminus$ en OCaml ne sont pas connues. C'est un travail en cours. La complexité semblerait être en $N \ln(N)$ avec N le cardinal de l'ensemble résultant de l'opération. Mais l'implantation parallèle étant indépendante de l'implantation séquentielle, les coûts BSP seront donnés indépendamment des complexités des opérations séquentielles.

```
(* inter: t→t→t *)
let inter (psets1,(step1,nbal1)) (psets2,(step2,nbal2))= match (nbal1, nbal2) with
    (true,true) (* deux ensembles non−rebalancés *) →(parfun2 LocSet.inter psets1 psets2,(step1+step2,true))
 | (true,false) (* second a été re−balancé *) →
     let diff=parfun2 LocSet.diff psets2 psets1 in
     let inter=parfun2 LocSet.inter psets2 psets1 in
     let diff'= from_sets_to_parset diff in
      (parfun2 LocSet.union inter (parfun2 LocSet.inter psets1 diff'),(step1+step2,true))
 | (false,true) (* premier a été re−balancé *) →
     let diff=LocSet.diff psets1 psets2 in
     let inter=LocSet.inter psets2 psets1 in
     let diff'= from_sets_to_sets_without diff in
      (parfun2 LocSet.union inter (parfun2 LocSet.inter psets2 diff'),(step1+step2,true))
 | (false,false) (* deux ensembles re−balancés *) →
     let inter = parfun2 LocSet.inter psets1 psets2 in
     let diff1 = parfun2 LocSet.diff psets1 inter
     and diff2 = parfun2 LocSet.diff psets2 inter in
      let dist = distributed_three_sets diff1 diff2 inter in
      let new_set = parfun3 (fun a b c→LocSet.union c (LocSet.inter a b)) (fst (snd dist)) (snd (snd dist)) (fst dist) in
       (new_set, (step1+step2,true))
```

Figure 8.3 — Intersection de deux ensembles parallèles

Troisième cas. Dans ce dernier cas, les deux ensembles parallèles ont été re-balancés. Les éléments des deux ensembles peuvent avoir été distribués sur n'importe quel processeur. Pour calculer l'union, nous faisons tout d'abord des unions locales en chaque processeur. Puis nous redistribuons les éléments (remettre «à destination») afin d'éliminer les éventuels doublons. Ceci correspond au schéma suivant :



avec $\forall j\ s'_j = s_j^1 \cup s_j^2$ et $s''_j = \bigcup_{i=0}^{p-1} s'^i_j$ et s'^i_j = l'ensemble des éléments envoyés par le processeur i au processeur j à partir des ensembles s'_j. Le coût BSP est donc :

$$\max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s_j^1 \cup s_j^2)) + (\max_{j=0}^{\mathbf{p}-1}\mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1} s'^i_j)) \times \mathbf{g} + \mathbf{l} + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1} s'^i_j))$$

Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s_i^1$ et $S^2 = \bigcup_{i=0}^{p-1} s_i^2$. Nous avons, par définition, $S = S^1 \cup S^2$. L'ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1} s''_i$.

Lemme 45

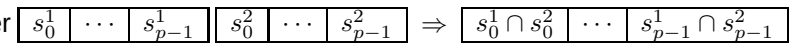
$S \equiv S'$

Preuve . Voir en section 8.A.1 de l'annexe de ce chapitre. ■

8.5.4 Intersection de deux ensembles

Pour calculer l'intersection de deux ensembles parallèles, nous avons trois cas. Dans le premier cas, les deux ensembles parallèles n'ont pas été re-balancés et les éléments sont tous «à destination». Dans le deuxième cas, un des ensembles parallèles a été re-balancé. Dans le troisième cas, les deux ensembles ont été re-balancés. Par conséquent, les éléments ont pu être distribués sur n'importe quel processeur. La figure 8.3 donne le code BSML de l'intersection de deux ensembles parallèles. Notons que l'ensemble parallèle résultant sera toujours non re-balancé, c'est-à-dire avec tous les éléments «à destination».

Premier cas. Il suffit de calculer de manière asynchrone les intersections locales, ce qui correspond au schéma suivant :



Le coût BSP est donc le temps maximal utilisé par un processeur pour faire l'intersection de ces 2 ensembles : $\max_{i=0}^{p-1} \mathcal{C}_{\mathbf{t}}(s_i^1 \cap s_i^2)$.

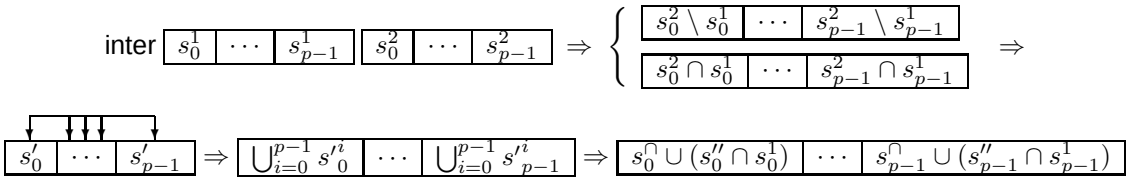
Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s_i^1$ et $S^2 = \bigcup_{i=0}^{p-1} s_i^2$ tels que S^1 et S^2 n'ont pas été re-balancés. Nous avons, par définition, $S = S^1 \cap S^2$ et donc $\forall x \in S^1$ et $x \in S^2 \iff x \in S$. L'ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1} s_i^1 \cap s_i^2$.

Lemme 46

$S \equiv S'$

Preuve. Voir en section 8.A.2 de l'annexe de ce chapitre. ■

Deuxième cas. Nous sommes dans le cas où l'un des ensembles parallèles a été re-balancé. Supposons que ce soit le deuxième ensemble (le fonctionnement du cas dual est identique). Ses éléments peuvent avoir été distribués sur n'importe quel processeur. Pour pouvoir pratiquer l'intersection, il nous faut alors remettre ces éléments «à destination». Pour cela, nous calculons la différence et l'intersection locales, éliminant ainsi les éléments déjà présents dans le premier ensemble parallèle. Puis nous redistribuons ces ensembles (non «à destination»). Cela se fait donc sans conserver les éléments qui étaient déjà «à destination». En effet, ceux-ci sont déjà présents dans l'intersection locale. Pour finir, nous pratiquons une intersection locale entre les éléments échangés et nous faisons une union avec les éléments des intersections locales (éléments qui étaient tous «à destinations»). Ceci correspond au schéma suivant :



avec $\forall j\ s'_j = s_j^2 \setminus s_j^1$, $s_i^\cap = s_i^1 \cap s_i^2$ et $s''_j = \bigcup_{i=0}^{p-1} s'^i_j$ et s'^i_j = ensemble des éléments envoyés par le processeur i au processeur j à partir des ensembles s'_j. Le coût BSP est :

$$\max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s_j^2 \setminus s_j^1)) + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s_j^2 \cap s_j^1)) + (\max_{j=0}^{\mathbf{p}-1}\mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1} s'^i_j)) \times \mathbf{g} + \mathbf{l} + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1} -s'^i_j \cap s_j^1) \cup s_j^\cap))$$

Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s_i^1$ et $S^2 = \bigcup_{i=0}^{p-1} s_i^2$ tel que S^1 n'a pas été re-balancé. Nous avons, par définition, $S = S^1 \cap S^2$. L'ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1} s_i^1 \cap s''_i$.

Lemme 47

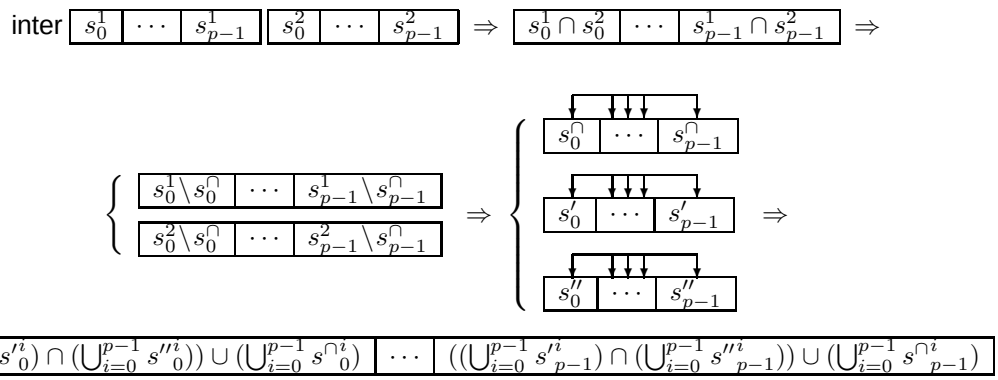
$S \equiv S'$

Preuve. Voir en section 8.A.2 de l'annexe de ce chapitre. ■

Troisième cas. Dans ce dernier cas, les deux ensembles parallèles ont été re-balancés. Les éléments des deux ensembles peuvent avoir été distribués sur n'importe quel processeur. Pour calculer l'intersection, nous calculons les intersections locales. Puis nous calculons les deux différences afin de séparer les éléments qui ne sont pas présents dans les intersections. Enfin nous redistribuons ces trois ensembles et nous calculons l'intersection finale afin de supprimer les éventuels doublons. Ceci correspond au schéma suivant :

```
(* diff: t→t→t *)
let diff (psets1,(step1,nbal1)) (psets2,(step2,nbal2)) = match (nbal1, nbal2) with
    (true,true) (* deux ensembles non re−balancés *) →(parfun2 LocSet.diff psets1 psets2,(step1+step2,true))
  | (true,false) (* le second ensemble a été re−balancé *) →
      let diff1 = parfun2 LocSet.diff psets1 psets2
      and diff2 = parfun2 LocSet.diff psets2 psets1 in
      let diff' = from_sets_to_parset diff2 in
        (parfun2 LocSet.diff diff1 diff2, (step1+step2,true))
  | (false,true) (* le premier ensemble a été re−balancé *) →
      let diff1 = parfun2 LocSet.diff psets1 psets2 in
      let distr = from_sets_to_parsets diff1 in
        (parfun2 LocSet.diff distr psets2, (step1+step2,true))
  | (false,false) (* deux ensembles re−balancés *) →
      let diff1 = parfun2 LocSet.diff psets1 psets2
      and diff2 = parfun2 LocSet.diff psets2 psets1 in
      let distr = distributed_two_sets diff1 diff2 in
      let new_set = parfun2 LocSet.diff (fst distr) (snd distr) in
        (new_set, (step1+step2,true))
```

Figure 8.4 — Calcul de la différence de deux ensembles parallèles



avec $\forall j\ s_j^\cap = s_j^1 \cap s_j^2$ et $s'_j = s_j^1 \setminus s_j^\cap$ et $s''_j = s_j^2 \setminus s_j^\cap$ et $s'^i_j =$ (resp. s''^i_j et $s^{\cap i}_j$) ensemble des éléments envoyés par le processeur i au processeur j à partir de l'ensemble s'_j (resp. s''_j et $s^\cap{}_j$). Le coût BSP est :

$$\max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s_j^1 \cap s_j^2) + \mathcal{C}_{\mathbf{t}}(s_j^1 \setminus s_j^2) + \mathcal{C}_{\mathbf{t}}(s_j^2 \setminus s_j^1)) + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1} s'^i_j) + \mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1} s''^i_j))$$
$$+(\max_{j=0}^{\mathbf{p}-1}(\mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1} s'^i_j) + \mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1} s''^i_j))) \times \mathbf{g} + \mathbf{l}$$

Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s_i^1$ et $S^2 = \bigcup_{i=0}^{p-1} s_i^2$. Nous avons, par définition, $S = S^1 \cap S^2$. L'ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1}((\bigcup_{j=0}^{p-1} s'^j_i) \cap (\bigcup_{j=0}^{p-1} s''^j_i)) \cup (\bigcup_{j=0}^{p-1} s^{\cap j}_i)$.

Lemme 48

$S \equiv S'$

Preuve . Voir en section 8.A.2 de l'annexe de ce chapitre. ■

8.5.5 Différence de deux ensembles

A la différence des deux autres opérateurs qui sont commutatifs, celui-ci ne l'est pas. Nous avons donc quatre cas. Dans le premier cas, les deux ensembles parallèles n'ont pas été re-balancés et les éléments sont tous «à destination». Dans le deuxième (resp. troisième) cas, le premier (resp. second) des ensembles parallèles a été re-balancé. Dans le quatrième et dernier cas, les deux ensembles ont été re-balancés. Par conséquent, les éléments peuvent être situés sur n'importe quel processeur. La figure 8.4 donne le code BSML de la différence de deux ensembles parallèles. Notons que l'ensemble parallèle résultant sera toujours non re-balancé, c'est-à-dire avec tous les éléments «à destination».

Premier cas. Il suffit de calculer de manière asynchrone les différences locales, ce qui correspond au schéma suivant :

$$\text{diff}\ \boxed{s_0^1 \mid \cdots \mid s_{p-1}^1}\ \boxed{s_0^2 \mid \cdots \mid s_{p-1}^2} \Rightarrow \boxed{s_0^1 \setminus s_0^2 \mid \cdots \mid s_{p-1}^1 \setminus s_{p-1}^2}$$

Le coût BSP est donc le temps maximal utilisé par un processeur pour faire la différence entre ces deux ensembles : $\max_{i=0}^{p-1} \mathcal{C}_{\mathbf{t}}(s_i^1 \setminus s_i^2)$.

Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s_i^1$ et $S^2 = \bigcup_{i=0}^{p-1} s_i^2$ tels que S^1 et S^2 n'ont pas été re-balancés. Nous avons, par définition, $S = S^1 \setminus S^2$ et donc $\forall x \in S^1$ et si $x \notin S^2 \iff x \in S$. L'ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1} s_i^1 \setminus s_i^2$.

Lemme 49

$S \equiv S'$

Preuve. Voir en section 8.A.3 de l'annexe de ce chapitre. ■

Deuxième cas. Nous sommes dans le cas où le premier des ensembles parallèles a été re-balancé. Ses éléments peuvent avoir été distribués sur n'importe quel processeur. Pour pouvoir effectuer la différence, il nous faut alors remettre ses éléments «à destination». Pour cela, nous calculons la différence locale, éliminant ainsi les éléments déjà «à destination». Ensuite, nous redistribuons (remettre «à destination») les éléments afin d'éliminer les éventuels doublons. Puis nous calculons la différence entre ces éléments et le second ensemble afin de calculer les derniers éléments à enlever. Ceci correspond au schéma suivant :

$$\text{diff}\ \boxed{s_0^1 \mid \cdots \mid s_{p-1}^1}\ \boxed{s_0^2 \mid \cdots \mid s_{p-1}^2} \Rightarrow \boxed{s_0^1 \setminus s_0^2 \mid \cdots \mid s_{p-1}^1 \setminus s_{p-1}^2} \Rightarrow$$

$$\boxed{s'_0 \mid \cdots \mid s'_{p-1}} \Rightarrow \boxed{(\bigcup_{i=0}^{p-1} {s'}^i_0) \setminus s_0^2 \mid \cdots \mid (\bigcup_{i=0}^{p-1} {s'}^i_{p-1}) \setminus s_{p-1}^2}$$

avec $\forall j\ s'_j = s_j^1 \setminus s_j^2$ et ${s'}^i_j =$ ensemble des éléments envoyés par le processeur i au processeur j à partir des ensembles s'_j. Le coût BSP est :

$$\max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s_j^1 \setminus s_j^2)) + (\max_{j=0}^{\mathbf{p}-1} \mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1} {s'}^i_j)) \times \mathbf{g} + \mathbf{l} + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1} {s'}^i_j \setminus s_j^2))$$

Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s_i^1$ et $S^2 = \bigcup_{i=0}^{p-1} s_i^2$ tel que S^2 n'a pas été re-balancé. Nous avons, par définition, $S = S^1 \setminus S^2$. L'ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1}(s''_i \setminus ((\bigcup_{j=0}^{p-1} {s'}^j_i) \cap s_i^2))$.

Lemme 50

$S \equiv S'$

Preuve. Voir en section 8.A.3 de l'annexe de ce chapitre. ■

Troisième cas. Nous somme dans le cas où le deuxième des ensembles parallèles a été re-balancé. Ses éléments peuvent avoir été distribués sur n'importe quel processeur. Pour pouvoir calculer la différence, il nous faut alors remettre ses éléments «à destination». Pour cela, nous calculons les deux différences locales. Puis nous re-distribuons les éléments de cette deuxième différence, et cela sans conserver les éléments qui étaient déjà «à destination» (ces éléments ayant déjà été éliminés par la première différence). Enfin, nous faisons une différence locale afin d'éliminer ces derniers éléments. Ceci correspond au schéma suivant :

$$\text{diff}\ \boxed{s_0^1 \mid \cdots \mid s_{p-1}^1}\ \boxed{s_0^2 \mid \cdots \mid s_{p-1}^2} \Rightarrow \left\{ \begin{array}{l} \boxed{s_0^1 \setminus s_0^2 \mid \cdots \mid s_{p-1}^1 \setminus s_{p-1}^2} \\ \boxed{s_0^2 \setminus s_0^1 \mid \cdots \mid s_{p-1}^2 \setminus s_{p-1}^1} \end{array} \right. \Rightarrow$$

$$\boxed{s''_0 \mid \cdots \mid s''_{p-1}} \Rightarrow \boxed{s'_0 \setminus (\bigcup_{i=0}^{p-1} {s''}^i_0) \mid \cdots \mid s'_{p-1} \setminus (\bigcup_{i=0}^{p-1} {s''}^i_{p-1})}$$

avec $\forall j\ s'_j = s^1_j \backslash s^2_j$ er $s''_j = s^2_j \backslash s^1_j$ et $s''^i_j =$ ensemble des éléments envoyés par le processeur i au processeur j à partir des ensembles s''_j Le coût BSP est :

$$\max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s^2_j \setminus s^1_j)) + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s^1_j \setminus s^2_j)) + (\max_{j=0}^{\mathbf{p}-1}\mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1} s''^i_j)) \times \mathbf{g} + \mathbf{l} + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s'_j \setminus \bigcup_{i=0}^{\mathbf{p}-1} s''^i_j))$$

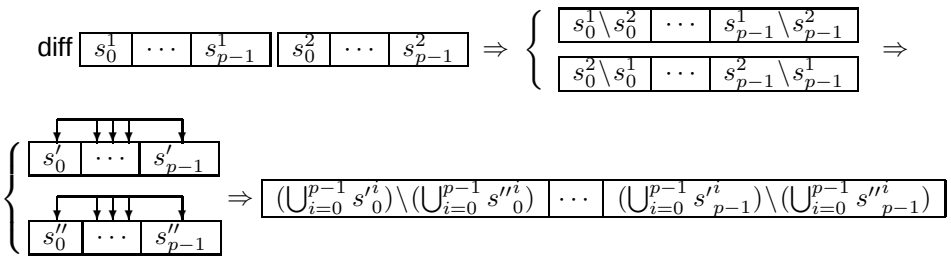
Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s^1_i$ et $S^2 = \bigcup_{i=0}^{p-1} s^2_i$ tels que S^1 n'a pas été re-balancé. Nous avons, par définition, $S = S^1 \setminus S^2$. L'ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1}(s'_i \setminus (\bigcup_{j=0}^{p-1} s''^j_i))$.

Lemme 51

$S \equiv S'$

Preuve . Voir en section 8.A.3 de l'annexe de ce chapitre. ■

Quatrième cas. Dans ce dernier cas, les 2 ensembles parallèles ont été re-balancés. Les éléments des 2 ensembles peuvent donc avoir été distribués sur n'importe quel processeur. Pour calculer la différence, nous effectuons d'abord les différences locales. Puis nous redistribuons ces ensembles afin de mettre les éléments «à destination». Enfin, nous terminons le calcul par des différences locales qui suppriment les doublons. Ceci correspond au schéma suivant :



avec $\forall j\ s'_j = s^1_j \backslash s^2_j$ et $s''_j = s^2_j \backslash s^1_j$ et $s'^i_j =$ (resp. s''^i_j) ensemble des éléments envoyés par le processeur i au processeur j à partir de l'ensemble s'_j (resp. s''_j). Le coût BSP est :

$$\begin{aligned}\max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(s^1_j \backslash s^2_j) + \mathcal{C}_{\mathbf{t}}(s^2_j \backslash s^1_j)) + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1} s'^i_j) + \mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1} s''^i_j)) + \\ (\max_{j=0}^{\mathbf{p}-1}(\mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1} s'^i_j) + \mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1} s''^i_j))) \times \mathbf{g} + \mathbf{l}\end{aligned}$$

Nous avons donc $S^1 = \bigcup_{i=0}^{p-1} s^1_i$ et $S^2 = \bigcup_{i=0}^{p-1} s^2_i$. Nous avons, par définition, $S = S^1 \setminus S^2$. L'ensemble résultant du calcul est $S^r = \bigcup_{i=0}^{p-1}((\bigcup_{j=0}^{p-1} s'^j_i) \setminus (\bigcup_{j=0}^{p-1} s''^j_i))$

Lemme 52

$S \equiv S'$

Preuve . Voir en section 8.A.3 de l'annexe de ce chapitre. ■

8.6 Implantation des Piles et Files

La bibliothèque standard d'OCaml fournit des structures de données plus impératives comme par exemple les piles («Stack») ou les files («Queue»). Comme il est possible d'utiliser (mais de manière limitée, voir [C4]) les traits impératifs d'OCaml en BSML, nous pouvons aussi implanter en BSML ce genre de structures. Nous prenons comme exemple la structure de la pile. L'implantation des files est similaire. Notons que ces structures ne seront pas re-balancées. En effet, il est facile de voir qu'il sera impossible d'avoir toutes les données sur une seule composante de la machine parallèle. Les données vont être distribuées de manière uniforme, une par processeur et ceci au fur et à mesure.

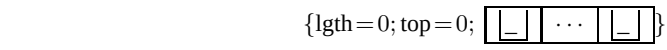
Les piles parallèles sont définies comme suit :

type α t = { stacks: α Stack.t **par**; **mutable** top: int; **mutable** length:int}

où length est le nombre d'éléments dans la pile, top est le processeur où est mémorisé le sommet de la pile et stacks est un vecteur parallèle de piles locales. Ainsi chaque élément est mémorisé au dernier top. Nous créons une pile parallèle avec le code suivant :

```
(* create: unit→α t *)
let create () = { length=0; top=0; stacks= mkpar (fun _ →Stack.create())}
```

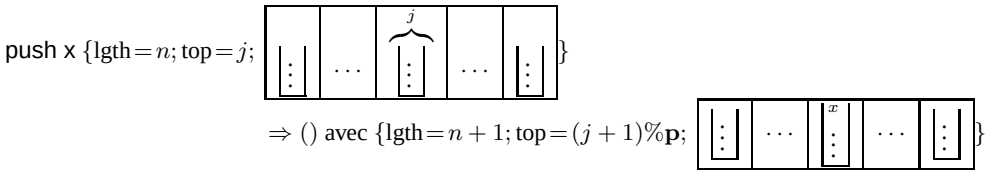
Nous avons donc bien une pile par processeur, et ceci correspond au schéma suivant :



Pour ajouter un élément à la pile, nous augmentons la taille de la pile puis nous ajoutons l'élément au sommet de la pile locale définie par top et enfin nous décalons top.

```
(* push: α →α t→unit *)
let push x s =
  s.length<−s.length+1;
  ignore(applyat s.top (fun locs →Stack.push x locs) (fun _ →()) s.stacks);
  s.top<−(s.top+1) mod (bsp_p())
```

Ce qui correspond au schéma suivant :

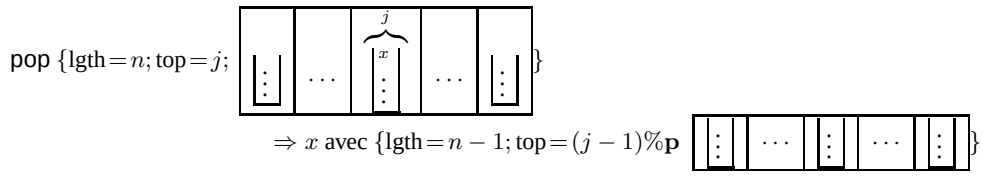


Le coût BSP est le temps utilisé par un processeur pour ajouter l'élément au sommet de sa pile locale et le temps d'augmenter length et top.

Pour enlever (et retourner) l'élément du sommet de la pile parallèle, nous devons juste diffuser cet élément du «processeur sommet» aux autres processeurs avec la primitive de projection **proj**, puis décaler le top et diminuer la taille de la pile :

```
(* pop: α t→α  *)
let pop s =
 if s.length = 0 then raise Empty else
  let f=proj (applyat s.top (fun locq →Some (Stack.pop locq)) (fun _ →None) s.stacks) in
  let x=noSome (f s.top) in
   s.length<−s.length−1;
   s.top<−(s.top−1) mod (bsp_p());
   x
```

ce qui correspond au schéma suivant :



Le coût BSP est donc : $(\mathbf{p}-1)\times\mathcal{S}(x)\times\mathbf{g}+\mathbf{l}$ tel que x est l'élément de tête de la pile. Nous pouvons ensuite définir facilement les autres fonctions de base du module, notamment un itérateur parallèle :

```
(* is_empty: α t→bool et length: α t→int *)
let is_empty s = s.length = 0
let length s = s.length

(* async_iter: (α →unit) par→α  t→unit *)
let async_iter parf s = ignore(apply2 (replicate Stack.iter) parf s.stacks)
```

```
module AtomKey = struct type t = int * int list let compare = compare end
module AtomSet = Set.Make(AtomKey)

(* nth_nn : int →AtomSet.elt →AtomSet.t *)
let rec nth_nn = let memory=Hashtbl.create 1 in
 fun n (i, io)→ try Hashtbl.find memory (n,i) with Not_found→match n with
   0 →AtomSet.singleton (i,io)
 | 1 →let nn=bonds.(i−1) in if io = zero then nn else
         let aux (j,jo) s=AtomSet.add (j,add_i io jo) s in
          AtomSet.fold aux nn AtomSet.empty
 | n →let pprev=nth_nn (n−2) (i,io) and prev=nth_nn (n−1) (i,io) in
         let aux j t=AtomSet.union (nth_nn 1 j) t in
         let t=AtomSet.fold aux prev AtomSet.empty in
         let t'=AtomSet.diff (AtomSet.diff t prev) pprev in
           Hashtbl.add memory (n, i) t'; t'
```

Figure 8.5 — Code Séquentiel

8.7 Exemple d'une application scientifique

Pour illustrer l'utilisation des structures de données parallèles pour le calcul scientifique (et notamment «symbolique»), nous présentons, dans cette section, les performances séquentielles et parallèles d'une petite application : la recherche des n plus proches voisins d'un sommet d'un graphe non-orienté et infiniment répété. Ce graphe représente la topologie d'une molécule où les sommets sont des atomes et les arêtes des liaisons chimiques entre les atomes constituant ladite molécule. Cette recherche permet une meilleure compréhension de la structure physique de la molécule [140]. Une solution et son code associé en OCaml sont décrits dans [140].

A partir d'un graphe non-orienté et infiniment répété, la tâche consiste à trouver les 1ers, 2èmes, ..., $n$èmes voisins d'un sommet du graphe. Nous appelons 1ers voisins d'un sommet, l'ensemble des sommets du graphe connectés par une unique arrête au sommet initial. Les 2èmes voisins forment l'ensemble des sommets du graphe voisin d'un des 1ers voisins du sommet initial et excluant les 1ers voisins eux-mêmes (différences d'ensembles). Et cela ainsi de suite. Cette définition peut ainsi être généralisée en une relation de récurrence qui calcule les $n$èmes voisins par des opérations ensemblistes. Pour rendre les choses plus intéressantes, le graphe est infiniment repété ce qui exclut les algorithmes de recherches classiques sur les graphes. De plus, le nombre d'arêtes est important, (e.g. $> 10^5$) rendant impossible l'utilisation d'une matrice d'adjacence : il faudrait pouvoir stocker 10^{10} éléments dans la mémoire principale.

La figure 8.5 donne le code séquentiel de la partie principale de l'application où bonds:AtomSet.t array est l'ensemble initial d'atomes (la molécule), add_i: int list→int list→int list calcule un nouveau voisin à partir de deux coordonnées et zero:int list est l'atome intial.

La figure 8.6 donne le code parallèle, c'est-à-dire le code séquentiel légèrement modifié pour mieux utiliser la structure de données parallèle ParSet : nous avons remplacé les fold par des async_fold et où from_list_of_parlist:elt list **par** list→t est une fonction qui donne un ensemble parallèle à partir d'un vecteur de listes d'éléments et list_of_set:AtomSet.t→ AtomSet.elt list est une fonction qui retourne une liste de tous les éléments d'un ensemble parallèle (cette fonction implique donc un échange total de ces éléments).

Nous avons effectué quelques expériences de cette application en utilisant 10 ou 5 nœuds de la grappe avec 1Go de RAM par nœud pour comparer l'utilisation des structures de données séquentielles et parallèles. Le graph de test représente un silicate de 100000 atomes[3]. Nous avons utilisé les versions MPI, PUB et TCP/IP de la BSMLlib. Pour chaque n (calcul des $n$èmes voisins), l'algorithme a été exécuté cent fois de suite avec un atome initial tiré au sort. Cette opération a été repétée cinq fois et la moyenne de ces exécutions a ensuite été utilisée.

Les figures 8.7 et 8.8 donnent ces résultats. Dans la première figure, nous comparons les versions séquentielles et parallèles avec respectivement 10 et 5 processeurs. Dans la deuxième figure, nous comparons les versions avec ou sans re-balancement, pour toujours 10 et 5 processeurs, mais avec des n bien plus grands

[3]Téléchargeable à `http://ffconsultancy.com/products/ocaml_for_scientists/`

```
module Balance = struct
 let need_balance ()=true and unbalanced ()=1000 and max_op_without_balance ()=10 end
module AtomKey = struct type t = int * int list let compare = compare let hash=Hashtbl.hash end
module AtomSet = ParSet.Make(AtomKey)(Balance)

(* async_nth: int*int list →(int*int list) list par *)
let async_nth (i,io) = let nn = bonds.(i − 1) in
 if io = zero then (set_to_parlist nn) else
  let aux (j, jo) l = (j, add_i io jo)::l in (AtomSet.async_fold aux nn [])

let rec nth_nn = let memory = Hashtbl.create 1 in
 fun n (i, io) →try Hashtbl.find memory (n, i) with Not_found →match n with
   0 →AtomSet.singleton (i, io)
 | 1 →let nn = bonds.(i − 1) in if io = zero then nn else
          let aux (j, jo) l = (j, add_i io jo)::l in
           from_list_of_parlist [(AtomSet.async_fold aux nn [])]
 | n →let pprev = nth_nn (n−2) (i, io) and prev = nth_nn (n−1) (i, io) in
         let lt = List.map (fun j →(async_nth j)) (list_of_set prev) in
         let t = from_list_of_parlist lt in
         let t' = AtomSet.diff (AtomSet.diff t prev) pprev in
          Hashtbl.add memory (n, i) t'; t'
```

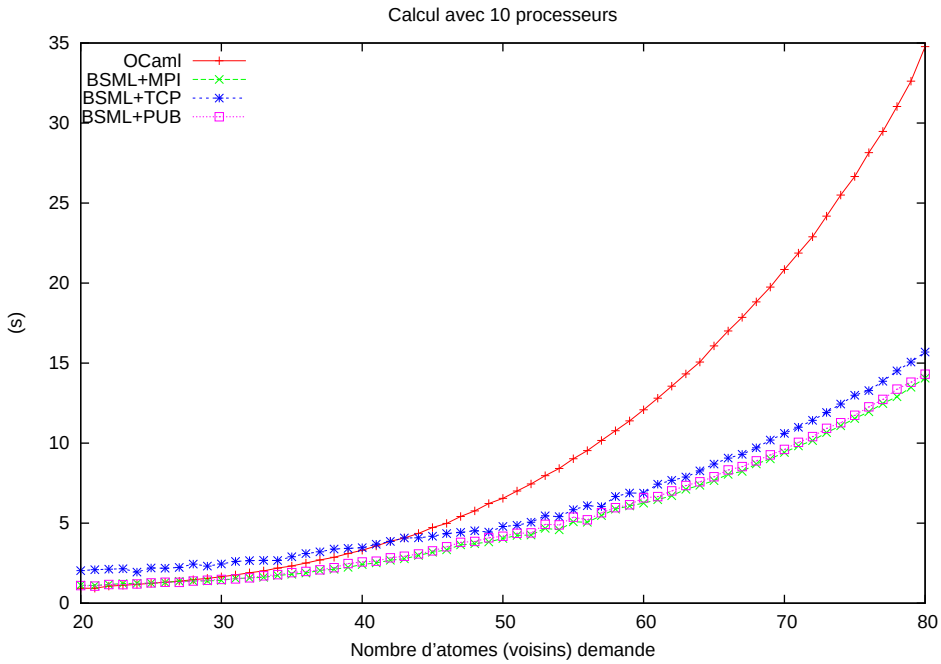
Figure 8.6 — Code parallèle

(et donc avec des ensembles d'atomes plus importants). Nous notons sur les courbes de performances "+LB" quand un re-balancement a été utilisé. Nous utilisons les paramètres de re-balancement donnés dans le code de la figure 8.6.

Pour de petits n, les performances de la version parallèle de l'application sont équivalentes (voir moindres) que celles de la version séquentielle. Cela est dû au surcoût des barrières de synchronisation. Par contre, les performances de la version parallèle sont sensiblement meilleures que celles de la version séquentielle quand les n augmentent. Notons qu'avec 5 processeurs, les performances sont meilleures qu'avec 10 processeurs, ce qui est dû au grand nombre de barrières de synchronisation qu'implique l'application : celles-ci sont bien plus coûteuses avec 10 processeurs qu'avec 5. Grâce à l'utilisation de la superposition, réorganiser (ou re-balancer) deux ou trois ensembles parallèles peut être effectué en une seule (et unique) super-étape, limitant ainsi le nombre de barrières lors de l'exécution de l'application. Avec 5 processeurs, la version avec re-balancement est clairement plus efficace que la version naïve (sans re-balancement). Par contre, toujours à cause du coût des barrières, le re-balancement avec 10 processeurs n'apporte quasiment rien.

Nous avons aussi testé le code séquentiel avec différents n sur un PC portable. Nous donnons ici, à titre d'information, les cardinalités maximales des ensembles ainsi que le temps nécessaire et la mémoire maximale utilisée :

Éième	Cardinal	Temps (s)	Mémoire utilisée (Mo)
60	12878	19.82	40
80	23254	51.71	79
100	36624	114.19	148
120	52917	233.62	251
140	72341	424.72	382



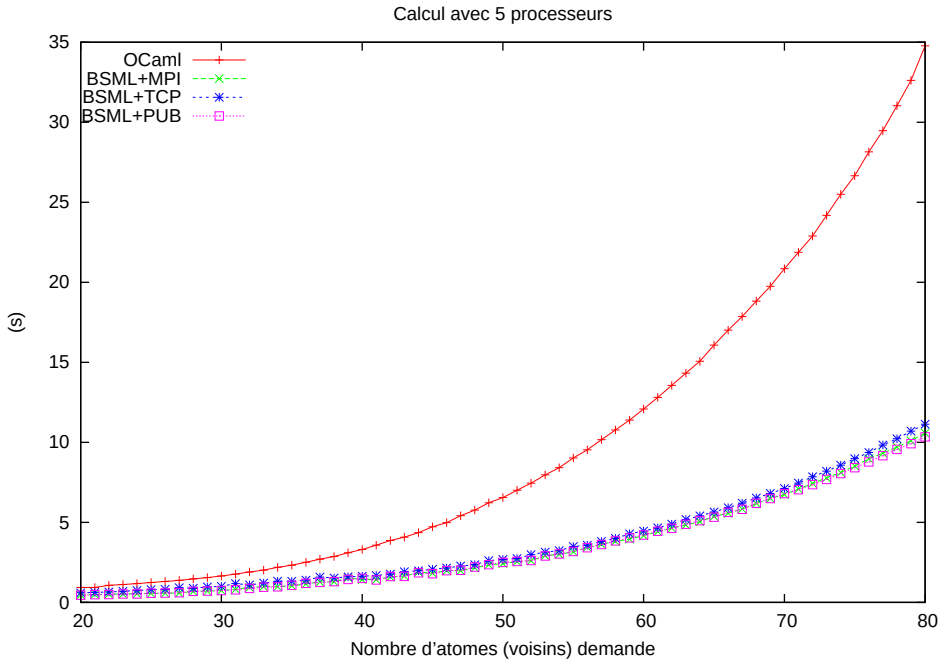
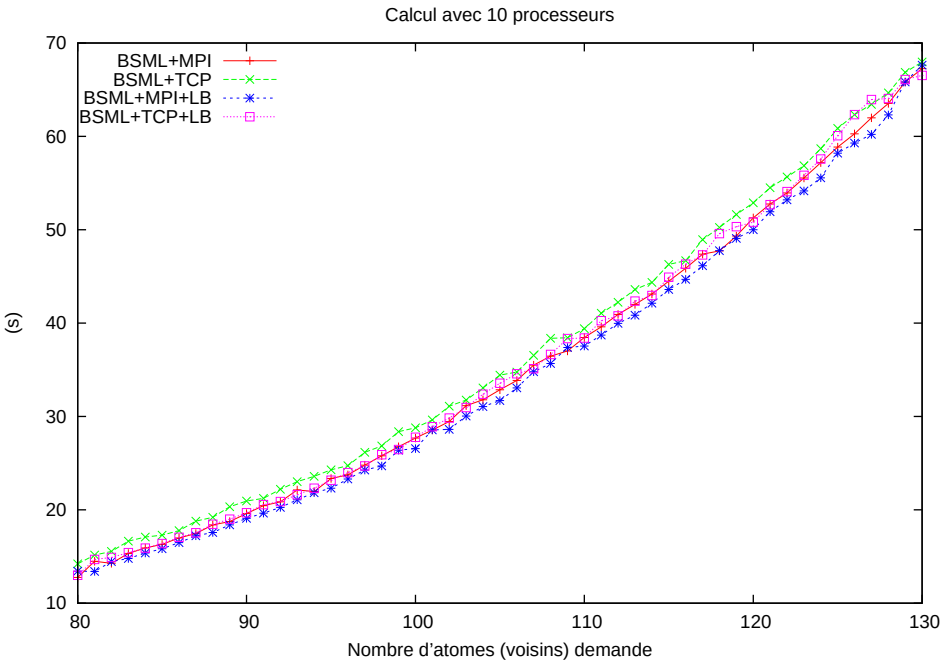


Figure 8.7 — Performances mesurées, séquentielles et parallèles, pour le calcul des énièmes voisins les plus proches



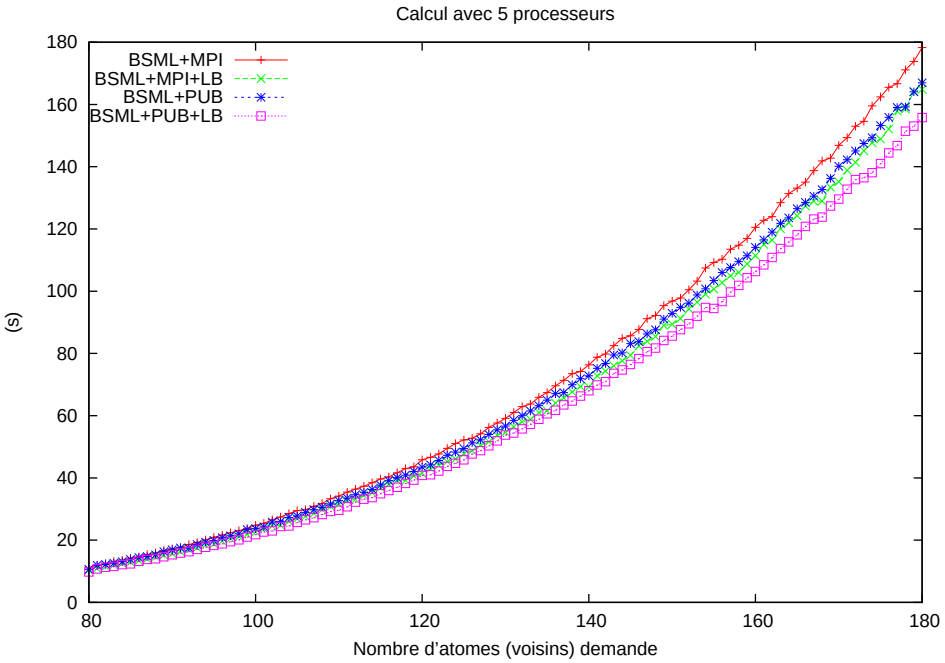


Figure 8.8 — Performances mesurées, avec ou sans rebalancement, pour le calcul des énièmes voisins les plus proches

8.A Preuves des opérations ensemblistes

8.A.1 Preuves des opérations de l'union

Pour toutes les preuves, les ensembles donnés aux opérateurs sont valides, c'est-à-dire que si notre ensemble $s = \bigcup_{k=0}^{p-1} s_k$ alors $\forall i, j\ i \neq j$ et $\forall x \in s_i$, alors $x \notin s_j$. Ceux-ci permet d'appliquer le lemme 54 sur tous nos ensembles d'origine. Notons que si un ensemble vérifie le lemme 53 alors nous pouvons aussi appliquer le lemme 54 sur cet ensemble. Nous notons, comme il usuellement admis, $\exists!$ pour : "existe un et unique".

Outils. Soit $s_0, \ldots, s_{p-1}$, p ensembles.

Hypothèse 1 (*Redistribution*)

Soit un ensemble s_k. Si l'ensemble s_k provient d'une redistribution des données alors $\forall x \in s_k\ h(x) = k$.

Lemme 53

Soit x tel que $x \in s_i$, $h(x) = i$ et j tel que $i \neq j$, alors $x \notin s_j$

Preuve. Par apagogie. Si $x \in s_j$ alors $j = h(x) = i$. Ce qui contredit $i \neq j$. ■

Lemme 54

Soit $S = \bigcup_{k=0}^{p-1} s_k$. Si nous avons la propriété suivante : $\forall i, j$ tel que $i \neq j$, soit $\forall x \in s_i$, alors $x \notin s_j$. Soit x tel que $x \in S$, alors il existe un (et unique) k tel que $x \in s_k$

Preuve. Evident par apagogie. ■

Premier cas. Nous l'avons l'hypothèse que $\forall x \in s_k^1,\ h(x) = k$ et $\forall x \in s_k^2,\ h(x) = k$.

Prop. 1) Si $x \in S$ alors $x \in S^1 \cup S^2$, donc $x \in (\bigcup_{k=0}^{p-1} s_k^1) \cup (\bigcup_{k=0}^{p-1} s_k^2)$. Par le lemme 54, $x \in s_i^1$ ou $x \in s_i^2$. Alors $x \in s_i^1 \cup s_i^2$ et donc $x \in \bigcup_{k=0}^{p-1} s_k^1 \cup s_k^2$.

Prop. 2) Si $x \in S^r$ alors $x \in \bigcup_{k=0}^{p-1} s_k^1 \cup s_k^2$. Grâce au lemme 53, nous appliquons le lemme 54 et nous avons $x \in (s_i^1 \cup s_i^2)$. Si $x \in s_i^1$ alors $x \in S^1$ et $x \in S$. Si $x \in s_i^2$ alors $x \in S^2$ et $x \in S$.

Prop. 3) $S^r = \bigcup_{k=0}^{p-1} s_k^1 \cup s_k^2$. Soit $x \in s_i^1 \cup s_i^2$. Alors par hypothèse, $\forall j\ i \neq j\ x \notin s_j^1$ et $x \notin s_j^2$. Par conséquence $x \notin s_j^1 \cup s_j^2$.

Deuxième cas. Nous l'avons l'hypothèse que $\forall x \in s_k^1,\ h(x) = k$.

Prop. 1) Si $x \in S^1 \cup S^2$ alors $x \in (\bigcup_{k=0}^{p-1}) s_k^1 \cup (\bigcup_{i=0}^{p-1} s_k^2)$. Nous avons alors deux cas :

- $x \in \bigcup_{k=0}^{p-1} s_k^1$. Par le lemme 54, nous avons $x \in s_i^1 \Rightarrow x \in s_i^1 \cup s_i'' \Rightarrow S^r$
- $x \in \bigcup_{k=0}^{p-1} s_k^2$. Par le lemme 54, nous avons $x \in s_i^2$. Nous avons alors deux cas :
  - si $x \notin s_i^2 \setminus s_i^1$ alors $x \in s_i^1$ et donc $x \in S^r$
  - si $x \in s_i^2 \setminus s_i^1$ alors $x \in s_i' \Rightarrow\ \exists j\ x \in s_j'' \Rightarrow x \in S^r$.

Lemme 55

Soit x tel que $x \in s_k'^i$ ($s_k'^i$ défini à la section 8.5.3) alors $h(x) = k$.

Preuve. Par hypothèse de la redistribution. ■

Lemme 56

Soit x tel que $x \in s''_k$ (s''_k défini à la section 8.5.3) alors $h(x) = k$.

Preuve. Si $x \in s''_k$ alors $x \in \bigcup_{i=0}^{p-1} s'^i_k$. Par application du lemme 55 $h(x) = k$. ■

Prop. 2) Si $x \in S^r$ alors $x \in \bigcup_{k=0}^{p-1} s''_k \cup s^1_k$. Par application de l'hypothèse sur s^1_k et des lemmes 56 et 54, nous avons $x \in s''_i \cup s^1_i$. Deux cas se présentent :

- si $x \in s^1_i$ alors $x \in S^1$ et $x \in S$.
- si $x \in s''_i$ alors $x \in \bigcup_{j=0}^{p-1} s'^j_i \Rightarrow x \in s'^j_i \Rightarrow x \in s'_i \Rightarrow x \in s^2_i \setminus s^1_i \Rightarrow x \in s^2_i \Rightarrow x \in S^1 \Rightarrow x \in S$.

Prop. 3) $x \in s''_i \cup s^1_i$. Si $x \in s''_i$ alors $\exists j\ x \in s'^j_i \Rightarrow x \in s'_j \Rightarrow x \in s^2_j \setminus s^1_j \Rightarrow x \in s^2_j$. Par hypothèse, nous avons donc $\forall k\ k \neq j\ x \notin s^2_k$. Par conséquent, $\forall k\ k \neq j\ x \notin s^2_k \setminus s^1_k \Rightarrow x \notin s'_k$. Par hypothèse sur la redistribution, nous avons $\exists! j\ x \in s'^j_i$ et donc $\forall k\ k \neq i\ x \notin s''_k$. Si $x \in s^1_i$, alors par hypothèse $\forall k\ k \neq i\ x \notin s^1_k$. Nous concluons donc bien $\forall k\ k \neq i\ x \notin s''_k \cup s^1_k$.

Troisème cas.

Prop. 1) Si $x \in S^1 \cup S^2$ alors $x \in (\bigcup_{k=0}^{p-1}) s^1_k \cup (\bigcup_{i=0}^{p-1} s^2_k)$. Nous avons alors deux cas :

- $x \in \bigcup_{k=0}^{p-1} s^1_k$. Par le lemme 54, nous avons $x \in s^1_i \Rightarrow x \in s^1_i \cup s^2_i = s'_i \Rightarrow \exists j\ x \in s''_j \Rightarrow x \in S^r$
- $x \in \bigcup_{k=0}^{p-1} s^2_k$. Par le lemme 54, nous avons $x \in s^2_i \Rightarrow x \in s^1_i \cup s^2_i = s'_i \Rightarrow \exists j\ x \in s''_j \Rightarrow x \in S^r$

Lemme 57

Soit x tel que $x \in s'^i_k$ (s'^i_k défini à la section 8.5.3) alors $h(x) = k$.

Preuve. Par hypothèse de la redistribution. ■

Lemme 58

Soit x tel que $x \in s''_k$ (s''_k défini à la section 8.5.3) alors $h(x) = k$.

Preuve. Si $x \in s''_k$ alors $x \in \bigcup_{i=0}^{p-1} s'^i_k$. Par application du lemme 57 $h(x) = k$. ■

Prop. 2) Si $x \in S^r$ alors $x \in \bigcup_{k=0}^{p-1} s''_k$. Par application des lemmes 58 et 54, nous avons $x \in s''_i$. Nous avons alors $x \in \bigcup_{j=0}^{p-1} s'^j_i \Rightarrow x \in s'^j_i \Rightarrow x \in s'_i \Rightarrow x \in s^1_i \cup s^2_i$. Nous avons alors si $x \in s^1_i$ (resp. s^2_i) alors $x \in S$.

Prop. 3) $x \in s''_i$. Par définition, $x \in \bigcup_{j=0}^{p-1} s'^j_i$ et donc $\exists j\ x \in s'_j \Rightarrow x \in s^1_j \cup s^2_j$. Par hypothèse, $\forall k\ k \neq j\ x \notin s^1_k$ et $x \notin s^2_k$. Donc $x \notin s^1_k \cup s^2_k \Rightarrow x \notin s'_k$. Par hypothèse sur la redistribution, $\exists! j\ x \in s'^j_i$ et nous en concluons que $\forall k\ k \neq i\ x \notin s''_k$.

8.A.2 Preuves des opérations de l'intersection

Premier cas. Nous avons l'hypothèse que $\forall x \in s^1_k,\ h(x) = k$ et $\forall x \in s^2_k,\ h(x) = k$.

Prop. 1) Si $x \in S$ alors $x \in S^1 \cap S^2$, donc $x \in (\bigcup_{k=0}^{p-1} s^1_k) \cap (\bigcup_{k=0}^{p-1} s^2_k)$. Par le lemme 54, $x \in s^1_i$ et $x \in s^2_i$. Alors $x \in s^1_i \cap s^2_i$ et donc $x \in \bigcup_{k=0}^{p-1} s^1_k \cap s^2_k$.

Prop. 2) Si $x \in S^r$ alors $x \in \bigcup_{k=0}^{p-1} s_k^1 \cap s_k^2$. Grâce au lemme 53, nous appliquons le lemme 54 et nous avons $x \in (s_i^1 \cap s_i^2)$. Comme $x \in s_i^1$ et $x \in s_i^2$ alors $x \in S^1$ et $x \in S^2$. On a donc bien que $x \in S$.

Prop. 3) $S^r = \bigcup_{k=0}^{p-1} s_k^1 \cap s_k^2$. $x \in s_i^1 \cap s_i^2$. Par hypothèse, $\forall k\ k \neq i\ x \notin s_k^1$ et $x \notin s_k^2$. Donc $x \notin s_k^1 \cap s_k^2$.

Deuxième cas.

Nous l'avons l'hypothèse que $\forall x \in s_k^1,\ h(x) = k$.

Prop. 1) Si $x \in S^1 \cap S^2$ alors $x \in (\bigcup_{k=0}^{p-1} s_k^1) \cap (\bigcup_{i=0}^{p-1} s_k^2)$. Nous avons alors $x \in \bigcup_{k=0}^{p-1} s_k^1$ et $x \in \bigcup_{k=0}^{p-1} s_k^2$. Par le lemme 54, nous avons $x \in s_i^1$ et $x \in s_j^2$. Nous avons lors deux cas :

- $i = j$, alors $x \in s_i^\cap \Rightarrow x \in s_i^\cap \cup (s_i'' \cap s_i^1) \Rightarrow x \in S^r$
- $i \neq j$ alors $x \in s_j^2 \cap s_j^1 = s_j'$. Par hypothèse, $h(x) = i$ donc $x \in s_i'' \Rightarrow x \in s_i'' \cap s_i' \Rightarrow x \in s_i^\cap \cup (s_i'' \cap s_i^1) \Rightarrow x \in S^r$

Lemme 59

Soit x tel que $x \in s_k^\cap \cup (s_k'' \cap s_k^1)$ ($s_k^\cap$, s_k'' et s_k^1) (ensembles définis à la section 8.5.4), alors $h(x) = k$.

Preuve. Nous avons deux cas :

- si $x \in s_k'' \cap s_k^1$ alors $x \in s_k^1$ et donc $h(x) = k$ par hypothèse
- si $x \in s_k^\cap$ alors $x \in s_k^1 \cap s_k^2$ et donc $h(x) = k$ par hypothèse.

■

Prop. 2) Si $x \in S^r$ alors $x \in \bigcup_{k=0}^{p-1} s_k^\cap \cup (s_k'' \cap s_k^1)$. Par application du lemme 54 (déduit du lemme 59), nous avons $x \in s^\cap \cup (s_i'' \cap s_i^1)$. Nous avons alors deux cas :

- si $x \in s_i^\cap$ alors $x \in s_i^1 \cap s_i^2$. Par conséquent, $x \in S^1$ et $x \in S^2$. Donc $x \in S$
- si $x \in s_i'' \cap s_i^1$ alors $\begin{cases} x \in s_i^i \Rightarrow x \in S^1 et \\ x \in s_i'' \text{ donc } \exists j\ x \in s_j' \text{ et } x \in s_i^2 \cap s_i^1 \Rightarrow x \in S^2 \end{cases}$. Par conséquent, $x \in S$.

Prop. 3) $x \in s_i^\cap \cup (s_i'' \cap s_i^1)$. $s_i^\cap = s_i^1 \cap s_i^2$. Par hypothèse, $\forall k\ k \neq i\ x \notin s_k^1$ et $x \notin s_k^2$. Par conséquent, $x \notin (s_k'' \cap s_k^1)$ et nous concluons que $x \notin s_k^\cap \cup (s_k'' \cap s_k^1)$

Troisième cas.

Prop. 1) Si $x \in S^1 \cap S^2$ alors $x \in (\bigcup_{k=0}^{p-1} s_k^1) \cap (\bigcup_{k=0}^{p-1} s_k^2)$. Par application du lemme 54 (déduit des hypothèses sur S^1 et S^2), nous obtenons que $x \in s_i^1$ et $x \in s_j^2$. Nous avons alors deux cas :

- si $i = j$ alors $x \in s_i^1 \cap s_i^2 \Rightarrow x \in s_i^\cap \Rightarrow \exists j\ x \in s^{\cap j}_i \Rightarrow x \in S^r$
- si $i \neq j$ alors $x \in s_i^1 \setminus s_i^2 = s_i'$ et $x \in s_j^2 \setminus s_j^1 = s_j''$. Par conséquent, $x \in s_i'^k$ et $x \in s_j''^k$. Donc $x \in S^r$.

Lemme 60

Soit x tel que $x \in \bigcup_{k=0}^{p-1} s^{\cap i}_k$ ($s^{\cap i}_k$ défini à la section 8.5.4) alors $h(x) = k$.

Preuve. Par hypothèse de la redistribution. ■

Lemme 61

Soit x tel que $x \in \bigcup_{k=0}^{p-1} s_k'^i$ ($s_k'^i$ défini à la section 8.5.4) alors $h(x) = k$.

Preuve. Par hypothèse de la redistribution ■

Lemme 62

Soit x tel que $x \in \bigcup_{k=0}^{p-1} s''^i_k$ (s''^i_k défini à la section 8.5.4) alors $h(x) = k$.

Preuve. Par hypothèse de la redistribution. ■

Prop. 2) Si $x \in \bigcup_{k=0}^{p-1} ((\bigcup_{i=0}^{p-1} s'^i_k) \cap (\bigcup_{i=0}^{p-1} s''^i_k)) \cup (\bigcup_{i=0}^{p-1} s^{\cap i}_k))$ alors, par application des lemmes 60, 61, 62 et 54, nous obtenons que $x \in ((\bigcup_{i=0}^{p-1} s'^i_j) \cap (\bigcup_{i=0}^{p-1} s''^i_j)) \cup (\bigcup_{i=0}^{p-1} s^{\cap i}_j)$. Nous avons alors deux cas :

- si $x \in \bigcup_{i=0}^{p-1} s^{\cap i}_j$ alors par le lemme 60, nous avons $x \in s^{\cap i}_j \Rightarrow x \in s^{\cap}_i \Rightarrow x \in s^1_i \cap s^2_i \Rightarrow x \in S$
- si $x \in (\bigcup_{i=0}^{p-1} s''^i_j) \cap (\bigcup_{i=0}^{p-1} s'^i_j)$ alors par les lemmes 61 et 62, nous avons $x \in s'^{k_1}_j$ et $x \in s''^{k_1}_j$. En conséquence de l'hypothèse de la redistribution, $x \in s'_{k_1}$ et $x \in s''_{k_2}$. Donc $x \in s^1_{k_1} \setminus s^{\cap}_{k_1}$ et $x \in s^2_{k_2} \setminus s^{\cap}_{k_2}$. Donc $x \in s^1_{k_1}$ et $x \in s^2_{k_2}$. Nous en concluons que $x \in S^1$ et $x \in S^2$, et donc $x \in S$.

Prop. 3) Si $x \in ((\bigcup_{i=0}^{p-1} s'^i_j) \cap (\bigcup_{i=0}^{p-1} s''^i_j)) \cup (\bigcup_{i=0}^{p-1} s^{\cap i}_j)$ alors d'après les lemmes 60, 61, 62 et 54, $x \in s'^{k_1}_j$ ou $x \in s''^{k_2}_j$ ou $x \in s^{\cap k_3}_j$. Dans les trois cas, x vient d'une redistribution des éléments et donc par hypothèse de ces redistributions, les éléments sont tous «à destination» (comme dans les cas précédents). Il n'y a donc pas de doublons.

8.A.3 Preuves des opérations de la différence

Premier cas. Nous l'avons l'hypothèse que $\forall x \in s^1_k,\ h(x) = k$ et $\forall x \in s^2_k,\ h(x) = k$.

Prop. 1) Si $x \in S$ alors $x \in S^1 \setminus S^2$, donc $x \in (\bigcup_{k=0}^{p-1} s^1_k) \setminus (\bigcup_{k=0}^{p-1} s^2_k)$. Par le lemme 54, $x \in s^1_i$, $x \notin s^2_i$ et par hypothèse $\forall k\ k \neq i\ x \notin s^1_k$ et $x \notin s^2_k$. Alors $x \in s^1_i \setminus s^2_i$ et donc $x \in \bigcup_{k=0}^{p-1} s^1_k \setminus s^2_k$.

Prop. 2) Si $x \in S^r$ alors $x \in \bigcup_{k=0}^{p-1} s^1_k \setminus s^2_k$. Grâce au lemme 53, nous appliquons le lemme 54 et nous avons $x \in (s^1_i \setminus s^2_i)$. Comme $x \in s^1_i$ et $x \notin s^2_i$ et par hypothèse $\forall k\ k \neq i\ x \notin s^1_k$ et $x \notin s^2_k$, alors $x \in S^1$ et $x \notin S^2$. On a donc bien que $x \in S$.

Prop. 3) $S^r = \bigcup_{k=0}^{p-1} s^1_k \setminus s^2_k$. $x \in s^1_i \setminus s^2_i$. Par hypothèse, $\forall k\ k \neq i\ x \notin s^1_k$ et $x \notin s^2_k$. Donc $x \notin s^1_k \setminus s^2_k$.

Deuxième cas. Nous avons l'hypothèse que $\forall x \in s^2_k,\ h(x) = k$.

Prop. 1) Si $x \in S$ alors $x \in S^1 \setminus S^2$ et $x \in (\bigcup_{k=0}^{p-1} s^1_k) \setminus (\bigcup_{k=0}^{p-1} s^2_k)$. Par hypothèse, $x \in s^1_i$ et $\forall j\ x \notin s^1_j$ (si $j \neq i$) et $x \notin s^2_j$. Donc $x \in s^1_i \setminus s^2_i \Rightarrow x \in s'_i \Rightarrow \exists j\ x \in s'^i_j$ et $x \in S^r$ car $x \notin s^2_j$.

Lemme 63

Soit x tel que $x \in (\bigcup_{i=0}^{p-1} s'^i_k) \setminus s^2_k$ (s^2_k défini à la section 8.5.5) alors $h(x) = k$.

Preuve. Par hypothèse de la redistribution. ■

Prop. 2) Si $x \in S^r$ alors $x \in \bigcup_{k=0}^{p-1} (\bigcup_{i=0}^{p-1} s'^i_k) \setminus s^2_k$. En appliquant les lemmes 63 et 54, on obtient $x \in (\bigcup_{i=0}^{p-1} s'^i_j) \setminus s^2_j$. Alors $x \notin s^2_j$ et par hypothèse sur S^2 $\forall i\ x \notin s^2_i$ et donc $x \notin S^2$. Par conséquent, $x \in s'^i_j \Rightarrow x \in s'_i \Rightarrow x \in s^1_i \setminus s^2_i \Rightarrow x \in s^1_i \Rightarrow x \in S^1$. On en conclut que $x \in S^1 \setminus S^2$.

Prop. 3) Si $x \in (\bigcup_{i=0}^{p-1} s'^i_j) \setminus s^2_j$ alors par hypothèse sur la redistribution $\forall k\ k \neq j\ x \notin \bigcup_{i=0}^{p-1} s'i_k$ et donc $x \notin (\bigcup_{i=0}^{p-1} s'^i_k) \setminus s^2_k$.

Troisième cas. Nous avons l'hypothèse que $\forall x \in s^1_k,\ h(x) = k$.

Prop. 1) Si $x \in S$ alors $x \in S^1 \setminus S^2$ et $x \in (\bigcup_{k=0}^{p-1} s^1_k) \setminus (\bigcup_{k=0}^{p-1} s^2_k)$. Par hypothèse, $x \in s^1_i$ et $\forall j\ x \notin s^1_j$ (si $j \neq i$) et $x \notin s^2_j$. Donc $x \in s^1_i \setminus s^2_i$ et $x \notin s^2_i \setminus s^1_i$. Par conséquent, $x \in s'_i$ et $x \notin s''_i$. On en déduit que $x \in s'_i \setminus (\bigcup_{j=0}^{p-1} s''^j_i)$ et donc $x \in S^r$.

Lemme 64

Soit x tel que $x \in s'_k \setminus (\bigcup_{i=0}^{p-1} s''^i_k)$ (s''^i_k défini à la section 8.5.5) alors $h(x) = k$.

Preuve. Par définition, $s'_k = s^1_k \setminus s^2_k$. Donc par hypothèse sur S^1, $h(x) = k$. ■

Prop. 2) Si $x \in S^r$ alors par l'application du lemme 64, $x \in s'_j \setminus (\bigcup_{i=0}^{p-1} s''^i_j)$. On a donc $x \in s'_j \Rightarrow x \in s^1_j \setminus s^2_j \Rightarrow s^1_j \Rightarrow x \in S^1$. Maintenant prouvons que $x \notin S^2$. Si $x \in S^2$ alors par hypothèse $\exists k$ tel que $x \in s^2_k$. Nous avons alors deux cas :

- si $k = j$ alors $x \notin s^1_j \setminus s^2_j \Rightarrow x \notin s'_j$. contradiction
- si $k \neq j$ alors $x \in s^2_k \setminus s^1_k \Rightarrow x \in s''_k$. Par hypothèse de la redistribution, $x \in s''^k_j \Rightarrow x \notin s'_j \setminus (\bigcup_{i=0}^{p-1} s''^i_j)$. contradiction.

Donc $x \notin S^2$ et par conséquent $x \in S^1 \setminus S^2 = S$.

Prop. 3) Si $x \in s'_j \setminus (\bigcup_{i=0}^{p-1} s''^i_j)$ alors par hypothèse sur s'_j $\forall k\ k \neq j\ x \notin s'_k$ et donc $x \notin s'_k \setminus (\bigcup_{i=0}^{p-1} s''^i_k)$.

Quatrième cas.

Prop. 1) Si $x \in S$ alors $x \in S^1 \setminus S^2$ et $x \in (\bigcup_{k=0}^{p-1} s^1_k) \setminus (\bigcup_{k=0}^{p-1} s^2_k)$. Par hypothèse, $x \in s^1_i$ et $\forall j\ x \notin s^1_j$ (si $j \neq i$) et $x \notin s^2_j$. Par conséquent, $x \in s^1_i \setminus s^2_i \Rightarrow x \in s'_i$. Donc, par hypothèse de la redistribution, $\exists j\ x \in s'^i_j$. On a aussi que $x \notin s^2_i \setminus s^1_i \Rightarrow x \notin s''_i$ et donc $\forall i, j\ x \notin s''^i_j$. On en conclut $x \in S^r$.

Lemme 65

Soit x tel que $x \in \bigcup_{i=0}^{p-1} s'^i_k$ ou $x \in \bigcup_{i=0}^{p-1} s''^i_k$ (s'^i_k et s''^i_k définis à la section 8.5.5) alors $h(x) = k$.

Preuve. Par hypothèse de la redistribution sur les deux cas. ■

Prop. 2) Si $x \in S^r$ alors $x \in \bigcup_{k=0}^{p-1} (\bigcup_{i=0}^{p-1} s'^i_k) \setminus (\bigcup_{i=0}^{p-1} s''^i_k)$. Par application des lemmes 64 et 54, $x \in (\bigcup_{i=0}^{p-1} s'^i_j) \setminus (\bigcup_{i=0}^{p-1} s''^i_j)$. On a donc que $x \in (\bigcup_{i=0}^{p-1} s'^i_j) \Rightarrow x \in s'_i \Rightarrow s^1_i \setminus s^2_i \Rightarrow x \in S_1$. Maintenant prouvons que $x \notin S^2$. Si $x \in S^2$ alors par hypothèse $\exists k$ tel que $x \in s^2_k$. Nous avons alors deux cas :

- si $k = i$ alors $x \notin s^1_i \setminus s^2_i \Rightarrow x \notin s'_i$. contradiction
- si $k \neq i$ alors $x \in s^2_k \setminus s^1_k \Rightarrow x \in s''_k$. Par définition de la distribution, $x \in s''^k_j \Rightarrow x \notin (\bigcup_{i=0}^{p-1} s'^i_j) \setminus (\bigcup_{i=0}^{p-1} s''^i_j)$. contradiction

Donc $x \notin S^2$ et par conséquent $x \in S^1 \setminus S^2 = S$.

Prop. 3) Si $x \in (\bigcup_{i=0}^{p-1} s_j'^i) \setminus (\bigcup_{i=0}^{p-1} s_j''^i)$ alors par hypothèse de la redistribution $\forall k\ k \neq j \forall i\ x \notin s_k'^i$ et donc $x \notin (\bigcup_{i=0}^{p-1} s_k'^i) \setminus (\bigcup_{i=0}^{p-1} s_k''^i)$.

```
type transfer = {how_many:int; origin: int; destination: int}
type message = {n: int; to_where: int}

let rebalance_tmp pardata_to_datas take_from_datastructure union datastructure_of_list datas_to_pardata histo ps =
 let ps = pardata_to_datas ps in
 let histo_arrays = parfun Array.of_list histo in
 (* le calcul des tailles, origines et destinations de tous les transferts nécessaires au re-balancement à partir de l' histogramme *)
 let all_transfers histo =
  let size = Array.fold_left (+) 0 histo in
  let deltas = Array.map (fun n ->n - size/(bsp_p())) histo
  and transfers = ref [] in
   (for i=0 to (bsp_p())-1 do
      for j=0 to (bsp_p())-1 do
       if (deltas.(i) > 0) & (deltas.(j) <= 0)
        then let transfer_size = maxint (minint deltas.(i) (-deltas.(j))) 1 in
             transfers:={how_many=transfer_size; origin=i; destination=j}::!transfers;
             deltas.(i) <- deltas.(i) - transfer_size;
             deltas.(j) <- deltas.(j) + transfer_size
        else
         if (deltas.(i) <= 0) & (deltas.(j) > 0) then
          let transfer_size = maxint (minint (-deltas.(i)) deltas.(j)) 1 in
           transfers:={how_many=transfer_size; origin=j; destination=i}::!transfers;
           deltas.(i) <- deltas.(i) + transfer_size;
           deltas.(j) <- deltas.(j) - transfer_size
        else ()
      done
     done;
    !transfers) in
(* Le calcul des messages à émettre depuis le processeur from_pid durant le re-balancement. Le résultat est en
   chaque processeur, un tableau d' entiers, où le j-ème décrit le nombre de messages à envoyer au processeur j. *)
 let messages from_pid histo =
  let all_transf = all_transfers histo
  and how_many_to = Array.make (bsp_p()) 0 in
   (List.iter (fun {how_many=x; origin=i; destination=j} ->
                  if i=from_pid then how_many_to.(j) <- how_many_to.(j) + x else ()) all_transf;
    Array.mapi (fun i x ->{n=x; to_where=i}) how_many_to) in
(* Chaque processeur enlève ses données locales pour les envoyer *)
 let msgs = apply (mkpar messages) histo_arrays
 and msg_lists_and_new_trees = (fun msg_array t ->
          let tr = ref t and msg_list = ref [] in
           for j=0 to ((bsp_p())-1) do
            let (values, new_t) = take_from_datastructure (msg_array.(j)).n !tr in
             tr := new_t;
             msg_list := ((msg_array.(j)).to_where,values)::!msg_list
           done;
           (!tr,!msg_list)) in
let reduced_trees_and_msg_lists = parfun2 msg_lists_and_new_trees msgs ps in
let reduced_trees = parfun fst reduced_trees_and_msg_lists
and msg_lists = parfun snd reduced_trees_and_msg_lists in
let received_value_lists = parfun DataTools.concat_tail (put_list msg_lists) in
(* Expression principale de la fonction de re-balancement *)
 datas_to_pardata (parfun2 (fun l t ->union (datastructure_of_list l) t) received_value_lists reduced_trees)

(* val detect_unbalance: int->int list->bool *)
let detect_unbalance unbalanced l = List.exists (fun n ->List.exists (fun m ->abs(m-n)>unbalanced) l) l

let rebalance_if_needed pardata_to_datas take_from_datastructure union datastructure_of_list
                        datas_to_pardata if_nobalance peraps_need_balance unbalanced histogram pdata =
 if (peraps_need_balance pdata) then
  let histo_lists = rpl_total_exchange (histogram pdata) in
   if (detect_unbalance unbalanced histo_lists)
    then (rebalance_tmp pardata_to_datas take_from_datastructure union datastructure_of_list
                        datas_to_pardata (replicate histo_lists) pdata)
   else (if_nobalance pdata)
 else pdata
```

Figure 8.9 — Re-balancement d'un vecteur de données

9 Mémoires externes en BSML

Une version préliminaire d'une partie de ce chapitre a fait l'objet des articles [C2] et [R1]. Un premier travail sur des traits non fonctionnels en BSML a été préalablement publié, en collaboration avec Frédéric Loulergue, dans [C4].

Sommaire

POUR des applications à grande échelle où le traitement parallèle est utile et où la quantité de données à manipuler excède souvent la somme totale de toutes les mémoires principales (aussi appelées mémoires vives ou RAM) des machines disponibles, l'utilisation de disques externes, appelés aussi mémoires secondaires (ou externes), devient une nécessité. Dans ce chapitre, nous présentons une bibliothèque de dispositifs d'entrées/sorties (E/S) pour BSML sur les potentiels disques parallèles d'une machine BSP. Comme à l'accoutumée, nous donnerons une sémantique dynamique formelle ainsi qu'un modèle de coût pour ces accès aux disques. Sont également donnés quelques résultats expérimentaux de l'exécution d'un programme sur notre grappe de tests.

9.1 Introduction

Pour remédier aux problème des grandes quantités de données à traiter, une première solution pour augmenter la capacité de mémoire disponible est d'employer le mécanisme de *mémoire virtuelle* des systèmes d'exploitation modernes. Ce mécanisme de pagination a été établi comme une méthode standard pour contrôler et fournir, *via* les mémoires externes, une plus grande capacité de mémoire qui est nécessaire aux applications. Son principal avantage est de permettre aux applications de disposer d'un grand espace de mémoire virtuelle, sans avoir à tenir compte de la gestion de ces disques.

Malheureusement, cette solution est inefficace pour la pratique du calcul scientifique et ce quel que soit le système de pagination utilisé [60]. Les algorithmes et les structures de données existants sont souvent mal adaptés aux applications *out-of-core*. Pour obtenir de meilleurs temps d'exécution, ce genre d'application doit alors être restructuré avec des algorithmes comportant explicitement des entrées/sorties sur les données

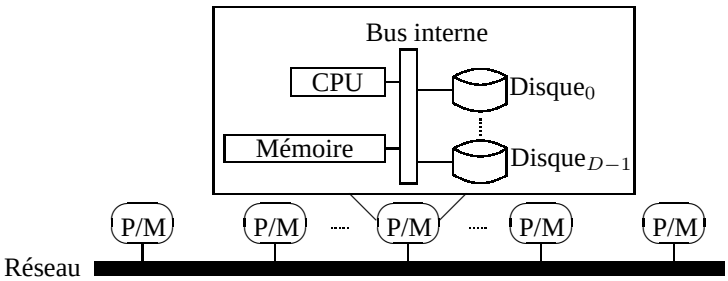


Figure 9.1 — Une machine BSP avec de la mémoire secondaire

des mémoires externes. De tels algorithmes sont généralement appelés «à mémoires externes» (*external memory algorithms* dans la litérature anglo-saxonne) et sont donc conçus pour les problèmes informatiques dans lesquels la taille de la mémoire principale de l'ordinateur (parallèle ou non) est seulement une petite fraction de la taille du problème [267, 268, 269].

Cette astreinte d'une réécriture complète des algorithmes est en grande partie due au besoin de localité des données et n'est pas nécessaire pour les algorithmes séquentiels en mémoire RAM ou les algorithmes parallèles PRAM (où chaque accès à une donnée a un coût constant). Ce n'est plus le cas dans un environnement à mémoires secondaires, où le coût d'accès à une donnée peut être bien plus important si celle-ci se trouve non pas dans la mémoire interne mais dans l'une des mémoires externes. Une application accroît alors ses performances si elle diminue ce nombre d'accès : elle ne doit garder dans sa mémoire principale que ce qui lui est nécessaire sur le moment.

La recherche d'algorithmes efficaces utilisant les mémoires secondaires a récemment suscité une attention considérable. Au cours de ces dernières années, des modèles complets de calcul et de coût qui incorporent des disques et des processeurs multiples ont été proposés [181, 267, 269], mais rarement avec tous les éléments décrits ci-dessus. Dans les articles [91, 93], les auteurs ont proposé un modèle abstrait, à la BSP, pour les machines parallèles incorporant des disques parallèles. Ce modèle est basé sur le modèle BSP où chaque machine a accès à une mémoire externe, sous la forme d'un ensemble de disques. Cette mémoire externe est alors manipulée par des opérations d'entrées/sorties.

Notre recherche visant à combiner le modèle BSP avec la programmation fonctionnelle, nous devons naturellement prolonger également notre langage avec des opérations d'entrées/sorties pour rendre possible la programmation de ce type d'algorithmes.

Ce chapitre est la suite directe de notre travail sur les dispositifs impératifs de BSML. Il est organisé comme suit : tout d'abord, nous présenterons brièvement le modèle BSP avec mémoires externes comme il est décrit dans [91]. Ensuite, ensuite nous expliciterons les problèmes qui apparaissent en BSML lorsque l'on veut naïvement ajouter les opérations classiques d'entrées/sorties d'OCaml. Nous donnerons alors les nouvelles primitives de notre langage, ainsi qu'une sémantique dynamique et un modèle de coût associés à cette extension. Nous terminerons avec un exemple complet utilisant ces primitives.

9.2 Mémoires externes

9.2.1 Modèle EM-BSP

Les ordinateurs modernes comportent typiquement plusieurs niveaux de mémoire tels que la mémoire principale (appelé couramment mémoire vive ou RAM), les caches (ou tampons) systèmes du processeur et des périphériques ainsi que la mémoire externe sous la forme de disques durs. Ce grand nombre de niveaux rend la modélisation complète d'un ordinateur bien trop compliquée et de tels modèles auraient l'inconvénient de ne pas pouvoir être portables. Nous nous limitons donc à un modèle à deux niveaux (mémoire principale et mémoires externes), tel qu'il est décrit dans [269], car la différence de rapidité d'accès entre les disques et la mémoire principale est bien plus significative qu'avec les autres niveaux. Sur ce principe, [93] a étendu le modèle BSP pour inclure des mémoires externes locales (appelées aussi mémoires secondaires dans un modèle à deux niveaux). La figure 9.1 illustre cette idée. Chaque processeur de la machine BSP, a, en plus de sa mémoire principale, une mémoire externe (EM) sous la forme d'un ensemble de disques. Cette idée étend le modèle BSP en sa version avec mémoire externe, appelée **EM-BSP**, avec les paramètres suivants :

1. M est la taille en octets de la mémoire principale en chaque processeur,
2. D est le nombre de disques en chaque processeur,
3. B est la taille en octets d'un bloc de transfert en chaque disque,
4. G est le ratio de capacité de calcul local (nombre d'opérations de calculs locaux) divisé par la capacité locale d'entrées/sorties (nombre de blocs de taille B pouvant être tranférés entre les disques et la mémoire principale) par unité de temps.

Dans de nombreuses machines parallèles, tous les ordinateurs composant la machine BSP ont bien le même nombre de disques. Ce modèle, dans l'esprit d'uniformité du modèle BSP, se restreint à ce cas. Notons aussi que le modèle interdit différentes tailles de mémoires principales.

Chaque disque est noté par $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$ et est constitué d'une séquence de plages (*tracks* en anglais). Chacune de ces plages est en accès direct et contient exactement un bloc de B octets. Chaque processeur peut utiliser, de manière concurrente, tous ces D disques et ainsi transférer depuis ou vers la mémoire principale $D \times B$ octets en une seule opération d'entrée/sortie. Cette opération aura alors un coût constant G. Dans une telle opération, seule une plage par disque est accessible. Il n'est pas spécifié quelle est la plage utilisée en chaque disque. Notons qu'une opération impliquant moins de disques n'entraîne pas l'utilisation d'une autre constante que G : toute opération d'écriture ou de lecture sur les disques se fera en un temps constant G. Notons aussi que chaque ordinateur composant la machine BSP est capable de contenir plusieurs blocs provenant des disques, c'est-à-dire $M \ggg DB$.

Comme dans le modèle BSP, un programme EM-BSP est structuré en une succession de super-étapes. Le coût des communications est identique. Le modèle EM-BSP autorise l'utilisation de plusieurs opérations d'entrée/sortie sur les disques lors de la phase de calcul d'une super-étape. Le coût total de chaque super-étape est donc défini ainsi :

$$t_{comp,E/S} + t_{comm} + l$$

où $t_{comp,E/S}$ est le coût des calculs locaux et des opérations d'E/S durant les super-étapes, c'est-à-dire $t_{comp,E/S} = \sum_s \max_i(w_i^s + m_i^s)$ où m_i^s est le temps de calcul utilisant la mémoire secondaire au processeur i pendant la super-étape s. La figure 9.2 donne le coût EM-BSP d'algorithmes BSP classiques en utilisant le résultat de simulation des algorithmes BSP dans le modèle EM-BSP décrit dans [93]. Le modèle séquentiel utilisant plusieurs disques est celui de [269] où les paramètres des disques sont ceux du modèle EM-BSP, mais pour un seul et unique processeur.

9.2.2 Présentation d'algorithmes existant en mémoires externes

Notre premier exemple est l'inversion d'une matrice. Celle-ci est largement employée dans les applications scientifiques comme méthode directe pour résoudre les systèmes *linéaires*. Le calcul de l'inverse d'une matrice A peut être dérivé de sa factorisation dite LU. [61] présente, à cet effet, la factorisation LU par blocs. Pour cet algorithme parallèle de factorisation, la matrice est divisée en blocs de colonnes appelés *super-blocs*. La taille du super-bloc est déterminée par la quantité de mémoire physique disponible dans la mémoire principale : seuls les blocs du super-bloc courant sont dans cette mémoire centrale, les autres sont conservés sur les disques. L'algorithme factorise la matrice de gauche à droite, super-bloc par super-bloc. Chaque fois qu'un nouveau super-bloc de la matrice est mis dans la mémoire principale (appelée le super-bloc *actif*), tous les pivotements précédents et les mises à jour d'après un historique de l'algorithme sont appliquées au super-bloc actif. Une fois que le dernier super-bloc est factorisé, la matrice est relue pour appliquer le pivotement restant sur les précédents super-blocs. Notons que le calcul est fait «*data in place*», c'est-à-dire sans aucun échange des données de la matrice mais par des échanges des pivotements effectués. La matrice a donc tout d'abord été distribuée sur les processeurs. Pour un bon équilibrage des charges, une distribution cyclique des données est employée.

L'article [66] présente des algorithmes PRAM utilisant une mémoire externe centralisée pour des problèmes sur les graphes tels que le calcul des composants bi-connexes. L'un d'eux est le problème des 4 couleurs d'un graphe appliqué au problème dit du *list ranking* : déterminer pour chaque nœud v d'une liste, le rang de ce nœud défini par le nombre de liens depuis v jusqu'à la fin de la liste. La méthode utilisée pour résoudre un tel problème est de mettre à jour des groupes de nœuds de la liste (en recolorisant les nœuds du groupe) sans avoir à trier ou parcourir la liste entière. Comme avant, l'algorithme travaille groupe par groupe, avec seulement un groupe dans la mémoire centrale.

Problème	Coût séquentiel avec E/S (un processeur, plusieurs disques)	Coût BSP (multiples processeurs, pas de disque)	Coût EM-BSP (multiples processeurs, multiples disques)
Tri	$\Theta(G \times \frac{n}{BD} \log_{M/B}(\frac{n}{B}))$	$L = l$ $t_{comp} = O(\frac{n \log(n)}{p})$ $H = O(\frac{n}{p})$	$t_{comp} = \tilde{O}(\frac{n \log(n)}{p})$ $t_{comm} = \tilde{O}(g(\frac{n}{p} + \log(\frac{n}{pB})))$ $t_{E/S} = \tilde{O}(G\frac{n}{pBD})$ $L = \tilde{O}(l \times D \times \log(\frac{n}{pB}))$
Permutation	$\Theta(G \times \min(\frac{n}{D}, \frac{n}{DB} \log_{M/B}(\frac{n}{B})))$	**idem**	**idem**
Transposé d'une matrice (avec r lignes, c colonnes et $n = r \times c$)	$\Theta(G \times \frac{n}{BD} \times \frac{\log(\min(M, r, c, \frac{n}{B}))}{\log(\frac{M}{B})})$	**idem**	**idem**

Figure 9.2 — Exemples de coûts de quelques algorithmes EM-BSP

Le dernier exemple est le problème de la recherche de motifs qui consiste à déterminer laquelle des k chaînes de caractères données est «sous-chaîne» d'une première chaîne de caractères. D'importantes applications, notamment en biologie, utilisent de très grandes chaînes de caractères (comme le génome d'un être vivant) et requièrent des algorithmes performants de recherches de motifs. [104] décrit un algorithme pour ce problème, avec un nombre constant de super-étapes, basé sur la distribution d'une structure de données appropriée sur les processeurs et les disques durs pour réduire et équilibrer le coût des communications d'une recherche naïve. Cette structure de données est basée sur la construction d'arbres équilibrés à partir des suffixes des chaînes de caractères. L'algorithme travaille sur le plus long préfixe commun de ces arbres et ceci par ordre lexicographique. On tire profit des disques en maintenant seulement une partie des arbres dans la mémoire principale et en rassemblant les parties des arbres pendant les super-étapes.

9.3 Mémoires externes en BSML

9.3.1 Rappel des E/S en OCaml

Les fonctions d'entrées/sorties de OCaml calculent une valeur, mais durant ce calcul, elles effectuent une modification de l'état de la mémoire externe. Deux types sont prédéfinis : in_channel et out_channel pour respectivement les *canaux de communication* d'entrées et de sorties. Les canaux peuvent être vus comme des «pointeurs» sur les fichiers. La création d'un canal utilise une des fonctions suivantes :

open_in: string→in_channel et open_out: string→out_channel

open_in ouvre un fichier en lecture s'il existe (déclenche une exception sinon) et open_out crée le fichier indiqué s'il n'existe pas ou l'écrase sinon (supprime son contenu pour permettre l'écriture de nouvelles données). Les fonctions de fermetures des canaux sont :

close_in: in_channel→unit et close_out: out_channel→unit

La bibliothèque standard de OCaml propose ensuite un grand nombre de fonctions de manipulation des canaux, notamment de manipuler les fichiers «à la Unix», caractère par caractère[1]. Les fonctions de plus «haut niveau» (qui donc nous intéressent), permettent l'écriture ou la lecture de n'importe quel type de données[2]. Ces fonctions sont :

to_channel: out_channel→α →unit et from_channel: in_channel→α

La fonction to_channel prend en argument un canal de sortie, une valeur et l'écrit sur le fichier pointé par le canal. Cette valeur a été préalablement sérialisée (on parle aussi de linéarisation, *serialize* dans la littérature anglo-saxonne) afin d'être transformée en une séquence d'octets. Réciproquement, la fonction from_channel lit sur le canal une valeur et la retourne. Plusieurs valeurs peuvent être stockées dans un même fichier. Elles pourront être lues séquentiellement.

9.3.2 Problèmes des E/S OCaml en BSML

Faisons l'hypothèse que chaque nœud de la machine parallèle possède un (ou plusieurs) disques. Le problème principal en ajoutant naïvement de la mémoire externe, et par conséquent des opérateurs d'E/S en BSML, est de conserver le fait que, dans le contexte global, les valeurs répliquées, c'est-à-dire les valeurs habituelles d'OCaml dupliquées en chaque processeur, soient identiques. De telles valeurs sont consacrées au contrôle global des algorithmes parallèles. Prenons par exemple l'expression suivante (nous faisons l'hypothèse que chaque nœud possède un fichier "file.dat") :

```
let chan=open_in "file.dat" in
 if (at (mkpar(fun pid→(pid mod 2)=0)) (input_value chan))
  then scan_direct (+) 0 (replicate 1)
 else (replicate 1)
```

Il n'est (sûrement) pas exact que le fichier file.dat contienne en chaque processeur la même valeur. Dans ce cas-ci, chaque processeur peut lire sur sa mémoire secondaire une valeur différente. Nous obtenons une évaluation incohérente de l'expression car chaque processeur peut lire un entier différent sur le canal (*channel* en anglais) chan. Ainsi, certains d'entre eux voudront exécuter l'expression scan_direct, qui

[1] Notons que ces fonctions ne sont pas alors toutes portables.

[2] Il existe quelques exceptions que nous ne donnerons pas ; le manuel de référence de OCaml les donne.

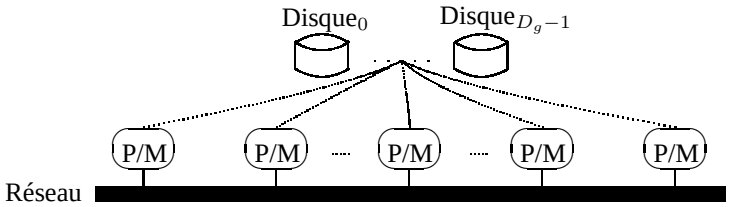


Figure 9.3 — Une machine BSP avec des disques partagés

nécessite une synchronisation globale, tandis que d'autres voudront exécuter la fonction replicate qui ne nécessite pas une synchronisation (calcul asynchrone). Ceci rompt le modèle d'exécution BSP avec ses synchronisations globales. Si cette expression avait été évaluée avec la bibliothèque BSMLlib, nous aurions eu une erreur d'exécution (ou pire, un inter-blocage des processeurs) de la machine de BSP car **at** est une fonction appelant **proj** qui est une primitive globale synchrone. Notons que nous avons également ce genre de problème dans la bibliothèque BSPlib [147] où les auteurs notent que seules les opérations d'E/S du premier processeur sont sûres.

Un autre problème vient des *effets de bord* qui peuvent se produire en un processeur. Prenons par exemple l'expression suivante :

```
try
 let a=mkpar(fun i→if i=0 then (open_in "file.dat");() else ())
  in (open_out "file.dat"); (replicate 1)
with _ →scan_direct (+) 0 (replicate 1)
```

Si cette expression avait été évaluée avec la bibliothèque BSMLlib, seul le premier processeur aurait ouvert le fichier file.dat en mode lecture. Ensuite, tous les processeurs auraient ouvert le fichier file.dat mais cette fois-ci en écriture, excepté le premier, celui-ci ayant déjà ouvert ce fichier. Nous aurions également eu une évaluation incohérente de l'expression car seul le premier processeur aurait soulevé une exception dans le contexte global. Ainsi seul ce premier processeur aurait évalué le scan_direct ce qui, comme précédemment, aurait entraîné une erreur d'exécution de la machine BSP.

Ce problème des effets de bord peut également être combiné avec le problème susmentionné, même s'il n'y a aucun fichier dans la mémoire secondaire en début du calcul. Prenons par exemple l'expression suivante :

```
let chan=open_out "file.dat" in
let x=mkpar(fun i→if i=0 then (ouput_value 0) else ()) in
 ouput_value 1; close cha;
 let chan=open_in "file.dat" in
  if (at (mkpar(fun pid→(pid mod 2)=0)) (input_value chan))
   then scan_direct (+) 0 (replicate 1)
  else (replicate 1)
```

Le premier processeur ajoute les entiers 1 et 2 dans le fichier file.dat. Les autres processeurs n'ajoutent que l'entier 2 sur leurs fichiers. Comme dans l'exemple précédent, nous aurons une panne de la machine BSP car une fois encore, l'entier lu n'est pas le même en chaque processeur dans le contexte global.

9.3.3 Solution proposée

La solution proposée dans ce manuscrit est d'avoir deux genres de fichier : les globaux et les locaux. De cette façon, nous avons deux types d'opérations d'E/S. Les opérateurs locaux d'E/S ne doivent pas être évalués dans le contexte global et les opérateurs globaux d'E/S dans un contexte local (c'est-à-dire dans un vecteur parallèle).

Les fichiers locaux sont dans des systèmes locaux de fichiers qui sont présents sur toutes les composantes de la machine BSP comme dans le modèle EM-BSP. Les fichiers globaux sont dans un système global de fichiers. Ces fichiers doivent être les mêmes du point de vue de chaque nœud. Le système global de fichiers peut donc être situé sur des disques partagés (comme dans la figure 9.3) ou comme une copie en chaque processeur. Ces fichiers globaux donneront ainsi toujours les mêmes valeurs dans le contexte global. Notons que si seuls des disques partagés sont présents (pas de disques locaux), ce qui peut être le cas de machines

parallèles à la mémoire principale partagée (comme des PC bi-processeurs par exemple), les systèmes de fichiers locaux peuvent parfaitement être dans différents répertoires, un par processeur.

Avoir des disques partagés est avantageux pour le cas des programmes qui n'ont pas leurs données distribuées au début du calcul. Par exemple, la recherche de motifs ou un algorithme de tri où les données de départ sont initialement dans un (ou des) fichier(s). On est aussi en droit d'attendre le résultat final dans un fichier et pas dans différents fichiers, distribués sur les processeurs. Dans le cas d'un système de fichiers globaux répartis, les données globales sont également distribuées et les programmes sont alors moins sensibles aux différents problèmes de pannes d'une machine.

Ainsi, nous avons deux cas importants pour le système global de fichiers. C'est donc un nouveau paramètre du modèle : avons-nous des disques partagés ou non ?

Nous verrons par la suite que dans le premier cas (présence de disques partagés), la condition que les fichiers globaux soient les mêmes du point de vue de chaque processeur va nécessiter des synchronisations pour certaines des opérations globales d'E/S, comme créer, ouvrir ou supprimer un fichier. Par exemple, il est impossible (ou plus exactement non-déterministe) pour un processeur de créer un fichier dans le système global de fichiers si, au «même moment», un autre processeur le supprime. Par contre, lire (resp. écrire) des valeurs à partir (resp. sur) des fichiers ne nécessitera pas de synchronisation : tous les processeurs liront les mêmes valeurs dans un fichier global et seul un des processeurs devra réellement écrire la valeur sur les disques partagés. Notons que ceci permet aussi d'éviter les problèmes des goulots d'étranglement des disques partagés («*bottleneck of the shared disk*» dans la littérature anglo-saxonne) car, dans le cas d'une opération globale d'écriture, seul un des processeurs écrit la valeur, et dans le cas d'une opération globale de lecture, la valeur est d'abord lue sur les disques par un des processeurs, puis sur les caches du système d'exploitation (ou des périphériques) par les autres processeurs.

Dans le deuxième cas (absence de disques partagés), tous les fichiers, locaux et globaux, sont distribués et aucune synchronisation n'est donc nécessaire, chaque processeur lisant/écrivant/etc. dans son propre système de fichiers. Mais au début, les système de fichiers globaux devront être vides ou répliqués en chaque processeur.

Notons qu'un grand nombre de machines parallèles modernes ont des disques partagés concurrents. De tels disques sont généralement considérés comme des disques utilisateurs, c'est-à-dire des disques où les utilisateurs mettent les données requises pour les calculs tandis que des disques locaux sont seulement employés pour les calculs parallèles des programmes. Par exemple, le *earth simulator*[3] propose 1,5 Pentabytes aux utilisateurs sous la forme, dixit la page web, de «disques de mémoire de masse». Un réseau spécial y a été installé pour accéder à ces disques. Notons aussi que s'il n'y a aucun disque concurrent partagé, le système NFS ou des bibliothèques de niveau bas (dont le fonctionnement de certaines est décrit dans [182]) sont en mesure de simuler de tels disques.

9.3.4 Nouveau modèle, le modèle EM-BSP[2]

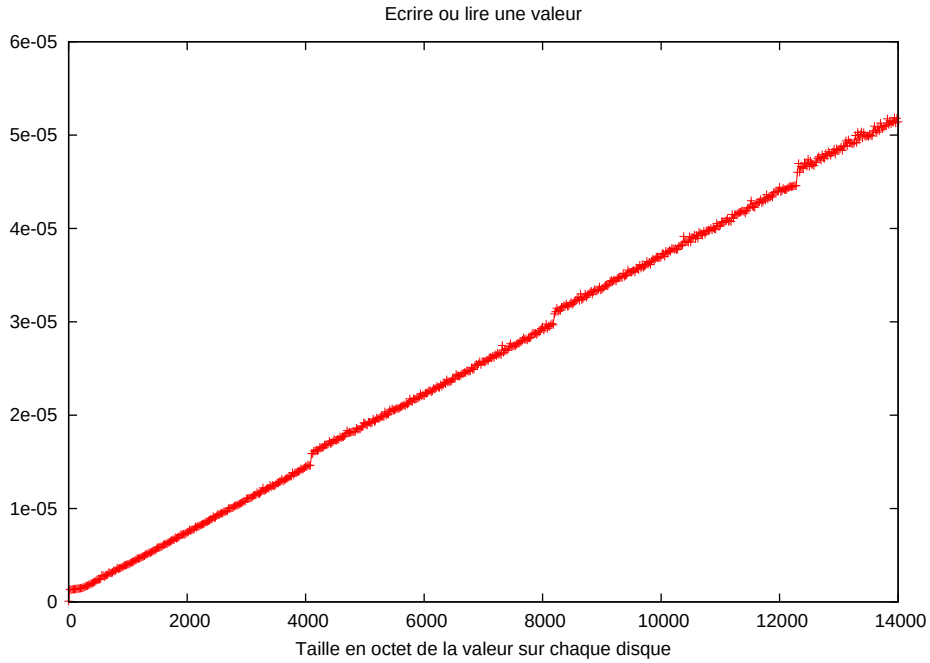
Après quelques expériences pour déterminer les paramètres **EM-BSP** de notre cluster de test, nous avons constaté que les systèmes d'exploitation ne permettaient pas l'écriture ou la lecture des données en un temps constant mais plutôt en un temps linéaire, proportionnel à la taille des données. Nous remarquons également qu'il existe un surplus de temps dépendant de la taille des blocs. Nous avons :

$$n \times (DB) < s < (n+1) \times DB$$

où s est la taille en octets de la donnée. Nous constatons $n+1$ surcoûts (*overhead* en anglais) de temps pour lire ou écrire cette donnée sur les D disques concurrents. La figure 9.4 donne les résultats de cette expérience sur un PC utilisant 3 disques, chaque disque étant paramétré par le système d'exploitation par des blocs de 4096 octets (les secondes sont tracées sur l'axe vertical). Ce test a été exécuté 10000 fois et la moyenne a été prise. Les résultats ne sont pas foncièrement différents si nous diminuons le nombre de disques.

La solution proposée donnait aux processeurs l'accès à deux genres de fichier : les globaux et les locaux. De cette façon, le modèle, ici présenté et appelé EM^2-BSP , prolonge le modèle BSP en sa version explicitant deux genres de mémoires secondaires : les mémoires externes locales et les mémoires externes globales. Chaque système local de fichiers sera sur les disques concurrents locaux comme dans le modèle **EM-BSP**. Le système global de fichiers sera soit sur des disques partagés concurrents (comme dans la figure 9.3)

[3] Page web à `earth-simulator.org`.



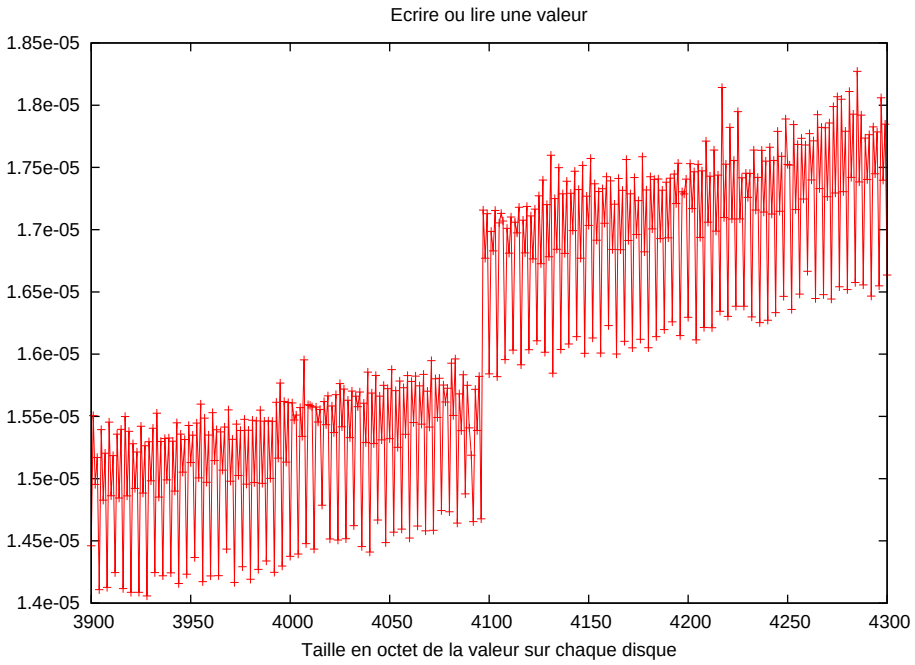


Figure 9.4 — Benchmarks des paramètres de la mémoire secondaire

s'ils existent, soit repliqué sur les disques locaux. Le modèle EM^2-BSP peut ainsi tenir compte des coûts de lecture et de distribution des données aux processeurs ainsi que de l'écriture des résultats finaux. Les paramètres suivants s'ajoutent de ce fait aux paramètres standards du modèle BSP :

1. M est la taille en octet de la mémoire principale en chaque processeur,
2. D^l est le nombre de disques locaux en chaque processeur,
3. B^l est la taille en octet d'un bloc de transfert en chaque disque local,
4. G^l est le coût pour écrire ou lire un octet sur chaque disque local,
5. O^l est le surcoût des disques concurrents locaux,
6. D^g est le nombre de disques globaux,
7. B^g est la taille en octet d'un bloc de transfert en chaque disque global,
8. G^g est le coût pour écrire ou lire un octet sur chaque disque global,
9. O^g est le surcoût des disques concurrents globaux,

Bien entendu, s'il n'y a aucun disque partagé ou pas de disques locaux, nous avons : $D^l = D^g$, $B^l = B^g$, $G^l = G^g$ et $O^l = O^g$. Un processeur est en mesure de lire ou d'écrire n octets sur ses disques locaux en un temps :

$$\lceil \frac{n}{D^l} \rceil \times G^l + \lceil \frac{n+1}{D^l B^l} \rceil \times O^l$$

et n octets sur les disques globaux en un temps :

$$\lceil \frac{n}{D^g} \rceil \times G^g + \lceil \frac{n+1}{D^g B^g} \rceil \times O^g$$

Comme dans le modèle **EM-BSP**, le calcul dans le modèle EM^2-BSP procède en une succession de super-étapes. Les coûts de communication sont identiques au modèle BSP et plusieurs opérations sur les disques locaux et globaux sont également permises pendant la phase de calcul d'une super-étape. Notons que G^g n'est pas g, même si les processeurs accèdent aux disques partagés *via* le même réseau (cas des machines parallèles comme les grappes de PCs) : g est le temps nécessaire pour exécuter une 1-relation tandis que G^g est le temps pour écrire ou lire D mots sur les disques partagés (disques globaux). Cette constante peut dépendre de g, dans le cas de certaines machines parallèles, mais aussi de beaucoup d'autres paramètres du matériel si, par exemple, il existe un réseau supplémentaire pour accéder aux disques concurrents partagés.

9.3.5 Nouvelles primitives

Dans cette section, nous décrivons le noyau de notre bibliothèque de primitives d'E/S, c'est-à-dire l'ensemble minimal de primitives nécessaires pour programmer des algorithmes EM^2-BSP. Cette bibliothèque a été incorporée dans la dernière BSMLlib en tant que nouveau module. Celui-ci est basé sur les éléments donnés par la figure 9.5.

Comme pour les paramètres BSP de la BSMLlib, nous avons des primitives pour accéder aux nouveaux paramètres EM^2-BSP de la machine parallèle. Par exemple, **embsp_loc_D**() est D^l le nombre de disques locaux et **glo_shared**() précise si le système global de fichiers est partagé ou non. Puisque nous avons deux systèmes de fichiers, nous avons besoin de deux types de noms et de types abstraits de canaux de sortie (resp. canaux d'entrée) : glo_out_channel (resp. glo_in_channel) et loc_out_channel (resp. loc_in_channel) pour l'écriture (resp. la lecture) de valeurs sur les fichiers locaux ou globaux.

Nous pouvons ouvrir un fichier en écriture. La primitive renverra un nouveau canal sur ce fichier. Ce dernier est écrasé s'il existe déjà. Sinon, il est créé, ou la primitive levera une exception si le fichier ne peut être ouvert. Nous avons deux types de primitives : l'un pour les fichiers globaux et l'autre pour les locaux : (**glo_open_out** f) ouvre le fichier global f en écriture et retourne un canal global pointant au début de ce fichier, tandis que (**loc_open_out** f) ouvre le fichier local f et retourne un canal local.

D'une manière similaire, nous avons deux primitives, **glo_open_in** et **loc_open_in**, permettant d'ouvrir un fichier en lecture. De telles primitives retournent de nouveaux canaux locaux (ou globaux) pointant au début des fichiers. Dans le cas de disques partagés globaux, une synchronisation se produit pour chaque

Paramètres EM^2-BSP :

embsp_loc_D:unit→int	**embsp_loc_B**:unit→int	**embsp_loc_G**:unit→float
embsp_glo_D:unit→int	**embsp_glo_B**:unit→int	**embsp_glo_G**:unit→float
embsp_loc_O:unit→float	**embsp_glo_O**:unit→float	**glo_shared**:unit→bool

Primitives globales d'E/S	Primitives Locales d'E/S :
glo_open_out:glo_name→glo_out_channel	**loc_open_out**: loc_name→loc_out_channel
glo_open_in:glo_name→glo_in_channel	**loc_open_in**:loc_name→loc_out_channel
glo_output_value:glo_out_channel→α →unit	**loc_output_value**:loc_out_channel→α →unit
glo_input_value:glo_in_channel→α option	**loc_input_value**:loc_in_channel→α option
glo_close_out:glo_out_channel→unit	**loc_close_out**:loc_out_channel→unit
glo_close_in:glo_in_channel→unit	**loc_close_in**:loc_in_channel→unit
glo_delete:glo_name→unit	**loc_delete**:loc_name→unit
glo_seek:glo_in_channel→int→unit	**loc_seek**:loc_in_channel→int→unit

Du local au global :

glo_copy:(int→loc_name∗glo_name option)→unit

Figure 9.5 — Primitives d'E/S pour la BSMLlib

«**open**» global. Avec cette synchronisation globale, chaque processeur peut signaler aux autres s'il est parvenu à ouvrir le fichier. Dans le cas contraire (si l'un d'entre eux n'a pas pu ouvrir le fichier, suite à une défaillance du système de fichiers comme une panne d'un disque), les processeurs lèvent une exception globale.

Avec nos canaux, nous pouvons lire ou écrire des valeurs sur les fichiers. Ce dispositif est appelé *persistance*, car la valeur écrite sur un fichier peut rester présente même après la fin du programme.

Pour écrire la représentation d'une valeur de n'importe quel type sur un canal (global ou local), nous utilisons les primitives suivantes : (**glo_output_value** chaglo v) qui écrit la valeur v sur le fichier global ouvert en écriture et pointé par chaglo et (**loc_output_value** chaloc v) qui écrit localement la valeur locale v sur le fichier local pointé par chaloc.

La valeur ainsi écrite peut alors être relue par les primitives de lecture : (**glo_input_value** chaglo) (resp. (**loc_input_value** chaloc)) retourne du canal global chaglo (resp. canal local chaloc) la valeur répliquée Some v (resp. valeur locale) ou None s'il n'y a plus de valeur dans le fichier global (resp. local). C'est la fin du fichier.

Les primitives d'écriture et de lecture lisent ou écrivent la représentation physique d'une valeur, c'est-à-dire la transformation de la valeur en une chaîne de caractères. La lecture de cette chaîne n'est pas sûre en OCaml [178, 112]. Par exemple, si le fichier contient un flottant et que le programme veut lire un entier, aucune exception n'est levée et s'il tente de manipuler ce flottant comme un entier, il peut entraîner une éventuelle erreur d'exécution. L'équipe de développement de OCaml travaille sur ce problème sans avoir recours à un typage dynamique.

Notre bibliothèque contient également la primitive **glo_seek** telle que (**glo_seek** chaglo n) (resp. **loc_seek**) permet de faire pointer le canal à la $n$ème valeur d'un fichier global (resp. local). Le comportement des primitives décrites ci-dessus est non spécifié si l'une d'entre elles est appelée sur le canal ayant été préalablement fermé.

Notons que seules des valeurs locales ou repliquées, c'est-à-dire des valeurs OCaml traditionelles, peuvent être écrites sur les fichiers locaux comme globaux. En effet, l'emboîtement des vecteurs parallèles est interdit et **loc_input_value** ne doit donc lire que des valeurs OCaml. Il est également impossible d'écrire sur un fichier global partagé un vecteur parallèle de valeurs (une valeur globale) car le contenu de cette valeur est différent en chaque processeur. **glo_output_value** étant une primitive asynchrone (voir la section 9.5 pour plus de détails), de telles valeurs pourraient être écrites dans n'importe quel ordre ce qui aurait pour conséquence un mélange de ces valeurs sur le fichier. C'est pourquoi seules des valeurs locales et/ou répliquées sont autorisées en lecture et en écriture.

Après la lecture et l'écriture des valeurs sur nos canaux, il nous faut les fermer, c'est-à-dire rendre impossible la lecture ou l'écriture sur les fichiers auxquels les canaux font référence. Comme précédemment, nous avons besoin de quatre genres de primitives : deux pour les canaux de lecture (les locaux et globaux) et deux pour les canaux d'écriture. Par exemple, (**glo_close_out** chaglo), ferme le canal global d'écriture

chaglo qui avait été créé par un **glo_open_out**. La primitive **glo_delete** (resp. **loc_delete**) supprime un fichier global (resp. local) s'il a été préalablement fermé, c'est-à-dire s'il n'y a plus un canal pointant sur le fichier et qui soit encore ouvert.

La dernière primitive copie des fichiers locaux sur le système global de fichiers. (**glo_copy** g) copie le fichier local floc du processeur n dans le système global de fichiers, sous le nom de fglo, si les conditions suivantes sont réunies : (g n) = Some (floc,fglo), floc existe (et est fermé) et fglo n'existe pas. Si (g n) = None, aucun fichier du processeur n n'est copié dans le système global de fichiers.

Cette primitive peut être utile à la fin d'un programme BSML quand il faut copier les résultats locaux, provenant de fichiers locaux, dans le système global de fichiers (système de fichiers utilisateurs). Notons que ce n'est pas une primitive de communication. Cette primitive pourrait être simulée par un **proj** et vice-versa, mais les coûts ne seraient alors plus ceux attendus et aucune optimisation utilisant des bibliothèques de bas niveau ne pourrait être mise en œuvre. En effet, si on simulait le **glo_copy** par un **proj**, dans le cas d'un système global partagé de fichiers, le fichier local serait préalablement diffusé, *via* le réseau, à tous les autres processeurs. Puis il serait copié sur le système global de fichiers. Cette phase de diffusion peut être évitée si **glo_copy** est une primitive. Dans le cas où **proj** est simulé par un **glo_copy**, les données envoyées par le **proj** sont tout d'abord copiées dans le système global de fichiers, puis relues par les autres processeurs. Dans le cadre d'un système réparti de fichiers, les données sont aussi mises sur le réseau pour être dupliquées sur tous les processeurs. Cette phase de lecture/écriture n'est pas effectuée si **proj** est une primitive.

Notons aussi qu'en n'utilisant que ces primitives pour les E/S, le résultat final d'un programme sera identique (mais naturellement pas avec les mêmes coûts) sur une architecture possédant ou non des disques partagés. Ces primitives permettent donc la portabilité des opérations d'E/S d'un programme BSML. Pour aider la compréhension du fonctionnement de ces nouvelles primitives et pouvoir raisonner sur les programmes BSML avec E/S, nous allons décrire dans la prochaine section, une sémantique formelle de BSML munie de ce dispositif pour la persistance des données.

9.4 Sémantique dynamique

Pour la sémantique dynamique, nous utilisons les mêmes notations que dans les chapitres précédents. Nous n'allons ajouter et décrire que ce qui est relatif aux traits persistants. Nous ne noterons pas non plus les coûts dans cette sémantique car cela la rendrait totalement illisible. Ceux-ci seront donnés séparément car leurs intégration dans la sémantique est basique.

Nous notons $\{f_i\}$ pour le système de fichiers du processeur i. Nous faisons l'hypothèse que chaque processeur a accès à un système de fichiers représenté comme une séquence potentiellement infinie de fichiers. Ces séquences sont différentes en chaque processeur. Nous notons $\{f\} = \{\{f_0\}, \ldots, \{f_{p-1}\}\}$ pour l'ensemble des systèmes locaux de fichiers de la machine parallèle et $\{\mathcal{F}\}$ pour le système global de fichiers.

L'ensemble des opérations primitives est étendu avec les primitives d'E/S : $\mathbf{open^r}$ (resp. $\mathbf{open^w}$) pour ouvrir un fichier en lecture (resp. écriture) comme un canal, $\mathbf{close^r}$ et $\mathbf{close^w}$ pour fermer un canal, $\mathbf{read}$, $\mathbf{write}$ pour lire ou écrire dans un canal, $\mathbf{delete}$ pour supprimer un fichier et $\mathbf{seek}$ pour changer de position dans un fichier. Toutes ces opérations seront distinguées par une étiquette qui est loc s'il s'agit d'un opérateur local, glo pour un opérateur global.

Définition 27 (*Fichiers et canaux*).
Nous avons aussi deux types de fichiers, les locaux et les globaux, définis comme suit :

- f *pour le nom d'un fichier ;*
- $f_{\mathbf{w}}$ *pour un canal d'écriture et* $f_{\mathbf{r}}$ *pour un canal de lecture ;*
- g_k^{ξ} *pour un pointeur de canal, pointant sur la* k*ème valeur d'un fichier tel que* ξ *est le nom du canal associé ;*
- ${}^{?}f\begin{bmatrix} v_n \\ \vdots \\ v_0 \end{bmatrix}$ *pour un fichier avec* $?$ *qui peut être* **c**, **r** *ou* **w** *pour un fichier fermé ou ouvert en lecture ou en écriture et tel que* $v_0, \ldots, v_n$ *sont les valeurs contenues par le fichier.*

Quand un fichier sera ouvert en lecture, il sera associé aux pointeurs $[g_n^a, \ldots, g_m^z]$ des canaux qui pointeront sur lui. Les positions de ces canaux seront aussi mémorisées.

Nous étendons les valeurs et les expression avec les noms des fichiers et les canaux. La version persistante de la sémantique à petits pas de BSML a la forme suivante : $\{\mathcal{F}\}/e/\{f\} \rightharpoonup \{\mathcal{F}'\}/e'/\{f'\}$. Comme à

l'accoutumée nous notons $\stackrel{*}{\rightharpoonup}$ pour la fermeture transitive et réflexive de $\rightharpoonup$, c'est-à-dire que nous notons $\{\mathcal{F}^0\}/e_0/\{f^0\} \stackrel{*}{\rightharpoonup} \{\mathcal{F}\}/v/\{f\}$ pour :

$$\{\mathcal{F}^0\}/e_0/\{f^0\} \rightharpoonup \{\mathcal{F}^1\}/e_1/\{f^1\} \rightharpoonup \{\mathcal{F}^2\}/e_2/\{f^2\} \rightharpoonup \ldots \rightharpoonup \{\mathcal{F}\}/v/\{f\}$$

Définition 28 (*Relations de la sémantique à «petits pas» avec persistance*).
Pour définir la relation $\rightharpoonup$, nous avons des règles pour deux types de réductions :

1. $e/\{f_i\} \stackrel{i}{\rightharpoonup} e'/\{f_i'\}$ *qui peut être lu comme «avec le système local de fichiers et initial $\{f_i\}$, au processeur i, l'expression e est réduite en e' avec le système global de fichier $\{f_i'\}$» ;*

2. $\{\mathcal{F}\}/e/\{f\} \stackrel{\bowtie}{\rightharpoonup} \{\mathcal{F}'\}/e'/\{f\}$ *qui peut être lu comme «avec le système global (et initial) de fichiers $\{\mathcal{F}\}$ et avec l'ensemble des systèmes locaux de fichiers $\{f\}$, l'expression e est réduite en e' avec le système global de fichier $\mathcal{F}'$ et le même ensemble de systèmes locaux de fichiers» ;*

avec les deux types de réductions définies de la manière suivante :

$$\stackrel{i}{\rightharpoonup} \;=\; \underset{i}{\overset{\varepsilon}{\rightharpoonup}} \cup \underset{\delta_i}{\rightharpoonup} \cup \underset{\delta_i}{\overset{io}{\rightharpoonup}} \qquad \textit{et} \qquad \stackrel{\bowtie}{\rightharpoonup} \;=\; \underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}} \cup \underset{\delta_\bowtie}{\rightharpoonup} \cup \underset{\delta_{\lightning}}{\rightharpoonup} \cup \underset{\delta_\bowtie}{\overset{io}{\rightharpoonup}}$$

Premièrement, pour définir ces réductions, nous commençons par quelques axiomes. Nous prenons, tout d'abord, ceux des réduction fonctionnelles $\overset{\varepsilon}{\rightharpoonup}$, c'est-à-dire, des substitution dans les expressions. Ces réductions sont étendues par deux versions. La première, $\underset{i}{\overset{\varepsilon}{\rightharpoonup}}$ est locale au processeur i tandis que la seconde $\underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}}$ est globale à tous les processeurs :

$$\frac{e \stackrel{\varepsilon}{\rightharpoonup} e' \quad \text{(donnée à la figure 9.6)}}{e \;/\; \{f_i\} \underset{i}{\overset{\varepsilon}{\rightharpoonup}} e' \;/\; \{f_i\}} \qquad \frac{e \stackrel{\varepsilon}{\rightharpoonup} e' \quad \text{(donnée à la figure 9.6)}}{\{\mathcal{F}\} \;/\; e \;/\; \{f\} \underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}} \{\mathcal{F}\} \;/\; e' \;/\; \{f\}}$$

Pour les opérations primitives, nous avons aussi besoin d'axiomes. Nous utilisons là encore les règles δ définies dans les précédents chapitres. Celles-ci seront notées $\underset{\delta}{\rightharpoonup}$. Comme précédemment, nous avons deux versions de ces réductions, l'une $\underset{\delta_i}{\rightharpoonup}$, locale au processeur i et l'autre, $\underset{\delta_\bowtie}{\rightharpoonup}$, globale à tous les processeurs :

$$\frac{e \underset{\delta}{\rightharpoonup} e' \quad \text{(donnée à la figure 9.6)}}{e \;/\; \{f_i\} \underset{\delta_i}{\rightharpoonup} e' \;/\; \{f_i\}} \qquad \frac{e \underset{\delta}{\rightharpoonup} e' \quad \text{(donnée à la figure 9.6)}}{\{\mathcal{F}\} \;/\; e \;/\; \{f\} \underset{\delta_\bowtie}{\rightharpoonup} \{\mathcal{F}\} \;/\; e' \;/\; \{f\}}$$

De telles réductions, qui ne sont pas des réductions avec de la persistance, n'ont pas besoin des systèmes de fichiers. Seules les primitives d'E/S vont les modifier.

Ensuite, pour les primitives parallèles, nous avons naturellement des règles de réductions, $\underset{\delta_{\lightning}}{\rightharpoonup}$, mais celles-ci sont similaires à celles des chapitres précédents. Il faut juste ajouter le fait que, pour des raisons évidentes, il n'est pas possible qu'un processeur puisse envoyer un canal à un autre processeur. Celui-ci n'a pas à lire (ou écrire) sur ce canal, car ce serait une communication cachée et non prise en compte par le modèle BSP. Ainsi, nous devons tester si les valeurs envoyées contiennent ou non des canaux. Pour cela, nous utilisons la fonction $\mathcal{A}_c$ qui parcourt inductivement et trivialement la valeur pour savoir si celle-ci contient des canaux. Notons que ce travail est effectué par OCaml quand il linéarise une valeur. La fonction $\mathcal{A}_c$ est appliquée à toutes les valeurs calculées par le send. Cette fonction indique aussi, de la même manière, si une valeur contient ou non un vecteur parallèle. Cela nous permet d'éviter d'écrire sur un fichier une valeur parallèle que nous serions incapables d'écrire du fait de son caractère asynchrone.

Enfin, nous complétons notre sémantique en donnant les règles $\delta \overset{io}{\rightharpoonup} \delta$ de nos primitives d'E/S. Celles-ci sont données à la figure 9.7. Détaillons leur fonctionnement :

- Les règles 9.1 et 9.2 donnent l'ouverture d'un fichier en lecture. Deux cas se posent :
  1. Le fichier est fermé (règle 9.1). Le fichier est alors ouvert en lecture et nous obtenons un nouveau canal pointant au début du fichier ;
  2. Le fichier a déjà été ouvert en lecture (règle 9.2). Nous obtenons alors un nouveau canal pointant au début du fichier et qui est ajouté aux autres canaux ;
- Les règles 9.3 et 9.4 donnent l'ouverture d'un fichier en écriture. Deux cas se posent :

$$
\begin{array}{rcl}
(\overline{(\lambda.e)[s]}\ v) & \xrightarrow{\varepsilon} & e[v \circ s] \\
\overline{n+1}[v \circ s] & \xrightarrow{\varepsilon} & \overline{n}[s] \\
\overline{1}[v \circ s] & \xrightarrow{\varepsilon} & v[\bullet] \\
(\mu.e)[s] & \xrightarrow{\varepsilon} & \overline{e[\mu.e \circ s]} \\
(\lambda.e)[s] & \xrightarrow{\varepsilon} & \overline{(\lambda.e)[s]} \\
v[s] & \xrightarrow{\varepsilon} & v \quad \text{si } v \neq \langle \cdots \rangle \text{ et } v \neq (v_0, v_1) \\
(e_1\ e_2)[s] & \xrightarrow{\varepsilon} & (e_1[s]\ e_2[s]) \\
(e_1, e_2)[s] & \xrightarrow{\varepsilon} & (e_1[s], e_2[s]) \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)[s] & \xrightarrow{\varepsilon} & \textbf{if } e_1[s] \textbf{ then } e_2[s] \textbf{ else } e_3[s] \\
(\textbf{proj}\ e)[s] & \xrightarrow{\varepsilon} & (\textbf{proj}\ e[s]) \\
(\textbf{put}\ e)[s] & \xrightarrow{\varepsilon} & (\textbf{put}\ e[s]) \\
(\textbf{apply}\ e_1\ e_2)[s] & \xrightarrow{\varepsilon} & (\textbf{apply}\ e_1[s]\ e_2[s]) \\
(\textbf{mkpar}\ e)[s] & \xrightarrow{\varepsilon} & (\textbf{mkpar}\ e[s]) \\
\langle e_0, \ldots, e_{p-1} \rangle[s] & \xrightarrow{\varepsilon} & \langle e_0[s], \ldots, e_{p-1}[s] \rangle \\
[e_0, \ldots, e_{p-1}][s] & \xrightarrow{\varepsilon} & [e_0[s], \ldots, e_{p-1}[s]]
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{access}\ ([v_0, \ldots, v_i, \ldots, v_{p-1}], i) & \xrightarrow[\delta]{} & v_i \\
\textbf{init}\ (n, f) & \xrightarrow[\delta]{} & [(f\,0), \ldots, (f\,(n-1))] \\
\textbf{fst}\ (v_1, v_2) & \xrightarrow[\delta]{} & v_1 \\
\textbf{snd}\ (v_1, v_2) & \xrightarrow[\delta]{} & v_2 \\
+\ (n_1, n_2) & \xrightarrow[\delta]{} & n_1 + n_2 \\
\textbf{isnc nc} & \xrightarrow[\delta]{} & \textbf{true} \\
\textbf{isnc}\ v & \xrightarrow[\delta]{} & \textbf{false si } v \neq \textbf{nc} \\
\textbf{if true then } e_2 \textbf{ else } e_3 & \xrightarrow[\delta]{} & e_2 \\
\textbf{if false then } e_2 \textbf{ else } e_3 & \xrightarrow[\delta]{} & e_3
\end{array}
$$

Figure 9.6 — Réductions fonctionnelles et opérations génériques (rappel)

1. Le fichier existait déjà (règle 9.3). Le fichier est alors ouvert en écriture et les anciennes données sont supprimées (le fichier est écrasé). Un canal pointant au début du fichier est retourné ;
2. Le fichier n'existait pas (règle 9.4). Il est alors créé. Un canal pointant au début de ce nouveau fichier est retourné ;

- Les règles 9.5, 9.6, 9.7 et 9.8 donnent la fermeture d'un fichier ouvert en lecture. Quatre cas se posent :
  1. Le canal n'a pas déjà été fermé et il existe d'autres canaux sur le fichier (règle 9.5). Le canal est supprimé et le fichier reste toujour ouvert ;
  2. Le canal a déjà été fermé et il existe d'autres canaux sur le fichier (règle 9.6). Rien n'est fait.
  3. Le canal n'a pas déjà été fermé et il n'existe pas d'autres canaux sur le fichier (règle 9.7). Le canal est supprimé et le fichier fermé ;
  4. Le canal a déjà été fermé et il n'existe pas d'autres canaux sur le fichier (règle 9.8). Rien n'est fait.
- La règle 9.9 est pour la fermeture d'un fichier *via* un canal d'écriture. Le fichier est fermé s'il était ouvert ou sinon, reste fermé ;
- La règle 9.10 est pour l'accés à la valeur du fichier. Le canal pointe alors sur la valeur suivante ;
- La règle 9.11 est pour la lecture d'une valeur dans le fichier. Mais le canal ne pointe plus sur une valeur : nous sommes à la fin du fichier. La valeur vide est retournée.
- La règle 9.12 permet d'écrire une valeur. Cette valeur est ajoutée dans le fichier si seulement elle ne contient pas un canal ou qu'elle ne soit pas locale ou répliquée (ne contient pas un vecteur parallèle ou une primitive parallèle) ;
- La règle 9.13 permet de déplacer le pointeur du canal (d'un fichier ouvert en écriture) afin de lire une autre valeur ;
- La règle 9.14 permet la suppression du fichier, si celui est fermé.

Ces opérations sont distinguées par une étiquette (loc ou glo) notée eti, nous avons besoins de règles, une pour les fichiers locaux $\xrightarrow[\delta_i]{io}$ et une autre pour les fichiers globaux $\xrightarrow[\delta_{\bowtie}]{io}$:

$$\frac{e\ /\ \{f_i\} \xrightarrow[\delta]{io} e'\ /\ \{f_i'\}}{e\ /\ \{f_i\} \xrightarrow[\delta_i]{io} e'\ /\ \{f_i'\}} \qquad \frac{e\ /\ \{\mathcal{F}\} \xrightarrow[\delta]{io} e'\ /\ \{\mathcal{F}'\}}{\{\mathcal{F}\}\ /\ e\ /\ \{f\} \xrightarrow[\delta_{\bowtie}]{io} \{\mathcal{F}'\}\ /\ e'\ /\ \{f\}}$$

Pour un processeur i, l'opération d'E/S fonctionne sur son système local de fichiers. Pour la machine parallèle, nous avons les mêmes opérations, excepté qu'elles sont exécutées sur le système global de fichiers. La

primitive $\mathbf{copy}_{\bowtie}$ qui copie des fichiers locaux en globaux, a la sémantique suivante :

$$\{F', .., F''\}/(\mathbf{copy}\ g)/\{f_0, \ldots, f_{p-1}\} \xrightarrow[\delta_{\bowtie}]{io} \{F', .., F'', \mathfrak{C}F_{j_1}\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, \mathfrak{C}F_{j_m}\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right]\}/()/\{f_0, \ldots, f_{p-1}\}$$

tel que $\forall j \in \{0 \ldots (p-1)\}$ tel que $(g\ j) = (g_j, F_j)$ et si $F_j \notin \{F', .., F''\}$ et $f_j = \{g_i, \ldots, \mathfrak{C}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\}$. Chaque fichier local est copié dans le système global suivant le paramètre g et si les fichiers locaux sont fermés et si les fichiers globaux n'existent pas déjà.

Les contextes d'évaluation sont les mêmes que ceux de la section 3.2.3 du chapitre 3. La figure 9.8 les rappelle. Avec ces contextes, nous pouvons réduire en «profondeur» dans les expressions. Pour cela, nous utilisons des règles d'inférences.

Définition 29 (*Sémantique à «petits pas» avec persistance*).
La sémantique de BSML avec persistance $\rightharpoonup$ est définie par :

$$\frac{e\ /\ \{f_i\} \xrightarrow{i} e'\ /\ \{f_i'\}}{\{\mathcal{F}\}\ /\ \Gamma_l^i(e)\ /\ \{f\} \rightharpoonup \{\mathcal{F}\}\ /\ \Gamma_l^i(e')\ /\ \{f'\}} \quad \text{où} \quad \begin{cases} \{f\} = \{\{f_0\}, \ldots, \{f_i\}, \ldots, \{f_{p-1}\}\} \\ \{f'\} = \{\{f_0\}, \ldots, \{f_i'\}, \ldots, \{f_{p-1}\}\} \end{cases}$$

et

$$\frac{\{\mathcal{F}\}\ /\ e\ /\ \{f\} \overset{\bowtie}{\rightharpoonup} \{\mathcal{F}'\}\ /\ e'\ /\ \{f\}}{\{\mathcal{F}\}\ /\ \Gamma(e)\ /\ \{f\} \rightharpoonup \{\mathcal{F}'\}\ /\ \Gamma(e')\ /\ \{f\}}$$

Nous pouvons donc réduire à l'intérieur d'un vecteur parallèle et le contexte précise dans quel processeur l'expression est évaluée. Nous avons donc bien, une règle pour la réduction locale et une autre pour la réduction globale. Nous avons les résultats suivants·

Lemme 66 (*Confluence forte*)

Soit un système global de fichiers $\{\mathcal{F}\}$, une expression e et des systèmes locaux de fichiers $\{f\}$.
Si $\{\mathcal{F}\}/e/\{f\} \rightharpoonup \{\mathcal{F}^1\}/e^1/\{f^1\}$
et $\{\mathcal{F}\}/e/\{f\} \rightharpoonup \{\mathcal{F}^2\}/e^2/\{f^2\}$
alors il existe un système global de fichiers $\{\mathcal{F}^3\}$, une expression e^3 et des systèmes locaux de fichiers $\{f^3\}$
tels que $\{\mathcal{F}^1\}/e^1/\{f^1\} \rightharpoonup \{\mathcal{F}^3\}/e^3/\{f^3\}$
et $\{\mathcal{F}^2\}/e^2/\{f^2\} \rightharpoonup \{\mathcal{F}^3\}/e^3/\{f^3\}$.

Preuve. Voir en section 9.A de l'annexe de ce chapitre. ■

Ce qui nous donne :

Théorème 13 (*Confluence*)

Soit une expression «programmeur e^p» tel que $e = \mathcal{T}_\bullet(e^p)$, un système global de fichiers $\{\mathcal{F}\}$ et des systèmes locaux de fichiers $\{f\}$. Si $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^1\}/v_1/\{f^1\}$ et $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^2\}/v_2/\{f^2\}$ alors $v_1 = v_2$, $\mathcal{F}^1 = \mathcal{F}^2$ et $f^1 = f^2$.

Preuve. La relation $\rightharpoonup$ est fortement confluente, donc d'après le lemme 1, elle est confluente. ■

Notons que la sémantique n'est pas déterministe mais confluente, car plusieurs règles peuvent être appliquées à un instant donné, le parallélisme provenant des contextes.

9.5 Coût des nouvelles primitives et expérimentations

9.5.1 Coût EM^2-BSP des primitives

Le modèle de coût associé à nos programmes est celui de EM^2-BSP. Nous ne revenons pas sur la partie BSP de nos primitives. Il suffit pour chacune d'elles de rajouter les coûts d'E/S dans les coûts locaux. Prenons le cas de **mkpar** pour illustrer cette modification triviale. Si le temps de calcul et d'E/S de l'évaluation du

Ouverture d'un fichier :

$$(\mathbf{open}^{\mathbf{r}}_{eti}\ f)/\{f',\ldots,{}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}},\ldots,f''\} \quad \overset{io}{\underset{\delta}{\rightharpoonup}} \quad f^{a}_{\mathbf{r}}\ /\ \{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a}_{0}],\ldots,f''\} \tag{9.1}$$

$$(\mathbf{open}^{\mathbf{r}}_{eti}\ f)/\{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{z}],\ldots,f''\} \overset{io}{\underset{\delta}{\rightharpoonup}} f^{\xi}_{\mathbf{r}}\ /\ \{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{z},g^{\xi}_{0}],\ldots,f''\} \tag{9.2}$$

$$(\mathbf{open}^{\mathbf{w}}_{eti}\ f)/\{f',\ldots,{}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}},\ldots,f''\} \quad \overset{io}{\underset{\delta}{\rightharpoonup}} \quad f^{\xi}_{\mathbf{w}}\ /\ \{f',\ldots,{}^{\mathbf{w}}f\boxed{\emptyset},\ldots,f''\} \tag{9.3}$$

$$(\mathbf{open}^{\mathbf{w}}_{eti}\ f)/\{f',\ldots,f''\} \quad \overset{io}{\underset{\delta}{\rightharpoonup}} \quad f^{\xi}_{\mathbf{w}}\ /\ \{f',\ldots,{}^{\mathbf{w}}f\boxed{\emptyset},\ldots,f''\} \text{ si } f\notin\{f',\ldots,f''\} \tag{9.4}$$

Fermeture d'un fichier :

$$(\mathbf{close}^{\mathbf{r}}_{eti}\ f^{\xi}_{\mathbf{r}})/\{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{\xi}_{k},\ldots,g^{z}],\ldots,f''\} \quad \overset{io}{\underset{\delta}{\rightharpoonup}} \quad ()\ /\ \{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{z}],\ldots,f''\} \tag{9.5}$$

$$(\mathbf{close}^{\mathbf{r}}_{eti}\ f^{\xi}_{\mathbf{r}})/\{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{z}],\ldots,f''\} \quad \overset{io}{\underset{\delta}{\rightharpoonup}} \quad ()\ /\ \{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{z}],\ldots,f''\} \tag{9.6}$$

$$(\mathbf{close}^{\mathbf{r}}_{eti}\ f^{\xi}_{\mathbf{r}})/\{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{\xi}_{k}],\ldots,f''\} \quad \overset{io}{\underset{\delta}{\rightharpoonup}} \quad ()\ /\ \{f',\ldots,{}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}},\ldots,f''\} \tag{9.7}$$

$$(\mathbf{close}^{\mathbf{r}}_{eti}\ f^{\xi}_{\mathbf{r}})/\{f',\ldots,{}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}},\ldots,f''\} \quad \overset{io}{\underset{\delta}{\rightharpoonup}} \quad ()\ /\ \{f',\ldots,{}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}},\ldots,f''\} \tag{9.8}$$

$$(\mathbf{close}^{\mathbf{w}}_{eti}\ f^{\xi}_{\mathbf{w}}/\{f',\ldots,{}^{?}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}},\ldots,f''\} \quad \overset{io}{\underset{\delta}{\rightharpoonup}} \quad ()\ /\ \{f',\ldots,{}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}},\ldots,f''\} \text{ où } ?=\mathbf{w} \text{ ou } ?=\mathbf{c} \tag{9.9}$$

Lecture et écriture d'une valeur :

$$(\mathbf{read}_{eti}\ f^{\xi}_{\mathbf{r}})/\{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}\vdots\\ v_k\\ \vdots\end{smallmatrix}}[g^{a},\ldots,g^{\xi}_{k},\ldots,g^{z}],\ldots,f''\} \overset{io}{\underset{\delta}{\rightharpoonup}} v_k\ /\ \{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}\vdots\\ v_k\\ \vdots\end{smallmatrix}}[g^{a},\ldots,g^{\xi}_{m},\ldots,g^{z}],\ldots,f''\}$$
avec $m=k+1$. v_k est la $k$ème valeur de f (9.10)

$$(\mathbf{read}_{eti}\ f^{\xi}_{\mathbf{r}})/\{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{\xi}_{k},\ldots,g^{z}],\ldots,f''\} \overset{io}{\underset{\delta}{\rightharpoonup}} \mathbf{nc}\ /\ \{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{\xi}_{k},\ldots,g^{z}],\ldots,f''\}$$
si $k>n$ (9.11)

$$(\mathbf{write}_{eti}\ (v,f^{\xi}_{\mathbf{w}}))/\{f',\ldots,{}^{\mathbf{w}}f\boxed{\vdots},\ldots,f''\} \overset{io}{\underset{\delta}{\rightharpoonup}} ()/\{f',\ldots,{}^{\mathbf{w}}f\boxed{\begin{smallmatrix}v\\ \vdots\end{smallmatrix}},\ldots,f''\} \text{ si } \mathcal{A}_c(v)\neq\mathbf{true} \text{ et } \mathcal{V}_\bullet(v)=\mathbf{true} \tag{9.12}$$

Déplacement dans le fichier et supression d'un fichier :

$$(\mathbf{seek}_{eti}\ f^{\xi}_{\mathbf{r}}\ k)/\{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{\xi}_{m},\ldots,g^{z}],\ldots,f''\} \overset{io}{\underset{\delta}{\rightharpoonup}} \mathbf{nc}\ /\ \{f',\ldots,{}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^{a},\ldots,g^{\xi}_{k},\ldots,g^{z}],\ldots,f''\} \tag{9.13}$$

$$(\mathbf{delete}_{eti}\ f)/\{f',\ldots,{}^{\mathbf{c}}f\boxed{\vdots},\ldots,f''\} \quad \overset{io}{\underset{\delta}{\rightharpoonup}} \quad ()/\{f',\ldots,f''\} \tag{9.14}$$

Figure 9.7 — Règle des opérations d'E/S

$$
\begin{array}{rcl}
\Gamma & ::= & [] \\
& | & \Gamma\ e \\
& | & v\ \Gamma \\
& | & (\Gamma, e) \\
& | & (v, \Gamma) \\
& | & \textbf{if}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{mkpar}\ \Gamma) \\
& | & (\textbf{apply}\ \Gamma\ e) \\
& | & (\textbf{apply}\ v\ \Gamma) \\
& | & (\textbf{put}\ \Gamma) \\
& | & (\textbf{proj}\ \Gamma)
\end{array}
\qquad
\begin{array}{rcl}
\Delta_i & ::= & \Delta_i\ e \\
& | & v\ \Delta_i \\
& | & (\Delta_i, e) \\
& | & (v, \Delta_i) \\
& | & \textbf{if}\ \Delta_i\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{mkpar}\ \Delta_i) \\
& | & (\textbf{apply}\ \Delta_i\ e) \\
& | & (\textbf{apply} v\ \Delta_i) \\
& | & (\textbf{put}\ \Delta_i) \\
& | & (\textbf{proj}\ \Delta_i) \\
& | & \langle e, \ldots, \overbrace{\Gamma^l[e]}^{i}, \ldots, e \rangle
\end{array}
\qquad
\begin{array}{rcl}
\Gamma^l & ::= & [] \\
& | & \Gamma^l\ e \\
& | & v\ \Gamma^l \\
& | & (\Gamma^l, e) \\
& | & (v, \Gamma^l) \\
& | & \textbf{if}\ \Gamma^l\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & [\Gamma^l, e_1, \ldots, e_n] \\
& | & [v_0, \Gamma^l, \ldots, e_n] \\
& | & \ldots \\
& | & [v_0, v_1, \ldots, \Gamma^l]
\end{array}
$$

Figure 9.8 — Rappel des contextes d'évaluation

paramètre de **mkpar** est w_{all} et que le temps d'évaluation asynchrone de chaque composante du vecteur est $w_i + m_i$ (temps des calcul et temps des E/S) alors le temps d'évaluation du vecteur parallèle est :

$$w_{all}/\!\!\phi w_0 + m_0, \ldots, w_{p-1} + m_{p-1}\!\!\phi$$

Comme pour les primitives parallèles, nos opérations d'E/S ont un coût que nous allons donner à l'aide du modèle EM^2-BSP et nous faisons l'hypothèse que les arguments des opérations d'E/S ont été préalablement évalués (stratégie d'appel par valeur).

Comme nous l'avons expliqué dans la section 9.2.1, chaque transfert de (resp. vers) la mémoire externe locale vers (resp. depuis) la mémoire principale coûte $\lceil \frac{n}{D^l} \rceil \times G^l + \lceil \frac{n+1}{D^l B^l} \rceil \times O^l$ pour n octets. Le coût pour la mémoire externe globale est $\lceil \frac{n}{D^g} \rceil \times G^g + \lceil \frac{n+1}{D^g B^g} \rceil \times O^g$. Notons que dans le cas d'un fichier vide, aucune valeur n'est lue sur le fichier. Le coût d'un telle opération n'est donc que le surcoût O^l (où O^g pour un fichier global). De cette manière, nous avons, de manière très simple, le coût de l'appel système pour l'accès à un fichier. Ce sont ces constantes que nous utiliserons pour les coûts des opérations ne nécessitant qu'un accès constant aux systèmes de fichiers.

Suivant que le système global de fichiers est partagé ou distribué, les coûts des opérations d'E/S globaux (figure 9.9) seront différents.

Les opérations locales sont asynchrones. Elles participent donc à la première phase d'une super-étape. Dans le cas d'un système global distribué, les opérations globales ont les mêmes coûts que celles locales. Nous sommes en présence d'une architecture «shared nothing» et donc toutes les opérations d'E/S sont asynchrones car tout est distribué et/ou dupliqué.

Dans le cas d'un système global partagé (présence de disques partagés), les opérations globales sont synchrones car elles modifient le contexte global de la machine EM^2-BSP . Par exemple, ouvrir un fichier global nécessite une synchronisation. En effet, l'exécution du programme serait non-déterministe si un processeur pouvait écrire dans un fichier global alors qu'un autre processeur écrit une autre valeur dans ce fichier ou tente d'ouvrir ce même fichier en lecture. Avec cette synchronisation globale, tous les processeurs ouvrent (resp. ferment ou détruisent) le fichier et ils se communiquent leur réussite ou leur échec. $p - 1$ booléens sont ainsi transmis sur le réseau par chaque processeur et une exception globale peut être levée si un processeur a eu un problème.

Deux exceptions notoires sont à noter. **glo_output_value** et **glo_input_value** ne nécessitent pas fondamentalement de synchronisation. Comme nous sommes dans le contexte global, toutes les valeurs répliquées à lire ou à écrire sont identiques (elles ne contiennent pas un vecteur parallèle) en chaque processeur. La lecture de ces valeurs peut donc se faire dans un ordre quelconque. Différent canaux sont positionnés en différents points du fichier mais lisent la même valeur en un même point[4]. L'écriture des valeurs sur un même fichier n'étant faite que par un seul processeur, aucun problème d'écriture concurrente (de différents processeurs) sur un canal d'écriture ne peut avoir lieu. Notons que la sémantique (et par conséquent l'implantation) de **glo_open_out** n'autorise l'ouverture d'un fichier que pour un seul canal d'écriture,

[4]Cela n'est plus vrai si un programme extérieur «s'amusait» à modifier ces fichiers, mais cela relève d'un problème de sécurité que nous ne traiterons pas dans ce manuscrit. Notons que l'on trouve ce même problème à l'ouverture d'un fichier.

Opérateur	Coût
loc_open_in (resp. out)	temps constant O^l
(**loc_output_value** v)	$\lceil \frac{size(v)}{D^l} \rceil \times G^l + \lceil \frac{size(v)+1}{D^l B^l} \rceil \times O^l$
loc_input_value	$\lceil \frac{size(v)}{D^l} \rceil \times G^l + \lceil \frac{size(v)+1}{D^l B^l} \rceil \times O^l$ où v est la valeur lue
loc_close_in (resp. out)	temps constant O^l
loc_delete	temps constant O^l
glo_open_in	$\begin{cases} (p-1) \times g \ + \ O^g \ + \ l & \text{Si le système global de fichiers est partagé} \\ O^l & \text{Autrement} \end{cases}$
glo_open_out	$\begin{cases} (p-1) \times g \ + \ O^g \ + \ l & \text{Si le système global de fichiers est partagé} \\ O^l & \text{Autrement} \end{cases}$
(**glo_output_value** v)	$\begin{cases} \lceil \frac{size(v)}{D^g} \rceil \times G^g + \lceil \frac{size(v)+1}{D^g B^g} \rceil \times O^g & \text{Si partagé} \\ \lceil \frac{size(v)}{D^l} \rceil \times G^l + \lceil \frac{size(v)+1}{D^l B^l} \rceil \times O^l & \text{Autrement} \end{cases}$
glo_input_value	$\begin{cases} \lceil \frac{size(v)}{D^g} \rceil \times G^g + \lceil \frac{size(v)+1}{D^g B^g} \rceil \times O^g & \text{Si partagé} \\ \lceil \frac{size(v)}{D^l} \rceil \times G^l + \lceil \frac{size(v)+1}{D^l B^l} \rceil \times O^l & \text{Autrement} \\ \text{et où } v \text{ est la valeur lue} \end{cases}$
glo_close_in	$\begin{cases} (p-1) \times g \ + \ O^g \ + \ l & \text{Si le système global de fichiers est partagé} \\ O^l & \text{Autrement} \end{cases}$
glo_close_out	$\begin{cases} (p-1) \times g \ + \ O^g \ + \ l & \text{Si le système global de fichiers est partagé} \\ O^l & \text{Autrement} \end{cases}$
glo_delete	$\begin{cases} (p-1) \times g \ + \ O^g \ + \ l & \text{Si le système global de fichiers est partagé} \\ O^l & \text{Autrement} \end{cases}$
(**glo_copy**f)	$\begin{cases} \lceil \frac{size(f_i)}{D^g} \rceil \times G^g + \lceil \frac{size(f_i)}{D^g B^g} \rceil \times O^g + \lceil \frac{size(f_i)}{D^l} \rceil \times G^l + \lceil \frac{size(f_i)}{D^l B^l} \rceil \times O^l + l & \text{Si le système global de fichiers est partagé} \\ (\lceil \frac{size(f_i)}{D^l} \rceil \times G^l + \lceil \frac{size(f_i)}{D^l B^l} \rceil \times O^l) \times 2 + size(f_i) \times g + 2 \times l & \text{Autrement} \\ \text{si } \forall i \in \{0 \ldots (p-1)\} \text{ tel que } (f\ i) = (f_i, F) \end{cases}$

Figure 9.9 — Coût des opérations d'E/S parallèles

interdisant de fait l'écriture concurrente de différents processeurs, *via* ces canaux, sur ce même fichier. Notons que l'écriture de vecteurs parallèles n'est pas autorisée car ces valeurs sont différentes en chaque processeur, et il faudrait pouvoir toutes les écrire, et dans le bon ordre, pour pouvoir être ensuite relu. Cela impliquerait une synchronisation que nous ne voulons pas pour des raisons d'efficacité de l'écriture dans un fichier global.

9.5.2 Implantation des primitives

Les canaux glo_channel et loc_channel sont des types abstraits et sont implantés comme des tableaux de canaux, un canal par disque.

L'implantation actuelle emploie les fonctionnalités des processus légers («*threads*» dans la littérature anglo-saxonne) d'OCaml pour pouvoir écrire (ou lire) sur les D-disques de chaque composante de la machine parallèle : nous créons D processus légers qui écrivent (ou lisent) sur les D canaux. Chacun contient une partie de la donnée, structurée en une suite d'octets, et chacun l'écrit sur un disque parallèlement aux autres processus-légers. Pour transformer une valeur en une séquence d'octets, nous devons la linéariser. Cette séquence d'octets pourra par la suite être décodée en une valeur. Le module Marshal de la bibliothèque standard d'OCaml fournit un tel dispositif.

Dans le cas de disques partagés globaux, un des processeurs est choisi pour réellement écrire la valeur. Dans notre première implantation, c'est chacun à tour de rôle. Avec des bibliothèques de plus bas niveaux, nous pourrions implanter un «processus démon» qui n'autoriserait l'écriture qu'au premier des processeurs qui a atteint dans son code, la primitive d'écriture globale.

Pour communiquer les booléens, nous utilisons le module de communication Comm de l'implantation modulaire de la BSMLlib. Un échange total des booléens, indiquant si le processeur a bien ouvert (resp. fermé) son fichier global, permet de savoir si de manière globale, ce fichier a bien été ouvert (resp. fermé) ou non.

Les systèmes globaux (et locaux) de fichiers sont dans différents répertoires. Les chemins de ces répertoires sont des paramètres systèmes de la BSMLlib ainsi qu'un booléen qui indique si le système de fichiers est partagé ou non. Le répertoire pour les fichiers globaux est supposé être monté pour pouvoir accéder aux disques partagés. Le chemin du répertoire global doit être différent de celui du répertoire des fichiers locaux. De cette manière, les primitives globales accèdent aux fichiers globaux et les primitives locales aux fichiers locaux. Dans le cas d'une architecture ne comprenant que des disques partagés, par exemple, une machine séquentielle (comme un PC) à qui on voudrait donner le rôle de machine parallèle, les primitives locales utilisent le pid des processeurs pour distinguer les fichiers locaux des différents processeurs (dans l'implantation actuelle).

9.5.3 Exemple d'utilisation de nos primitives

Notre exemple est le calcul classique des préfixes d'une liste. Ici nous faisons l'hypothèse que les éléments de la liste sont distribués sur tous les processeurs comme un fichier qui contient une sous-partie de la liste initiale. Chaque fichier est découpé en sous-listes contenant $\frac{D^l \times B^l}{s}$ éléments où s est la taille d'un élément. Nous rappellons d'abord les parties purement séquentielles en OCaml de notre algorithme :

```
let isnc=function None→true | _→false

(* seq_scan_last:(α →α →α )→α →α list→α *α list*)
let seq_scan_last op e l =
 let rec seq_scan' last l accu = match l with
             []→(last,(List.rev accu))
      | hd::tl→(let new_last = (op last hd)
                  in seq_scan' new_last tl (new_last::accu))
  in seq_scan' e l []
```

avec List.rev $[v_0; v_1; \ldots; v_n] = [v_n; \ldots; v_1; v_0]$. Pour calculer les préfixes d'une liste, nous calculons préalablement les préfixes des listes représentées par des fichiers locaux. Pour cela, nous utilisons le code suivant :

```
(* seq_scan_list_io:(α →α →α )→α →loc_name→loc_name→α  *)
let seq_scan_list_io op e name_in name_tmp=
 let cha_in =loc_open_in name_in in
 let cha_tmp=loc_open_out name_tmp in
 let rec seq_scan' last =
  let block=(loc_input_value cha_in) in
   if (isnc block) then last
    else let block2=(seq_scan_last op last (noSome block)) in
      loc_output_value cha_tmp (snd block2);
      seq_scan' (fst block2) in
 let res=seq_scan' e in
  loc_close_in cha_in;loc_close_out cha_tmp;res
```

Le fichier local est tout d'abord ouvert, ainsi qu'un fichier temporaire. Pour chacune des sous-listes du fichier, nous calculons les préfixes et le dernier élément ainsi calculé. Ensuite, nous écrivons ces préfixes dans le fichier temporaire et nous fermons les deux fichiers. Maintenant, nous calculons les préfixes parallèles de ces derniers éléments. Il ne nous reste plus alors qu'à ajouter ces valeurs dans les fichiers temporaires :

```
(* add_last:(α →α →α )−α →loc_name→loc_name→unit *)
let add_last op e name_tmp name_out =
```

```
 let cha_tmp=loc_open_in name_tmp in
 let cha_out=loc_open_out name_out in
 let rec seq_add () =
  let block = (loc_input_value cha_tmp) in
   if (isnc block) then () else
    loc_output_value cha_out(List.map (op e)(noSome block));
    seq_add () in
 seq_add ();loc_close_in cha_tmp;
 loc_close_out cha_out; loc_delete name_tmp
```

Le fonctionnement de add_last est similaire à celui de seq_scan_list_io. La fonction finale n'est alors plus que la composition des fonctions susmentionnées :

```
(*scan:(α →α →α )→α →loc_name→loc_name→loc_name→unit par*)
let scan_list_direct_io op e name_in name_tmp name_out =
 let lasts=parfun (seq_scan_list_io op e name_in)
                                        (replicate name_tmp) in
 let tmp_values=scan_direct op lasts in
  parfun3 (add_last op) tmp_values
                    (replicate name_tmp) (replicate name_out)
```

Pour un exemple d'utilisation des fichiers globaux, nous donnons le code de la distribution des sous-listes aux composantes de la machine parallèle. Pour chaque bloc de la liste initiale, un processeur l'écrit dans son fichier local :

```
(* distribut:glo_name→loc_name→unit *)
let distribut name_in name_out =
 let cha_in=glo_open_in name_in in
 let cha_outs=parfun loc_open_in (replicate name_out) in
 let rec distri m =
  let block=glo_input_value cha_in in
   if (isnc block) then () else
    (apply2 (mkpar (fun pid→if pid=m then loc_output_value
                                  else (fun a b→())))
            cha_outs (replicate (noSome block)));
            distri ((m+1) mod (bsp_p())) in
  distri 0;parfun loc_close_out cha_outs;glo_close_in cha_in
```

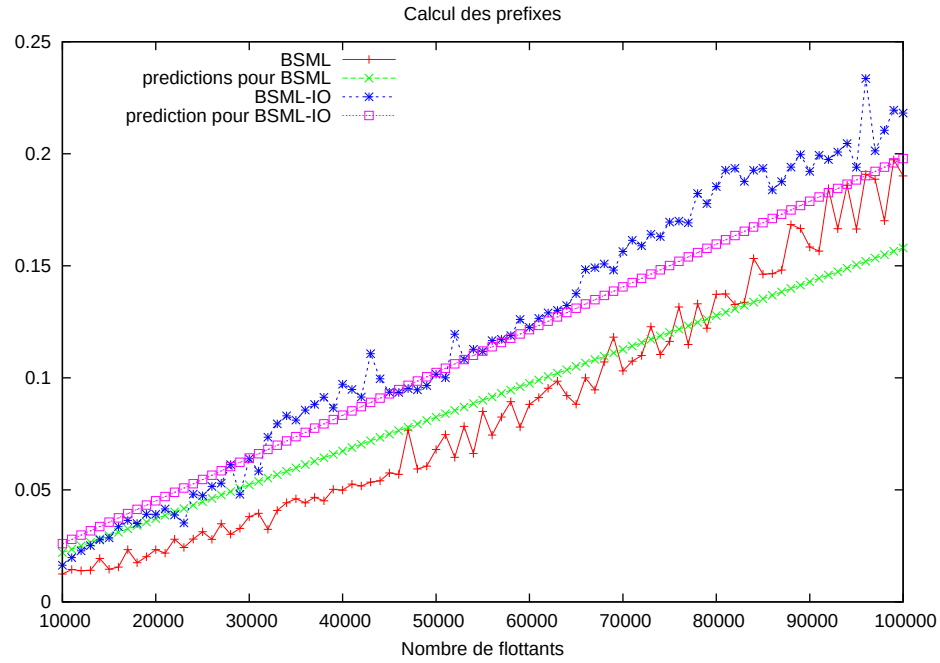
Nous avons la formule de coût EM^2-BSP suivante pour la version avec mémoire externe du calcul des préfixes. Notons que nous utilisons un algorithme direct pour le calcul des préfixes parallèles :

$$(p-1) \times s \times g + 4 \times N \times (B^l \times G^l + O^l) + 2 \times r \times N \times (D^l \times B^l) + T_1 + l$$

si nous lisons les sous-listes du fichier par blocs de taille $D^l B^l$ et où s dénote la taille en octets d'un élément, N est la longueur maximale d'un fichier d'un processeur et $T_1 = 12 \times O^l$ est le temps pour ouvrir et fermer les fichiers. Nous avons donc le temps pour lire les fichiers, écrire les résultats temporaires, calculer les préfixes parallèles, lire les fichiers temporaires et écrire le résultat final dans les fichiers finaux. Le coût EM^2-BSP de la distribution des données est :

$$\begin{cases} p \times N \times (\lceil \frac{D^l B^l}{D^g} \rceil \times G^g + \lceil \frac{D^l B^l}{D^g B^g} \rceil \times O^g) + N \times (B^l \times G^l + O^l) + 2 \times l + T_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Si les disques globaux sont partagés} \\ p \times N \times (B^l \times G^l + O^g) + N \times (B^l \times G^l + O^l) + T_2' \quad \text{Autrement} \end{cases}$$

tel que T_2 et T_2' sont les temps pour ouvrir et fermer les fichiers. Nous avons le temps pour lire les données depuis le fichier global (lecture par blocs de taille $D^l B^l$) et de les écrire sur les fichiers locaux. Nous avons aussi deux barrières de synchronisations dues au opérateurs **glo_open_in** et **glo_close_in**.



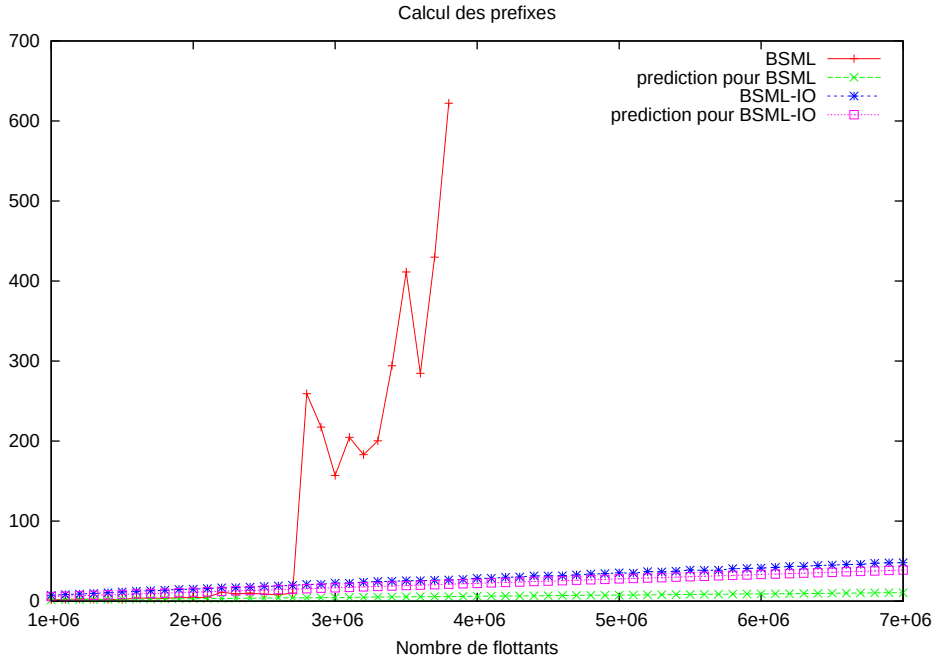


Figure 9.10 — Benchmarks du calcul des préfixes E/S d'une liste

9.5.4 Expérimentations

Des premières expériences ont été faites sur une grappe de 6 nœuds chacun avec 256Mo de RAM. Les nœuds sont des Intel pentium IV 2.8 Ghz avec des cartes Gigabit Ethernet et interconnectés par un réseau Gigabit Ethernet (10/100/1000). Une Mandrake clic 2.0 a été utilisée comme système d'exploitation et les programmes ont été compilés avec OCaml 3.08.02 en mode natif. Chaque nœud contient un disque local. Le serveur du cluster possède un disque partagé par les nœuds via NFS.

Ces tests ont été effectués pour comparer les temps d'exécution entre un algorithme BSP utilisant seulement la BSMLlib et un algorithme EM^2-BSP utilisant nos nouvelles primitives. L'algorithme BSP lit les données à partir d'un fichier global et les maintient dans les mémoires centrales des processeurs. L'algorithme EM^2-BSP a distribué les données comme décrit dans la section précédente. La figure 9.10 récapitule les temps d'exécution des 2 programmes. Ceux-ci ont été exécutés 100 fois pour le graphique du haut et 3 fois pour le graphique du bas. Ensuite, la moyenne des temps a été prise.

Seul le calcul des préfixes, à proprement parler, a été pris en considération. Le temps nécessaire à la distribution des données a été ignoré car le cluster n'a pas un vrai disque partagé, mais un simulé en utilisant le système NFS. Par conséquent, la distribution des données est lente (G^g dépend de g) et celle-ci est similaire dans les deux algorithmes.

Notre grappe a les paramètres EM^2-BSP suivants :

p	$=$	6	nœuds	D^l	$=$	1	octets	D^g	$=$	1	octets
r	$=$	469	Mflops/s	B^l	$=$	4096	octets	B^g	$=$	4096	octets
g	$=$	28	flops	G^l	$=$	1.2	flops	G^g	$=$	33.33	flops
l	$=$	227512	flops	O^l	$=$	100	flops	O^g	$=$	120	flops

en utilisant la version MPI du module de communication de la BSMLlib. Les paramètres d'E/S ont été obtenus en employant des *benchmarks* tels que ceux de la figure 9.4. Les temps d'exécutions prévus (théoriques), utilisant ces paramètres, sont aussi présentés dans la figure 9.10. Nous avons utilisé des listes de flottants avec $e = 0.$, $op = +.$ et nous avons approximativement 140 flottants dans un bloc. Ainsi, les listes sont coupées en des sous-listes ayant approximativement 140 éléments.

Pour de petites listes et donc pour un nombre restreint de données, les surcoûts pour la gestion de la mémoire externe rendent le programme BSML plus efficace que celui en EM^2-BSML. Cependant, une fois que toutes les mémoires internes sont employées, les temps d'exécution du programme de BSML dégénèrent, ce qui est dû au coût du mécanisme de pagination pour avoir de la mémoire virtuelle. Les temps d'exécution du programme EM^2-BSML, eux, continuent sans à-coup et surpassent clairement ceux du programme BSML.

Nous pouvons constater une légère différence entre les temps des prévisions des exécutions et ceux mesurés sur notre grappe de test. Mises à part les divergences entre courbes de prédiction et courbes réelles, nous trouvons là les coûts dus au «ramasseur de miettes» (*Garbage Collector* dans la littérature anglo-saxonne) d'OCaml. En effet, dans la famille des langages ML, la machine abstraite contrôle les ressources et la mémoire, à la différence de C ou C++ où le programmeur doit assigner et libérer les données de la mémoire.

Utiliser des primitives d'E/S et un algorithme moins naïf quand le nombre de données à traiter est important, permet de réaliser des gains de performances considérables et améliore la prédiction des performances.

9.A Preuve du théorème

Lemme 67 (*Déterminisme des règles fonctionnelles*)

Soit e une expression.

1. *Si $e \xrightarrow{\varepsilon} e^1$ et $e \xrightarrow{\varepsilon} e^2$ alors $e^1 = e^2$;*
2. *Si $e \underset{\delta}{\rightarrow} e^1$ et $e \underset{\delta}{\rightarrow} e^2$ alors $e^1 = e^2$.*

Preuve. Par cas sur les règles. ■

Lemme 68 (*Déterminisme des règles globales*)

Soit un un système global de fichiers $\{\mathcal{F}\}$, une expression e et des systèmes de fichiers locaux $\{f\}$.

1. *Si $\{\mathcal{F}\} \,/\, e \,/\, \{f\} \underset{\bowtie}{\xrightarrow{\varepsilon}} \{\mathcal{F}^1\} \,/\, e^1 \,/\, \{f^1\}$ et $\{\mathcal{F}\} \,/\, e \,/\, \{f\} \underset{\bowtie}{\xrightarrow{\varepsilon}} \{\mathcal{F}^2\} \,/\, e^2 \,/\, \{f^2\}$ alors $e^1 = e^2$, $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$ et $\{f^1\} = \{f^2\}$;*
2. *Si $\{\mathcal{F}\} \,/\, e \,/\, \{f\} \underset{\delta_{\bowtie}}{\rightarrow} \{\mathcal{F}^1\} \,/\, e^1 \,/\, \{f^1\}$ et $\{\mathcal{F}\} \,/\, e \,/\, \{f\} \underset{\delta_{\bowtie}}{\rightarrow} \{\mathcal{F}^2\} \,/\, e^2 \,/\, \{f^2\}$ alors $e^1 = e^2$, $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$ et $\{f^1\} = \{f^2\}$.*

Preuve. Par application du lemme 67 et par cas sur les règles des primitives et des opérations globales. ■

Lemme 69 (*Déterminisme des règles de persistance*)

Soit un un système global de fichier $\{\mathcal{F}\}$, une expression e et un système local de fichiers $\{f_i\}$.

1. *Si $e \,/\, \{f_i\} \underset{\delta}{\xrightarrow{io}} e^1 \,/\, \{f_i^1\}$ et $e \,/\, \{f_i\} \underset{\delta}{\xrightarrow{io}} e^2 \,/\, \{f_i^2\}$ alors $e^1 = e^2$ et $\{f_i^1\} = \{f_i^2\}$;*
2. *Si $e \,/\, \{\mathcal{F}\} \underset{\delta}{\xrightarrow{io}} e^1 \,/\, \{\mathcal{F}^1\}$ et $e \,/\, \{\mathcal{F}\} \underset{\delta}{\xrightarrow{io}} e^2 \,/\, \{\mathcal{F}^2\}$ alors $e^1 = e^2$ et $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$.*

Preuve. Par examen exhaustif des règles des opérations de persistance. ■

Lemme 70 (*Déterminisme des règles*)

Soit un un système global de fichiers $\{\mathcal{F}\}$, une expression e et des systèmes locaux de fichiers $\{f\}$ ou un système global de fichiers $\{f_i\}$.

1. *Si $e \,/\, \{f_i\} \xrightarrow{i} e^1 \,/\, \{f_i^1\}$ et $e \,/\, \{f_i\} \xrightarrow{i} e^2 \,/\, \{f_i^2\}$ alors $e^1 = e^2$ et $\{f_i^1\} = \{f_i^2\}$;*
2. *Si $\{\mathcal{F}\} \,/\, e \,/\, \{f\} \xrightarrow{\bowtie} \{\mathcal{F}^1\} \,/\, e^1 \,/\, \{f^1\}$ et $\{\mathcal{F}\} \,/\, e \,/\, \{f\} \xrightarrow{\bowtie} \{\mathcal{F}^2\} \,/\, e^2 \,/\, \{f^2\}$ alors $e^1 = e^2$, $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$ et $\{f^1\} = \{f^2\} = \{f\}$.*

Preuve. Par application des lemmes 67 et 69 pour (1) et des lemmes 69 et 68 pour (2). ■

Notons que les contextes sont identiques à ceux du chapitre 3. Donc nous utilisons les mêmes lemmes sur ces contextes.

Lemme 71 (*Confluence forte*)

Soit un système global de fichiers $\{\mathcal{F}\}$, une expression e et des systèmes locaux de fichiers $\{f\}$.

Si	$\{\mathcal{F}\}/e/\{f\} \rightharpoonup \{\mathcal{F}^1\}/e^1/\{f^1\}$
et	$\{\mathcal{F}\}/e/\{f\} \rightharpoonup \{\mathcal{F}^2\}/e^2/\{f^2\}$
alors il existe	*un système global de fichiers $\{\mathcal{F}^3\}$, une expression e^3 et des systèmes locaux de fichiers $\{f^3\}$*
tels que	$\{\mathcal{F}^1\}/e^1/\{f^1\} \rightharpoonup \{\mathcal{F}^3\}/e^3/\{f^3\}$
et	$\{\mathcal{F}^2\}/e^2/\{f^2\} \rightharpoonup \{\mathcal{F}^3\}/e^3/\{f^3\}$.

Preuve . Par le lemme 9, nous avons deux types de réductions bien distincts.

Si $\rightharpoonup$ est une réduction globale, alors par le lemme 7 il n'existe qu'un contexte global et par le lemme 70, la réduction est déterministe.

Si $\rightharpoonup$ est une réduction locale, alors nous avons $e = \Delta^i[e^i]$ et $e = \Delta^j[e^j]$. Nous avons alors deux cas :

1. Si $i = j$ alors par le lemme 8.1 il n'existe qu'un contexte et par le lemme 70, la réduction est déterministe ;
2. Si $i \neq j$ alors par le lemme 8.2, il n'existe qu'un seul vecteur à réduire. Par le lemme 70, les deux réductions sont déterministes. Il est alors facile de constater que les règles peuvent donc s'entrelacer : les réductions interviennent dans deux composantes d'un même vecteur et dans deux systèmes de fichiers locaux différents.

Les deux types de réductions sont donc fortement confluentes. ■

Troisième partie

Opérations globalisées

10 ML parallèle minimalement synchrone

Ce chapitre est une extension des articles [C7] et [R4] écrits en collaboration avec Frédéric Loulergue, Frédéric Dabrowski et Myrto Arapinis.

Sommaire

La première étape avant de définir un langage pour le méta-calcul consiste à obtenir un langage parallèle fonctionnel proche de BSML, mais sans les barrières de synchronisation du modèle BSP. Pour cela, nous nous basons sur le modèle *Message Passing Machine* (MPM) [36, 231] et proposons le langage *Minimally Synchronous Parallel ML* (MSPML) semblable à BSML mais avec une évaluation distribuée bien différente. Ce langage nous servira de «cobaye» pour les extensions asynchrones nécessaires au méta-calcul. Dans ce chapitre, nous présentons trois sémantiques équivalentes de MSPML, la sémantique de haut niveau correspondant à la vue du programmeur (modèle de programmation), une sémantique à «petits pas» décrivant les coûts des programmes et enfin la sémantique distribuée correspondant à une vue proche de l'exécution sur une machine parallèle (modèle d'exécution). L'implantation et quelques expériences terminent ce chapitre.

10.1 Introduction

L'étude d'un nouveau langage fonctionnel parallèle, sans barrière de synchronisation, permettrait de déplacer l'équilibre de BSML entre souplesse de programmation, performances et simplicité de la prédiction de

ces performances. Dans ce nouveau langage, nous gagnerons, avec plus de désynchronisations, de la souplesse pour la programmation (notamment pouvoir y exprimer des algorithmes non BSP) mais, en contrepartie, nous perdrons en simplicité du modèle de coût. Ce langage, appelé MSPML, nous servira de base pour l'étude des problèmes de désynchronisations qu'implique un environnement de méta-calcul (se référer au chapitre 11 pour plus de détails).

MSPML a une syntaxe et une sémantique de «haut niveau» proche de celle de BSML, afin, entre autre, de conserver le travail qui a pu être effectué sur la preuve des programmes (cf. chapitre 6). Le modèle d'exécution (structuration des algorithmes) sera néanmoins différent, nonobstant la ressemblance des programmes. Ceux-ci seront plus efficaces dans le cas de données non équilibrées. Avec ce nouveau langage, nous souhaitons :

1. Une sémantique sans inter-blocages ;
2. Un modèle de coût simple et réaliste mais sans barrière de synchronisation ;
3. Une comparaison de l'efficacité de MSPML et de BSML comme BSP a été comparé avec d'autres paradigmes,
4. Une étude de l'expressivité de MSPML pour les algorithmes non BSP.

Les deux derniers points ne seront pas traité dans ce manuscrit. C'est un travail en cours. Nous y reviendrons dans les perspectives (chapitre 13).

MSPML sera aussi notre sujet d'étude pour la recherche et la conception d'extensions qui ne sont pas évidentes en BSML, comme, par exemple, des primitives permettant la programmation d'algorithmes «clients-serveurs». Nous mixerons aussi MSPML avec BSML pour obtenir un langage dédié au méta-calcul. (voir au chapitre 11).

Ce chapitre est organisé de la manière suivante. Tout d'abord, nous présentons le modèle d'exécution et de coût qui sera utilisé pour MSPML. Ensuite, nous décrivons informellement les primitives parallèles de notre nouveau langage, puis nous donnons différentes sémantiques, du plus haut (modèle de programmation) au plus bas niveau (modèle d'exécution). Nous donnons quelques résultats d'expériences sur une grappe de PC d'un programme utilisant une implantation MSPML d'un patron algorithmique.

10.2 Modèle de coût et d'exécution

Plutôt que de passer directement à la conception d'un modèle et d'un langage à deux niveaux pour l'utilisation de grappes de machines parallèles, nous avons considéré la conception d'un langage proche de BSML mais sans les barrières de synchronisation.

Il est souvent admis que les barrières de synchronisation ne sont pas un handicap pour les performances (dans le cas *d'une seule* machine parallèle), notamment parce qu'une vue globale du calcul permet des optimisations qui ne sont pas possibles dans le cas d'un parallélisme moins structuré (voir par exemple [164]). L'éventualité de performances un peu moins bonnes n'est pas si désavantageuse compte tenu de la plus grande facilité de conception, de correction et de vérification des algorithmes (et des programmes).

Toutefois il y a de nombreux programmes parallèles implémentés, en particulier en MPI, qui ne suivent pas le modèle BSP mais pour lesquels on souhaite pouvoir raisonner sur le coût. C'est ce qui a conduit au modèle *BSP sans barrière* (*BSP without barrier*, BSPWB, en anglais) [232] puis au modèle MPM [231, 36].

BSPWB est un modèle directement inspiré du modèle BSP. Il propose de remplacer la notion de super-étape par la notion de m-étape définie comme suit. À chaque m-étape, chaque processeur effectue une phase de calcul suivie par une phase de communication. Durant la phase de communication, les processeurs échangent les données dont ils ont besoin pour la m-étape suivante.

La machine parallèle est caractérisée par les trois paramètres suivants (les deux derniers sont exprimés comme multiples de la puissance de calcul s des processeurs) :

- Le nombre de processeurs p ;
- La latence L du réseau ;
- Le temps g pour échanger un mot entre deux processeurs.

Le temps nécessaire à un processeur i pour exécuter une m-étape s est $t_{(s,i)}$ borné par T_s le temps nécessaire à l'exécution de la m-étape s par la machine parallèle.

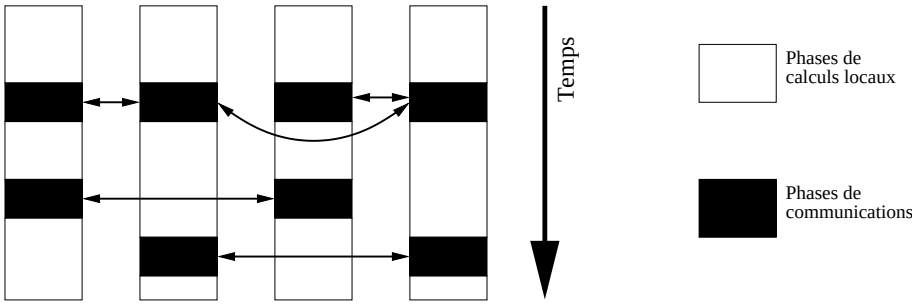


Figure 10.1 — Des m-étapes d'une exécution MPM

mpm_p: unit→int **mpm_l**: unit→float **mpm_g**: unit→float
mkpar: (int→α)→α **par**
apply: ($\alpha \rightarrow \beta$)**par**→α **par**→β **par**
mget: (int→α option)**par**→(int→bool)**par**→(int→α option)**par**
mat: α option **par**→(int→bool)→int→α option

Figure 10.2 — Les primitives de la MSPMLlib

T_s est défini inductivement par :

$$\begin{cases} T_1 = \max\limits_{i=0}^{p-1}\{w_{(1,i)}\} + \max\{g \times h_{(1,i)} + L\} \\ T_s = T_{s-1} + \max\limits_{i=0}^{p-1}\{w_{(s,i)}\} + \max\limits_{i=0}^{p-1}\{g \times h_{(s,i)} + L\} \end{cases}$$

où $s \in \{2, \ldots, R\}$ et avec R qui est le nombre de m-étapes du programme. $w_{(s,i)}$ et $h_{(s,i)}$ sont respectivement le temps de calcul local au processeur i durant la m-étape s et $h_{(s,i)} = max\{h^+_{(s,i)}, h^-_{(s,i)}\}$ où $h^+_{(s,i)}$ (resp. $h^-_{(s,i)}$) est le nombre de mots reçus (resp. envoyés) par le processeur i durant la m-étape s.

Dans ce modèle il y a toutefois une barrière implicite à chaque étape, le coût de la barrière elle-même étant nul. De ce fait, ce modèle est une approximation trop grossière. Une meilleure borne, $\Phi_{(s,i)}$, est donnée par le modèle *Message Passing Machine* [231]. Les paramètres de ce modèle sont identiques à ceux du modèle BSPWB. On utilise l'ensemble $\Omega_{(s,i)}$ pour un processeur i et une m-étape s (Figure 10.1) définie par :

$$\Omega_{(s,i)} \ = \ \{j \mid \text{processeur } j \text{ envoie un message au processeur } i \text{ à la } m\text{-étape } s\} \bigcup \{i\}$$

Les processeurs de l'ensemble $\Omega_{(s,i)}$ sont appelés «partenaires entrants» du processeur i à la m-étape s. La borne $\Phi_{(s,i)}$ est définie inductivement par :

$$\begin{cases} \Phi_{(1,i)} = \max\limits_{i=0}^{p-1}\{w_{(1,j)} | j \in \Omega_{(1,i)}\} + (g \times h_{(1,i)} + L) \\ \Phi_{(s,i)} = \max\limits_{i=0}^{p-1}\{\Phi_{(s-1,j)} + w_{(s-1,j)} | j \in \Omega_{(s,i)}\} + (g \times h_{(s,i)} + L) \end{cases}$$

où $h_{(s,i)} = \max\{h^+_{(s,i)}, h^-_{(s,i)}\}$ pour $i \in \{0, \ldots, p-1\}$ et $s \in \{2, \ldots, R\}$ (R étant toujours le nombre de m-étapes). Le temps d'exécution pour un programme est donc borné par :

$$\Psi = \max\{\Phi_{(R,j)} | j \in \{0, 1, \ldots, p-1\}\}$$

Le modèle MPM prend en compte le fait qu'un processeur ne se synchronise qu'avec chacun de ses partenaires entrants et est donc plus précis que BSPWB. Les premières expériences menées montrent (section 10.6) que ce modèle s'appliquera bien à MSPML.

10.3 Langage MSPML

Il n'y a pas d'implantation d'un langage MSPML complet mais une implantation partielle sous forme d'une bibliothèque pour OCaml.

La bibliothèque MSPMLlib est basée sur les éléments donnés à la figure 10.2. La structure de données parallèle (appelée vecteur parallèle), l'accès aux paramètres de la machine parallèle (mais ici les paramètres MPM), la création de vecteur parallèle, l'application point-à-point sont identiques aux primitives de la BSMLlib. Les différences proviennent des primitives pour les communications : les communications sont exprimées à l'aide des primitives **mget** et **mat**.

La primitive **mget** permet à un processeur de demander des données à plusieurs processeurs durant la même m-étape et d'envoyer des messages différents à des demandeurs différents. Sa sémantique est :

$$\textbf{mget}\ \boxed{f_0 \mid \cdots \mid f_{p-1}}\ \boxed{b_0 \mid \cdots \mid b_{p-1}} = \boxed{g_0 \mid \cdots \mid g_{p-1}}$$

où $g_i =$ **fun** $j \rightarrow$**if** $(0 \leq j < p-1)$ **then** (**if** $(b_i\ j)$ **then** $(f_j\ i)$ **else** None) **else** None

La primitive **mat** permet à des processeurs de projeter leurs valeurs et donc de les diffuser aux autres processeurs. Ces diffusions ne sont effectuées que si les autres processeurs ont besoin de ces valeurs. La sémantique de cette primitive est donc :

$$\textbf{mat}\ \boxed{v_0 \mid \cdots \mid v_{p-1}}\ f = g$$

où $g =$ **fun**$j \rightarrow$**if** $(0 \leq j < p-1)$ **then** (**if** $(f\ j)$ **then** v_j **else** None) **else** None

Notons qu'en BSML, les primitives de communication n'ont qu'un argument (les valeurs à envoyer), alors qu'en MSPML ces primitives en ont deux : les valeurs qui peuvent être envoyées et les processeurs qui nécessitent ces valeurs. Ensuite, dans les deux langages, le résultat est une (ou des) fonction(s) donnant les valeurs reçues.

10.3.1 Exemples

Tout comme la bibliothèque BSMLlib, la distribution MSPMLlib contient une bibliothèque standard. De nombreuses fonctions sont identiques mais dès que des communications sont nécessaires, il y a des différences. Par exemple, la diffusion directe[1] peut être codée ainsi :

```
(* get: α par→int par→α par *)
let get v vi =
 let proc=(parfun (fun i j →i=j) vi)
 and values=parfun (fun v i →Some v) v in
  parfun (fun f i →noSome (f i)) (mget proc values) vi

(* bcast_direct: int→α par→α par *)
let bcast_direct root vv = get vv (replicate root)
```

Son coût MPM est $(p-1) \times s \times g\ +\ L$, où s est la taille de la valeur v_n au processeur n. On peut aussi la programmer avec un **mat** de la manière suivante :

```
(* at: α par→int→α  *)
let at v n =
 let rcv=mat (applyat n (fun v →Some v) (fun _ →None)) (fun i →i=n) in
  noSome (rcv n)
(* bcast_direct_rpl: α par→int→α  *)
let bcast_direct_rpl = at
(* bcast_direct: α par→int→α par*)
let bcast_direct vv root = replicate (bcast_direct_rpl vv root)
```

Certaines fonctions de la bibliothèque standard sont récursives mais restent plates (sans composition parallèle), comme la diffusion en $\log p$ m-étapes :

```
(* bcast_logp: int→α par→α par *)
let bcast_logp root vv =
  let from n =
    mkpar(fun i→let j=natmod (i+(p())−root) (p()) in
      if (n/2<=j)&&(j<n) then i−(n/2) else i) in
```

[1] Dans l'implantation actuelle, pour des raisons historiques, cette fonction est une primitive du langage.

```
  let rec aux n vv =
    if n<1 then vv else get (aux (n/2) vv) (from n)
  in aux (p()) vv
```

On peut aussi programmer deux types d'échange total, dont l'un est répliqué :

```
(* val totex: α par→(int→α )par *)
let totex vv =
 parfun (compose noSome) (mget (parfun (fun v i→v) vv) (replicate (fun _ →true)))

(* val totex_rpl: α par→int→α *)
let totex_rpl vv =
 let rcv=(mat (parfun (fun v →Some v) vv) (fun _ →true)) in
  (compose noSome f)
```

Son coût parallèle est $(p-1) \times s \times g + L$ où s est la taille de la plus grande des valeurs contenues par les processeurs.

Notons que l'on peut tout à fait implanter les primitives **mget** et **mat** en BSML. Le chapitre 2 le montre d'ailleurs pour **mget**. Toutefois, les programmes MSPML ainsi exécutés, seront beaucoup moins efficaces qu'avec l'emploi direct des primitives du langage. En effet, les implantations BSML nécessitent deux super-étapes BSP et donc deux barrières de synchronisation pour l'exécution d'une seule primitive. A l'inverse, on peut écrire entièrement les primitives **put** et **proj** en MSPML. Par exemple, **put** peut être codé ainsi :

```
let put f =
 let values_to_send = parfun2 List.map f (replicate (procs())) in
 let send values =
  let procs_at n = applyat n (fun _ →procs()) (fun _ →[]) in
  let rec aux l n =
   if n=mpm_p() then replicate []
    else let h= parfun List.hd values and t=parfun List.tl values in
          let v=get_list h (procs_at n) in
           parfun2 List.append vl (aux t (n+1))
  in
     aux values 0 in
  parfun List.nth (send values)
```

Pour le moment, la différence principale dans l'écriture des programmes est donc simplement la différence de style pour les communications. À partir du moment où l'on n'utilise plus les primitives de communication pour écrire des programmes, mais des fonctions de communication comme bcast_direct, les programmes seront identiques. Toutefois, leurs coûts seront différents suivant qu'on utilise MSPML ou BSML.

10.4 Sémantiques

Cette section traite de la sémantique formelle de MSPML. Les expressions données par le programmeur sont donc similaires à celles de notre mini-BSML (voir au chapitre 3), excepté le fait que les primitives parallèles diffèrent. La première sémantique est celle qui correspond au modèle de programmation. Elle est semblable à la sémantique naturelle de BSML. La deuxième sémantique est aussi proche de celle de BSML, mais les coûts y seront différents. La dernière sémantique est dans le même esprit que l'évaluation distribuée du chapitre 3 et correspond donc au modèle d'exécution. Les différences entre les sémantiques distribuées des deux langages apparaîtront au niveau des opérations de communication.

10.4.1 Syntaxe d'un mini langage parallèle applicatif

Raisonner sur une définition complète et exhaustive d'un langage fonctionnel et parallèle comme MSPML serait trop complexe. Pour notre propos et afin de simplifier la présentation, cette section introduit un mini langage applicatif qui peut être considéré comme un noyau de MSPML. Ce mini-langage a la forme d'un calcul mais n'en est pas un, à proprement parler, puisqu'une stratégie d'évaluation sera donnée. Nous ferons

donc un abus de langage en l'appelant calcul par opposition à MSPML. Par la suite, nos termes seront notés avec éventuellement des indices[2].

Définition 30 (*Langage source*).
Les expressions initiales sont définies par la grammaire suivante :

e^p	::=	$\lambda x.e^p$	*abstraction (fonction)*
	|	$(e^p\ e^p)$	*application*
	|	(e^p, e^p)	*paire*
	|	$\mathbf{mkpar}\ e^p$	*création d'un vecteur parallèle*
	|	$\mathbf{apply}\ e^p\ e^p$	*application parallèle*
	|	$\mathbf{mget}\ e^p$	*communication*
	|	$\mathbf{mat}\ e^p$	*projection globale*
	|	$\mathbf{c}$	*constante*
	|	$\mathbf{op}$	*opération prédéfinie*
	|	$\mathbf{x}$	*variable*
	|	$\mu x.e^p$	*définition d'une fonction récursive*
	|	$\mathbf{if}\ e^p\ \mathbf{then}\ e^p\ \mathbf{else}\ e^p$	*conditionnelle*

Ces expressions sont celles données par le «programmeur». Ce sont donc les termes de notre mini-langage de base. L'ensemble des constantes contient par exemple les entiers, les booléens etc. Nous utilisons aussi la constante $\mathbf{nc}$ (pour «no-communication») qui correspondra au None de OCaml. Les opérateurs peuvent être les opérations arithmétiques, logiques etc. Notons que $\lambda x.e^p$ se traduit en OCaml par **fun** x →e et $\mu f.e$ par (**let rec** f = e **in** f).

Comme pour BSML (et pour les mêmes raisons), nous utilisons des substitutions explicites dans nos sémantiques.

Définition 31 (*Expressions des sémantiques*).
Nos expressions ont donc la forme suivante :

e	::=	$(e)[s]$	*expression munie d'une substitution*
	|	$\lambda.e$	*abstraction*
	|	$\overline{(\lambda.e)[s]}$	*fermeture*
	|	$(e\ e)$	*application*
	|	(e, e)	*paire*
	|	$\mathbf{mkpar}\ e$	*création d'un vecteur parallèle*
	|	$\mathbf{apply}\ e\ e$	*application parallèle*
	|	$\mathbf{mget}\ e$	*communication*
	|	$\mathbf{mat}\ e$	*projection globale*
	|	$\mathbf{c}$	*constante*
	|	$\mathbf{op}'$	*opérateur*
	|	$\overline{\mathbf{n}}$	*variables de substitution (indices)*
	|	$\mu.e$	*définition d'une fonction récursive*
	|	$\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$	*conditionnelle*
	|	$\langle e, \ldots, e \rangle$	*vecteur parallèle de taille p*
	|	$[e, \ldots, e]$	*tableau fonctionnel*

où $\mathbf{op}' ::= \mathbf{op} \cup \{\mathbf{delpar}, \mathbf{init}, \mathbf{access}, \mathbf{initthread}, \mathbf{get}, \mathbf{build}\}$.

Notons que nous avons ajouté les vecteurs parallèles de taille p fixe, ainsi que des tableaux (qui seront purement fonctionnels, c'est-à-dire sans effets de bord). Nous aurons donc une sémantique par valeur de p. L'ensemble des opérateurs est étendu avec des opérations internes au calcul : la suppression du constructeur de vecteur parallèle $\langle\rangle$, la création d'un tableau purement fonctionnel, l'accès à ces valeurs, la création de processus légers de communication, la réception d'une valeur, la suppression de ces processus légers. Notons que, comme dans BSML, nous différencions une abstraction (avec sa substitution) $(\lambda.e)[s]$ et une fermeture.

[2]Nous utiliserons un parenthèsage dit prioritaire à gauche (comme en OCaml) afin d'améliorer la lisibilité de nos règles.

$$\begin{array}{rcl|rcl}
\mathcal{T}_{\mathcal{E}}(\lambda x.e^p) & = & \lambda.T_{\{x\mapsto\overline{1},R_x(\mathcal{E})\}}(e^p) & \mathcal{T}_{\mathcal{E}}(\mathbf{apply}\ e_1^p\ e_2^p) & = & \mathbf{apply}\mathcal{T}_{\mathcal{E}}(e_1^p)\ \mathcal{T}_{\mathcal{E}}(e_2^p) \\
\mathcal{T}_{\mathcal{E}}(\mu x.e^p) & = & \mu.T_{\{x\mapsto\overline{1},R_x(\mathcal{E})\}}(e^p) & \mathcal{T}_{\mathcal{E}}(\mathbf{mget}\ e^p) & = & \mathbf{mget}\ \mathcal{T}_{\mathcal{E}}(e^p) \\
\mathcal{T}_{\mathcal{E}}((e_1^p\ e_2^p)) & = & (\mathcal{T}_{\mathcal{E}}(e_1^p)\ \mathcal{T}_{\mathcal{E}}(e_2^p)) & \mathcal{T}_{\mathcal{E}}(\mathbf{mat}\ e^p) & = & \mathbf{mat}\ \mathcal{T}_{\mathcal{E}}(e^p) \\
\mathcal{T}_{\mathcal{E}}((e_1^p, e_2^p)) & = & (\mathcal{T}_{\mathcal{E}}(e_1^p), \mathcal{T}_{\mathcal{E}}(e_2^p)) & \mathcal{T}_{\mathcal{E}}(\mathbf{c}) & = & \mathbf{c} \\
\mathcal{T}_{\mathcal{E}}(\mathbf{mkpar}\ e^p) & = & \mathbf{mkpar}\ \mathcal{T}_{\mathcal{E}}(e^p) & \mathcal{T}_{\mathcal{E}}(\mathbf{op}) & = & \mathbf{op}
\end{array}$$

$$\mathcal{T}_{\mathcal{E}}(\mathbf{if}\ e_1^p\ \mathbf{then}\ e_2^p\ \mathbf{else}\ e_3^p) = \mathbf{if}\ \mathcal{T}_{\mathcal{E}}(e_1)\ \mathbf{then}\ \mathcal{T}_{\mathcal{E}}(e_2)\ \mathbf{else}\ \mathcal{T}_{\mathcal{E}}(e_3)$$

$$\mathcal{T}_{\mathcal{E}}(x) = \overline{n} \text{ si } \mathcal{E} = \{\cdots, x \mapsto \overline{n}, \cdots, \bullet\}$$

Figure 10.3 — Instanciation des variables en des indices de De Brujin

Définition 32 (*substitutions et valeurs*).
Les substitutions et les valeurs (sous-ensemble des expressions) sont classiquement définies, comme dans BSML, par :

$$\begin{array}{rcll}
s & ::= & \bullet & \textit{substitution vide} \\
 & | & v \circ s & \textit{valeur suivie de la suite de la substitution}
\end{array}$$

$$v \quad ::= \quad \mathbf{op} \quad | \quad \mathbf{c} \quad | \quad \overline{(\lambda.e)[s]} \quad | \quad (v, v) \quad | \quad \langle v, \ldots, v \rangle$$

La traduction des «expressions du programmeur» en nos expressions, se définit inductivement (figure 10.3) comme dans BSML. $\mathcal{E}$ est un environnement de substitution des variables (un dictionnaire entre les variables et les indices) défini par :

$$\mathcal{E} ::= \bullet \ | \ \{x \mapsto \overline{n}, \mathcal{E}\}$$

avec la fonction suivante de mise à jour de l'environnement :

$$\begin{array}{rcl}
\mathcal{R}_x(\bullet) & = \bullet & \\
\mathcal{R}_x(\{x \mapsto \overline{n}, \mathcal{E}\}) & = & \mathcal{R}_x(\mathcal{E}) \\
\mathcal{R}_x(\{y \mapsto \overline{n}, \mathcal{E}\}) & = & \{y \mapsto \overline{n+1}, \mathcal{R}_x(\mathcal{E})\} \quad \text{si } y \neq x
\end{array}$$

Propriété 3

Soit l'expression e^p telle que $e = \mathcal{T}_\bullet(e^p)$, alors e^p est sans variables libres (ainsi que e par conséquent).

Preuve . Par induction triviale sur la traduction de l'expression e^p. ■

Notons que, considérant notre calcul comme le noyau de notre langage MSPML, le fait de ne pas avoir de variables libres n'est pas un souci du point de vue de l'expressivité : elles seront de toutes façons systématiquement rejetées par un compilateur lors de l'analyse statique du terme (ou lors de sa compilation).

10.4.2 Sémantique naturelle

Nous pouvons maintenant définir la sémantique naturelle de notre calcul ; elle est exprimée à l'aide d'une relation d'induction.

Nous définissons deux types de relations : $\rhd_g$ pour la réduction globale de l'expression et $\rhd_i$ pour la réduction locale au processeur i. Comme un grand nombre de règles sont communes, nous avons utilisé un ensemble de règles $\rhd$ génériques aux deux réductions. Nous avons aussi les règles $\rhd_\bowtie$ pour les primitives parallèles. Les règles sont toutes de la forme $s, e \rhd v$, qui peut se lire comme «dans l'environnement s (défini comme une substitution), l'expression e s'évalue en la valeur v». Les règles de la sémantique naturelle sont similaires à celles du chapitre 3. Nous ne les redétaillons pas. Nous les rappelons (pour MSPML) dans la figure 10.4. Par contre, nous détaillons celles qui correspondent aux nouvelles primitives de communication.

Gestion des substitutions :

$$\frac{s, \overline{n} \rhd v}{v' \circ s, \overline{n+1} \rhd v} \tag{10.1}$$

$$\frac{}{v \circ s, \overline{1} \rhd v} \tag{10.2}$$

$$\frac{}{s, \lambda.e \rhd \overline{(\lambda.e)[s]}} \tag{10.3}$$

$$\frac{\text{si } v \neq \langle \cdots \rangle \text{ et } v \neq (v_0, v_1)}{s, v \rhd v} \tag{10.4}$$

Noyau fonctionnel :

$$\frac{s, e_1 \rhd v_1 \qquad s, e_2 \rhd v_2}{s, (e_1, e_2) \rhd (v_1, v_2)} \tag{10.5}$$

$$\frac{s, e_1 \rhd \overline{(\lambda.e)[s']} \qquad s, e_2 \rhd v' \qquad v' \circ s', e \rhd v}{s, (e_1\ e_2) \rhd v} \tag{10.6}$$

$$\frac{s, e_1 \rhd_l \mathbf{op} \quad s, e_2 \rhd_l v \quad s, \overline{\mathbf{op}(v)} \rhd_l v'}{s, (e_1\ e_2) \rhd_l v'} \qquad \frac{s, e_1 \rhd_g \mathbf{op} \quad s, e_2 \rhd_g v \quad s, \overline{\mathbf{op}(v)} \rhd_g v'}{s, (e_1\ e_2) \rhd_g v'} \tag{10.7}$$

$$\frac{\forall i \in \{0, \ldots, p-1\} \quad s, e_i \rhd v_i}{s, [e_0, \ldots, e_{p-1}] \rhd [v_0, \ldots, v_{p-1}]} \tag{10.8}$$

$$\frac{(\mu.e) \circ s, e \rhd v}{s, \mu.e \rhd v} \tag{10.9}$$

$$\frac{s, e_1 \rhd \mathbf{true} \qquad s, e_2 \rhd v}{s, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd v} \tag{10.10}$$

$$\frac{s, e_1 \rhd \mathbf{false} \qquad s, e_3 \rhd v}{s, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd v} \tag{10.11}$$

Primitives parallèles :

$$\frac{s, e \rhd_\bowtie f \qquad \bullet, \langle (f\ 0), \ldots, (f\ (p-1)) \rangle [\bullet] \rhd_\bowtie \langle v_0, \ldots, v_{p-1} \rangle \quad \text{si } \mathcal{V}_\bullet(f) = \mathbf{true}}{s, \mathbf{mkpar}\ e \rhd_\bowtie \langle v_0, \ldots, v_{p-1} \rangle} \tag{10.12}$$

$$\frac{\forall i \in \{0, \ldots, p-1\} \quad \bullet, e_i \rhd_i v_i}{\bullet, \langle e_0, \ldots, e_{p-1} \rangle \rhd_\bowtie \langle v_0, \ldots, v_{p-1} \rangle} \tag{10.13}$$

$$\frac{s, e_1 \rhd_\bowtie \langle f_0, \ldots, f_{p-1} \rangle \quad s, e_2 \rhd_\bowtie \langle v_0, \ldots, v_{p-1} \rangle \quad \bullet, \langle (f_0\ v_0), \ldots, (f_{p-1}\ v_{p-1}) \rangle \rhd_\bowtie \langle v'_0, \ldots, v'_{p-1} \rangle}{s, \mathbf{apply}\ e_1\ e_2 \rhd_\bowtie \langle v'_0, \ldots, v'_{p-1} \rangle} \tag{10.14}$$

(10.15)

Figure 10.4 — Règles de la sémantique naturelle de mini-MSPML

mget. Pour la première primitive, $\mathbf{mget}$, nous avons la règle suivante

$$\frac{s, e_1 \rhd_{\bowtie} vf \quad s, e_2 \rhd_{\bowtie} vb \quad \bullet, (\mathbf{build}\ (\mathbf{prep}\ (tv, tb)))[\bullet] \rhd_{\bowtie} \langle g_0, \ldots, g_{p-1}\rangle}{s, \mathbf{mget}\ e_1\ e_2 \rhd_{\bowtie} \langle g_0, \ldots, g_{p-1}\rangle}$$

avec

$$\begin{array}{rclcrcl} vf & = & \langle f_0, \ldots, f_{p-1}\rangle & & tv & = & \langle(\mathbf{init}\ (f_0, p)), \ldots, (\mathbf{init}\ (f_{p-1}, p))\rangle \\ vb & = & \langle b_0, \ldots, b_{p-1}\rangle & & tb & = & \langle(\mathbf{init}\ (b_0, p)), \ldots, (\mathbf{init}\ (b_{p-1}, p))\rangle \end{array}$$

et où $\mathbf{prep}$ (donné en syntaxe ML pour être plus lisible) est :

```
fun tvtb →
 initthreads (fun i →get (i,(parfun (fun t →access (t,i)) (fst tvtb),
                              parfun (fun t →access (t,i)) (snd tvtb))))
```

et dans notre syntaxe (notons que nous avons remplacé parfun par sa définition) :

$$\lambda.\mathbf{initthreads}\ (\lambda.\mathbf{get}\ (\overline{1}, ((\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.\mathbf{access}\ (\overline{1}, \overline{3})))\ (\mathbf{fst}\ \overline{2})), (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.\mathbf{access}\ (\overline{1}, \overline{3})))\ (\mathbf{snd}\ \overline{2}))))$$

Dans cette règle, nous créons p «processus légers» qui seront chargés de lire les valeurs des autres processus (opération $\mathbf{get}$). Ces «processus légers» sont créés par l'opération $\mathbf{initthreads}$. L'ensemble des résultats est récolté par l'opération $\mathbf{buid}$, laquelle construit le vecteur final avec les valeurs reçues par les processeurs. Chaque $\mathbf{get}$ est donc exécuté séparément et cela permet à chaque processeur de lire (ou non, si cette valeur n'est pas demandée) la valeur d'un seul autre processeur i.
Nous avons alors les nouveaux opérateurs suivants :

$$\begin{array}{rcl} s, \mathbf{initthreads}\ f & \rhd_{\bowtie} & [\![(f\ 0), \ldots, (f\ (p-1))]\!] \\ s, \mathbf{build}\ [\![\langle v_0^0, \ldots, v_{p-1}^0\rangle, \ldots, \langle v_0^{p-1}, \ldots, v_{p-1}^{p-1}\rangle]\!] & \rhd_{\bowtie} & \langle g_0, \ldots, g_{p-1}\rangle[\bullet] \\ s, \mathbf{get}\ i\ \langle v_0, \ldots, v_{p-1}\rangle\ \langle b_0, \ldots, b_{p-1}\rangle & \rhd_{\bowtie} & \langle v'_0, \ldots, v'_{p-1}\rangle \text{ tel que } \begin{cases} v'_j & = & v_i \text{ si } b_j = \mathbf{true} \\ v'_j & = & \mathbf{nc} \text{ sinon} \end{cases} \end{array}$$

où $\forall i \in \{0, \ldots, p-1\}$, $g_i = (\lambda.\mathbf{if}\ (0 \leq \overline{1} < p-1)\ \mathbf{then}\ (\mathbf{access}\ (\overline{1}, [v_i^0, \ldots, v_i^{p-1}]))\ \mathbf{else}\ \mathbf{nc})$.
La règle sur les «processus légers» est la suivante :

$$\frac{\forall i \in \{0, \ldots, p-1\} \qquad s, e_i \rhd_{\bowtie} v_i}{s, [\![e_0, \ldots, e_{p-1}]\!] \rhd_{\bowtie} [\![v_0, \ldots, v_{p-1}]\!]}$$

mat. La deuxième primitive, $\mathbf{mat}$, a la règle suivante :

$$\frac{s, e_1 \rhd_{\bowtie} v \quad s, e_2 \rhd_{\bowtie} f \quad \bullet, (\mathbf{delpar}\ (\mathbf{mget}\ \mathit{vf}\ \langle f, \ldots, f\rangle))[\bullet] \rhd_{\bowtie} g}{s, \mathbf{mat}\ e_1\ e_2 \rhd_{\bowtie} g}$$

avec :

$$\begin{array}{rcl} v & = & \langle v_0, \ldots, v_{p-1}\rangle \\ \mathit{vf} & = & \langle(\lambda.\mathbf{if}\ (f\ \overline{1})\ \mathbf{then}\ v_0\ \mathbf{else}\ \mathbf{nc}), \ldots, (\lambda.\mathbf{if}\ (f\ \overline{1})\ \mathbf{then}\ v_{p-1}\ \mathbf{else}\ \mathbf{nc})\rangle \end{array}$$

Comme précédemment, la primitive de projection se construit avec la primitive de communication et avec une suppression du constructeur des vecteurs parallèles. Nous définissons alors :

Définition 33 (*Sémantique naturelle*).
La sémantique naturelle est définie par : $\rhd_i = \rhd$ *et* $\rhd_g = \rhd \cup \rhd_{\bowtie}$.

et nous avons le résultat suivant :

Théorème 14 (*déterminisme*)

Si $e = \mathcal{T}_{\bullet}(e^p)$ *et si* $\bullet, e \rhd_g v_1$ *et* $\bullet, e \rhd_g v_2$ *alors* $v_1 = v_2$.

Preuve. Par induction triviale sur la réduction (confère chapitre 3). ■

Prenons par exemple, l'expression e^p (programmeur) de diffusion (depuis le processeur 0) suivante :

(**apply** (**mkpar** (**fun** i→**fun** g→(g 0)))
(**mget** (**mkpar** (**fun** i→**fun** j→i)) (**mkpar** (**fun** i→**fun** j→j=0))))

Nous obtenons par $\mathcal{T}_\bullet$ l'expression e suivante :

$$(\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))\ (\mathbf{mget}\ (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2}))\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0))))$$

La figure 10.1 donne l'évaluation de cette expression avec la sémantique naturelle (avec trois processeurs) de notre mini-langage.

Cette sémantique, bien que très simple, n'est pas suffisante pour les mêmes raisons qu'au chapitre 3 : il est impossible de raisonner sur des programmes qui ne terminent pas et toutes les opérations semblent synchrones. Il nous faut donc une autre sémantique.

10.4.3 Sémantique à «petits pas»

Définition 34 (*Coûts*).
L'algèbre de coûts que nous utilisons est toujours celle des entiers munie de l'addition $\oplus$, de la multiplication $\otimes$ usuelles (donc toutes deux communicatives et associatives) et de deux constantes g et L.

Ces constantes correspondent aux paramètres g et L du modèle MPM qui seront donc des entiers (ce qui n'est pas un problème en soit car nous donnerons des coûts symboliques et non des coûts réels).

Notre sémantique à «petits pas» a la forme suivante :

$$e/\langle c_0^m,\ldots,c_{p-1}^m\rangle/\langle c_0,\ldots,c_{p-1}\rangle \rightharpoonup e/\langle c'^m_0,\ldots,c'^m_{p-1}\rangle/\langle c'_0,\ldots,c'_{p-1}\rangle$$

où e est une expression, $\langle c_0,\ldots,c_{p-1}\rangle$ sont les coûts en chaque processeur de la m-étape courante et $\langle c_0^m,\ldots,c_{p-1}^m\rangle$ sont les coûts des m-étapes précédentes. Nous notons $\langle c^m\rangle$ les coûts des m-étapes précédentes lorsqu'ils ne sont pas utilisés par la règle.

Notons tout de suite la différence entre ces coûts et ceux de BSML. Dans le modèle MPM, du fait de son asynchronisme, les coûts sont tous locaux. Le coût final du programme ne peut être donné qu'à la toute fin. Dans le modèle BSP, à la fin de chaque super-étape, on peut prendre directement le maximum des coûts locaux et on a une barrière implicite à la fin du programme. En BSP, l'opérateur **send** effectue toutes les communications et la barrière de synchronisation. Les coûts locaux, ainsi que les communications, sont alors ajoutés au coût global qui est le coût global du programme. En MPM, le coût global ne peut être donné qu'à la toute fin de l'exécution du programme. Les coûts des communications ne sont ajoutés aux coûts de chaque processeur qu'à la fin de la m-step, c'est-à-dire lorsque l'opérateur **buid** est exécuté. En quelque sorte, les coûts locaux correspondent donc aux $\Phi_{(s,j)}$ du modèle MPM.

Nous notons $\stackrel{*}{\rightharpoonup}$ la fermeture transitive et réflexive de $\rightharpoonup$. L'évaluation complète d'une expression e^p donnée par le programmeur sera :

$$(\mathcal{T}_\bullet(e^p))[\bullet]/\langle 0,\ldots,0\rangle/\langle 0,\ldots,0\rangle \stackrel{*}{\rightharpoonup} v/\langle c_0^m,\ldots,c_{p-1}^m\rangle/\langle c_0,\ldots,c_{p-1}\rangle$$

Cela peut être lu comme «dans un environnement (de «traduction») vide, une substitution vide et des coûts zéro, l'expression traduite de e^p s'évalue en v pour des coûts locaux $\langle c_0,\ldots,c_{p-1}\rangle$ et des coûts des m-étapes $\langle c_0^m,\ldots,c_{p-1}^m\rangle$». Le temps d'exécution final MPM du programme e^p sera alors de $\max_{i=0}^{p-1}(c_i \oplus c_i^m)$ (on a une barrière de synchronisation implicite à la fin du programme).

Définition 35 (*Relations de la sémantique à «petits pas»*).
Pour définir la relation $\rightharpoonup$ nous définissons préalablement deux types de réduction :

1. *$\stackrel{i}{\rightharpoonup}$ est la réduction locale d'une expression (dans une composante d'un vecteur parallèle)*
2. *$\stackrel{\bowtie}{\rightharpoonup}$ est la réduction globale d'une expression (en dehors d'un vecteur parallèle).*

avec :

$$\stackrel{i}{\rightharpoonup} = \underset{i}{\stackrel{\varepsilon}{\rightharpoonup}} \cup \underset{i}{\stackrel{\delta}{\rightharpoonup}} \qquad \stackrel{\bowtie}{\rightharpoonup} = \underset{\bowtie}{\stackrel{\varepsilon}{\rightharpoonup}} \cup \underset{\bowtie}{\stackrel{\delta}{\rightharpoonup}}$$

Nous commençons d'abord par une série d'axiomes (règles) qui sont communs aux deux relations. Ceux-ci sont de la forme $e \rightharpoonup e', c$. Nous avons tout d'abord les relations de réduction de la β-réduction (avec

$$\dfrac{\dfrac{\dfrac{\dots}{\bullet, (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0))) \rhd \langle f_0^1, f_1^1, f_2^1\rangle} \quad \dfrac{\dots}{\bullet, (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2})) \rhd \langle f_0^2, f_1^2, f_2^2\rangle} \quad \dfrac{\dots}{\bullet, (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1}, 0))) \rhd \langle f_0^3, f_1^3, f_2^3\rangle} \quad \dfrac{\text{lire ci-dessous}}{\bullet, (\mathbf{build}\ (\overline{\mathbf{prep}}[\bullet])\ (tv, tb)) \rhd}}{\bullet, (\mathbf{mget}\ (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2}))\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1}, 0)))) \rhd \langle g_0, g_1, g_2\rangle}}{\bullet, (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))\ (\mathbf{mget}\ (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2}))\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1}, 0))))) \rhd \langle 0, 0, 0\rangle}$$

avec

$$\forall i \in \{0,1,2\} \begin{cases} f_i^1 & = & (\overline{\lambda.(\overline{1}\ 0)})[i \circ \bullet] \\ f_i^2 & = & (\overline{\lambda.\overline{2}})[i \circ \bullet] \\ f_i^3 & = & (\lambda.\ =(\overline{1}, 0))[i \circ \bullet] \end{cases}$$

et

$$\begin{array}{rcl} tv & = & \langle (\mathbf{init}\ f_0^2), (\mathbf{init}\ f_1^2), (\mathbf{init}\ f_2^2)\rangle \\ tb & = & \langle (\mathbf{init}\ f_0^3), (\mathbf{init}\ f_1^3), (\mathbf{init}\ f_2^3)\rangle \end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\bullet, \overline{\mathbf{prep}[\bullet]} \rhd \overline{\mathbf{prep}[\bullet]}} \quad \dfrac{\dots}{\bullet, (tv, tb) \rhd (tv, tb)} \quad \dfrac{\dfrac{}{(tv, tb) \circ \bullet, (\lambda.\mathbf{get}\ (\overline{1}, ((\cdots), (\cdots)))) \rhd \overline{(\lambda.\mathbf{get}\ (\overline{1}, ((\cdots), (\cdots))))[(tv, tb) \circ \bullet]}} \quad \dfrac{\text{voir ci-dessous}}{(tv, tb) \circ \bullet, [\![(f\ 0), (f\ 1), (f\ 2)]\!] \rhd [\![\langle 0, \mathbf{nc}, \mathbf{nc}\rangle, \langle 0, \mathbf{nc}, \mathbf{nc}\rangle, \langle 0, \mathbf{nc}, \mathbf{nc}\rangle]\!]}}{(tv, tb) \circ \bullet, \mathbf{initthreads}\ (\lambda.\mathbf{get}\ (\overline{1}, ((\cdots), (\cdots)))) \rhd [\![\langle 0, \mathbf{nc}, \mathbf{nc}\rangle, \langle 0, \mathbf{nc}, \mathbf{nc}\rangle, \langle 0, \mathbf{nc}, \mathbf{nc}\rangle]\!]}}{\bullet, (\overline{\mathbf{prep}})[\bullet]\ (tv, tb) \rhd [\![\langle 0, \mathbf{nc}, \mathbf{nc}\rangle, \langle 0, \mathbf{nc}, \mathbf{nc}\rangle, \langle 0, \mathbf{nc}, \mathbf{nc}\rangle]\!]}}{\bullet, (\mathbf{build}\ (\overline{\mathbf{prep}})[\bullet]\ (tv, tb)) \rhd \langle g_0, g_1, g_2\rangle}$$

$$\forall i \in \{0,1,2\} \quad \dfrac{\dfrac{\dots}{(tv, tb) \circ \bullet, (\overline{1}, ((\cdots), (\cdots))) \rhd (i, (\langle [0,0,0], [1,1,1], [2,2,2]\rangle, \langle [\mathbf{true}, \mathbf{false}, \mathbf{false}], [\mathbf{true}, \mathbf{false}, \mathbf{false}], [\mathbf{true}, \mathbf{false}, \mathbf{false}]\rangle))}}{(tv, tb) \circ \bullet, \mathbf{get}\ (\overline{1}, ((\cdots), (\cdots))) \rhd \langle 0, \mathbf{nc}, \mathbf{nc}\rangle}$$

$$(tv, tb) \circ \bullet, [\![(f\ 0), (f\ 1), (f\ 2)]\!] \rhd [\![\langle 0, \mathbf{nc}, \mathbf{nc}\rangle, \langle 0, \mathbf{nc}, \mathbf{nc}\rangle, \langle 0, \mathbf{nc}, \mathbf{nc}\rangle]\!]$$

Table 10.1 — Exemple de la sémantique naturelle de MSPML

$$
\begin{array}{rcl}
(\overline{(\lambda.e)[s]}\ v) & \xrightarrow{\varepsilon} & e[v \circ s], 1 \\
\overline{n+1}[v \circ s] & \xrightarrow{\varepsilon} & \overline{n}[s], 0 \\
\overline{1}[v \circ s] & \xrightarrow{\varepsilon} & v[\bullet], 1 \\
(\mu.e)[s] & \xrightarrow{\varepsilon} & e[\mu.e \circ s], 1 \\
(\lambda.e)[s] & \xrightarrow{\varepsilon} & \overline{(\lambda.e)[s]}, 1 \\
v[s] & \xrightarrow{\varepsilon} & v, 0 \quad \text{si } v \neq \langle \cdots \rangle \text{ et } v \neq (v_0, v_1) \\
(e_1\ e_2)[s] & \xrightarrow{\varepsilon} & (e_1[s]\ e_2[s]), 0 \\
(e_1, e_2)[s] & \xrightarrow{\varepsilon} & (e_1[s], e_2[s]), 0 \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)[s] & \xrightarrow{\varepsilon} & \textbf{if } e_1[s] \textbf{ then } e_2[s] \textbf{ else } e_3[s], 0 \\
(\textbf{proj } e)[s] & \xrightarrow{\varepsilon} & (\textbf{proj } e[s]), 0 \\
(\textbf{put } e)[s] & \xrightarrow{\varepsilon} & (\textbf{put } e[s]), 0 \\
(\textbf{apply } e_1\ e_2)[s] & \xrightarrow{\varepsilon} & (\textbf{apply } e_1[s]\ e_2[s]), 0 \\
(\textbf{mkpar } e)[s] & \xrightarrow{\varepsilon} & (\textbf{mkpar } e[s]), 0 \\
(\textbf{super } e_1\ e_2)[s] & \xrightarrow{\varepsilon} & (\textbf{super } e_1[s]\ e_2[s]), 0 \\
\langle e_0, \ldots, e_{p-1} \rangle[s] & \xrightarrow{\varepsilon} & \langle e_0[s], \ldots, e_{p-1}[s] \rangle, 0 \\
[e_0, \ldots, e_{p-1}][s] & \xrightarrow{\varepsilon} & [e_0[s], \ldots, e_{p-1}[s]], 0
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{access } ([v_0, \ldots, v_i, \ldots, v_{p-1}], i) & \xrightarrow{\delta} & v_i, 1 \\
\textbf{init } (n, f) & \xrightarrow{\delta} & [(f\ 0), \ldots, (f\ (n-1))], 1 \\
\textbf{fst } (v_1, v_2) & \xrightarrow{\delta} & v_1, 1 \\
\textbf{snd } (v_1, v_2) & \xrightarrow{\delta} & v_2, 1 \\
+\ (n_1, n_2) & \xrightarrow{\delta} & n_1 + n_2, 1 \\
\textbf{isnc nc} & \xrightarrow{\delta} & \textbf{true}, 1 \\
\textbf{isnc } v & \xrightarrow{\delta} & \textbf{false}, 1 \text{ si } v \neq \textbf{nc} \\
\textbf{LocId } v & \xrightarrow[i]{\delta} & v, 1 \\
\textbf{if true then } e_2 \textbf{ else } e_3 & \xrightarrow{\delta} & e_2, 1 \\
\textbf{if false then } e_2 \textbf{ else } e_3 & \xrightarrow{\delta} & e_3, 1
\end{array}
$$

Figure 10.5 — Réductions fonctionnelles et opérations génériques (rappel)

$$\textbf{mkpar } f/\langle c^m \rangle/\langle c_0, \ldots, c_{p-1} \rangle \xrightarrow[\bowtie]{\delta} \langle (f\ 0), \ldots, (f\ (p-1)) \rangle[\bullet]/\langle c^m \rangle/\langle c_0 \oplus 1, \ldots, c_{p-1} \oplus 1 \rangle \quad \text{si } \mathcal{V}_\bullet(f) = \textbf{true} \tag{10.16}$$

$$\textbf{apply}\langle f_0, \ldots, f_{p-1} \rangle\ \langle v_0, \ldots, v_{p-1} \rangle/\langle c^m \rangle/\langle c_0, \ldots, c_{p-1} \rangle \xrightarrow[\bowtie]{\delta} \langle (f_0\ v_0), \ldots, (f_{p-1}\ v_{p-1}) \rangle[\bullet]/\langle c^m \rangle/\langle c_0 \oplus 1, \ldots, c_{p-1} \oplus 1 \rangle \tag{10.17}$$

$$\textbf{mget } \langle f_0, \ldots, f_{p-1} \rangle\ \langle b_0, \ldots, b_{p-1} \rangle/\langle c^m \rangle/\langle c_0, \ldots, c_{p-1} \rangle \xrightarrow[\bowtie]{\delta} (\textbf{build } (\textbf{prep } (tv, tb)))[\bullet]/\langle c^m \rangle/\langle c_0 \oplus 1, \ldots, c_{p-1} \oplus 1 \rangle \tag{10.18}$$

$$\textbf{mat } \langle v_0, \ldots, v_{p-1} \rangle\ f/\langle c^m \rangle/\langle c_0, \ldots, c_{p-1} \rangle \xrightarrow[\bowtie]{\delta} (\textbf{delpar } (\textbf{mget } \mathit{vf}\ \langle f, \ldots, f \rangle))[\bullet]/\langle c^m \rangle/\langle c_0 \oplus 1, \ldots, c_{p-1} \oplus 1 \rangle \tag{10.19}$$

avec

$$
\begin{array}{rcl}
tv & = & \langle (\textbf{init } f_0), \ldots, (\textbf{init } f_{p-1}) \rangle \\
tb & = & \langle (\textbf{init } b_0), \ldots, (\textbf{init } b_{p-1}) \rangle \\
\mathit{vf} & = & \langle (\lambda.\textbf{if } (f\ \overline{1}) \textbf{ then } v_0 \textbf{ else nc}), \ldots, (\lambda.\textbf{if } (f\ \overline{1}) \textbf{ then } v_{p-1} \textbf{ else nc}) \rangle
\end{array}
$$

Figure 10.6 — Règles des primitives MSPML parallèles

substitution explicite sur une fermeture), les δ-réductions correspondant aux opérateurs fonctionnels prédéfinis et les règles de propagation des substitutions. Les règles de $\xrightarrow{i}$ et $\xrightarrow{\bowtie}$ sont définies comme dans le chapitre 3. Nous les redonnons (pour notre sémantique) à la figure 10.5.

La figure 10.6 donne les δ-règles des primitives parallèles. Celles-ci sont très proches de la définition des règles de la sémantique naturelle. Les règles 10.16 et 10.17 sont sans surprise. Elles sont similaires à celles de BSML.

La règle 10.18 de la primitive **mget** permet la création des processus légers de communications qui iront chacun lire (ou non, suivant le paramètres défini dans tb) une valeur d'un autre processeur. **prep** (défini précédemment) prend deux vecteurs : le premier tv avec les valeurs à envoyer et le second tb avec les demandes de communications. Ces vecteurs sont créés avec les fonctions f_i et b_i et ceux pour tous les processeurs. Ensuite, dans **prep**, p processus légers de communications sont exécutés, chaque processus

$$\begin{array}{rcl|rcl}
\mathcal{S}_s(e_1\ e_2) & = & \mathcal{S}_s(e_1) \oplus \mathcal{S}_s(e_2) \oplus 1 & \mathcal{S}_s(\lambda.e) & = & \mathcal{S}_{(x \circ s)}(e) \oplus 1 \\
\mathcal{S}_s(e_1, e_2) & = & \mathcal{S}_s(e_1) \oplus \mathcal{S}_s(e_2) \oplus 1 & \mathcal{S}_s(\overline{(\lambda.e)[s']}) & = & \mathcal{S}_{(x \circ s')}(e) \oplus 1 \\
\mathcal{S}(\mathbf{op}) & = & 1 & \mathcal{S}_s(\mu.e) & = & \mathcal{S}_{(x \circ s)}(e) \oplus 1 \\
\mathcal{S}(\mathbf{c}) & = & 1 & \mathcal{S}_{(v \circ s)}(\overline{n+1}) & = & \mathcal{S}_s(\overline{n}) \\
\mathcal{S}_s([e_0, \ldots, e_{p-1}]) & = & \sum_{i=0}^{p-1} \mathcal{S}_s(e_i) & \mathcal{S}_{(v \circ s)}(\overline{1}) & = & \mathcal{S}_s(v) \\
 & & & \mathcal{S}_s(x) & = & 0
\end{array}$$

$$\mathcal{S}_s(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) = \mathcal{S}_s(e_2) \oplus \mathcal{S}_s(e_2) \oplus \mathcal{S}_s(e_3) \oplus 1$$

Figure 10.7 — Fonction définissant la taille des données

léger exécutant un **get** qui est une demande de communication. Cette demande, au processeur i, se fait (ou non), pour le processeur j, si $(b_i\ j) = \mathbf{true}$. Le processeur i reçoit alors du processeur j une valeur. Ensuite, l'opérateur **build** synchronise tous les processus légers de communication et construit le vecteur de fonctions final.

La règle 10.19 est celle de la primitive **mat**. Elle dit que cette primitive se réécrit en un appel de la primitive **mget** avec ensuite la suppression du constructeur des vecteurs parallèles. Les paramètres de **mget** sont alors définis à partir de ceux de **mat**, c'est-à-dire la fonction f (donnant qui doit envoyer sa valeur ou non, c'est-à-dire la projeter) et vf qui donnera v_i si $(f\ i) = \mathbf{true}$.

Notons que dans toutes ces règles, nous ajoutons le coût 1 à toutes les composantes du vecteur de coût puisque la primitive est exécutée globalement. Les règles des opérateurs globaux sont alors les suivantes :

$$\begin{array}{rcl}
\mathbf{delpar}\ \langle f, \ldots, f\rangle/\langle c^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle & \overset{\delta}{\underset{\bowtie}{\rightharpoonup}} & f/\langle c^m\rangle/\langle c_0 \oplus 1, \ldots, c_{p-1} \oplus 1\rangle \\
\mathbf{GloId}\ v/\langle c^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle & \overset{\delta}{\underset{\bowtie}{\rightharpoonup}} & v/\langle c^m\rangle/\langle c_0 \oplus 1, \ldots, c_{p-1} \oplus 1\rangle
\end{array}$$

et

$$\mathbf{initthreads}\ f/\langle c^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \overset{\delta}{\underset{\bowtie}{\rightharpoonup}} [\![(f\ 0), \ldots, (f\ (p-1))]\!]/\langle c^m\rangle/\langle c_0 \oplus 1, \ldots, c_{p-1} \oplus 1\rangle$$

$$\mathbf{get}\ i\ \langle v_0, \ldots, v_{p-1}\rangle\ \langle b_0, \ldots, b_{p-1}\rangle/\langle c^m\rangle/\langle c\rangle \overset{\delta}{\underset{\bowtie}{\rightharpoonup}}$$
$$\langle v'_0, \ldots, v'_{p-1}\rangle/\langle c^m\rangle/\langle c\rangle \text{ tel que } \begin{cases} v'_j = v_i & \text{si } b_j = \mathbf{true} \\ v'_j = \mathbf{nc} & \text{sinon} \end{cases}$$

$$\mathbf{build}\ [\![\langle v_0^0, \ldots, v_{p-1}^0\rangle, \ldots, \langle v_0^{p-1}, \ldots, v_{p-1}^{p-1}\rangle]\!]/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \overset{\delta}{\underset{\bowtie}{\rightharpoonup}}$$
$$\langle g_0, \ldots, g_{p-1}\rangle[\bullet]/\langle {c'}_0^m, \ldots, {c'}_{p-1}^m\rangle/\langle 0\rangle$$

où $\forall i \in \{0, \ldots, p-1\}$, $g_i = (\lambda.\mathbf{if}\ (0 \leq \overline{1} < p-1)\ \mathbf{then}\ (\mathbf{access}\ (\overline{1}, [v_i^0, \ldots, v_i^{p-1}]))\ \mathbf{else}\ \mathbf{nc})$ et avec

$${c'}_i^m = (\max_{j=0 \text{ si } v_i^j \neq \mathbf{nc}}^{p-1} c_j^m \oplus c_j \oplus 1) \oplus \max((\sum_{j=0}^{p-1} \mathcal{S}(v_j^i)), (\sum_{j=0}^{p-1} \mathcal{S}(v_i^j))) \otimes g \oplus L$$

Nous faisons bien apparaître, au niveau des coûts, que les processeurs ne se synchronisent qu'avec leurs propres «partenaires entrants» (processeurs avec qui un autre processeur communique). Ainsi, nous prenons le maximal des coûts de la m-étape courante et des précédentes (des processeurs «partenaires entrants»). Ensuite, nous prenons le maximal entre les messages envoyés et les messages reçus. Enfin, nous ajoutons le temps de latence L. Cette règle termine donc la m-étape courante. Les coûts de la m-étape sont donc réduits à zéro. La figure 10.7 redonne la fonction de taille des valeurs pour MPSML (très similaire à celle de BSML)[3].

[3]Notons que nous ne donnons pas la taille pour les primitives parallèles ou les vecteur parallèles car ceux-ci ne sont pas présent dans les vecteurs parallèles et ne sont donc pas transmissibles.

Nous avons alors les règles de passage (permettant d'appliquer les règles génériques) suivantes :

$$\frac{e \xrightarrow{\varepsilon} e', c}{e \xrightarrow[i]{\varepsilon} e', c} \qquad \frac{e \xrightarrow{\delta} e', c}{e \xrightarrow[i]{\delta} e', c}$$

$$\frac{e \xrightarrow{\varepsilon} e', c}{e/\langle c^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \xrightarrow[\bowtie]{\varepsilon} e'/\langle c^m\rangle/\langle c_0 \oplus c, \ldots, c_{p-1} \oplus c\rangle}$$

$$\frac{e \xrightarrow{\delta} e', c}{e/\langle c^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \xrightarrow[\bowtie]{\delta} e'/\langle c^m\rangle/\langle c_0 \oplus c, \ldots, c_{p-1} \oplus c\rangle}$$

La différence avec la sémantique à «petits pas» de BSML vue au chapitre 3 est qu'il n'y a plus de coûts globaux. Les opérations fonctionnelles effectuées au niveau global ajoutent donc leur coût à chaque composante du vecteur de coûts locaux de la m-étape courante.

On constate aisément qu'il n'est pas toujours possible de faire des «réductions de tête». Il faut donc ajouter des règles de contexte. Ceci se fait à l'aide des contextes de la figure 10.8. Nous avons, comme pour BSML, les contextes Γ pour définir une réduction globale, les contextes Δ^i pour définir une réduction locale au processeur i et nous avons en plus, les contextes Ω^i qui permettent une communication au processus légers de communication i.

Nous pouvons alors définir notre sémantique.

Définition 36 (*Sémantique à «petits pas»*).
La sémantique est définie par les trois règles de contexte (global, local et «processus léger» de communication), qui sont :

$$\frac{e/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \xrightarrow{\bowtie} e'/\langle {c'}_0^m, \ldots, {c'}_{p-1}^m\rangle/\langle c'_0, \ldots, c'_{p-1}\rangle}{\Gamma[e]/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \rightharpoonup \Gamma[e']/\langle {c'}_0^m, \ldots, {c'}_{p-1}^m\rangle/\langle c'_0, \ldots, c'_{p-1}\rangle}$$

$$\frac{e \xrightarrow{i} e', c}{\Delta^i[e]/\langle c^m\rangle/\langle \ldots, c_i, \ldots\rangle \rightharpoonup \Delta^i[e']/\langle c^m\rangle/\langle \ldots, c_i \oplus c, \ldots\rangle}$$

$$\frac{e/\langle c^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \rightharpoonup e'/\langle c^m\rangle/\langle c'_0, \ldots, c'_{p-1}\rangle}{\Omega^i[e]/\langle c^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \rightharpoonup \Omega^i[e']/\langle c^m\rangle/\langle c'_0, \ldots, c'_{p-1}\rangle}$$

Le contexte Γ (resp. Δ^i) permet l'évaluation (application d'une règle) de l'expression en dehors d'un vecteur parallèle (resp. dans la $i$ème composante d'un vecteur, c'est-à-dire au $i$ème processeur). Enfin, le contexte Ω^i permet de choisir (de manière indéterministe), quel «processus léger» de communication se réduit.

La figure 10.2 donne l'évaluation de l'exemple donné dans la section précédente mais cette fois-ci avec notre sémantique à «petits pas» (avec toujours trois processeurs). Nous avons les résultats suivants.

Lemme 72 (*Confluence forte*)

Soit une expression e.

Si	$e/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \cdots, c_{p-1}\rangle \rightharpoonup e^1/\langle {c_0^m}^1, \ldots, {c_{p-1}^m}^1\rangle//\langle c_0^1, \cdots, c_{p-1}^1\rangle$
et	$e/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \cdots, c_{p-1}\rangle \rightharpoonup e^2/\langle {c_0^m}^2, \ldots, {c_{p-1}^m}^2\rangle/\langle c_0^2, \cdots, c_{p-1}^2\rangle$
alors il existe	e^3, $\langle {c_0^m}^3, \ldots, {c_{p-1}^m}^3\rangle$ *et* $\langle c_0^3, \cdots, c_{p-1}^3\rangle$
tel que	$e^1/\langle {c_0^m}^1, \ldots, {c_{p-1}^m}^1\rangle//\langle c_0^1, \cdots, c_{p-1}^1\rangle \rightharpoonup e^3/\langle {c_0^m}^3, \ldots, {c_{p-1}^m}^3\rangle/\langle c_0^3, \cdots, c_{p-1}^3\rangle$
et	$e^2/\langle {c_0^m}^2, \ldots, {c_{p-1}^m}^2\rangle/\langle c_0^2, \cdots, c_{p-1}^2\rangle \rightharpoonup e^3/\langle {c_0^m}^3, \ldots, {c_{p-1}^m}^3\rangle/\langle c_0^3, \cdots, c_{p-1}^3\rangle$.

Preuve. Voir en section 15 de l'annexe de ce chapitre. ■

$$
\begin{array}{rl}
 & e[\bullet]/\langle 0,0,0\rangle/\langle 0,0,0\rangle \\
= & (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))\ (\mathbf{mget}\ (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2}))\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0))))[\bullet]/\langle 0,0,0\rangle/\langle 0,0,0\rangle \\
\rightharpoonup & (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.(\overline{1}\ 0)))[\bullet]\ (\mathbf{mget}\ (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2}))\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet])/\langle 0,0,0\rangle/\langle 0,0,0\rangle \\
\rightharpoonup & (\mathbf{apply}\ (\mathbf{mkpar}\ \overline{(\lambda.\lambda.(\overline{1}\ 0))[\bullet]}\ (\mathbf{mget}\ (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2}))\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet])/\langle 0,0,0\rangle/\langle 0,0,0\rangle \\
\rightharpoonup & (\mathbf{apply}\ (\mathbf{mkpar}\ \overline{(\lambda.\lambda.(\overline{1}\ 0))[\bullet]})\ (\mathbf{mget}\ (\mathbf{mkpar}\ \overline{(\lambda.\lambda.\overline{2})})\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet])/\langle 0,0,0\rangle/\langle 1,1,1\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle \overline{((\lambda.\lambda.(\overline{1}\ 0))[\bullet]\ 0)}, \overline{((\lambda.\lambda.(\overline{1}\ 0))[\bullet]\ 1)}, \overline{((\lambda.\lambda.(\overline{1}\ 0))[\bullet]\ 2)}\rangle\ (\mathbf{mget}\ (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2}))\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet])/\langle 0,0,0\rangle/\langle 2,2,2\rangle \\
\cdots & \\
\rightharpoonup & (\mathbf{apply}\ \langle \overline{(\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[1\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[2\circ\bullet]}\rangle\ (\mathbf{mget}\ (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2}))\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet])/\langle 0,0,0\rangle/\langle 4,4,4\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle \overline{(\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[1\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[2\circ\bullet]}\rangle\ (\mathbf{mget}\ (\mathbf{mkpar}\ (\lambda.\lambda.\overline{2}))[\bullet]\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet]))/\langle 0,0,0\rangle/\langle 4,4,4\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle \overline{(\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[1\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[2\circ\bullet]}\rangle\ (\mathbf{mget}\ (\mathbf{mkpar}\ \overline{(\lambda.\lambda.\overline{2})[\bullet]})\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet]))/\langle 0,0,0\rangle/\langle 4,4,4\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle \overline{(\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[1\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[2\circ\bullet]}\rangle\ (\mathbf{mget}\ (\mathbf{mkpar}\ \overline{(\lambda.\lambda.\overline{2})[\bullet]})\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet]))/\langle 0,0,0\rangle/\langle 5,5,5\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle (\lambda.(\overline{1}\ 0))[0\circ\bullet], (\lambda.(\overline{1}\ 0))[1\circ\bullet], (\lambda.(\overline{1}\ 0))[2\circ\bullet]\rangle\ (\mathbf{mget}\ \langle \overline{((\lambda.\lambda.\overline{2})[\bullet]\ 0)}, \overline{((\lambda.\lambda.\overline{2})[\bullet]\ 1)}, \overline{((\lambda.\lambda.\overline{2})[\bullet]\ 2)}\rangle\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet]))/\langle 0,0,0\rangle/\langle 6,6,6\rangle \\
\cdots & \\
\rightharpoonup & (\mathbf{apply}\ \langle \overline{(\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[1\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[2\circ\bullet]}\rangle\ (\mathbf{mget}\ \langle \overline{(\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\overline{2})[1\circ\bullet]}, \overline{(\lambda.\overline{2})[2\circ\bullet]}\rangle\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet]))/\langle 0,0,0\rangle/\langle 8,8,8\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle \overline{(\lambda.(\overline{1}\ 0))[0\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[1\circ\bullet]}, \overline{(\lambda.(\overline{1}\ 0))[2\circ\bullet]}\rangle\ (\mathbf{mget}\ \langle \overline{(\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\overline{2})[1\circ\bullet]}, \overline{(\lambda.\overline{2})[2\circ\bullet]}\rangle\ (\mathbf{mkpar}\ (\lambda.\lambda.\ =(\overline{1},0)))[\bullet]))/\langle 0,0,0\rangle/\langle 9,9,9\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle (\lambda.(\overline{1}\ 0))[0\circ\bullet], (\lambda.(\overline{1}\ 0))[1\circ\bullet], (\lambda.(\overline{1}\ 0))[2\circ\bullet]\rangle\ (\mathbf{mget}\ \langle \overline{(\lambda.\overline{2})[0\circ\bullet]}, \overline{(\lambda.\overline{2})[1\circ\bullet]}, \overline{(\lambda.\overline{2})[2\circ\bullet]}\rangle\ \langle \overline{((\lambda.\lambda.\ =(\overline{1},0))[\bullet]\ 0)}, \overline{((\lambda.\lambda.\ =(\overline{1},0))[\bullet]\ 1)}, \overline{((\lambda.\lambda.\ =(\overline{1},0))[\bullet]\ 2)}\rangle))/\langle 0,0,0\rangle/\langle 10,10,10\rangle \\
\cdots & \\
\rightharpoonup & (\mathbf{apply}\ \overline{\langle (\lambda.(\overline{1}\ 0))[0\circ\bullet], (\lambda.(\overline{1}\ 0))[1\circ\bullet], (\lambda.(\overline{1}\ 0))[2\circ\bullet]\rangle}\ (\mathbf{mget}\ \overline{\langle (\lambda.\overline{2})[0\circ\bullet], (\lambda.\overline{2})[1\circ\bullet], (\lambda.\overline{2})[2\circ\bullet]\rangle}\ \overline{\langle (\lambda.\ =(\overline{1},0))[\bullet], (\lambda.\ =(\overline{1},0))[1\circ\bullet], (\lambda.\ =(\overline{1},0))[2\circ\bullet]\rangle}))/\langle 0,0,0\rangle/\langle 12,12,12\rangle \\
= & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ (\mathbf{mget}\ \langle f_0^2, f_1^2, f_2^2\rangle\ \langle f_0^3, f_1^3, f_2^3\rangle))/\langle 0,0,0\rangle/\langle 12,12,12\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ (\mathbf{mget}\ \langle f_0^2, f_1^2, f_2^2\rangle\ \langle f_0^3, f_1^3, f_2^3\rangle))/\langle 0,0,0\rangle/\langle 13,13,13\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ (\mathbf{build}\ \overline{(\mathbf{prep}[\bullet]}\ (tv, tb))))/\langle 0,0,0\rangle/\langle 14,14,14\rangle \\
\cdots & \\
\rightharpoonup & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ (\mathbf{build}\ \overline{(\mathbf{prep}[\bullet]}\ (tvv, tbv))))/\langle 0,0,0\rangle/\langle 31,31,31\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ (\mathbf{build}\ (\mathbf{initthreads}\ (\lambda.\mathbf{get}\ (\overline{1}, ((\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.\mathbf{access}\ (\overline{1},\overline{3})))\ (\mathbf{fst}\ \overline{2})), (\mathbf{apply}\ (\mathbf{mkpar}\ (\lambda.\lambda.\mathbf{access}\ (\overline{1},\overline{3})))\ (\mathbf{snd}\ \overline{2}))))[(tvv, tbv)\circ\bullet])))/\langle 0,0,0\rangle/\langle 31,31,31\rangle \\
\cdots & \\
\rightharpoonup & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ (\mathbf{build}\ [\![(\mathbf{get}\ (0, (\langle 0,1,2\rangle, \langle \mathbf{true},\mathbf{true},\mathbf{true}\rangle)))), (\mathbf{get}\ (0, (\langle 0,1,2\rangle, \langle \mathbf{false},\mathbf{false},\mathbf{false}\rangle))), (\mathbf{get}\ (0, (\langle 0,1,2\rangle, \langle \mathbf{false},\mathbf{false},\mathbf{false}\rangle)))]\!]))/\langle 0,0,0\rangle/\langle 45,45,45\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ (\mathbf{build}\ [\![(\mathbf{get}\ (0, (\langle 0,1,2\rangle, \langle \mathbf{true},\mathbf{true},\mathbf{true}\rangle))), \langle \mathbf{nc},\mathbf{nc},\mathbf{nc}\rangle, (\mathbf{get}\ (0, (\langle 0,1,2\rangle, \langle \mathbf{false},\mathbf{false},\mathbf{false}\rangle)))]\!]))/\langle 0,0,0\rangle/\langle 45,45,45\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ (\mathbf{build}\ [\![(\mathbf{get}\ (0, (\langle 0,1,2\rangle, \langle \mathbf{true},\mathbf{true},\mathbf{true}\rangle))), \langle \mathbf{nc},\mathbf{nc},\mathbf{nc}\rangle, \langle \mathbf{nc},\mathbf{nc},\mathbf{nc}\rangle]\!]))/\langle 0,0,0\rangle/\langle 45,45,45\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ (\mathbf{build}\ [\![\langle 0,0,0\rangle, \langle \mathbf{nc},\mathbf{nc},\mathbf{nc}\rangle, \langle \mathbf{nc},\mathbf{nc},\mathbf{nc}\rangle]\!]))/\langle 0,0,0\rangle/\langle 45,45,45\rangle \\
\rightharpoonup & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ \langle g_0, g_1, g_2\rangle)/\langle (0\oplus 45\oplus 1)\oplus(\max(1\oplus 1,0)\otimes g)\oplus L, \max_{i\in\{0,1\}}(0\oplus 45\oplus 1)\oplus(\max(0,1)\otimes g)\oplus L, \max_{i\in\{0,2\}}(0\oplus 45\oplus 1)\oplus(\max(0,1)\otimes g)\oplus L\rangle/\langle 0,0,0\rangle \\
= & (\mathbf{apply}\ \langle f_0^1, f_1^1, f_2^1\rangle\ \langle g_0, g_1, g_2\rangle)/\langle 46\oplus(2\otimes g)\oplus L, 46\oplus g\oplus L, 46\oplus g\oplus L\rangle/\langle 0,0,0\rangle \\
\cdots & \\
\rightharpoonup & \langle 0,0,0\rangle/\langle 46\oplus(2\otimes g)\oplus L, 46\oplus g\oplus L, 46\oplus g\oplus L\rangle/\langle 10,10,10\rangle
\end{array}
$$

avec

$$
\begin{array}{rcl}
f_i^1 & = & \overline{(\lambda.(\overline{1}\ 0))[i\circ\bullet]} \\
f_i^2 & = & \overline{(\lambda.\overline{2})[i\circ\bullet]} \\
f_i^3 & = & \overline{(\lambda.\ =(\overline{1},0))[i\circ\bullet]}
\end{array}
\qquad
\begin{array}{rcl}
tv & = & \langle(\mathbf{init}\ f_0^2), (\mathbf{init}\ f_1^2), (\mathbf{init}\ f_2^2)\rangle \\
tb & = & \langle(\mathbf{init}\ f_0^3), (\mathbf{init}\ f_1^3), (\mathbf{init}\ f_2^3)\rangle \\
tvv & = & \langle[0,0,0],[1,1,1],[2,2,2]\rangle \\
tbv & = & \langle[\mathbf{true},\mathbf{false},\mathbf{false}],[\mathbf{true},\mathbf{false},\mathbf{false}],[\mathbf{true},\mathbf{false},\mathbf{false}]\rangle
\end{array}
$$

$$g_i = \overline{(\lambda.\mathbf{if}\ (0\leq\overline{1}<p-1)\ \mathbf{then}\ (\mathbf{access}\ (\overline{1},[0,\mathbf{nc},\mathbf{nc}]))\ \mathbf{else}\ \mathbf{nc})[\bullet]}$$

coût MPM final=$\max\left(\begin{cases} 46\oplus(2\otimes g)\oplus L\oplus 10 \\ 46\oplus g\oplus L\oplus 10 \\ 46\oplus g\oplus L\oplus 10 \end{cases}\right)$=$56\oplus(2\otimes g)\oplus L$

Table 10.2 — Exemple de la sémantique à «petits pas» de MSPML

$$
\begin{array}{rcl}
\Gamma & ::= & [] \\
& | & \Gamma\ e \\
& | & v\ \Gamma \\
& | & (\Gamma, e) \\
& | & (v, \Gamma) \\
& | & \textbf{if } \Gamma \textbf{ then } e \textbf{ else } e \\
& | & (\textbf{mkpar } \Gamma) \\
& | & (\textbf{apply } \Gamma\ e) \\
& | & (\textbf{apply } v\ \Gamma) \\
& | & (\textbf{mget } \Gamma\ e) \\
& | & (\textbf{mget } v\ \Gamma) \\
& | & (\textbf{mat } \Gamma\ e) \\
& | & (\textbf{mat } v\ \Gamma)
\end{array}
\qquad
\begin{array}{rcl}
\Delta_i & ::= & \Delta_i\ e \\
& | & v\ \Delta_i \\
& | & (\Delta_i, e) \\
& | & (v, \Delta_i) \\
& | & \textbf{if } \Delta_i \textbf{ then } e \textbf{ else } e \\
& | & (\textbf{mkpar } \Delta_i) \\
& | & (\textbf{apply } \Delta_i\ e) \\
& | & (\textbf{apply } v\ \Delta_i) \\
& | & (\textbf{mget } v\ \Delta_i) \\
& | & (\textbf{mget } v\ \Delta_i) \\
& | & (\textbf{mat } v\ \Delta_i) \\
& | & (\textbf{mat } v\ \Delta_i) \\
& | & \langle e, \ldots, \overbrace{\Gamma^l[e]}^{i}, \ldots, e \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\Gamma^l & ::= & [] \\
& | & \Gamma^l\ e \\
& | & v\ \Gamma^l \\
& | & (\Gamma^l, e) \\
& | & (v, \Gamma^l) \\
& | & \textbf{if } \Gamma^l \textbf{ then } e \textbf{ else } e \\
& | & [\Gamma^l, e_1, \ldots, e_n] \\
& | & [v_0, \Gamma^l, \ldots, e_n] \\
& | & \ldots \\
& | & [v_0, v_1, \ldots, \Gamma^l]
\end{array}
\qquad
\begin{array}{rcl}
\Omega_i & ::= & \Omega_i\ e \\
& | & v\ \Omega_i \\
& | & (\Omega_i, e) \\
& | & (v, \Omega_i) \\
& | & \textbf{if } \Omega_i \textbf{ then } e \textbf{ else } e \\
& | & (\textbf{mkpar } \Omega_i) \\
& | & (\textbf{apply } \Omega_i\ e) \\
& | & (\textbf{apply } v\ \Omega_i) \\
& | & (\textbf{mget } \Omega_i\ e) \\
& | & (\textbf{mget } v\ \Omega_i) \\
& | & (\textbf{mat } \Omega_i\ e) \\
& | & (\textbf{mat } v\ \Omega_i) \\
& | & [\![e, \ldots, \overbrace{[]}^{i}, \ldots, e]\!]
\end{array}
$$

Figure 10.8 — Contextes d'évaluation

Théorème 15 (*Confluence*)

Soit une expression «programmeur» e^p *tel que* $e = \mathcal{T}_\bullet(e^p)$.

Si $e[\bullet]/\langle 0, \ldots, 0\rangle/\langle 0, \cdots, 0\rangle \stackrel{*}{\rightharpoonup} v/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \cdots, c_{p-1}\rangle$

et $e[\bullet]/\langle 0, \ldots, 0\rangle/\langle 0, \cdots, 0\rangle \stackrel{*}{\rightharpoonup} v'/\langle c'^m_0, \ldots, c'^m_{p-1}\rangle/\langle c'_0, \cdots, c'_{p-1}\rangle$

alors $v = v'$ *et* $\forall i \in \{0, \ldots, p-1\}\ c_i = c'_i$ *et* $c_i^m = c'^m_i$.

Preuve. La relation $\rightharpoonup$ est fortement confluente, donc, d'après le lemme 1, elle est confluente. ■

Théorème 16 (*Équivalence*)

Soit une expression programmeur e^p *telle que* $e = \mathcal{T}_\bullet(e^p)$. *Alors :*

1. *si* $e[\bullet]/\langle 0, \ldots, 0\rangle/\langle 0, \ldots, 0\rangle \stackrel{*}{\rightharpoonup} v/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle$ *alors* $\bullet, e \rhd_g v$
2. *si* $\bullet, e \rhd_g v$ *alors* $e[\bullet]/\langle 0, \ldots, 0\rangle/\langle 0, \ldots, 0\rangle \stackrel{*}{\rightharpoonup} v/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle$

Preuve. Preuve similaire à celle de 3.A.3 du chapitre 3. ■

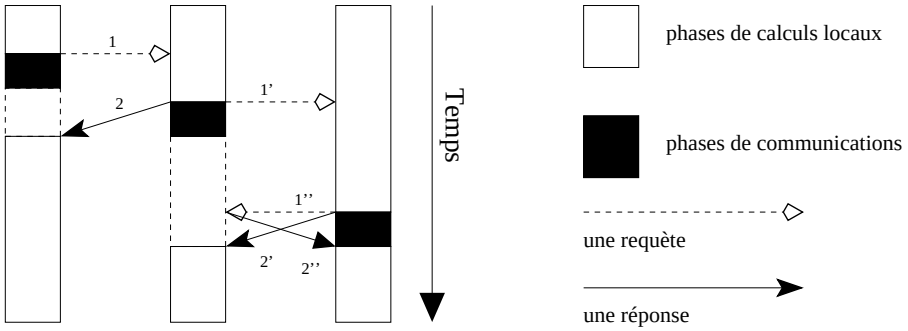


Figure 10.9 — Des m-étapes d'une exécution MSPML

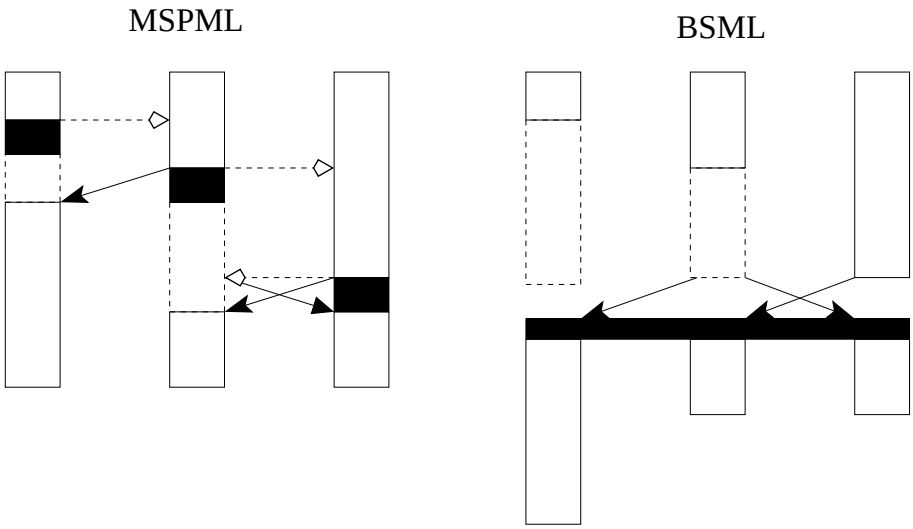


Figure 10.10 — Des m-étapes et une super-étape

10.4.4 Sémantique distribuée

Dans la sémantique précédente, toutes les opérations de communication semblent être synchrones. De plus l'asynchronisme des m-étapes n'apparaît pas : celles-ci sont encore synchrones dans la sémantique à «petits pas». C'est pourquoi nous proposons maintenant une évaluation distribuée afin de remédier à ces défauts. Cette sémantique est donc plus éloignée de celle du programmeur mais plus proche de l'exécution réelle sur une machine parallèle. La sémantique est basée sur l'asynchronisme illustré à la figure 10.9. La différence entre BSML et MSPML pour un même programme est donnée à la figure 10.10.

Les termes de la sémantique distribuée, appelés termes projetés, sont ceux présentés précédemment mais avec des vecteurs parallèles de taille 1, c'est-à-dire :

$$e ::= \cdots \mid \langle e \rangle$$

Les valeurs sont modifiées de la même manières.

Pour passer de la sémantique à «petits pas» dont les termes sont des termes de vecteurs parallèles, à une évaluation distribuée (plus proche de l'exécution MPM), dont les termes sont des vecteurs de termes, nous allons définir une *projection* de nos termes. Cette projection, pour le processeur i, est une fonction de nos expressions usuelles vers les expressions distribuées. Elle est présentée à la figure 10.11 où cette projection est définie comme suit sur les substitutions :

$$\begin{array}{rcl} \mathcal{P}_i(\bullet) & = & \bullet \\ \mathcal{P}_i(v \circ s) & = & \mathcal{P}_i(v) \circ \mathcal{P}_i(s) \end{array}$$

En fait les expressions ne sont pas modifiées, sauf les vecteurs parallèles, dont on ne prend que la $i$ème composante.

La transformation d'une expression e, en vue de son évaluation distribuée, procède en deux étapes :

$$
\begin{array}{rcl|rcl}
\mathcal{P}_i((e)[s]) & = & \mathcal{P}_i(e)[\mathcal{P}_i(s)] & \mathcal{P}_i(\mu.e) & = & (\mu.\mathcal{P}_i(e)) \\
\mathcal{P}_i(\lambda.e) & = & \lambda.\mathcal{P}_i(e) & \mathcal{P}_i(\mathbf{mkpar}\ e) & = & (\mathbf{mkpar}\ \mathcal{P}_i(e)) \\
\mathcal{P}_i(\overline{(\lambda.e)[s]}) & = & \overline{(\lambda.\mathcal{P}_i(e))[\mathcal{P}_i(s)]} & \mathcal{P}_i(\mathbf{mget}\ e_1\ e_2) & = & (\mathbf{mget}\ \mathcal{P}_i(e_1)\ \mathcal{P}_i(e_2)) \\
\mathcal{P}_i(e_1\ e_2) & = & (\mathcal{P}_i(e_1)\ \mathcal{P}_i(e_2)) & \mathcal{P}_i(\mathbf{mat}\ e_1\ e_2) & = & (\mathbf{mat}\ \mathcal{P}_i(e_1)\ \mathcal{P}_i(e_2)) \\
\mathcal{P}_i(e_1, e_2) & = & (\mathcal{P}_i(e_1), \mathcal{P}_i(e_2)) & \mathcal{P}_i(\mathbf{init}\ e) & = & (\mathbf{init}\ \mathcal{P}_i(e)) \\
\mathcal{P}_i(\mathbf{apply}\ e_1\ e_2) & = & (\mathbf{apply}\ \mathcal{P}_i(e_1)\ \mathcal{P}_i(e_2)) & \mathcal{P}_i(\mathbf{c}) & = & \mathbf{c} \\
\mathcal{P}_i([e_0, \ldots, e_{p-1}]) & = & [\mathcal{P}_i(e), \ldots, \mathcal{P}_i(e)] & \mathcal{P}_i(\mathbf{op}) & = & \mathbf{op} \\
\mathcal{P}_i(\langle \ldots, e_i, \ldots \rangle) & = & \langle \mathcal{P}_i(e_i) \rangle & \mathcal{P}_i(\overline{\mathbf{n}}) & = & \overline{\mathbf{n}}
\end{array}
$$

$$\mathcal{P}_i(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) = \mathbf{if}\ \mathcal{P}_i(e_1)\ \mathbf{then}\ \mathcal{P}_i(e_2)\ \mathbf{else}\ \mathcal{P}_i(e_3)$$

Figure 10.11 — Projection d'une expression sur un processeur i

1. L'expression est projetée sur chaque processeur donnant p termes projetés avec chacun d'eux un environnement de communication et un numéro de m-étape ;
2. Ces termes sont rassemblés en un terme distribué.

Nous avons donc pour une expression e le terme distribué suivant :

$$\langle\!\langle \mathcal{E}_{\mathcal{C}0}, n_0, \mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}p-1}, n_{p-1}, \mathcal{P}_{p-1}(e) \rangle\!\rangle$$

Nous étendons aussi l'ensemble des opérateurs parallèles avec une opération **request** (requête de communication) qui sera utilisée pour qu'un «processus léger» de communication «attende» la valeur qu'il doit recevoir d'un autre processeur.

L'opérateur **request** sera employé pour permettre l'évaluation d'un **mget** sans avoir de barrière de synchronisation globale. À chaque étape de communication (un appel à un **mget** ou à un **mat**), lors de l'exécution de l'opérateur **get**, chaque processeur i stocke le numéro de la m-étape courante n_i et la valeur qu'il détient pour le processeur j. Ces triplets sont stockés dans un *environnement de communications* noté $\mathcal{E}_{\mathcal{C}}$. Ces environnements sont vus comme des listes d'associations et évoluent de façon asynchrone durant l'exécution. Le processeur i demande ensuite la valeur contenue dans l'environnement de communications du processeur j de la m-étape n_i. Cette demande est formellement écrite : $\mathbf{request}\ j\ n_i$. La réduction locale peut créer des expressions **request** mais elle ne peut les faire disparaître. Ceci ne peut être fait qu'au niveau distribué. Notons que les expressions **request** ne sont pas des valeurs mais bien une attente d'une valeur.

Les environnements de communications sont initialement vides (noté $\bullet$) et l'évaluation commence bien entendu avec les numéros de m-étapes à 0. La sémantique distribuée a donc la forme suivante :

$$\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/e_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/e_{p-1} \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}}{}^1_0/n^1_0/e^1_0, \ldots, \mathcal{E}_{\mathcal{C}}{}^1_{p-1}/n^1_{p-1}/e^1_{p-1} \rangle\!\rangle$$

Nous notons $\overset{*}{\rightsquigarrow}$ pour la fermeture transitive et réflexive de $\rightsquigarrow$.

Définition 37 (*Relations de la sémantique distribuée*).
La sémantique distribuée $\rightsquigarrow$ est définie en deux étapes :

1. *L'évaluation locale ($\overset{i}{\rightsquigarrow}$ et $\overset{\bowtie_i}{\rightsquigarrow}$) d'un terme projeté (sous-terme d'un terme distribué effectué par un processeur i) ;*
2. *La réduction des expressions distribuées qui permet l'évaluation des requêtes de communication.*

avec :

$$\overset{i}{\rightsquigarrow} = \underset{i}{\overset{\varepsilon}{\rightsquigarrow}} \cup \underset{i}{\overset{\delta}{\rightsquigarrow}} \qquad \overset{\bowtie_i}{\rightsquigarrow} = \underset{\bowtie_i}{\overset{\varepsilon}{\rightsquigarrow}} \cup \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}}$$

Comme en BSML, l'évaluation distribuée d'un terme distribué ne peut se faire que si les termes projetés proviennent tous d'une même expression. Nous notons $\overset{\bowtie_i}{\rightsquigarrow}$ l'évaluation au processeur i d'une règle globale et $\overset{i}{\rightsquigarrow}$ l'évaluation locale (dans un vecteur projeté) au processeur i.

Nous définissons les règles de substitution, propagation de la substitution (dans les sous-expressions), δ-règles génériques (règles classiques de la programmation fonctionnelle) de la même manière que dans

$$
\begin{array}{rcl}
(\overline{(\lambda.e)[s]}\ v) & \overset{\varepsilon}{\rightsquigarrow} & e[v \circ s] \\
\overline{n+1}[v \circ s] & \overset{\varepsilon}{\rightsquigarrow} & \overline{n}[s] \\
\overline{1}[v \circ s] & \overset{\varepsilon}{\rightsquigarrow} & v[\bullet] \\
(\mu.e)[s] & \overset{\varepsilon}{\rightsquigarrow} & \overline{e[\mu.e \circ s]} \\
(\lambda.e)[s] & \overset{\varepsilon}{\rightsquigarrow} & \overline{(\lambda.e)[s]} \\
v[s] & \overset{\varepsilon}{\rightsquigarrow} & v \quad \text{si } v \neq \langle \cdots \rangle \text{ et } v \neq (v_0, v_1) \\
(e_1\ e_2)[s] & \overset{\varepsilon}{\rightsquigarrow} & (e_1[s]\ e_2[s]) \\
(e_1, e_2)[s] & \overset{\varepsilon}{\rightsquigarrow} & (e_1[s], e_2[s]) \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)[s] & \overset{\varepsilon}{\rightsquigarrow} & \textbf{if } e_1[s] \textbf{ then } e_2[s] \textbf{ else } e_3[s] \\
(\textbf{mget } e_1\ e_2)[s] & \overset{\varepsilon}{\rightsquigarrow} & (\textbf{mget } e_1[s]\ e_2[s]) \\
(\textbf{mat } e_1\ e_2)[s] & \overset{\varepsilon}{\rightsquigarrow} & (\textbf{mat } e_1[s]\ e_2[s]) \\
(\textbf{apply } e_1\ e_2)[s] & \overset{\varepsilon}{\rightsquigarrow} & (\textbf{apply } e_1[s]\ e_2[s]) \\
(\textbf{mkpar } e)[s] & \overset{\varepsilon}{\rightsquigarrow} & (\textbf{mkpar } e[s]) \\
\langle e \rangle[s] & \overset{\varepsilon}{\rightsquigarrow} & \langle e[s] \rangle \\
[e_0, \ldots, e_{p-1}][s] & \overset{\varepsilon}{\rightsquigarrow} & [e_0[s], \ldots, e_{p-1}[s]]
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{access } ([v_0, \ldots, v_i, \ldots, v_{p-1}], i) & \overset{\delta}{\rightsquigarrow} & v_i \\
\textbf{init } (n, f) & \overset{\delta}{\rightsquigarrow} & [(f\ 0), \ldots, (f\ (n-1))] \\
\textbf{fst } (v_1, v_2) & \overset{\delta}{\rightsquigarrow} & v_1 \\
\textbf{snd } (v_1, v_2) & \overset{\delta}{\rightsquigarrow} & v_2 \\
+ (n_1, n_2) & \overset{\delta}{\rightsquigarrow} & n_1 + n_2 \\
\textbf{isnc nc} & \overset{\delta}{\rightsquigarrow} & \textbf{true} \\
\textbf{isnc } v & \overset{\delta}{\rightsquigarrow} & \textbf{false si } v \neq \textbf{nc} \\
\textbf{if true then } e_2 \textbf{ else } e_3 & \overset{\delta}{\rightsquigarrow} & e_2 \\
\textbf{if false then } e_2 \textbf{ else } e_3 & \overset{\delta}{\rightsquigarrow} & e_3
\end{array}
$$

Figure 10.12 — Réductions fonctionnelles et opérations génériques

la sémantique à petits pas (figure 10.12). Les règles de passage (pour appliquer les réductions génériques) sont les suivantes :

$$
\frac{e \overset{\varepsilon}{\rightsquigarrow} e'}{e \underset{i}{\overset{\varepsilon}{\rightsquigarrow}} e'} \qquad \frac{e \overset{\delta}{\rightsquigarrow} e'}{e \underset{i}{\overset{\delta}{\rightsquigarrow}} e'} \qquad \frac{e \overset{\varepsilon}{\rightsquigarrow} e'}{\mathcal{E}_{\mathcal{C}i}/n_i/e \underset{\bowtie_i}{\overset{\varepsilon}{\rightsquigarrow}} \mathcal{E}_{\mathcal{C}i}/n_i/e'} \qquad \frac{e \overset{\delta}{\rightsquigarrow} e'}{\mathcal{E}_{\mathcal{C}i}/n_i/e \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} \mathcal{E}_{\mathcal{C}i}/n_i/e'}
$$

Les δ-règles des opérateurs locaux et globaux sont :

$$
\textbf{LocId } v \quad \underset{i}{\overset{\delta}{\rightsquigarrow}} \quad v
$$

$$
\begin{array}{rcl}
\mathcal{E}_{\mathcal{C}i}/n_i/\textbf{delpar } \langle f \rangle & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & \mathcal{E}_{\mathcal{C}i}/n_i/f \\
\mathcal{E}_{\mathcal{C}i}/n_i/\textbf{GloId } v & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & \mathcal{E}_{\mathcal{C}i}/n_i/v \text{ si } \mathcal{V}_\bullet(v) = \textbf{true}
\end{array}
$$

et les règles des primitives parallèles sont les suivantes :

$$
\begin{array}{rcl}
\mathcal{E}_{\mathcal{C}i}/n_i/\textbf{mkpar } f & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & \mathcal{E}_{\mathcal{C}i}/n_i/\langle (f\ i) \rangle[\bullet] \quad \text{si } \mathcal{V}_\bullet(f) = \textbf{true} \\
\mathcal{E}_{\mathcal{C}i}/n_i/\textbf{apply } \langle f \rangle\ \langle v \rangle & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & \mathcal{E}_{\mathcal{C}i}/n_i/\langle (f\ v) \rangle[\bullet] \\
\mathcal{E}_{\mathcal{C}i}/n_i/\textbf{mget } \langle f \rangle\ \langle b \rangle & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & \mathcal{E}_{\mathcal{C}i}/n_i/(\textbf{build } (\textbf{prep } (tv, tb)))[\bullet] \\
\mathcal{E}_{\mathcal{C}i}/n_i/\textbf{mat } \langle v \rangle\ f & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & \mathcal{E}_{\mathcal{C}i}/n_i/(\textbf{delpar } (\textbf{mget } \mathit{vf}\ \langle f \rangle))[\bullet]
\end{array}
$$

où : $tv = \langle (\textbf{init } f) \rangle$, $tb = \langle (\textbf{init } b) \rangle$ et $\mathit{vf} = \langle (\lambda.\textbf{if } (f\ \overline{1}) \textbf{ then } v \textbf{ else nc}) \rangle$.
Elles sont similaires (utilisation des opérations parallèles de plus «bas niveau») à celles de la sémantique à «petits pas». Les environnements de communications restent inchangés (ainsi que les numéros des m-étapes). Les δ-règles des opérateurs de communications sont :

$$
\begin{array}{rcl}
\mathcal{E}_{\mathcal{C}i}/n_i/\textbf{initthreads } f/\langle c^m \rangle/\langle c_0, \ldots, c_{p-1} \rangle & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & \mathcal{E}_{\mathcal{C}i}/n_i/[\![(f\ 0), \ldots, (f\ (p-1))]\!] \\
\mathcal{E}_{\mathcal{C}i}/n_i/\textbf{get } j\ \langle v \rangle\ \langle b \rangle & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & ((n_i, j), v) \circ \mathcal{E}_{\mathcal{C}i}/n_i/ \left\{ \begin{array}{ll} \textbf{request } j\ n_i & \text{si } b = \textbf{true} \\ \textbf{nc} & \text{sinon} \end{array} \right. \\
\mathcal{E}_{\mathcal{C}i}/n_i/\textbf{build}[\![\langle v_0 \rangle, \ldots, \langle v_{p-1} \rangle]\!] & \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} & \mathcal{E}_{\mathcal{C}i}/n_i + 1/\langle g \rangle[\bullet]
\end{array}
$$

avec $g = (\lambda.\textbf{if } (0 \leq \overline{1} < p-1) \textbf{ then } (\textbf{access } (\overline{1}, [v_0, \ldots, v_{p-1}])) \textbf{ else nc})$.

Pour l'opération $\mathbf{initthreads}$ la règle est celle attendue : création de p processus légers de communications. La règle pour le $\mathbf{build}$ est aussi naturelle. Á partir des valeurs reçues, l'opérateur construit le vecteur final et on passe à la m-étape suivante.

L'opération $\mathbf{get}$ est plus complexe. En premier lieu, elle sauvegarde dans l'environnement de communications le valeur v. Celle-ci pourra ensuite être lue par un autre processeur *via* un $\mathbf{request}$. En second lieu, l'opérateur $\mathbf{get}$ se transforme en un $\mathbf{request}$ afin de récupérer la valeur d'un autre processeur. Ceci est fait si $b = \mathbf{true}$, c'est-à-dire si l'on demande bien la valeur à cet autre processeur. Nous pouvons maintenant donner la sémantique distribuée.

Définition 38 (*Sémantique distribuée*).
Nous avons alors les quatre règles de contexte suivantes pour définir l'évaluation distribuée $\rightsquigarrow$:

$$\frac{\mathcal{E}_{\mathcal{C}i}/n_i/e_i \overset{\bowtie_i}{\rightsquigarrow} \mathcal{E}_{\mathcal{C}i}'/n_i'/e_i'}{\langle\!\langle \ldots, \mathcal{E}_{\mathcal{C}i}/n_i/\Gamma[e_i], \ldots \rangle\!\rangle \rightsquigarrow \langle\!\langle \ldots, \mathcal{E}_{\mathcal{C}i}'/n_i'/\Gamma[e_i'], \ldots \rangle\!\rangle}$$

$$\frac{e_i \overset{i}{\rightsquigarrow} e_i'}{\langle\!\langle \ldots, \mathcal{E}_{\mathcal{C}i}/n_i/\Delta[e_i], \ldots \rangle\!\rangle \rightsquigarrow \langle\!\langle \ldots, \mathcal{E}_{\mathcal{C}i}/n_i/\Delta[e_i'], \ldots \rangle\!\rangle}$$

$$\frac{\mathcal{E}_{\mathcal{C}i}/n_i/e_i \rightsquigarrow \mathcal{E}_{\mathcal{C}i}'/n_i'/e_i'}{\langle\!\langle \ldots, \mathcal{E}_{\mathcal{C}i}/n_i/\Omega^j[e_i], \ldots \rangle\!\rangle \rightsquigarrow \langle\!\langle \ldots, \mathcal{E}_{\mathcal{C}i}'/n_i'/\Omega^j[e_i'], \ldots \rangle\!\rangle}$$

$$\frac{((n_i, j), v) \in \mathcal{E}_{\mathcal{C}j}}{\langle\!\langle \ldots, \mathcal{E}_{\mathcal{C}i}/n_i/\Omega^j[\mathbf{request}\ j\ n_i], \ldots \rangle\!\rangle \rightsquigarrow \langle\!\langle \ldots, \mathcal{E}_{\mathcal{C}i}/n_i/\Omega^j[\langle v \rangle], \ldots \rangle\!\rangle}$$

La première règle permet l'évaluation d'un terme projeté global (en dehors d'un vecteur parallèle). L'environnement de communications (resp. le numéro de la m-étape) peut être modifié par l'opérateur parallèle $\mathbf{get}$ (resp. $\mathbf{build}$).

La deuxième règle précise la réduction locale d'un processeur. La troisième règle permet la réduction d'un «processus léger» de communications. Là encore, l'environnement de communications peut changer.

La quatrième règle définit une communication entre deux processeurs (une requête). Cette règle signifie que, si un processeur i demande la valeur détenue par un processeur j à la m-étape n_i et que l'environnement de communications $\mathcal{E}_{\mathcal{C}j}$ du processeur j contient la valeur v à la m-étape n_i pour le processeur i, alors cette valeur est envoyée au processeur i. Sinon la règle ne peut pas être appliquée : le processeur j n'a pas atteint la m-étape numéro n_i et le processeur i doit attendre.
Nous avons alors les résultats suivants.

Lemme 73 (*Confluence forte*)

Soit $\langle\!\langle e_0, \ldots, e_{p-1} \rangle\!\rangle$ un terme distribuée.
Si $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/e_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/e_{p-1} \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}^1/n_0^1/e_0^1, \ldots, \mathcal{E}_{\mathcal{C}p-1}^1/n_{p-1}^1/e_{p-1}^1 \rangle\!\rangle$
et $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/e_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/e_{p-1} \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}^2/n_0^2/e_0^2, \ldots, \mathcal{E}_{\mathcal{C}p-1}^2/n_{p-1}^2/e_{p-1}^2 \rangle\!\rangle$
alors il existe $\langle\!\langle \mathcal{E}_{\mathcal{C}0}^3/n_0^3/e_0^3, \ldots, \mathcal{E}_{\mathcal{C}p-1}^3/n_{p-1}^3/e_{p-1}^3 \rangle\!\rangle$
tel que $\langle\!\langle \mathcal{E}_{\mathcal{C}0}^1/n_0^1/e_0^1, \ldots, \mathcal{E}_{\mathcal{C}p-1}^1/n_{p-1}^1/e_{p-1}^1 \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}^3/n_0^3/e_0^3, \ldots, \mathcal{E}_{\mathcal{C}p-1}^3/n_{p-1}^3/e_{p-1}^3 \rangle\!\rangle$
et $\langle\!\langle \mathcal{E}_{\mathcal{C}0}^2/n_0^2/e_0^2, \ldots, \mathcal{E}_{\mathcal{C}p-1}^2/n_{p-1}^2/e_{p-1}^2 \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}^3/n_0^3/e_0^3, \ldots, \mathcal{E}_{\mathcal{C}p-1}^3/n_{p-1}^3/e_{p-1}^3 \rangle\!\rangle$.

Preuve. Voir en section 17 de l'annexe de ce chapitre. ■

Théorème 17 (*Confluence*)

Soit une expression «programmeur» e^p tel que $e = \mathcal{T}_\bullet(e^p)$ et $\forall i \in \{0, \ldots, p-1\}$ $e_i = \mathcal{P}_i(e)$. Alors :
Si $\langle\!\langle \bullet/0/e_0[\bullet], \ldots, \bullet/0/e_{p-1}[\bullet] \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/v_{p-1} \rangle\!\rangle$
et $\langle\!\langle \bullet/0/e_0[\bullet], \ldots, \bullet/0/e_{p-1}[\bullet] \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/v_0', \ldots, \mathcal{E}_{\mathcal{C}p-1}'/n_{p-1}'/v_{p-1}' \rangle\!\rangle$
alors $\forall i \in \{0, \ldots, p-1\}$ $v_i = v_i'$, $n_i = n_i'$ *et* $\mathcal{E}_{\mathcal{C}i} = \mathcal{E}_{\mathcal{C}i}'$.

Preuve. La relation $\rightsquigarrow$ est fortement confluente, donc, d'après le lemme 1, elle est confluente. ■

p: unit→int
pid: unit→int
store: α →unit
request: int→int→α
reset_mstep: unit→unit

Figure 10.13 — Le module Tcpip

Théorème 18 (*Équivalence*)

Soit e^p une expression «programmeur» telle que $e = \mathcal{T}_\bullet(e^p)$. Alors :

1. *Si* $e[\bullet]/\langle 0,\ldots,0\rangle/\langle 0,\ldots,0\rangle \stackrel{*}{\rightharpoonup} v/\langle c_0^m,\ldots,c_{p-1}^m\rangle/\langle c_0,\ldots,c_{p-1}\rangle$
   alors $\langle\!\langle \bullet/0/\mathcal{P}_0(e[\bullet]),\ldots,\bullet/0/\mathcal{P}_{p-1}(e[\bullet])\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/v_0,\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/v_{p-1}\rangle\!\rangle$
   tel que $\forall i \in \{0,\ldots,p-1\}\ \mathcal{P}_i(v) = v_i$ *;*
2. *Si* $\langle\!\langle \bullet/0/\mathcal{P}_0(e[\bullet]),\ldots,\bullet/0/\mathcal{P}_{p-1}(e[\bullet])\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/v_0,\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/v_{p-1}\rangle\!\rangle$
   alors $e[\bullet]/\langle 0,\ldots,0\rangle/\langle 0,\ldots,0\rangle \stackrel{*}{\rightharpoonup} v/\langle c_0^m,\ldots,c_{p-1}^m\rangle/\langle c_0,\ldots,c_{p-1}\rangle$
   tel que $\forall i \in \{0,\ldots,p-1\}\ \mathcal{P}_i(v) = v_i$.

Preuve . Par application de la proposition 7 (voir en section 10.A.3 de l'annexe de ce chapitre). ■

10.5 Implantation de MSPML

MSPML est implanté sous forme d'une bibliothèque[4] pour OCaml. Celle-ci ne nécessite pas forcement une bibliothèques de communications supplémentaires (et sous-jaccente) comme MPI car cette première implantation repose sur le module Unix d'OCaml et des processus légers. Il n'y a pas encore d'implantation modulaire comme la BSMLlib mais l'organisation des modules s'en rapproche. Ainsi le module contenant les primitives, le module Mpsml, repose sur un module de plus «bas niveau» (pour l'instant unique), le module Tcpip (écrit dans un style SPMD) et qui propose des fonctions de plus «bas niveau» dont certaines sont proches des fonctions MPI. Ce dernier modeule est donc interne à l'implantation et par conséquent, n'est pas disponible à l'utilisateur de la bibliothèque MSPMLlib. Cette utilisateur ne peut donc pas écrire de programmes SPMD (tout comme en BSML).

10.5.1 Module Tcpip

Les fonctions essentielles du module Tcpip sont données à la figure 10.13.

p et pid donnent respectivement le nombre total de processeurs et le numéro de processeur. Au début de l'exécution d'un programme MSPML, il y a deux processus légers par processeur : l'un correspond à la réduction locale de la sémantique distribuée (section 10.4) et l'autre est créé à l'initialisation et prend en charge les communications. Ce second processus léger répond aux requêtes venant des autres processeurs.

MSPML suit le modèle d'exécution MPM. L'exécution d'un programme utilisant la MSPMLlib est donc une succession de phases de calculs locaux et de phases de communications. Chaque couple calcul-communication constitue une m-étape. Durant l'évaluation d'un programme MSPML, chaque processeur i a une variable $\mathbf{mstep}_i$ contenant le numéro de la m-étape courante.

À chaque m-étape, chaque processeur stocke une valeur dans ce qui est appelé son environnement de communications. Le stockage est effectué en utilisant la fonction store. Un autre processeur peut obtenir cette valeur en utilisant la fonction request. L'argument de la fonction request est le numéro de processeur destinataire de la requête. Implicitement la requête est faite pour le numéro de m-étape courant du processeur requérant.

[4] La version actuelle de MSPML est disponible à `http://mspml.free.fr`.

10.5.2 Module Mspml

L'implantation du noyau de la MSPMLlib suit le style de programmation SPMD. Ainsi, pour un processeur, le type abstrait des vecteurs parallèles est défini par : **type** α **par** = α. L'utilisation d'un tel type abstrait permet d'éviter le problème que l'on rencontre dans les langages data-parallèles SPMD, lesquels ne permettent pas de distinguer les variables dont les valeurs sont dépendantes du numéro de processeur de celles qui ne le sont pas.

L'implantation des primitives qui ne nécessitent pas de communication est alors aisée :

```
let mkpar f = f (Tcpip.pid())
let apply f v = f v
```

L'implantation de **mget** nécessite plusieurs requêtes qui peuvent être faites en même temps. Chaque requête est effectuée par un processus léger différent. À un processeur i, la fonction, premier argument de **mget**, est stockée dans l'environnement de communications avec la valeur de $\mathbf{mstep}_i$. $\mathbf{mstep}_i$ est ensuite incrémentée.

Après, la fonction des entiers vers les booléens (second argument de **mget**) est appliquée aux entiers entre 0 et $p-1$. Lorsque le résultat est **true** un nouveau processeur léger est créé pour traiter la requête correspondante.

Une requête du processeur i est une demande au processeur j pour avoir la valeur que celui-ci a dans son environnement de communications à la m-étape de numéro $\mathbf{mstep}_i$. Lorsque le processeur j reçoit cette requête (un processus léger est créé pour traiter cette requête) il y a deux cas :

1. $\mathbf{mstep}_j \geq \mathbf{mstep}_i$: ce qui veut dire que le processeur j a déjà atteint la même m-étape que le processeur i. Le processeur j a donc déjà stocké une valeur dans son environnement de communications pour la m-étape de numéro $\mathbf{mstep}_i$. Il lui suffit de l'envoyer au processeur i ;

2. $\mathbf{mstep}_j < \mathbf{mstep}_i$: rien ne peut être fait tant que le processeur j n'a pas atteint la m-étape $\mathbf{mstep}_i$.

Si $i = j$ l'étape 2 n'est bien sûr pas effectuée.

Quand un processeur j reçoit une requête d'un processeur i pour une valeur stockée par un **mget** et que celle-ci peut être traitée, la valeur stockée est «désérialisée». Le résultat est alors renvoyé au processeur j. Lorsque toutes les requêtes ont été traitées, la fonction résultat est créée.

La taille des environnements de communications est bien sûr bornée. Cette taille est un paramètre donné par l'utilisateur au moment de l'exécution. L'implantation actuelle impose une barrière de synchronisation pour vider les environnements de communications lorsque ceux-ci sont pleins. Bien sûr, à moins que des données de très grande taille soient échangées à chaque m-étape, cette valeur maximale peut être assez grande pour que les barrières soient rares. [194] propose une solution pour éviter ces barrières de synchronisation au prix d'un léger surcoût à chaque échange. Cette solution sera disponible dans la prochaine implantation de la bibliothèque MSPMLlib.

10.6 Exemple et expériences

10.6.1 Réduction et préfixe parallèles

Il existe différentes versions de la réduction parallèle. Il est possible de réduire un vecteur de type α **par** mais aussi un vecteur parallèle de type α collection **par** où par exemple collection peut être list ou array. Dans la figure 10.14, fold_direct est une réduction parallèle d'un vecteur de type α **par**. Elle est dite directe car les valeurs sont échangées entre les processus en une seule m-étape. Le coût de la phase de communication est $s \times (p-1) \times g + L$ où s est la taille de la plus grande valeur dans le vecteur parallèle. Suivant les paramètres s, p, L et g, une version utilisant $\log_2(p)$ m-étapes peut être plus efficace. Chaque phase de communications de la m-étape sera alors $s \times g + L$. Nous n'utiliserons par la suite que la version directe du calcul de la réduction parallèle, afin de simplifier la présentation de l'implantation du patron.

La réduction parallèle d'un vecteur de listes peut être facilement définie avec fold_direct. Grâce au polymorphisme de OCaml (polymorphisme «à la ML»), nous pouvons implanter une réduction parallèle générique d'un vecteur parallèle de collections et ceci en utilisant n'importe quelle version de la réduction (directe ou non) :

```
let generic_wide_fold sfold fold op neutral vv =
  let local_fold = parfun (sfold op neutral) vv in
```

```
(* val fold_direct: (α →β →α )→α →β par→α par *)
let fold_direct op neutral vv =
  parfun (List.fold_left op neutral) (totex_list vv)

(* val wide_fold_list_direct: (α →α →α )→α →α  list par→α  list par *)
let wide_fold_list_direct op neutral vv =
  let local_fold = parfun (List.fold_left op neutral) vv in
  fold_direct op neutral local_fold

(* val prescan_direct: (α →β →α )→α →β par→α par *)
let prescan_direct op neutral v =
  let com = mget (parfun (fun v i→v) v) (mkpar(fun i j→j<i))
  and sfold = List.fold_left op neutral
  and lists = mkpar(fun i→from_to 0 (i−1)) in
  parfun sfold (parfun2 List.map (parfun (compose noSome) com) lists)

(* val generic_scan :
  ((α →β )→γ → δ → ϵ )→((α →β )→γ → ζ par→α par)→(β → η → ϑ )→
  ( ϵ → ζ  option *  η )→( ϵ → ζ  option *  η )→(α →β )→γ → δ par→ ϑ par *)
let generic_scan sscan prescan map last cutlast op neutral vv =
  let local_scan = parfun (sscan op neutral) vv in
  let tmp = applyat (p()−1) last cutlast local_scan in
  let last_elements = parfun (compose noSome fst) tmp
  and new_lists = parfun snd tmp in
  let values_to_add = prescan op neutral last_elements in
    parfun2 map (parfun op values_to_add) new_lists

(* val scanl: (α →α →α )→α →α  list par→α  list par *)
let scanl op e =
  generic_scan sscan prescan_direct List.map last cutlast op e
```

Figure 10.14 — Diffusion directe, réduction et préfixe parallèles

```
  fold op neutral local_fold
```

avec sfold qui est une réduction séquentielle sur la collection et fold une réduction parallèle d'un vecteur. Cette fonction peut être instanciée (spécialisée) à une collection particulière, par exemple les listes avec une réduction parallèle directe :

```
let wide_fold_list_direct op e vv=
  generic_wide_fold List.fold_left fold_direct op e vv
```

Cette technique peut être appliquée au calcul des préfixes parallèles. generic_scan (figure 10.14) est une fonction générique qui peut calculer la somme des préfixes de n'importe quel vecteur parallèle de collections (de n'importe quel type).

generic_scan est une fonction qui opère sur un vecteur de collections d'éléments. Elle prend en paramètre :

- Une structure de données qui représente la collection des éléments ;
- Une fonction séquentielle qui réduit les éléments de la collection ;
- Une fonction parallèle prescan dont la sémantique est :

$$\text{prescan } \oplus \boxed{v_0 \mid \cdots \mid v_{p-1}} = \boxed{\imath_\oplus \mid v_0 \mid \cdots \mid \oplus_{k=0}^{p-2} v_k}$$

- Une fonction qui itère sur ce genre de collection,
- Une fonction qui prend (last) et enlève (cutlast) le dernier élément de la collection

La figure 10.14 donne le code d'une prescan. Mais il est possible de l'écrire d'une manière plus simple :

```
let prescan_direct op neutral v =
  let com = get_list v (mkpar(fun i→from_to 0 (i−1))) in
  parfun (List.fold_left op neutral) com
```

qui utilise la fonction de communication get_list définie comme suit :

$$\text{get\_list}\ \boxed{v_0 \mid \cdots \mid v_{p-1}}\ \boxed{[i_1^0;\ldots;i_{k_0}^0] \mid \cdots \mid [i_1^{p-1};\ldots;i_{k_{p-1}}^j]}$$

$$= \boxed{[v_{i_1^0};\ldots;v_{i_{k_0}^0}] \mid \cdots \mid [v_{i_1^{p-1}};\ldots;v_{i_{k_{p-1}}^j}]}$$

Maintenant, nous allons détailler le fonctionnement de generic_scan. Pour simplifier, nous considérons que les collections sont des listes. Les arguments séquentiels doivent satisfaire les égalités suivantes :

$$\begin{array}{rcl} \text{sscan} & : & (\alpha \to \alpha \to \alpha) \to \alpha\ \text{list} \to \alpha\ \text{list} \\ \text{sscan}\ (\oplus)\ [x_1, x_2, \ldots, x_n] & = & [i_\oplus; x_1; x_1 \oplus x_2; \ldots; x_1 \oplus x_2 \oplus \ldots \oplus x_n] \end{array}$$

$$\begin{array}{rcl} \text{cutlast} & : & \alpha\ \text{list} \to \alpha\ \text{option} * \alpha\ \text{list} \\ \text{cutlast}\ [x_1;\ldots;x_n] & = & (\text{Some}\ x_n,\ [x_1;\ldots;x_{n-1}]) \\ \text{cutlast}\ [\,] & = & (\text{None},\ [\,]) \end{array}$$

$$\begin{array}{rcl} \text{last} & : & \alpha\ \text{list} \to \alpha\ \text{option} * \alpha\ \text{list} \\ \text{last}\ [x_1;\ldots;x_n] & = & (\text{Some}\ x_n,\ [x_1;\ldots;x_n]) \\ \text{last}\ [\,] & = & (\text{None},\ [\,]) \end{array}$$

La première étape consiste à calculer les réductions locales, c'est-à-dire que si nous avons :

$$\text{vv} = \boxed{[x_1; x_2; \ldots; x_{n/p}] \mid \cdots \mid [x_{n+1-n/p}; x_{n+2-n/p}; \ldots; x_n]}$$

alors :

$$\text{local\_scan} = \boxed{[\text{c}; \text{op2 c}\ x_1; \ldots; \text{op2}\ (\ldots)\ x_{n/p}] \mid \cdots \mid [\text{c}; \text{op2 c}\ x_{n+1-n/p}; \ldots; \text{op2}\ (\ldots)\ x_{n/p}]}$$

Chaque processeur contient $\frac{n}{p} + 1$ valeurs. Pour chaque processeur, excepté le dernier, nous enlevons le dernier élément de la collection. Le vecteur tmp est alors un vecteur de paires dont la première composante est $(\text{Some}\ v_i)$ (v_i étant le dernier élément de la liste) et dont la seconde composante est la liste sans ledit dernier élément. Au processeur $p-1$, la seconde composante est une liste qui n'a pas été modifiée. A partir de ce vecteur, nous obtenons last_elements qui est un vecteur composé avec les derniers éléments et new_lists, un vecteur parallèle de listes sans leur dernier élément. values_to_add calcule la réduction partielle du vecteur last_elements. Au processeur i, seules les $i-1$ premières valeurs seront réduites avec prescan. Pour terminer, les valeurs obtenues sont ajoutées au vecteur de listes new_lists.

10.6.2 Implantation du patron algorithmique «Diffusion»

Les langages à patrons [69, 221, 226] (appelés aussi squelettes algorithmiques) sont des langages dans lesquels seul un ensemble fini de primitives (les patrons) sont parallèles. Du point de vue d'un programmeur, ces langages constituent une approche aisée de la programmation parallèle.

Dans cette section, nous montrons empiriquement qu'il est possible d'implanter des patrons parallèles en MSPML. Nous avons choisi le patron *Diffusion* [1]. Ce patron est dérivé du *théorème de diffusion* [153] et est défini en termes des patrons plus classiques **map**, **reduce** et **scan**. Il fournit une bonne abstraction de la combinaison de primitives parallèles. En utilisant le théorème de diffusion, des fonctions récursives, définies sous une forme spécifique et sous certaines conditions, peuvent être exprimées comme une instanciation du patron «Diffusion».

Un autre avantage est que, sous certains conditions, des techniques de «déforestation» peuvent être utilisées pour remplacer la composition de différents patrons de diffusion en une unique instantiation de ce patron [151].

Le patron de diffusion peut être défini comme suit avec le formalisme Bird-Meertens (BMF) [33, 239] :

$$\mathbf{diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ xs\ c \quad = \quad \mathbf{reduce}\ (\oplus)\ (\mathbf{map}\ k\ as)\ \oplus\ g_1\ b$$

```
let diff op1 op1neutral op2 k g1 g2 xs c =
  let reducer op neutral e vec =
    let local_fold = parfun (List.fold_left op neutral) vec in
    fold_direct op e local_fold in
  let bs'=scanl op2 c (parfun (List.map g2) xs) in
  let nocut l = None,l in
  let b',bs=parpair_of_pairpar(applyat (p()−1) cutlast nocut bs') in
  let (Some b)=at b' (p()−1) in
  reducer op1 op1neutral (g1 b) (parfun2 (List.map2 k) xs bs)
```

Figure 10.15 — Implantation MSPML du patron «Diffusion»

où $\oplus$ et $\otimes$ sont des opérations associatives avec un élément neutre. Nous avons :

$$\begin{aligned} as &= \mathbf{zip}\ xs\ bs \\ bs +\!\!+ [b] &= \mathbf{map}\ (c\otimes)\ (\mathbf{scan}\ \otimes\ (\mathbf{map}\ g_2\ xs)) \end{aligned}$$

où $+\!\!+$ est l'opération de concaténation de 2 listes et où

$$\begin{aligned} \mathbf{reduce}\ (\oplus)\ [x_1, x_2, \ldots, x_n] &= x_1 \oplus x_2 \oplus \ldots \oplus x_n \\ \mathbf{zip}\ [x_1, x_2, \ldots, x_n]\ [y_1, y_2, \ldots, y_n] &= [(x_1, y_1), (x_2, y_2, \ldots, (x_n, y_n)] \end{aligned}$$

Nous nous référons à [239] pour une liste exhaustive des opérations possibles en BMF.

L'implantation en MSPML du patron est donnée à la figure 10.15. Elle utilise seulement les fonctions décrites dans les sections et chapitres précédents.

10.6.3 Plus petits éléments

Pour tester expérimentalement notre implantation, nous donnons une version avec «diffusion» du programme des «plus petits éléments» dans une liste. Le problème est de retirer de la liste tous les éléments plus petits que leurs prédécesseurs dans ladite liste. La fonction qui calcule cette liste peut être écrite avec le patron «diffusion» comme suit :

```
let se_direct xs =
  let k x c = if x<c then [] else [x]
  and g1 x = []
  and g2 x = x in
  diff (@) [] max k g1 g2 xs min_int
```

Les opérations k, g1, g2 et max ont une complexité constante. La concaténation de 2 listes (@) a une complexité linéaire en la longueur de la première liste et est utilisée dans le diff comme argument pour la réduction parallèle reducer. Dans ce cas, son premier argument est le résultat d'une application de k : une liste vide ou bien une liste de longueur 1. Ainsi, le temps séquentiel nécessaire pour l'exécution de se_direct est linéaire en la taille de liste xs donnée en entrée (xs est en réalité un vecteur de listes ; la liste doit être vue comme la concaténation des listes du vecteur).

Les communications sont tout d'abord produites par scanl où en chaque processeur i le coût est $i \times g + L$ et ensuite par reducer qui est une réduction directe qui utilise un échange total de coût $(p-1) \times g + L$. Cet échange total implique une synchronisation globale et donc le coût total MPM des communications est $2 \times ((p-1) \times g + 2 \times L)$. Le coût des communications ne dépend donc pas de l'entrée. Ainsi, nos expériences feront apparaître des courbes linéaires.

10.6.4 Expériences

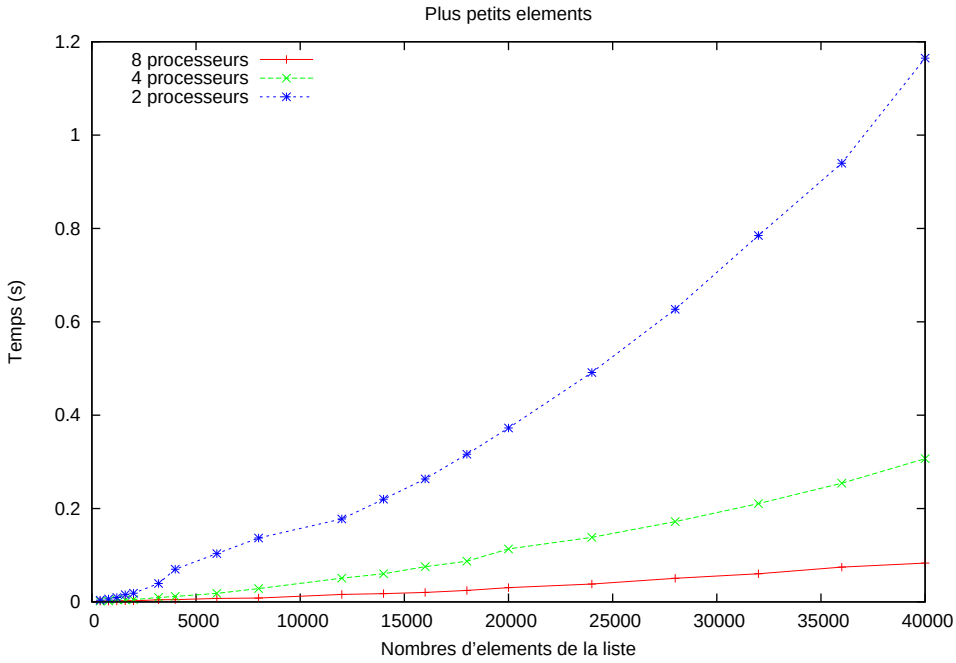
Nous avons réalisé des expériences de notre nouveau langage. L'implantation qui a été utilisée ici, possède un get primitif et non simulé par un **mget**. Les expériences réalisées dans l'article [193] montrent que le **mget** primitif est tout aussi efficace que le get.

Les tests de performances ont été effectués sur une grappe de 8 nœuds chacun doté de 256Go de RAM. Les nœuds sont des Intel pentium III 1 Ghz et interconnectés avec des cartes Fast Ethernet. Une Mandrake Clic 1.0 a été utilisée comme système d'exploitation et les programme ont été compilés avec OCaml 3.07.0 en mode natif.

La figure 10.16 (graphique du haut) montre le temps d'exécution en secondes mesuré pour l'évaluation de ce programme sur une liste dont la taille est donnée en abscisse. Cette liste est découpée en p sous-listes de même taille. La figure 10.16 (graphique du bas) montre le temps d'exécution en secondes de ce même programme où, en abscisse, sont notées les tailles de ces sous-listes.

Les mesures ont été obtenues sur une grappe de 2, 4 et 8 processeurs avec des listes de taille entre 200 et 20000 éléments par processeur. Chaque test a été effectué 4 fois et la moyenne a été prise, chaque test mesurant le temps d'exécution de 10 appels du programme sur la même liste.

Comme prévu, les courbes obtenues sont linéaires. Une accélération super-linéaire est obtenue lorsque le nombre de processeurs augmente. Ce n'est pas une surprise car avec moins de processeurs, chaque processeur a plus de données et donc plus d'appels au glaneur de cellules. Plus étranges sont les résultats obtenus à la figure 10.16 (graphique du bas) : pour une même taille par processeur, le temps d'exécution est moindre lorsque plus de processeurs sont utilisés. Ce phénomène est peut être dû à la stratégie de mise en tampon de la couche TCP/IP. Les tampons sont vidés plus rapidement lorsqu'il y a plus de processeurs car il y a plus de communications et donc les tampons sont pleins plus rapidement. Des expériences plus précises (et sur d'autres types de machines) restent à faire pour expliquer (et vérifier) ce phénomène.



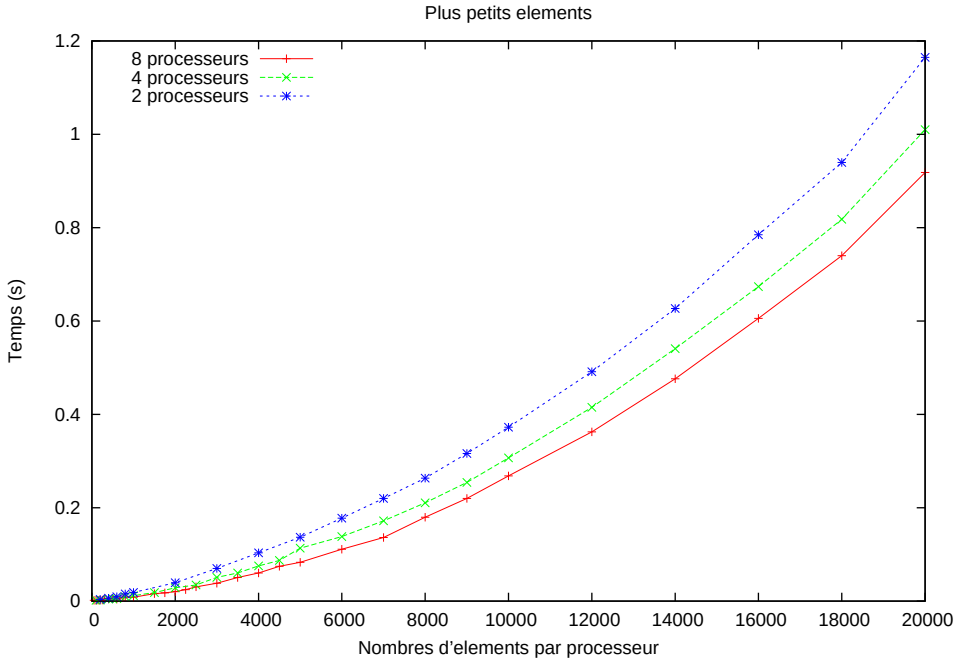


Figure 10.16 — Performances du calcul des plus petits éléments

10.A Annexe, preuves des lemmes

10.A.1 Confluence forte de $\rightharpoonup$

Lemme 74 (*Déterminisme des règles fonctionnelles*)

1. *Si* $e \overset{\varepsilon}{\rightharpoonup} e^1, c^1$ *et* $e \overset{\varepsilon}{\rightharpoonup} e^2, c^2$ *alors* $e^1 = e^2$ *et* $c^1 = c^2$;
2. *Si* $e \overset{\delta}{\rightharpoonup} e^1, c^1$ *et* $e \overset{\delta}{\rightharpoonup} e^2, c^2$ *alors* $e^1 = e^2$ *et* $c^1 = c^2$.

Preuve. Par cas sur les règles. ■

Lemme 75 (*Déterminisme des règles globales*)

1. *On a :*
   si $e/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}} e^1/\langle {c^1}_0^m, \ldots, {c^1}_{p-1}^m\rangle/\langle c_0^1, \ldots, c_{p-1}^1\rangle$
   et $e/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}} e^2/\langle {c^2}_0^m, \ldots, {c^2}_{p-1}^m\rangle/\langle c_0^2, \ldots, c_{p-1}^2\rangle$
   alors $e^1 = e^2$ *et* $\forall i\ {c_i^m}^1 = {c_i^m}^2$ *et* $c_i^1 = c_i^2$;
2. *On a :*
   si $e/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \underset{\bowtie}{\overset{\delta}{\rightharpoonup}} e^1/\langle {c^1}_0^m, \ldots, {c^1}_{p-1}^m\rangle/\langle c_0^1, \ldots, c_{p-1}^1\rangle$
   et $e/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \underset{\bowtie}{\overset{\delta}{\rightharpoonup}} e^2/\langle {c^2}_0^m, \ldots, {c^2}_{p-1}^m\rangle/\langle c_0^2, \ldots, c_{p-1}^2\rangle$
   alors $e^1 = e^2$ *et* $\forall i\ {c_i^m}^1 = {c_i^m}^2$ *et* $c_i^1 = c_i^2$.

Preuve. Par application du lemme 74 et par cas sur les règles des primitives et des opérations globales. ■

Lemme 76 (*Déterminisme des règles*)

1. *On a :*
   Si $e/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \overset{\bowtie}{\rightharpoonup} e^1/\langle {c^1}_0^m, \ldots, {c^1}_{p-1}^m\rangle/\langle c_0^1, \ldots, c_{p-1}^1\rangle$
   et $e/\langle c_0^m, \ldots, c_{p-1}^m\rangle/\langle c_0, \ldots, c_{p-1}\rangle \overset{\bowtie}{\rightharpoonup} e^2/\langle {c^2}_0^m, \ldots, {c^2}_{p-1}^m\rangle/\langle c_0^2, \ldots, c_{p-1}^2\rangle$
   alors $e^1 = e^2$ *et* $\forall i\ {c^1}_i^m = {c^2}_i^m$ *et* $c_i^1 = c_i^2$;
2. *Si* $e \overset{i}{\rightharpoonup} e^1, c^1$ *et* $e \overset{i}{\rightharpoonup} e^2, c^2$ *alors* $e^1 = e^2$ *et* $c^1 = c^2$.

Preuve. Par application du lemme 75 pour (1) et du lemme 74 pour (2). ■

Notons que les contextes des expressions sont identiques à ceux du chapitre 3. Nous utilisons donc les mêmes lemmes sur ces contextes.

Lemme 77 (*Déterminisme des contextes des threads de communications*)

1. $\forall i$, *si* $e = \Omega^i[e^1]$ *et* $e = \Omega^i[e^2]$ *alors* $e^1 = e^2$;
2. $\forall i, j\ i \neq j$ *si* $e = \Omega^i[e^1]$ *et* $e = \Omega^j[e^2]$ *alors* $e = \Gamma[[\![\cdots, e^1, \cdots, e^2, \cdots]\!]]$.

Lemme 78 (*Unicité du choix d'un type de contexte*)

Si $e = \Gamma[e']$ *alors* $\nexists\ \Omega^i$ *tel que* $e = \Omega^i[e'']$.

Preuve. Par construction de nos contextes. En effet, ceux-ci ne permettent l'application d'une règle :

- Globale que dans une unique sous-expression ;
- Locale que dans un unique vecteur parallèle mais possiblement dans deux composantes différentes de ce vecteur ;

- D'un thread de communications que dans un unique tableau de threads mais possiblement dans deux threads différents. ■

Lemme 79 (*Confluence forte*)

Soit une expression e.

Si	$e/\langle c_0^m,\ldots,c_{p-1}^m\rangle/\langle c_0,\cdots,c_{p-1}\rangle \rightharpoonup e^1/\langle c_0^{m^1},\ldots,c_{p-1}^{m^1}\rangle//\langle c_0^1,\cdots,c_{p-1}^1\rangle$
et	$e/\langle c_0^m,\ldots,c_{p-1}^m\rangle/\langle c_0,\cdots,c_{p-1}\rangle \rightharpoonup e^2/\langle c_0^{m^2},\ldots,c_{p-1}^{m^2}\rangle/\langle c_0^2,\cdots,c_{p-1}^2\rangle$
alors il existe	e^3, $\langle c_0^{m^3},\ldots,c_{p-1}^{m^3}\rangle$ *et* $\langle c_0^3,\cdots,c_{p-1}^3\rangle$
tel que	$e^1/\langle c_0^{m^1},\ldots,c_{p-1}^{m^1}\rangle//\langle c_0^1,\cdots,c_{p-1}^1\rangle \rightharpoonup e^3/\langle c_0^{m^3},\ldots,c_{p-1}^{m^3}\rangle/\langle c_0^3,\cdots,c_{p-1}^3\rangle$
et	$e^2/\langle c_0^{m^2},\ldots,c_{p-1}^{m^2}\rangle/\langle c_0^2,\cdots,c_{p-1}^2\rangle \rightharpoonup e^3/\langle c_0^{m^3},\ldots,c_{p-1}^{m^3}\rangle/\langle c_0^3,\cdots,c_{p-1}^3\rangle$.

Preuve. Par les lemmes 78 et 9, nous avons trois types de réductions distincts.

Si $\rightharpoonup$ est une réduction globale, alors par le lemme 7 il n'existe qu'un contexte global et par le lemme 76, la réduction est déterministe.

Si $\rightharpoonup$ est une réduction locale, alors nous avons $e = \Delta^i[e^i]$ et $e = \Delta^j[e^j]$. Nous avons alors deux cas :

1. Si $i = j$ alors par le lemme 8.1 il n'existe qu'un contexte et par le lemme 76, la réduction est déterministe
2. Si $i \neq j$ alors, comme précédemment, les règles peuvent s'entrelacer : les réductions interviennent dans deux composantes différentes d'un même vecteur.

Si $\rightharpoonup$ est une réduction d'un thread de communication, alors comme précédemment, soit la règle s'applique au même thread et par conséquent, le résultat est déterministe, soit les règles s'entrelacent pour un même tableau de threads. Les deux coûts sont ajoutés dans un ordre différent. Mais comme $\oplus$ est commutative, le coût final reste le même.

Les trois types de réductions sont donc fortement confluentes. ■

10.A.2 Confluence forte de $\rightsquigarrow$

Lemme 80 (*Déterminisme des règles fonctionnelles*)

1. *Si* $e \overset{\varepsilon}{\rightsquigarrow} e^1$ *et* $e \overset{\varepsilon}{\rightsquigarrow} e^2$ *alors* $e^1 = e^2$;
2. *Si* $e \overset{\delta}{\rightsquigarrow} e^1$ *et* $e \overset{\delta}{\rightsquigarrow} e^2$ *alors* $e^1 = e^2$.

Preuve. Par cas sur les règles. ■

Lemme 81 (*Déterminisme des règles globales*)

1. *Si* $\mathcal{E}_{\mathcal{C}i}/n_i/e \underset{\bowtie_i}{\overset{\varepsilon}{\rightsquigarrow}} \mathcal{E}_{\mathcal{C}}{}_i^1/n_i^1/e^1$ *et* $\mathcal{E}_{\mathcal{C}i}/n_i/e \underset{\bowtie_i}{\overset{\varepsilon}{\rightsquigarrow}} \mathcal{E}_{\mathcal{C}}{}_i^2/n_i^2/e^2$ *alors* $e^1 = e^2$, $\mathcal{E}_{\mathcal{C}}{}_i^1 = \mathcal{E}_{\mathcal{C}}{}_i^2$ *et* $n_i^1 = n_i^2$;
2. *Si* $e \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} e^1$ *et* $e \underset{\bowtie_i}{\overset{\delta}{\rightsquigarrow}} e^2$ *alors* $e^1 = e^2$.

Preuve. Par application du lemme 80 et par cas sur les règles des primitives et des opérations globales. ■

Lemme 82 (*Déterminisme des règles*)

1. *Si* $\mathcal{E}_{\mathcal{C}i}/n_i/e_i \overset{\bowtie_i}{\rightsquigarrow} \mathcal{E}_{\mathcal{C}}{}_i^1/n_i^1/e_i^1$ *et* $\mathcal{E}_{\mathcal{C}i}/n_i/e_i \overset{\bowtie_i}{\rightsquigarrow} \mathcal{E}_{\mathcal{C}}{}_i^2/n_i^2/e_i^2$ *alors* $e_i^1 = e_i^2$, $\mathcal{E}_{\mathcal{C}}{}_i^1 = \mathcal{E}_{\mathcal{C}}{}_i^2$ *et* $n_i^1 = n_i^2$;
2. *Si* $e_i \overset{i}{\rightsquigarrow} e_i^1$ *et* $e_i \overset{i}{\rightsquigarrow} e_i^2$ *alors* $e_i^1 = e_i^2$.

Preuve. Par application du lemme 81 pour (1) et du lemme 80 pour (2). ■

Notons que les contextes des expressions sont similaires à ceux du chapitre 3. Nous utilisons donc les mêmes lemmes sur ces contextes. Nous utilisons aussi le lemme des contextes sur les processus légers de communication de ce chapitre.

Lemme 83 (*Confluence forte*)

Soit $\langle\!\langle e_0, \ldots, e_{p-1}\rangle\!\rangle$ *un terme distribué.*

Si $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/e_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/e_{p-1}\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}{}^1/n_0^1/e_0^1, \ldots, \mathcal{E}_{\mathcal{C}p-1}{}^1/n_{p-1}^1/e_{p-1}^1\rangle\!\rangle$
et $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/e_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/e_{p-1}\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}{}^2/n_0^2/e_0^2, \ldots, \mathcal{E}_{\mathcal{C}p-1}{}^2/n_{p-1}^2/e_{p-1}^2\rangle\!\rangle$
alors il existe $\langle\!\langle \mathcal{E}_{\mathcal{C}0}{}^3/n_0^3/e_0^3, \ldots, \mathcal{E}_{\mathcal{C}p-1}{}^3/n_{p-1}^3/e_{p-1}^3\rangle\!\rangle$
tel que $\langle\!\langle \mathcal{E}_{\mathcal{C}0}{}^1/n_0^1/e_0^1, \ldots, \mathcal{E}_{\mathcal{C}p-1}{}^1/n_{p-1}^1/e_{p-1}^1\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}{}^3/n_0^3/e_0^3, \ldots, \mathcal{E}_{\mathcal{C}p-1}{}^3/n_{p-1}^3/e_{p-1}^3\rangle\!\rangle$
et $\langle\!\langle \mathcal{E}_{\mathcal{C}0}{}^2/n_0^2/e_0^2, \ldots, \mathcal{E}_{\mathcal{C}p-1}{}^2/n_{p-1}^2/e_{p-1}^2\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}{}^3/n_0^3/e_0^3, \ldots, \mathcal{E}_{\mathcal{C}p-1}{}^3/n_{p-1}^3/e_{p-1}^3\rangle\!\rangle$.

Preuve. Par les lemmes 78 et 9, nous avons trois types de réductions distincts. Les deux premières réductions se font respectivement au processeur i et j. La dernière est la réduction d'un processus léger de communication. Nous avons alors trois cas :

1. Si $i = j$ alors $\rightsquigarrow$ est une réduction (au processeur i) globale ou locale. Comme précédemment, la réduction est déterministe (lemme 82)
2. Si $i \neq j$ alors comme précédemment, les règles s'entrelacent et sont confluentes : les réductions interviennent dans deux composantes différentes d'un même vecteur distribué.
3. Si $\rightsquigarrow$ est une réduction (d'un processeur a) d'un processus léger b. Si c'est une réduction globale «normale» alors comme précédemment la règle est déterministe. Si c'est une réduction d'un **request** alors il est facile de constater que la valeur reçue est toujour la même car les règles de réduction sont déterministes.

Les trois types de réductions sont donc fortement confluentes. ■

10.A.3 Équivalence entre $\rightsquigarrow$ et $\rightharpoonup$

Dans cette section, nous faisons fi des coûts de la sémantique de $\rightharpoonup$ afin de simplifier la lecture de la preuve (les coûts n'apparaissant pas dans la sémantique $\rightsquigarrow$).

Nous notons $\overset{+}{\rightsquigarrow}$ la fermeture transitive de $\rightsquigarrow$.

Lemme 84 (*Un* $\rightharpoonup$ *vaut plusieurs* $\rightsquigarrow$)

Si $e \rightharpoonup e'$ *et* $\langle\!\langle \mathcal{E}_{\mathcal{C}0}{}'/n_0'/\mathcal{P}_0(e'), \ldots, \mathcal{E}_{\mathcal{C}p-1}{}'/n_{p-1}'/\mathcal{P}_{p-1}(e')\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}0}{}''/n_0''/v_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}{}''/n_{p-1}''/v_{p-1}\rangle\!\rangle$
alors $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}0}{}'/n_0'/\mathcal{P}_0(e'), \ldots, \mathcal{E}_{\mathcal{C}p-1}{}'/n_{p-1}'/\mathcal{P}_{p-1}(e')\rangle\!\rangle$.

Preuve. On montre d'abord le résultat pour une réduction de tête, en examinant les axiomes de réduction :

- Si $(\overline{\lambda.e[s]}\ v) \rightharpoonup e[v \circ s]$ alors

$$\begin{array}{rl} & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0((\overline{\lambda.e[s]}\ v)), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}((\overline{\lambda.e[s]}\ v))\rangle\!\rangle \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/(\overline{\lambda.\mathcal{P}_0(e)[\mathcal{P}_0(s)]}\ \mathcal{P}_0(v)), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/(\overline{\lambda.\mathcal{P}_{p-1}(e)[\mathcal{P}_{p-1}(s)]}\ \mathcal{P}_{p-1}(v))\rangle\!\rangle \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/(\overline{\lambda.e_0[s_0]}\ v_0), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/(\overline{\lambda.e_{p-1}[s_{p-1}]}\ v_{p-1})\rangle\!\rangle \\ \overset{*}{\rightsquigarrow} & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/e_0[v_0 \circ s_0], \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/e_{p-1}[v_{p-1} \circ s_{p-1}]\rangle\!\rangle \quad \text{par } p \text{ applications} \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e)[\mathcal{P}_0(v) \circ \mathcal{P}_0(s)], \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e)[\mathcal{P}_{p-1}(v) \circ \mathcal{P}_{p-1}(s)]\rangle\!\rangle \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e[v \circ s]), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e[v \circ s])\rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu.
- Si $(e^1\ e^2)[s] \rightharpoonup (e^1[s]\ e^2[s])$ alors

$$\begin{array}{rl} & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0((e^1\ e^2)[s]), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}((e^1\ e^2)[s])\rangle\!\rangle \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/(\mathcal{P}_0(e^1)\ \mathcal{P}_0(e^2))[\mathcal{P}_0(s)], \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/(\mathcal{P}_{p-1}(e^1)\ \mathcal{P}_{p-1}(e^2))[\mathcal{P}_{p-1}(s)]\rangle\!\rangle \text{ alors par } p \text{ applications} \\ \overset{*}{\rightsquigarrow} & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/(\mathcal{P}_0(e^1)[\mathcal{P}_0(s)]\ \mathcal{P}_0(e^2)[\mathcal{P}_0(s)]), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/(\mathcal{P}_{p-1}(e^1)[\mathcal{P}_{p-1}(s)]\ \mathcal{P}_{p-1}(e^2)[\mathcal{P}_{p-1}(s)])\rangle\!\rangle \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0((e^1[s]\ e^2[s])), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}((e^1[s]\ e^2[s]))\rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu.
- Si $(\mathbf{mkpar}\ f) \rightharpoonup \langle(f\ 0), \ldots, (f\ (p-1))\rangle$ alors (en utilisant le lemme 21 sur f) :

$$\begin{array}{rll} & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(\mathbf{mkpar}\ f), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(\mathbf{mkpar}\ f)\rangle\!\rangle & \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathbf{mkpar}\ \mathcal{P}_0(f), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathbf{mkpar}\ \mathcal{P}_{p-1}(f)\rangle\!\rangle & \text{alors par } p \text{ applications} \\ \overset{*}{\rightsquigarrow} & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\langle(\mathcal{P}_0(f)\ 0)\rangle, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\langle(\mathcal{P}_{p-1}(f)\ (p-1))\rangle\rangle & \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(\langle(f\ 0)\rangle), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(\langle(f\ (p-1))\rangle)\rangle\!\rangle & \end{array}$$

  ce qui est bien le résultat attendu.

- Tout les autres cas asynchrones (ne fonctionnant que sur un seul processeur à la fois) sont similaires.
- Si $\mathbf{get}\ j\ \langle v_0,\ldots,v_{p-1}\rangle\ \langle b_0,\ldots,b_{p-1}\rangle \rightharpoonup \langle v'_0,\ldots,v'_{p-1}\rangle$ tel que $\begin{cases} v'_k &=& v_j & \text{si } b_k = \mathbf{true} \\ v'_k &=& \mathbf{nc} & \text{sinon} \end{cases}$

  alors nous avons :

$$\begin{array}{rl} & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e),\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathbf{get}\ j\ \langle\mathcal{P}_0(v_0)\rangle\ \langle\mathcal{P}_0(b_0)\rangle,\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathbf{get}\ j\ \langle\mathcal{P}_{p-1}(v_{p-1})\rangle\ \langle\mathcal{P}_{p-1}(b_{p-1})\rangle\rangle\!\rangle \\ \overset{*}{\rightsquigarrow} & \langle\!\langle (n_0,j,v_0)\circ\mathcal{E}_{\mathcal{C}0}/n_0/a_0,\ldots,(n_{p-1}j,v_{p-1})\circ\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/a_{p-1}\rangle\!\rangle \text{ par } p \text{ applications} \end{array}$$

  Pour chaque a_i, nous avons deux cas :

  1. $a_i = \mathbf{nc}$ si $b_i = \mathbf{false}$
  2. $a_i = \mathbf{request}\ j\ n_i$. $\mathbf{request}$ n'est pas une valeur et donc par hypothèse $\mathbf{request}\ j\ n_i$ se réduit en v_j tel que $(n_i, j, v_j) \in \mathcal{E}_{\mathcal{C}j}$.

  Nous avons donc bien $\forall i\ \mathcal{P}_i(\langle v'_0,\ldots,v'_{p-1}\rangle) = a_i$ ce qui est bien le résultat attendu.

Pour finir la preuve, il faut montrer que tout cela passe bien au contexte, c'est-à-dire :

- si $\Gamma[e] \rightharpoonup \Gamma[e']$ alors $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(\Gamma[e]),\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(\Gamma[e])\rangle\!\rangle \overset{+}{\rightsquigarrow}$ $\langle\!\langle \mathcal{E}_{\mathcal{C}0}'/n'_0/\mathcal{P}_0(\Gamma[e']),\ldots,\mathcal{E}_{\mathcal{C}'p-1}/n'_{p-1}/\mathcal{P}_{p-1}(\Gamma[e'])\rangle\!\rangle$
- si $\Delta^i[e] \rightharpoonup \Delta^i[e']$ alors $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(\Delta[e]),\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(\Delta[e])\rangle\!\rangle \overset{+}{\rightsquigarrow}$ $\langle\!\langle \mathcal{E}_{\mathcal{C}0}'/n'_0/\mathcal{P}_0(\Delta[e']),\ldots,\mathcal{E}_{\mathcal{C}'p-1}/n'_{p-1}/\mathcal{P}_{p-1}(\Delta[e'])\rangle\!\rangle$
- si $\Omega^j[e] \rightharpoonup \Omega^j[e']$ alors $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(\Omega^j[e]),\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(\Omega^j[e])\rangle\!\rangle \overset{+}{\rightsquigarrow}$ $\langle\!\langle \mathcal{E}_{\mathcal{C}0}'/n'_0/\mathcal{P}_0(\Omega^j[e']),\ldots,\mathcal{E}_{\mathcal{C}'p-1}/n'_{p-1}/\mathcal{P}_{p-1}(\Omega^j[e'])\rangle\!\rangle$

Cela se fait par récurrence structurelle sur les contextes Γ, Δ^i et Ω^j :

- Le cas de base $\Gamma = []$ est immédiat avec le résultat précédent sur les réductions de tête.
- Cas d'un contexte global, le cas $\Gamma = (\Gamma'\ e_2)$. Nous avons alors $(\Gamma'[e]\ e^2) \rightharpoonup (\Gamma'[e']\ e^2)$ et :

$$\begin{array}{rl} & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(\Gamma'[e]\ e^2),\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(\Gamma'[e]\ e^2)\rangle\!\rangle \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/(\mathcal{P}_0(\Gamma'[e])\ \mathcal{P}_0(e^2)),\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/(\mathcal{P}_{p-1}(\Gamma'[e])\ \mathcal{P}_{p-1}(e^2))\rangle\!\rangle \\ \overset{*}{\rightsquigarrow} & \langle\!\langle \mathcal{E}_{\mathcal{C}0}'/n'_0/(\mathcal{P}_0(\Gamma'[e'])\ \mathcal{P}_0(e^2)),\ldots,\mathcal{E}_{\mathcal{C}'p-1}/n'_{p-1}/(\mathcal{P}_{p-1}(\Gamma'[e'])\ \mathcal{P}_{p-1}(e^2))\rangle\!\rangle \text{ par hypothèse de récurrence} \\ = & \langle\!\langle \mathcal{E}_{\mathcal{C}0}'/n'_0/\mathcal{P}_0(\Gamma'[e']\ e^2),\ldots,\mathcal{E}_{\mathcal{C}'p-1}/n'_{p-1}/\mathcal{P}_{p-1}(\Gamma'[e']\ e^2)\rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu. Les autres cas sont similaires.

- Cas d'un contexte local, le cas Δ^i. Par le lemme 8.2, nous avons $\Gamma[\langle\ldots,\overbrace{\Gamma'[e]}^{i},\ldots\rangle] \rightharpoonup \Gamma[\langle\ldots,\overbrace{\Gamma'[e']}^{i},\ldots\rangle]$. Donc nous avons :

$$\begin{array}{rl} & \langle\!\langle \ldots,\mathcal{E}_{\mathcal{C}i}/n_i/\mathcal{P}_i(\Gamma[\langle\Gamma'[e]\rangle]),\ldots\rangle\!\rangle \\ = & \langle\!\langle \ldots,\mathcal{E}_{\mathcal{C}i}/n_i/(\Gamma[\langle\mathcal{P}_i(\Gamma'[e])\rangle]),\ldots\rangle\!\rangle \\ \rightsquigarrow & \langle\!\langle \ldots,\mathcal{E}_{\mathcal{C}i}/n_i/(\Gamma[\langle\mathcal{P}_i(\Gamma'[e'])\rangle]),\ldots\rangle\!\rangle \text{ par hypothèse} \\ = & \langle\!\langle \ldots,\mathcal{E}_{\mathcal{C}i}/n_i/\mathcal{P}_i(\Gamma[\langle\Gamma'[e']\rangle]),\ldots\rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu.

- Cas d'un contexte de communication, le cas Ω^j. Par le lemme 77.2, nous avons $\Gamma[[\![\cdots,e^j,\cdots]\!]] \rightharpoonup \Gamma[[\![\cdots,e'^j,\cdots]\!]]$ Donc nous avons :

$$\begin{array}{rl} & \langle\!\langle \ldots,\mathcal{E}_{\mathcal{C}i}/n_i/\mathcal{P}_i(\Gamma[[\![\cdots,e^j,\cdots]\!]]),\ldots\rangle\!\rangle \\ = & \langle\!\langle \ldots,\mathcal{E}_{\mathcal{C}i}/n_i/\Gamma[[\![\cdots,\mathcal{P}_i(e^j),\cdots]\!]],\ldots\rangle\!\rangle \\ \rightsquigarrow & \langle\!\langle \ldots,\mathcal{E}_{\mathcal{C}i}/n_i/\Gamma[[\![\cdots,\mathcal{P}_i(e'^j),\cdots]\!]],\ldots\rangle\!\rangle \text{ par hypothèse} \\ = & \langle\!\langle \ldots,\mathcal{E}_{\mathcal{C}i}/n_i/\mathcal{P}_i(\Gamma[[\![\cdots,e'^j,\cdots]\!]]),\ldots\rangle\!\rangle \end{array}$$

  ce qui est bien le résultat attendu. ■

Lemme 85 (*Un $\rightsquigarrow$ implique l'existence d'un $\rightharpoonup$*)

Si $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e),\ldots,\mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}'/n'_0/e'_0,\ldots,\mathcal{E}_{\mathcal{C}'p-1}/n'_{p-1}/e'_{p-1}\rangle\!\rangle \overset{*}{\rightsquigarrow}$ $\langle\!\langle \mathcal{E}_{\mathcal{C}0}''/n''_0/v_0,\ldots,\mathcal{E}_{\mathcal{C}''p-1}/n''_{p-1}/v_{p-1}\rangle\!\rangle$ *alors* $\exists e'$ *tel que* $e \rightharpoonup e'$

Preuve. Par induction sur e.

- Cas $e = c$, $e = \mathbf{op}$, $e = \overline{n}$, $e = \lambda.e'$, $e = \overline{\lambda.e'[s]}$ et $e = \mu.e'$.
  Nous avons $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\ldots,\mathcal{P}_{p-1}(e)\rangle\!\rangle \not\rightsquigarrow$.

- Cas $e = e'[s]$. Nous avons donc $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle$
$= \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e')[\mathcal{P}_0(s)], \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e')[\mathcal{P}_{p-1}(s)]\rangle\!\rangle$.
Si $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e')[\mathcal{P}_0(s)], \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e')[\mathcal{P}_{p-1}(s)]\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}''_{\mathcal{C}0}/n''_0/e''_0, \mathcal{E}''_{\mathcal{C}p-1}/n''_{p-1}/\ldots, e''_{p-1}\rangle\!\rangle$
alors nous avons plusieurs cas pour chaque e'_i :
  - $e'_i = (e^1_i\ e^2_i)$ alors $e = (e^1\ e^2)[s] \rightharpoonup (e^1[s]\ e^2[s])$
  - $e'_i = \lambda.e^1_i$ alors $e = (\lambda.e^1)[s] \rightharpoonup \overline{(\lambda.e^1)[s]}$
  - Les cas $e'_i = \mathbf{op}$, $\mathbf{c}$, $\mu.e^1_i$, $\mathbf{mkpar}\ e^1$, $\overline{n}$ etc. sont similaires
- Cas $e = (e^1, e^2)$. Nous avons donc $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle$
$= \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/(\mathcal{P}_0(e^1), \mathcal{P}_0(e^2)), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/(\mathcal{P}_{p-1}(e^1), \mathcal{P}_{p-1}(e^2))\rangle\!\rangle$. Nous avons alors deux cas :
  - $\forall i\ \mathcal{P}_i(e^1) \neq v^1_i$. Par le lemme 24, nous avons $e^1 \neq v^1$ et donc par induction $e = (e^1, e^2) \rightharpoonup (e'^1, e^2) = e'$
  - $\forall i\ \mathcal{P}_i(e^1) = v^1_i$ et $\mathcal{P}_i(e^2) \neq v^2_i$. Par le lemme 24, nous avons $e^2 \neq v^2$ et $e^1 = v^1$ par le lemme 23. Nous avons donc par induction $e = (e^1, e^2) = (v^1, e^2) \rightharpoonup (v^1, e'^2) = e'$
- Cas $e = \mathbf{mkpar}\ e^1$. Nous avons donc $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle$
$= \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathbf{mkpar}\ \mathcal{P}_0(e^1), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathbf{mkpar}\ \mathcal{P}_{p-1}(e^1)\rangle\!\rangle$.
Or $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathbf{mkpar}\ \mathcal{P}_0(e^1), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathbf{mkpar}\ \mathcal{P}_{p-1}(e^1)\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/e'_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/e'_{p-1}\rangle\!\rangle$
donc $e^1_i = f_i$ (une valeur) et ceux $\forall i$. Donc $e^1 = f$ et par le lemme 25, $\mathcal{V}_\bullet(f) = \mathbf{true}$. Par conséquent
$\mathbf{mkpar}\ f \rightharpoonup \langle (f\ 0), \ldots, (f\ (p-1))\rangle[\bullet]$
- Les cas $e = \mathbf{apply}\ e^1\ e^2$, $\mathbf{proj}\ e^1$, $\mathbf{put}\ e^1$, et $\mathbf{if}\ e^1\ \mathbf{then}\ e^2\ \mathbf{e^3}$ sont similaires
- Cas $e = \langle e^0, \ldots, e^{p-1}\rangle$. Nous avons donc $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle$
$= \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\langle e^0\rangle, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\langle e^{p-1}\rangle\rangle\!\rangle$ avec
$\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\langle e^0\rangle, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\langle e^{p-1}\rangle\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\langle e^0\rangle, \ldots, \mathcal{E}_{\mathcal{C}i}/n_i/\langle e^i\rangle, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\langle e^{p-1}\rangle\rangle\!\rangle$.
Par le lemme 26, $e^i \xrightarrow{i} e'^i$. On construit $e' = \langle e^0, \ldots, e'^i, \ldots, e^{p-1}\rangle$ et nous avons $e \rightharpoonup e'$ car $e = \Delta^i[e_i]$ et $e' = \Delta^i[e'_i]$
- Cas $e = (e^1 e^2)$. Nous avons donc $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle$
$= \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/(\mathcal{P}_0(e^1)\ \mathcal{P}_0(e^2)), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/(\mathcal{P}_{p-1}(e^1)\ \mathcal{P}_{p-1}(e^2))\rangle\!\rangle$. Nous avons alors trois cas :
  - $\forall i\ \mathcal{P}_i(e^1) \neq v^1_i$. Par le lemme 24, nous avons $e^1 \neq v^1$ et donc par induction $e = (e^1\ e^2) \rightharpoonup (e'^1\ e^2) = e'$
  - $\forall i\ \mathcal{P}_i(e^1) = v^1_i$ et $\mathcal{P}_i(e^2) \neq v^2_i$. Par le lemme 24, nous avons $e^2 \neq v^2$ et $e^1 = v^1$ par le lemme 23. Nous avons donc par induction $e = (e^1\ e^2) = (v^1\ e^2) \rightharpoonup (v^1\ e'^2) = e'$
  - $\forall i\ \mathcal{P}_i(e^1) = v^1_i$ et $\mathcal{P}_i(e^2) = v^2_i$. Par le lemme 23 nous avons trois cas :
    - $\forall i\ v^1_i = \mathbf{op}$ donc $v^1 = \mathbf{op}$ et donc $(\mathbf{op}\ v^2) \rightharpoonup \overline{\mathbf{op}\ v^2}$
    - $\forall i\ v^1_i = \overline{\lambda.e''_i[s_i]}$ alors $e = (\overline{\lambda.e''[s]}\ v^2) \rightharpoonup e''[v^2 \circ s] = e'$
    - $\forall i\ v^1_i = \mathbf{request}$ et donc $v^2_i = (j, n_i)$. Alors par hypothèse on sait que
$\langle\!\langle \mathcal{E}'_{\mathcal{C}0}/n'_0/\mathbf{request}\ j_0\ n_0, \ldots, \mathcal{E}'_{\mathcal{C}p-1}/n'_{p-1}/\mathbf{request}\ j_{p-1}\ n_{p-1}\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle \mathcal{E}''_{\mathcal{C}0}/n''_0/v_0, \ldots, \mathcal{E}''_{\mathcal{C}p-1}/n''_{p-1}/v_{p-1}\rangle\!\rangle$.
Donc $\forall i\ \mathbf{request}\ j_i\ n_i$ se réduit en v_{j_i}. On construit donc
$e' = \langle v_{j_0}, \ldots, v_{j_{p-1}}\rangle$ et nous avons $e \rightharpoonup e'$ ce qui est bien le résultat attendu. ■

Proposition 7 (*Équivalence*)

Soit e^p une expression «programmeur» telle que $e = \mathcal{T}_\bullet(e^p)$. Alors :

1. *Si $e[\bullet]/\langle 0, \ldots, 0\rangle/\langle 0, \ldots, 0\rangle \xrightarrow{*} v/\langle c^m_0, \ldots, c^m_{p-1}\rangle/\langle c_0, \ldots, c_{p-1}\rangle$*
alors $\langle\!\langle \bullet/0/\mathcal{P}_0(e[\bullet]), \ldots, \bullet/0/\mathcal{P}_{p-1}(e[\bullet])\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/v_{p-1}\rangle\!\rangle$
tel que $\forall i \in \{0, \ldots, p-1\}\ \mathcal{P}_i(v) = v_i$;
2. *Si $\langle\!\langle \bullet/0/\mathcal{P}_0(e[\bullet]), \ldots, \bullet/0/\mathcal{P}_{p-1}(e[\bullet])\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/v_{p-1}\rangle\!\rangle$*
alors $e[\bullet]/\langle 0, \ldots, 0\rangle/\langle 0, \ldots, 0\rangle \xrightarrow{*} v/\langle c^m_0, \ldots, c^m_{p-1}\rangle/\langle c_0, \ldots, c_{p-1}\rangle$
tel que $\forall i \in \{0, \ldots, p-1\}\ \mathcal{P}_i(v) = v_i$.

Preuve. Nous avons :

1. Soit $e \xrightarrow{*} v$. On montre la conclusion par induction sur la longueur de cette réduction :
   - Si $e = v$ alors par le lemme 20, $\mathcal{P}_i(v) = v_i$ et donc $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(v), \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/\mathcal{P}_{p-1}(v)\rangle\!\rangle =$
$\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/v_{p-1}\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}p-1}/n_{p-1}/v_{p-1}\rangle\!\rangle$ où $\forall i\ \mathcal{P}_i(v) = v_i$.

- Si $e \rightharpoonup e' \overset{*}{\rightharpoonup} v$. Par hypothèse d'induction, $\langle\!\langle \mathcal{E}_{\mathcal{C}'_0}/n'_0/\mathcal{P}_0(e'), \ldots, \mathcal{E}_{\mathcal{C}'_{p-1}}/n'_{p-1}/\mathcal{P}_{p-1}(e')\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}''_0}/n''_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}''_{p-1}}/n''_{p-1}/v_{p-1}\rangle\!\rangle$ avec $\forall i\ \mathcal{P}_i(v) = v_i$. Par le lemme 84, $\langle\!\langle \bullet/0/\mathcal{P}_0(e), \ldots, \bullet/0/\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}'_{p-1}}/n'_{p-1}/\mathcal{P}_0(e'), \ldots, \mathcal{E}_{\mathcal{C}'_{p-1}}/n'_{p-1}/\mathcal{P}_{p-1}(e')\rangle\!\rangle$. Donc $\langle\!\langle \bullet/0/\mathcal{P}_0(e), \ldots, \bullet/0/\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}''_0}/n''_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}''_{p-1}}/n''_{p-1}/v_{p-1}\rangle\!\rangle$ où $\forall i\ \mathcal{P}_i(v) = v_i$.

2. Soit $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}_{p-1}}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}'_0}/n'_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}'_{p-1}}/n'_{p-1}/v_{p-1}\rangle\!\rangle$. On montre la conclusion par induction sur la longueur de cette réduction :
   - Si $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}_{p-1}}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle = \langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}_{p-1}}/n_{p-1}/v_{p-1}\rangle\!\rangle$ alors par le lemme 23, $e = v$ et donc $v \overset{*}{\rightharpoonup} v$; ce qui est bien le résultat attendu.
   - Si $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}_{p-1}}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{E}_{\mathcal{C}'_0}/n'_0/e'_0, \ldots, \mathcal{E}_{\mathcal{C}'_{p-1}}/n'_{p-1}/e'_{p-1}\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}''_0}/n''_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}''_{p-1}}/n''_{p-1}/v_{p-1}\rangle\!\rangle$ alors par le lemme 23, $\forall i\ \mathcal{P}_i(v) = v_i$ alors v est une valeur. Par le lemme 85, nous avons $\exists e'$ tel que $e \rightharpoonup e'$. Par lemme 22, nous avons $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}_{p-1}}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}'_0}/n'_0/\mathcal{P}_0(e'), \ldots, \mathcal{E}_{\mathcal{C}'_{p-1}}/n'_{p-1}/\mathcal{P}_{p-1}(e')\rangle\!\rangle$. Or par le théorème 6, la relation $\rightharpoonup$ est confluente, donc $\langle\!\langle \mathcal{E}_{\mathcal{C}'_0}/n'_0/\mathcal{P}_0(e'), \ldots, \mathcal{E}_{\mathcal{C}'_{p-1}}/n'_{p-1}/\mathcal{P}_{p-1}(e')\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle v_0, \ldots, \mathcal{E}_{\mathcal{C}''_{p-1}}/n''_{p-1}/v_{p-1}\rangle\!\rangle$. La longueur de cette dérivation est égale à celle de $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{*}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}''_0}/n''_0/v_0, \ldots, \mathcal{E}_{\mathcal{C}''_{p-1}}/n''_{p-1}/v_{p-1}\rangle\!\rangle$ moins la longueur de la dérivation $\langle\!\langle \mathcal{E}_{\mathcal{C}0}/n_0/\mathcal{P}_0(e), \ldots, \mathcal{E}_{\mathcal{C}_{p-1}}/n_{p-1}/\mathcal{P}_{p-1}(e)\rangle\!\rangle \overset{+}{\rightsquigarrow} \langle\!\langle \mathcal{E}_{\mathcal{C}'_0}/n'_0/\mathcal{P}_0(e'), \ldots, \mathcal{E}_{\mathcal{C}'_{p-1}}/n'_{p-1}/\mathcal{P}_{p-1}(e')\rangle\!\rangle$ qui est non nul. On peut donc appliquer l'hypothèse d'induction pour conclure que $e' \overset{*}{\rightharpoonup} v$. Par conséquent $e \overset{*}{\rightharpoonup} v$ avec $\forall i\ \mathcal{P}_i(v) = v_i$, ce qui est bien le résultat attendu. ■

11 Programmation de méta-ordinateurs

Le contenu de ce chapitre a été partiellement publié dans les articles [C3] et [R2] écrits avec Frédéric Loulergue.

Sommaire

DANS le chapitre précédent, nous avions vu comment permettre de désynchroniser les opérations de communications en utilisant le modèle MPM. Nous allons maintenant appliquer ce travail afin de définir un langage dédié au méta-calcul.

11.1 Introduction

Dans ce chapitre nous nous limitons au méta-calcul [213, 244] dit départemental [11, 12, 13], en ce sens que nous considérons que les machines parallèles font partie d'une même organisation et que le réseau les reliant, même si ses performances sont faibles, est fiable (le trafic y est à peu près constant). Nous évitons alors, pour la modélisation de l'architecture, les problèmes liés à la sécurité (celle-ci étant «assurée» par le serveur de l'organisation) et au manque de prédiction d'un réseau à plus grande échelle. Une topologie constante peut ainsi être établie pour caractériser notre méta-ordinateur, évitant par là-même les problèmes liés à la tolérance aux pannes qui rendent les prédictions des performances difficiles à analyser.

Nous allons considérer les grappes de machines parallèles comme des grappes de machines BSP. Les algorithmes pour ces architectures sont conçus selon le modèle *Deparmental Metacomputing Model* (DMM) décrit dans la section 11.2. Le langage associé *Departmental Metacomputing ML* (DMML, section 11.3) est un langage à deux niveaux de parallélisme. Chaque unité parallèle est programmée avec BSML (avec une légère modification) et la coordination est réalisée avec un niveau supplémentaire proche de MSPML, c'est-à-dire sans barrière de synchronisation. Outre le modèle et le langage, sont traitées dans cette section la sémantique (section 11.4), l'implantation et des expériences (section 11.5).

11.2 DMM, un modèle pour le méta-calcul départemental

Il existe un certain nombre de modèles pour le méta-calcul dont nous discuterons dans le chapitre 12. En particulier, le modèle BSP^2 [197] propose d'utiliser le modèle BSP pour chaque machine parallèle et un

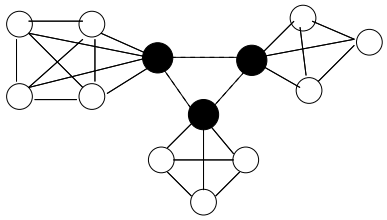


Figure 11.1 — Architecture d'un méta-ordinateur départemetal

niveau supplémentaire pour la coordination de ces machines. Ce modèle impose que les différentes unités BSP soient identiques et que le niveau supplémentaire soit semblable au modèle BSP. L'exécution procède en *hyper-étapes* qui se terminent par une synchronisation globale. Toutefois les algorithmes BSP^2 n'ont pas montré d'avantages significatifs par rapport aux algorithmes BSP et le modèle de prévision de performances n'a pas été très concluant. Ce modèle a néanmoins été mis en pratique pour le cas d'une grappe de machines parallèles à mémoire partagée.

Nous reprenons l'idée d'un niveau supplémentaire, non pas semblable au modèle BSP, mais semblable au modèle MPM. Nous introduisons ainsi une couche d'asynchronisme dans le modèle BSP.

La figure 11.1 illustre la structure générale d'un méta-ordinateur départemental. Les cercles noirs représentent les passerelles (serveurs) tandis que les cercles blancs représentent les unités de calculs. Nous considérons, tout au long de ce chapitre, qu'un méta-ordinateur est constitué d'unités BSP qui sont toutes reliées entre elles par un même réseau dit départemental. Chaque unité BSP possède un serveur qui permet la connexion au réseau départemental.

Nous appelons ce modèle *DMM* pour *Departmental Metacomputing Model*. Dans ce modèle, un méta-ordinateur est un ensemble de machines parallèles, chacune étant une machine BSP, reliées par un réseau permettant des communications point-à-point entre chaque unité. Nous prenons aussi en compte que beaucoup de machines parallèles sont différentes en autorisant dans le modèle DMM, des unités BSP hétérogènes. Un méta-ordinateur est caractérisé par les paramètres suivants :

- Les paramètres départementaux :
  - P est le nombre de machines parallèles constituant le méta-ordinateur ;
  - L est la latence du réseau reliant les machines parallèles entre elles ;
  - G est le temps nécessaire à l'échange de deux mots entre deux machines parallèles par le réseau départemental.
- Les paramètres des unités parallèles :
  - $\mathcal{P} = \{p_0, \ldots, p_{P-1}\}$ est la liste des nombres de processeurs de chaque unité parallèle
  - $\mathcal{S} = \{s_0, \ldots, s_{P-1}\}$ est la liste des vitesses des processeurs de chaque unité parallèle
  - $\mathcal{L} = \{l_0, \ldots, l_{P-1}\}$ est la liste des temps nécessaires à une barrière de synchronisation
  - $\mathcal{G} = \{g_0, \ldots, g_{P-1}\}$ est la liste des temps nécessaires à la réalisation d'une 1-relation.

L'exécution d'un programme DMM est une succession de d-étapes définies comme suit. À chaque d-étape, chaque unité BSP effectue une phase de calcul parallèle (séquence de super-étapes) puis une phase de communication avec d'autres unités parallèles du méta-ordinateur.

Un message envoyé d'un processeur d'une unité a vers un processeur d'une unité b va traverser le réseau local de l'unité a jusqu'à la passerelle, puis va traverser le réseau départemental jusqu'à la passerelle de l'unité b puis aller au processeur de destination en traversant le réseau local de l'unité b.

Pendant une phase de communication, les processeurs échangent les données nécessaires à la phase de calcul parallèle de la d-étape suivante. L'ensemble $\Omega_{(d,a)}$ est l'ensemble des partenaires entrants d'une unité BSP a à la d-étape d, c'est-à-dire l'ensemble des unités BSP qui échangent des messages avec l'unité BSP a à la phase de communication de la d-étape d.

Le temps d'exécution à la fin d'une d-étape d d'une unité BSP a est notée $\Phi_{(d,a)}$ et est définie par :

$$\begin{cases} \Phi_{(1,a)} = \max\limits_{b\in\Omega_{(1,a)}}(W_{(1,b)}, W_{(1,a)}) \\ \qquad\qquad + \max \begin{cases} \max\limits_{b\in\Omega_{(1,a)}}((g_b+G)\times h^b_{(1,a)}+l_b)+L+\sum\limits_{b\in\Omega_{(1,a)}}(h^b_{(1,a)})\times g_a+l_a \\ h^1_a\times g_a+l_a \end{cases} \\ \Phi_{(d,a)} = \max\limits_{b\in\Omega_{(d,a)}}(\Phi_{(d-1,b)}+W_{(d,b)}, \Phi_{(d-1,a)}+W_{(d,a)}) \\ \qquad\qquad + \max \begin{cases} \max\limits_{b\in\Omega_{(d,a)}}((g_b+G)\times h^b_{(d,a)}+l_b)+L+\sum\limits_{b\in\Omega_{(d,a)}}(h^b_{(d,a)})\times g_a+l_a \\ h^d_a\times g_a+l_a \end{cases} \end{cases}$$

pour $a \in \{0,\ldots,P-1\}$ et où $h^b_{(d,a)}$ est le nombre de mots reçus par l'unité a depuis l'unité b durant la d-étape d et h^d_a est le maximal de mots envoyés/reçus par les processeurs de l'unité a durant la d-étape d à d'autres processeurs de l'unité a et à la passerelle (on compte alors la somme des messages reçus par la passerelle et provenant des processeurs de l'unité a). $W_{(d,b)}$ est le travail parallèle (séquence de super-étapes) de l'unité BSP b :

$$W_{(d,b)} = \sum_{k=0}^{s^b_d}(\max_{i=0}^{p_b}(w^k_i)) \ + \ (\sum_{k=0}^{s^b_d}(\max_{i=0}^{p_b}(\dot{h}^k_i)))\times g_b \ + \ (s^b_d \times l_b)$$

où s^b_d est le nombre de super-étapes nécessaires à la phase de calcul parallèle de l'unité b pour la d-étape d et w^k_i est le temps nécessaire à la phase de calcul au processeur i pour la super-étape k et $\dot{h}^k_i = \max\{\dot{h}^k_{i+}, \dot{h}^k_{i-}\}$ où $\dot{h}^k_{i+}$ (resp. $\dot{h}^k_{i-}$) est le nombre de mots envoyés (resp. reçus) par le processeur i durant la super-étape k.

L'exécution d'un programme DMM est bornée par :

$$\Psi = \max\{\Phi_{(R,b)} | b \in \{0,1,\ldots,P-1\}\}$$

où R est le nombre de d-étapes nécessaires à l'exécution du programme.

11.3 ML pour le méta-calcul départemental

11.3.1 Noyau de primitives

La figure 11.3.1 donne les primitives DMML nécessaires à la programmation d'algorithmes DMM. La bibliothèque DMMLlib comprend les primitives de la bibliothèque BSMLlib : **mkpar**, **apply**, **put** et **proj**. Elle donne également accès aux paramètres DMM du méta-ordinateur. Par exemple, **dm_l**() est la latence, constante exprimée en unités de temps, du réseau départemental. Notons que les fonctions d'accès aux paramètres BSP sont différentes de celles que l'on trouve dans la bibliothèque BSMLlib. En effet ces fonctions prennent en argument le numéro d'unité. Par exemple, (**dm_bsp_p** 0) donne le nombre de processeurs de la première unité.

On a également une fonction supplémentaire **dm_bsp_s** donnant la vitesse des processeurs d'une unité donnée. Les fonctions d'accès aux paramètres l et g d'une unité ont également été modifiées sur un autre point : elles ne sont plus exprimées en fonction de la vitesse des processeurs mais en unité de temps. Dans le cas contraire, on pourrait, par exemple, avoir des valeurs identiques pour la performance de la barrière de synchronisation, pour des unités différentes, alors que ces valeurs mesurées en unité de temps seraient différentes.

Le second ensemble de primitives DMML rassemble les fonctions nécessaires à la manipulation des vecteurs *départementaux*, de taille P qui contiennent une valeur par unité BSP. Le type abstrait polymorphe pour ces vecteurs est α **dep**. Le type α peut être instancié soit par un type ML habituel soit par un type parallèle global. L'emboîtement de vecteurs parallèles et départementaux est interdite et une expression départementale ne peut être sous-expression d'une expression parallèle globale.

Les vecteurs départementaux sont créés à l'aide de **mkdep** de la même façon que les vecteurs parallèles le sont avec **mkpar**. L'application départementale **applydep** manipule des vecteurs départementaux comme **apply** manipule des vecteurs parallèles.

Paramètres BSP des unités :
dm_bsp_p: int→int **dm_bsp_s**: int→float **dm_bsp_g**: int→float **dm_bsp_l**: int→float

Paramètres MPM du méta-ordinateur :
dm_p: unit→int **dm_l**: unit→float **dm_g**: unit→float

Primitives BSP des unités :
mkpar: (int→α)→α **par**
apply: (α →β)**par**→α **par**→β **par**
put: (int→α option)**par**→(int→α option)**par**
proj: α option **par**→int→α option

Primitives MPM du méta-ordinateur :
mkdep: (int→α)→α **dep**
applydep: (α →β)**dep**→α **dep**→β **dep**
getdep: (int→int→int option)**par dep**→(int→α option)**par dep**→(int→int→α option)**par dep**
projdep: α option **par dep**→int→int→α option

Figure 11.2 — Primitives DMML

La projection **projdep** est similaire au **proj** avec la différence suivante : on projette la valeur d'un processeur (troisième argument) donné d'une unité donnée (deuxième argument).

Un algorithme DMM est une succession de phases de calculs parallèles BSP effectuées de façon asynchrone sur chaque unité BSP (première phase d'une d-étape) et de phases de communications entre les unités BSP (seconde phase d'une d-étape). La première phase est programmée à l'aide de **mkdep** et **applydep** et la seconde à l'aide de **projdep** et **getdep**. Considérons l'expression suivante :

getdep (**mkdep** (**fun** a→**mkpar** (**fun** i→$\mathsf{f}_{a,i}$))) (**mkdep** (**fun** b→**mkpar** (**fun** j→$\mathsf{v}_{b,j}$)))

Pour qu'un processeur i de l'unité BSP a, reçoive la $n$ème valeur d'un processeur j de l'unité BSP b la fonction $\mathsf{f}_{a,i}$ au processeur i de l'unité BSP a doit être telle que $(\mathsf{f}_{a,i}\ b\ j)$ s'évalue en Some n. Pour ne rien recevoir $(\mathsf{f}_{a,i}\ b\ j)$ doit s'évaluer en None.

Le résultat de l'évaluation d'une primitive **getdep** est un vecteur départemental de vecteurs parallèles de fonctions $\mathsf{f'}_{a,i}$ décrivant les messages reçus par chaque processeur i de chaque unité BSP a.

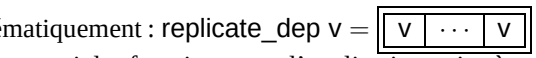
Au processeur i de l'unité BSP a $(\mathsf{f'}_{a,i}\ b\ j)$ vaut None si le processeur i de l'unité BSP a n'a pas demandé de message au processeur j de l'unité BSP b ou si $(v_{b,j}\ n)$ vaut None (le processeur j de l'unité BSP b n'a pas de $n$ème valeur). Cette expression vaudra Some $v^n_{b,j}$ si le processeur i de l'unité BSP a a reçu une valeur du processeur j de l'unité BSP b et si $(v_{b,j}\ n)$ vaut (Some $v^n_{b,j}$).

11.3.2 Premiers exemples asynchrones

Tout comme en BSML, on est souvent amené à employer un certain nombre de fonctions dans la programmation de programmes DMML. Cette section présente celles qui sont le plus utilisées.

Comme pour BSML, nous pouvons répliquer une même valeur en chaque composante d'un vecteur départemental :

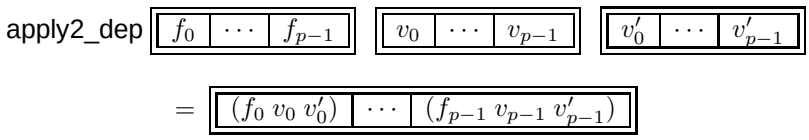
```
(* replicate_dep: α →α dep*)
let replicate_dep v = mkdep (fun clus→v)
```

où plus schématiquement : replicate_dep v = 

Nous avons aussi des fonctions pour l'application point-à-point de vecteurs départementaux et l'application d'une fonction en chaque composante d'un vecteur départemental :

```
(* apply2_dep: (α →β →γ )dep→α dep→β dep→γ dep *)
let apply2_dep f v1 v2 = applydep (applydep f v1) v2

(* parfun_dep: (α →β )→α dep→β dep *)
let parfun_dep f v = applydep (replicate_dep f) v
```

Par exemple :



et parfun_dep f $\boxed{v_0 \mid \cdots \mid v_{p-1}}$ $=$ $\boxed{(\mathsf{f}\ v_0) \mid \cdots \mid (\mathsf{f}\ v_{p-1})}$

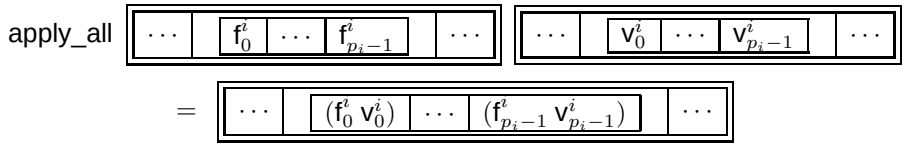
On retrouve aussi ce type de fonction pour les deux niveaux du langage. Nous avons donc :

(∗ replicate_all: $\alpha \rightarrow \alpha$ par dep ∗)
let replicate_all x = **mkdep** (**fun** a→replicate x)

qui correspond au schéma suivant : replicate_all v = $\boxed{\mathsf{v} \mid \cdots \mid \mathsf{v}} \cdots \boxed{\mathsf{v} \mid \cdots \mid \mathsf{v}}$
et

(∗ apply_all: ($\alpha \rightarrow \beta$)par dep→α par dep→β par dep ∗)
let apply_all gf gv = **applydep** (**applydep** (**mkdep** (**fun** a f v→**apply** f v)) gf) gv

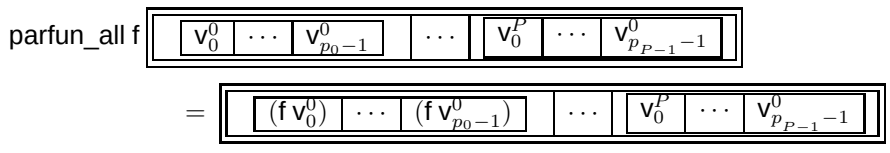
qui correspond au schéma suivant :



et

(∗ parfun_all: ($\alpha \rightarrow \beta$)$\rightarrow \alpha$ par dep→β par dep ∗)
let parfun_all f vv = apply_all (replicate_all f) vv

qui correspond au schéma suivant :



11.4 Sémantique dynamique

Nous présentons maintenant la sémantique formelle d'un mini-DMML.

11.4.1 Syntaxe

Les *expressions* de mini-DMML, notées e^p et ses variantes, ont la syntaxe abstraite suivante :

Définition 39 (*Langage source*).
Les expressions initiales sont définies par la grammaire suivante :

e^p	::=	$\lambda x.e^p$	*abstraction (fonction)*
	|	$(e^p\ e^p)$	*application*
	|	(e^p, e^p)	*paire*
	|	$\mathbf{mkpar}\ e^p$	*création d'un vecteur parallèle*
	|	$\mathbf{apply}\ e^p\ e^p$	*application parallèle*
	|	$\mathbf{put}\ e^p$	*communication parallèle*
	|	$\mathbf{proj}\ e^p$	*projection parallèle*
	|	$\mathbf{mkdep}\ e^p$	*création d'un vecteur départemental*
	|	$\mathbf{applydep}\ e^p\ e^p$	*application départementale*
	|	$\mathbf{getdep}\ e^p\ e^p$	*communication départementale*
	|	$\mathbf{projdep}\ e^p$	*projection départementale*
	|	$\mathbf{c}$	*constante*
	|	$\mathbf{op}$	*opération prédéfinie*
	|	$\mathbf{x}$	*variable*
	|	$\mu x.e^p$	*définition d'une fonction récursive*
	|	$\mathbf{if}\ e^p\ \mathbf{then}\ e^p\ \mathbf{else}\ e^p$	*conditionnelle*

Dans cette grammaire, $\mathbf{x}$ appartient à un ensemble dénombrable de variables. Les constantes contiennent les entiers, les booléens ainsi que $()$ et la valeur $\mathbf{nc}$ qui correspond au None de OCaml dans les primitives de communications. Les opérations prédéfinies $\mathbf{op}$ sont les opérations usuelles comme, par exemple, celles sur les entiers où les booléens ainsi que les primitives d'accès aux paramètres de la machine DMM. Cette syntaxe est celle du programmeur, mais la réduction d'une expression DMML peut créer des vecteurs parallèles et départementaux énumérés. Les vecteurs parallèles seront notés $\langle e, \ldots, e\rangle$. Les vecteurs départementaux seront, eux, notés $\langle\!\|e, \ldots, e\|\!\rangle$. Comme dans le chapitre 3, nous avons la grammaire des termes de l'évaluation. Ces termes sont définis comme dans le chapitre 3 :

Définition 40 (*Expressions des sémantiques*).
Nos expressions ont donc la forme suivante :

e	::=	$(e)[s]$	*expression munie d'une substitution*
	|	$\lambda.e$	*abstraction*
	|	$\overline{(\lambda.e)[s]}$	*fermeture*
	|	$(e\ e)$	*application*
	|	(e, e)	*paire*
	|	$\mathbf{mkpar}\ e$	*création d'un vecteur parallèle*
	|	$\mathbf{apply}\ e\ e$	*application parallèle*
	|	$\mathbf{put}\ e$	*communication parallèle*
	|	$\mathbf{proj}\ e$	*projection parallèle*
	|	$\mathbf{mkdep}\ e$	*création d'un vecteur départemental*
	|	$\mathbf{applydep}\ e\ e$	*application départementale*
	|	$\mathbf{getdep}\ e$	*communication départementale*
	|	$\mathbf{projdep}\ e$	*projection départementale*
	|	$\mathbf{c}$	*constante*
	|	$\mathbf{op}$	*opérateur*
	|	$\overline{\mathbf{n}}$	*variables de substitution (indices)*
	|	$\mu.e$	*définition d'une fonction récursive*
	|	$\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$	*conditionnelle*
	|	$\langle e, \ldots, e\rangle$	*vecteur parallèle*
	|	$\langle\!\|e, \ldots, e\|\!\rangle$	*vecteur départemental*
	|	$[e, \ldots, e]$	*tableau fonctionnel*

où $\mathbf{op}' ::= \mathbf{op} \cup \{\mathbf{delpar}, \mathbf{deldep}, \mathbf{init}, \mathbf{access}, \mathbf{send}, \mathbf{senddep}, \mathbf{mkanswer}\}$.

Il y a une sémantique par taille de machine DMM. Par «taille», nous entendons nombre P d'unités BSP et nombre p_a de processeurs dans chaque unité BSP a. La taille des vecteurs départementaux est P, celle des vecteurs parallèles peut être n'importe quel p_a. Les valeurs et les substitutions sont :

$$
\begin{array}{rcl}
\mathcal{T}_{\mathcal{E}}(\lambda x.e^p) & = & \lambda.T_{\{x\mapsto\overline{1},R_x(\mathcal{E})\}}(e^p) \\
\mathcal{T}_{\mathcal{E}}(\mu x.e^p) & = & \mu.T_{\{x\mapsto\overline{1},R_x(\mathcal{E})\}}(e^p) \\
\mathcal{T}_{\mathcal{E}}((e_1^p\ e_2^p)) & = & (\mathcal{T}_{\mathcal{E}}(e_1^p)\ \mathcal{T}_{\mathcal{E}}(e_2^p)) \\
\mathcal{T}_{\mathcal{E}}((e_1^p, e_2^p)) & = & (\mathcal{T}_{\mathcal{E}}(e_1^p), \mathcal{T}_{\mathcal{E}}(e_2^p)) \\
\mathcal{T}_{\mathcal{E}}(\mathbf{c}) & = & \mathbf{c} \\
\mathcal{T}_{\mathcal{E}}(\mathbf{op}) & = & \mathbf{op}
\end{array}
\quad\Big|\quad
\begin{array}{rcl}
\mathcal{T}_{\mathcal{E}}(\mathbf{apply}\ e_1^p\ e_2^p) & = & \mathbf{apply}\mathcal{T}_{\mathcal{E}}(e_1^p)\ \mathcal{T}_{\mathcal{E}}(e_2^p) \\
\mathcal{T}_{\mathcal{E}}(\mathbf{put}\ e^p) & = & \mathbf{put}\ \mathcal{T}_{\mathcal{E}}(e^p) \\
\mathcal{T}_{\mathcal{E}}(\mathbf{proj}\ e^p) & = & \mathbf{proj}\ \mathcal{T}_{\mathcal{E}}(e^p) \\
\mathcal{T}_{\mathcal{E}}(\mathbf{applydep}\ e_1^p\ e_2^p) & = & \mathbf{applydep}\mathcal{T}_{\mathcal{E}}(e_1^p)\ \mathcal{T}_{\mathcal{E}}(e_2^p) \\
\mathcal{T}_{\mathcal{E}}(\mathbf{getdep}\ e_1^p\ e_2^p) & = & \mathbf{detdep}\mathcal{T}_{\mathcal{E}}(e_1^p)\ \mathcal{T}_{\mathcal{E}}(e_2^p) \\
\mathcal{T}_{\mathcal{E}}(\mathbf{projdep}\ e^p) & = & \mathbf{projdep}\ \mathcal{T}_{\mathcal{E}}(e^p) \\
\mathcal{T}_{\mathcal{E}}(\mathbf{mkpar}\ e^p) & = & \mathbf{mkpar}\ \mathcal{T}_{\mathcal{E}}(e^p) \\
\mathcal{T}_{\mathcal{E}}(\mathbf{mkdep}\ e^p) & = & \mathbf{mkpar}\ \mathcal{T}_{\mathcal{E}}(e^p)
\end{array}
$$

$$\mathcal{T}_{\mathcal{E}}(\mathbf{if}\ e_1^p\ \mathbf{then}\ e_2^p\ \mathbf{else}\ e_3^p) = \mathbf{if}\ \mathcal{T}_{\mathcal{E}}(e_1)\ \mathbf{then}\ \mathcal{T}_{\mathcal{E}}(e_2)\ \mathbf{else}\ \mathcal{T}_{\mathcal{E}}(e_3)$$

$$\mathcal{T}_{\mathcal{E}}(x) = \overline{n} \text{ si } \mathcal{E} = \{\cdots, x \mapsto \overline{n}, \cdots, \bullet\}$$

Figure 11.3 — Instanciation des variables en des indices de De Brujin

Définition 41 (*Substitutions et valeurs*).
Les substitutions et les valeurs (sous-ensemble des expressions) sont classiquement définies par :

$$
\begin{array}{rcll}
s & ::= & \bullet & \textit{substitution vide} \\
 & | & v \circ s & \textit{valeur suivie de la suite de la substitution}
\end{array}
$$

$$v ::= \overline{(\lambda.e)[s]} \mid \mathbf{c} \mid \mathbf{op} \mid (v, v) \mid \langle v, \ldots, v\rangle \mid \langle\!\langle v, \ldots, v|\!|\rangle$$

Les valeurs sont les fermetures, les constantes, les opérateurs et les vecteurs énumerés parallèles ou départementaux.

La tranformation des termes «programmeur» en des termes «d'évaluation» fonctionne de la même manière similaire qu'au chapitre 3, c'est-à-dire le remplacement des variables par des indices de De Brujin. Ceci est défini inductivement (figure 11.3) où $\mathcal{E}$ est un environnement de substitution des variables (un dictionnaire entre les variables et les indices) défini par :

$$\mathcal{E} ::= \bullet \mid \{x \mapsto \overline{n}, \mathcal{E}\}$$

avec la fonction suivante de mise à jour de l'environnement :

$$
\begin{array}{rcl}
\mathcal{R}_x(\bullet) & = \bullet & \\
\mathcal{R}_x(\{x \mapsto \overline{n}, \mathcal{E}\}) & = & \mathcal{R}_x(\mathcal{E}) \\
\mathcal{R}_x(\{y \mapsto \overline{n}, \mathcal{E}\}) & = & \{y \mapsto \overline{n+1}, \mathcal{R}_x(\mathcal{E})\} \quad \text{si } y \neq x
\end{array}
$$

Propriété 4

Soit l'expression e^p telle que $e = \mathcal{T}_\bullet(e^p)$, alors e^p est sans variables libres (ainsi que e par conséquent).

Preuve . Par induction triviale sur la traduction de l'expression e^p. ■

11.4.2 Règles de réduction

La sémantique de mini-DMML est donnée ici sous la forme d'une sémantique à «petits pas». La relation de réduction est définie sur les expressions. Elle décrit pas à pas comment une expression est réduite jusqu'à une valeur. La réduction est notée $\rightharpoonup$ et sa fermeture réflexive transitive $\overset{*}{\rightharpoonup}$. Notons que nous ne donnerons pas les coûts formels des primitives. Ceux-ci ce déduisent facilement des règles comme nous l'avions fait avec BSML et MSPML.

Pour définir cette relation, nous commençons par la définition de trois relations, une pour chaque sorte d'expression : locales (expressions ML habituelles), parallèles (expressions BSML) et départementales (expressions DMML). Nous noterons les numéros d'unités BSP a, b et variantes et les numéros de processeurs i, j et variantes.

$$
\begin{array}{rcl}
(\mathbf{isnc\ nc}) & \xrightarrow[\delta]{\varepsilon} & \mathbf{true} \\
(\mathbf{isnc}\ v) & \xrightarrow[\delta]{\varepsilon} & \mathbf{false}\ \text{si}\ v \neq \mathbf{nc} \\
(\mathbf{dm\_p}\ ()) & \xrightarrow[\delta]{\varepsilon} & \mathbf{P} \\
\mathbf{init}\ (n, f) & \xrightarrow[\delta]{\varepsilon} & [(f\ 0), \ldots, (f\ (n-1))], 1 \\
+\ (n_1, n_2) & \xrightarrow[\delta]{\varepsilon} & n_1 + n_2, 1
\end{array}
\qquad
\begin{array}{rcl}
\mathbf{if\ true\ then}\ e_1\ \mathbf{else} e_2 & \xrightarrow[\delta]{\varepsilon} & e_1 \\
\mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2 & \xrightarrow[\delta]{\varepsilon} & e_2 \\
\mathbf{access}\ ([v_0, \ldots, v_i, \ldots, v_{p-1}], i) & \xrightarrow[\delta]{\varepsilon} & v_i, 1 \\
(\mathbf{dm\_bsp\_p}\ a) & \xrightarrow[\delta]{\varepsilon} & \mathbf{p}_a \\
\mathbf{fst}\ (v_1, v_2) & \xrightarrow[\delta]{\varepsilon} & v_1, 1 \\
\mathbf{snd}\ (v_1, v_2) & \xrightarrow[\delta]{\varepsilon} & v_2, 1
\end{array}
$$

Figure 11.4 — Opérations fonctionnelles prédéfinies

Définition 42 (*Relations de la sémantique de DMML*).

1. $e \underset{i,a}{\rightharpoonup} e'$ *au processeur* i *de l'unité BSP* a *l'expression* e *est réduite en* e' *;*
2. $e \underset{\bowtie_a}{\rightharpoonup} e'$ *à l'unité BSP* a*, l'expression* e *est réduite en* e' *;*
3. $e \underset{\Leftrightarrow}{\rightharpoonup} e'$ *: l'expression* e *est réduite en* e' *par tout le méta-ordinateur.*

avec les trois relations définies comme suit :

$$\underset{i,a}{\rightharpoonup} = \xrightarrow{\varepsilon} \cup \xrightarrow[\delta]{\varepsilon} \quad \textit{et} \quad \underset{\bowtie_a}{\rightharpoonup} = \xrightarrow[\bowtie_a]{\varepsilon} \cup \xrightarrow[\delta]{\varepsilon} \cup \xrightarrow{\varepsilon} \quad \textit{et} \quad \underset{\Leftrightarrow}{\rightharpoonup} = \xrightarrow[\Leftrightarrow]{\varepsilon} \cup \xrightarrow[\delta]{\varepsilon} \cup \xrightarrow{\varepsilon}$$

Nous commençons d'abord par une série d'axiomes (règles) qui sont communs aux deux relations. Chacune de ces relations contient la relation $\xrightarrow{\varepsilon}$. Pour commencer nous avons les règles pour la β-réduction :

$$
\begin{array}{rcl}
(\overline{(\lambda.e)[s]}\ v) & \xrightarrow{\varepsilon} & e[v \circ s] \\
\overline{n+1}[v \circ s] & \xrightarrow{\varepsilon} & \overline{n}[s] \\
\overline{1}[v \circ s] & \xrightarrow{\varepsilon} & v \\
(\mu.e)[s] & \xrightarrow{\varepsilon} & e[\mu.e \circ s]
\end{array}
$$

ensuite nous avons celles pour la propagation de substitution :

$$
\begin{array}{rcl}
(\lambda.e)[s] & \xrightarrow{\varepsilon} & \overline{(\lambda.e)[s]} \\
\mathbf{op}[s] & \xrightarrow{\varepsilon} & \mathbf{op} \\
\mathbf{c}[s] & \xrightarrow{\varepsilon} & \mathbf{c} \\
(e_1\ e_2)[s] & \xrightarrow{\varepsilon} & (e_1[s]\ e_2[s]) \\
(e_1, e_2)[s] & \xrightarrow{\varepsilon} & (e_1[s], e_2[s]) \\
(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)[s] & \xrightarrow{\varepsilon} & \mathbf{if}\ e_1[s]\ \mathbf{then}\ e_2[s]\ \mathbf{else}\ e_3[s] \\
(\mathbf{PRIM}\ e)[s] & \xrightarrow{\varepsilon} & \mathbf{PRIM}(e[s]) \quad \text{où}\ \mathbf{PRIM} = \{\mathbf{proj}, \mathbf{put}, \mathbf{mkpar}, \mathbf{mkdep}, \mathbf{projdep}\} \\
(\mathbf{PRIM}'\ e_1\ e_2)[s] & \xrightarrow{\varepsilon} & (\mathbf{PRIM}'\ e_1[s]\ e_2[s]) \quad \text{où}\ \mathbf{PRIM}' = \{\mathbf{apply}, \mathbf{applydep}, \mathbf{getdep}\}
\end{array}
$$

Nous avons ensuite des règles pour les opérations prédéfinies. Ces règles sont usuelles et nous n'en donnons que quelques unes à la figure 11.4. Les règles pour les primitives parallèles et départementales sont données respectivement aux figures 11.5 et 11.6.

Nous donnons ici la sémantique BSML de la primitive **put** en la décomposant en deux étapes, correspondant fidèlement à l'implantation actuelle de BSML (voir aussi aux chapitres 2 et 3). En premier lieu, chaque processeur crée un tableau purement fonctionnel de valeurs en appliquant la fonction qu'il détient à tous les numéros de processeurs possibles dans l'unité où la réduction a lieu. Notons que l'opération de création d'un tableau prend, dans cette sémantique, un argument entier définissant la taille du tableau. Ensuite une primitive de plus bas niveau **send** fait les échanges et retourne un vecteur parallèle de tableaux. Cette règle (11.6) est une formalisation du mécanisme décrit dans le chapitre 5. La valeur à l'indice j du tableau au processeur i est envoyée, si ce n'est pas **nc**, au processeur j qui la stocke à l'indice i du tableau résultat. La fonction **mkf** construit le vecteur parallèle de fonctions à partir du vecteur parallèle de tableaux.

$$\mathbf{mkpar}\ v \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad \langle (v\ 0), \ldots, (v\ (p_a - 1)) \rangle \text{ si } \mathcal{V}^g_\bullet(v) = \mathbf{true} \tag{11.1}$$

$$\mathbf{apply}\ \langle v_0, \ldots, v_{p_a-1} \rangle\ \langle v'_0, \ldots, v'_{p_a-1} \rangle \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad \langle (v_0\ v'_0), \ldots, (v_{p_a-1}\ v'_{p_a-1}) \rangle \tag{11.2}$$

$$\mathbf{proj}\ \langle \ldots, v_i, \ldots \rangle \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad \mathbf{delpar}\,(\mathbf{put}\ \langle \ldots, (\lambda.v_i)[\bullet], \ldots \rangle) \tag{11.3}$$

$$\mathbf{delpar}\ \langle v, \ldots, v \rangle \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad v \tag{11.4}$$

$$\mathbf{put}\ \langle v_0, \ldots, v_i, \ldots, v_{p_a-1} \rangle \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad (\mathbf{mkf}\ (\mathbf{send}\ \langle \ldots, (\mathbf{init}\ v_i\ p_a), \ldots \rangle)) \tag{11.5}$$

$$\mathbf{send}\ \langle [v_0^0, .., v_0^{p_a-1}], \ldots, [v_{p_a-1}^0, .., v_{p_a-1}^{p_a-1}] \rangle \quad \xrightarrow[\bowtie_a]{\varepsilon} \quad \langle [v_0^0, .., v_{p_a-1}^0], \ldots, [v_0^{p_a-1}, .., v_{p_a-1}^{p_a-1}] \rangle \tag{11.6}$$

où $\mathbf{mkf} = \mathbf{apply}(\mathbf{mkpar}(\lambda.\lambda.\mathbf{if}\ (0 \leq \overline{1})\&(\overline{1} < (\mathbf{dm\_bsp\_p}\ \overline{3}))\ \mathbf{then}\ (\mathbf{access}\ \overline{2}\ \overline{1})\ \mathbf{else}\ \mathbf{nc}))$

Figure 11.5 — Réductions parallèles BSP

Les règles pour la réduction départementale de tête sont sans surprise pour les primitives **mkdep** et **applydep**. La création d'un vecteur départemental ne peut se faire que si la fonction ne contient pas elle-même une référence à un autre vecteur. Notons que pour éviter l'emboîtement des vecteurs (parallèles et départementaux), nous avons dû modifier la définition du test du **mkpar**. Les deux tests sont inductivement (et facilement) définis à la figure 11.11. Nous ne les détaillons pas.

La primitive **projdep**, comme **proj**, utilise la primitive de communication pour diffuser les valeurs puis une opération de destruction du type **dep**. Le traitement du **getdep** impose, comme pour le **put** au niveau parallèle global, d'introduire deux opérations de plus bas niveau : **senddep** et **mkanswer**.

La première (règle 11.12) est une adaptation au niveau départemental du **send** du niveau global. L'argument est un vecteur départemental de vecteurs parallèles de tableaux de tableaux de valeurs. Le résultat est une valeur «du même type». L'argument de cette opération est tel qu'à un processeur donné i d'une unité a le tableau de tableaux indique, pour chaque couple (b, j) de numéro d'unité et numéro de processeur, une valeur à transmettre au processeur j de l'unité b. Le résultat, quant à lui, est tel qu'à un processeur donné i d'une unité a, le tableau de tableaux indique, pour chaque couple (b, j), la valeur que le processeur j de l'unité b a transmise au processeur i de l'unité a.

La deuxième opération (règle 11.13) prend pour argument un vecteur départemental de vecteurs parallèles de tableaux de tableaux d'entiers augmentés de la valeur **nc**. Ce vecteur indique au processeur i d'une unité a à l'indice j dans le tableau et à l'indice b du tableau de tableaux, le numéro de la valeur demandée par le processeur j de l'unité b au processeur i de l'unité a. Les valeurs sont données par le vecteur départemental de vecteurs parallèles de fonctions, second argument de cette opération. Le résultat est le vecteur départemental de vecteurs parallèles de tableaux de tableaux des valeurs à transmettre.

La règle 11.11 permet l'enchaînement suivant :

- Un vecteur départemental de vecteurs parallèles de tableaux de tableaux indiquant quels numéros de valeurs doivent être demandés est créé à partir du premier argument du **getdep**. C'est la réduction des expressions t_i^a qui créera ces tableaux de tableaux.
- Ces numéros de valeurs sont transmis aux processeurs concernés : c'est l'opération **senddep** appliquée au vecteur départemental décrit à l'étape précédente qui réalise ceci.
- Les valeurs à renvoyer aux demandeurs sont préparées en faisant appel à l'opération **mkanwer**.
- Les valeurs sont transmises avec le second appel à **senddep**.
- Pour terminer, la fonction **mkf2** permet de transformer le vecteur départemental de vecteurs parallèles de tableaux de tableaux en vecteur départemental de vecteurs parallèles de fonctions, résultat du **getdep**.

11.4.3 Règles de contexte

On constate aisément qu'il n'est pas toujours possible de faire des réductions de tête. Il faut donc ajouter des règles de contextes. Ces derniers sont définis à la figure 11.7 où **PRIM** et **PRIM′** désigne les primitives parallèles et départementales (suivant le nombre d'arguments).

$$\mathbf{mkdep}\ v \xrightarrow[\Leftrightarrow]{\varepsilon} \langle\!\|(v\ 0),\ldots,(v\ (P-1))\|\!\rangle \text{ si } \mathcal{V}^d_\bullet(v) = \mathbf{true} \tag{11.7}$$

$$\mathbf{applydep}\ \begin{array}{l}\langle\!\|v_0,\ldots,v_{P-1}\|\!\rangle\\ \langle\!\|v'_0,\ldots,v'_{P-1}\|\!\rangle\end{array} \xrightarrow[\Leftrightarrow]{\varepsilon} \langle\!\|(v_0\ v'_0),\ldots,(v_{P-1}\ v'_{P-1})\|\!\rangle \tag{11.8}$$

$$\mathbf{projdep}\ \langle\!\|\ldots,\langle\ldots,v_i^a,\ldots\rangle,\ldots\|\!\rangle \xrightarrow[\Leftrightarrow]{\varepsilon} \mathbf{deldep}\ (\mathbf{getdep}\ \begin{array}{l}\langle\!\|\ldots,\langle\ldots,\lambda.\lambda.0,\ldots\rangle,\ldots\|\!\rangle\\ \langle\!\|\ldots,\langle\ldots,\lambda.v_i^a,\ldots\rangle,\ldots\|\!\rangle\end{array}) \tag{11.9}$$

$$\mathbf{deldep}\langle\!\|\langle v,\ldots,v\rangle,\ldots,\langle v,\ldots,v\rangle\|\!\rangle \xrightarrow[\Leftrightarrow]{\varepsilon} v \tag{11.10}$$

$$\begin{array}{l}\mathbf{getdep}\ \begin{array}{l}\langle\!\|\ldots,\langle\ldots,f_i^a\ldots\rangle,\ldots\|\!\rangle\\ \langle\!\|\ldots,\langle\ldots,g_i^a\ldots\rangle,\ldots\|\!\rangle\end{array} \xrightarrow[\Leftrightarrow]{\varepsilon}\\ \qquad t_i^a = \end{array} \begin{array}{l}(\mathbf{mkf2}(\mathbf{senddep}(\mathbf{mkanswer}\\ \quad(\mathbf{senddep}\langle\!\|\ldots,\langle\ldots,t_i^a\ldots\rangle,\ldots\|\!\rangle)\\ \quad\langle\!\|\ldots,\langle\ldots,g_i^a\ldots\rangle,\ldots\|\!\rangle)))\ \text{où}\\ \quad(\mathbf{init}(\lambda.\\ \quad(\mathbf{init}\ (f_i^a\ \overline{1})\ (\mathbf{dm\_bsp\_p}\ \overline{1})))\ P)[\bullet]\end{array} \tag{11.11}$$

$$\begin{array}{l}\mathbf{senddep}\ \langle\!\|\ldots,\langle\ldots,t_i^a,\ldots\rangle,\ldots\|\!\rangle\ \text{où}\\ t_i^a = \left[\begin{array}{c}[n_{(i,0)}^{(a,0)},\ldots,n_{(i,p_0-1)}^{(a,0)}],\\ \ldots,\\ [n_{(i,0)}^{(a,P-1)},\ldots,n_{(i,p_{P-1}-1)}^{(a,P-1)}]\end{array}\right]\end{array} \xrightarrow[\Leftrightarrow]{\varepsilon} \begin{array}{l}\langle\!\|\ldots,\langle\ldots,t_i'^a,\ldots\rangle,\ldots\|\!\rangle\ \text{où}\\ t_i'^a = \left[\begin{array}{c}[n_{(0,i)}^{(0,a)},\ldots,n_{(p_0-1,i)}^{(0,a)}],\\ \ldots,\\ [n_{(0,i)}^{(P-1,a)},\ldots,n_{(p_{P-1}-1,i)}^{(P-1,a)}]\end{array}\right]\end{array} \tag{11.12}$$

$$\begin{array}{l}\mathbf{mkanswer}\ \begin{array}{l}\langle\!\|\ldots,\langle\ldots,t_i^a\ldots\rangle,\ldots\|\!\rangle\\ \langle\!\|\ldots,\langle\ldots,g_i^a\ldots\rangle,\ldots\|\!\rangle\end{array} \xrightarrow[\Leftrightarrow]{\varepsilon}\\ \qquad t_i'^a = \end{array} \begin{array}{l}\langle\!\|\ldots,\langle\ldots,t_i'^a\ldots\rangle,\ldots\|\!\rangle\ \text{où}\\ \quad(\mathbf{init}(\lambda.(\mathbf{init}(\lambda.\\ \quad((\lambda.\mathbf{if}\ (\mathbf{isnc}\ \overline{1})\ \mathbf{then\ nc}\\ \quad\mathbf{else}\ (g_i^a\ \overline{1}))\ (\mathbf{dm\_bsp\_p}\ \overline{3}))\ P))\\ \quad(\mathbf{access}(\mathbf{access}\ t_i^a\ \overline{2})\ \overline{1})))[\bullet]\end{array} \tag{11.13}$$

$$\text{où } \mathbf{mkf2} = \left\{\begin{array}{l}\mathbf{applydep}(\mathbf{mkpdep}(\lambda.\\ \quad\mathbf{apply}(\mathbf{mkpar}(\lambda.\lambda.\lambda.\mathbf{if}\ ((0\leq\overline{2})\&(\overline{2}<P)\&(0\leq\overline{1})\&(\overline{1}<(\mathbf{dm\_bsp\_p}\ \overline{2}))\\ \qquad\qquad\mathbf{then}\ (\mathbf{access}(\mathbf{access}\ \overline{3}\ \overline{2})\ \overline{1})\ \mathbf{else\ nc}))))\end{array}\right.$$

Figure 11.6 — Réduction départementale

$$
\begin{array}{lll}
\Gamma_i^a ::= \Gamma_i^a\ e & \Gamma^a ::= \Gamma^a\ e & \Gamma_d ::= [] \\
\quad \mid v\ \Gamma_i^a & \quad \mid v\ \Gamma^a & \quad \mid \Gamma_d\ e \\
\quad \mid (\Gamma_i^a, e) & \quad \mid (\Gamma^a, e) & \quad \mid v\ \Gamma_d \\
\quad \mid (v, \Gamma_i^a) & \quad \mid (v, \Gamma^a) & \quad \mid (\Gamma_d, e) \\
\quad \mid \mathbf{let}\ x = \Gamma_i^a\ \mathbf{in}\ e & \quad \mid \mathbf{let}\ x = \Gamma^a\ \mathbf{in}\ e & \quad \mid (v, \Gamma_d) \\
\quad \mid \mathbf{if}\ \Gamma_i^a\ \mathbf{then}\ e\ \mathbf{else}\ e & \quad \mid \mathbf{if}\ \Gamma^a\ \mathbf{then}\ e\ \mathbf{else}\ e & \quad \mid \mathbf{let}\ x = \Gamma_d\ \mathbf{in}\ e \\
\quad \mid \mathbf{PRIM}\ \Gamma_i^a & \quad \mid \mathbf{PRIM}\ \Gamma^a & \quad \mid \mathbf{if}\ \Gamma_d\ \mathbf{then}\ e\ \mathbf{else}\ e \\
\quad \mid \mathbf{PRIM'}\ \Gamma_i^a\ e & \quad \mid \mathbf{PRIM'}\ \Gamma^a\ e & \quad \mid \mathbf{PRIM}\ \Gamma_d \\
\quad \mid \mathbf{PRIM'}\ v\ \Gamma_i^a & \quad \mid \mathbf{PRIM'}\ v\ \Gamma^a & \quad \mid \mathbf{PRIM'}\ \Gamma_d\ e \\
\quad \mid \langle\!|e, \ldots, \overbrace{\Delta_i}^{a}, e, \ldots, e|\!\rangle & \quad \mid \langle\!|e, \ldots, \overbrace{\Gamma_g}^{a}, e, \ldots, e|\!\rangle & \quad \mid \mathbf{PRIM'}\ v\ \Gamma_d
\end{array}
$$

$$
\begin{array}{lll}
\Delta_i ::= \Delta_i\ e & \Gamma_g ::= [] & \Gamma_l ::= [] \\
\quad \mid v\ \Delta_i & \quad \mid \Gamma_g\ e & \quad \mid \Gamma_l\ e \\
\quad \mid (\Delta_i, e) & \quad \mid v\ \Gamma_g & \quad \mid v\ \Gamma_l \\
\quad \mid (v, \Delta_i) & \quad \mid (\Gamma_g, e) & \quad \mid (\Gamma_l, e) \\
\quad \mid \mathbf{let}\ x = \Delta_i\ \mathbf{in}\ e & \quad \mid (v, \Gamma_g) & \quad \mid (v, \Gamma_l) \\
\quad \mid \mathbf{if}\ \Delta_i\ \mathbf{then}\ e\ \mathbf{else}\ e & \quad \mid \mathbf{let}\ x = \Gamma_g\ \mathbf{in}\ e & \quad \mid \mathbf{let}\ x = \Gamma_l\ \mathbf{in}\ e \\
\quad \mid \mathbf{PRIM}\ \Delta_i & \quad \mid \mathbf{if}\ \Gamma_g\ \mathbf{then}\ e\ \mathbf{else}\ e & \quad \mid \mathbf{if}\ \Gamma_l\ \mathbf{then}\ e\ \mathbf{else}\ e \\
\quad \mid \mathbf{PRIM'}\ \Delta_i\ e & \quad \mid \mathbf{PRIM}\ \Gamma_g & \quad \mid [\Gamma_l, e_1, \ldots, e_n] \\
\quad \mid \mathbf{PRIM'}\ v\ \Delta_i & \quad \mid \mathbf{PRIM'}\ \Gamma_g\ e & \quad \mid [v_0, \Gamma_l, \ldots, e_n] \\
\quad \mid \langle e, \ldots, \overbrace{\Gamma_l}^{i}, e, \ldots, e\rangle & \quad \mid \mathbf{PRIM'}\ v\ \Gamma_g & \quad \mid \ldots \\
 & & \quad \mid [v_0, v_1, \ldots, \Gamma_l]
\end{array}
$$

$$
\text{où} \begin{cases} \mathbf{PRIM} = \{\mathbf{proj}, \mathbf{put}, \mathbf{mkpar}, \mathbf{mkdep}, \mathbf{projdep}\} \\ \mathbf{PRIM'} = \{\mathbf{apply}, \mathbf{applydep}, \mathbf{getdep}\} \end{cases}
$$

Figure 11.7 — Contextes d'évaluation

Définition 43 (*Sémantique de DMML*).
La sémantique de notre mini-langage se définit avec les règles suivantes :

$$\frac{e \underset{i,a}{\rightharpoonup} e'}{\Gamma_i^a[e] \rightharpoonup \Gamma_i^a[e']} \qquad \frac{e \underset{\bowtie_a}{\rightharpoonup} e'}{\Gamma^a[e] \rightharpoonup \Gamma^a[e']} \qquad \frac{e \underset{\Leftrightarrow}{\rightharpoonup} e'}{\Gamma_d[e] \rightharpoonup \Gamma_d[e']}$$

Nous avons une règle pour réduire l'expression d'un processeur i d'un unité BSP a, puis une règle pour réduire l'expression d'une unité BSP a et enfin une règle pour réduire de manière départementale, notre expression. Nous avons alors les résultats suivants.

Lemme 86 (*Confluence forte*)

Soit e une expression. Si $e \rightharpoonup e_1$ et $e \rightharpoonup e_2$ alors il existe e_3 tel que $e_1 \rightharpoonup e_3$ et $e_2 \rightharpoonup e_3$.

Preuve. Voir en section 11.A de l'annexe de ce chapitre. ■

Théorème 19 (*Confluence*)

Soit e^p une expression tel que $e = \mathcal{T}_\bullet(e^p)$. Si $e \overset{*}{\rightharpoonup} v_1$ et $e \overset{*}{\rightharpoonup} v_2$ alors $v_1 = v_2$.

Preuve. La relation $\rightharpoonup$ est fortement confluente, donc, d'après le lemme 1, elle est confluente. ■

11.5 Exemples et expérimentations

Quelques exemples ont fait l'objet d'expériences en utilisant une première implantation de la bibliothèque DMMLlib.

11.5.1 Diffusion d'une valeur dans un méta-ordinateur

La fonction suivante est semblable au get de MSPML, à la différence que le second argument est un vecteur départemental de vecteurs parallèles de couples d'entiers, le premier désignant l'unité source et le second le processeur source :

```
(* get_one_all: α par dep→(int * int)par dep→α par dep *)
let get_one_all datas srcs =
 let send=parfun_all (fun v n→Some v) datas
 and n_srcs=parfun_all (fun(a,i)→let ap = (natmod a (dm_p())) in
                                  (ap,(natmod i (dm_bsp_p ap)))) srcs in
 let ask = parfun_all (fun (a,i) cluster pid→
    if (cluster=a)&&(pid=i) then Some 0 else None) n_srcs in
 parfun2_all(fun f (a,i)→(noSome (f a i)))(getdep ask send) n_srcs
```

avec natmod:int→int→int, fonction naturelle de *modulo* et

```
(* parfun2_all: (α →β →γ )→α par dep→β par dep→γ  par dep *)
let parfun2_all f v1 v2 = apply_all (parfun_all f v1) v2
```

qui fonctionne comme parfun_all mais avec des fonctions à deux arguments.

La diffusion d'une valeur dans un méta-ordinateur peut être faite de façon directe, en envoyant une valeur à partir d'un processeur donné d'une unité donnée vers tous les processeurs de toutes les unités. Cela peut être schématisé ainsi :

bcast_direct_all $a\ i$ 

En utilisant la fonction de communication get_one_all, on peut programmer cette diffusion directe :

```
(* bcast_direct_all: int→int→α par dep→α par dep *)
let bcast_direct_all rclus rpid vv =
 if rclus<0||rpid<0||rclus>=dm_p()||rpid>=(dm_bsp_p rclus)
 then raise Bcast
 else get_one_all vv (replicate_all(rclus,rpid))
```

Son coût DMM est (pour l'émission d'une valeur d'un processeur i de l'unité BSP a) :

$$\max \begin{cases} \max\limits_{b\in\{0\ldots P-1\}} (\mathcal{S}(v) \times p_b \times (g_b + G + g_a) + l_a + l_b + L) \text{ si } b \neq a \\ (\sum\limits_{b=0}^{P-1} (g_a \times p_b \times \mathcal{S}(v))) + l_a \end{cases}$$

où v est la valeur diffusée et $\mathcal{S}$ désigne la taille d'une valeur.

Il est possible également d'utiliser la diffusion en deux super-étapes BSP sur chaque unité. L'algorithme DMM est alors le suivant :

- Le processeur source répartit la valeur à diffuser sur les processeurs de chaque unité BSP. Un processeur donné n'aura pas la valeur en entier mais une unité dans son ensemble aura tous les morceaux constituant la valeur en entier ;
- Chaque unité utilise alors la seconde phase de la diffusion BSP avec échange total.

Le coût est alors :

$$\max \begin{cases} \max\limits_{b\in\{0\ldots P-1\}} ((\mathcal{S}(v) \times (g_b + G + g_a) + l_a + l_b + L + (p_b \times g_b \times \lceil \frac{\mathcal{S}(v)}{p_b} \rceil + l_b)) \text{ pour } b \neq a \\ g_a \times P \times \mathcal{S}(v) + l_a + (p_a \times g_a \times \lceil \frac{\mathcal{S}(v)}{p_a} \rceil + l_a) \end{cases}$$

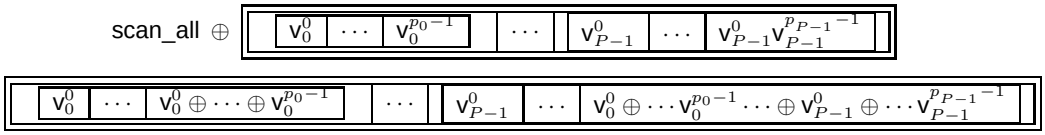
La figure 11.8 donne le code d'une telle fonction de diffusion départementale. Comme pour BSML, la fonction prend en paramètre une fonction permettant la découpe et le «recollage» de la valeur diffusée.

```
(* scatter_all: (α →int→β option)→int→int→α par dep→β par dep *)
let scatter_all partition rootclus rootpid v =
 if not (within_bounds_all2 rootclus rootpid)
  then raise Scatter
  else
  let mkmsg = mkdep (fun clus →mkpar (fun pid →if (pid=rootpid) && (clus=rootclus)
                                                then partition else fun v n →None))
  and mkask = mkdep (fun clus →mkpar (fun pid a b →if (a=rootclus) && (b=rootpid
                                                    then Some pid else None)) in
  let msg = getdep mkask (apply_all mkmsg v) in
   parfun_all noSome (apply2_all msg (replicate_all rootclus) (replicate_all rootpid))

(* bcast_totex_gen_all: (α →int→β option)→((int→β )→γ )→int→int→α par dep →γ par dep *)
let bcast_totex_gen_all partition paste rootclus rootpid vv =
  if not (within_bounds_all2 rootclus rootpid) then raise Bcast else
  let phase1 = scatter_all partition rootclus rootpid vv in
  let phase2 = applydep (mkdep (fun clus vpar →totex vpar)) phase1 in
   parfun_all paste phase2
```

Figure 11.8 — Code de l'algorithme binaire de la diffusion départementale

11.5.2 Calcul des préfixes départementaux

Le calcul des préfixes directs BSP est consistué d'une phase de communication ayant un coût égal à un échange total puis d'une réduction locale en chaque processeur. On peut évidemment le faire aussi pour DMM mais cela est coûteux. Le calcul des préfixes départementaux peut être schématisé comme suit :



Nous décrivons ici une solution alternative dans le cas où les objets manipulés sont des polynômes et où l'opération pour la réduction est la multiplication de polynômes. Nous avons les hypothèses suivantes :

- Les machines parallèles sont ordonnées par puissance croissante ;
- Les coefficients des polynômes $\sum_{i=0}^{m} c_i X^i$ sont stockés dans des tableaux.

On note $\mathcal{S}(n)$ la taille d'un polynôme de degré n. $\mathcal{S}(poly1 \times poly2) = \mathcal{S}(poly1) + \mathcal{S}(poly2)$ si on suppose que les tailles des coefficients ne dépendent pas de leurs valeurs.

Tout d'abord, chaque unité BSP a calcule les préfixes de ses polynômes. Le coût est alors :

$$A_a = (n \times p_a)^2 \times r_a + (p_a - 1) \times \mathcal{S}(n) \times g + l_a$$

où n est le degré maximal et r_a le temps nécessaire à une multiplication flottante.

Ensuite, un processeur de chaque unité BSP reçoit les polynômes de l'unité précédente. Les coûts de réception et d'envoi sont respectivement :

$$B_a = \sum_{\forall b<a} (l_b + (g_b + G + g_a) \times \mathcal{S}(p_a \times n) + l_a) + L$$

et

$$C_a = l_a + \mathcal{S}(p_a \times n) \times g_a \times (P - a)$$

Enfin, avec les polynômes reçus, l'unité BSP est capable de finir la réduction et le coût est :

$$D_a = (p_a - 1) \times (\sum_{\forall b<a} \mathcal{S}(p_b \times n)) \times g_a + l_b + (\sum_{\forall b<a} \mathcal{S}(p_b \times n)) \times r_a$$

```
(* scan_all: (α →α →α )→α par dep→α par dep *)
let scan_all op v =
 let bsp_scan= parfun_dep (scan_direct op) v in
 let other_bsp= get_list_all bsp_scan (mkdep (fun clus →mkpar (fun pid →
          if pid=0 then ff (clucs()) clus
          else [])) in
 let exchange_other=parfun_dep (bcast_direct 0) other_bsp in
 parfun2_all (fun deb li →List.fold_left op deb li) bsp_scan exchange_other
```

Figure 11.9 — Code de l'algorithme de calcul des préfixes départementaux

Le temps total d'exécution est alors :

$$\max_{a\in\{0\ldots P-1\}}(A_a + \max(B_a, C_a) + D_a)$$

La figure 11.9 donne le code d'un calcul des préfixes départementaux. Le code utilise la fonction get_list_all: α **par dep**→(int $*$ int) list **par dep**→α list **par dep** qui généralise la foncion get_one_all mais avec, cette fois-ci, des listes de requêtes de valeurs à d'autres processeurs. clucs: unit→int list retourne une liste contenant les «pids» des unités BSP.

11.5.3 Implantation

Une version de test de l'implantation des primitives DMML a été faite. Celle-ci repose sur l'utilisation de MPI au niveau des unités BSP et TCP/IP au niveau du réseau départemental. L'utilisation de TCP/IP oblige la création d'un réseau privé virtuel VPN (*virtual private network* en anglais) pour avoir des adresses IP compatibles à tous les nœuds du méta-ordinateur. L'implantation TCP/IP est très proche de celle de MSPML et le code de la BSMLlib a été directement utilisé pour les communications BSP.

11.5.4 Expériences

Des expériences préliminaires ont été effectuées sur un méta-ordinateur avec 6 noeuds Pentium IV 2.8 Ghz inter-connectés avec un réseau Gigabit Ethernet et 3 noeuds Celeron III interconnectés avec un réseau Fast Ethernet. Les deux machines parallèles sont reliées par un réseau Ethernet peu rapide. Chacun des programmes a été exécuté 100 fois de suite et la moyenne des exécutions a été prise pour les graphiques. Le système d'exploitation de la première grappe est une Mandrake clic 2.0 et celui de la seconde est une Mandrake clic 1.0. Le serveur de la première grappe est aussi un Pentium IV 2.8 Ghz, doté d'une carte réseau Gigabit Ethernet. Le serveur de la seconde grappe est un Pentium II avec une carte Fast Ethernet.

La figure 11.10 résume les résultats obtenus. La version MPI de la BSMLlib a été utilisée pour l'exécution de la version BSP du calcul des préfixes. Nous avons noté sur les graphiques «listes en BSML» quand plusieurs polynômes sont placés sur un processeur : le calcul des préfixes DMM utilise 9 polynômes, dans notre cas, qui sont distribués sur les 6 nœuds de la première grappe. Les algorithmes naïfs, des exemples précédemment présentés, sont moins bons. Nous avons constaté également que l'ajout de la seconde grappe permettait d'avoir une meilleure performance. Cela est dû au fait que le calcul de la multiplication de polynômes (complexité naïvement polynomiale dans notre programme de test) est plus lent que la transmission d'un polynôme (coût linéaire) *via* les réseaux. L'ajout du second cluster diminue donc le coût de calcul local, sans toutefois trop augmenter les temps de communication : il n'y a pas une synchronisation globale de tous les processeurs du méta-ordinateur, qui serait fort coûteuse.

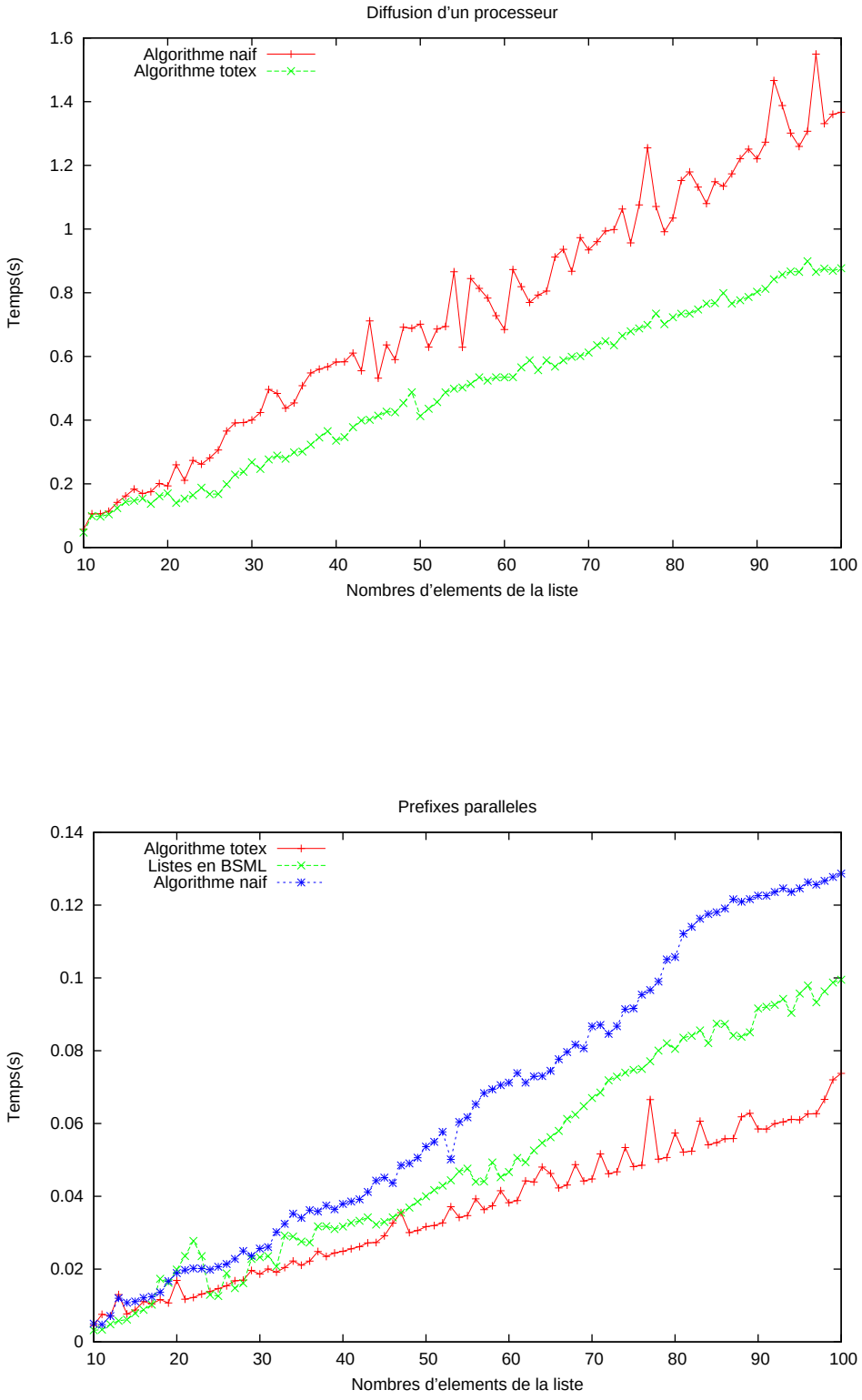


Figure 11.10 — Diffusion et calcul des préfixes départementaux

Vérification pour la création d'un vecteur parallèle :

$$
\begin{array}{rcl}
\mathcal{V}_s^g(\mathbf{mkpar}\ e) & = & \mathbf{false} \\
\mathcal{V}_s^g(\mathbf{proj}\ e) & = & \mathbf{false} \\
\mathcal{V}_s^g(\mathbf{put}\ e) & = & \mathbf{false} \\
\mathcal{V}_s^g(\mathbf{apply}\ e\ e) & = & \mathbf{false} \\
\mathcal{V}_s^g(\langle \ldots \rangle) & = & \mathbf{false} \\
\mathcal{V}_s^g(\mathbf{mkdep}\ e) & = & \mathbf{false} \\
\mathcal{V}_s^g(\mathbf{projdep}\ e) & = & \mathbf{false} \\
\mathcal{V}_s^g(\mathbf{getdep}\ e) & = & \mathbf{false} \\
\mathcal{V}_s^g(\mathbf{applydep}\ e\ e) & = & \mathbf{false} \\
\mathcal{V}_s^g(\langle\!\| \ldots \|\!\rangle) & = & \mathbf{false} \\
\mathcal{V}_s^g(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & = & \mathcal{V}_s^g(e_2) \wedge \mathcal{V}_s^g(e_2) \wedge \mathcal{V}_s^g(e_3) \\
\mathcal{V}_s^g(e_1\ e_2) & = & \mathcal{V}_s^g(e_1) \wedge \mathcal{V}_s^g(e_2)) \\
\mathcal{V}_s^g(e_1, e_2) & = & \mathcal{V}_s^g(e_1) \wedge \mathcal{V}_s^g(e_2)
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{V}_s^g(\mathbf{op}) & = & \mathbf{true} \\
\mathcal{V}_s^g(\mathbf{c}) & = & \mathbf{true} \\
\mathcal{V}_s^g(\lambda.e) & = & \mathcal{V}_{(x \circ s)}^g(e) \\
\mathcal{V}_s^g(\overline{(\lambda.e)[s']}) & = & \mathcal{V}_{(x \circ s')}^g(e) \\
\mathcal{V}_s^g(\mu.e) & = & \mathcal{V}_{(x \circ s)}^g(e) \\
\mathcal{V}_{(v \circ s)}^g(\overline{n+1}) & = & \mathcal{V}_s^g(\overline{n}) \\
\mathcal{V}_{(v \circ s)}^g(\overline{1}) & = & \mathcal{V}_s^g(v) \\
\mathcal{V}_s^g(x) & = & \mathbf{true}
\end{array}
$$

$$\mathcal{V}_s^g([e_0, \ldots, e_{p-1}]) = \bigwedge_{i=0}^{p-1} \mathcal{V}_s^g(e_i)$$

Vérification pour la création d'un vecteur départemental :

$$
\begin{array}{rcl}
\mathcal{V}_s^d(\mathbf{mkpar}\ e) & = & \mathcal{V}_s^d(e) \\
\mathcal{V}_s^d(\mathbf{proj}\ e) & = & \mathcal{V}_s^d(e) \\
\mathcal{V}_s^d(\mathbf{put}\ e) & = & \mathcal{V}_s^d(e) \\
\mathcal{V}_s^d(\mathbf{apply}\ e_1\ e_2) & = & \mathcal{V}_s^d(e_1) \wedge \mathcal{V}_s^d(e_2) \\
\mathcal{V}_s^d(\langle \ldots \rangle) & = & \mathbf{true} \\
\mathcal{V}_s^d(\mathbf{mkdep}\ e) & = & \mathbf{false} \\
\mathcal{V}_s^d(\mathbf{projdep}\ e) & = & \mathbf{false} \\
\mathcal{V}_s^d(\mathbf{getdep}\ e) & = & \mathbf{false} \\
\mathcal{V}_s^d(\mathbf{applydep}\ e\ e) & = & \mathbf{false} \\
\mathcal{V}_s^d(\langle\!\| \ldots \|\!\rangle) & = & \mathbf{false} \\
\mathcal{V}_s^d(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & = & \mathcal{V}_s^d(e_2) \wedge \mathcal{V}_s^d(e_2) \wedge \mathcal{V}_s^d(e_3) \\
\mathcal{V}_s^d(e_1\ e_2) & = & \mathcal{V}_s^d(e_1) \wedge \mathcal{V}_s^d(e_2)) \\
\mathcal{V}_s^d(e_1, e_2) & = & \mathcal{V}_s^d(e_1) \wedge \mathcal{V}_s^d(e_2)
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{V}_s^d(\mathbf{op}) & = & \mathbf{true} \\
\mathcal{V}_s^d(\mathbf{c}) & = & \mathbf{true} \\
\mathcal{V}_s^d(\lambda.e) & = & \mathcal{V}_{(x \circ s)}^d(e) \\
\mathcal{V}_s^d(\overline{(\lambda.e)[s']}) & = & \mathcal{V}_{(x \circ s')}^d(e) \\
\mathcal{V}_s^d(\mu.e) & = & \mathcal{V}_{(x \circ s)}^d(e) \\
\mathcal{V}_{(v \circ s)}^d(\overline{n+1}) & = & \mathcal{V}_s^d(\overline{n}) \\
\mathcal{V}_{(v \circ s)}^d(\overline{1}) & = & \mathcal{V}_s^d(v) \\
\mathcal{V}_s^d(x) & = & \mathbf{true}
\end{array}
$$

$$\mathcal{V}_s^d([e_0, \ldots, e_{p-1}]) = \bigwedge_{i=0}^{p-1} \mathcal{V}_s^d(e_i)$$

Figure 11.11 — Vérification d'une expression lors de la création d'un vecteur parallèle ou départemental

11.A Preuve de la confluence forte de $\rightharpoonup$

Lemme 87 (*Déterminisme des règles fonctionnelles*)

Soit e une expression.

1. *Si $e \xrightarrow[]{\varepsilon} e^1$ et $e \xrightarrow[]{\varepsilon} e^2$ alors $e^1 = e^2$;*
2. *Si $e \xrightarrow[\delta]{\varepsilon} e^1$ et $e \xrightarrow[\delta]{\varepsilon} e^2$ alors $e^1 = e^2$.*

Preuve. Par cas sur les règles. ■

Lemme 88 (*Déterminisme des règles globales et départementales*)

Soit e une expression.

1. *Si $e \xrightarrow[\bowtie_a]{\varepsilon} e^1$ et $e \xrightarrow[\bowtie_a]{\varepsilon} e^2$ alors $e^1 = e^2$;*
2. *Si $e \xrightarrow[\diamond]{\varepsilon} e^1$ et $e \xrightarrow[\diamond]{\varepsilon} e^2$ alors $e^1 = e^2$.*

Preuve. Par cas sur les règles. ■

Lemme 89 (*Déterminisme des règles*)

Soit e une expression.

1. *Si $e \xrightarrow[i,a]{} e^1$ et $e \xrightarrow[i,a]{} e^2$ alors $e^1 = e^2$;*
2. *Si $e \xrightarrow[\bowtie_a]{} e^1$ et $e \xrightarrow[\bowtie_a]{} e^2$ alors $e^1 = e^2$;*
3. *Si $e \xrightarrow[\diamond]{} e^1$ et $e \xrightarrow[\diamond]{} e^2$ alors $e^1 = e^2$.*

Preuve. Par application du lemme 87 pour 1). Par application des lemmes 88 et 87 pour 2) et 3). ■

Lemme 90 (*Déterminisme des contextes départementaux*)

Soit e une expression. Si $e = \Gamma_{d^1}[e^1]$ et $e = \Gamma_{d^2}[e^2]$ alors $\Gamma_{d^1} = \Gamma_{d^2}$ et $e^1 = e^2$.

Lemme 91 (*Confluence des contextes globaux*)

Soit e une expression.

1. *$\forall a$, si $e = \Gamma^a[e^1]$ et $e = \Gamma^a[e^2]$ alors $e^1 = e^2$;*
2. *$\forall a, b\ a \neq b$; si $e = \Gamma^a[e^1]$ et $e = \Gamma^b[e^2]$ alors $e = \Gamma_d[\langle\!| \ldots, \Gamma_{g^1}[e^1], \ldots, \Gamma_{g^2}[e^2], \ldots |\!\rangle]$;*
3. *Si $e = \Gamma_{g^1}[e^1]$ et $e = \Gamma_{g^2}[e^2]$ alors $\Gamma_{g^1} = \Gamma_{g^2}$ et $e^1 = e^2$.*

Lemme 92 (*Confluence des contextes locaux*)

Soit e une expression.

1. *$\forall i, a$; si $e = \Gamma_i^a[e^1]$ et $e = \Gamma_i^a[e^2]$ alors $e^1 = e^2$;*
2. *$\forall i, j, a\ i \neq j$; si $e = \Gamma_i^a[e^1]$ et $e = \Gamma_j^a[e^2]$ alors $e = \Gamma_d[\langle\!| \cdots, \Gamma_g[\langle \ldots, \Gamma_{l^1}[e^1], \ldots, \Gamma_{l^2}[e^2], \ldots\rangle], \cdots |\!\rangle]$;*
3. *$\forall i, j, a, b\ a \neq b$; si $e = \Gamma_i^a[e^1]$ et $e = \Gamma_j^b[e^2]$ alors $e = \Gamma_d[\langle\!| \cdots, \Gamma_{g^1}[\langle \ldots, \Gamma_{l^1}[e^1], \ldots\rangle], \ldots, \Gamma_{g^2}[\langle \ldots, \Gamma_{l^2}[e^2], \ldots\rangle], \ldots |\!\rangle]$;*
4. *Si $e = \Gamma_{l^1}[e^1]$ et $e = \Gamma_{l^2}[e^2]$ alors $\Gamma_{l^1} = \Gamma_{l^2}$ et $e^1 = e^2$.*

Lemme 93 (*Unicité du choix d'un type de contexte*)

Soit e une expression.

1. *Si $e = \Gamma_d[e']$ alors $\nexists \Gamma^a$ tel que $e = \Gamma^a[e'']$;*
2. *Si $e = \Gamma^a[e']$ alors $\nexists \Gamma^a_i$ tel que $e = \Gamma^a_i[e'']$.*

Preuve . Par construction de nos contextes. En effet, ceux-ci ne permettent l'application d'une règle :

- Départementale que dans une unique sous-expression ;
- Globale que dans un unique vecteur départemental mais possiblement dans deux composantes différentes ;
- Locale que dans un unique vecteur départemental mais possiblement dans deux composantes différentes de ce vecteur et dans des vecteurs parallèles différents (et donc dans des composantes possiblement différentes de ces vecteurs).

Nous avons aussi, par construction, que les contextes Γ_l et Γ_g sont déterministes. ■

Lemme 94 (*Confluence forte*)

Soit e une expression. Si $e \rightharpoonup e_1$ et $e \rightharpoonup e_2$ alors il existe e_3 tel que $e_1 \rightharpoonup e_3$ et $e_2 \rightharpoonup e_3$.

Preuve . Par le lemme 93, nous avons trois types de réductions bien distincts.

Si $\rightharpoonup$ est une réduction départemental, alors par le lemme 90 il n'existe qu'un contexte départemental et par le lemme 89, la réduction est déterministe.

Si $\rightharpoonup$ est une réduction globale, alors nous avons $e = \Gamma^a[e^a]$ et $e = \Gamma^b[e^b]$. Nous avons alors deux cas :

1. Si $a = b$ alors par le lemme 91.1 il n'existe qu'un contexte et par le lemme 89, la réduction est déterministe ;
2. Si $a \neq b$ alors par le lemme 91.2, il n'existe qu'un seul vecteur départemental à réduire. Dans ce cas, nous avons $e = \Gamma_d[\langle\!| \ldots, \Gamma_{g^a}[e^a], \ldots, \Gamma_{g^b}[e^b], \ldots |\!\rangle]$. Par les lemmes 91.3 et 89, les deux réductions sont déterministes. Donc par *b*, $e \rightharpoonup \Gamma_d[\langle\!| \ldots, \Gamma_{g^a}[e^a], \ldots, \Gamma_{g^b}[e'^b], \ldots |\!\rangle]$ et par *a*,
$e \rightharpoonup \Gamma_d[\langle\!| \ldots, \Gamma_{g^a}[e'^a], \ldots, \Gamma_{g^b}[e^b], \ldots |\!\rangle]$. Il est alors facile de constater que l'ordre dans lequel ces deux réductions sont appliquées, indiffère le résultat final : les réductions interviennent dans deux composantes différentes d'un même vecteur. Les règles peuvent donc s'entrelacer. Nous pouvons donc avoir $\Gamma_d[\langle\!| \ldots, \Gamma_{g^a}[e'^a], \ldots, \Gamma_{g^b}[e'^b], \ldots |\!\rangle]$.

Si $\rightharpoonup$ est une réduction locale, alors nous avons $e = \Gamma^a_i[e^a_i]$ et $e = \Gamma^b_j[e^b_j]$. Nous avons alors trois cas :

1. Si $a = b$ et $i = j$ alors par le lemme 92.1 il n'existe qu'un contexte et par le lemme 89, la réduction est déterministe ;
2. Si $a = b$ et $i \neq j$ alors par le lemme 92.2, il n'existe qu'un seul vecteur parallèle à réduire (provenant d'un unique vecteur départemental). Comme précédemment, les règles s'entrelacent et sont chacune déterministe ;
3. Si $a \neq b$ alors par le lemme 92.3, il n'existe qu'un seul vecteur départemental à réduire. Comme précédemment, les règles s'entrelacent et sont chacune déterministes.

Les réductions des trois types sont donc fortement confluentes. ■

12 Travaux connexes

Sommaire

NOUS présentons des travaux connexes, d'une part effectués sur la programmation parallèle de «haut-niveau» avec des modèles de parallélisme, sur la preuve de programmes ou les structures de données parallèles, et d'autre part, des modèles et des langages pour le méta-calcul.

12.1 Modèles et langages parallèles de «haut niveau»

12.1.1 Modèles de parallélisme structuré

Le modèle PRAM. Le premier modèle théorique de machines parallèles à avoir été réellement utilisé est le modèle PRAM, Parallel Random Access Machine, introduit dans [110]. Ce modèle comprend p processeurs et une mémoire partagée. Chaque processeur dispose, en outre, d'une mémoire locale de petite taille. Les processeurs opèrent de façon synchrone, où, à chaque étape d'un algorithme PRAM, certains processeurs sont actifs et exécutent la même opération qui est soit un calcul en mémoire locale, soit une lecture de la mémoire partagée, soit une écriture dans cette même mémoire. Les autres processeurs sont inactifs. De plus, selon le mode d'accès (concurrent ou exclusif durant la lecture ou l'écriture) à la mémoire partagée, une machine PRAM est classifiée comme suit :

- Le modèle EREW PRAM (Exclusif Read, Exlusif Write) permet à un seul processeur au plus, durant une étape de l'algorithme, d'avoir accès à une cellule de la mémoire partagée afin d'y lire ou d'y écrire. Tout comportement contraire n'est pas acceptéé ;
- Le modèle CREW PRAM (Concurrent Read, Exclusif Write) : la lecture d'une cellule de mémoire partagée par plusieurs processeurs est permise. Toutefois, une écriture simultanée n'est pas tolérée ;
- Le modèle CRCW PRAM (Concurrent Read, Concurrent Write) introduit des conflits lorsque plusieurs processeurs écrivent dans la même cellule de la mémoire partagée. De nombreux modèles ont été proposés pour permettent de résoudre les conflits ainsi créés comme par exemple ;
- Le modèle QRQW PRAM (Queue Read, Queue Write) [118] qui n'autorise, pour chaque cellule mémoire, qu'à un seul processeur la lecture ou l'écriture. Toutes les autres demandes de lecture ou d'écriture des autres processeurs sont stockées dans une «file» pour être exécutées dans une étape ultérieure [56]. Le temps d'accès à une cellule de la mémoire partagée, à un instant donné, est alors proportionnel au nombre d'accès concurrents (en lecture ou en écriture) à cette même cellule.

Les modèles PRAM permettent une description simple et universelle des algorithmes parallèles. Malheureusement, ils ignorent les coûts de communications inter-processeurs, rendant les temps d'exécution des algorithmes difficilement prévisibles sur un grand nombre d'architectures parallèles comme les grappes de PC. Les modèles BSP et MPM, eux, sont portables sur un plus grand nombre d'architectures.

Le modèle LogP. Le modèle LogP, introduit dans [79], est un modèle asynchrone pour machines à la mémoire distribuée. Contrairement au modèle BSP, la synchronisation y est abandonnée dans le but de refléter, et ceci de manière plus précise, les caractéristiques des machines réelles, par l'utilisation de quatre paramètres :

1. **L** est la latence (*latency* en anglais) liée à la communication d'un message contenant un octet (ou plutôt, un petit nombre d'octets) d'un processeur à un autre,

2. **o** est le surcoût (*overhead* en anglais) dans une communication liée à l'envoi ou à la réception d'un message,

3. **g** est le temps minimum écoulé (*gap* dans la littérature anglo-saxonne) entre deux envois ou réceptions consécutifs de messages sur un processeur,

4. **P** est le nombre de couples processeurs/mémoires.

Dans ce modèle, lors d'une communication point à point, l'envoi d'un message contenant un paquet de données nécessite un temps $2 \times \mathbf{o} + \mathbf{L}$, alors que l'envoi de n paquets de données ne nécessite que $2 \times \mathbf{o} + \mathbf{L} + (n-1) \times \max(\mathbf{o}, \mathbf{g})$. Les paramètres **L**, **g** et **o** sont exprimés comme étant des multiples du cycle du processeur (durée d'une exécution de calcul élémentaire). Le modèle suppose que le réseau d'interconnexions a une capacité finie, de manière à ce que, au plus $\lceil \mathbf{L}/\mathbf{g} \rceil$ messages puissent transiter *via* le réseau à un moment donné. Le modèle LogP tente ainsi de résoudre le problème de saturation du réseau d'interconnexions, en supposant que seuls des messages élémentaires (de petite taille) peuvent être échangés entre les processeurs.

Des extensions de ce modèle comme LogGP [4] ou LogP étendu [281], ont été proposées pour supporter des communications mettant en œuvre des messages longs. Notons qu'il est possible de simuler le modèle BSP dans le modèle LogP et *vice versa* [32]. Néanmoins, les auteurs de cet article notent que le modèle BSP est plus simple d'utilisation et permet une meilleure portabilité. De plus, d'un point vue asymptotique, les deux modèles sont équivalents. Le modèle MPM permet de s'affranchir des synchronisations globales BSP. Il est donc un modèle asynchrone mais plus structuré que LogP et possède les mêmes paramètres que ceux du modèle BSP (seules leurs valeurs changent).

Le modèle CGM. Le modèle CGM (Coarse Grained Multicomputer) a été introduit dans [94]. Une machine CGM est constituée de p processeurs $P_1, \ldots, P_p$ où chaque processeur dispose d'une mémoire locale de taille $O(s/p)$ (certains travaux, comme dans [174], supposent une mémoire locale de taille $\Omega(n/p)$ où n est la taille des données à traiter). La taille de la mémoire locale de chaque processeur est supérieure à celle donnée par le modèle PRAM, ce qui qualifie ce modèle de «gros grain». Ce modèle est en fait une version simplifiée du modèle BSP qui s'affranchit des paramètres l et g, ainsi que de l'étape de synchronisation.

Un algorithme CGM est une suite d'étapes alternant d'une part du calcul local et d'autre part une ronde de communication globale. Dans cette ronde, une seule h-relation, avec $h = O(n/p)$, est effectuée, c'est-à-dire chaque processeur envoie ou reçoit au plus $O(n/p)$ données. Le temps d'exécution d'un algorithme CGM est donc la somme des traitements locaux avec celui des rondes de communication. Notons que dans le modèle CGM, une étape de traitement combinant du calcul local suivi d'une ronde de communication est équivalent à une super-étape du modèle BSP.

Le modèle CGM a été utilisé pour définir de nombreux algorithmes [90] comme par exemple des algorithmes sur les graphs [64]. Néanmoins, un algorithme CGM est facilement simulable comme un algorithme BSP [90], ce qui fait que, généralement, les coûts de l'algorithme sont donnés dans les deux modèles. De plus, le modèle CGM est dans certains cas, comme dans [18, 19, 20], trop rigide car le rééquilibrage des charges de certains algorithmes ou la distribution des données (l'implantation des structures de données parallèles présentée dans ce manuscrit en est aussi un exemple), peuvent nécessiter des h-relations de volumes h imprévisibles.

12.1.2 Modèles dérivés de BSP

Mémoires externes. Le modèle des disques parallèles (*Parallel Disk Model*, PDM), introduit dans [269], est employé pour modéliser une hiérarchie à deux niveaux de mémoires, se composant de D disques parallèles et avec $p \geq 1$ processeurs inter-connectés par une mémoire partagée ou un réseau. La mesure de coût de PDM est le nombre d'opérations d'E/S exigées par un algorithme où les éléments peuvent être transférés sur la mémoire interne aux disques en une seule opération d'E/S. Le modèle PDM capture les calculs et les coûts d'E/S, mais il a été conçu pour un type spécifique de réseau de transmission où les opérations de communication prennent une simple unité de temps, comparable à une instruction de l'unité centrale de traitement. Le modèle BSP (et les modèles en dérivant), capturent mieux les communications et leurs coûts pour une classe plus générale de réseaux mais ne capturent pas les coûts des E/S.

Un algorithme parallèle *out-of-core* pour le calcul de l'inversion de grandes matrices a été présenté dans [61]. L'algorithme emploie seulement la diffusion comme primitive de communication. Celle-ci a un coût proche de la diffusion directe BSP. Les coûts des E/S sont similaires aux nôtres (voir au chapitre 9), c'est-à-dire linéaires (et non constants) à la taille de la donnée écrite (ou lue) sur les disques parallèles.

Dans l'article [81], les auteurs se sont concentrés sur l'optimisation de quelques algorithmes parallèles de tri. Ces algorithmes utilisent les mémoires externes ainsi que de multiples couches de mémoire de la machine parallèle, comme par exemple les caches (ou tampons) de la mémoire vive ou des disques. Les auteurs se sont servis d'un langage de bas niveau et le grand nombre de paramètres de ce modèle implique une complexité algorithmique difficilement analysable. Aussi, dans l'article [92], les auteurs ont implanté des opérations d'E/S pour mesurer leur modèle mais toujours à l'aide d'un langage de bas niveau. De la même manière, une bibliothèque d'E/S pour une extention à mémoire externe d'un modèle proche de BSP est décrite dans [132].

À notre connaissance, notre bibliothèque est la première pour une extention du modèle BSP avec des mémoires externes, appelés EM^2-BSP (comprenant disques locaux ou disques partagés), et pour un langage fonctionnel parallèle doté d'une sémantique formelle et d'un modèle formel de coût.

Modèles pour les algorithmes «diviser-pour-régner» et la sous-synchronisation. Une façon d'implanter des algorithmes «diviser-pour-régner» BSP, dans le cadre d'un langage à objets, est présentée dans [258]. Il n'y a, pour l'instant, ni de sémantique formelle ni d'implantation. Le principe des opérations des objets est très proche de la superposition parallèle. Le même auteur propose dans l'article [197] une nouvelle extension du modèle BSP qui permet d'aborder facilement les algorithmes «diviser-pour-régner», en ajoutant un niveau supplémentaire au modèle BSP et de nouveaux paramètres.

L'article [280] présente un langage à patrons qui offre des patrons «diviser-pour-régner». Toutefois, le modèle de coût n'est plus le modèle BSP mais le modèle D-BSP [89] qui permet la synchronisation de sous-réseaux. Des arguments pour rejeter une telle possibilité sont présentés dans [133].

Dans la bibliothèque BSPlib [147], la synchronisation de sous-réseaux n'est pas autorisée comme expliqué dans les articles [133, 242]. La bibliothèque PUB [40] offre des caractéristiques supplémentaires par rapport à la proposition de standard BSPlib notamment, la synchronisation de sous-réseaux (suivant le modèle BSP* [23]). Un exemple d'application implantée en utilisant ces possibilités est donné dans [41].

Modèles pour la désynchronisation. Il existe de nombreux travaux sur la désynchronisation des barrières BSP qui se basent sur différentes méthodes de comptage de messages [6, 103, 166]. À notre connaissance, la seule extension qui a donné lieu à une implantation disponible est celle que l'on trouve dans la bibliothèque PUB. La synchronisation bsp_oblsync prend en argument le nombre de messages devant être reçus par le processeur à une super-étape donnée. Lorsque ce nombre de messages a été reçu, le processeur passe à la super-étape d'après sans prendre part à une synchronisation globale. Le modèle de coût MPM est plus simple et MSPML permet en plus une confluence des résultats qui n'est pas garantie avec la primitive bsp_oblsync de la PUB.

12.1.3 Langages et bibliothèques de «haut niveau»

On trouve de nombreux langages fonctionnels pour la programmation parallèle. Les présenter tous serait trop long. Nous avons donc choisi de ne présenter que les plus intéressants pour notre propos.

Eden. Le langage Eden [28, 52, 167] étend Haskell[1] avec des primitives de créations dynamiques de processus qui échangent des données sur des canaux de communications implicites vus par le programmeur comme des «listes paresseuses». La création et l'exécution de processus se fait avec :

```
process :: (Trans a, Trans b) => (a -> b) -> Process a b
( # ) :: (Trans a, Trans b) => Process a b -> a -> b
```

tel que `(process (\x -> e1)) # e2` permet la création d'un processus qui exécutera `e1` avec `e2` comme paramètre. Le processus parent est responsable du transfert des données vers ce nouveau processus. Un canal de communication implicite a donc été construit entre ces deux processus. Le programmeur peut aussi créer ses propres canaux avec les fonctions suivantes :

```
class NFData a => Trans a where (...)

type ChanName a = ...
new :: Trans a => (ChanName a -> a -> b) -> b
parfill :: Trans a => ChanName a -> a -> b -> b
```

où `Trans` est une sous-classe de `NFData` (*Normal Form Data*), l'ensemble des données transmissibles. L'implantation (avec ces fonctionnalités) d'un patron «diviser-pour-régner» a été présentée dans [29]. Aucun modèle de parallélisme n'est utilisé dans ce langage. La prévision des performances y est donc très difficile, à la différence de BSML qui repose sur le modèle BSP.

Gph. GpH [137, 260] est aussi une extension du langage Haskell. Gph utilise un parallélisme explicite avec deux constructeurs «Par» et «Seq». Toutes les expressions données au constructeur «Seq» seront évaluées séquentiellement, tandis qu'un processus est exécuté pour évaluer l'expression donnée au constructeur «Par». La programmation parallèle y est donc donnée dans sa version la plus simple. Mais aucune prédiction des performances n'y est possible car la création des processus dépend des heuristiques du système. De même, aucun modèle de parallélisme n'est utilisé.

Dans ces extentions parallèles de Haskell (Gph et Eden), la sûreté et la confluence des opérations d'E/S sont assurées par l'utilisation de *monads* [270] et de mémoires externes locales. L'utilisation de disques partagés n'est pas spécifiée. Ces langages parallèles autorisent également aux processeurs d'échanger des canaux d'écriture ou de lecture et donnent ainsi la possibilité à un processeur d'écrire (ou de lire) sur les fichiers d'un autre processeur (si celui-ci n'est pas déjà ouvert en écriture). Cela augmente l'expressivité du langage mais diminue fortement la prévision des performances car il devient difficile de donner un coût au opérations d'E/S : elles peuvent être locales ou nécessiter le réseau. Trop de communications sont ainsi cachées.

OcamlP3L. OcamlP3L [83] est une extension de OCaml avec des patrons algorithmiques sur des flux de données (*stream* en anglais). Les patrons employés sont ceux de P3L [15]. Par exemple, nous avons les patrons suivants :

farm: α stream$\rightarrow$($\alpha \rightarrow \beta$)$\rightarrow\beta$ stream
farm f $= f(x_n); \cdots ; f(x_0)$ en utilisant n processus (travailleurs)

pipe: α stream$\rightarrow$($\alpha \rightarrow \beta$)$\rightarrow\beta$ stream (noté ||)
f1 || f2 || ... || fm $= f_m(\cdots f_2(f1(x_0))); \cdots ; f_m(\cdots f_2(f1(x_0)))$

pour un flux $x_n; \cdots ; x_0$. La figure 12.1 illustre l'exécution parallèle de ces patrons.

L'implantation des patrons a été faite avec les fonctionnalités TCP/IP de OCaml, ce qui ne la rend pas très portable, à la différence de BSML qui ne repose pas spécialement sur une bibliothèque de communication pré-définie. OcamlP3L étant basé sur les patrons P3L, ceux-ci peuvent être eux-mêmes modélisés avec LogP [79]. OcamlP3L a été utilisé pour programmer des applications numériques de simulations géologiques [67]. OcamlP3L ne proposent pas de patron de structures de données parallèles.

NESL. NESL [38] est le véritable ancêtre des langages fonctionnels *data-parallel* et est basé sur le modèle PRAM. NESL permet l'application de fonctions sur des tableaux vues comme des flux de données. Par exemple, `factorial(i) : i in [3,1,7]` applique la fonction de factorielle sur 3, puis 1 et enfin

[1] `http://www.haskell.org`

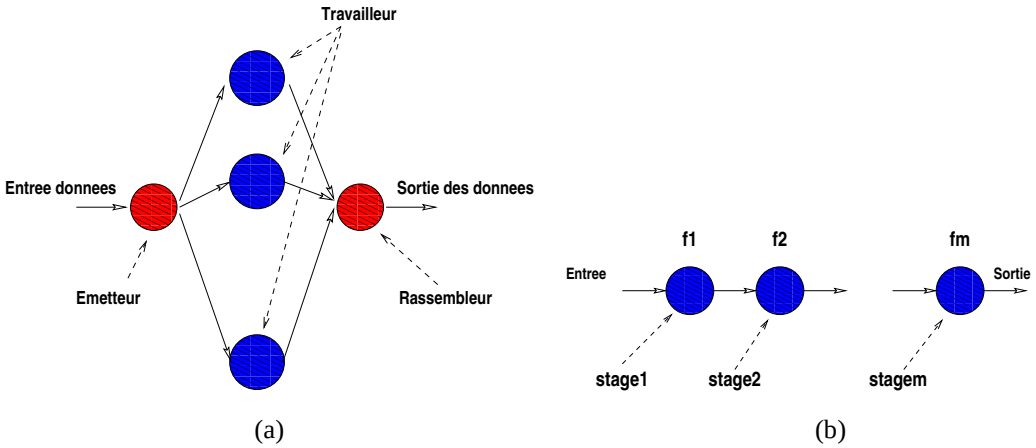


Figure 12.1 — Deux exemples de patron de OcamlP3L

7. De plus, NESL fournit un certain nombre de fonctions manipulant directement ces tableaux, comme par exemple `take(tab,i)` qui permet d'accéder à la $i$ème composante du tableau `take`.

La puissance de ce langage se fonde sur la possibilité d'emboîter les fonctions et les tableaux (ce qui peut être vu comme une possibilité «diviser-pour-régner»). Les communications entre les processeurs sont donc toutes implicites. NESL propose aussi des constructions syntaxiques de reconnaissance de formes (*pattern-matching* en anglais) afin de simplifier les programmmes. Malheureusement, les prédictions des performances sont très difficiles et le polymorphisme y est limité. L'implantation de NESL n'est pas portable et n'est disponible que sur certaines architectures parrallèles (BSML lui est portable).

SAC. SAC [125] est un langage avec une syntaxe à la C pour la manipulation de tableaux à multiples dimensions. Le langage fournit un certain nombre d'opérations de «haut niveau» sur ces tableaux qui ont un type et une représentation prédéfinis. Par exemple, `dim : basic_type[]->int` retourne la dimension du tableau donné en paramètre ou `take : int[]->basic_type[]->basic_type[]` est tel que `take(vec,A)` retourne une sous-matrice de `A` dont les indices sont définis dans `vec`.

La compilation d'un tel langage recourt un grand nombre d'optimisations (et d'heuristiques) sur la manipulation de ces tableaux. Le compilateur est capable d'optimiser le code si la machine cible est à mémoire partagée ou non. Le langage a été conçu pour les applications numériques qui manipulent un grand nombre de données dans des tableaux. Pour des raisons d'efficacité, le polymorphisme des tableaux est limité : seules des données de base (entiers, booléens) sont acceptées. Le modèle PRAM peut être utilisé pour ce langage. Les opérations d'E/S en SAC ont été écrites pour les architectures à disques partagés et sans une sémantique formelle. Le programmeur est responsable des problèmes de non-déterminisme de ce genre d'opérations. En BSML, les primitives d'E/S sont confluentes (cf. chapitre 9) et basées sur un modèle de coût pour la prévision des performances.

Caml-Flight. Caml-Flight [108] peut être considéré comme l'ancêtre de BSML. C'est une extension déterministe de OCaml pour la programmation SPMD. L'évaluation des programmes est basée sur un mécanisme de *vagues*. La primitive `sync` est utilisée pour indiquer quels processeurs peuvent échanger des messages durant l'utilisation de la primitive `get`. Appliquée à une expression e et un numéro de processeur i, cette dernière primitive permet au processeur qui l'exécute de demander l'évaluation distante d'une expression e sur un processeur i. Aucun modèle de performances n'est utilisé. Le mécanisme des vagues est plus complexe que celui des communications de MSPML et il n'y a pas de sémantique purement fonctionnelle (dans le d'un sens système de réécriture d'ordre supérieur) de Caml-Flight [109, 262].

12.1.4 Sémantiques et certification de programmes BSP

Sémantiques d'opérations d'entrées/sorties. L'article [122] présente une sémantique dynamique d'un mini-langage fonctionnel avec une stratégie d'appel par valeur, doté d'opérations d'E/S, ne travaillant pas sur des fichiers, mais plutôt sur une entrée unique (entrée standard) et une sortie unique (standard).

L'article [101] propose un langage fonctionnel (pur) et concurrent (à la Gph) pour raisonner sur les E/S. Il montre que sous certaines conditions, l'évaluation des programmes dans ce langage est confluente. Mais les fichiers sont seulement des fichiers locaux et aucun modèle de coût n'est donné.

Preuves de programmes parallèles. L'article [113] présente la création (avec une vérification automatique) d'une bibliothèque de programmes parallèles (pour des machines parallèles dites hypercubes) en utilisant le «prouveur automatique» de Boyer-Moore, Nqthm[2], basé sur la logique du premier ordre. Mais cette catégorie de programme n'est pas portable et ne bénéficie pas d'une sémantique confluente. De plus, l'analyse des programmes et des coûts est très complexe et il est difficile d'écrire des formules logiques de haut niveau pour spécifier les programmes et leurs fonctionnements. Il est à noter qu'il existe un logiciel dédié à la preuve formelle de programmes d'algèbre linéaire, où l'utilisateur peut définir et prouver le bon placement des données [130].

Des travaux [65, 160, 246] ont été réalisés sur la possibilité d'étendre la logique de Hoare (avec les pré- et post-conditions) pour des programmes impératifs BSP. Ces programmes sont écrits dans un langage algorithmique avec une exécution SPMD. Ce langage est enrichi de primitives de communications (get et put, soit sur des variables partagées, soit par l'échange des valeurs de ces variables, c'est-à-dire écrire la valeur de telle variable dans telle variable de tel autre processeur), d'un opérateur de synchronisation globale rendant les valeurs échangées accessibles, et parfois, de boucles parallèles. Malheureusement, aucun de ces travaux n'a pu fournir un logiciel pour vérifier, à l'aide d'un assistant de preuves, les programmes ainsi annotés. De plus, aucun programme réel n'a pu être vérifié formellement, faute d'un tel logiciel. Cette défaillance vient essentiellement d'un manque de formalisme pour générer les obligations de preuves.

Preuves de programmes BSP. La première sémantique formelle du modèle BSP était une logique axiomatique «à la Hoare» avec un parallélisme explicite pour un petit langage impératif avec une mémoire partagée [160]. Les programmes étaient donnés avec leurs spécifications et chaque pas du calcul devait être justifié par un raisonnement mathématique. Les spécifications étaient basées en aplatissant chaque processus en une séquence de traces qui peuvent être combinées pour déterminer le fonctionnement de chaque composant. Malheureusement, aucun programme complet n'a été certifié avec ces règles algébriques et aucune implantation de ce modèle de certification n'a été faite. Mais ce travail suggère la possibilité de prouver des algorithmes parallèles BSP avec des règles formelles.

Dans d'autres approches [247, 248], le raisonnement est fait en utilisant une séquence de transformations (globales) d'états parallèles. Le raisonnement est rendu plus facile car les opérations parallèles peuvent être décrites par ces transformations. Le même article montre l'utilisation de ces règles sur un algorithme BSP du plus court chemin d'un graphe (extension aisée de l'algorithme de Floyd). L'article [240] présente une extension du «Refinement calculus» (un calcul de «pré-conditions faibles») pour permettre la dérivation de programmes BSP impératifs. Toutes ces approches sont basées sur des langages impératifs avec des preuves faites «à la main», sans un assistant de preuves.

Notre approche a les avantages suivants. Elle est basée sur un langage purement fonctionnel (sans effets de bord) avec parallélisme explicite (les vecteurs parallèles) doté d'une sémantique de «haut niveau». Le raisonnement en est ainsi grandement simplifié, ce qui permet de faire des preuves sur les programmes (le modèle BSP, en séparant calculs et communications, rend impossible les inter-blocages). En utilisant un assistant de preuves comme **Coq**, nos preuves sont partiellement automatisées ce qui n'est le cas dans aucune des approches précédentes. De plus, après les preuves, nous pouvons générer des programmes certifiés (grâce à l'extraction) portables (grâce au modèle BSP) à partir des formules logiques et bénéficier des développements ultérieurs faits en **Coq** (bibliothèque standard et contributions des autres utilisateurs).

Méthodes sémantiques pour les algorithmes «diviser-pour-régner». Une méthodologie (et un langage dans [144]) permettant une décomposition parallèle des programmes «diviser-pour-régner» est présentée dans [143]. Elle utilise un modèle de calcul géométrique basé sur la transformation de coordonnées où le temps (l'ordonnanceur), et l'espace (les processeurs), sont explicites. Cette technique peut être appliquée à une classe de fonctions récursives «diviser-pour-régner» et donne des programmes fonctionnels à patrons (implantés avec MPI). Dans cet environnement à patrons, la prédiction des performances y est difficile et les optimisations algorithmiques (comme dans BSP) encore plus difficiles.

[2] `http://www.cs.utexas.edu/users/boyer/ftp/nqthm/`

Une formulation générale *data-parallel*, pour une classe de problèmes «diviser-pour-régner», a été évaluée dans l'article [14]. Une combinaison de différentes techniques a été employée pour réorganiser les flots de données entre les processeurs, réduisant ainsi les communications et permettant une meilleure utilisation du réseau et de la mémoire. Mais ces techniques n'ont été définies que pour le langage de bas niveau qu'est *High Performance Fortran*.

Dans l'article [145], les auteurs proposent une approche distinguant trois niveaux d'abstractions et leurs instanciations :

1. Un petit langage, étendant ML et définissant les parties (statiques) parallèles d'un programme. Le langage est évalué avec un transformateur de programmes (basé sur le modèle des multi-stages de [252]) qu'est MetaOCaml[3]. Ce langage ne doit être utilisé que par les programmeurs spécialistes en parallèlisme afin de fournir des bibliothèques parallèles aux non spécialistes ;

2. Une implantation d'un patron parallèle «diviser-pour-régner» démontre comment la méta-programmation apportée par MetaOCaml permet de générer un ensemble approprié de communications pour un processus particulier défini à l'aide d'une spécification abstraite ;

3. Une application utilisant ce patron a été imaginée comme écrite par un non-spécialiste. Le but étant de tester si un non-spécialiste pouvait écrire du code parallèle sans avoir à considérer tous les détails de la programmetion parallèle.

Le code fourni par MetaOCaml a été expérimenté sur un cluster. Malheureusement, la prédictions des coûts y est encore impossible. De plus, du code natif (permettant de bonnes performances comparé à C+MPI) ne peut être généré (sur certaines architectures) car MetaOCaml a été employé.

Sémantiques et preuves de correction des machines abstraites. Les premières preuves de compilateurs ont été effectuées avec des sémantiques à grands pas. Ces preuves présentent l'intérêt d'être relativement simples (elles se font par induction sur l'arbre d'évaluation des termes). En revanche, elles ne permettent pas de prouver la correction de tous les programmes mais seulement des programmes qui terminent. Il est alors impossible de prouver, par exemple, qu'un simulateur 3D (parallèle comme dans [5]) ne s'arrête pas brusquement et retourne un message d'erreur. De plus, les preuves sont très dépendantes du compilateur. Ce type de preuves a été utilisé pour prouver la correction de machines abstraites comme la CAM, la SECD [173], ou bien la FAM (*Functional Abstract Machine*) [58].

Dans [84, 138], les auteurs proposent une autre approche pour la preuve d'un compilateur. Ils font remarquer qu'il est possible de dériver un compilateur d'un interpréteur du langage source et réciproquement. Ils prouvent ensuite la correction du compilateur ainsi obtenu en montrant que l'interprétation du langage source rend le même résultat que le code-objet. Ils limitent cependant la preuve aux programmes qui se terminent et leurs techniques pour automatiser les dérivations [53] sont très «manuelles» et correspondent en fait à une preuve du lemme de progression de [216]. Une étude de cas complète avec différentes interprétations et donc différentes version de la SECD a été proposée dans [85]. Ces interprétations ont souvent pour but des optimisations et des extensions pour la normalisation forte (réduire sous un λ et donc calculer dans les fermetures afin d'optimiser leurs créations).

Une autre approche est celle de la preuve de correction du compilateur dans un assistant de preuves. Un premier travail a été fait dans ce sens dans l'article [50] où l'auteur a refait la preuve de correction de la CAM en **Coq** (la preuve par induction). Dans [127], l'auteur a prouvé (entièrement en **Coq**) la correction de la ZAM (dont une version étendue pour la normalisation forte). La preuve est basée essentiellement sur les techniques de [216] (quelques raffinements ont été introduits pour leur cas particulier).

Les premières machines abstraites BSP pour BSML se trouvent dans les articles [207, 208] qui étendent respectivement la CAM et la SECD. Dans les deux cas, les machines sont étendues avec des instructions BSP pour les communications. Des implantations et des expériences de ces machines ont été réalisées. Mais les programmes sont naturellement lents (machine abstraite) et aucune preuve de correction n'est disponible. De plus, la compilation des primitives parallèles dans [207] est dépendante de p (le nombre de processeurs) ce qui rend les programmes non-portables. C'est en contradiction avec l'esprit des machines abstraites (et BSP) où les programmes sont portables tant qu'une machine abstraite a été implantée sur la machine réelle cible. Aussi, dans les deux machines, les instructions BSP sont non triviales et donc difficilement implantables contrairement à notre schéma de compilation qui est, lui, sûr et facilement implantable

[3] `http://www.cs.rice.edu/~taha/MetaOCaml`

(on profite alors des optimisations de code du compilateur OCaml). Cette compilation a été faite pour notre extension modulaire de la BSMLlib décrite dans le chapitre 5.

12.1.5 Structures de données

Il existe un grand nombre de bibliothèques, pour différents langages, qui sont des implantations efficaces de structures de données. La plus connue est sans doute LEDA [206] pour C++. Malheureusement, toutes ces bibliothèques ont été conçues pour de la programmation séquentielle. Utilisant des langages de «bas niveau», sans un système de modules aussi perfectionné que celui de OCaml, il paraît difficile d'en fournir des versions parallèles sans être obligé de tout reprogrammer.

Structures de données parallèles. Depuis le tout début des machines parallèles, des travaux considérables ont été effectués pour la réalisation de structures de données distribuées et de multiples bibliothèques ont ainsi été obtenues pour des architectures particulières [10]. Nous pouvons aussi citer [205] (resp. [115]) qui présente des algorithmes PRAM (resp. BSP) pour l'implantation efficace de dictionnaires parallèles (resp. de files de priorités). Dans ces deux travaux, des expériences ont été effectuées mais aucune application scientifique n'a été codée en utilisant ces implantations faites en C. Dans [115], l'implantation a été effectuée en C+BSPlib, ce qui a pour conséquence une maintenance plus difficile due à l'utilisation d'un langage de bas niveau. Notons aussi les travaux de [96] qui présentent un algorithme CGM pour un problème scientifique proche du calcul des n voisins les plus proches.

Dans l'article [278], est présentée une bibliothèque fournissant l'implantation parallèle de différentes structures de données comme les ensembles ou les files de priorités. Cette bibliothèque a été testée sur des applications de calcul symbolique comme une version parallèle de la méthode de Knuth-Bendix (permettant de connaître la normalisation ou non d'un système de réécriture). Mais aucun modèle d'exécution parallèle n'a été utilisé (implantation bas niveau en C++ avec des primitives d'envois/réceptions de données) ce qui rend les coûts des opérations sur ces structures imprévisibles. De la même manière, l'implantation en C+MPI de deux structures de données permettant de modéliser un système lagrangien de particules est décrite dans [155]. Plus récemment, nous trouvons dans [256], des implantations avec C+MPI de structures de données parallèles. Dans ce cas, seules les opérations collectives de MPI sont utilisées permettant une éventuelle analyse BSP de ces implantations.

Structures de données parallèles et langages parallèles de «haut niveau». OcamlP3L ne fournit pas l'implantation de structures de données. Sans une distribution explicite des données, il paraît difficile d'implanter des structures de données parallèles, permettant ainsi d'étendre l'ensemble des squelettes avec des «squelettes de structures de données». Dans les langages Eden et Gph, des travaux d'implantation de squelettes algorithmiques ont été effectués [29, 136], et comme il est possible de définir une distribution parallèle des données, l'implantation parallèle de structure de données, comme il a été proposé dans ce chapitre, est envisageable. Malgré tout, il revient au système (basé sur des heuristiques) de distribuer les processus sur les processeurs, rendant la prédiction des performances très difficile. Dans notre cas, cette prédiction y est possible en utilisant le modèle BSP.

Dans l'article [135], les auteurs décrivent une première implantation des ensembles parallèles en BSML. Malheureusement, ceux-ci n'ont pas utilisé les modules de OCaml pour factoriser leur code et ont utilisé leur propre implantation séquentielle des ensembles. De plus, aucune application n'a été codée avec ces ensembles et la plupart des opérations binaires transformaient préalablement un des ensembles parallèles en un ensemble séquentiel (avec un échange total), rendant ces opérations parallèles moins efficaces (sur notre grappe de test) que leurs versions séquentielles.

12.2 Modèles et langages pour le méta-calcul

12.2.1 Sémantiques de la désynchronisation des barrières BSP

Il a été montré dans [17] que NESL est plus efficace lorsque la taille des vecteurs est constante. Même si ce n'est pas le cas, la plupart des opérations de NESL peut être implantée en MSPML. En particulier les listes emboîtées peuvent être implantées comme dans [152]. De ce point de vue, MSPML peut paraître de plus bas niveau que NESL. Mais MSPML offre les fonctions d'ordre supérieur, ce qui n'est pas le cas dans NESL.

L'article [227] décrit le mécanisme des *horloges structurelles* qui permet l'exécution de programmes data-parallèles écrits dans un petit langage impératif SPMD. La difficulté dans ce cadre est que le nombre de phases de communication peut être différent sur chaque processeur car il y existe un opérateur de composition parallèle. Nous aurons également besoin d'un mécanisme plus complexe si nous ajoutons une juxtaposition parallèle à MSPML. La sémantique de haut-niveau d'une telle primitive serait identique à celle présentée au chapitre 7.

12.2.2 Modèles pour le méta-calcul

Le modèle HiHCoHP. L'article [57] présente le modèle HiHCoHP (*Hierarchical Hypercluster of Heterogeneous Processors*) qui caractérise une hyper-grappe (une grappe de grappes $\cdots$ de grappes de processeurs) *via* une série de paramètres idéalisant les communications entres les processeurs des grappes. Ce modèle a été utilisé dans [57] pour comparer différents algorithmes de diffusions de messages.

Une hyper-grappe est constituée de N nœuds hétérogènes, $P_1, \ldots, P_N$ chacun composé d'un processeur et d'une mémoire locale. Les nœuds sont interconnectés par une hiérarchie de réseaux comportant l niveaux. Les niveaux sont ordonnés en taille décroissante, avec des latences décroissantes, mais des capacités de transmission croissantes.

Chaque P_i est connecté à un réseau d'un niveau k dans la hiérarchie. Cette hiérarchie peut donc se voir comme un arbre de profondeur maximale l, où les branches provenant d'un même nœud de l'arbre forment un réseau et dont les feuilles sont les processeurs.

Le modèle s'intéresse aux communications d'un processeur P_a à un processeur P_b d'un même niveau k. Dans ce modèle, les messages, envoyés par paquets, devront passer par k niveaux différents : un message de P_a à P_b passe par le réseau de niveau 1, puis par le réseau supérieur 2, et ainsi de suite (il remonte vers la racine de l'arbre), jusqu'à un réseau qui est un ancêtre commun de P_a et P_b. Le message redescend alors vers le processeur P_b en passant par les différents réseaux intermédiaires.

Pour chaque niveau k et processeur a, les paramètres de transmission sont :

- σ_a^k, le coût pour l'initialisation de la communication ;
- π_a^k, le coût pour émettre le paquet (sur le réseau) ;
- κ^k, la capacité maximale d'un réseau ;
- λ^k, la latence d'un réseau ;
- β^k, le temps pour transmettre un paquet.

avec $\forall a$, $\sigma_a^k < \sigma_a^{(k+1)}$, $\pi_a^k < \pi_a^{(k+1)}$, $\lambda^k < \lambda^{(k+1)}$ et $\beta^k > \beta^{(k+1)}$. Le coût total pour envoyer p paquets de P_a à P_b est donc :

$$(\sigma_a^k + \sigma_b^k) + (\pi_a^k + \pi_b^k) \times p + \lambda^k + \Delta(p)$$

où $\Delta(p) = (p-1)/\beta^k$ dans un réseau en pipe-line, $\Delta(p) = (p-1) \times \lambda^k$ dans un «réseau sauvegarde-puis-oublie» (*save-and-forgot network* en anglais) et $p \times \beta^k < \kappa^k$.

Le nombre important de paramètres rend la conception des algorithmes très complexe. De plus, aucun modèle d'exécution n'est donné pour structurer les algorithmes. Il n'y a pas de langage de «haut niveau» basé sur ce modèle, ni de sémantique formelle.

Les mêmes problèmes se retrouvent dans [235]. L'article [31] propose un modèle hiérarchique qui manque aussi de structuration et pour lequel, par exemple, les interblocages peuvent apparaître.

Le modèle BSP^2. L'article [197] présente un modèle pour la programmation de grappes de machines parallèles homogènes à mémoire partagée (réseau de bi-PC par exemple). Une machine BSP^2 consiste en un nombre uniforme d'unités BSP connectées par un réseau. Le modèle étend donc BSP avec les paramètres suivants :

- P est le nombre d'unités BSP ;
- L est la latence du réseau entre les unités BSP ;
- G est le temps mis pour envoyer un octet entre 2 unités.

L'exécution est une séquence d'hyper-étapes terminées par des barrières de synchronisation de tous les processeurs de toutes les unités BSP. Le coût d'une hyper-étape est donc :

$$C_l + n_c \times G + L$$

avec C_l le temps de calcul BSP maximal d'une unité et n_c le nombre maximal d'octets échangés entre les différentes unités.

Est aussi présentée l'analyse de coût de différents algorithmes, comme par exemple, un tri parallèle. Les auteurs de ce modèle notent que les performances des programmes BSP^2 seront équivalentes à ceux de BSP, à cause du coût de la synchronisation globale d'une hyper-étape et du manque de flexibilité du modèle.

Notons que l'on peut facilement simuler les programmes BSP avec des programmes BSP^2, en utilisant les paramètres BSP^2 suivants en tant que paramètres BSP : $(p, g, l)_{bsp}$=$(p \times P, l + L, g + p \times G)$.

Le modèle DMM est donc proche de ce modèle mais remplace le second niveau BSP par un second niveau MPM et supprime l'hypothèse d'homogénéité des unités parallèles et permet d'éviter le coût de synchronisation globale de toutes les machines, ce qui rend le modèle plus «souple».

Le modèle HBSP^k. Le modèle k-hétérogène BSP de [273] généralise le modèle BSP^2. Dans ce modèle, le méta-ordinateur est aussi un arbre dont chaque nœud est un serveur du réseau sous-jacent (formé par les branches). Les paramètres sont les suivants :

- m_i est le nombre de machines de la machine HBSP^i. Ces machines sont étiquetées $M_{i,0}, M_{i,1}, \ldots, M_{i,m_i}$ pour tout niveau i avec $1 \leq i \leq k$;
- $m_{i,j}$ est le nombre de machines sous-jacentes de la machine $M_{i,j}$;
- g est le temps minimal utilisé par une machine pour injecter un octet sur les réseaux ;
- $r_{i,j}$ est la vitesse du processeur ;
- $L_{i,j}$ est le coût pour effectuer une barrière de synchronisation de toutes les machines du sous-arbre de $M_{i,j}$.

Dans ce modèle, l'exécution procède également en hyper-étapes (appelées super^i-étapes), dont le temps d'exécution est :

$$w_i + g \times h + L_{i,j}$$

où w_i est le temps de calcul maximal d'un processeur du sous-arbre de $M_{i,j}$ et h le plus grand nombre d'octets envoyés ou reçus par un processeur. Les serveurs participent aussi aux super^i-étapes, et sont souvent considérés comme des nœuds à part entière. Dans [273] sont décrits quelques algorithmes pour des opérations de communication collective ainsi que des tests de performances d'une implantation en C+PVM [114]. Ce travail est la continuité de [211, 272] où les auteurs présentent des versions hétérogènes de BSP et CGM.

Le modèle P-LoGP. L'article [165] introduit une extension à deux niveaux du modèle Log-GP [4], appelée «LoGP paramétré». Le modèle d'exécution est celui de LogP, mais avec les paramètres suivants :

- P est le nombre de processeurs ;
- L est la latence du réseau ;
- os et or sont respectivement les surcoûts pour l'envoi et la réception d'un message ;
- g est le temps pour transmettre un octet d'un nœud à un autre.

Chaque réseau est donc caractérisé par P, L, os, or, g. Par exemple, L_1 est la latence du premier cluster et L_w celle du réseau d'inter-connexion entre les clusters. Le temps pour envoyer/recevoir n paquets de taille m est $(os + or) \times n + n \times m \times g + L$.

Ce modèle a été utilisé pour l'analyse et l'optimisation d'une implantation de différentes fonctions de communication collective de MPI, appelée MagPIe. Une méthode pour l'obtention automatique des paramètres est aussi présentée, ainsi que des résultats d'expériences sur une suite de grappes de PC. La suite de tests, permettant automatiquement de déterminer les paramètres du modèle, a été implantée et testée, afin de vérifier les optimisations qui ont été apportées à la bibliothèque de méta-calcul. Notons que seul ce modèle, dans ceux présentés ici, possède une telle suite de tests.

[222] présente aussi une étude pratique sur l'utilisation d'un modèle de programmation à deux niveaux pour un certain nombre d'algorithmes de communication. Un gain important est obtenu par rapport à des algorithmes conçus pour un modèle non hiérarchique. Mais, dans les deux cas, aucun langage de «haut niveau» n'a été proposé.

Dynamic BSP et GRID-BSP. L'article [265] a proposé un modèle BSP pour le GRID. Un ordinateur BSP-GRID est une machine BSP avec une mémoire partagée par les processeurs. Cependant, à la différence du modèle BSP, les données locales sont toutes effacées à la fin d'une super-étape. Pour les applications nécessitant beaucoup de données, des processeurs virtuels sont introduits et en cas de panne(s) d'un processeur, un protocole de recouvrement des processus virtuelles a aussi été proposé : une barrière supplémentaire est introduite afin de migrer les processus depuis un processeur désigné comme maître. Les paramètres de ce modèle sont ceux de BSP où à chaque super-étape le nombre de processus varie. Le problème majeur d'un tel modèle est une distribution statique des processus qui est effectuée à chaque super-étape : la panne d'un seul processus entraîne une barrière de synchronisation de l'ensemble des processus. De plus aucune hétérogénéité des réseaux comme des processeur n'a été introduite dans ce modèle.

L'article [198] propose une amélioration de ce modèle. Le modèle consiste en un processeur maître, des processeurs esclaves et un serveur de données (mémoire virtuelle partagée). Un programme procède une suite de super-étapes GRID qui fonctionnent comme suit : le processeur maître crée des tâches qui sont ensuite exécutées par les processeurs esclaves. Si l'un d'eux tombe en panne, les tâches qui lui avaient été attribuées sont réparties sur les autres processeurs. Quand un nouveau processeur esclave se connecte au maître, celui-ci attend la fin de la super-étape pour avoir des tâches à effectuer. Quand une tâche est terminée, le processeur esclave attend une nouvelle tâche du maître. Lorsque toutes les tâches ont été effectuées, le programme passe à la super-étape suivante.

Dans les deux cas, aucune implantation n'a été réalisée. La faisabilité de tel modèle reste donc hypothétique et la prédiction des performances BSP des programmes dans de tels environnements reste à prouver.

12.2.3 Langages pour le méta-calcul

Il n'existe pas (à notre connaissance) de langages de «haut niveau» vraiment dédiés au méta-calcul (ou sur une grille de calcul, *grid-computing* en anglais). On trouve plutôt des implantations de langages parallèles où les programmes sont déployés sur la grille.

Par exemple, Lithium[4] est une implantation Java de squelettes P3L. L'article [95] présente une librairie de composants afin de faciliter la programmation Java, MPI ou Corba dans en environnement grille[5]. [59] est une bibliothèque Java pour la mobilité et le calcul sur la grille[6].

Un autre exemple est Grid-Gph [279] qui permet d'exécuter des programmes Gph sur une grille. Dans ce dernier, les performances ne sont pas celles espérées : les heuristiques employées pour le calcul parallèle avec un réseau homogène, ne sont plus viables pour un méta-ordinateur. Les systèmes ont tendance à concentrer tous les calculs sur une seule machine parallèle... La conception de nouvelles heuristiques reste un sujet d'étude des auteurs. En DMML, le placement des processus est explicite, ce qui peut compliquer la programmation. Mais notre langage est basé sur un modèle de coût qui permettra la prévision des performances des programmes.

[4]page web à `http://www.di.unipi.it~/~marcod/Lithium/`
[5]page web à `http://www.irisa.fr/paris/General/grid.htm`
[6]page web à `http://www-sop.inria.fr/oasis/ProActive/`

13 Conclusion et perspectives

DANS le cadre du projet CARAML, notre travail visait trois objectifs principaux. D'une part, augmenter la confiance que les utilisateurs peuvent avoir dans le langage de programmation *Bulk Synchronous Parallel ML* (BSML) et fournir un cadre pour la certification de programmes BSML. D'autre part, étendre le langage BSML et sa bibliothèque standard pour en faciliter l'utilisation : offrir une interaction sûre entre les entrées/sorties de OCaml et les primitives parallèles BSML par la conception de primitives d'entrées/sorties parallèles pour BSML ; offrir la possibilité d'une multiprogrammation BSML avec l'ajout de primitives de compositions parallèles ; développer des modules applicatifs en utilisant ces primitives. Enfin, étendre notre approche à la programmation de méta-ordinateurs par la conception d'un langage à deux niveaux, l'un étant identique à BSML l'autre étant un dérivé de BSML sans barrière de synchronisation. Nous revenons sur les différents résultats tant pratiques que théoriques des trois axes de recherche qui ont été menés. Nous décrirons ensuite les différentes perspectives envisagées.

13.1 Apports de cette thèse

13.1.1 Partie 1 : Sémantiques, sûreté, implantation et certification

BSML présente deux modèles. Le premier est un modèle de programmation dans lequel un programme BSML est un programme ML habituel sur une structure de données générique appelée vecteur parallèle qui peut être implantée aussi bien par une structure de données ML classique, que de façon répartie. Le deuxième est un modèle d'exécution dans lequel un programme BSML est vu comme p copies d'un même programme ML qui s'échangent des valeurs par passage de messages.

Sémantique et sûreté. Pour pouvoir assurer la sûreté de la programmation BSML, il faut garantir l'équivalence entre ces deux modèles et permettre à l'utilisateur de certifier ses propres programmes BSML (à l'aide de la sémantique du modèle de programmation). Nous avons alors élaboré deux sémantiques : une sémantique naturelle (chapitre 3) qui formalise le modèle de programmation et une machine abstraite parallèle (chapitre 4) qui formalise le modèle d'exécution.

Afin de garantir l'équivalence entre les modèles, il nous fallait prouver celle des sémantiques. Toutefois, trop différentes dans leur nature, ceci ne pouvait être fait directement.

Ainsi, notre sémantique naturelle assure une compréhension aisée des primitives parallèles. Mais elle n'est pas suffisante pour raisonner sur les programmes qui ne terminent pas et pour l'entrelacement des calculs (propre au parallélisme). La sémantique à «petits pas», elle, met en évidence l'interprétation parallèle des réductions et des coûts. Mais cette sémantique laisse apparaître des synchronisations inutiles. Par contre, la sémantique distribuée permet de s'affranchir de ces synchronisations. Mais c'est au prix d'une interprétation plus difficile des programmes. Ces trois sémantiques sont confluentes et équivalentes (confère chapitre 3).

Enfin, nous avons prouvé la correction de la machine abstraite par rapport à cette sémantique distribuée (voir au chapitre 4), établissant ainsi l'équivalence du modèle d'exécution et du modèle de programmation. Cette preuve n'aurait pu être faite sans les substitutions explicites de notre langage de base.

Implantation. La description d'une machine abstraite et la définition des sémantiques ont aussi permis une modularisation de la BSMLlib, améliorant grandement sa portabilité. En effet, ce travail nous a permis d'extraire l'ensemble minimal d'instructions (trois instructions : **PID**, **SEND** et **NPROC**) qui est néces-

saire pour passer d'un ML à l'implantation de BSML. L'implantation de la BSMLlib peut donc dépendre d'un module unique où seules sont implantées ces trois instructions[1].

Trois implantations de ces instructions ont été réalisées : la première avec MPI, la seconde avec la PUB et la dernière avec les fonctionnalités TCP/IP de OCaml. Des tests de performances ont été réalisés dans le chapitre 5 pour différentes fonctions de la bibliothèque standard. Des comparaisons avec les prédictions BSP ont aussi été réalisées.

Certification. Une fois la confiance en BSML établie, il s'agit de permettre aux utilisateurs de certifier leurs programmes. Le chapitre 6 est un exemple d'adéquation de l'assistant de preuves **Coq** à la spécification et à la réalisation de programmes parallèles. Nous avons formalisé les primitives parallèles du langage BSML et, en utilisant cette axiomatisation, nous avons pu développer des programmes BSML certifiés qui sont fréquemment utilisés et qui constituent un sous-ensemble important de la bibliothèque BSMLlib. Nous avons donc pu prouver formellement des propriétés de programmes. L'assistant de preuves **Coq** semble donc être propice aux preuves de correction de programmes parallèles.

Par conséquent, nous fournissons un environnement de programmation sûr et portable pour le développement de programmes BSP *purement fonctionnels*, dont les performances sont prévisibles. La figure 13.1 résume ce propos.

13.1.2 Partie 2 : Opérations de multi-traitement, structures de données parallèles et entrées/sorties

Dans une utilisation pratique de BSML, le programmeur peut être amené à implanter des d'algorithmes «diviser-pour-régner» (qui divise spatialement et récursivement les resources de la machine parallèle) que l'on trouve dans la littérature. Or ce travail peut être extrêmement difficile avec les seules primitives présentées dans la première partie puisqu'il faut, en fait, préalablement transformer l'algorithme pour en faire disparaître ses aspects spatialement récursifs. Nous avons donc étendu BSML par une primitive de composition parallèle simplifiant ainsi cette programmation.

Aussi, idéalement, le programmeur devrait pouvoir développer rapidement des programmes BSML à partir d'une bibliothèque standard d'algorithmes dans lequel le parallélisme est transparent pour l'utilisateur. La programmation BSML s'apparenterait alors à la programmation avec des patrons algorithmiques, avec la différence (importante) qui est qu'en cas d'absence d'un patron adéquat (pour l'écriture de l'algorithme) le programmeur BSML peut se tourner vers une utilisation directe des primitives parallèles.

Enfin, comme nous l'avons souligné, le travail précédent s'applique uniquement aux programmes BSML purement fonctionnels. Toutefois la plupart des programmes qui sont développés en Ocaml utilisent les traits impératifs, ne serait-ce que pour les entrées/sorties. Celles-ci permettent une manipulation efficace d'un grand nombre de données. Or, Les travaux effectués précédemment ne garantissent en aucun cas la sûreté de l'exécution de programmes BSML employant ces traits.

Ce deuxième axe nous a donc conduit à définir des extensions de BSML et ce, avec des sémantiques opérationnelles et des modèles de coûts. Les modèles permettent l'analyse et la dicibilité des temps d'exécution des programmes, tandis que les sémantiques autorisent le raisonnement sur les résultats.

Deux extensions sont présentées dans ce tapuscrit (deux autres sont présentées respectivement dans les articles [C4] et [C6]) : l'une sur sur le multi-traitement data-parallèle (cf. chapitre 7) et l'autre sur des entrées/sorties parallèles BSP (voir au chapitre 9). Le chapitre 8 expose l'implantation de structures de données parallèles en BSML. La partie gauche du schéma de la figure 13.2 résume ce propos.

Multi-traitement. La première extension est dédiée à la composition parallèle de programmes. La superposition parallèle est une nouvelle primitive de BSML. Elle permet d'exprimer naturellement des algorithmes «diviser-pour-régner», et cela sans casser le modèle d'exécution BSP (super-étapes et barrières de synchronisations globales). Deux expressions (programmes) sont évaluées par deux *super-threads* qui partagent leurs phases de communication et synchronisation.

Comparée aux autres primitives proposées dans [188, 190], la superposition n'a pas les désavantages de ses prédécesseurs : les deux expressions de la superposition peuvent ne pas nécessiter le même nombre de super-étapes et elle peut être vue comme une primitive purement fonctionnelle (donc utile pour la preuve des programmes BSML). Le chapitre 7 présente une sémantique formelle du modèle d'exécution de la

[1] Notons qu'il serait aussi possible de concevoir un compilateur BSML vers du code OCaml SPMD appelant ces trois instructions.

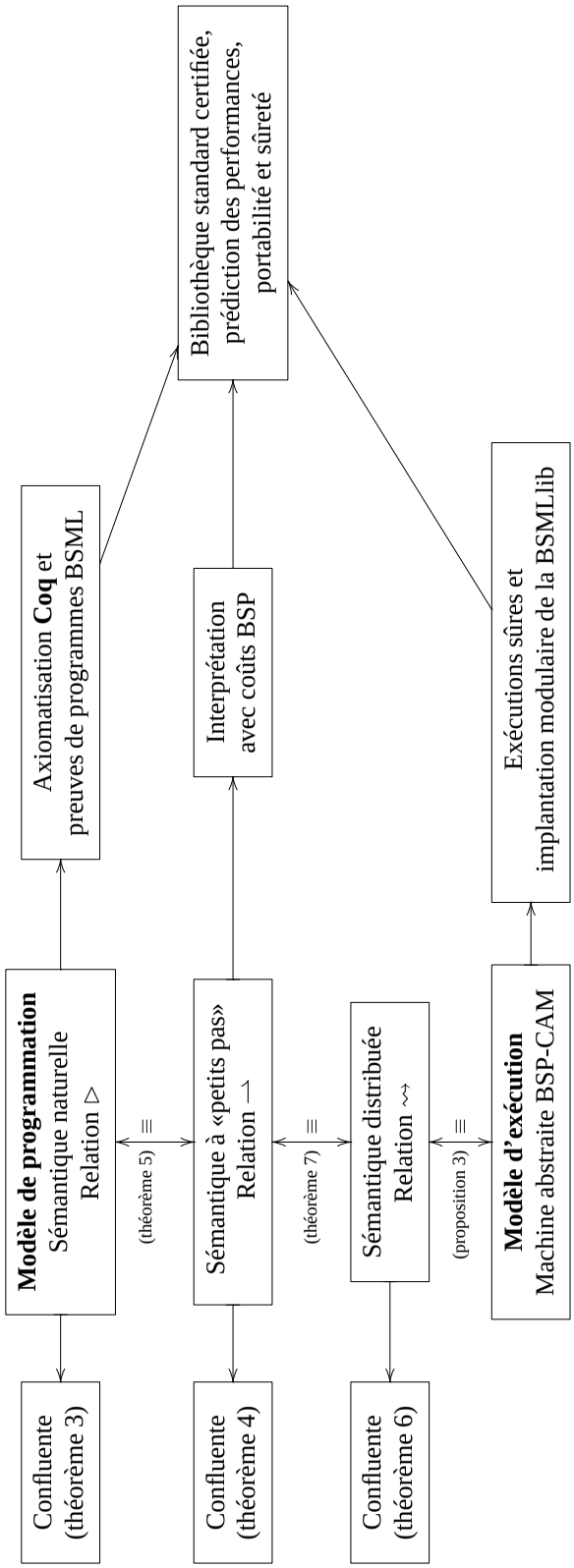


Figure 13.1 — Schéma des résultats de la première partie

Figure 13.2 — Schéma des résultats des deuxième et troisième parties

superposition basée sur les super-threads, et décrit ainsi ce qui est nécessaire pour une implantation portable de la superposition.

Cette sémantique permet (entre autres) de comparer les coûts des programmes avec et sans super-threads. Ceux-ci sont moindres (dans un algorithme «diviser-pour-régner») avec les super-threads car les communications et les synchronisations sont fusionnées.

De plus, la superposition simule une autre opération de multi-traitement : la juxtaposition parallèle [190]. Celle-ci divise le réseau en sous-réseaux (composition parallèle qui autorise d'évaluer deux expressions, chacune sur un sous-ensemble disjoint de la machine parallèle) et qui suit toujours le modèle BSP (qui interdit la synchronisation des sous-réseaux). Cette nouvelle construction est particulièrement intéressante pour la multiprogrammation car elle simplifie grandement l'écriture des algorithmes parallèles «diviser-pour-régner».

Structures de données parallèles. Nous avons aussi décrit, dans le chapitre 8, l'implantation en BSML de structures de données classiquement utilisées dans la programmation fonctionnelle. Les modules de OCaml ont été massivement utilisés afin de factoriser le code et rendre ainsi sa maintenance plus aisée. Les structures de données parallèles ont la même signature (du point de vue des modules) que celles séquentielles, ce qui facilite leur utilisation pour les néophytes en programmation parallèle. Les coûts BSP des opérations ont été donnés, ce qui est impossible avec des compilateurs à parallélisation automatique. Les implantations utilisent la primitive de superposition afin de redistribuer les données entre les processeurs. Des preuves du bon fonctionnement des opérations ont aussi été données (ces preuves seront la base d'une certification de l'implantation de ces structures). Un exemple d'application montre de manière expérimentale que ces implantations sont relativement efficaces et qu'il est possible d'obtenir une application parallèle, à partir d'un code séquentiel, sans avoir à connaître tous les détails de la programmation parallèle.

Entrées/sorties parallèles. La deuxième extension est dédiée à la programmation d'algorithmes nécessitant un nombre important de données. En effet, Le langage BSML permet d'écrire des algorithmes BSP en mode direct et cela de manière fonctionelle. Mais pour quelques applications où la taille du problème est vraiment importante, l'utilisation de la mémoire externe est nécessaire. Dans le chapitre 9, nous avons présenté une extention, appelé EM^2-BSP , du modèle BSP pour la prise en compte des coûts des accès à la mémoire externe. Nous avons ensuite étendu l'ensemble des primitives de la BSMLlib pour pouvoir effectuer des opérations d'entrées/sorties sûres sur ces mémoires externes. Une sémantique dynamique a été donnée, ainsi que les coûts EM^2-BSP des primitives. Quelques expériences ont été faites pour tester les gains qu'apporte cette nouvelle extension.

13.1.3 Partie 3 : Opérations asynchrones et globalisées

Le troisième et dernier axe est la définition d'extensions de BSML pour des opérations data-parallèles asynchrones et globalisées. La partie droite du schéma de la figure 13.2 résume cette approche.

MSPML. Pour ce faire, nous avons modifié les primitives de BSML afin de désynchroniser les super-étapes du modèle BSP. Ce nouveau langage appelé *Minimally Synchronous Parallel ML* (MSPML) est basé sur le modèle de coût *Message Passing Model* (MPM). MSPML est donc un langage fonctionnel parallèle dont la syntaxe et la sémantique de «haut niveau» sont similaires à celles de BSML. Par contre, la sémantique de «bas niveau» et l'implantation sont différentes car basées sur deux modèles d'exécution distincts : MSPML permet la programmation d'algorithmes MPM où les processeurs ne se synchronisent qu'avec ceux auxquels ils communiquent. La sémantique distribuée a été prouvée équivalente aux sémantiques de «haut niveau». Les premières expériences du prototype sur l'implantation d'un patron algorithmique montrent que le modèle de coût MPM s'applique bien à MSPML.

Méta-calcul. Il a été prouvé, dans de nombreux articles, comme par exemple dans [171, 242], que le modèle BSP permet la création de programmes parallèles efficaces et portables. Pourtant, la complexité supplémentaire qu'apporte un méta-ordinateur demande une révision du modèle. De nombreux travaux, comme celui décrit dans l'article [165], ont montré que certaines applications pouvaient tirer parti de l' environnement hiérarchique d'un méta-ordinateur.

Nous avons ainsi étendu le modèle BSP avec le modèle MPM, pour donner un modèle à deux niveaux. MSPML a donc été fusionné avec BSML pour obtenir un langage à deux couches dédié au méta-calcul

(dit départemental car nous ne considérons que des machines parallèles qui font partie d'une même organisation). MSPML est utilisé comme langage pour coordonner des unités BSP qui sont, elles, programmées avec BSML. Ce nouveau langage, appelé *Departmental Metacomputing ML* (DMML) est toujours basé sur une sémantique opérationnelle et sur un modèle de coût appelé *Departmental Metacomputing Model* (DMM) afin de pouvoir raisonner (comme précédemment) sur les temps d'exécutions et les résultats des programmes.

DMML est donc consacré à la programmation d'algorithmes DMM. En se Basant sur ce modèle d'exécution, il est alors possible de faire une analyse de coût DMM des programmes DMML. Une première implantation a été effectuée, mais elle n'est pas encore très portable. De premières expérimentations ont été réalisées pour tester les gains de performances qu'apportent le modèle et le langage.

13.2 Perspectives

13.2.1 Le projet PROPAC

Le projet PROPAC[2] de l'ACI Jeunes Chercheurs vise à poursuivre les efforts de recherche présentés dans la première partie de ce mémoire. La poursuite de ces travaux pourrait se faire selon plusieurs dimensions, toujours dans une thématique générale de certification. Il est ainsi prévu d'étendre les résultats d'équivalence des sémantiques de la première partie à un BSML plus complet, comprenant entre autres des extensions pour les exceptions parallèles. Nous ne détaillerons pas cet axe, notamment parce qu'il fait l'objet d'un travail de thèse à partir de cette année. Dans le cadre de ce projet, les perspectives possibles, dans la continuité des travaux de recherches ici présentés et auxquelles nous nous consacrerons, seront :

- La poursuite du travail commencé au chapitre 6 sur la certification de programmes BSML et MSPML purement fonctionnels ;
- L'extension de l'approche à des programmes impératifs ;
- La conception d'un nouveau système de typage plus général empêchant notamment l'emboîtement de vecteurs (parallèles pour BSML et MSPML, parallèles et départementaux pour DMML) ;
- Enfin, en ce qui concerne la certification des programmes BSML, nous nous sommes intéressés à la correction de la sémantique fonctionnelle des programmes, indépendamment de tout aspect parallèle. C'est pourquoi il faut également travailler sur la prévision de performance (semi)-automatisée des programmes.

Preuves de programmes fonctionnels parallèles. Les fonctions ici certifiées sont simples à comprendre, mais tout en étant d'une variété suffisante pour se convaincre que le traitement d'un grand nombre d'algorithmes BSP pourra être effectué.

Nous pensons notamment à la validation de programmes BSP plus complexes tels que le calcul scientifique [35] ou les squelettes algorithmiques [151]. Notre but est d'avoir une bibliothèque standard BSMLlib certifiée plus importante. Nous avons déjà certifié une grande partie des fonctions de la bibliothèque standard de la BSMLlib. Un travail intéressant serait la preuve de correction de l'implantation du patron Diffusion puisqu'il est la base d'un formalisme présenté dans [152] qui permet de rendre «plat» des programmes avec du parallélisme emboîté.

Dans [106], les auteurs ont certifié l'implantation en OCaml des ensembles et des dictionnaires. Un travail intéressant sera la certification en **Coq** des implantations parallèles décrites dans le chapitre 8.

Nous pourrions également traiter de la certification de programmes MSPML. L'axiomatisation sera très proche de celle de BSML. De plus, la cohérence est assurée car la primitive de communication get de MSPML peut être écrite avec la primitive **put** de BSML et a déjà été prouvée correcte.

Le travail présenté dans le chapitre 6 pourrait aussi être appliqué à d'autres extensions parallèles des langages fonctionnels, notamment les langages à squelettes (comme par exemple [82]) en donnant les squelettes comme des paramètres, leurs axiomes associés et une implantation séquentielle équivalente pour la cohérence dans le but de valider d'autres formes d'algorithmes parallèles.

Programmes parallèles impératifs certifiés. Il existe dans la littérature de nombreux algorithmes parallèles BSP décrits avec un mini langage algorithmique impératif. Ces programmes décrivent comment

[2] page web à `http://wwwpropac.free.fr`, projet auquel l'auteur participe

calculer mais ne donnent jamais la signification formelle du calcul ni la raison pour laquelle celui-ci est correct. Dans le meilleur des cas, les commentaires insérés dans le code fournissent une partie de cette information. Une méthode formelle donnerait la capacité de dire comment la machine BSP doit calculer et pourquoi cela est-il correct.

Des travaux [65, 160, 246] ont été réalisés sur la possibilité d'étendre la logique de Hoare (avec les pré- et post-conditions) pour des programmes impératifs parallèles ou BSP. Ces programmes sont écrits dans un langage algorithmique avec une exécution SPMD. Ce langage est enrichi de boucles parallèles, de primitives de communications (get et put) soit sur des variables partagées, soit par l'échange des valeurs de ces variables (écrire la valeur de telle variable dans telle variable de tel autre processus) et d'un opérateur de synchronisation globale rendant les valeurs échangées accessibles. Malheureusement, aucun de ces travaux n'a pu fournir un logiciel pour vérifier à l'aide d'un assistant de preuves les programmes ainsi annotés. De plus, aucun programme réel n'a pu être vérifié formellement faute d'un tel logiciel. Cette défaillance vient essentiellement du manque de formalisme pour générer les obligations de preuves.

Le but de cet axe est d'étendre le logiciel WHY [105, 107] pour faire la vérification de programmes parallèles BSP utilisant de tels formalismes[3]. Nous pourrons utiliser les résultats de WHY sur la transformation des programmes impératifs en termes du λ-calcul pour développer une transformation automatique des programmes BSP en BSλ-calcul à partir des annotations déjà définies dans [65, 160, 246] et ainsi faire de la génération d'obligations de preuves. Une extension comme dans le logiciel Caduceus [54] pourra alors être envisagée pour annoter des programmes C utilisant la BSPlib, ou Krakatoa [169] pour des programmes utilisant JBSP [128] pour Java. Ces outils pourraient alors servir à certifier des algorithmes BSP déjà existants.

Typage. Dans les sémantiques du chapitre 3 l'emboîtement des vecteurs parallèles est empêché ici *dynamiquement*. Il existe un système de typage polymorphe avec inférence de types pour un mini-BSML [R3]. Toutefois, il ne s'agit là que d'un fragment relativement petit et fonctionnel pur de BSML.

Pour assurer complétement la sûreté des programmes BSML, MSPML et DMML, il nous faudrait aussi concevoir un système de types rejetant les programmes non valides : emboîtement des vecteurs parallèles ou départementaux, opérations locales exécutées globalement (et *vice versa*), affectations incorrectes, *etc.* Ce système devra être suffisemment extensible (et complet) pour résoudre tous ces problèmes. Nous nous baserons sur les analyses de flots de [224]. C'est un travail en cours.

Prévision automatique de performances. Une direction complémentaire est la description d'analyses statiques des performances d'un programme. L'analyse automatique et statique des performances d'un programme, communément appelée analyse de coût, permet une estimation des programmes donnée en terme d'une métrique désirée telle que le temps ou l'espace. Elle a des applications importantes notamment dans l'optimisation des programmes. En effet, les prévisions de performances sont alors des bornes supérieures des évaluations du programme. C'est une certification de la bonne exécution dudit programme sur une machine réelle disposant des ressources nécessaires.

Une analyse automatique de coût peut ainsi être utilisée pour aider le programmeur à vérifier si son programme est viable ou non et à choisir un autre algorithme pour optimiser son programme. Dans le contexte de grille de calculs, une architecture parallèle peut refuser (ou accepter) l'exécution de tel ou tel programme en fonction de sa formule de coût et des paramètres réels de la machine. Les programmes ne sont exécutés que si leurs «certificats de coûts» sont acceptables.

Au niveau des langages séquentiels de programmation, les systèmes pour les méthodes de micro analyse expriment les coûts sous forme de constantes. Ces constantes décrivent le temps estimé pour l'évaluation des opérateurs élémentaires (comme par exemple le temps moyen d'une addition d'entiers) et des constructions habituelles du langage (telles que la mise dans l'environnement d'une valeur). Ces constantes peuvent être obtenues en utilisant des tests.

L'article [228] expose une analyse statique de programmes fonctionnels avec traits impératifs pour donner une estimation de leurs performances et où les formules de coût peuvent être composées. Une analyse

[3]WHY prend en entrée des programmes annotés, écrits dans un langage impératif incluant du polymorphisme, des fonctions d'ordre supérieur, des exceptions et des tableaux. Il génère des obligations de preuves pour vérifier la correction du programme. Ces obligations peuvent être données pour différents assistants de preuves (comme par exemple PVS [225] ou **Coq**). Pour réaliser cette transformation, WHY utilise une interprétation des programmes impératifs dans le λ-calcul (ces assistants de preuves étant basés sur des extensions typées du λ-calcul) en utilisant une analyse statique basée sur un système de typage des effets [253]. À partir de la sémantique formelle de ces programmes, WHY est alors à même de fournir les hypothèses pour que les pré- et post- conditions soient vérifiables (et prouvables formellement). Il fournit ainsi les obligations de preuves nécessaires pour la correction du programme.

de coûts des programmes extraits du système de développement de preuves Nuprl [74, 214] a été présentée dans l'article [27]. Les programmes sont alors beaucoup plus simples à analyser car ceux-ci ont toujours une récursion bien fondée basée sur la récursion primitive. Les formules de coûts peuvent être réduites en utilisant un système tel que Mathematica. On retrouve aussi cette génération d'équation de coûts dans [264] mais cette fois-ci pour un mini-ML. L'article [158] propose une analyse basée sur la notion de forme : la forme d'un tableau est par exemple sa taille. Les coûts des programmes donnés par l'analyse sont compositionnels et forment une algèbre de coûts. Ceci a été appliqué à un mini-langage à patrons [159]. Une analyse de coût de DML [277], qui est un ML avec types dépendants (initialement conçu pour la preuve de programmes), a été proposée dans [126].

Un travail futur sera donc d'étendre les analyses de [27, 71, 142, 175, 228, 264], pour l'analyse de coûts de programmes parallèles BSML, MSPML et DMML (basée sur les modèles de coûts). Ces analyses pourront soit se faire sur des programmes parallèles quelconques (et permettre au programmeur de rapidement constater la bonne réalisation de ses programmes), soit sur des programmes extraits par un assistant de preuves (afin de certifier le programme et son coût, c'est-à-dire vérifier que le programme est correct vis-à-vis de la spécification et correct vis-à-vis de la complexité algorithmique).

L'ajout d'annotations (indications) de coûts qui seront soit vérifiées automatiquement par le système soit demandées à être prouvées avec l'aide d'un assistant de preuves est aussi une voie à suivre. L'emploi d'autres environnements pour la certification de bibliothèque de programmes (tel que `FoC`[4] par exemple) est une possibilité.

13.2.2 BSML : applications et nouvelles implantations

Il n'y a, pour l'instant, que de petits exemples de l'utilisation de nos bibliothèques, en plus des bibliothèques standards qui fournissent déjà un certain nombre de cas d'utilisation. Toutefois, il est nécessaire, pour mener des expériences plus importantes d'avoir plus d'applications. L'implantation d'utilitaires dans des domaines autres que purement informatiques, comme par exemple la bio-informatique avec un algorithme EM-BSP de recherche de motifs de [104], permettrait de valider expérimentalement le langage BSML et créerait sûrement de nouveaux besoins.

Plusieurs directions peuvent être suivies. La première est la conception et l'implantation d'autres structures de données parallèles telles que les listes (et certains algorithmes sur celles-ci, tels que décrits dans [129]) ou les files de priorités parallèles [115, 116].

OCamlgraph [73] est une bibliothèque générique de manipulation des graphes pour OCaml. Celle-ci utilise les dictionnaires (ou les tables de hachage) pour représenter les sommets d'un graphe et les ensembles pour représenter les arêtes. En remplaçant ces structures par nos structures de données parallèles, nous pourrions obtenir une représentation parallèle des graphes. Par exemple avec des ensembles parallèles pour représenter les arêtes, celles-ci seraient distribuées sur les différentes composantes de la machine parallèle, et l'on pourrait itérer en parallèle sur les arêtes d'un graphe donné : il serait alors possible d'implanter des algorithmes parallèles sur les graphes comme décrits dans [64].

Au niveau des entrées/sorties parallèles, plusieurs directions sont possibles. La première est l'implantation des primitives de persistance avec des bibliothèques dédiées aux E/S parallèles (comme celles décrites dans [161]). Par exemple, des bibliothèques de niveau bas pour les disques RAID partagés pourraient être utilisées, permettant ainsi une implantation tolérante aux pannes de nos primitives.

Une direction complémentaire est l'implantation d'algorithmes BSP classiques [90, 203, 238] et leurs transformations, comme décrit dans [93], en des algorithmes EM^2-BSP . Nous aurions ainsi une nouvelle bibliothèque de fonctions, comme dans la bibliothèque standard de la BSMLlib, mais dédiée aux problèmes manipulant une grande quantité de données. Une implantation avec E/S des structures de données parallèles pourrait être un premier pas. Nous avons également étendu le modèle BSP pour inclure les disques partagés. Pour valider le modèle de coût et la prévision des performances de nos programmes, nous avons besoin d'un programme de test qui déterminera automatiquement les paramètres EM^2-BSP en plus des paramètres BSP [34].

13.2.3 Langages pour le méta-calcul et les grilles de calcul

La programmation ML dédiée au méta-calcul ou au calcul sur grilles de calcul n'en est qu'à ses balbutiements. Les perspectives sont donc nombreuses et plus exploratoires.

[4] page web à `http://www-spi.lip6.fr/foc`

Minimally Synchronous Parallel ML. De nouveaux travaux sur MSPML ont été entrepris à l'été 2005 [25, 26]. Il reste encore beaucoup à faire, et une nouvelle thèse a commencé en septembre 2005.

Les travaux en cours prolongent [25, 26] afin d'avoir une preuve de correction des mécanismes de gestion des environnements de communications [25, 194]. Cette preuve pourrait être effectuée avec les *Abstract State Machines* [42, 131, 236], les logiques temporelles du premier ordre [24] et leurs outils de *models-checkings* (vérificateurs de modèles automatiques) respectifs.

Au niveau sémantique, nous pourrions également donner la définition d'une machine abstraite MPM. Celle-ci pourrait alors être utilisée pour guider une implantation modulaire de MSPML comme celle de BSML dans le chapitre 4. En effet, pour le moment, MSPML est seulement implantée avec les outils TCP/IP d'OCaml. Cela rend les programmes non-portables car dans un grand nombre d'architectures parallèles, les processeurs n'ont pas d'adresse IP. Une implantation modulaire permettrait l'utilisation de bibliothèques de plus «bas niveau» et tolérantes aux pannes [49] ou optimales sur certains types de réseaux.

Une dernière direction sera la comparaison formelle et réelle des coûts BSP et MPM de nos sémantiques. Par réelle, nous parlons, bien entendu, des temps d'exécution des programmes BSML et MSPML sur une machine parallèle. L'utilisation de plusieurs implantations de MSPML serait alors nécessaire : il nous faudrait comparer les performances des programmes avec les différentes implantations de BSML et de MSPML.

Algorithmes en Departemental Metacomputing ML. On trouve dans la littérature de nombreux algorithmes BSP pour différents domaines d'application. Un travail intéressant serait l'adaptation et l'implantation de ces algorithmes en DMML. Un premier pas serait la définition d'un ensemble d'algorithmes classique DMM comme il a été fait pour BSP [147] et d'un programme de tests permettant de déterminer automatiquement les paramètres DMM d'un méta-ordinateur (comme celui présenté pour BSP dans [34]).

Tolérance aux pannes. La tolérance aux pannes pour la bibliothèque BSPlib a été traitée dans [146]. La documentation de la dernière version de l'Oxford BSPlib indiquait que la version suivante devait contenir la migration de processus appliquée à l'équilibrage de charges et à la tolérance aux pannes. Mais cette version n'a jamais été diffusée. L'actuelle implantation de la bibliothèque PUB propose une version basée sur TCP/IP permettant de faire migrer des processus, mais la tolérance aux pannes n'est pas évoquée.

Bien entendu, il est possible d'avoir actuellement une tolérance aux pannes par le biais de MPICH-V [43, 46, 47, 49]. La structuration des programmes BSP autorise toutefois d'avoir des solutions plus simples. De même, ce qui est mis en oeuvre dans MPICH-V ne semble pas être valable en pratique pour le calcul sur la grille (*Grid* dans la litérature anglo-saxonne). Celui-ci nécessite une plus grande structuration. Le modèle DMM et le langage DMML offrent une telle structuration. Il est alors tout à fait envisageable d'améliorer notre implantation actuelle avec une meilleur prise en charge des pannes.

Autres langages pour les grilles de calcul. La mobilité de processus est utilisée dans le cadre de la tolérance aux pannes. Mais il est également intéressant, pour la programmation sur une grille de calcul, de pouvoir disposer de la mobilité pour des agents parallèles et non pas seulement d'un processus dans un calcul parallèle.

Ainsi, on peut imaginer dans un système *Grid* qu'un utilisateur soumet un agent (intégrant un programme) à exécuter et que, selon différentes contraintes, cet agent migre sur la machine parallèle permettant une exécution dudit programme dans le respect de ces contraintes. Une possibilité est de plonger le BSλ-calcul dans le join-calcul [120] afin de voir quels peuvent être les possibilités offertes et les problèmes potentiels.

Une autre approche intéressante serait d'implanter les primitives BSML pour Dynamic BSP [198] ou de créer de nouvelles primitives pour ce modèle. Nous pourrions utiliser les fonctionnalités de mobilité de langages fonctionnels déjà existant comme JoCaml [72] ou Acute [237].

Publications

Revues internationales

[R1] F. Gava. External Memory in Bulk-Synchronous Parallel ML. *Parallel and Distributed Computing Practices*, Nova Science Publishers, à paraître. Version révisée et étendue de [C2].

[R2] F. Gava et F. Loulergue. A Functional Language for Departmental Metacomputing. *Parallel Processing Letters*, 15(3) :(289–304), 2005.

[R3] F. Gava et F. Loulergue, A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5) :(665–671), 2005. disponible en ligne sur site de la revue. Version révisée de [C8]

[R4] F. Loulergue, F. Gava, M. Arapinis et F. Dabrowski, Semantics and Implementation of Minimally Synchronous Parallel ML. *International Journal of Computer and Information Science*, ACIS, 5(3) : (182–199), 2004. Version révisée et étendue de [C5].

[R5] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3) :(365–376), 2003.

Conférences internationales (avec sélection)

[C1] F. Loulergue, F. Gava et D. Billiet Bulk-Synchronous Parallel ML : Modular Implementation and Performance Prediction. In V. S. Sunderam and G. Dick vab Albada and P. M. A. Sloot and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2005), Part IV*, LNCS, pages 1046–1054. Springer Verlag, 2005.

[C2] F. Gava. Parallel I/O in Bulk Synchronous Parallel ML. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2004), Part III*, LNCS, pages 339–346. Springer Verlag, 2004.

[C3] F. Gava. Design of Deparmental Metacomputing ML. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2004)*, LNCS, pages 50–53. Springer Verlag, 2004.

[C4] F. Gava et F. Loulergue. Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features. In G. Joubert, W. Nagel, F. Peters, and W. Walter, editors, *Parallel Computing : Software Technology, Algorithms, Architectures and Applications, Proceeding of the 10th ParCo Conference*, Dresden, 2003. North Holland/Elsevier, 2004.

[C5] F. Gava, F. Loulergue et F. Dabrowski. A Parallel Categorical Abstract Machine for Bulk Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03)*, pages 293–300. ACIS, 2003.

[C6] F. Dabrowski, F. Loulergue et F. Gava. Pattern Matching of Parallel Values in Bulk Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03)*, pages 301–308. ACIS, 2003.

[C7] M. Arapinis, F. Loulergue, F. Gava et F. Dabrowski. Semantics of Minimally Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03)*, pages 260–267. ACIS, 2003.

[C8] F. Gava et F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 215–229. Springer Verlag, 2003.

[C9] F. Gava et F. Loulergue. A Parallel Virtual Machine for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS 2003), Part I*, number 2657 in LNCS, pages 155–164. Springer Verlag, june 2003.

Manifestations internationales (avec sélection)

[W1] F. Gava. A modular implementation of data structures in Bulk-Synchronous Parallel ML *Third Workshop on High-Level Parallel Programming and Applications* (HLPP'2005). 2003.

Conférences nationales (avec sélection)

[N1] F. Gava. Une implantation de la juxtaposition. In T. Hardin, editor, *Journées Francophones des Langages Applicatif, JFLA*, INRIA, january 2006, à paraître.

[N2] F. Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. In V. Ménissier Morain, editor, *Journées Francophones des Langages Applicatif, JFLA*, pages 55–68. INRIA, january 2004.

[N3] F. Gava et F. Loulergue. Synthèse de types pour Bulk Synchronous Parallel ML. In J.C. Filliâtre, editor, *Journées Francophones des Langages Applicatifs (JFLA 2003)*, pages 153–168, january 2003.

Bibliographie

[1] S. Adachi, H. Iwasaki, and Z. Hu. Diff : A Powerfull Parallel Skeleton. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, volume 4, pages 425–527. CSREA Press, 2000.

[2] G. Akerholt, K. Hammond, S. Peyton-Jones, and P. Trinder. Processing transactions on GRIP, a parallel graph reducer. In Bode et al. [39].

[3] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting Skeleton Programs : How to Evaluate the Data-Parallel Stream-Parallel Tradeof. In S. Gorlatch, editor, *First International Workshop on Constructive Methods for Parallel Programming (CMPP'98)*, Research Report MIP-9805. University of Passau, May 1998.

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP : Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1) :71–79, July 1997.

[5] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffin. Net Juggler : Running VR Juggler with Multiple Displays on a commodity Component cluster. In *IEEE VR'2002*, 2002.

[6] R. Alpert and J. Philbin. cbsp : Zero-cost synchronization in a modified bsp model. Technical Report 97-054, NEC Research Institute, 1997.

[7] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag, 2nd ed. edition, 1997.

[8] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory Algorithms for Processing Line Segments in Geographic Information Systems. In Paul G. Spirakis, editor, *Algorithms - ESA '95, Third Annual European Symposium, Corfu, Greece, September 25-27, 1995, Proceedings*, volume 979 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 1995.

[9] Arvind and R. Nikhil. I-structures : Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.

[10] M. J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J.-J. Tsay. Multisearch techniques : Parallel data structures on mesh-connected computers. *Journal of Parallel and Distributed Computing*, 20 :1–13, 1994.

[11] O. Aumage, L. Bougé, and al. Madeleine II :A Portable and Efficient Communication Library for High-Performance Cluster Computing. *Parallel Computing*, 28(4) :607–626, 2002.

[12] O. Aumage, L. Bougé, A. Denis, L. Eyraud, R. Namyst, and C. Pérez. Communications efficaces au sein d'une interconnexion hétérogène de grappes : Exemple de mise en oeuvre dans la bibliothèque madeleine. In F. Baude, editor, *Calcul réparti à grande échelle*, chapter 4, pages 103–128. Hermès, Notice, lavoisier edition, 2002.

[13] O. Aumage and G. Mercier. Mpich/MadIII : a cluster of clusters enabled mpi implementation. In *3rd International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, Japan, Tokyo, may 2003. IEEE/ACM.

[14] M. Aumor, F. Arguello, J. Lopez, O. Plata, and L. Zapata. A data-parallel formulation for divide-and-conquer algorithms. *The Computer Journal*, 44(4) :303–320, 2001.

[15] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L : a structured high-level parallel language, and its structure support. *Concurrency : Practice and Experiences*, 7(3) :225–255, May 1995.

[16] O. Ballereau, F. Loulergue, and G. Hains. High-level BSP Programming : BSML and BSλ. In G. Michaelson and Ph. Trinder, editors, *Trends in Functional Programming*, pages 29–38. Intellect Books, 2000.

[17] M. Bamha. L'implémentation d'un langage portable à parallélisme emboîté en processus statiques. Mémoire de DEA d'informatique, LIFO, Université d'Orléans, Septembre 1996.

[18] M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3) :317–328, 2003.

[19] M. Bamha and G. Hains. Frequency-adaptive join for Shared Nothing machines. *Parallel and Distributed Computing Practices*, 2(3) :333–345, 1999.

[20] M. Bamha and G. Hains. An Efficient equi-semi-join Algorithm for Distributed Architectures. In V. Sunderam, D. van Albada, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2005)*, LNCS. Springer, 2005. to appear.

[21] H. P. Barendregt. Functional programming and lambda calculus. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science (vol. B)*, pages 321–364. Elsevier, 1990.

[22] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université de Paris VII, 1999.

[23] W. Bäumker, A. adn Dittrich and F. Meyer auf der Heide. Truly efficient parallel algorithms : c-optimal multisearch for an extension of the BSP model. In 3^{rd} *European Symposium on Algorithms (ESA)*, pages 17–30, 1995.

[24] D. Beauquier and A. Slissenko. A first order logic for specification of timed algorithms : basic properties and a decidable class. *Annals of Pure and Applied Logic*, 113, 2002.

[25] A. Belbekkouche. Mspml : Environnements de communication et tolérance au pannes. Master's thesis, Université d'Orléans (LIFO), 2005.

[26] R. Benheddi. Composition parallèle pour mspml : sémantique et implémentation. Master's thesis, Université d'Orléans (LIFO), 2005.

[27] R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318 :79–103, 2004.

[28] J. Berthold. Towards a Generalised Runtime Environment for Parallel Haskells. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2004), Part III*, LNCS, pages 307–314. Springer Verlag, 2004.

[29] J. Berthold and R. Loogen. Analysing dynamic channels for topology skeletons in eden. Technical Report 0408, Institut für Informatik, Lübeck, September 2004. (IFL'04 workshop), C. Grelck and F. Huch eds.

[30] Y. Berthot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[31] P.B. Bhat, V.K. Prasanna, and C.S Raghavendra. Adaptative communication algorithms for distributed heterogeneous systems. *Parallel and Distributed Computation*, 59 :252–279, 1999.

[32] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP versus LogP. *Algorithmica*, 24 :405–422, 1999.

[33] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[34] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

[35] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations : Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.

[36] V. Blanco, J. A. González, C. León, C. Rodríguez, G. Rodríguez, and M. Printista. Predicting the performance of parallel programs. *Parallel Computing*, 30 :337–356, 2004.

[37] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[38] G.E. Blelloch. NESL : A Nested Data-Parallel Language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[39] A. Bode, M. Reeve, and G. Wolf, editors. *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, Munich, June 1993. Springer.

[40] O. Bonorden, B. Juurlink, I. Von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2) :187–207, 2003.

[41] O. Bonorden, F. Meyer auf der Heide, and R. Wanka. Composition of Efficient Nested BSP Algorithms : Minimum Spanning Tree Computation as an Instructive Example. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2002.

[42] E. Börger and R. Stärk. *Abstract State Machines*. Springer, 2003.

[43] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. Mpich-v : Toward a scalable fault tolerant mpi for volatile nodes. In *High Performance Networking and Computing (SC2002)*, Baltimore, USA, NOvember 2002. IEEE/ACM.

[44] L. Bougé. Le modèle de programmation à parallélisme de données : une perspective sémantique. *RAIRO Technique et Science Informatiques*, 12(5), 1993.

[45] L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard, and B. Virot. Formal Validation of Data–Parallel Programs : a Two–Component Assertional Proof System for a Simple Language. *Theoretical Computer Science*, 189(1-2) :71–107, 1997.

[46] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette. Mpich-v2 : a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *High Performance Networking and Computing (SC2003)*, Phoenix, USA, NOvember 2003. IEEE/ACM.

[47] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello. Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI. In *19th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 05)*, Denver, USA, Avril 2005. IEEE/ACM.

[48] A. Bouteiller, H.-L.Bouziane, T. Herault, P. Lemarinier, and F. Cappello. Hybrid preemptive scheduling of mpi applications on the grids. In *5th International Workshop on Grid Computing (Grid04)*, Pittsburgh, USA, November 2004. IEEE/ACM.

[49] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated Checkpoint versus Message log for Fault Tolerant MPI. *Journal of high performance computing and networking (IJHPCN)*, 2005. Inderscience Publisher.

[50] S. Boutin. Proving correctness of the translation from mini-ml to the cam with the coq proof development system. Technical Report 2536, INRIA, 1995.

[51] A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999.

[52] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Pe na Marí. Eden– the paradise of functional concurrent programming. In L. Bouge and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, number 1124 in LNCS, pages 359–368, Lyon, France, August 1996. Springer.

[53] F. Cabestre, C. Percebois, and J.-P. Bodeveix. Abstract machine construction through operational semantics refinements. *Future Generation Computer Systems*, 16(7) :753–769, May 2000.

[54] The Caduceus verification tool for C programs. Web pages at `why.fri.fr/caduceus`, 2003.

[55] R. Calinescu. Bulk synchronous parallel scheduling of uniform dags. In Luc Bouge et al., editors, *Euro-Par'96. Parallel Processing*, volume 2 of *Lecture Notes in Computer Science 1124*, pages 555–562. Springer-Verlag, 1996.

[56] D. K. G. Campbell. A survey of models of parallel computation. Technical report, Department of Computer Science, University of York, March 1997.

[57] F. Cappello, P. Fraigniaud, B. Mans, and A.L. Rosenberg. HiHCoHP toward a realistic communication model for hierarchical hyperclusters of heterogeneous processors. In *IEEE/ACM IPDPS'2001*. IEEE press, 2001.

[58] L. Cardelli. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 208–217, Austin, Texas, August 1984. ACM.

[59] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in java. *Concurrency : Practive and Experience*, 10(11–13) :1043–1061, 1998.

[60] E. Caron, O. Cozette, D. Lazure, and G. Utard. Virtual memory management in data parallel applications. In *HPCN Europe*, pages 1107–1116, 1999.

[61] E. Caron and G. Utard. On the performance of parallel factorization of out-of-core matrices. *Parallel Computing*, 30(3) :357–375, 2004.

[62] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly France, 2000. freely available in english at `http://caml.inria.fr/oreilly-book`.

[63] M. M. T. Chakravarty, F. W. Schröer, and M. Simons. V – nested parallelism in C. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Proceedings of "Programming Models for Massively Parallel Computers 1995"*, pages 167–174, Berlin, 1996. IEEE Computer Society Press.

[64] A. Chan, F. Dehne, and R. Taylor. Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *Journal of High Performance Computing Applications*, 2005. to appear.

[65] Y. Chen and W. Sanders. Top-Down Design of Bulk-Synchronous Parallel Programs. *Parallel Processing Letters*, 13(3) :389–400, 2003.

[66] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, D. E. Vengroff, and J. S. Vitter. External-memory Graphs Algorithms. In *ACM-SIAM Symp on Discrete Algorithms*, pages 139–149, 1995.

[67] F. Clément, A. Vodicka, R. Di Cosmo, and P.Weis. Parallel programming with the ocamlp3l system, application to to numerical code coupling. Technical Report RR-5131, INRIA, 2004.

[68] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, and M. Holzner. Performance issues of the parallel pam-crash code. *Supercomputer Applications and High Performance Computing*, 11(1) :3–11, Spring 1997.

[69] M. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press, 1989. Available at `homepages.inf.ed.ac.uk/mic/Pubs`.

[70] M. Cole. Bringing Skeletons out of the Closet : A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3) :389–406, 2004.

[71] M. Cole and Y. Hayashi. Static Performance Prediction of Skeletal Programs. *Parallel Algorithms and Applications*, 17(1) :59–84, 2002.

[72] S. Conchon and F. Le Fessant. Jocaml : Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, pages pages 22–29. IEEE Press, 1999.

[73] S. Conchon, J.-C. Filliâtre, and J. Signoles. OcamlGraph. Web pages at `http://www.lri.fr/~filliatr/ocamlgraph`.

[74] Robert L. Constable et al. *Implementing Mathematics in the Nuprl Proof Development System*. Prentice Hall, 1986.

[75] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 37(2-3), 1988.

[76] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8 :173–202, 1987.

[77] G. Cousineau and M. Mauny. *Approche Fonctionnelle de la Programmation*. Ediscience International, 1995.

[78] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized External Memory Algorithms for Geometric Problems. In *ACM Annual Conf on Computational Geometry*, pages 259–268, 1998.

[79] D. E. Culler, R. M. Karp, David A. Patterson, A. Sahey, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP : towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7) :1–12, July 1993.

[80] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Birkhäuser, Boston, second edition, 1993.

[81] C. Cérin and J. Hai, editors. *Parallel I/O for Cluster Computing (Hardback)*. Kojan Page Science, hermes spenton edition, 2002.

[82] M Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel functional programming with skeletons : the OcamlP3L experiments. In *The ML Workshop*, 1998.

[83] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Ocamlp3l a functional parallel programming system. Liens-98-1, ENS, 1998.

[84] O. Danvy. A journey from interpreters to compilers and virtual machines. In F. Pfenning and Y. Smaragdakis, editors, *GPCE'2003*, number 2830 in LNCS. Springer, 2003.

[85] O. Danvy. A Rational Deconstruction of Landin's SECD Machine. In C. Grelck, F. Huch, G. Michaelson, and P. W.Trinder, editors, *IFL'04*, volume 3474 of *LNCS*, pages 52–71. Springer, 2005.

[86] J. Darlington, A. J. Field, P. G. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In Bode et al. [39].

[87] J. Darlington, Y.K Guo, H.W. To, and Y. Jing. Skeletons for structured parallel composition. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[88] N.G. De Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, whith application to the Church-Rosser theorem. *Indag. Math.*, 34 :381–392, 1972.

[89] P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In L. Bouge and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, number 1124 in LNCS, pages 359–368, Lyon, France, August 1996. Springer.

[90] F. Dehne. Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14 :173–421, 1999.

[91] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Algorithmica*, 36 :97–122, 2003.

[92] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Parallel virtual memory. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 889–890, Baltimore, MD, 1999.

[93] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*, 35 :567–598, 2003.

[94] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3) :379–400, 1996.

[95] A. Denis, C. Pérez, and T. Priol. Padicotm : An open inegration framework for communication middleware and runtimes. *Future Generation Computer Systems*, 19(4) :575–585, 2003.

[96] N. Deo and P. Micikevicius. Coarse-grained parallelization of distance-bound smoothing for the molecular conformation problem. In S. K. Das and S. Bhattacharya, editors, *4th International Workshop Distributed Computing, Mobile and Wireless Computing (IWDC)*, volume 2571 of *LNCS*, pages 55–66. Springer, 2002.

[97] R. Di Cosmo and S. Pelagatti. A Calculus for Dense Array Distributions. *Parallel Processing Letters*, 13(3) :377–388, 2003.

[98] W. Dittrich and D. Hutchinson. Blocking in Parallel Multisearch Problems. *Theory of Computing Systems*, 34 :145–189, 2001.

[99] W. Dosch. A pair-based functional skeleton and its list homomorphic implementation. In K. Li, Y. Pan, and S. G. Akl, editors, *Tenth International Conference on Parallel and Distributed Computing and Systems (PDCS'98)*, pages 561–567. Acta Press, 1998.

[100] W. Dosch and B. Wiedemann. Calculating list homomorphism for parallel bracket matching. In H.R. Arabnia, editor, *Proceedings of the Internatioanl Conference on Parallel and Distributed Processing Techniques and Applications*, volume IV, pages 1710–1717. CSREA Press, 2000.

[101] M. Dowse and A. Butterfield. A language for reasoning about concurrent functional I/O. Technical Report 0408, Institut für Informatik, Lübeck, September 2004. (IFL'04 workshop), C. Grelck and F. Huch eds.

[102] D. C. Dracopoulos and S. Kent. Speeding up genetic programming : A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996.

[103] A. Fahmy and A. Heddaya. Communicable memory and lazy barriers for bulk synchronous parallelism in bspk. Technical Report BU-CS-96-012, Boston University, 1996.

[104] P Ferragina and F. Luccio. String search in coarse-grained parallel computers. *Algorithmica*, 24(3) :177–194, 1999.

[105] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 2003. to appear.

[106] J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384, Barcelona, Spain, April 2004.

[107] J.-C. Filliâtre. Why : a software verification tool. Web pages at `why.lri.fr`, 2003.

[108] C. Foisy and E. Chailloux. Caml Flight : a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.

[109] C. Foisy, J. Vachon, and G. Hains. DPML : de la sémantique à l'implantation. In P. Cointe, C. Queinnec, and B. Serpette, editors, *Journées Francophones des Langages Applicatifs*, number 11 in Collection Didactique, Noirmoutier, Février 1994. INRIA.

[110] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual Symposium on Theory of Computing (STOC)*. ACM, May 1978.

[111] P. Fradet and J. Mallet. Compilation of a Specialized Functional Language for Massively Parallel Computers. *Journal of Functional Programming*, 10(6) :561–605, 2000.

[112] J. Furuse and P. Weis. Safe value I/O in Caml. In Catherine Dubois, editor, *Journées Francophones des Langages Applicatifs (JFLA)*, pages 79–99. INRIA, 2000.

[113] E. Gascard and L. Pierre. Formal proof of applications distributed in symmetric interconnection networks. *Parallel Processing Letters*, 13(1), 2003.

[114] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation Series. MIT Press, 1994.

[115] A .V. Gerbessiotis and C. J. Siniolakis. Architecture Independent Parallel Selection with Applications to Parallel Priority Queues. *Theoretical Computer Science*, 301(1-3) :119–142, 2003.

[116] A. V. Gerbessiotis, C. J. Siniolakis, and A. Tiskin. Parallel priority queue and list contraction : The bsp approach. *Computing and Informatics*, 21 :59–90, 2002.

[117] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22 :251–267, 1994.

[118] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW-PRAM : Accouting for convention in parallel algorithms. In d ; d ; sleator, editor, *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 638–648. ACM Press, 1994.

[119] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[120] G. Gonthier and C. Fournier. The Join Calculus : a Language for Distributed Mobile Programming. In G. Barthe, P. Dyjber, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 268–332. Springer, 2002.

[121] M. T. Goodrich, J. J. Tsay, D. E. Vengroff, and J. S. Vitter. External-Memory Computational Geometry. In *IEEE Symp on Foundations of Computer Science*, pages 714–723, 1993.

[122] A. Gordon and R. L. Crole. A sound metalogical semantics for input/output effects. *Mathematical Structures in Computer Science*, 9 :125–188, 1999.

[123] S. Gorlatch. Message passing without send-receive. *Future Generation Computer Systems*, 18(6) :797–805, 2002.

[124] L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601, Las Vegas, USA, 1998. IASTED/ACTA Press.

[125] C. Grelck and Sven-Bodo Scholz. Classes and objects as basis for I/O in SAC. In *Proceedings of IFL'95*, pages 30–44, Gothenburg, Sweden, 1995.

[126] B. Grobauer. Cost recurrence for dml programs. *ACM SIGPLAN Not.*, 36(10) :253–264, 2001.

[127] B. Grégoire. *Compilation de termes de preuves : un (nouveau) mariage entre Coq et OCaml*. PhD thesis, Université de Paris 7, 2004.

[128] Yan Gu, Bu-Sung Lee, and Wentong Cai. JBSP : A BSP programming library in Java. *Journal of Parallel and Distributed Computing*, 61(8) :1126–1142, August 2001.

[129] I. Guérin-Lassous and J. Gustedt. Portable List Ranking : an Experimental Study. *ACM Journal of Experiments Algorithms*, 7(7) :1–18, 2002.

[130] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME : Formal Linear Algebra Methods Environment. *ACM Transaction of Mathematical Software*, 21(4) :422–455, December 2001.

[131] Y. Gurevich. Evolving Algebras 1993 : Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[132] J. Gustedt. Towards realistic implementations of external memory algorithms using a coarse grained paradigm. Technical Report 4719, INRIA, 2003.

[133] G. Hains. Subset synchronization in BSP computing. In H.R Arabnia, editor, *PDPTA'98, International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 242–246, Las Vegas, July 1998. CSREA Press.

[134] G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in LNCS, pages 56–67, Munich, June 1993. Springer.

[135] G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation : Theory and Practice, pages 165–178. Nova Science Publishers, august 2002.

[136] K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3) :413–424, 2003.

[137] K. Hammond, P.W. Trinder, and all. Comparing parallel functional languages : Programming and performance. *Higher-order and Symbolic Computation*, 15(3), 2003.

[138] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Sciences*, 20(2) :415–459, 1992.

[139] T. Hardin, L. Maranget, and B. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2) :131–176, 1998.

[140] J. D. Harrop. Objective Caml for Scientists, 2005. Flying Frog Consultancy Ltd.

[141] Y. Hayashi. *Shaped-based Cost Analysis of Skeletal Parallel Programs*. PhD thesis, University of Edinburgh, 2001.

[142] Y. Hayashi and M. Cole. Bsp-based cost analysis of skeletal programs. In G. Michaelson, P. Trinder, and H.-W. Loidl, editors, *Trends in Functional Programming*, chapter 2, pages 20–28. Intellect, 2000.

[143] C. A. Hermann and C. Lengauer. On the space-time mapping of a class of divide-and-conquer recursions. *Parallel Processing Letter*, 6 :525–537, 1996.

[144] C. A. Hermann and C. Lengauer. Hdc : A high-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2-3) :239–250, 2000.

[145] C. A. Herrmann. Functional meta-programming in the construction of parallel programs. *Parallel Processing Letters*, 2005. to appear.

[146] J. M. D. Hill, S. R. Donaldson, and T. Lanfear. Process migration and fault tolerance of BSPlib programs running on networks of workstations. In *Euro-Par'98*, 1998.

[147] J.M.D. Hill, W.F. McColl, and al. BSPlib : The BSP Programming Library. *Parallel Computing*, 24 :1947–1980, 1998.

[148] Jonathan M. D. Hill and David B. Skillicorn. Practical Barrier Synchronisation. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*. IEEE Computer Society Press, January 1998.

[149] C. A. R. Hoare. An axiomatic basis for computer programmation. *Communication of the ACM*, 1969.

[150] Guy Horvitz and Rob H. Bisseling. Designing a BSP version of ScaLAPACK. In Bruce Hendrickson et al., editor, *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, PA, 1999.

[151] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *European Symposium on Programming*, number 2035 in LNCS, pages 83–97. Springer, 2002.

[152] Z. Hu, T. Takahashi, H. Iwasaki, and M. Takeichi. Segmented Diffusion Theorem. In *IEEE International Conference on Systems, Man and Cybernetics (SMC 02)*. IEEE Press, October 6-9 2002.

[153] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion : Calculating Efficient Parallel Programs. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, pages 85–94. ACM Press, January 22-23 1999.

[154] D. A. Hutchinson, A. Maheshwari, and N. Zeh. An External Memory Data Structure for shortest Path Queries. In *5th Annual Combinatorics and Computing Conf (COCOON '99)*, volume 1627 of *LNCS*, pages 51–60, Berlin, 1999. Springer-Verlag.

[155] K. Höfler, M. Müller, and S. Schwarzer. Design and application of object oriented parallel data structures in particle and continuous systems. In E. Krause and W. Jäger, editors, *High Performance Computing in Science and Engineering*, LNCS. Springer, 1999.

[156] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[157] S.A. Jarvis, J.M.D Hill, C.J. Siniolakis, and V.P. Vasilev. Portable and architecture independent parallel performance tuning using BSP. *Parallel Computing*, 28 :1587–1609, 2002.

[158] B. Jay. A Semantics for Shape. *Science of Computer Programming*, 25 :251–283, 1995.

[159] C.B. Jay, M. I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97*, number 1300 in Lecture Notes in Computer Science, pages 650–661, Passau, 1997. Springer.

[160] H. Jifeng, Q. Miller, and L. Chen. Algebraic Laws for BSP Programming. In L. Bouge and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, number 1124 in LNCS, pages 359–368, Lyon, France, August 1996. Springer.

[161] H. Jin, T. Cortes, and R. Buyya, editors. *High Performance Mass Storage and Parallel I/O*. IEEE Press, wiley-interscience edition, 2002.

[162] J.M.D.Hill, W.F.McColl, D.C.Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T.Suel, T.Tsantilas, and R. Bisseling. BSPlib, the BSP programming library. Technical report, BSP Worldwide, May 1997. URL=`http ://www.bsp-worldwide.org/`.

[163] G. Jones. *Programming in Occam*. Prentice-Hall, 1987.

[164] Y. Kee and S. Ha. An Efficient Implementation of the BSP Programming Library for VIA. *Parallel Processing Letters*, 12(1) :65–77, 2002.

[165] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. H. Hofman. Network performance-aware collective communication for clustered wide-aera systems. *Parallel Computing*, 27 :1431–1456, 2001.

[166] Jin-Soo Kim, Soonhoi Ha, and Chu Shik Jhon. Relaxed barrier synchronization for the BSP model of computation on message-passing architectures. *Information Processing Letters*, 66(5) :247–253, 1998.

[167] U. Klusik, Y. Ortega, and R. Pena. Implementing EDEN : Dreams becomes reality. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of IFL'98*, volume 1595 of *LNCS*, pages 103–119. Springer-Verlag, 1999.

[168] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[169] The Krakatoa tool for JML-annotated Java programs. Web pages at `krakatoa.lri.fr`, 2003.

[170] M. Van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer (editor). Algorithms Foundations of Geographics Information Systems. In *International Symposium on High Performance Computing*, number 1340 in Lecture Notes in Computer Science. Springer, 1997.

[171] P. Krusche. Experimental evaluation of bsp programming libraries. In A. Tiskin and F. Loulergue, editors, *HLPP 2005*. University of Warwick, 2005.

[172] V. Kumar and E. Schwabe. Improved Algorithms and Data Structures for Solving Graph Problems in External Memory. In *IEEE Symp on Parallel and Distributed Processing*, 1996.

[173] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 4(6) :308–320, 1964.

[174] I. Guérin Lassous. *Algorithmes parallèles de traitement de graphes : une approche basée sur l'analyse expérimentale*. PhD thesis, Université de Paris VII, 1999.

[175] D. LeMétayer. ACE : an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2) :248–266, 1988.

[176] X. Leroy. The ZINC experiment : An economical implementation of the ML language. Technical Report 117, INRIA, 1991.

[177] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3) :269–303, 2000.

[178] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4) :431–463, 1993.

[179] P. Lescanne. From lambda-sigma to lambda-upsilon, a journey through calculi of explicit substitutions. In *21st ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 60–69. ACM, January 1994.

[180] P. Letouzey. *Programmation fonctionnelle certifée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, 2004.

[181] Z. Li, P. H. Mills, and J. H. Reif. Models and resource metrics for parallel and distributed computation. *Parallel Algorithms and Applications*, 9(4) :35–59, 1995.

[182] W. B. Ligon and R. B. Ross. *Beowulf Cluster Computing with Linux*, chapter PVFS : Parallel Virtual File System, pages 391–430. T. Sterling, mit press edition, November 2001.

[183] D. Louis-Régis. Certification de programmes bsml avec juxtaposition parallèle. Master's thesis, Université d'Orléans (LIFO), 2005.

[184] F. Loulergue. BSML : Programmation BSP purement fonctionnelle. In D. Méry and G.-R. Perrin, editors, *Dixièmes Rencontres Francophones du Parallélisme (Renpar'10)*, pages 243–246, Strasbourg, june 1998.

[185] F. Loulergue. Extension du BSλ-calcul. In P. Weis, editor, *JFLA'99 : Journées Francophones des Langages Applicatifs*, pages 93–112, Morzine-Avoriaz, February 1999.

[186] F. Loulergue. BSλ_p : Functional BSP Programs on Enumerated Vectors. In J. Kazuki, editor, *International Symposium on High Performance Computing*, number 1940 in Lecture Notes in Computer Science, pages 355–363. Springer, October 2000.

[187] F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, 6(4) :423–437, 2001.

[188] F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, editor, *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect Books, 2001.

[189] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In *14th IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.

[190] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In H. Kosch, L. Boszorményi, and H. Hellwagner, editors, *Euro-Par 2003*, number 2790 in LNCS, pages 781–788. Springer Verlag, 2003.

[191] F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS 2003), Part III*, number 2659 in LNCS, pages 223–232. Springer Verlag, june 2003.

[192] F. Loulergue. A Calculus of Functional BSP Programs with Explicit Substitution. In G. Joubert, W. Nagel, F. Peters, and W. Walter, editors, *Parallel Computing : Software Technology, Algorithms, Architectures and Applications, Proceeding of the 10th ParCo Conference*, Dresden, 2004. North Holland/Elsevier.

[193] F. Loulergue. Communication Primitives for Minimally Synchronous Parallel ML. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2004), Part I*, LNCS, pages 411–414. Springer Verlag, 2004.

[194] F. Loulergue. Management of Communication Environments for Minimally Synchronous Parallel ML. In *DAPSYS 2004*. Kluwer, 2004. to appear.

[195] F. Loulergue. *Programmation fonctionnelle d'ordinateur parallèles et de méta-ordinateur : sémantique, systèmes et preuves. HDR*. PhD thesis, Université de Paris XII Val-de-Marne, 2004.

[196] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3) :253–277, 2000.

[197] J. M. R. Martin and A. Tiskin. BSP modelling a two-tiered parallel architectures. In B. M. Cook, editor, *WoTUG'90*, pages 47–55, 1999.

[198] J. M. R. Martin and A. V. Tiskin. Dynamic BSP : towards a flexible approach to parallel computing over the grid. In I. East et al., editor, *Communicating Process Architectures : Proceedings of CPA*, pages 219–226. IOS Press, 2004.

[199] B. Di Martino, A. Mazzeo, M. Mazzocca, and U. Villano. Parallel program analysis and restructuring by detection of point-to-point interaction patterns and their transformation into collective communication constructs. *Science of Computer Programming*, 40(2-3) :235–263, 2001.

[200] M. Mauny. *Compilation des langages fonctionnels dans les combinateurs catégoriques. Application au langage ML*. PhD thesis, Université de Paris 7, 1985.

[201] W F McColl. Scalable parallel computing : A grand unified theory and its practical development. In B Pehrson and I Simon, editors, *Proc. 13th IFIP World Computer Congress. Volume 1 (Invited Paper)*. Elsevier, 1994.

[202] W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today : Recent Trends and Developments*, number 1000 of *LNCS*, pages 46–61. Springer-Verlag, 1995.

[203] W. F. McColl. Scalability, portability and predictability : The BSP approach to parallel programming. *Future Generation Computer Systems*, 12 :265–272, 1996.

[204] W. F. McColl. Universal computing. In L. Bouge and al., editors, *Proc. Euro-Par '96*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.

[205] M. Medidi and N. Deo. Parallel dictionaries using avl trees. *J. Parallel Distrib. Comput.*, 49(1) :146–155, 1998.

[206] Kurt Mehlhorn and Stefan Näher. Leda : a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1) :96–102, 1995.

[207] A. Merlin and G. Hains. La Machine Abstraite Catégorique BSP. In *Journées Francophones des Langages Applicatifs*. INRIA, 2002.

[208] A. Merlin, G. Hains, and F. Loulergue. A SPMD Environment Machine for Functional BSP Programs. In *Proceedings of the Third Scottish Functional Programming Workshop*, august 2001.

[209] Q. Miller. BSP in a Lazy Functional Context. In S. Gilmore, editor, *Trends in Functional Programming, Volume 3*, pages 1–13. Intellect Books, 2002.

[210] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1) :1–40 and 41–77, September 1992.

[211] P. Morin. Coarse grained parallel computing on heterogeneous systems. In *ACM Symposium on Applied Computing (SAC'98)*, pages 628–634. ACM Press, 1998.

[212] K. Munagala and A. Ranade. I/O Complexity of Graph Algorithms. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.

[213] Z. Nemeth and V. Sunderam. Characterizing grids : Attributes, definitions, and formalisms. *Journal of Grid Computing*, 1 :9–23, 2003.

[214] The Nuprl Proof Assistant. Web pages at `www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.

[215] C. Okasaki. *Purely Functional Data-Structures*. Cambridge University Press, 1998.

[216] Bruno Pagano. *Des calculs de substitution explicite et de leur application à la compilation des langages fonctionnel*. PhD thesis, Université Pierre et Marie Curie (Paris 6), 1997.

[217] P. Panangaden and J. Reppy. The essence of concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science. Springer, 1996.

[218] C. Parent. Developing certified programs in the system coq : The program tactic. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, LNCS, pages 291–312. Springer, Berlin, Heidelberg, 1993.

[219] C. Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur*. PhD thesis, Habilitation à Diriger des Recherches, Université Paris-Sud Orsay, 1996.

[220] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15 :607–640, 1993.

[221] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[222] A. Plaat, H. E. Bal, and al. Sensitivity of Parallel Applications to large differences in bandwidth and latency in two-layer interconnects. *Futur Generation Computer Systems*, 2004. to appear.

[223] E. Polonovski. *Substitutions explicites, logiques et normalisation*. PhD thesis, Université de Paris 7, 2004.

[224] F. Pottier and V. Simonet. Information Flow Inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1) :117–158, January 2003.

[225] The PVS Specification and Verification System. Web pages at `pvs.csl.sri.com`.

[226] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[227] X. Rebeuf. *Un modèle de coût symbolique pour les programmes parallèles asynchrones à dépendances structurées*. PhD thesis, Université d'Orléans, LIFO, 2000.

[228] B. Reistad and D. K. Gifford. Static dependent costs for estimating program execution time. In *Conference on Lisp and Functional Programming*, 1994.

[229] D. Rémy. Using, Understanding, and Unravellling the OCaml Language. In G. Barthe, P. Dyjber, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 413–536. Springer, 2002.

[230] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[231] J. L. Roda, C. Rodríguez, D. G. Morales, and F. Almeida. Predicting the execution time of message passing models. *Concurrency : Practice and Experience*, 11(9) :461–477, 1999.

[232] C. Rodriguez, J.L. Roda, F. Sande, D.G. Morales, and F. Almeida. A new parallel model for the analysis of asynchronous algorithms. *Parallel Computing*, 26 :753–767, 2000.

[233] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6) :409–424, 1998.

[234] J. M. Del Rosario and A. Choudhary. High performance I/O for massively parallel computers : Problems and prospects. *IEEE Computer*, 27(3) :59–68, 1994.

[235] A.L. Rosenberg. Optimal sharing of partitionable workloads in heterogeneous networks of workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 413–419. CSREA Press, 2000.

[236] J. Schmid, E. Börger, and R. Stark. *Java and the JVM : Definition, Verification, Validation*. Springer Verlag, 2001.

[237] P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Zappa Nardelli, P. Habouzit, and V. Vafeiadis. *Acute : high-level programming language design for distributed computation. Design rationale and language definition*. University of Cambridge Computer Laboratory and INRIA Rocquencourt (Moscova project), 2004. http ://www.cl.cam.ac.uk/users/pes20/acute/acute2-long.pdf.

[238] J. F. Sibeyn and M. Kaufmann. BSP-Like External-Memory Computation. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *LNCS*, pages 229–240. Springer-Verlag, 1997.

[239] D. B. Skillicorn. *Foundations of Parallel Programming*. Number 6 in International Series on Parallel Computation. Cambridge University Press, 1994.

[240] D. B. Skillicorn. Building BSP Progams Using the Refinement Calculus . In *Formal Methods for Parallel Programming and Applications workshop at IPPS/SPDP'98*, 1998.

[241] D. B. Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. Optimising data-parallel programs using the BSP cost model. In *Europar'98*, volume 1470 of *LNCS*, pages 698–715. Springer Verlag, 1998.

[242] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.

[243] David Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3) :249–274, Fall 1997.

[244] L. Smarr and C. E. Catlett. Metacomputing. *CACM*, 35(6) :44–52, 1992.

[245] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[246] A. Stewart and M. Clint. BSP-style Computation : a Semantic Investigation. *The Computer Journal*, 44(3) :174–185, 2001.

[247] A. Stewart, M. Clint, and J. Gabarró. Axiomatic Frameworks for Developing BSP-Style Programs. *Parallel Algorithms and Applications*, 14 :271–292, 2000.

[248] A. Stewart, M. Clint, and J. Gabarró. Algebraic Rules for Reasoning about BSP Programs. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation : Theory and Practice. Nova Science Publishers, august 2002.

[249] J. Sérot. Tagged-token data-flow for skeletons. *Parallel Processing Letters*, 11(4) :377–392, 2001.

[250] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the skipper project. *Parallel Computing*, 28(12) :1785–1808, 2002.

[251] J. Sérot, D. Ginhac, R. Chapuis, and J.P. Dérutin. Fast prototyping of parallel vision applications using functional skeletons. *Journal of Mahcine Vision and Applications*, 12(6) :271–290, 2001.

[252] W. Taha. A gentle introduction to multi-stage programming. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, number 3016 in LNCS, pages 30–50. Springer-Verlag, 2004.

[253] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. *Information and Computation*, 111(2) :245–296, June 1994.

[254] The Coq's team. The Coq Proof Assistant (version 8.0). Web pages at `http://coq.inria.fr`, 2004.

[255] R. Thakur, E. Lusk, and W. Gropp. I/O characterization of a portable astrophysics application on the ibm sp and intel paragon. Technical Report MCS-P534-0895, Argonne National Laboratory, October 1995.

[256] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. Framework for adaptive algorithm selection in stapl. In *ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, 2005. to appear.

[257] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998.

[258] A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, 11(4) :409–422, 2001.

[259] J. L. Traeff and J. Worringen. Verifying collective mpi calls. In *Proceedings of the 11th EuroPVM/MPI conference*, LNCS. Springer, 2004.

[260] P.W. Trinder and all. GPH : An Architecture-independent Functional Language. *IEEE transactions on Software Engineerig*, 1999.

[261] J. D. Ullman and M. Yannakakis. The I/O Complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3(4) :331–360, 1991.

[262] J. Vachon. Une analyse statique pour le contrôle des effets de bords en Caml-Flight beta. In C. Queinnec, V. V. Donzeau-Gouge, and P. Weis, editors, *JFLA*, number 13 in Collection Didactique. INRIA, Janvier 1995.

[263] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, August 1990.

[264] P. B. Vasconcelos and K. Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *IFL'02*, LNCS, pages 110–125. Springer Verlag, 2003.

[265] Vasil P. Vasilev. BSPGRID : Variable Resources Parallel Computation and Multiprogrammed Parallelism. *Parallel Processing Letters*, 13(3) :329–340, 2003.

[266] D. E. Vengroff and J. S. Vitter. Supporting I/O-efficient Scientific Computation in TPIE. In *IEEE Symposium on Parallel and Distributed Computing*, 1995.

[267] J. S. Vitter. External memory algorithms and data structures : Dealing with massive data. *ACM Computing Surveys*, 33(2) :209–271, June 2001.

[268] J.S. Vitter. External memory algorithms. In *ACM Symp. Principles of Database Systems*, pages 119–128, 1998.

[269] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory, two -level memories. *Algorithmica*, 12(2) :110–147, 1994.

[270] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2 :461–493, 1992.

[271] B. Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université de Paris VII, 1994.

[272] T. L. Williams and R. J. Parsons. The heterogeneous bulk-synchronous parallel model. In J. Rolim and al., editors, *IPDPS*, number 1800 in LNCS, pages 102–108. Springer-Verlag, 2000.

[273] T. L. Williams and R. J. Parsons. Exploiting hierarchy in heterogeneous environments. In *IEEE/ACM IPDPS'2001*, pages 140–147. IEEE press, 2001.

[274] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1993.

[275] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1975.

[276] X. Leroy and D. Doligez and J. Garrigue and D. Rémy and J. Vouillon. The Objective Caml System release 3.08.3. Web pages at `http://caml.inria.fr`, 2005.

[277] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, january 1999. page web `http://www.cs.bu.edu/~hwxi/DML/DML.html`.

[278] K. A. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, and C. Wen. Parallel data structures for symbolic computation. In *Workshop on Parallel Symbolic Languages and Systems*, 1995.

[279] A. Al Zain, P. Trinder, H.-W. Loidl, and G. Michaelson. Managing heterogeneity in a grid parallel haskell. In V. Sunderam, D. van Albada, P. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2005)*, LNCS. Springer, 2005.

[280] A. Zavanella. *Skeletons and BSP : Performance Portability for Parallel Programming*. PhD thesis, Universita degli studi di Pisa, 1999.

[281] W. Zimmermann, M. Middendoff, and W. Loewe. On optimal kappa-linear scheduling of tree-like task graphs for LogP-Machines. *Lecture Notes in Computer Science*, 1470 :328– ? ?, 1998.

Résumé. Certains problèmes nécessitent des performances que seules les machines massivement parallèles ou les méta-ordinateurs peuvent offrir. L'écriture d'algorithmes pour ce type de machines demeure plus difficile que pour celles strictement séquentielles et la conception de langages adaptés est un sujet de recherche actif nonobstant la fréquente utilisation de la programmation concurrente. En effet, la conception d'un langage de programmation est le résultat d'un compromis qui détermine l'équilibre entre les différentes qualités du langage telles que l'expressivité, la sûreté, la prédiction des performances, l'efficacité ou bien la simplicité de la sémantique.

Dans des travaux antérieurs à cette thèse, il a été entrepris d'approfondir la position intermédiaire que le paradigme des patrons occupe. Toutefois il ne s'agissait pas de concevoir un ensemble *a priori* fixé d'opérations puis de concevoir des méthodologies pour la prédiction des performances, mais de fixer un modèle de parallélisme structuré (avec son modèle de coûts) puis de concevoir un ensemble universel d'opérations permettant de programmer n'importe quel algorithme de ce modèle. L'objectif est donc le suivant : parvenir à la conception de langages universels dans lesquels le programmeur peut se faire une idée du coût à partir du code source.

Cette thèse s'inscrit dans le cadre du projet «CoordinAtion et Répartition des Applications Multiprocesseurs en objective camL» (CARAML) de l'ACI GRID dont l'objectif était le développement de bibliothèques pour le calcul haute-performance et globalisé autour du langage OCaml. Ce projet était organisé en trois phases successives : sûreté et opérations data-parallèles irrégulières mono-utilisateur ; opérations de multi-traitement data-parallèle ; opérations globalisées pour la programmation de grilles de calcul.

Ce tapuscrit est organisé en 3 parties correspondant chacune aux contributions de l'auteur dans chacune des phases du projet CARAML : une étude sémantique d'un langage fonctionnel pour la programmation BSP et la certification des programmes écrits dans ce langage ; une présentation d'une primitive de composition parallèle (et qui permet aussi la programmation d'algorithmes «diviser-pour-régner» parallèles), un exemple d'application *via* l'utilisation et l'implantation de structures de données parallèles et une extension pour les entrées/sorties parallèles en BSML ; l'adaption du langage pour le méta-calcul.

Mots clefs. Programmation fonctionnel parallèle, sémantiques formelles, méta-calul, Objective Caml, modèles de coûts, certification, Coq.

Abstract. Some problems require performances that can only be provided by massively parallel machines and meta-computers. Nevertheless, algorithm writing for this kind of machine remains more difficult than for those strictly sequential and the conception of adapted languages is an active subject of research notwithstanding the frequent use of the concurrent programming. Indeed, the conception of a programming language is the result of a compromise which determines balance between various qualities of the language such as expressivity, safety, effectiveness or the simplicity of the semantics.

In the prior works to this thesis, it was undertaken to study, with greater details, the intermediate position that the paradigm of the skeletons occupies. However, this did not involve conceiving an *a priori* fixed sets of operations and then to design performance models, but, to fix a structured parallelism model (with its performance model) and then to design a universal number of operations making it possible to program any algorithm of this model. The objective is therefore as follows : have an universal language in which the programmer can have an idea of the cost from the source code.

This thesis comes under the project "CoordinAtion and Distribution of Multiprocessor Applications in objective camL" (CARAML) of the ACI GRID. The objective was the development of libraries for high-performance and globalised computations around the OCaml language. This project was organized in three successive phases : safety and irregular data-parallel operations (mono-user) ; data-parallel multiprocessing operations ; globalised operations for grid computing.

The thesis follows the organization in three phases of the CARAML project and presents the contributions of the author in each phase in the three parts : a semantic study of a functional language for BSP programming and the certification of the written programs in this language ; a presentation of a primitive of parallel composition (and which also allows programming parallel "divide-and-conquer" algorithms), an example of application using an implementation of parallel data structures and an extension for parallel input/outputs in BSML ; the adaption of the language for meta-computing.

Keywords. Parallel functional programming, formal semantics, meta-computing, Objective Caml, cost models, certification, Coq.