Génération de code réparti par distribution de données

Jean-Louis Pazat

HABILITATION A DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Institut de Formation Supérieure
en Informatique et en Communication

par

Jean-Louis Pazat

GÉNÉRATION DE CODE RÉPARTI PAR DISTRIBUTION DE DONNÉES

soutenue le 27 novembre 1997 devant le jury composé de

M.	Jean-Pierre Banâtre	Président
MM.	Luc Bougé	Rapporteurs
	Paul Feautrier	
	Guy René Perrin	
MM.	Françoise André	Examinateurs
	Jean-Marc Geib	
	Claude Jard	
	Thierry Priol	

A Marc Le Fur

Remerciements

Je remercie Jean-Pierre Banâtre, directeur de l'IRISA pour avoir accepté de présider ce jury et pour m'avoir encouragé à préparer ce document.

Luc Bougé, Paul Feautrier et Guy-René Perrin me font l'honneur d'être rapporteurs de ce travail et je les en remercie ; leurs remarques m'ont aidé à améliorer ce document. C'est avec grand plaisir que j'ai participé au "thème 1.1 du GDR/PRC Parallélisme, Réseaux et Systèmes (PRS)", un nom bien barbare pour un groupe de travail sympathique qui a permis des échanges fructueux dans la communauté française du parallélisme de données et même au delà. Je crois que ces groupes de travail sont essentiels pour que la recherche scientifique progresse. Luc Bougé a co-dirigé PRS et a soutenu nos actions, je le remercie de m'avoir confié l'animation du nouveau thème "grappes". Guy-René Perrin a animé le "thème 1.1" qui a été un endroit d'échanges important. Je remercie aussi Paul Feautrier pour sa présence aux réunions de travail et pour ses exposés ardus mais fort pédagogiques.

Je remercie Jean-Marc Geib pour sa participation à ce jury, j'espère qu'il trouvera de l'intérêt aux perspectives de ces travaux.

Je remercie Claude Jard, responsable du projet PAMPA pour la confiance qu'il m'accorde en participant à ce jury.

Je suis très heureux que Thierry Priol qui est devenu un spécialiste des machines parallèles à l'INRIA soit dans ce jury.

C'est grâce à Françoise André que ces travaux ont pu se faire, d'abord parce qu'elle a eu l'initiative du projet PANDORE et ensuite parce qu'elle m'a fait confiance dès mon arrivée, ce qui m'a permis très vite de participer à l'encadrement des doctorants. Françoise André a toujours su encourager vivement les personnes qui ont travaillé avec elle et c'est sur son initiative que le projet PAMPA a pu se créer. Sous sa direction, de nombreuses activités de recherche fructueuses ont pu y être menées de manière cohérente.

C'est lors de mon DEA sous la direction de Françoise André dans le projet API que j'ai découvert ce qu'était un travail de recherche ; je remercie Patrice Quinton qui était responsable du projet API de m'avoir encouragé à poursuivre une thèse.

Lors de mon séjour à Bordeaux j'ai apprécié l'esprit d'équipe et la disponibilité de tous les membres du groupe de travail Cheops : Marie-Christine Counilh, Jean-Michel Lépine, Jean Roman, Franck Rubi et bien sûr Bernard Vauquelin qui a encadré mon travail de thèse. Les discussions que j'ai pu avoir avec André Arnold m'ont éclairé sur les notions de parallélisme et de non déterminisme.

Un travail comme celui qui est relaté dans ce document est avant tout un travail

d'équipe. De nombreux étudiants de DEA et des stagiaires ont contribué à ce travail, il ne m'est pas possible de les citer tous ici, mais leur concours à été très utile.

Les doctorants qui ont contribué à ce travail sont (par ordre chronologique) Henry Thomas, Olivier Chéron, Yves Mahéo, Marc Le Fur et Pascale Launay dont le travail est relaté dans la conclusion de ce document. C'est tout d'abord Henry Thomas qui a eu la lourde responsabilité de faire la première thèse sur PANDORE; il a fait partie des "pionniers" de la distribution de données dans le monde. Olivier Chéron a transformé le prototype d'Henry en un outil "modulairextensible" que Yves Mahéo et Marc Le Fur ont étendu. Ils sont pour moi bien plus que de simples thésards `<ESC><BS>` doctorants. J'ai bien sûr une pensée particulière pour Marc, décédé en Juin 96 dans un accident de plongée.

Je remercie tous les membres de l'équipe PAMPA (d'hier et d'aujourd'hui) pour l'esprit amical qui y règne.

Je remercie Jean Camillerapp, directeur du département informatique de l'INSA et de tous mes collègues enseignants-chercheurs pour leur soutien.

Table des matières

Avant propos

Démarche scientifique

Le but de mes travaux de recherche est de fournir des outils (ou les éléments permettant de les construire) pour faciliter la conception raisonnée et la mise en œuvre correcte de programmes efficaces et adaptables. Par efficace j'entends ici "qui utilise au mieux les ressources de la machine" ; dans ce document j'utiliserai l'expression "efficace en temps" lorsque le but est d'obtenir une durée d'exécution la plus courte possible et j'utiliserai l'expression "efficace en espace" pour dire que l'utilisation des ressources mémoire est optimisée. "Adaptable", signifie ici la possibilité pour un programme de pouvoir s'exécuter sur des machines différentes soit sans aucune modification, soit par des modifications automatisées qui ne remettent pas en cause la structure du logiciel. Dire que cette conception doit être raisonnée suppose que le modèle de programmation utilisé est suffisamment abstrait pour être adapté au problème que le programmeur doit résoudre. Enfin je qualifie une mise en œuvre de correcte si il existe une certaine équivalence[1] entre le comportement du programme et sa spécification.

On verra dans ce document que j'ai pris comme hypothèse de travail "un modèle d'exécution ≡ un langage de programmation" ; comme si un langage de programmation ne possédait qu'une seule sémantique opérationnelle (une seule mise en œuvre). Cette hypothèse n'est pas vérifiée pour de nombreux langages (en particulier les langages fonctionnels et les langages de programmation logique) mais la programmation impérative séquentielle par laquelle j'ai abordé l'informatique s'appuie très fortement sur une sémantique opérationnelle intuitive, le modèle de programmation étant très proche du modèle d'exécution de Von Neumann. Même dans le cadre des langages impératifs, cette hypothèse de travail peut être considérée comme un peu restrictive; on peut par exemple encapsuler soigneusement un modèle d'exécution (donc une sémantique opérationnelle) dans des bibliothèques et ne présenter qu'une interface au programmeur qui est compatible avec une autre sémantique opérationnelle. En particulier dans le cadre d'un langage à objets, il a été montré que l'on pouvait encapsuler un modèle d'exécution parallèle SPMD dans un ensemble de classes qui peuvent s'utiliser "comme si" les méthodes invoquées réalisaient les opérations séquentiellement [Jézéquel 93]. L'intérêt de cette hypothèse de travail est de séparer clairement la vision du programmeur (une sémantique opérationnelle intuitive qui est très proche du modèle de programmation impératif que nous fournissons) de celle qui est imposée par l'utilisation d'une machine parallèle et de son système d'exploitation (le modèle d'exécution).

Le cadre de mes travaux est l'utilisation des machines parallèles à mémoire distribuée. Une des premières machines à mémoire distribuée (l'iPSC 1 d'Intel) à été installée à Rennes pendant mon DEA, je dois dire que nous avons vécu cela comme le début d'une grande aventure, car tout était à inventer sur ce type de machine. Lors de ma thèse [Pazat 89b] j'ai contribué au projet de conception d'une machine à mémoire distribuée (CHEOPS) à Bordeaux, projet dont la partie "matérielle" n'a

1. Nous définirons cette équivalence en temps opportun.

malheureusement pas abouti. Néanmoins les travaux relatifs à CHEOPS ont eu une suite notamment les études sur le placement de tâches [Pellegrini 95].

Les travaux présentés ici ont pour ambition de contribuer à la maîtrise de la programmation d'applications parallèles et réparties. Plus précisément, le problème qui est posé ici est de permettre une utilisation simple et efficace des architectures parallèles à mémoire distribuée.

Organisation du document

Ce document relate le travail que nous avons mené sur le projet PANDORE depuis octobre 1989. Le chapitre suivant replace ces travaux dans la problématique de la programmation des architectures parallèles à mémoire distribuée.

Dans le projet PANDORE, nous avons tout d'abord défini des extensions à un langage de programmation permettant de spécifier la distribution de données. L'expression de la distribution dans les langages est décrire au chapitre 2.

Des règles de transformation de programmes utilisant la distribution de données pour générer automatiquement du code parallèle et distribué ont été définies et ont donné lieu à la réalisation d'un premier compilateur. Dans cette première approche décrite au chapitre 3, nous nous sommes concentrés sur l'aspect systématique de la génération de code en effectuant une transformation instruction par instruction.

Afin de valider l'intérêt de la génération de code par distribution de données dans le cadre du calcul hautes performances, nous avons étudié la génération de code réparti ayant une efficacité (en temps et en espace) comparable à celle d'un code écrit "à la main". Ce travail de recherche est relaté au chapitre 4.

Les perspectives de ces travaux sont regroupées dans le dernier chapitre de ce document.

Chapitre 1

Introduction

1.1 La programmation de des calculateurs parallèles à mémoire distribuée

1.1.1 Les calculateurs parallèles à mémoire distribuée

Les calculateurs parallèles à mémoire distribuée sont des machines composées de nœuds de calcul reliés par un réseau de communication. Les nœuds de calcul des machines que nous considérons ici comprennent une mémoire unique qui n'est accessible directement que par un processeur de calcul (comme sur le Paragon d'Intel). L'interface entre les nœuds et le réseau de communication permet de réaliser des échanges entre les mémoires de nœuds différents sous le contrôle des processeurs locaux à chacun de ceux-ci. Le réseau de communication permet de réaliser des échanges point à point entre les nœuds et parfois des échanges globaux ou semi globaux (diffusion globale et diffusion multiple). Les architectures que nous considérons sont des architectures MIMD (*Multiple Instruction stream, Multiple Data stream*) pour lesquelles chaque nœud de calcul possède son propre compteur de programme.

Ces architectures permettent d'atteindre de très grandes performances. Le Cray T3E-900 par exemple, peut atteindre 1.8 teraflops (10^{12} opérations de calcul flottant par seconde) en utilisant plusieurs milliers de processeurs ; chaque processeur peut recevoir jusqu'à 2 Go de mémoire et le réseau d'interconnexion de cette machine a une bande passante supérieure au Go/s.

Il existe d'autres types d'architectures parallèles :

- les architectures *dataflow* pour lesquelles le séquencement des opérations est dirigé par les données ;
- les architectures de type SIMD (*Single Program Multiple Data*) pour lesquelles l'exécution des instructions est synchrone (il n'existe qu'un seul compteur de programme) ;
- les architectures parallèles MIMD à mémoire partagée dans lesquelles plusieurs processeurs peuvent physiquement accéder à une même mémoire.

Nous n'avons pas considéré ici les architectures dataflow car celles-ci n'ont jamais dépassé le stade de prototypes ni les architectures SIMD d'une part parce que ces dernières ont un champ d'application limité et d'autre part parce qu'elles tendent à disparaître.

Le cas des architectures à mémoire partagée est particulier. A la fin des années 80 il était admis que ces architectures n'avaient aucun avenir car elles étaient chères et peu extensibles. Depuis, l'augmentation de la puissance de calcul des processeurs fait que les architectures multiprocesseurs faiblement parallèles (de 2 à 8 processeurs) sont devenues intéressantes pour de nombreuses applications. Nous n'avions donc pas considéré ces architectures au début de notre étude et il faudrait maintenant étudier l'adaptation de nos techniques à ces architectures, et surtout à des réseaux de machines de ce type comme le Power Challenge de Silicon Graphics.

Il est également possible d'obtenir des puissance de calcul moindres mais pour un coût modéré par rapport à des architectures spécifiques en interconnectant des stations de travail de type PC par un réseau rapide (jusqu'à 1Gbit/s pour le réseau Myrinet, environ 100 Mbit/s avec Fast Ethernet). A titre d'exemple, il est possible d'obtenir une puissance[1] de l'ordre du GFlops avec un réseau de 16 PC (pentium P6 a 200 MHz sous Linux) interconnectés par Fast Ethernet. Ce type d'architecture est en constant développement et sera probablement beaucoup plus largement utilisé que les machines spécifiquement conçues pour le calcul parallèle.

Dans la suite nous considérons comme support d'exécution des machines parallèles à mémoire distribuée MIMD munies d'un système d'exploitation qui offre des possibilités de communication par échange de messages.

1.1.2 Les domaines d'application

La maîtrise de la programmation de telles architectures est crucial dans de nombreux domaines d'activité et avant tout pour le calcul "hautes performances". D'autres domaines peuvent également profiter des progrès faits dans la programmation des calculateurs à mémoire distribuée.

Calcul hautes performances

Les machines les plus puissantes étant actuellement des architectures à mémoire distribuée, la programmation de ces machines est incontournable pour les applications nécessitant une très grande puissance de calcul ou une très grande taille mémoire. La simulation qui est très largement utilisée en physique quantique, en dynamique moléculaire ou en mécanique des fluides, est particulièrement "gourmande" en temps de calcul et en espace mémoire. Des applications de cette nature, comme les prévisions météorologiques ou la simulation de l'évolution du climat représentent les grands défis pour la recherche et sont également réputées stratégiques (parfois même au sens militaire du terme). Dans ce cadre, même si le critère essentiel est la performance du code produit, il est nécessaire de maîtriser les coûts de

1. réellement mesurée sur un calcul

développement, de mise au point et de portage de ces applications. Les architectures à hautes performances évoluant très vite, le temps consacré au portage d'un code ne doit pas dépasser la durée de vie d'une machine ! De plus il faut s'assurer que le portage est correct, c'est-à-dire que l'on n'a pas introduit d'erreurs lors de l'adaptation de l'application. Deux facteurs me paraissent alors essentiels : assurer au mieux la correction des portages des codes et permettre une grande adaptabilité des logiciels aux architectures.

Les architectures à mémoire distribuée étant souvent moins chères que d'autres architectures parallèles, leur utilisation dans des entreprises ou dans des laboratoires de recherche (non informatique !) est actuellement largement envisagée. La possibilité de migration de code séquentiel vers des architectures parallèles à été démontrée sur des applications réelles dans le cadre de projets européens (projets Europort I et II en particulier). Des applications aussi diverses que la simulation de réservoirs de pétrole, la synthèse de molécules ou le coloriage de dessins animés ont été portées sur des machines parallèles dont un un grand nombre étaient des machines à mémoire distribuée. Cependant, une utilisation massive de ces architectures est encore freinée par les difficultés de programmation et la nécessité de disposer d'un expert pour réaliser le portage. Le coût de développement et de portage des applications doit ici être impérativement minimisé pour que l'utilisation de ces architectures soit réaliste.

Autres domaines d'application

Les architectures utilisées dans le cadre du développement d'outils communicants (automobile, domotique, bureautique communicante) sont des architectures réparties qui offrent un modèle d'exécution très proche de celui des architectures parallèles à mémoire distribuée. Les réseaux de machines (locaux ou à grande distance comme l'Internet) qui constituent le support naturel des applications réparties (interactives ou non) peuvent également rentrer dans ce cadre. Là encore, le développement des applications est limité par les difficultés de conception et de maîtrise de la programmation de ces architectures.

Dans le cadre des applications interactives (travail coopératif, télévision interactive, WWW, ...) les seules applications actuellement largement développées sont basées sur des modèles limitatifs (souvent le client/serveur). L'essor de ces nouveaux outils passe là aussi par la maîtrise de la conception de logiciels répartis complexes. A ce titre, une action de recherche (Rusken) concernant le travail coopératif dont un des objectifs est de favoriser la maîtrise des techniques et outils en matière de travail coopératif est actuellement en cours à l'IRISA.

1.1.3 Quelques difficultés liées à l'utilisation de ces machines

L'utilisation habituelle des architectures parallèles à mémoire distribuée consiste à se référer à un modèle de programmation qui est seulement un sous-ensemble du modèle d'exécution fourni par la machine (et son système d'exploitation) [Bougé 96]. Cette utilisation cumule les difficultés intrinsèques au parallélisme et celles qui sont introduites par la répartition et la communication.

Le parallélisme, l'aspect distribué de la mémoire et la communication entre les processeurs doivent être prise en compte ensemble ce qui rend très vite la programmation de ces machines "cauchemardesque". Le découpage d'une application en tâches doit tenir compte du nombre de processeurs de la machine cible ; de ce découpage dépendra le placement des données dans les mémoires locales et donc la structure des communications. Une modification de l'une quelconque de ces étapes remettra en cause les choix d'implémentation et parfois même l'architecture du logiciel. Même si l'on effectue un paramétrage "forcené" du logiciel (qui le rendra illisible), l'adaptation d'un programme sera difficile car il faudra de nouveau faire un grand nombre de tests sur le logiciel.

Il apparaît donc nécessaire de définir un modèle de programmation de plus haut niveau pour s'affranchir des problèmes liés au parallélisme, à la répartition et à la communication. Malheureusement il n'est pas possible de masquer l'ensemble de ces problèmes dans tous les cas.

Le parallélisme : Dans le cadre d'applications intrinsèquement parallèles (outils communicants, applications interactives), le parallélisme est une donnée intéressante du problème et doit donc faire partie du modèle de programmation. De nombreux ouvrages, algorithmes, outils et même langages de programmation développés dans les années 70-80 nous aident à maîtriser les problèmes liés au parallélisme. Nous avons ainsi les moyens d'exprimer le parallélisme et de mettre en œuvre les synchronisations à condition de faire abstraction de l'aspect distribué de la mémoire.

Au contraire, lorsque la seule raison pour utiliser ces machines est d'accélérer la vitesse de traitement d'un algorithme, le parallélisme d'exécution n'est en général pas celui qui doit être exprimé dans le modèle de programmation.

L'aspect distribué de la mémoire : Sur une machine à mémoire distribuée, la mémoire est composée de l'ensemble des mémoires locales des nœuds ; elle n'est donc pas contiguë et son accès n'est pas uniforme. Ceci oblige à placer, voire à découper les structures de données des programmes en fonction de l'organisation physique de la mémoire. Il devient alors nécessaire de prendre en compte la localisation des données auxquelles on accède dans le programme les accès aux données selon leur localisation physique. Lorsque le choix de la distribution est lié à des critères de performance, il est souvent préférable que les éléments auxquels une tâche accède soient placés dans la mémoire associée au processeur qui supporte l'exécution de la tâche.

Pour des applications intrinsèquement réparties comme le travail coopératif ceci est un problème que l'on peut raisonnablement considérer comme lié à l'application ; l'aspect distribué de la mémoire doit donc faire partie du modèle de programmation.

Par contre, lorsque l'utilisation de ces machines est uniquement lié à des critères de performances, ces considérations sont sans aucun lien avec la logique du programme et be devraient pas faire partie du modèle de programmation.

La communication : Lorsque l'on doit faire communiquer des tâches pour assurer leur coopération sur une machine parallèle à mémoire distribuée, le type de

communication dépend des possibilités offertes par le système d'exploitation et le matériel sous-jacent. Lorsque les tâches sont situées sur un même nœud il peut être possible de leur faire partager des variables, par contre, lorsque les tâches sont situées sur des processeurs différents, la communication se fait le plus souvent par échange de messages[2]. On doit alors prendre en compte dans le programme le fait que la communication n'est pas instantanée. Le délai qui sépare l'envoi de la réception d'un message est observable et même d'un ou plusieurs ordres de grandeur supérieur au temps d'exécution d'une instruction.

Là encore pour des applications de nature répartie, on peut souhaiter garder la communication dans le modèle de programmation. Néanmoins l'expression de la communication devrait être la plus indépendante possible de la machine et du placement relatif des tâches.

Dans le cadre du calcul hautes performances où les communications ne font pas partie de la logique de l'application, les communications doivent être implicites dans le modèle de programmation.

1.2 Éléments de réponse

Je pense que le modèle de programmation idéal pour le calcul hautes performances doit fournir une abstraction de l'aspect fragmenté de la mémoire et du parallélisme de la machine, il doit être facile à appréhender et bien fondé (avoir une sémantique bien définie).

La différence entre modèle de programmation et modèle d'exécution dépend du niveau où l'on se place : un modèle de programmation peut être implanté directement et il n'y a alors pas de différence avec son modèle d'exécution (c'est le cas lorsque l'on programme en assembleur par exemple) ; il peut également être implanté en passant par un compilateur ou un transformateur de programmes (c'est le cas de la programmation en Pascal par exemple). Enfin, on peut passer par plusieurs niveaux intermédiaires qui peuvent être considérés comme des modèles d'exécution ou des modèles de programmation.

1.2.1 Modèles de programmation et d'exécution

Un programme "parallèle non réparti" comporte la description d'activités parallèles (constructions parallèles par exemple) dans lequel l'expression du parallélisme peut se faire sur le contrôle ou sur les données. L'accès aux données y est uniforme (du point de vue du nommage) et partagé.

- L'expression du parallélisme peut se faire sur le contrôle : il existe plusieurs sous-classes de langages à parallélisme de contrôle (non réparti) où le parallélisme est exprimé par des structures de contrôle (les boucles parallèles, les sections de code parallèle, les coroutines, les tâches, etc.). C'est typiquement

2. Il existe également des lectures et écritures à distance sur certains systèmes comme celui du T3D.

le type de programme que peut générer un paralléliseur ou ce que les langages de type "Fortran parallèle" comme PCF [Forum 88] permettent d'écrire.

- L'expression du parallélisme peut également se faire sur les données : c'est le modèle dit à "parallélisme de données" [Bougé 96]. Il existe plusieurs niveaux d'abstraction dans le parallélisme de données : le niveau macroscopique et le niveau microscopique.
  - Au niveau macroscopique je ferai la distinction entre un niveau "collection" et un niveau explicitement parallèle (SEQ of PAR selon [Bougé 96]). Au niveau "collection", on utilise des opérateurs permettant d'effectuer des opérations globales et des réductions. Ces opérateurs s'appliquent à des structures de données de haut niveau (collections) et il n'est pas nécessaire de préciser l'ordonnancement des calculs sur les éléments eux-mêmes. C'est le cas des expression tableau de Fortran 90 et du `FORALL` de HPF dont la sémantique *copy-in, copy-out* spécifie que les valeurs des variables en partie droite sont celles d'avant l'exécution du `FORALL` (sémantique de multi-affectation). Au niveau explicitement parallèle on précise si ces constructions "data parallèles" peuvent s'exécuter dans n'importe quel ordre (et donc en parallèle) ; c'est par exemple le cas de l'annotation `INDEPENDENT` du langage HPF qui peut porter sur un constructeur `FORALL`. Notons que le cas des boucles parallèles qui ne contiennent qu'une instruction (parallélisme de contrôle) peut être considéré comme une forme d'expression (un peu obscure) de parallélisme de données.
  - le niveau microscopique est un niveau explicitement parallèle qui représente le modèle d'exécution du point de vue du parallélisme (PAR of SEQ).

Enfin, J'appelle "programme réparti" un programme décrivant plusieurs activités (non nécessairement parallèles[3]) qui s'exécutent dans des espaces de nommage différents. C'est le modèle d'exécution habituellement disponible sur les machines parallèles à mémoire distribuée.

Pour résumer nous considérons 4 types de modèles de programmation ou d'exécution :

- le modèle purement séquentiel ;
- le modèle "data parallèle" de niveau "collection" ;
- le modèle "data parallèle" de niveau SEQ of PAR ;
- le modèle réparti.

Nous n'avons pas considéré le cas du modèle parallèle général non réparti (PAR of SEQ) dans une première approche ; par contre nous y reviendrons dans les perspectives.

3. Lors d'un appel de procédure à distance synchrone, du point de vue de l'appelant le programme est bien réparti mais pas parallèle.

1.2.2 Mise en œuvre d'un modèle de programmation

Dans le cadre de la programmation des machines parallèles à mémoire distribuée, on peut mettre en œuvre un modèle de programmation en considérant qu'il n'est qu'un sous ensemble du modèle d'exécution à condition de fournir des mécanismes système ou des bibliothèques permettant de s'affranchir de certaines contraintes comme la distribution de la mémoire ou la limitation du nombre de processeurs. Nous regrouperons ces techniques sous le terme de **virtualisation** puisqu'elles ont pour objectif de fournir une machine virtuelle d'exécution. Le terme de virtualisation est malheureusement aussi utilisé dans un sens différent dans le cadre de la programmation SIMD (pour signifier le placement des processeurs virtuels sur les processeurs physiques).

Une autre manière de résoudre ces problèmes est de séparer le modèle de programmation du modèle d'exécution, ce qui nécessite d'avoir recours à des techniques de **transformation de programmes**. Nous utiliserons ce terme plutôt que celui de compilation pour préciser que ces transformations se font au niveau du programme source[4].

Ces deux solutions peuvent être utilisées conjointement c'est-à-dire que l'on peut chercher à compiler un langage de plus haut niveau vers une "machine virtuelle" un peu plus évoluée que ce que fournissent la plupart des systèmes des machines parallèles à mémoire distribuée.

1.2.3 Techniques de virtualisation

Nous regroupons ici des bibliothèques implantées au niveau "utilisateur" et des outils implantés qu niveau système. Leur but reste identique : fournir une machine virtuelle plus abstraite.

Virtualisation des processeurs

La virtualisation des processeurs consiste à fournir un modèle d'exécution dans lequel le nombre de flots de contrôle à un instant donné ne dépend pas de la configuration matérielle de la machine.

Posix est une norme pour les bibliothèques de *threads* (tâches légères). Celles-ci permettent d'exécuter plusieurs tâches de manière efficace sur un même processeur. Elles permettent un premier niveau de virtualisation mais fournissent un modèle dans lequel les processeurs sont toujours visibles.

Athapascan 0 [Christaller 96] offre une abstraction des processeurs qui permet d'exécuter un programme composé de tâches parallèles grâce à des serveurs qui sont implantés sous forme de tâches légères. La migration de tâches entre des processeurs différents n'est pas possible. Des mécanismes de régulation de charge devront permettre de placer les tâches sur les processeurs les moins chargés.

4. Transformations de haut niveau selon [Bodin 97].

Chant [Haines et al. 94] et **PM²** [Namyst et Méhaut 95] (Parallel Multithreaded Machine) réalisent l'abstraction des processeurs uniquement en gérant des tâches légères sur les processeurs. Par contre, les processeurs sont quand même visibles dans le modèle de programmation fourni. PM² repose sur une bibliothèque de tâches légères optimisée Marcel tandis que Chant utilise une bibliothèque de tâches légères standard non optimisée.

Virtualisation de la mémoire

La virtualisation de la mémoire permet de fournir un modèle de programmation qui fait abstraction de l'aspect fragmenté de la mémoire (modèle parallèle non réparti PAR of SEQ) soit en fournissant un espace d'adressage unique (mémoires virtuellement partagées), soit en fournissant un espace de nommage unique au niveau des objets manipulés par les programmes (mémoires virtuelles d'"objets").

Koan et **Myoan** [Cabillic et al. 94] sont des mémoires virtuellement partagées de niveau système. Le modèle de programmation est la mémoire partagée qui est implantée par un mécanisme de pagination. En outre Koan et Myoan fournissent la possibilité de diffusion explicite de pages entre processeurs et la possibilité d'utiliser plusieurs mécanismes de cohérence. L'intérêt de ce type de virtualisation de la mémoire est discuté au chapitre 4.

Linda [Carriero et al. 86] offre un mécanisme d'accès partagé à un espace de nommage (le *tuple space* dans lequel on peut déposer et retirer des objets par des primitives atomiques de lecture et d'écriture.

DosMos [Brunie et Levèvre 94], **Global Arrays** [Nieplocha et al. 94] et **Cidre** [André et Mahéo 96] sont des bibliothèques de gestion d'"objets" distribués au niveau utilisateur. Ces bibliothèques ont des fonctionnalités très similaires : elles permettent d'allouer un tableau distribué, d'accéder à ses éléments de manière transparente et de gérer la cohérence au niveau d'un bloc ou d'une section de tableau par des appels explicites a des primitives de cohérence (acquire, release). Les différences entre ces bibliothèques se situent plus au niveau de la mise en œuvre que du modèle de programmation fourni. L'exécutif (la machine virtuelle) du compilateur PANDORE est basée sur un sous-ensemble de Cidre spécifiquement optimisé pour servir de cible à notre compilateur.

Virtualisation des communications

les bibliothèques de communications portables permettent de définir un premier niveau de virtualisation des communications car celles-ci sont indépendantes du protocole de communication de bas niveau utilisé. PVM [Geist et al. 94] a été très largement utilisée, y compris sur des réseaux de stations de travail. Des implémentations de MPI [Snir et al. 95] commencent à être largement diffusées et vont probablement remplacer PVM. Ces bibliothèques fournissent essentiellement

l'abstraction de l'envoi de messages non bloquant et de la réception bloquante ou non plus quelques possibilités de communication de groupe.

La bibliothèque POM [Guidec et Mahéo 95a] que nous avons développée dans l'équipe PAMPA offre les fonctionnalités de communication minimales de PVM mais y ajoute la possibilité d'utiliser des horloges globales qui sont très utiles pour analyser des traces d'exécution. De plus il est possible de surveiller une application pendant son déroulement grâce à un nœud observateur. Cette bibliothèque est utilisée par le compilateur PANDORE et par la bibliothèque cidre.

Athapascan 0 [Cavalheiro et Doreille 96] permet le passage de paramètres par valeur aux tâches lors de leur lancement et la récupération de résultats lors de leur terminaison. Des communications de groupe (diffusion, concentration) ont été optimisées pour permettre l'exécution efficace de programmes "data parallèles".

PM^2 a comme intérêt principal de réaliser la virtualisation des communications qui est complètement associée à la virtualisation des processeurs. La base de PM^2 est l'"appel de méthode léger" qui consiste à créer une tâche légère sur un processeur distant pour exécuter un service auquel on passe des paramètres. Les tâches peuvent également migrer d'un processeur à un autre.

1.2.4 Transformations de programmes

On peut classer les transformations de programmes auxquelles nous nous intéressons en deux catégories : les techniques de parallélisation et les techniques de répartition.

Parallélisation de programmes

La "parallélisation" de programmes impératifs regroupe la découverte d'actions parallèles et la transformation d'un programme séquentiel en un nouveau programme parallèle. Pour avoir un intérêt pratique, la parallélisation d'un programme doit préserver une "certaine sémantique", c'est-à-dire qu'il doit exister une équivalence entre le programme parallèle généré et le programme initial. On se contente généralement d'assurer que l'état des variables est le même à la fin de l'exécution de chacun des programmes (équivalence de résultat). Une transformation est alors qualifiée de "légale" si elle respecte les dépendances de données (respect des conditions de Bernstein) et les dépendances de contrôle. Autrement dit, une transformation ne peut rendre parallèles deux actions S1 et S2 que si :

- les variables modifiées par S1 ne sont pas lues par S2 (dépendance directe) ;
- les variables lues par S1 ne sont pas modifiées par S2 (antidépendance) ;
- aucune variable n'est modifiée à la fois par S1 et S2 (dépendance de sortie) ;

– l'exécution de S2 ne dépend pas du résultat de S1 (ou vice-versa), ce qui est le cas lorsque l'une de ces instruction est l'évaluation d'une conditionnelle (dépendance de contrôle).

La recherche de parallélisme a donné lieu à de très nombreux travaux et il ne m'est pas possible d'en donner ici une bibliographie complète. On se reportera à [Chapman et Zima 90] pour les notions de base sur les analyses de dépendances et à [Wolfe 89] pour les bases des techniques de parallélisation. Les méthodes basées sur l'utilisation des polyèdres ont été décrites dans [Feautrier 96a]. Je citerai simplement ici quelques travaux sur la parallélisation qui ont contribué à définir un cadre unifié a ces techniques qui n'étaient vu jusqu'à présent que comme une liste de "recettes".

– Allen et Kennedy [Allen et Kennedy 87] ont développé une technique permettant de faire apparaître des instructions vectorielles dans un nid de boucles comportant une seule affectation.

– Barnerjee montre dans [Banerjee 90] que les transformations de boucles classiques (*interchange, reversal, swewing*) et leur composition peuvent s'exprimer dans un cadre unifié pour des nids de boucles de profondeur 2 dont les dépendances sont uniformes. On peut alors considérer qu'il s'agit de transformations d'index dont la matrice est unimodulaire. Il prouve qu'il est alors toujours possible de trouver une transformation valide produisant un nid de boucle dont la boucle la plus interne est parallèle. Ces résultats ont été généralisés par Wolf et Lam [Wolf et Lam 87] qui génèrent des nids de boucles parallèles (sauf éventuellement la boucle externe) pour des nids de boucles de profondeur n.

– Une technique de transformation plus avancée dite "d'ordonnancement affine par instruction" a été présentée par Feautrier [Feautrier 92a]. Cette technique s'applique à des nids de boucles où les dépendances sont affines ainsi qu'aux systèmes d'équations récurrentes uniformes (SERU). Cette technique permet de produire un nid de boucles dont la boucle externe représente le temps et toutes les boucles internes qui sont parallèles permettent de décrire l'ensemble des calculs devant être effectué à chaque instant (programme SIMD). Cette technique est généralisée dans [Feautrier 92b] au cas où le programme ne possède pas de base de temps affine. L'ordonnancement généré est alors multi-dimensionnel. Le temps est ainsi parcouru par un nid de boucles (on considère alors l'ordre lexicographique des indices). Sur ces bases, Darte et Vivien [Darte et Vivien 94] ont construit un algorithme qui maximise le parallélisme.

On trouvera dans [Darte et Vivien 95] une comparaison de ces trois algorithmes.

Il faut noter que la parallélisation de programmes séquentiel et la recherche d'ordonnancements pour les systèmes d'équations récurrentes sont des problèmes très proches, bien que les SERU ne soient pas tous calculables. Feautrier [Feautrier 91] a montré qu'un programme Fortran à contrôle statique peut être transformé en un

programme à assignation unique qui a son tour peut être transformé en SERU.

- L'outil Alpha permet d'exprimer un programme sous forme de SERU et de réaliser des transformations de programmes légales sans sortir du cadre des SERU. En Alpha, les calculs sont décrits par des équations définissant des variables multi-dimensionnelles. On peut définir des calculs point-à-point entre les variables, mais aussi effectuer des calculs sur les domaines de ces variables. L'utilisation initiale d'Alpha était la synthèse de circuits systoliques mais Alpha permet aussi la génération de code data parallèle [Quinton et al. 94].

- Opera [Loechner et Mongenet 96] est un outil qui permet d'écrire et de transformer un programme écrit sous forme de d'équations affines récurrentes paramétrées. La génération d'un programme parallèle à partir d'Opera se fait par le calcul d'une base de temps affine et d'une allocation linéaire.

- PEI [Genaud et al. 95] propose un formalisme reposant sur un modèle géométrique qui permet de réaliser des transformations de programmes sous forme d'équations récurrentes. Un programme PEI définit une relation entre un multi-ensemble de valeurs d'entrées et un autre multi-ensemble de valeurs de sorties. Les valeurs des éléments sont placées sur un domaine discret pour former un champ de données. En PEI on peut définir ou transformer une expression en appliquant des opérations :

  - calcul (sur l'ensemble des points du domaine)
  - changement de base,
  - routage (expression des dépendances)
  - superposition de champs de données

  Un champ de données peut aussi être défini de manière récursive ce qui permet de définir des scan et des réductions. Un programme PEI est un programme data parallèle (une fois les définitions récursives résolues). PEI doit donc être vu comme un outil de transformation de programmes data parallèles.

- HELP [Lazure 95] repose également sur le modèle géométrique. La sémantique des opérations est basée sur le référentiel (ie une opération globale s'exécute sur un ensemble de processeurs virtuels) comme dans PEI. Les données des algorithmes sont décrits par des "objets data parallèles" (dpo). Les calculs "data parallèles" s'effectuent sur l'ensemble des points d'un "domaine de conformité" qui est spécifié par un constructeur `on` qui se réfèrent à un dpo. Contrairement à PEI, le modèle HELP à été mis en œuvre par un langage impératif (C-HELP). La génération de code se base sur des projections de l'hyper-espace dans lequel sont plongés les dpo HELP a servi de base à la définition du langage IDOLE.

Répartition de programmes impératifs

La répartition de programmes impératifs, souvent appelée "distribution", traite de la transformation d'un programme séquentiel ou parallèle en un programme réparti. De même que pour la parallélisation, la répartition d'un programme doit préserver une "certaine sémantique" pour être utilisable (l'équivalence de résultat est souvent suffisante). Il existe deux méthodes que l'on peut rendre automatiques pour répartir un programme :

- la répartition par distribution du contrôle qui réparti un programme en s'appuyant sur une description explicite de la distribution des instructions du programme (donnée par exemple grâce à des annotations portant sur les structures de contrôle comme les boucles) ;

- la répartition par les données qui répartit un programme en s'appuyant sur une description explicite de la distribution des données (fournie par exemple grâce à des annotations portant sur les structures de données comme les tableaux). Bareau et al [Bareau et al. 93] ont montré que la règle dite "des écritures locales" que nous utilisons et qui sera expliquée plus loin permet de préserver cette sémantique.

Pour générer un programme parallèle et réparti à partir d'un programme séquentiel, 4 choix sont envisageables :

1. paralléliser puis répartir par distribution du contrôle : c'est l'approche prise notamment par Fortran S [Bodin et al. 93] qui s'appuie sur l'utilisation d'une mémoire virtuellement partagée pour la distribution et l'accès aux données ;

2. paralléliser puis répartir par distribution des données : c'est l'approche prise par les compilateurs HPF et assimilés lorsqu'ils génèrent un code optimisé. Nous utilisons cette technique qui est expliquée au chapitre 4 ;

3. répartir par les données sans paralléliser : c'est la technique de base des compilateurs HPF et assimilés lorsque la méthode précédente n'est pas applicable. Nous expliquons les bases de cette technique dans la partie suivante et avec plus de détails dans le chapitre 3 ;

4. paralléliser et répartir en même temps : cette approche plus ambitieuse qui est présentée dans [Feautrier 96b] consiste à calculer une fonction de placement statique sur le DFG (*Data Flow Graph* d'un programme à contrôle statique. La solution est obtenue par un algorithme approché (le problème est probablement NP-complet) qui ne permet pas toujours de tirer pleinement parti du parallélisme maximum de l'architecture ou de l'application ; par contre l'intérêt de cette approche est d'être globale. Il est intéressant de remarquer que ce problème d'optimisation ressemble beaucoup au problème de placement de tâches.

1.2.5 Le prix à payer

Dans le cadre du calcul haute performances un des points essentiel est de réaliser un calcul le plus rapidement possible et de pouvoir traiter des volumes de données importants. On serait tenté de croire que les ressources matérielles doivent être exploitées au maximum de leurs possibilités ; mais il n'en est rien sauf pour quelques projets extrêmes. La sûreté de fonctionnement et la portabilité des codes sont des paramètres au moins aussi importants qu'on ne peut espérer optimiser en réalisant manuellement un grand nombre d'optimisations souvent spécifiques à une machine. Il est donc toujours préférable de rechercher cette "qualité logicielle" même s'il faut la payer en achetant une machine plus puissante ou en acceptant d'avoir un code moins rapide.

Cette dégradation de performance doit rester dans des limites raisonnables qui ne sont pas faciles à chiffrer et permettent des débats passionnés. Pour ma part, j'ai considéré qu'une approche automatique fournissant un code deux fois moins rapide (ou plus coûteux en mémoire) qu'un programme optimisé à la main devait être considérée comme excellente en me basant sur l'évolution de la technologie des processeurs qui nous fait encore espérer des gains substantiels en vitesse de traitement dans les années à venir. La définition de la borne en dessous de laquelle il ne me paraît pas raisonnable de descendre est le facteur 10 ; mais là je dois avouer que c'est un seuil "psychologique".

Enfin il faut replacer cette course à la performance "à tout prix" dans son cadre historique. Dans les années 80, il était admis qu'on ne pouvait espérer avoir mieux qu'un gain de puissance de calcul de 2 toutes les 10 années. Cela a d'ailleurs été un des moteurs du développement des architectures parallèles. Mais cette affirmation s'est avérée fausse ; par contre la complexité des architectures et des applications n'a cessé de croître rendant cruciale l'utilisation d'outils permettant aux programmeurs de faire abstraction des caractéristiques fines pour se concentrer sur la logique des applications.

Un autre argument en faveur de l'utilisation d'un modèle de programmation différent du modèle d'exécution tient à la complexité des optimisations de code. Il faut admettre qu'un programmeur est souvent moins apte à fournir du code tirant au mieux parti de l'ensemble des ressources d'une machine ou même d'un processeur qu'un outil automatique [Bodin 97]. Il n'est donc pas rédhibitoire d'empêcher le programmeur de réaliser lui-même de telles optimisations.

1.3 Présentation de mes travaux

1.3.1 Placement de processus

Parmi les outils utiles pour la virtualisation des processeurs, c'est-à-dire pour permettre une adaptation du parallélisme d'une application à une architecture donnée, j'ai étudié lors de mon DEA et au cours de ma thèse les problèmes de placement de programmes explicitement parallèles sur architecture à mémoire distribuée. J'ai développé et analysé des algorithmes approchés de placement statique de processus

qui permettent de décharger le programmeur du choix du placement et du regroupement des processus composant son application [André et Pazat 88].

Ces algorithmes ont été d'abord utilisés sur la machine hypercube iPSC/1 sur un programme de lancer de rayons [André et al. 89]. A l'époque cet aspect de la programmation des architectures à mémoire distribuée était critique vu les coûts des communications et le type de routage utilisé (*store and forward*).

Le modèle que j'ai utilisé suppose que le nombre de processus ainsi que la structure des communications sont connus statiquement ou tout au moins avant le début de l'exécution répartie (il est parfois possible d'effectuer un pré-traitement en séquentiel pour calculer le graphe de communication). On peut alors représenter un programme par un graphe dont les sommets représentent les tâches et dont les arcs représentent les communications. Le fait de supposer connu le nombre de processus avant l'exécution ne permet pas de traiter les programmes qui créent dynamiquement des tâches mais cette facilité est rarement utilisée dans le cadre du calcul hautes performances. De manière duale, l'architecture est supposée ne pas évoluer pendant la durée d'exécution du programme. Le nombre de processeurs ne pouvant pas changer il n'est pas possible d'en enlever pour tenir compte de pannes ni d'en ajouter pour tenir compte d'une reconfiguration dynamique de la machine. De même, les liens de communication entre les processeurs sont supposés fixes ce qui interdit de prendre en compte les architectures reconfigurables comme le T-node qui ont eu une courte vogue mais ont aujourd'hui disparu.

Le placement statique se fonde sur la recherche d'un extremum d'une fonction de "coût" qui représente l'adaptation entre le choix du placement des tâches sur les processeurs et les capacités de la machine en termes de traitement et de communication. Les fonctions intéressantes étant quadratiques, la recherche d'un extremum est un problème NP-complet [André et Pazat 88] ; c'est pourquoi les algorithmes que j'ai développés sont des algorithmes approchés.

Actuellement beaucoup d'efforts sont faits sur l'étude du "placement dynamique" incluant la possibilité de faire migrer des tâches durant l'exécution. Ces techniques permettent d'adapter un programme parallèle et distribué à une architecture comportant un nombre de processeurs et une configuration du réseau de communication différents de ceux initialement prévus dans le programme. Ces algorithmes ne permettent pas néanmoins à eux seuls de résoudre efficacement les problèmes de portage car si le nombre de processeurs est supérieur au nombre prévu ces algorithmes ne "peuvent rien pour nous" et réciproquement, lorsque le nombre de processeurs devient très inférieur au nombre de processus de l'application, le surcoût lié à la gestion des processus sur les nœuds devient prohibitive. Néanmoins, ces algorithmes sont intéressants à inclure dans un chargeur (pour les algorithmes statiques) ou dans des systèmes (pour les algorithmes dynamiques comme ceux de Gatostar [Folliot et Sens 94]) pour machines parallèles à mémoire distribuée.

1.3.2 Réexécution de programmes répartis

Toujours dans le cas où le modèle de programmation n'est qu'un sous-ensemble du modèle d'exécution, j'ai étudié lors de ma thèse les problèmes liés à la mise au

point de programmes répartis sur ce type d'architecture.

Lorsque le modèle de programmation est explicitement parallèle et distribué et qu'il n'est pas possible de contraindre le comportement des programmes, le travail de mise au point s'avère particulièrement difficile. En particulier, même en l'absence d'intrusion, un même programme peut produire plusieurs comportements observables différents. Il est alors utile de disposer de mécanismes de contrôle d'exécution permettant de reproduire une exécution donnée autant de fois que nécessaire. Le mécanisme que j'ai défini et validé [Pazat 89a] au cours de ma thèse s'appuie sur l'enregistrement des choix lors de l'exécution de primitives de réception non déterministes du type[5] `[P1?x -> code1 [] P2?y -> code2]` qui existent sous une forme syntaxique différente dans PVM, MPI et à l'époque sur NX2. Lors d'une réexécution contrôlée, nous remplaçons ces choix par la lecture des enregistrements réalisés. Ainsi `[P1?x -> code1 [] P2?y -> code2]` devient

```
[ Enregistrement[i] = "choix1" -> P1?x;code1
[] Enregistrement[i] = "choix2" -> P2?y;code2];
i=i+1
```

Ceci peut être réalisé très simplement par modification des bibliothèques de communication ou par interception des primitives du système de communication. L'intérêt principal de cette approche est de ne nécessiter aucun mécanisme global (pas de reconstruction d'un temps ou d'un état global).

Ce type d'outil est une aide précieuse pour la mise au point de programmes mais ne limite en rien la complexité de la mise au point : en effet, chercher à corriger un programme réparti en corrigeant chacun de ses comportements erronés est un processus fastidieux dont la terminaison n'est même pas assurée! Ce n'est donc qu'un "pis aller" alors qu'il serait plus raisonnable de contraindre les comportements des programmes par un modèle de programmation adapté.

1.3.3 Génération de code réparti

Lorsque le modèle de programmation est différent du modèle d'exécution il est nécessaire de mettre en œuvre des techniques de virtualisation et/ou de transformation de programmes. Le travail qui est détaillé dans la suite de ce document traite de la transformation de programmes séquentiels comportant des boucles parallèles (modèle SEQ of PAR) en un programme parallèle et réparti.

Modèle de programmation

Le modèle de programmation que nous prenons en référence est le modèle impératif "SEQ of PAR" dans lequel un programme est composé d'une part de variables et de structures de données et d'autre part d'instructions et de structures de contrôle dont les boucles parallèles. La distribution est présente en tant que directive de compilation mais n'influe pas sur la logique du programme. La distribution porte sur les structures de données de haut niveau (en fait seulement les tableaux).

5. J'utilise ici la syntaxe de CSP, mais je ne fais pas la supposition du rendez-vous.

Le chapitre suivant décrit quelques langages permettant d'exprimer ce modèle de programmation.

Machine virtuelle cible

Le niveau commun de virtualisation de ces architectures, autrement dit la machine virtuelle qui sert de cible à la transformation de programmes est le suivante :

- chaque nœud permet d'exécuter une ou plusieurs tâches en parallèle, tâches qui partagent un espace d'adressage physique unique ;
- les échanges de données entre les tâches s'exécutant sur des nœuds distants se font par coopération explicite, c'est-à-dire par échange de messages.

La topologie du réseau d'interconnexion (qui représente les liaisons physiques existantes), n'est pas prise en compte dans notre modèle et nous supposons seulement que les protocoles de communication et de routage utilisés permettent une mémorisation et un acheminement des messages sans perte ni déséquencement (files fifo) entre tous les nœuds. C'est en particulier le type de modèle offert par la bibliothèque POM que nous utilisons et qui a été développée dans l'équipe PAMPA [Guidec et Mahéo 95b].

Dans le projet PANDORE, nous avons de plus défini une virtualisation de la mémoire qui est décrite au chapitre 4.

Transformations de programmes

Pour aider le lecteur à suivre sans encombres la suite de ce document [6], je donne tout de suite les éléments essentiels de la méthode de transformation de programmes qui sera détaillée dans les chapitres suivants. Le lecteur averti pourra passer directement à la lecture du chapitre suivant.

La transformation d'un code séquentiel ou parallèle agissant sur un espace d'adressage global, à un code formé d'un ensemble de processus disposant chacun d'un espace mémoire local s'appuie sur le modèle SPMD et sur le principe des *écritures locales* [Callahan et Kennedy 88]. Les processus générés présentent un code générique, mais agissent chacun sur des données qui leur sont propres.

Pour comprendre le principe des *écritures locales*, il faut noter que la spécification d'une distribution de données établit une relation de *possession* entre les processus et les données (éléments de tableau) en associant des fragments de tableaux à des processus. Nous dirons qu'un processus *possède* une donnée, si la spécification de distribution indique que ce processus doit stocker la donnée dans sa mémoire locale. La règle des *écritures locales* que nous utilisons impose qu'une instruction du programme source qui modifie une donnée (i.e. une affectation) soit exécutée par le processus qui possède cette donnée [7] ; autrement dit l'évaluation de la partie droite d'une affectation est réalisée par le possesseur de la partie gauche.

6. Ceci peut également permettre au lecteur pressé et non spécialiste de passer directement à la conclusion.

7. Cette règle devrait plus justement s'appeler règle des calculs locaux puisque qu'une écriture est toujours locale sur une machine utilisant une bibliothèque d'échange de messages.

Prenons un exemple dans lequel, pour simplifier, les données distribuées sont des variables scalaires (et non pas des tableaux), et voyons quel serait le rôle des processus générés en appliquant cette méthode de compilation :

```
ent a, b, c ,d
distribuer a et d sur P1
distribuer b   sur P3
distribuer c   sur P2
b := 10    (1)
a := d * 2  (2)
c := a + b  (3)
```

La compilation de ce pseudo-code génère trois processus P1, P2 et P3 dont les actions effectives sont résumées par le tableau suivant :

Code original	Code de P1	Code de P2	Code de P3
(1) `b := 10`			`b := 10`
(2) `a := d * 2`	`a := d * 2`		
(3) `c := a + b`	envoyer(`a`) à P2	recevoir(`a`) de P1 recevoir(`b`) de P3 `c := a + b`	envoyer(`b`) à P2

L'instruction (1) du code séquentiel correspond à l'affectation de la constante 10 à la variable distribuée `b`. Cette variable a été attribuée au processus P3 ; la règle des écritures locales indique donc que c'est P3 qui doit réaliser l'affectation : les deux autres processus P1 et P2 ne sont pas impliqués dans l'exécution de cette instruction.

L'instruction (2) ne va pas non plus entraîner de coopération entre les processus. En effet, la variable lue (`d`) et la variable écrite (`a`) ont été attribuées au même processus (P1). Ce processus peut donc procéder directement à l'affectation qui ne met en jeu que des variables qui lui sont locales. La localité des accès réalisés par les deux instructions (1) et (2), rend possible l'exécution en parallèle des deux affectations.

L'instruction (3) est différente : elle présente des références en lecture aux variables distribuées `a` et `b` qui ont été respectivement attribuées aux processus P1 et P3, et une référence en écriture à la variable `c`, possédée par un troisième processus, le processus P2. P2, qui doit réaliser l'affectation, a besoin d'accéder aux valeurs courantes des variables `a` et `b` qui ne sont pas présentes dans son espace mémoire local. Pour ce faire, il se met en attente de ces valeurs qui sont respectivement envoyées par les processus P1 et P3. La non-localité des accès entraîne ainsi une coopération des trois processus par échanges de messages. Le programme ainsi généré est à la fois parallèle et distribué (les instructions (1) et (2) du programme s'exécutent en parallèle).

Dans l'exemple ci-dessus, on connaît statiquement, pour chaque référence, la distribution de la donnée à laquelle on accède. Ce n'est généralement pas le cas pour un accès indexé, comme par exemple `A[i][j]`, où la valeur des indices n'est connue qu'à l'exécution.

L'impossibilité de connaître statiquement la localisation des données oblige le compilateur à générer un code SPMD composé d'une suite d'instructions *gardées*. L'évaluation de ces gardes à l'exécution permet à chaque processus de calculer sa contribution à la réalisation d'une instruction comme cela est décrit dans [Pazat 91]. Prenons par exemple l'affectation A[i] := B[j]. Le code SPMD gardé, généré par le compilateur, présenterait la forme suivante :

cas
je possède B[j] *et* A[i] → *exécuter* A[i] := B[j]
je possède B[j] *et pas* A[i] → *envoyer* B[j] *au possesseur de* A[i]
je possède A[i] *et pas* B[j] → *recevoir* B[j]; *exécuter* A[i] := B[j]
autrecas → *rien*
fcas

Les techniques de transformation de programmes utilisant la distribution de données sont exposées dans les chapitres 3 pour le cas de la transformation instruction par instruction et au chapitre 4 dans le cas des boucles.

Chapitre 2

Langages à distribution de données

Ce chapitre présente les langages qui ont été développés pour la génération de code réparti par distribution de données. Il s'agit essentiellement d'annotations de type directives de compilation ou de modifications mineures de langages existants et non de véritablement nouveaux langages. Il n'y a donc pas ici de contribution scientifique importante malgré l'importante littérature consacrée au sujet [1]. Je présente toutefois quelques-uns de ces langages d'une part pour faciliter la lecture des deux chapitres suivants et d'autre part pour donner une idée plus précise du modèle de programmation offert.

2.1 Caractéristiques communes

Les langages que nous étudions ici sont intrinsèquement séquentiels et construits comme des sur-ensembles de langages impératifs existants comme Fortran, C ou Pascal. La plupart d'entre eux permettent d'exprimer une forme limitée de parallélisme. Tous ces langages permettent d'exprimer la distribution de données sur une architecture virtuelle de processeurs [2] ; cette distribution servant de guide à la génération du code réparti. Ces langages étant structurés, ils possèdent en particulier des structures de type procédural dont nous étudierons les interactions avec la distribution de données. D'une manière générale, ces langages sont bien adaptés à l'expression d'algorithmes réguliers sur des structures régulières (calcul matriciel par exemple). L'ajout de possibilités pour traiter des structures irrégulières ou creuses n'est pas discuté ici.

De nombreux travaux ont été menés depuis 1988, pour concevoir de tels langages. Les plus connus sont Fortran D [Hiranandani et al. 91] et Vienna Fortran [Chapman et al. 91] ; notre travail concerne le projet PANDORE.

1. Je ne me permettrai pas de faire de références bibliographiques ici.
2. Le lien avec l'architecture physique est de la responsabilité d'un "chargeur".

Parmi les précurseurs, citons également Kali [Koelbel et Mehrotra 90] et EPPP [Ning et al. 95] qui procèdent de la même approche. Pour tenter de concentrer les efforts de recherche et de développement et surtout pour convaincre les utilisateurs de l'intérêt de cette approche, le *High Performance Fortran Forum* composé d'industriels et d'universitaires a défini et tente d'imposer un "standard" : le langage "High Performance Fortran" (HPF) [HPFF 97] qui a été défini comme une extension de Fortran 90 et dont nous décrirons les principales caractéristiques.

2.1.1 Expression du parallélisme

Dans ces langages, le parallélisme peut s'exprimer sur les structures de contrôle comme les boucles sur lesquelles on peut spécifier que chaque itération peut s'exécuter indépendamment des autres (on parlera alors de parallélisme de contrôle). Le parallélisme peut aussi s'exprimer par des constructions dont la sémantique est celle de la multi-affectation [Chandy et Misra 88], c'est-à-dire le *copy-in/copy-out* qui signifie que toutes les lectures sont effectuées avant les écritures (on parlera alors de parallélisme de données).

Dans sa version la plus primitive, le parallélisme de données s'exprime par l'affectation entre tableaux (ou sections de tableaux) qui existe notamment dans le langage Fortran 90. Une généralisation en est la boucle à parallélisme de données (`forall`) introduite dans Fortran D, puis dans HPF. Chaque instance du corps de boucle travaille sur une copie de l'espace des données et ne peut donc utiliser que les valeurs définies avant la boucle. Cette construction n'est pas classique dans les langages impératifs séquentiels.

Comparons les deux boucles suivantes :

```
DO I = 2,N                    FORALL I = 2,N
   X(I) = X(I-1)                 X(I) = X(I-1)
END DO                        END FORALL
```

Après l'exécution de la version *séquentielle* DO, tous les éléments du tableau `X` ont pris la valeur de `X[1]`, alors que la version `FORALL`, opère en parallèle un décalage de `X`.

Cette construction permet d'exprimer des opérations globales sur les tableaux plus complexes que celles de Fortran 90, comme l'affectation de la diagonale d'une matrice :

```
FORALL I =1,N
  A(I,I) = B(I)
END FORALL
```

Dans certains langages comme Vienna Fortran seule la boucle parallèle plus classique existe[3] : elle signifie qu'il n'existe pas de dépendances entre les itérations.

Dans d'autres langages, comme Pandore, il n'existe pas de constructions spécifiques pour exprimer le parallélisme, ce qui nécessite de faire un certain nombre d'analyses supplémentaires à la compilation.

3. Cette construction s'appelle malheureusement `forall` en Vienna Fortran.

2.1.2 Spécification de la distribution des données

La distribution de données spécifie la manière dont les données structurées (en général les tableaux) sont découpées et placées sur les mémoires locales des processeurs.

La spécification de distribution de données a été introduite dans des langages impératifs séquentiels soit par augmentation du langage et donc modification de la syntaxe ou d'une manière plus pragmatique par des annotations (dans des commentaires pour HPF).

La distribution ne peut en général porter que sur les tableaux parce que ce sont les seules structures de données de haut niveau existantes dans ces langages (en particulier en Fortran 77). De plus, des techniques de compilation efficaces que nous présentons au chapitre 4 ont pu être développées dans ce cadre.

Toujours pour des raisons liées aux optimisations de compilation, le découpage des tableaux est réalisé en blocs de taille égale. Ces blocs sont placés sur des arrangements réguliers de processeurs virtuels (qui ne correspondent pas forcément à des processeurs physiques). La séparation entre découpage et placement n'est pas toujours claire et ces deux notions sont regroupées sous le terme général de "distribution".

Les distributions peuvent être statiques, c'est-à-dire spécifiées lors de la déclaration des tableaux et valides tout au long de la vie du tableau; ou être dynamiques, c'est-à-dire pouvant être changées lors de l'exécution du programme par des instructions de redistribution. Les distributions statiques sont plus faciles à traiter par les compilateurs alors que les distributions dynamiques sont souvent plus souples à utiliser.

Quelques tentatives d'introduction de spécifications de distribution plus complexes ont été faites mais ont rencontré peu de succès car elles empêchent très souvent de générer un code efficace.

2.1.3 Procédures et distribution

Tous les langages présentés ici s'appuient sur le concept de procédure comme élément de structuration. Les variables des programmes pouvant posséder une distribution, qui est en quelque sorte un enrichissement de leur type, nous préciserons comment celle-ci est prise en compte lors d'un appel de procédure.

- La distribution peut être liée aux paramètres formels des procédures. Dans ce cas, on connaît statiquement la distribution de toutes les données référencées dans une procédure sans avoir besoin de réaliser une analyse du graphe des appels. Le corps d'une procédure pourra alors être compilé en utilisant des techniques optimisées dès que cela est possible. C'est à l'exécution, lors de l'appel d'une procédure que les paramètres effectifs seront redistribués conformément à la spécification de distribution.

- La distribution peut être associée aux paramètres effectifs (on dit alors qu'elle est héritée). Dans ce cas, la distribution des paramètres n'étant connue qu'à l'appel de la procédure, il faut soit compiler différemment la procédure pour

chaque appel (technique dite de clonage qui est proche de l'*inlining*), soit attendre l'exécution pour connaître la distribution effective, ce qui empêche un grand nombre d'optimisations du code généré.

La réalisation et l'utilisation de bibliothèques pose alors un difficile problème car on doit disposer d'autant d'exemplaires des fonctions de bibliothèque qu'il existe de distributions possibles pour les paramètres si l'on veut éviter de coûteuses redistributions à l'exécution. Une solution pour écrire des bibliothèques a été proposée dans le langage Vienna Fortran où il est proposé d'ajouter des fonctions permettant de connaître la distribution d'une donnée lors de l'exécution. Cette technique permet entre autre de regrouper sous une même interface des procédures utilisant des distributions différentes. Une solution plus élégante consisterait à utiliser les mécanismes de surcharge et d'édition de liens dynamiques mais n'a pas à ma connaissance été proposée.

De plus, pour des raisons pratiques, des procédures dites "pures" qui ne génèrent pas de communications ont été introduites dans ces langages.

2.2 Vienna Fortran

Le projet VFCS (Vienna Fortran Compilation System) est la suite du projet SUPERB réalisé en 1988 par Michael Gerndt et Hans Zima [Zima et al. 88].

L'extension de SUPERB a d'abord consisté à définir un langage, Vienna Fortran [Chapman et al. 91], initialement construit autour de Fortran 77, puis remanié autour de Fortran 90. Ce langage permet d'exprimer directement dans le programme source la distribution des données et une certaine forme de parallélisme par l'intermédiaire d'un nouveau constructeur (`FORALL`).

2.2.1 Parallélisme

La construction `FORALL` permet d'indiquer au compilateur qu'une boucle est parallèle (sans dépendances inter-itérations). C'est la seule forme de parallélisme ajoutée à Fortran 90 par Vienna Fortran.

Dans cette construction on peut spécifier le processeur sur lequel doit s'exécuter chaque instance du corps de boucle par la clause `ON`, c'est-à-dire spécifier une distribution du contrôle éventuellement différente de la règle des calculs locaux. Cette spécification peut utiliser la fonction `OWNER` ou être directement donnée en terme de processeur. Lorsque le domaine d'itération est un parallélépipède, sa distribution peut également être spécifiée de manière analogue à la distribution d'un tableau.

Dans l'exemple suivant, les calculs seront exécutés sur les processeurs possédant la partie droite de l'affectation ce qui générera environ trois fois moins de communications que ne le ferait l'application de la règle des calculs locaux :

```
PARAMETER (N=1000, P=10)
PROCESSORS PROCS(P,P)
REAL,  DISTRIBUTED(BLOCK,:) TO PROCS :: A(N,N)
REAL,  DISTRIBUTED(:,BLOCK) TO PROCS :: B(N,N), C(N,N), D(N,N)
FORALL (I=1,N; J=1,N) ON OWNER(B(I,J))
 A(I,J) = B(I,J) + C(I,J) + D(I,J)
END FORALL
```

Les opérations de réduction sont exprimées (syntaxiquement) à l'aide de la boucle `FORALL`, mais avec une sémantique spécifique puisqu'elles comportent des dépendances sur la variable d'accumulation.

2.2.2 Distribution des tableaux

La spécification de la distribution des données utilise un ensemble de "processeurs virtuels" sur lesquels on distribue directement les tableaux. Les variables scalaires, quant à elles, sont répliquées.

L'ensemble des processeurs a une structure de tableau dont on construit d'abord la structure initiale (le "tableau primaire") sur laquelle on peut ensuite construire plusieurs vues. La distribution de données utilise ces vues ou le tableau primaire (par défaut).

Par exemple, dans la déclaration

```
PROCESSORS GRID(10,2) RESHAPE VECT(20)
```

le tableau `GRID` qui constitue le *tableau primaire*, forme une grille de `10x2` processeurs, dont `VECT` fournit une vue en ligne.

Il existe deux catégories de distributions : les distributions statiques qui restent valides pour toute la durée de vie des tableaux auxquels elles s'appliquent, et les distributions dynamiques. La catégorie de distribution est associée à un tableau lors de sa déclaration et ne peut donc être changée.

Voici quelques exemples de distribution statiques et la sémantique qui y est associée :

– distribution *directe*

```
PROCESSORS P1(10,10)
REAL A(10000)    DIST(CYCLIC(10))    TO P1
REAL B(100,2,100) DIST(BLOCK,:,BLOCK) TO P1
```

Les éléments du tableau `A`, groupés par blocs de 10 éléments, sont distribués de façon cyclique aux processeurs de `P1`. Comme le nombre de dimensions de

`A` est inférieur à celui de `P1`, les groupes sont répliqués sur la deuxième dimension du tableau de processeurs: les processeurs `P1[i][1..10]` possèdent les mêmes éléments de `A`. Pour le tableau `B`, la première et la troisième dimension sont distribuées en blocs alors que la deuxième dimension n'est pas distribuée: le processeur `P1[i][k]` possède les éléments `B[10×(i-1).. 10×i-1][1..2][10×(k-1).. 10×k-1]`.

– distribution *par alignement*
```
REAL C(2,100,100) ALIGN C(K1,K2,K3) WITH B(K2,K1,K3)
```
Dans la distribution spécifiée pour le tableau `C`, les dimensions 1, 2 et 3 sont respectivement alignées sur les dimensions 2, 1 et 3 du tableau `B`. Ainsi dans notre exemple, la première dimension de `C` n'est pas distribuée tandis que les deux dernières dimensions sont distribuées par blocs.

Il est également possible de spécifier des distributions irrégulières en utilisant des *tableaux de placements* (*mapping array*) dont les valeurs correspondent à des numéros de processeurs virtuels ou de définir ses propres fonctions de distribution et d'alignement.

2.2.3 Procédures

Le langage Vienna Fortran conserve la notion de procédure présente dans Fortran.

La distribution des paramètres peut être spécifiée dans la déclaration des paramètres formels ; à l'exécution les paramètres effectifs seront alors éventuellement redistribués afin de respecter la distribution spécifiée. Cette redistribution est locale à la procédure pour les tableaux "statiquement distribués" (i.e. les tableaux recouvrent leur distribution initiale à la sortie de la procédure) ; par contre, pour les tableaux "dynamiquement distribués", la redistribution reste effective à la sortie de la procédure, sauf spécification contraire (`RESTORE`). Pour des raisons d'efficacité, il est alors utile de spécifier le mode de passage de paramètres (`INTENT IN`, ou `OUT`, ou `INOUT`) pour éviter de gérer des redistributions inutiles qui sont coûteuses.

La distribution peut également être héritée du contexte de l'appelant (on utilise la distribution du paramètre effectif), dans ce cas, la distribution effective n'est connue qu'au moment de l'appel. Pour guider les optimisations du compilateur, on peut restreindre les distributions autorisées par héritage en spécifiant un attribut `RANGE`. Ceci permet au compilateur d'effectuer un "clonage" de la procédure pour chacune des distributions permises par la clause `RANGE`.

2.3 Fortran D

Le projet Fortran D [Hiranandani et al. 91] est dirigé par Ken Kennedy, qui est le premier à avoir défini la transformation de programmes séquentiels par l'application de la règle des calculs locaux et l'utilisation du modèle SPMD [Callahan et Kennedy 88]. Comme pour Vienna Fortran, ce langage a évolué vers Fortran 90 [Benkner 94].

2.3.1 Parallélisme

La construction `FORALL` offerte par le langage Fortran D permet d'exprimer le parallélisme d'un groupe d'affectations : c'est une vraie construction de parallélisme de données de par sa sémantique dite *copy-in, copy-out*. Comme en Vienna Fortran, la clause `on` peut être utilisée pour préciser l'attribution des itérations aux processeurs, la règle des écritures locales étant alors abandonnée. La spécification d'opérations de réduction est également possible.

2.3.2 Alignement et distribution des tableaux

L'approche adopté par Fortran D pour la distribution des tableaux diffère de celle choisie par Vienna Fortran par l'utilisation optionnelle de structures virtuelles utilisées pour la distribution.

Alignement

Les tableaux peuvent être "alignés" sur des structures virtuelles appelées *décompositions* qui représentent un espace d'indices sous forme de tableaux dont on doit spécifier la distribution. La localisation des données dépend ainsi de la distribution des *décompositions* et de l'alignement des tableaux sur ces *décompositions*. On peut également spécifier directement les distributions sans passer par les décompositions.

Outre l'alignement *exact*, le langage Fortran D propose notamment les alignements *décalés*, *effondrés* et *répliqués*, illustrés par la figure 2.1. La liste exhaustive des alignements exprimables peut être trouvée dans la thèse de Chau-Wen Tseng [Tseng 93].

Distribution

La distribution des tableaux ou des *décompositions* est effectuée sur une structure de processeurs dont la déclaration est implicite. Ainsi

```
DISTRIBUTE TEMPLATE_2D(BLOCK(2),BLOCK(4))
```

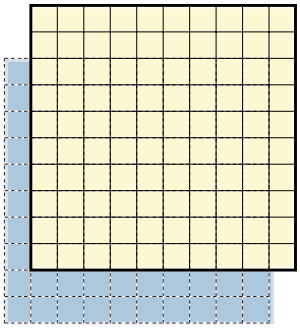
distribue la structure `TEMPLATE_2D` sur une structure de $2 \times x4$ processeurs. Le nombre maximal de processeurs utilisables dans les distributions est fixé pour tout le programme par le paramètre `N$PROC` dont la valeur est soit fixée à l'exécution, soit passée comme paramètre au compilateur.

On peut également laisser le compilateur choisir la taille de la structure de processeurs en fonction du nombre réel de processeurs comme dans l'expression de distribution

```
DISTRIBUTE TEMPLATE_1D(BLOCK)
```

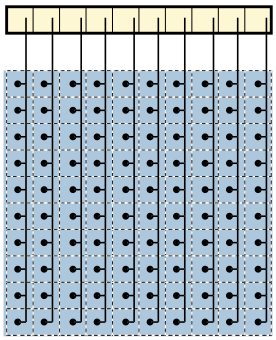
alignement décalé

```
REAL X1(N,N)
DECOMPOSITION B(N,N)
ALIGN X1(I,J) with B(I-2,J+1)
```



alignement dupliqué

```
REAL X3(N)
DECOMPOSITION B(N,N)
ALIGN X3(I) with B(*,I)
```



alignement effondré

```
REAL X2(N,N)
DECOMPOSITION A(N)
ALIGN X2(I,J) with A(J)
```

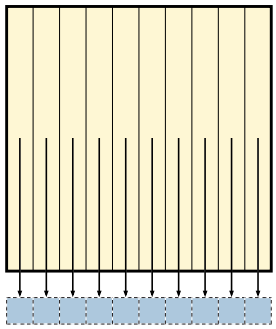


FIG. 2.1 – *Exemples d'alignements exprimables en Fortran D*

Il existe 3 fonctions de distribution pré-définies, les distributions étant effectuées dimension par dimension :

`BLOCK(P)` distribue la dimension (supposons-la de taille `N`) sur `P` processeurs en tranches de taille `N/P`.

`CYCLIC(P)` distribue la dimension en tranches de taille `1` en procédant de manière cyclique : le processeur `i` se voit attribuer les tranches `i`, `i+P`, `i+2P`, etc.

`BLOCK_CYCLIC(T,P)` est identique à la distribution `CYCLIC(P)` mis à part la taille des tranches qui n'est plus `1` mais `T`. Enfin, pour préciser qu'une dimension n'est pas distribuée, le symbole "`:`" est utilisé.

Le paramètre `P` est optionnel, lorsqu'il est omis, la taille de la dimension de la structure de processeurs est choisie par le compilateur.

Comme en Vienna Fortran, il est de plus possible de spécifier des distributions irrégulières en passant par l'intermédiaire d'un "tableau de placement".

Les spécifications d'alignement et de distribution se faisant par l'intermédiaire d'instructions et non pas de déclarations, le ré-alignement et la redistribution de n'importe quel tableau sont possibles même si le prototype du compilateur impose un certain nombre de restrictions.

2.3.3 Procédures

Dans le cadre de l'utilisation de procédures, le langage Fortran D adopte les conventions suivantes.

1) Les tableaux constituant les paramètres formels d'une procédure héritent de la distribution des paramètres effectifs. Cette distribution par défaut est valide jusqu'à la rencontre d'une instruction de distribution.

2) Un tableau distribué, passé en argument d'une procédure, retrouve, au retour de la procédure sa distribution d'avant l'appel, même si la procédure modifie la distribution de ses paramètres.

De plus, le compilateur Fortran D interdit tout appel de procédure à l'intérieur d'une boucle `FORALL`. En effet, chaque instance d'une itération d'un `FORALL` est destinée à être exécutée sur un processeur unique et ce, de façon indépendante. Dans ce cadre d'exécution indépendante, la coopération entre les processeurs, nécessaire à l'exécution de toute procédure, est irréalisable.

2.4 High Performance Fortran

Le langage High Performance Fortran (HPF) a été défini par le High Performance Fortran Forum (HPFF) [HPFF 97] comme une extension de Fortran 90. Ce consortium regroupe des utilisateurs, des constructeurs et des universitaires. Le premier document de spécification officiel du langage (qui n'est pas encore une norme) date

du mois de mai 1993, le document a été mis à jour en novembre 1994 et la version 2.0 est officielle depuis novembre 1996.

Décrire ici HPF dans son ensemble serait une gageure, je ne donc donne ici qu'un aperçu de ce langage renvoyant le lecteur courageux à la lecture du document de spécification du langage [HPFF 97].

2.4.1 Parallélisme

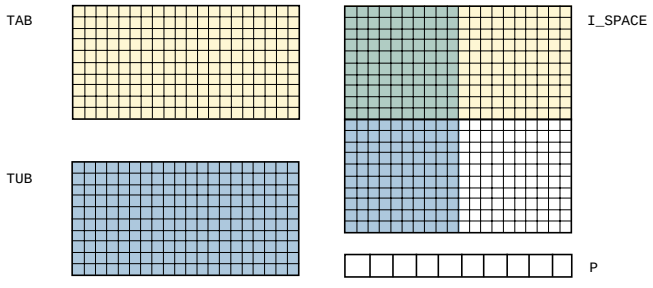
Comme Fortran D, HPF permet de spécifier le parallélisme de données avec une construction syntaxique `FORALL` qui généralise l'affectation tableau de Fortran 90. Cette construction possède également une forme syntaxique "allégée", l'instruction `FORALL`[4] ainsi que la possibilité d'exprimer des réductions.

Le parallélisme de contrôle s'exprime par une annotation (`!HPF$ INDEPENDENT`).

2.4.2 Alignement et distribution des tableaux

Alignement

Les directives d'alignement sont très similaires à celles de Fortran D, le terme `TEMPLATE` remplaçant `DECOMPOSITION`. Les distribution peuvent être spécifiées par alignement sur des *templates*, par alignement sur d'autres tableaux ou directement.




```
      REAL(100,200): TAB, TUB
!HPF$ PROCESSORS P(10)
!HPF$ TEMPLATE I_SPACE(200,200)
!HPF$ DISTRIBUTE I_SPACE(*,BLOCK) ONTO P
!HPF$ ALIGN TAB(I,J) WITH I_SPACE(I,J)
!HPF$ ALIGN TUB(I,J) WITH I_SPACE(J,I)
```

FIG. 2.2 – *Exemple de distribution de données*

4. Le langage Fortran n'a pas de structure de bloc, c'est pourquoi il existe une forme instruction lorsque le `FORALL` ne contient qu'une instruction et une construction `FORALL` pour regrouper plusieurs instructions.

Distribution

Contrairement à Fortran D et comme dans VFCS, la structure de processeurs qui est utilisée pour la distribution doit être explicitement déclarée. Plusieurs structures de processeurs peuvent coexister et être utilisables en un même point du programme. La cible de la distribution est précisée par le mot clé `ONTO`. Contrairement à VFCS, ces structures ne sont pas des vues sur une structure primaire : il n'y a aucun lien entre les différentes structures et leur placement sur les processeurs réels de la machine n'est pas spécifié dans le langage.

```
      SUBROUTINE JACOBI (B)
        PARAMETER(N = 1024)
        PARAMETER(NLOOP = 100)
        REAL *8 B(N,N)
        REAL *8 A(N,N)

!HPF$ PROCESSORS PROCS(NUMBER_OF_PROCESSORS ())
!HPF$ DISTRIBUTE (BLOCK, *) ONTO PROCS :: A, B

        INTEGER I, J, K

        DO K=1, NLOOP
          DO I=2, N-1
     DO J=2, N-1
               A(I,J) = V*B(I, J) + W*(B(I-1,J)+B(I+1, J)+
     1         B(I, J-1)+B(I, J+1))
     END DO
          END DO

          DO I=2, N-1
            DO J=2, N-1
              B(I, J) = A(I, J)
            END DO
          END DO
        END DO
      END

      PROGRAM JACOBI

      PARAMETER(N = 1024)
      PARAMETER(NLOOP = 100)
      REAL *8 Y(N,N)
      REAL *8 V
      REAL *8 W
!HPF$ PROCESSORS PROCS(NUMBER_OF_PROCESSORS ())
!HPF$ DISTRIBUTE (BLOCK, *) ONTO PROCS :: Y
      V = 0.5
      W = 0.125
C     Initialisation
      CALL JACOBI(Y)
C     utilisation des resultats

      END
```

FIG. 2.3 – *Exemple de programme* HPF

La distribution est réalisée par la déclaration `DISTRIBUTE` et des fonctions `BLOCK` et `CYCLIC` proches de celles de Fortran D.

La structure physique de la machine réelle est néanmoins accessible par des fonctions qui donnent le nombre de dimensions de la structure et son extension dans chaque dimension.

La figure 2.2 montre la distribution d'un tableau `TAB` par colonnes contiguës sur un tableau de processeurs. Le *template* utilisé est ici plus grand que le tableau, il peut permettre d'aligner d'autres tableaux (par exemple de manière symétrique par blocs de lignes sur les premiers processeurs pour le tableau `TUB`.

2.4.3 Procédures

Les paramètres effectifs sont redistribués selon la spécification de distribution affectée aux paramètres formels. Si celle-ci est absente, ou si l'attribut `INHERIT` est spécifié, les paramètres ne sont pas redistribués.

Un exemple de programme HPF est donné en figure 2.3.

2.5 C-Pandore

Le langage C-Pandore que nous avons défini reprend la syntaxe du langage C qui est un langage structuré possédant la notion de blocs et les constructions communes aux langages impératifs séquentiels.

Les pointeurs ont été exclus du langage car ils permettent au programmeur d'accéder directement à la représentation mémoire des données. Pour générer un code réparti correct à partir d'un programme séquentiel utilisant des pointeurs il faut être capable de retrouver l'emplacement de la mémoire pointée après distribution des données. De plus, une zone de mémoire contiguë dans le programme séquentiel peut se trouver fragmentée et répartie entre plusieurs processeurs dans le programme réparti.

Les constructeurs structure et union n'ont pas été inclus dans le langage car d'une part leur intérêt est limité en l'absence de pointeurs et d'autre part cela nécessitait la définition d'une nouvelle expression de distribution pour les objets composites ainsi construits.

Le langage C-Pandore contient des fonctions qui peuvent être de deux types :

- des fonctions qualifiées de *closes* (sans effet de bord) que l'on peut considérer comme des opérateurs. Lors de la génération de code réparti, l'exécution de ces fonctions ne nécessite pas de communication entre processus en dehors du passage des paramètres et peuvent ainsi être exécutées de manière indépendante par chaque processus ;

- des "phases distribuées" dont les paramètres sont des variables distribuées passées par copie dont l'exécution en réparti peut nécessiter des communications et que nous présentons plus en détails dans la suite.

Un programme C-Pandore se compose d'un ensemble de déclarations de phases distribuées suivi de la déclaration d'un programme principal. Ce dernier est également considéré comme une phase distribuée où peuvent apparaître des appels aux différentes phases distribuées précédemment déclarées. L'imbrication des appels de phases est également possible.

Notre langage ne contient pas de spécification de parallélisme de contrôle d'une part parce que nous générons automatiquement du code parallèle et distribué à partir de code séquentiel sans analyse statique comme nous le montrons en 3.1.3 et d'autre part parce que notre schéma de compilation optimisé décrit en 4.5 ne s'applique que sur des nids de boucles parallèles qui peuvent être détectés à la compilation. Nous n'avons pas non plus inclus de spécifications de parallélisme de données car il n'était pas dans nos objectifs d'étudier des techniques de compilations spécifiques a ce type de parallélisme.

```
#define NBPROCS 4
#define N 1024
#define NLOOP 100

dist Jacobi(double B[N][N] by block (N, N/NBPROCS) map regular( 1, 0) mode INOUT)
  double A[N][N] by block (N, N/NBPROCS) map regular( 1, 0);
{
  double V,W;
  int I, J, K;

  W = 1.25000000000000e-01;
  V = 5.00000000000000e-01;

  for (K=0;K<NLOOP;K++) {

    for (I=1;I<(N-1);I++)
       for(J=1;J<(N-1);J++)
           A[I][J] = ((V*B[I][J])+(W*(((B[I][J-1]+B[I][J+1]
           +B[I-1][J])+B[I+1][J])));

     for(I=1;I<(N-1);I++)
       for(J=1;J<(N-1);J++)
         B[I][J] = A[I][J];
   }
}

main() {
  double Y[N][N];

  /* initialisation */

  Jacobi(Y);

  /* utilisation des résultats */
}
```

FIG. 2.4 – *Exemple de programme C-Pandore*

2.5.1 Procédures (phases distribuées)

La *phase distribuée* est une procédure[5] pour laquelle à chaque paramètre formel est associée une spécification de distribution: il n'y a pas d'héritage de distribution dans notre langage car celui-ci impose de réaliser une analyse statique inter-procédurale pour connaître la distribution d'un tableau en un point donné du programme; lorsque cette analyse échoue, il n'est plus possible d'appliquer de schéma de compilation optimisé. Il existe trois classes de paramètres: les tableaux distribués, les tableaux répliqués et les scalaires (qui sont répliqués sur chaque processus).

Les paramètres sont passés par recopie et peuvent être modifiés dans une phase distribuée. A tout paramètre d'une phase distribuée est associé un "mode" qui précise si la valeur du paramètre effectif est significative:

- à l'entrée de la phase (IN), dans ce cas sa valeur doit être recopiée lors du passage de paramètres au début de l'exécution de la phase distribuée;
- à la sortie (OUT), dans ce cas sa valeur doit être recopiée à la sortie de la phase distribuée;
- ou à l'entrée et à la sortie (INOUT), dans ce cas deux recopies sont nécessaires.

L'indication de ce mode guide le compilateur dans la génération des communications entre les processus.

Il est possible de déclarer des variables locales (au sens de la visibilité) distribuées dans l'en-tête de phase dont la portée est limitée au corps de la phase.

2.5.2 Distribution des tableaux

Lors du passage de paramètres, un tableau distribué est décomposé en "blocs" de tailles égales qui sont associés aux processeurs. Pour un tableau distribué donné, l'ensemble des blocs associés à un processeur constitue la "partition locale[6]" à ce processeur. Une spécification de distribution d'un tableau d'entiers V à la forme suivante:

$$int\ V[h_0]...[h_{n-1}]\ by\ block\ (s_0,...,s_{n-1})\ map \left| \begin{array}{l} regular \\ wrapped \end{array} \right| (d_0,...,d_{n-1})$$

Les deux étapes de la distribution sont respectivement spécifiées par la fonction de *décomposition* des dimensions du tableau (*block*) et la fonction de *placement* qui associe les blocs aux processeurs (*regular* ou *wrapped*). Cette séparation nous paraît une bonne solution pour ne pas mélanger la logique du programme (la décomposition des données) avec les contraintes de l'architecture (le placement). Celle-ci n'est pas convenablement réalisée dans HPF qui utilise une notion de processeur "virtuel[7] " pour décrire la décomposition.

5. Même si elle a la forme syntaxique d'une fonction.
6. Au sens de la mémoire.
7. En pratique. les processeurs virtuels correspondent aux processeurs réels

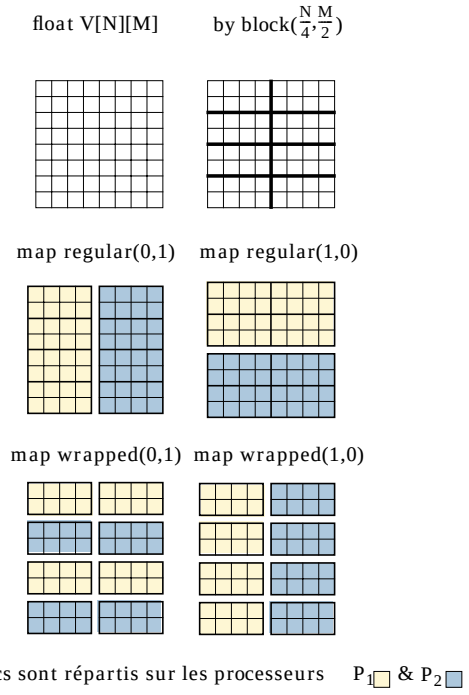


FIG. 2.5 – *Décomposition et placement des blocs de données*

La fonction "block" indique la taille de chacune des dimensions des blocs de données. Elle ne permet donc d'exprimer que des décompositions par blocs rectangulaires : d'autres types de décompositions, comme par exemple la constitution de blocs suivant une diagonale, ne sont pas exprimables en C-Pandore.

Les fonctions "regular" et "wrapped" précisent sur quels processeurs sont placés les blocs de données. La fonction "regular" constitue des groupes de blocs contigus en fonction du nombre de processeurs indiqué par le programmeur et du nombre de blocs générés par la "fonction de décomposition" ; elle place ensuite chaque groupe sur un processeur différent. La fonction "wrapped" associe les blocs aux processeurs un par un, en procédant de manière cyclique ; elle permet ainsi de placer sur un même processeur des blocs de données qui ne sont pas contigus dans le tableau original. La figure 2.5 illustre la *décomposition* et le *placement* d'un tableau à deux dimensions ainsi que la numérotation des blocs de données. Cela revient à considérer que l'architecture est un vecteur de processeurs et est sans doute un peu limitatif.

Dans les cas simples, les distributions exprimables avec C-Pandore sont les mêmes que celles de Fortran D, Vienna Fortran ou HPF avec un tableau de processeurs à une dimension mais dans C-Pandore le placement des blocs ne s'effectue pas dimension par dimension[8].

8. Nous savons construire des distributions presque aussi obscures que celles de HPF.

2.6 Comparaison de ces langages

2.6.1 Parallélisme

La boucle `FORALL` de Vienna Fortran et la spécification `$INDEPENDENT` de HPF permettent de décrire une boucle parallèle (parallélisme de contrôle). Elle permet d'indiquer au compilateur une absence de dépendances qui pourrait ne pas être détectée automatiquement. C'est donc une aide pour guider le compilateur dans les optimisations qui pourrait être facilement intégrée dans notre langage.

La boucle `FORALL` de HPF et Fortran D est une généralisation des opérations vectorielles de Fortran 90, c'est une véritable instruction de parallélisme de données qui a son intérêt propre indépendamment de la notion de distribution de données. De même, la possibilité d'exprimer des opérations de réduction a été ajoutée dans ces langages. Ceci permet aux compilateurs d'optimiser le traitement de telles opérations. Ces constructions sont absentes du langage C-Pandore: aucune opération spécifique au parallélisme de données n'a été incluse dans notre langage. Ceci restreint le modèle de programmation mais ne limite en rien l'intérêt de la distribution de données. L'ajout de telles primitives demanderait un travail important car la compilation de telles primitives et en particulier des réductions [Barreteau et Feautrier 95] est non trivial.

2.6.2 Distribution des tableaux

Bien que les fonctions de décomposition offertes par les trois langages soient similaires (par blocs ou cycliques), les langages HPF, Vienna Fortran et Fortran D ont un pouvoir d'expression de distribution supérieur à celui de C-Pandore. HPF et Fortran D possèdent notamment la notion d'*alignement* qui permet de caractériser la distribution d'un tableau en fonction de celles d'autres tableaux, en particulier ils offrent également la possibilité de répliquer des blocs selon certaines dimensions.

Lors de la définition du langage C-Pandore, nous avons choisi de privilégier la simplicité pour être capable de compiler efficacement les distributions exprimables. Nous aurions sans doute pu aller plus loin dans cette voie en prenant un sous-ensemble du langage Fortran D et en particulier exprimer la distribution dimension par dimension. De notre expérience, seuls les alignements manquent cruellement dans notre langage.

2.6.3 Procédures

Les procédures sont traitées dans les trois approches présentées. Les langages HPF et Vienna Fortran sont les moins restrictifs : ils autorisent l'imbrication d'appels (non récursifs), les paramètres pouvant hériter de la distribution du contexte d'appel ou être redistribués à l'entrée des procédures. Dans Fortran D et C-Pandore, les paramètres des procédures sont redistribués à l'entrée et recouvrent, en sortie, leur distribution d'avant l'appel. L'utilisation des spécifications `IN` et `OUT` nous paraît suffisante dans la plupart des cas pour limiter les redistributions inutiles.

2.6.4 Comparaison globale et critique

En cherchant à décrire ces langages je me suis demandé si la distribution de données devait être spécifiée seulement par des directives de compilation ou induire une modification de la syntaxe du langage. Les solutions proposées sont diverses: HPF propose du "tout directive" et C-Pandore du "tout langage". La bonne réponse est délicate à trouver car les spécifications de distribution modifient la sémantique opérationnelle du langage mais celle-ci garde une certaine équivalence avec la sémantique originale (sans distribution).

Dans le projet Pandore, il nous est paru plus élégant d'intégrer les spécifications de distribution au langage, mais c'était sans compter sur l'aspect psychologique: pour beaucoup d'utilisateurs potentiels nous avions créé un "nouveau" langage. Le choix de la syntaxe C était un choix techniquement bon (structure de bloc, pas de constructions désagréables comme les `COMMON`) mais qui a conduit à définir un langage vécu comme étrange (du C sans pointeurs).

Plus généralement cette réflexion amène à se poser des questions sur le choix des plate-formes d'expérimentation et le développement de prototypes et sur l'utilisation de "standards" ; nous y reviendrons dans le bilan qui est présenté dans la conclusion. Si "c'était à refaire" je travaillerai plutôt sur un sous-ensemble de Fortran; c'est pour ces raisons que je m'intéresse maintenant à la transformation de programmes JAVA plutôt qu'a la transformation de programmes EIFFEL, mais nous en reparlerons dans les perspectives.

Chapitre 3

Génération de code pour langages à distribution de données

Le schéma de compilation qui est décrit ici définit la transformation d'un programme séquentiel en un programme parallèle et réparti de type SPMD. Cette transformation que nous avons définie et mise en œuvre est dirigée par la distribution des données et suit la règle des "calculs locaux" : l'évaluation de l'expression permettant de réaliser une affectation sur la variable v est réalisée par l'ensemble[1] des processeurs propriétaires de v noté ici $owner(v)$.

Cette transformation peut être effectuée instruction par instruction (nous parlerons alors de schéma "de base") ou concerner des blocs ou des structures de contrôle plus grandes comme les boucles (nous parlerons alors de "schéma optimisé"). La suite de ce chapitre décrit le schéma de base qui a été mis en œuvre dans plusieurs compilateurs dont le premier prototype de Pandore. Le schéma optimisé fait l'objet du chapitre suivant.

3.1 Description du schéma de base

Habituellement, la description des méthodes de génération de code par distribution des données est explicitement liée à l'utilisation d'une bibliothèque d'échange de messages. La description de référence donnée par Callahan et Kennedy [Callahan et Kennedy 88] que nous rappelons en 3.4.1 bien qu'utilisant des opération de plus haut niveau (LOAD et STORE) n'est explicitée que pour le cas d'une mise en œuvre avec des messages. Nous avons formalisé cette description dans [Pazat 91] pour exprimer le schéma de compilation utilisé dans le compilateur Pandore.

1. Certaines variables peuvent être en partie voire totalement répliquées.

Par la suite, nous avons cherché à montrer qu'il était plus général d'exprimer ce schéma en utilisant la notion de copie. Nous avons ainsi pu unifier les descriptions de génération de code pour une bibliothèque d'échange de messages et pour l'utilisation d'une mémoire virtuellement partagée. Ceci n'est pas un simple "exercice de style" car grâce à cette description, nous avons pu modifier notre compilateur pour générer du code pour le système de mémoire virtuellement partagée Koan. La description que nous donnons ici est issue du travail de thèse de Yves Mahéo [Mahéo 95].

3.1.1 Synchronisations et masquage

Pour décrire le processus de compilation, nous faisons tout d'abord abstraction des problèmes d'accès aux variables. Ceux-ci seront abordés dans la partie suivante. Les problèmes plus techniques concernant la gestion de la mémoire seront détaillés en 3.2.4 et 3.3.2.

Pour une instruction $\mathcal{S}$ qui modifie un ensemble[2] de variables DEF($\mathcal{S}$) et qui lit un ensemble de variables USE($\mathcal{S}$) nous proposons de générer un code conforme au schéma suivant :

Synchroniser *owner*(DEF($\mathcal{S}$)) avec *owner*(USE($\mathcal{S}$))
Exécuter $\mathcal{S}$ sur *owner*(DEF($\mathcal{S}$))
Synchroniser *owner*(USE($\mathcal{S}$)) avec *owner*(DEF($\mathcal{S}$))

Où "synchroniser A avec B" signifie que A doit attendre B en ce point de contrôle.

La première synchronisation garantit que les variables de USE($\mathcal{S}$) qui seront lues ont une valeur conforme à celle prise avant $\mathcal{S}$ lors de l'exécution séquentielle. La seconde garantit que les variables de USE($\mathcal{S}$) ne sont pas modifiées avant d'être utilisées par *owner*(DEF($\mathcal{S}$)). "Exécuter $\mathcal{S}$ sur *owner*(DEF($\mathcal{S}$))" est le masquage qui permet de restreindre l'exécution de $\mathcal{S}$ pour suivre la règle des calculs locaux.

3.1.2 Copies de variables

Ce schéma de compilation peut être utilisé sur une machine à mémoire partagée où toutes les variables sont directement accessibles par tous les processus. Par contre, sur les machines à mémoire distribuée, la mémoire est "fragmentée" en un ensemble de mémoires "privées" qui ne sont accessibles que par le processeur qui leur est attaché. Dans ce cas, la distribution des données spécifie une localisation physique des données dans les mémoires privées. On parle alors de variables "locales" (présentes dans la mémoire privée d'un processeur) et de variables "distantes" (stockées dans la mémoire d'un autre processeur).

2. Toutes les variables de cet ensemble doivent être localisés sur le(s) même(s) processeur(s).

Nous pouvons maintenant préciser le schéma de compilation en faisant apparaître la notion de copie de variables :

(A)
Synchroniser *owner*(DEF($\mathcal{S}$)) avec *owner*(USE($\mathcal{S}$))
Copier USE($\mathcal{S}$) chez *owner*(DEF($\mathcal{S}$)) dans Copies(USE($\mathcal{S}$))
Exécuter $\mathcal{S}'$ sur *owner*(DEF($\mathcal{S}$))
Synchroniser *owner*(USE($\mathcal{S}$)) avec *owner*(DEF($\mathcal{S}$))

$\mathcal{S}'$ est l'instruction $\mathcal{S}$ dans laquelle on a remplacé les références aux variables USE($\mathcal{S}$) par leurs copies : $\mathcal{S}' = \mathcal{S}_{[\text{USE}(\mathcal{S})]\leftarrow\text{Copies}(\text{USE}(\mathcal{S}))]}$. L'opération "Copier V chez P" doit être bloquante pour P car l'exécution de $\mathcal{S}'$ utilise les copies effectuées par cette opération et non les variables USE($\mathcal{S}$).

Puisque $\mathcal{S}'$ utilise des copies de USE($\mathcal{S}$), la deuxième synchronisation peut être remontée juste après la copie (c'est-à-dire avant le masquage) :

(B)
Synchroniser *owner*(DEF($\mathcal{S}$)) avec *owner*(USE($\mathcal{S}$))
Copier USE($\mathcal{S}$) chez *owner*(DEF($\mathcal{S}$)) dans Copies(USE($\mathcal{S}$))
Synchroniser *owner*(USE($\mathcal{S}$)) avec *owner*(DEF($\mathcal{S}$))
Exécuter $\mathcal{S}'$ sur *owner*(DEF($\mathcal{S}$))

Sur cet exemple (qui ne prend que des scalaires pour simplifier les notations) on voit de manière évidente que la synchronisation peut être "remontée". Plus formellement on peut remarquer que l'anti-dépendance entre les affectations (2) et (3) a été transformée en une anti-dépendance entre (2bis) et (3).

```
Programme source:           Programme généré

      ent a, b

      distribuer a sur P1   ent a                  sur  P1
      distribuer b sur P2   ent b                  sur  P2
                            ent copie_a            sur  P2
(1)   a := 0                Exécuter a := 0        sur  P1

(2)   b := a                Synchroniser           P1 avec P2
                  (2bis)    Copier  a dans copie_a sur  P2
                  (2')      Exécuter b := copie_a  sur  P2
                  (2ter)    Synchroniser           P2 avec P1


(3)   a := 1                Exécuter a := 1        sur  P1
```

Ce qui donne l'exécution du code suivant sur chacun des processeurs (Post(e) "poste" un événement qui indique que le programme a atteint ce point de contrôle, et Wait(e) attend que l'événement e soit posté; la copie est réalisée par le processeur P2, ce qui est un exemple de copie bloquante pour P2) : les instructions (2') et (2ter) peuvent commuter.

```
P1:                                P2:
ent a                              ent b
                                   ent copie_a
a := 0                                                  (1)
Post(e1)                           Wait(e1)
                                   copie_a <-- a        (2bis)

                                   b := copie_a         (2')
Wait(e2)                           Post(e2)             (2ter)

a := 1                                                  (3)
```

La mise en œuvre de "Copier" dépend du support d'exécution dont on dispose : nous précisons cette mise en œuvre dans le cas de machines disposant d'une mémoire virtuellement partagée (MVP) et dans le cas de machines disposant de primitives d'échanges de messages. De la même façon on pourrait dériver une mise en œuvre pour un support d'exécution permettant la lecture ou l'écriture distante.

3.1.3 Propriétés du code généré

Le code généré respecte les dépendances Soient (S1) et (S2) sont deux instructions entre lesquelles il existe une relation de dépendance :

- si (S1) et (S2) s'exécutent sur le même processeur alors elles s'exécutent dans le même ordre que dans le programme original (le schéma de transformation ne ré-ordonnance pas le code) et aucune synchronisation n'est générée ;
- si (S1) et (S2) s'exécutent sur des processeurs différents
  - il ne peut y avoir de dépendance de sortie entre (S1) et (S2) de par la règle des calculs locaux ;
  - si il existe une dépendance directe (S1) $\rightarrow$ (S2) celle-ci est respectée par la première synchronisation ;
  - si il existe une anti-dépendance (S1) $\leftarrow$ (S2) celle-ci est respectée par la seconde synchronisation.

Le code généré est parallèle Les synchronisations générées ne font intervenir que les processeurs concernés par la copie de donnée(s) à réaliser.

Lorsqu'il n'existe pas de dépendances entre deux instructions (S1) et (S2) qui s'exécutent sur des processeurs différents, leur exécution se déroule en parallèle si et seulement si les variables lues par (S1) (resp. (S2)) sont sur un processeur différent de celui où se trouvent les variables écrites par (S2) (resp. (S1)). Autrement dit, le code généré est parallèle: ce schéma de transformation réalise une "parallélisation par distribution des données". Dans le schéma "de base", Les calculs des ensembles USE(S), DEF(S), de Owner et donc l'introduction des synchronisations sont effectués à l'exécution; la parallélisation est donc effectuée à l'exécution. Dans l'exemple suivant, le code généré s'exécutera en parallèle puisque les données référencées sont locales (sauf lors de la dernière itération exécutée par le processeur P(1)). P(1) exécutera les itérations 1 à 50 et P2 les itérations I=51 à 99 (après avoir parcouru les itérations I=1 à 50 sans effectuer aucune opération):

```
!HPF$ PROCESSORS(2):: P
      REAL DIMENSION(100):: A,B
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A,B
      DO I=1,99
        A(I) = B(I) + B(I+1)
      ENDDO
```

Le code contient des synchronisations inutiles Il peut néanmoins y avoir génération de synchronisations inutiles car une (première) synchronisation est générée pour tout échange de données dès que $\text{Owner}(\text{USE}(S2)) \cap \text{Owner}(\text{DEF}(S1)) \neq \emptyset$; or cela ne signifie pas qu'il y ait dépendance, c'est-à-dire que $\text{USE}(S2) \cap \text{DEF}(S1) \neq \emptyset$. Dans l'exemple suivant, la transformation générera un code qui s'exécutera de manière séquentielle car A(50) se trouve sur P(1) et est référencée sur P(2) lors de la première itération que P(2) doit effectuer (pour I=51). P(2) sera donc bloqué jusqu'à ce que P(1) ait atteint l'itération (I=51):

```
!HPF$ PROCESSORS(2):: P
      REAL DIMENSION(100):: A,B
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A,B
      DO I=2,100
        A(I) = B(I-1) + B(I)
      ENDDO
```

De même, la seconde synchronisation est générée même lorsqu'il n'existe pas d'anti-dépendance dès que $\text{Owner}(\text{USE}(S1)) \cap \text{Owner}(\text{DEF}(S2)) \neq \emptyset$.

3.2 Application du schéma sur mémoire virtuellement partagée

Sur une machine à mémoire virtuellement partagée (MVP), la mémoire étant physiquement distribuée, il y a copie pour la lecture d'une donnée distante. Cette copie peut être réalisée implicitement par le mécanisme de défaut de page du système ou être réalisé explicitement.

La génération de code pour MVP repose habituellement sur la distribution dirigée par le contrôle (les itérations sont réparties entre les différents processeurs). Pour limiter le nombre de défauts de page il est nécessaire d'optimiser la localité des accès; ce qui peut être réalisé sans analyse statique en liant explicitement la répartition des itérations d'un nid de boucles avec le placement des données induit par la distribution des itération d'une autre boucle. Cette technique dite d'affinités (*cf* [Bodin 97], chapitre *optimisation de code pour MVP*) revient à exprimer une distribution de code dirigée par les données de manière indirecte (et à mon avis peu claire). Il me semblerait préférable d'utiliser explicitement la distribution de données et d'utiliser le shéma proposé ici.

3.2.1 Copies implicites

L'utilisation triviale d'une mémoire virtuelle partagée consiste à considérer la mémoire comme partagée et donc à n'utiliser que le mécanisme de copie implicite. Le schéma de compilation du 3.1.1 s'applique donc directement puisque la copie des données nécessaires à l'exécution de $\mathcal{S}$ est implicitement réalisée lors de l'exécution de l'instruction $\mathcal{S}$.

Pour les synchronisations on peut utiliser une opération de synchronisation symétrique (barrières de synchronisation) que l'on trouve de manière courante sur les mémoires virtuelles partagées. Bien que cette synchronisation soit inutilement forte, son emploi n'entraîne pas de sur-coût notable en pratique (*cf* [Mahéo 95] p. 114–117).

Cette mise en œuvre est trivialement exemple d'inter-blocage de par la symétrie de l'opération de synchronisation.

3.2.2 Copies explicites

Transfert de pages

La plupart des mémoires virtuellement partagées permettent de réaliser des copies explicites de pages. Ceci est intéressant en particulier pour effectuer des diffusions efficaces qui sont nécessaires lorsqu'une variable écrite est répliquée. Si on n'utilisait que les copies explicites, il faudrait désactiver les mécanisme de gestion de cohérence qui risqueraient de réaliser des invalidations ou des copies "intempestives". On utiliserait donc uniquement les mécanismes de transformation d'adresses de la MVP ce qui ne tire pas pleinement partie des mécanismes de la MVP. Le schéma (B) s'appliquerait alors exactement comme lorsque l'on utilise une bibliothèque d'échange de messages 3.3.

Transfert de données élémentaires

Pour l'application du schéma de base, la granularité des échanges de données effectués par les mécanismes de mémoire virtuelle (page) n'est pas adapté à la granularité du code généré (traduction instruction par instruction). De plus, le coût des synchronisations étant du même ordre que le coût d'un envoi de messages, il est préférable (lorsque c'est possible) d'utiliser la technique proposée en 3.3.1 pour les échanges de données.

3.2.3 Copies explicites et implicites

Il est intéressant de mixer les deux approches: la copie explicite est utilisée pour les diffusions et la copie implicite dans les autres cas.

Si l'on conserve le mécanisme de cohérence forte [Lahjomri et Priol 92] sur l'ensemble du programme, le schéma de compilation (A) du 3.1.2 s'applique. La deuxième synchronisation ne peut pas être déplacée car si le possesseur de USE($\mathcal{S}$) n'effectuait pas d'attente, il serait susceptible de modifier ces variables et donc le système pourrait modifier ou invalider leurs copies implicitement chez DEF($\mathcal{S}$) avant la fin de l'exécution de $\mathcal{S}'$.

3.2.4 Représentation des données

L'utilisation de la mémoire virtuellement partagée possède tout de même un atout : la transformation des accès globaux en accès locaux est pratiquement gratuite[3] puisqu'elle est gérée par le matériel (MMU).

Données distribuées : Les données distribuées doivent bien évidemment être placées dans les régions partagées pour pouvoir utiliser les mécanismes de la mémoire virtuellement partagée. Il faut alors faire la correspondance entre la notion de propriétaire d'une page au sens de la mémoire (seul processeur autorisé à écrire dans cette page) et possesseur d'un bloc au sens du schéma de compilation (processeur responsable des écritures dans ce bloc).

La solution la plus simple consiste à faire coïncider ces deux notions. Ceci peut être facilement réalisé dans le cas des tableaux ne possédant aucun élément répliqué et ayant au moins une dimension non distribuée: l'idée est d'étendre une de ces dimension pour que la taille d'un bloc soit un multiple de la taille des pages. Une autre possibilité serait de restreindre les distributions au niveau du langage pour que la taille des blocs soit multiple de la taille des pages, mais cela nuirait à la portabilité des programmes.

Lorsque toutes les dimensions sont distribuées ou lorsque certains éléments sont répliqués, le placement des tableaux dans les pages est plus complexe à définir. Une mise en œuvre intégrant le cas où toutes les dimensions d'un tableau sont distribuées a été expérimentée dans le compilateur Pandore.

3. Il faut quand même tenir compte d'une éventuelle différence de représentation des données entre le programme source et le programme SPMD généré.

Le placement initial des pages ne correspond pas nécessairement à la distribution spécifiée car les mémoires virtuelles partagées ne permettent pas de définir de distributions aussi riches que celles de notre langage; ceci n'a pas d'importance car dès le premier accès en écriture, une page "trouvera sa place" grâce au mécanisme de défaut de page.

Données répliquées : Les données totalement répliquées doivent être placées en dehors de la mémoire virtuelle, (i.e. dans la partie locale de la mémoire de chaque processeur qui est privée). En effet, si l'on suit la règle des écritures locales tous les accès à ces variables sont des accès locaux puisque chaque processeur en possède une copie. De plus, ces données n'étant pas distribuées, elles ne sont pas fragmentées en blocs : on peut utiliser la même représentation mémoire dans le code généré que dans le programme source. Aucune transformation d'adresse n'est alors nécessaire.

3.3 Application du schéma sur bibliothèque de communication par messages

Classiquement, le schéma (B) est appliqué pour la génération de code lorsque l'on dispose d'une bibliothèque de communication par passage de messages.

3.3.1 Fonctions à mettre en œuvre

Afin de rester compatibles avec les notations adoptées dans [Pazat 91] et [Bareau et al. 93], nous regroupons les 3 premières phases dans une seule opération (*Refresh*), la dernière phase étant appelée *Exec*.

Le schéma de compilation de S s'exprime alors comme suit :

$$\begin{array}{l} \mathit{Refresh}(\mathrm{USE}(\mathcal{S}),\ \mathrm{DEF}(\mathcal{S}),\ \mathrm{Copies}(\mathrm{USE}(\mathcal{S}))) \\ \mathit{Exec}(\mathrm{DEF}(\mathcal{S}),\ \mathcal{S}') \end{array}$$

L'opération *Refresh* permet d'obtenir des copies à jour des variables distantes (en rafraîchit les copies) et *Exec* applique la règle des calculs locaux.

L'opération *Exec* est simplement mise en œuvre par un test sur la possession de la donnée :

$$\textbf{si } P \in \mathit{owner}(\mathrm{DEF}(\mathcal{S})) \textbf{ alors } \mathcal{S}'$$

et $\mathcal{S}' = \mathcal{S}_{[\mathrm{USE}(\mathcal{S})] \leftarrow \mathrm{Copies}(\mathrm{USE}(\mathcal{S}))]}$.

Une mise en œuvre possible de l'opération *Refresh* est la suivante :

$$\begin{array}{l} \textbf{si } P \in \mathit{owner}(\mathrm{DEF}(\mathcal{S})) \text{ et } P \notin \mathit{owner}(\mathrm{USE}(\mathcal{S})) \\ \quad \textbf{alors } \mathrm{Recevoir}(\mathrm{USE}(\mathcal{S}) \text{ de } \mathit{owner}(\mathrm{USE}(\mathcal{S}))) \text{ dans } \mathrm{Copies}(\mathrm{USE}(\mathcal{S})) \\ \textbf{si } \ P \in \mathrm{USE}(\mathcal{S}) \text{ et } P \notin \mathrm{DEF}(\mathcal{S}) \\ \quad \textbf{alors } \mathrm{Envoyer}(\mathrm{USE}(\mathcal{S}) \text{ à } \mathit{owner}(\mathrm{DEF}(\mathcal{S}))) \end{array}$$

"Recevoir" est la réception bloquante qui permet à la fois :

- la mise en œuvre de la première synchronisation (réalisée par l'exécution de Envoyer par $owner(\mathrm{DEF}(\mathcal{S}))$ et de Recevoir par $owner(\mathrm{USE}(\mathcal{S}))$) ;
- la réalisation bloquante de la copie pour $owner(\mathrm{USE}(\mathcal{S}))$.

Pour mettre en œuvre la seconde synchronisation, il faut utiliser pour "Envoyer" l'émission non bloquante et tamponnée (i.e. l'émetteur est libéré dès que le message à été envoyé ou recopié par le système).

Les communications se font par files fifo. Pour montrer que cette mise en œuvre est libre d'inter-blocage nous devons utiliser l'hypothèse de files fifo infinies car la détermination de la taille des files nécessaire à une exécution est indécidable dans le cas général [Caillaud 94]. La preuve l'absence d'inter-blocage est donnée dans [Bareau et al. 93].

L'hypothèse d'existence de files fifo infinies peut être gênante dans certaines mises en œuvre et il peut être souhaitable de disposer d'une détection de saturation des fifos, ce que, malheureusement, très peu de systèmes fournissent.

3.3.2 Représentation des données

Le schéma de compilation présenté ne tient pas compte de la représentation mémoire utilisée. Or celle ci doit être précisée car elle intervient dès la compilation d'une part pour la transformation des accès aux données (transformation de $\mathcal{S}$ en $\mathcal{S}'$) et d'autre part pour la gestion des emplacements mémoire utilisés pour les copies des variables distantes.

Nous distinguons deux types de données différents :

- les données qui ont été affectées à un processeur par la spécification de distribution (que nous appellerons "locales")
- les copies temporaires de données distantes ($\mathrm{Copies}(\mathrm{USE}(\mathcal{S}))$) effectuées lors de la phase d'échange (*Refresh*) que nous appellerons "rapatriées".

Pour toutes les données , il faut réaliser la transformation des accès exprimés dans l'espace de nommage global du programme source vers un domaine local. Ce domaine local est une représentation d'un sous-ensemble de l'espace global et a en général une représentation mémoire différente de celle du programme source comme dans l'exemple de la figure 3.1.

De plus, la représentation des données "rapatriée" peut également être différente de leur représentation "locale". On peut par exemple utiliser des variables temporaires auxquelles on accède directement.

On peut chercher à distinguer ces accès dans le code généré dès la compilation (ce qui n'est pas toujours possible), ou utiliser une représentation mémoire uniforme pour toutes les données (au moins au moment de la phase *Exec*).

Les représentations mémoire plus évoluées (adaptées aux accès ou uniformes et autres que la duplication) sont plus complexes à mettre en œuvre et peuvent nécessiter des analyses fines des accès à la compilation. Ces représentations seront étudiées dans le chapitre suivant.

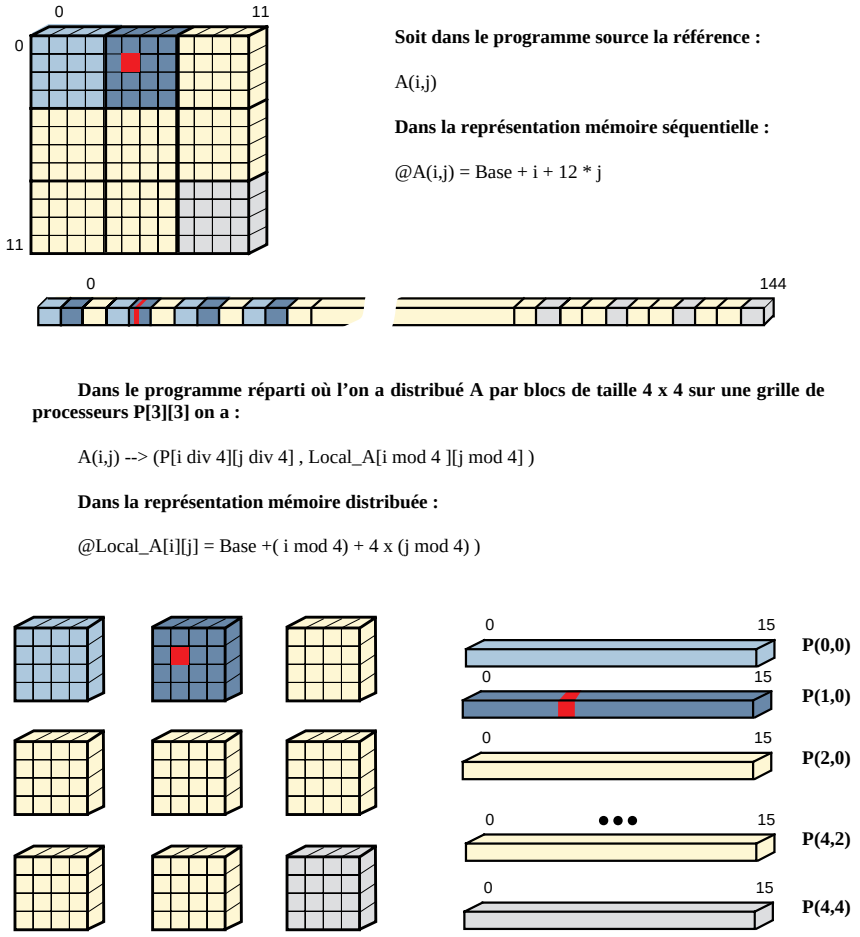


FIG. 3.1 – *Transformation des accès*

3.4 Autres travaux

3.4.1 Fortran D

La première description du schéma de compilation à été donnée dans l'article de Callahan et Kennedy [Callahan et Kennedy 88] et a servi de base au projet Fortran D [Hiranandani et al. 91]. Dans cet article, le schéma de compilation est décrit en tenant compte à la fois de la notion de distribution et de la représentation mémoire. Deux fonctions sont introduites : δ et α. La fonction δ est similaire à la fonction *owner* introduite au début de ce chapitre mais ne peut pas prendre en compte la notion de variable partiellement répliquée (elle rend l'identité d'un processeur ou 0 si la variable est répliquée). La fonction α représente la transformation

d'adresse de l'espace global vers l'espace local.

La compilation de `M0 = f( M1, ..., Mk)` correspond au pseudo-code suivant :

```
LOAD( M1, t1,δ(M0) )
LOAD( M2, t2,δ(M0) )
...
LOAD( Mk, tk,δ(M0) )
STORE(M0, f(t1,...,tn))
```

où `t1,...,tk` sont des variables temporaires et `P` représente l'identité du processeur qui exécute le code.

`LOAD(Mi,ti,Q)` se traduit par :

```
Cas
    δ(Mi)= P :
        /* on possède la référence (qui n'est pas répliquée) */
        si P = Q
          alors  ti← α(Mi)
          sinon  si Q ≠ 0
                    alors  envoyer α(Mi) à Q
                    sinon
                           t← α(Mi)
                           envoyer α(Mi) à tous les autres processeurs
    δ(Mi)≠ P et δ(Mi)≠ 0 :
        /* on ne possède pas la référence (qui n'est pas répliquée) */
        si P = Q ou  Q = 0
          alors  recevoir ti de δ(Mi)
    δ(Mi)= 0 :
        /* la référence est répliquée) */
        si P = Q ou  Q = 0
          alors  ti← α(Mi)
```

et `STORE M0=f(t1,...,tk)`

```
si δ(M0) = P ou  δ(M0) = 0
  α(M0)←f(t1,...,tk)
```

La représentation mémoire d'un tableau est ici constituée du bloc local et de variables temporaires. En pratique cette représentation mémoire n'est guère utilisée dans le compilateur Fortran D actuel qui optimise le code généré (voir au 4.3).

Les communications sont assurées par les primitives de la machine cible.

3.4.2 Superb et Vienna Fortran

Dans la thèse de Michael Gerndt [Gerndt 90], la description du schéma de compilation sépare plus clairement la représentation mémoire : celle-ci est supposée uniforme pour les données locales ou rapatriées. La mise en œuvre par la technique des *overlaps* sera présentée en 4.3.

Pour chaque instruction $\mathcal{S} \equiv$ `ref`$_0$ `= f(` `ref`$_1$`, ..., ref`$_k$`)` chaque référence en lecture `ref`$_i$ génère le code `EXCH(`$m_{\mathcal{S}}$`, ref`$_i$`)` , où $m_{\mathcal{S}}$ est un masque (une expression booléenne `P = owner(ref`$_0$`)`), qui est similaire à notre `Refresh`.

La règle des écritures locales est assurée en faisant précéder chaque instruction d'une garde ($\mathcal{S}$ devient $m_{\mathcal{S}} \rightarrow S$), ce qui est similaire à notre `Exec`.

Les communications sont assurées par les primitives de communications offertes par la machine cible.

3.5 Pandore

Notre description du schéma de compilation sépare la représentation mémoire de la même manière que Vienna Fortran; cette description est celle donnée en 3.3.1. Dans sa mise en œuvre, nous séparons également dès la compilation le cas des variables en partie gauche répliquées du cas des variables distribuées car cela est toujours possible dans notre langage [4]. Notre exécutif comporte donc deux fonction `Refresh` (`refresh_dist` et `refresh_rep`) et deux fonctions `Exec` (`exec_dist` et `exec_rep`).

La représentation mémoire que nous avions initialement choisie ([Thomas 91] et [Chéron 93] p. 72–77) consistait à représenter la partition locale de chaque tableau sous une forme linéarisée et à utiliser des variables temporaires pour les données rapatriées. Ainsi, une référence à `B[i]` était traduite par `(B, i div block_size, i mod block_size)` ; l'accès à une donnée locale se faisait alors par `Local_B[i mod block_size]` et le calcul du propriétaire était réalisé à partir de la valeur de `i div block_size`. La mise en œuvre des `refresh` utilisent une variable temporaire pour stocker la valeur de chaque référence en lecture à une variable distribuée, la durée de vie de ces temporaires était limitée à l'exécution de l'affectation. Ainsi l'affectation `x = B[j] + C[j+1]` dans laquelle le scalaire x est répliqué et les tableaux B et C sont découpés en blocs de 10 éléments sera compilé en :

```
{
 int tmp1, tmp2;
 refresh_rep(tmp1, B, j div 10, j mod 10);
 refresh_rep(tmp2, C, (j+1) div 10, (j+1) mod 10);
 exec_rep(x, tmp1 + tmp2);
}
```

Les expressions sur j correspondent aux couples (numéro de bloc, indice dans le bloc). Les temporaires sont alloués [5] par le mécanisme d'ouverture de bloc (`{`) et la déclaration de deux variables locales à ce bloc.

Les communications sont assurées par des fonctions de la bibliothèque de communication POM développée dans l'équipe PAMPA [Guidec et Mahéo 95a] qui est portable sur un grand nombre de machines à mémoire distribuée et possède des mécanismes d'observation et de traces.

4. Seuls les tableaux explicitement répliqués et les variables scalaires sont répliqués.
5. Du moins leur portée est indiquée au compilateur.

3.6 Pourquoi cela ne suffit pas

Nous avons vu dans ce chapitre une description d'un schéma de transformation de programmes permettant de transformer un programme impératif séquentiel en un programme parallèle et réparti. Ce schéma a été mis en œuvre dans les compilateurs existants ainsi que dans notre compilateur Pandore et permet de transformer tout programme séquentiel en programme parallèle et distribué. Il permet une utilisation transparente de l'ensemble de la mémoire distribuée de la machine et réalise une parallélisation du code sans analyse de dépendances. C'est donc un élément essentiel de tout compilateur pour langages à distribution de données.

A lui seul, ce schéma ne permet néanmoins pas de fournir des codes ayant des performances comparables à un code écrit manuellement. A moins de disposer d'un grand nombre de processeurs, les codes ainsi générés sont même parfois moins efficaces (en terme de temps) que le code séquentiel original d'une part à cause de la trop faible granularité des échanges mais surtout du fait du coût de l'évaluation de la fonction *Owner* pour chaque instruction du programme.

Le chapitre suivant propose des techniques permettant de produire des codes plus performants (en terme de temps d'exécution) pour les nids de boucles. Ces techniques sont compatibles avec "le schéma de base" qui est généralement utilisé lorsqu'aucun schéma optimisé ne peut s'appliquer.

Chapitre 4

Génération de code efficace

4.1 Introduction

Afin de valider l'intérêt de la génération de code par distribution de données, nous avons étudié la génération de code réparti ayant une efficacité comparable à l'écriture de code "à la main".

Après avoir précisément recensé les sources d'inefficacité du prototype PANDORE 1, nous avons cherché à définir et à mettre en œuvre des techniques qui soient les plus générales possibles : en d'autres termes nous n'avons pas réalisé un compilateur de *benchmarks*.

Dans l'optique de la réalisation d'un compilateur de niveau industriel il faudrait également associer à ces techniques un grand nombre d'optimisations ad-hoc mais le coût de développement d'un tel outil serait de beaucoup supérieur au coût du projet PANDORE. Pour donner un ordre de grandeur, dans le projet européen PREPARE, nous avons participé à la réalisation d'un compilateur HPF. Le budget représentait 50 personnes × années pour la réalisation d'un prototype industriel contre environ 15 personnes × années pour le projet PANDORE (4 thèses + encadrement) qui est avant tout un projet de recherche.

Le choix de solutions générales pour les optimisations nous a permis de mener un travail de recherche (thèses de Marc Le Fur et de Yves Mahéo) qui montre l'intérêt des techniques développées et donne une portée supérieure au seul développement d'un prototype. La conséquence négative de ce choix est que le compilateur que nous avons réalisé est parfois moins efficace que d'autres compilateurs utilisant des techniques moins évoluées, faute d'optimisations ad hoc pour les *benchmarks*.

Le type de langage que nous avons choisi (impératif séquentiel) nous a amené de manière naturelle à l'étude de l'efficacité des codes contenant des itérations et des structures de données tableaux. De nombreux codes de calcul scientifique utilisent intensivement ces structures, ce qui justifie la restriction de cette étude.

Nous avons étudié et mis en œuvre de techniques de compilation et un support d'exécution qui sont optimisés pour les nids de boucles commutatifs, qui incluent les nids de boucles parallèles.

4.2 Critères d'optimisation

Pour générer le code réparti correspondant à un nid de boucles parallèle dans un programme, il est classique de générer un code à deux phases :

- une phase d'échange de données pendant laquelle on réalise toutes les communications nécessaires aux calculs ;
- une phase de calcul lors de laquelle, pour chaque processeur, toutes les données référencées sont présentes localement.

Pour chacune de ces phases, il faut définir quel est l'ensemble de données qui doit être parcouru lors de l'exécution. Nous étudions ici des schémas de compilation pour lesquel la définition et le parcours de ces ensembles de données est calculable statiquement [1].

La représentation mémoire des données est également un point crucial dans la génération de code réparti. Celle-ci doit permettre de représenter les données distribuées et leurs éventuelles copies de manière à ce que leur accès puisse être efficacement réalisés par le compilateur.

4.2.1 Définition et parcours des ensembles

La règle des calculs locaux suppose que l'on exécute une instruction sur le processeur qui possède la donnée qui est modifiée. Ceci suppose qu'une instruction ne modifie qu'une donnée ou que toutes les données modifiées par une instruction se trouvent sur un même processeur.

Prenons l'exemple d'une boucle contenant une seule affectation très simple :

```
for i in 𝒟
  A[f(i)] := B[g(i)]
```

où $\mathcal{D}$ représente le domaine d'itération.

La distribution des données affecte à un processeur p des éléments de `A` et `B`. Nous notons :

$$Owned_{\mathtt{A}}(p) = \{\mathtt{i} \mid \mathtt{A[f(i)]} \text{ est placé sur } p\}$$
$$Owned_{\mathtt{B}}(p) = \{\mathtt{i} \mid \mathtt{B[g(i)]} \text{ est placé sur } p\}$$

Phase de calcul

La partie du domaine d'itération pour laquelle le processeur p doit exécuter l'affectation `A[f(i)] := B[g(i)]` est :

$$Compute(p) = \{\mathtt{i} \in \mathcal{D} \mid \mathtt{f(i)} \in Owned_{\mathtt{A}}(p)\}$$

L'énumération des points de la partie du domaine d'itération pour lequel un calcul doit être réalisé par un processeur doit être exact et non redondant. En effet, un calcul ne peut être fait que s'il est local et ne peut en général être effectué

1. Cela ne veut pas dire que ces ensembles sont connus statiquement.

plusieurs fois sans produire de résultat différent (ie `x:=x+1` ne peut être exécuté plusieurs fois).

Pour chaque processeur, les données lues ont été rendues accessibles grâce à la phase d'échange. Il ne reste plus qu'à réaliser l'accès à ces données durant cette phase. Pour résoudre l'accès à des données pouvant être des données locales ou des copies de données distantes, les deux solutions envisagées dans le cas non optimisé présenté en 3.3 peuvent être étudiées :

- soit distinguer ces accès dans le code généré dès la compilation (ce qui nécessite de fragmenter le code généré en autant de sous domaines qu'il existe de cas à distinguer [2]) ;
- soit avoir une représentation mémoire uniforme pour toutes les données.

Phase d'échange de données

La phase d'échange de données a pour rôle de rendre disponible localement toutes les données nécessaires à l'exécution d'une instruction. Dans notre exemple, pour le processeur p ces données sont celles dont les références sont contenues dans l'ensemble :

$$Viewed_{\mathtt{B}}(p) \;=\; \{j \,|\, \exists \mathtt{i} \in Compute(p)\,,\, j = \mathtt{g(i)}\}$$

L'ensemble des références qui doivent être communiquées d'un processeur p à un processeur p' est donc :

$$Comm(p,p') \;=\; Owned_{\mathtt{B}}(p) \cap Viewed_{\mathtt{B}}(p')$$

Dans la phase d'échange de données, il n'est pas nécessaire de limiter la communication à $Comm(p,p')$ qui représente les échanges nécessaires à la phase de calcul ; autrement dit on peut définir un ensemble de communication $Comm^{+}(p,p')$ tel que :

$$Comm^{+}(p,p') \;\supseteq\; Comm(p,p')$$

Il faut réaliser un compromis entre les propriétés et la taille de cet ensemble $Comm^{+}(p,p')$ en tenant compte à la fois du coût à l'exécution d'une énumération exacte des données à échanger et du sur-coût à l'exécution d'un échange de données inutiles.

La définition de cet ensemble et la méthode d'énumération de ses points doivent optimiser les facteurs suivants.

- Minimiser le coût de l'énumération : une énumération point par point de $Comm^{+}(p,p')$ peut être inutilement coûteuse. Dans certains cas une vectorisation des communications à la compilation est possible. On peut également choisir de ne réaliser l'énumération des données à échanger que sur les processeurs qui doivent envoyer des données et coder l'identité des données dans les messages.

2. La solution consistant a effectuer cette distinction lors de l'exécution est trop coûteuse et annulerait le gain apporté par l'optimisation du code de calcul.

- Minimiser le nombre de messages afin de diminuer l'effet de la latence : il est très souvent préférable de regrouper les données à échanger plutôt que d'associer un message à chaque donnée élémentaire. Ceci peut se faire par vectorisation (on échange alors des blocs de mémoire contigus même si toutes les données de ce vecteur ne sont pas utiles) ou par agrégation (lorsque la vectorisation n'est pas possible on peut toujours regrouper les données à échanger entre couples de processeurs).

- Minimiser la taille des messages : un message doit contenir une information aussi compacte que possible et il faut donc éviter les échanges de données inutiles ($Comm^+(p,p')$ ne doit pas être trop grand devant $Comm(p,p')$). Si on veut éviter de coder l'identité des données dans les messages, il est nécessaire d'énumérer pour chaque couple de processeurs (p,p'), les données à envoyer de p vers p' et celles que p' doit recevoir de p dans le même ordre[3],

- Minimiser les codages et décodages des données dans les messages afin de minimiser les recopies mémoire qui sont toujours coûteuses : dans le cas idéal le tampon de communication représente exactement un bloc de mémoire chez l'émetteur et chez le récepteur. Ceci est compatible avec la vectorisation mais pas avec l'agrégation.

4.2.2 Représentation mémoire

Les variables du programme source sont distribuées dans le programme généré. Pour les tableaux on doit donc passer d'une représentation globale (nom, vecteur d'indices) à un ensemble de représentations locales correspondant pour chaque processeur :

- à la représentation de la partition locale qui lui est affectée par la directive de distribution ;

- à des emplacements mémoire permettant de stocker temporairement des valeurs d'éléments appartenant à la partition locale d'un autre processeur ;

- à la description du placement de l'ensemble des partitions locales qui doit être accessible sur chaque processeur.

Le choix de la représentation locale doit permettre d'optimiser plusieurs paramètres :

compacité : la représentation de la partition locale doit être aussi compacte que possible et être compatible avec l'augmentation du nombre de processeurs (*scalability*) afin de profiter de l'extensibilité mémoire de ces machines ;

rapidité : lorsque la transformation d'adresse (global vers local) ne peut être réalisée statiquement, celle-ci doit être aussi peu coûteuse que possible ; de même,

3. Il n'existe pas d'ordre total sur les messages, par contre nous supposons que les messages ne se doublent pas.

la détermination du processeur possédant un élément doit également être très rapide ;

contiguïté : il est souhaitable que deux éléments contigus en mémoire dans la représentation séquentielle restent contigus dans la représentation répartie dans le cas où les éléments se trouvent dans le même bloc. Cela permet de mieux utiliser le cache, de faciliter la vectorisation des messages et de ne pas briser les optimisations faites au niveau du programme source ;

uniformité : l'accès aux éléments reçus d'un autre processeur doit être identique à l'accès aux éléments locaux si on veut éviter de séparer les calculs locaux des calculs utilisant des valeurs distantes ;

globalité : tant qu'un tableau n'est pas redistribué, sa représentation mémoire doit rester identique à elle même. Changer cette représentation en fonction des accès (par exemple selon le nid de boucle) peut s'avérer inutilement coûteux.

L'optimisation globale de la représentation mémoire résulte d'un compromis entre l'optimisation de chacun de ces paramètres. Notre représentation (voir en 4.6) satisfait aux critères d'uniformité et de globalité. Elle optimise la rapidité des accès et est raisonnablement compacte. La contiguïté n'est par contre pas toujours préservée à cause du choix de linéarisation des tableaux qui n'est pas standard.

4.3 Autres travaux

Les travaux sur la compilation efficace de langages à distribution de données portent essentiellement sur la compilation des boucles parallèles. Nous relatons brièvement ici les travaux sur la compilation de telles boucles dans le cas où les fonctions d'accès sont "régulières". Une étude plus détaillée est fournie dans [Coelho et al. 96].

Dans tous les cas le problème est d'abord de calculer les ensembles de données à envoyer, recevoir ou calculer ; puis de décrire le parcours de ces ensembles. Enfin, la représentation mémoire utilisée pour les partitions locales et les données reçues intervient dans la manière de parcourir ces ensembles ; c'est pourquoi nous commençons par l'étude des représentations mémoire.

4.3.1 Représentation mémoire

Représentation des partitions locales

La technique adoptée dans le compilateur Fortran D, comme dans Vienna Fortran [Benkner 94], consiste à représenter les partitions locales comme des tableaux ayant le même nombre de dimensions que le tableau initial mais dont la taille est divisée par le nombre de processeurs. Pour un tableau à une dimension, la transformation d'adresse prend donc la forme suivante :

$$\begin{array}{lcll} i & \rightarrow & i - \text{no processeur} \times \text{taille du bloc} & \text{pour les distributions par bloc} \\ i & \rightarrow & (i-1)\,div\,B + 1 & \text{pour les distributions cycliques} \end{array}$$

Dans des cas simples (fonctions d'accès unimodulaires), le compilateur Fortran D est capable de réaliser directement le parcours sur les indices locaux, ce qui évite de payer le coût de la transformation d'adresse à chaque accès. Dans Vienna Fortran, le cas des affectations entre sections de tableaux est traité spécifiquement et il est possible de représenter les partitions locales par des sections de tableau. Cette technique ne règle pas toujours efficacement le problème du compactage des partitions locales dans le cas de distributions cycliques car la fonction de transformation d'adresse est coûteuse.

La méthode proposée par Chaterjee et al [Chaterjee et al. 93] compacte les partitions locales et calcule la fonction de transformation d'adresse (global vers local) sous forme d'un automate d'états fini qui rend la valeur de l'incrément (on remplace le $i = i + 1$ implicite d'une boucle DO par $i = i + \Delta_l(i)$). Cet automate prend en compte la notion de cycle, ce qui permet de restreindre la taille de la table Δ_l.

Représentation des données reçues

Une solution simple revient à allouer localement uniquement les partitions locales (encore que ceci n'est pas toujours réellement trivial avec les distributions permises dans le langage HPF) et à utiliser des variables temporaires pour stocker les données reçues. Cette méthode a l'avantage d'être simple et de produire une représentation locale indépendante des accès, donc unique pour toute la durée d'un programme.

Dans le compilateur Fortran D, des tampons de réception sont alloués pour recevoir les données distantes lorsque celles-ci ne sont pas contiguës à la partition locale [Tseng 93]. Dans ce cas il est nécessaire que le compilateur sépare les accès totalement locaux (expressions ne faisant référence qu'à des données locales) des accès à des variables reçues qui se trouvent dans les tampons et auxquelles on n'accède pas de la même façon.

Pour avoir une représentation homogène des données locales et des données reçues, Gerndt a proposé d'élargir les partitions locales pour y stocker les variables reçues (technique dite de recouvrements [Zima et al. 88]). Cela implique que le compilateur soit capable de calculer l'extension des partitions. Lorsqu'un tel calcul échoue, il faut soit étendre le tableau entièrement dans la dimension considérée soit utiliser des tampons ou des temporaires pour le stockage des variables reçues.

Dans les compilateurs Vienna Fortran et Fortran D, les techniques de recouvrement et d'utilisation de tampons sont utilisées conjointement.

4.3.2 Calcul et parcours des ensembles

Nous divisons arbitrairement ces travaux en trois types comme dans [Coelho et al. 96] : les *closed forms* , les machines d'états finis, les polyèdres.

Closed forms

Les méthodes basées sur les *closed forms* peuvent se résumer ainsi : "trouver une formule paramétrique qui sera évaluée à l'exécution pour décrire les ensembles."

Chapman et Zima proposent de compiler les références à des sections de tableaux de cette manière [Zima et Chapman 93]. De telles références peuvent être considérés comme des boucles particulières. Les sections de tableaux formant une structure stable par l'intersection et par l'application de transformations affines, celles-ci sont suffisantes pour décrire les ensembles (émission, réception, calcul) dans ce cadre. De plus, le parcours de ces structures par une boucle est trivial.

Machines d'états finis

Dans [Chaterjee et al. 93], Chaterjee traite le cas des accès à des sections de tableau distribuées cycliquement. Il prend en compte la représentation mémoire dès la compilation en réalisant une compression des ensembles d'éléments locaux. Il génère le parcours correspondant en créant un automate d'états fini permettant de parcourir les motifs d'accès à la structure.

Polyèdres et treillis

L'approche la plus générale et la mieux formalisée est sans doute celle qui traite le problème de représentation des ensembles par la définition d'un système d'inéquations affines et ramène le problème de parcours de ces ensembles au problème de parcours de polyèdres ou de leur image par une fonction affine.

- Lorsque le domaine à parcourir est obtenu par une transformation unimodulaire d'un domaine initial polyédrique, ce domaine est également un polyèdre et peut donc être parcouru en utilisant des algorithmes basés sur l'élimination par paires des contraintes redondantes (algorithme dit de Fourier-Motzkin). Delaplace et Germain réalisent une vectorisation des communications dans le cas de distributions cycliques en utilisant des transformations unimodulaires [Germain et Delaplace 95]. Du même coup, ils résolvent la transformation d'adresse (global vers local).

- Lorsque la transformation n'est pas unimodulaire mais reste non singulière, tous les points entiers ne font pas partie de l'ensemble à parcourir ; seuls ceux ayant un antécédent dans le domaine initial sont à considérer. Ramanujam présente dans [Ramanujam 92] une théorie basée sur la décomposition de la matrice de transformation T en un produit H× U où H est la forme normale de Hermite de T et U une matrice unimodulaire. Cette théorie est à la base des travaux de Coelho [Irigoin et al. 93, Coelho 96] qui traitent le cas très général de boucles parallèles dans lequel les bornes et les fonctions d'accès sont des fonctions affines. Les tableaux sont distribués par alignement sur des *templates* qui peuvent être distribués cycliquement ou par blocs..

- Le cas des transformations singulières est plus délicat et se pose dès que le domaine image est de dimension inférieure au domaine initial (cas des projections). Dans ce cas, l'image du domaine initial devient une union disjointe de polyèdres. Ce cas se présente en particulier dès que l'on cherche à caractériser $Comm(p, p')$ pour un processeur p donné. Reffay [Reffay 96] propose

d'étendre la théorie de Ramanujan pour réaliser un parcours SIMD en utilisant une forme réduite de Hermite.

4.4 Nos choix

De manière classique, nous avons choisi de générer un code en deux phases[4]. Notre schéma s'applique pour des nids de boucles "commutatifs" c'est à dire des nids de boucles parfaitement imbriqués dont toutes les instances commutent (peuvent s'exécuter dans n'importe quel ordre). Ce cadre regroupe les boucles parallèles et les réductions mais notre schéma de compilation est surtout efficace dans le cas de nids de boucles parallèles. De plus, le domaine d'itération doit être un polyèdre (éventuellement paramétré) et les fonctions d'accès aux tableaux distribués doivent être affines. Les distributions de données que nous savons traiter sont les distributions dans lesquelles les tableaux sont soit répliqués soit partitionnés en blocs dont la taille est connue à la compilation ;le placement des blocs sur les processeurs n'intervient pas. Toutes les distributions exprimables avec le langage PANDORE sont traitées par contre la plupart des distribution HPF faisant intervenir des alignements ne rentrent pas dans ce cadre.

Notre méthode s'appuie sur la définition et le parcours de polyèdres qui sont sont utilisés pour décrire l'ensemble de données concernées (données à émettre ou données sur lesquelles s'effectue un calcul). Pour chaque processeur nous générons deux énumérations différentes des points du domaine d'itération (et non des données référencées): l'une pour les échanges de données et l'autre pour les calculs.

Ces énumérations sont faites indépendamment du placement des blocs sur les processeurs: dans le code généré, nous énumérons les blocs et non les processeurs, l'exécution de ce code est gardé par un test de possession du bloc parcouru qui est effectué à l'exécution. Ce choix permet de prendre en compte de manière identique les distributions du langage HPF de type `BLOCK` (pour lesquelles un processeur possède exactement un bloc de données) et `CYCLIC` (pour lesquelles un processeur possède éventuellement plusieurs blocs). Le sur-coût induit par le calcul du possesseur d'un bloc à l'exécution est acceptable lorsque la taille des blocs est suffisamment importante et si le nombre de blocs n'est pas trop grand devant le nombre de processeurs. Ce choix serait à reconsidérer pour prendre en compte efficacement le cas du `CYCLIC(1)`[5].

Pour le code d'échange de données, nous avons choisi de réaliser seulement l'énumération des données à envoyer. Plus précisément, les processeurs réalisent un parcours de l'enveloppe convexe des données à transférer. Ce parcours prend en compte les spécificités de la représentation mémoire pour en exploiter la contiguïté. Du point de vue du récepteur, seule la liste des processeurs émetteurs est constituée.

Pour le code de calcul nous avons choisi de générer un parcours exact sans distinguer les accès car nous utilisons une représentation mémoire uniforme.

4. D'autres solutions intéressantes ont également été envisagées [André 96] mais n'ont pas été mises en œuvre faute de moyens humains.

5. Dans ce cas, la représentation mémoire que nous avons adoptée pourrait aussi être remise en cause.

Le code généré s'appuie sur des fonctions d'un exécutif efficace particulièrement optimisé pour la gestion des tableaux distribués. Par ailleurs, nous avons étudié une technique traitant les autres cas (dits irréguliers) dans le cadre du projet PREPARE.

4.5 Technique de compilation des nids de boucles commutatifs

Notre approche est présentée ici ; elle a été formalisée et mise en œuvre par Marc Le Fur durant son travail de thèse. Ce paragraphe est essentiellement une traduction de [Le Fur et al. 95].

4.5.1 Notations et définitions

Soit x un vecteur (ligne ou colonne) à n composantes. x_q $(1 \leq q \leq n)$ dénote la $q^{\text{ième}}$ composante de x.

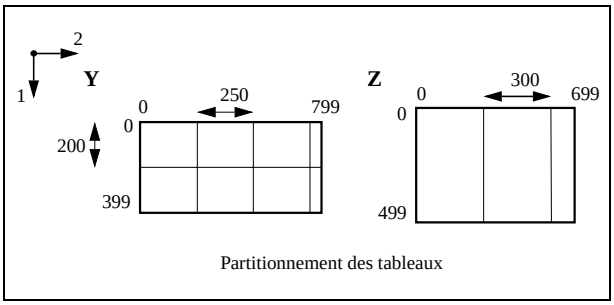
Soit u un un vecteur (ligne ou colonne) à n composantes $u_1, u_2, \ldots, u_n$, nous noterons $X[u]$ la référence $X[u_1, u_2, \ldots, u_n]$. Lorsque la fonction associée aux références est affine (par exemple $X[i+3, 2i+j+1]$), on peut la noter sous la forme matricielle suivante :

$$X[\begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \end{pmatrix}]$$

Enfin, u et v étant des vecteurs lignes ayant respectivement n et p composantes $(u\ v)$ représente la concaténation de u et v ayant $(n+p)$ composantes $(u_1 \ldots u_n\ v_1 \ldots v_p)$. Cette notation peut être étendue à un nombre de vecteurs quelconque.

Tableaux distribués : pour simplifier nous supposerons que la borne inférieure dans chaque dimension d'un tableau est 0, ce qui est le cas en C mais pas en Fortran.

X étant un tableau distribué à m dimensions, h_p^X représente le nombre d'éléments de X dans la dimension p et s_p^X la taille des blocs de X dans la dimension p. On notera $Part(X) = \{p \in 1..m \ / \ s_p^X < h_p^X\}$ l'ensemble des dimensions partitionnées (distribuées) de X et $d(X,q)$ la $q^{\text{ième}}$ dimension distribuée de X $(q \in 1..|Part(X)|)$.



Partitionnement des tableaux

Pour les tableaux Y et Z :

$h_1^Y = 400 \quad h_2^Y = 800$ | $h_1^Z = 500 \quad h_2^Z = 700$
$s_1^Y = 200 \quad s_2^Y = 250$ | $s_1^Z = 500 \quad s_2^Z = 300$
$Part(Y) = \{1, 2\}$ | $Part(Z) = \{2\}$
$d(Y,1) = 1 \quad d(Y,2) = 2$ | $d(Z,1) = 2$

On notera $Block(X, j)$ le bloc du tableau X $X[lbnd_1 \,..\, ubnd_1, \ldots, lbnd_m \,..\, ubnd_m]$ indexé par le vecteur j (vecteur ligne à n composantes) tq $\forall q \in 1 \,..\, n \quad 0 \leq j_q \leq \lceil h^X_{d(X,q)} / s^X_{d(X,q)} \rceil - 1$ qui est défini par :

- $lbnd_{d(X,q)} = j_q s^X_{d(X,q)}$ et $ubnd_{d(X,q)} = min(j_q s^X_{d(X,q)} + s^X_{d(X,q)} - 1, h^X_{d(X,q)} - 1)$ pour les dimension distribuées $d(X,q)$ de X $(q \in 1 \,..\, n)$,
- $lbnd_p = 0$ et $ubnd_p = h^X_p - 1$ pour les dimensions non distribuées p de X

Ce qui donne sur l'exemple pour les tableaux Y et Z :

$Block(Y, (0\ 1)) = Y[0 \,..\, 199, 250 \,..\, 499]$
$Block(Y, (1\ 3)) = Y[200 \,..\, 399, 750 \,..\, 799]$
$Block(Z, (0)) = Z[0 \,..\, 499, 0 \,..\, 299]$
$Block(Z, (2)) = Z[0 \,..\, 499, 600 \,..\, 699]$

et plus généralement :

$Block(Y, (j_1\ j_2)) = Y[200j_1 \,..\, 200j_1 + 199, 250j_2 \,..\, min(250j_2 + 249, 799)]$
$\forall j_1 \in 0 \,..\, 1 \quad \forall j_2 \in 0 \,..\, 3$
$Block(Z, (j_1)) = Z[0 \,..\, 499, 300j_1 \,..\, min(300j_1 + 299, 699)]$
$\forall j_1 \in 0 \,..\, 2$

Enfin, pour un vecteur à m composantes et un vecteur j à n composantes, $Belong(X, u, j)$ est l'ensemble d'inégalités qui doivent être satisfaites par le vecteur u pour que la référence $X[u]$ appartienne au bloc $Block(X, j)$ de X :

$j_q s^X_{d(X,q)} \leq u_{d(X,q)} \leq j_q s^X_{d(X,q)} + s^X_{d(X,q)} - 1 \quad \forall q \in 1 \,..\, n,$

$u_p \leq \lceil h^X_p / s^X_p \rceil - 1$ pour chaque dimension distribuée où h^X_p n'est pas un multiple de s^X_p [6].

Ce qui donne sur l'exemple, pour le vecteur $u = (i_1 + 1 \quad i_1 + 2i_2)$:
$Belong(Y, u, (j_1\ j_2)) = \{200j_1 \leq i_1 + 1 \leq 200j_1 + 199, \quad 250j_2 \leq i_1 + 2i_2 \leq 250j_2 + 249, \quad i_1 + 2i_2 \leq 799\}$
$Belong(Z, u, (j_1)) = \{300j_1 \leq i_1 + 2i_2 \leq 300j_1 + 299, \quad i_1 + 2i_2 \leq 699\}$

6. Cas du "dernier bloc" quand cela ne "tombe pas juste".

Nids de boucles : un nid de boucles parfaitement imbriqué dont le vecteur d'itération (ligne) est i, le domaine d'itération $\mathcal{D}$ et le corps $\mathcal{B}$ sera noté :

for i **in** $\mathcal{D}$ ou **for** i: $Ai^T + b \geq 0$
$\mathcal{B}$ $\mathcal{B}$

lorsque le domaine d'itération $\mathcal{D}$ est le polyèdre défini par l'ensemble de contraintes $Ai^T + b \geq 0$.

Polyèdres : considérons un nid de boucles parfaitement imbriqué dont le vecteur d'itération est i, dont le domaine est défini par $Ai^T + b \geq 0$ et dont le corps $\mathcal{B}$ est restreint à l'affectation $r_1 = f(r_2)$ où $r_1 \equiv X[C^X i^T + d^X]$ et $r_2 \equiv Y[C^Y i^T + d^Y]$.

Nous utilisons deux polyèdres $\mathcal{P}_1$ et $\mathcal{P}_2$ respectivement pour la génération du code de communication et pour la génération du code de calcul.

$\mathcal{P}_1$:

$\mathcal{P}_1(r_1, r_2)$ est l'ensemble des vecteurs de la forme
$((j_p^X)_{p\in 1\,..\,|Part(X)|}\ \ (j_q^Y)_{q\in 1\,..\,|Part(Y)|}\ \ i)$
tels que r_1 appartient au bloc $Block(X, (j_p^X))$ et r_2 au bloc $Block(Y, (j_q^Y))$. **Lorsque ces blocs sont situés sur des processeurs différents, une communication est nécessaire avant l'exécution de** $r_1 = f(r_2)$**.**

Ce polyèdre doit satisfaire l'ensemble d'inégalités suivant :

$$\forall p \in 1\,..\,|Part(X)| \quad 0 \leq j_p^X \leq \lceil h^X_{d(X,p)}/s^X_{d(X,p)} \rceil - 1$$
$$\forall q \in 1\,..\,|Part(Y)| \quad 0 \leq j_q^Y \leq \lceil h^Y_{d(Y,q)}/s^Y_{d(Y,q)} \rceil - 1$$
$$Ai^T + b \geq 0$$
$$Belong(X,\ C^X i^T + d^X,\ (j_p^X)_{p\in 1\,..\,|Part(X)|})$$
$$Belong(Y,\ C^Y i^T + d^Y,\ (j_q^Y)_{q\in 1\,..\,|Part(Y)|})$$

Pour les références $Y[i_1 + 1, i_1 + 2i_2]$ et $Z[i_2 - i_1, 3i_1 - 2]$ situées dans le nid de boucles parallèle dont le domaine d'itération est défini par $\{1 \leq i_1 \leq 230,\ \ i_1 + 1 \leq i_2 \leq 350\}$, les contraintes satisfaites par les vecteurs $(j_1^Y\ j_2^Y\ j_1^Z\ i_1\ i_2)$ de $\mathcal{P}_1(Y[i_1+1, i_1+2i_2], Z[i_2-i_1, 3i_1-2])$ sont les suivantes :
$0 \leq j_1^Y \leq 1,\ \ 0 \leq j_2^Y \leq 3$
$0 \leq j_1^Z \leq 2$
$1 \leq i_1 \leq 230,\ \ i_1 + 1 \leq i_2 \leq 350$
$200j_1^Y \leq i_1+1 \leq 200j_1^Y+199,\ \ 250j_2^Y \leq i_1+2i_2 \leq 250j_2^Y+249,\ \ i_1+2i_2 \leq 799$
$300j_1^Z \leq 3i_1 - 2 \leq 300j_1^Z + 299,\ \ 3i_1 - 2 \leq 699$

$\mathcal{P}_2$:

Pour la référence $r_1 \equiv X[C^X i^T + d]$ dans un nid de boucle parallèle, le polyèdre $\mathcal{P}_2(r_1)$ représente l'ensemble des vecteurs lignes de la forme $((j_p^X)_{p\in 1\,..\,|Part(X)|}\ \ i)$ satisfaisant l'ensemble d'inégalité suivant :

$$\forall p \in 1\,..\,|Part(X)| \quad 0 \leq j_p^X \leq \lceil h^X_{d(X,p)}/s^X_{d(X,p)} \rceil - 1$$

$Ai^T + b \geq 0$
$Belong(X,\ Ci^T + d,\ (j_p^X)_{p \in 1\,..\,|Part(X)|})$

Ce polyèdre est l'ensemble des vecteurs d'itération $i \in \mathcal{D}$ tel que r_1 appartient au bloc $Block(X, (j_p^X)$. **Il permet de représenter l'ensemble des calculs locaux au processeur possédant le bloc $Block(X, (j_p^X))$ de r_1.**

Soit avec le domaine d'itération précédent, $\mathcal{P}_2(Y[i_1 + 1, i_1 + 2i_2])$ est l'ensemble des vecteurs $(j_1\ j_2\ i_1\ i_2)$ satisfaisant :
$0 \leq j_1 \leq 1,\ \ 0 \leq j_2 \leq 3$
$1 \leq i_1 \leq 230,\ \ i_1 + 1 \leq i_2 \leq 350$
$200j_1 \leq i_1 + 1 \leq 200j_1 + 199,\ \ 250j_2 \leq i_1 + 2i_2 \leq 250j_2 + 249,\ \ i_1 + 2i_2 \leq 799$

et chaque vecteur $(j_1\ i_1\ i_2)$ de $\mathcal{P}_2(Z[i_2 - i_1, 3i_1 - 2])$ est tel que :
$0 \leq j_1 \leq 2$
$1 \leq i_1 \leq 230,\ \ i_1 + 1 \leq i_2 \leq 350$
$300j_1 \leq 3i_1 - 2 \leq 300j_1 + 299,\ \ 3i_1 - 2 \leq 699$

4.5.2 Principe de la synthèse du code

Nous présentons ici seulement le cas où la référence en partie gauche est une référence à un tableau distribué. Le cas d'une référence à un tableau répliqué est plus simple et est traité dans [Le Fur et al. 95].

Soit le nid de boucles suivant :

for i : $Ai^T + b \geq 0$
  $X[C^X i^T + d^X]\ :=\ Exp\,(ref_1, ref_2, \ldots, ref_k))$

où X est un tableau distribué et où il existe une référence $ref_i \equiv Y[C^Y i^T + d^Y]$, Y étant distribué.

Code de communication : le code de communication que nous générons est une séquence de codes de communication (1 code par référence). Chaque code est généré de la façon suivante :

1. génération du code énumérant[7] les vecteurs $(j^X\ \ j^Y\ \ i)$ du polyèdre $\mathcal{P}_1(X[C^X i^T + d^X], Y[C^Y i^T + d^Y])$

   for j^X **in** $\mathcal{D}_1$
     for j^Y **in** $\mathcal{D}_2(j^X)$
       for i **in** $\mathcal{D}_3(j^X, j^Y)$

   où $\mathcal{D}_1$, $\mathcal{D}_2(j^X)$ et $\mathcal{D}_3(j^X, j^Y)$ dénotent les domaines associés aux vecteurs d'itération j^X, j^Y and i respectivement.

7. Cette énumération est réalisée par l'algorithme présenté dans [Le Fur 96].

2. l'insertion des masques et des instructions de communication nous permet de produire le code SPMD suivant pour l'émission de messages :

```
for j^X in D_1
    if myself ≠ owner of Block(X, j^X) then
        for j^Y in D_2(j^X)
            if myself = owner of Block(Y, j^Y) then
                for i in D_3(j^X, j^Y)
                    pack Y[C^Y i^T + d^Y] in buffer
                send buffer to the owner of Block(X, j^X)
```

3. et le code suivant pour les réceptions

```
for j^X in D_1
    if myself = owner of Block(X, j^X) then
        for j^Y in D_2(j^X)
            if myself ≠ owner of Block(Y, j^Y) then
                receive buffer from the owner of Block(Y, j^Y)
                for i in D_3(j^X, j^Y)
                    unpack Y[C^Y i^T + d^Y] from buffer
```

dans lequel la fonction *unpack* extrait les éléments du tampon de communication pour les copier en mémoire.

Code de calcul : le code de calcul ne dépend que de la référence en partie gauche $X[C^X i^T + d^X]$ (règle des calculs locaux) :

1. génération du code énumérant les points du polyèdre $\mathcal{P}_2(X[C^X i^T + d^X])$:

   for j^X **in** $\mathcal{D}_4$
   for i **in** $\mathcal{D}_5(j^X)$

   Dans cette boucle, $\mathcal{D}_4$ et $\mathcal{D}_5(j^X)$ représentent les domaines d'itération associés respectivement à j^X et i ;

2. l'insertion des masques nous permet de produire le code SPMD suivant :

   for j^X **in** $\mathcal{D}_4$
   if *myself* = owner of $Block(X, j^X)$ **then**
   for i **in** $\mathcal{D}_5(j^X)$
   $X[C^X i^T + d^X] \ := \ Exp\,(\mathcal{D}ist \ \cup \ \mathcal{R}epl)$

4.5.3 Cas paramétré

Cette technique s'étend au cas "paramétré" dans lequel le nid de boucles parallèle dépend de variables (non affectées dans le nid de boucle) ou d'indices de boucles englobantes. On trouvera plus de détails dans [Le Fur et al. 95].

4.5.4 Méthode d'énumération

La synthèse des codes d'énumération repose sur des algorithmes de parcours de points entiers de polyèdres. Deux algorithmes ont été développés par Marc Le Fur pendant sa thèse. Ils reprennent la méthode des projections successives définie par Ancourt et Irigoin [Ancourt et Irigoin 91] qui est basée sur l'élimination par paires dite de Fourier-Motzkin. L'apport principal des algorithmes de Marc Le Fur est la politique d'élimination des contraintes redondantes qui permet de générer des parcours efficaces avec un temps de synthèse réduit. Ces algorithmes sont décrits en détail dans [Le Fur 95a] pages 77 à 102 et dans [Le Fur 96].

Feautrier a proposé l'utilisation d'un simplexe paramétré pour le calcul des bornes inférieures et supérieures dans chaque dimension d'un polyèdre qui est a la base de la réalisation de l'outil PIP (Parametric Integer Programming) [Feautrier 89]. Une autre technique basée sur l'algorithme de Chernikova et qui repose sur la représentation d'un polyèdre sous forme de sommets et de rayons a donné lieu a la mise en œuvre d'une bibliothèque de calculs sur les polyèdres [Le Verge et al. 94].

L'intérêt des algorithmes de parcours de points entiers de polyèdres dépassant le cadre de la compilation de HPF (et de C-Pandore), un logiciel indépendant du compilateur a été développé. Ce logiciel (ENUM) est disponible ; il prend en entrée un système de contraintes et rend une boucle (C ou Fortran) qui réalise le parcours du polyèdre.

4.6 Représentation mémoire

De même que pour la génération de code, nous avons recherché des solutions assez générales pour améliorer la gestion des données distribuées tant du point de vue de l'organisation mémoire que du point de vue des communications. Ce travail a fait l'objet de la thèse de Yves Mahéo.

Nous avons recherché à optimiser la gestion des tableaux distribués et en particulier à fournir un mécanisme d'accès uniforme aux éléments de ces tableaux. En effet, dans le cas de la compilation des nids de boucles commutatifs, il n'est généralement pas possible de séparer les accès aux éléments locaux des accès à des éléments reçus.

En plus de la gestion mémoire, nous avons défini des primitives de communication ([Mahéo 95] pages 67 à 77) adaptées au schéma de compilation optimisé qui ne sont pas détaillées ici.

4.6.1 Principe

La gestion des tableaux distribués réalisée dans le projet PANDORE est basée sur la "pagination logicielle". Les distributions permises par le langage C-Pandore sont seulement des distribution directes, c'est pourquoi nous ne présentons ici que ce cas. Cependant, la technique est applicable pour des distributions faisant intervenir des *templates*, comme c'est le cas du langage HPF.

Notre technique de pagination repose sur les mêmes bases que la pagination classique : l'espace mémoire logique est divisé en pages (ensemble d'éléments contigus) dont la taille est fixe. Les adresses logiques sont considérées comme la concaténation d'un numéro de page et d'une adresse dans la page (offset). Le numéro de page logique est converti en numéro de page physique avant l'accès à la mémoire. Dans la pagination classique cette transformation est réalisée par un composant matériel (MMU) ; lorsque la pagination est logicielle, cette transformation est faite par une fonction logicielle et est donc plus coûteuse.

On associe souvent à la pagination un mécanisme de défaut de page qui permet de recopier automatiquement une page non présente dans la mémoire soit à partir d'un disque (dans le cas de la mémoire virtuelle classique), soit depuis la mémoire locale d'un autre processeur (dans le cas de la mémoire virtuellement partagée sur un multiprocesseur [Lahjomri et Priol 92]). Ce mécanisme n'est pas présent dans notre exécutif car le schéma de compilation génère toutes les communications nécessaires avant un accès à une donnée.

4.6.2 Pagination des tableaux distribués

Nous associons à chaque tableau distribué une fonction de linéarisation $\mathcal{L}$ et une taille de page `S`. Autrement dit, nous ne paginons pas la mémoire mais chaque représentation de tableau de manière indépendante.

Nous nous sommes fixés comme contrainte d'avoir une fonction de transformation d'adresse aussi peu coûteuse que possible et de pouvoir retrouver le propriétaire d'un élément avec un très faible coût.

Pour satisfaire ces contraintes nous avons choisi de définir $\mathcal{L}$ et S de la façon suivante :

Les tableaux sont "virtuellement" étendus dans toutes leurs dimensions à la puissance de 2 immédiatement supérieure à leur taille dans la dimension considérée.

Pour un tableau dont au moins la dimension K n'est pas distribuée, la taille du tableau étendu dans cette dimension sera la taille S des pages de la représentation du tableau. La fonction de linéarisation opère sur le tableau virtuellement étendu et conserve la contiguïté dans la dimension K. Ceci permet qu'une page soit possédée par au plus un processeur : le numéro de page nous permet donc de trouver le possesseur d'un élément. De plus le calcul du numéro de page et de l'offset dans une page peut être réalisé sans aucune opération div ou mod. Les seules opérations qui restent sont des multiplications (par des puissances de 2, donc de simples décalages) si le tableau possède au moins 3 dimensions. Pour un tableau à 2 dimensions, les indices globaux correspondent exactement au numéro de page et à l'offset dans la page.

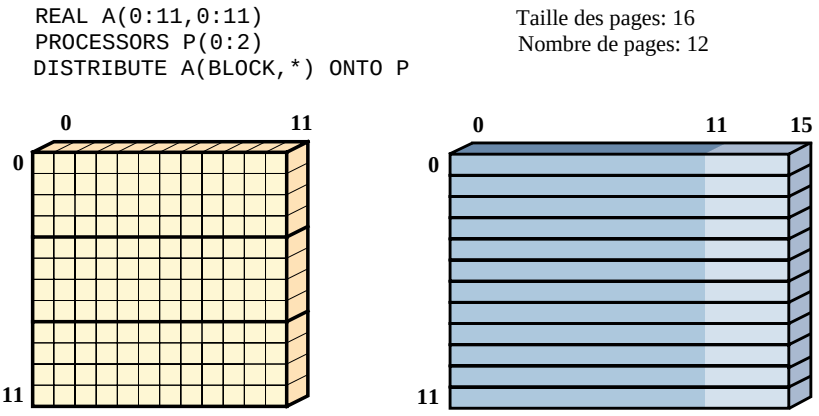


FIG. 4.1 – *Cas d'un tableau 2D avec une seule dimension distribuée*

Pour un tableau dont toutes les dimensions seraient distribuées, nous choisissons la dimension K comme étant celle qui est la plus étendue dans les blocs. S et $\mathcal{L}$ sont définies de la manière suivante : la taille des pages est la puissance de 2 immédiatement inférieure à la taille d'un bloc (dans cette dimension). Comme précédemment, la fonction de linéarisation opère sur le tableau virtuellement étendu et conserve la contiguïté dans la dimension K. Dans ce cas, une une page peut être possédée par (au plus) deux processeurs, ce qui impose un test sur l'offset pour connaître le possesseur d'un élément. Le calcul du numéro de page et de l'offset dans une page nécessite des opérations div et mod mais celles-ci portent sur des puissances de 2, ce qui peut être réalisé très efficacement par des décalages.

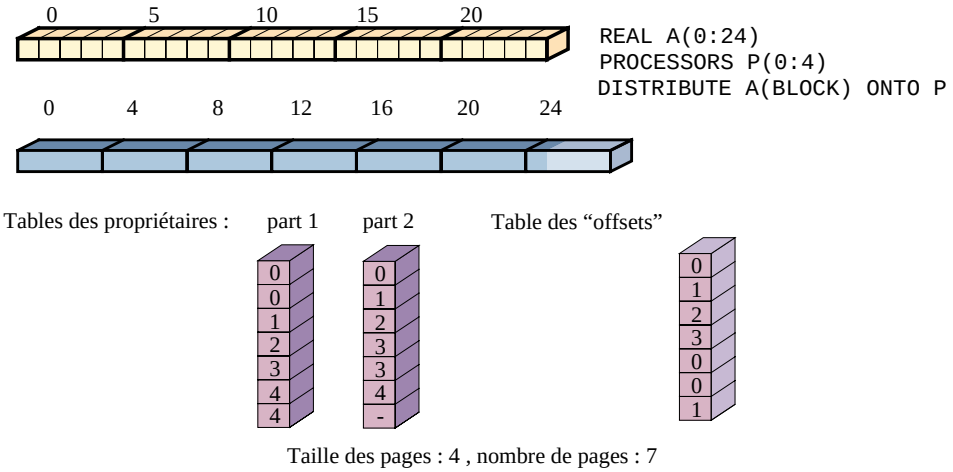


FIG. 4.2 – *Cas d'un tableau 1D (toutes "les dimensions" sont distribuées)*

4.6.3 Représentation mémoire des tableaux paginés

Chaque processeur possède :

- une table des propriétaires de pages qui indique pour chaque page, quel est le processeur où elle se trouve. Cette table peut être calculée statiquement.

- une table des pages qui contient l'adresse physique où se trouve la page si celle-ci est locale. Cette table contient également l'adresse physique des copies locales de pages distantes.

- la partition locale paginée qui est allouée de manière contiguë.

- des copies de pages distantes qui contiennent des données utilisées. Ces pages sont allouées dynamiquement et leur durée de vie est limitée à un nid de boucles.

Pour limiter la mémoire occupée, la dernière dimension utilisée pour la linéarisation n'est pas étendue (cela reviendrait à agrandir inutilement la table des pages) ; de même, seule la partie réellement utile des pages est allouée : pour la partition locale on ne paie donc pas le sur-coût de l'extension de tableau utilisée pour la pagination. Pour les copies de valeurs distantes, le schéma de compilation mis en œuvre réalise une vectorisation des messages et transmet des segments dont la taille est inférieure ou égale à la taille d'une page. L'allocation mémoire ne concerne que ces segments et non des pages entières.

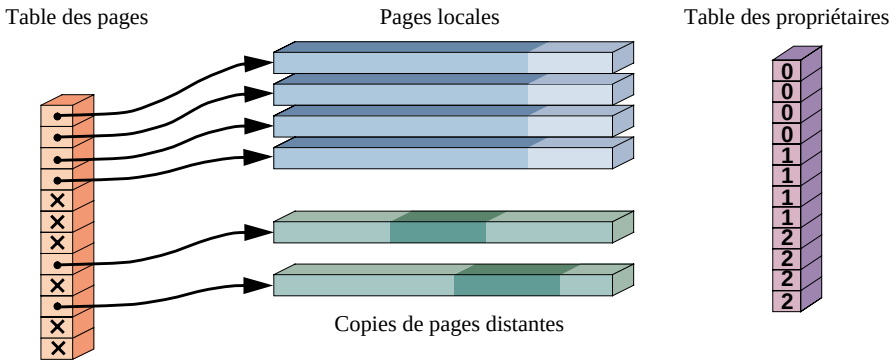


FIG. 4.3 – *Cas d'un tableau 2D avec une seule dimension distribuée*

4.6.4 Prise en compte de la représentation mémoire à la compilation

Afin d'optimiser les communications ainsi que la synthèse et le parcours des ensembles de description de données à envoyer, nous avons choisi d'envoyer (pour chaque référence et pour chaque bloc) l'enveloppe convexe des données à échanger. Ainsi nous ne construisons pas point par point l'ensemble exact d'émission comme cela est décrit en 4.5.

Nous décrivons directement l'enveloppe convexe des données à échanger ([Le Fur 95b] pages 54 à 62, [Le Fur et Mahéo 95]) sous forme d'une liste de segments contigus en mémoire ce qui permet à la fois de réduire la complexité de l'algorithme de génération de parcours et de diminuer sensiblement le temps de parcours de l'ensemble. De plus cette description génère des communications vectorisées. Dans le cas où la fonction d'accès est unimodulaire, cet ensemble est exactement l'ensemble des données à envoyer; sinon il est effectivement plus grand. L'efficacité de cette technique repose sur le gain obtenu d'une part sur le coût de l'énumération et d'autre par sur le transfert de blocs de mémoires contigus (du point de vue de l'émetteur comme du récepteur). Cette efficacité dépend du ratio nombre d'envois / nombre d'envois utiles.

La description des ensembles de communication étant effectué référence par référence, l'exécutif se charge d'éliminer les transferts redondants ([Le Fur et Mahéo 95]). Pour cela, la liste des segments à envoyer est mémorisée. Pour chaque page nous réalisons le transfert d'un seul segment qui est l'enveloppe convexe des segments à envoyer (le segment englobant).

4.7 Bilan

L'originalité de notre approche repose sur la prise en compte du placement des blocs à l'exécution et non à la compilation comme cela est habituellement fait. Ceci nous permet de faire entrer dans un même cadre les distribution `BLOCK` et

`CYCLIC(k)` de HPF ainsi que les placements `regular` et `cyclic` de PANDORE qui sont très différents. Nous avons pu vérifier que cette approche était viable si le nombre de blocs par processeur n'est pas trop grand. Cette approche n'est pas viable dans le cas de distribution `CYCLIC(1)` de HPF [8].

Notre gestion des données distribuées permet d'avoir une représentation uniforme des données et de leurs copies qui est compatible avec la vectorisation des messages mise en œuvre par notre compilateur. En dehors de la duplication ou de l'utilisation d'une mémoire virtuellement partagée , c'est à ma connaissance le seul mécanisme de gestion de données distribuées offrant ces possibilités.

Nous avons pu montrer que les optimisations réalisées tant sur le plan de la compilation que du support d'exécution étaient viables. Sur des exemples classiques, les accélérations mesurées étaient du même ordre de ce que l'on obtient par un codage manuel non optimisé. En particulier, sur le noyau de calcul de l'algorithme de Jacobi, l'efficacité (ici au sens accélération/nombre de processeurs) restaient supérieure à 80% jusqu'à 32 noeuds sur un iPSC/2 pour une matrice 512×512. De même pour l'algorithme du "Red/Black" l'efficacité mesurée était supérieure à 70% alors que les optimisations mises en œuvre dans plusieurs compilateurs commerciaux échouent dans ce cas. Ces résultats sont présentés dans la thèse de Yves Mahéo ([Mahéo 95], pages 101–110). Ces mesures pourraient être remises à jour pour des architectures plus actuelles mais seraient sans doute comparables sur des plate-formes telles que des station Suns inter-connectés par Myrinet que nous utilisons actuellement.

Quelques perspectives à court terme ont été envisagées et partiellement développées :

- Comme je l'ai déjà dit plus haut, des schémas de compilation prenant en compte l'alignement des tableaux et le placement des blocs ont été définis mais n'ont pas été mis en œuvre dans le prototype.

- L'intérêt de la bibliothèque de gestion de tableaux distribués dépassant le cadre de son utilisation conjointe avec le compilateur PANDORE, une bibliothèque de gestion de tableaux distribués a été re-développée sur les mêmes bases par Françoise André et Yves Mahéo [André et Mahéo 96].

- Un schéma d'exécution basé sur l'utilisation de tâches légères (*threads*) a été défini et expérimenté par Françoise André. Ce schéma qui permet d'ordonnancer les calculs en fonction de l'ordre d'arrivée des données distantes a été décrit dans [André et Pazat 96] et [André 96].

8. Mais cette distribution est-elle vraiment utile?

Chapitre 5

Bilan et perspectives

5.1 Bilan

Les travaux que nous avons réalisés montrent l'intérêt et la faisabilité des techniques de transformation de programmes séquentiels pour masquer le parallélisme d'exécution et la répartition dans le cadre du calcul hautes performances sur machines parallèles à mémoire distribuée.

5.1.1 Résultats

Le but de l'étude qui a été présentée ici était de définir et d'évaluer des techniques de transformation de code et les schémas d'exécution associés, le prototype servant de support de recherche et d'expérimentation. Quatre thèses ont été soutenues sur ce projet et de nombreuses publications ont permis de faire connaître les techniques développées.

- Le premier prototype développé avait permis de montrer la faisabilité la l'approche "compilation par distribution de données" et la nécessité de concevoir et d'implémenter des techniques de compilation permettant d'optimiser le code produit, en particulier pour regrouper les communications et diminuer le nombre de synchronisations. Ceci a motivé d'abord une réécriture complète du prototype pour le rendre plus facilement évolutif, puis les travaux de thèse sur la compilation et le support d'exécution.

- L'algorithme d'énumération des points entiers d'un polyèdre, qui est indépendant de la structure interne du compilateur, a été rendu disponible et est d'ores et déjà utilisé par d'autres équipes de recherche. Cet algorithme n'a pas été présenté ici car je considère que c'est la contribution personnelle de Marc Le Fur. Par contre, la technique de compilation et en particulier le lien entre la génération de code et la gestion de la mémoire sont le fruit du travail d'équipe que j'ai animé pendant plusieurs années.

- La technique de compilation pour les nids de boucles commutatifs ainsi que les principes de pagination qui ont présidé à la mise en œuvre du support d'exécution ont été expérimentés et publiés. Des extensions en ont été proposées bien que non mises en œuvre dans le compilateur PANDORE pour des raisons de temps.

- Un travail important d'expérimentation a été mené lors du développement des techniques de compilation optimisées et de l'exécutif afin d'évaluer précisément où faire porter les efforts. Ce travail a donné lieu au développement d'un prototype d'outil de mesures de performances utilisant la technique de profiling.

- Enfin, l'effort de formalisation de la description du schéma de compilation nous a permis de montrer que les techniques de compilation par distribution de données n'étaient pas totalement liées à l'utilisation de bibliothèques d'échanges de messages mais pouvaient s'adapter à d'autres supports d'exécution tels que la mémoire virtuellement partagée.

5.1.2 Diffusion et transfert

Le compilateur et l'exécutif réalisés n'ont pas été distribués d'une part par manque de moyens humains mais aussi parce que je pense que la contribution essentielle est dans les techniques qui sont applicables à d'autres langages et en particulier HPF ; le compilateur et son exécutif sont plutôt destinés à des expérimentations "en interne". Si le langage d'entrée du compilateur avait été Fortran, il aurait été intéressant de rendre notre outil accessible à d'autres, comme l'a fait Thomas Brandes avec le compilateur Adaptor [Brandes et Zimmermann 94].

Le transfert des techniques développées vers le milieu industriel a été partiellement réalisé dans le cadre du projet Esprit PREPARE. Notre participation a ce projet s'est faite dans le cadre de la génération systématique de code réparti (transformation instruction par instruction) et dans le cadre du traitement optimisé des boucles irrégulières. Nous avons ainsi pu transférer une partie de notre savoir-faire dans un prototype de niveau industriel. D'autre part, cette expertise est mise à profit pour l'évaluation du compilateur HPF de NEC sur la machine à mémoire distribuée Cenju 3 et son éventuelle adaptation à une mémoire virtuellement partagée dans le cadre d'une collaboration entre l'INRIA et NEC.

5.1.3 Prototypage

Pour évaluer finement une méthode de transformation de code, il me paraît essentiel de la mettre en œuvre. Certaines transformations, séduisantes sur le papier, peuvent se révéler ne produire que des codes peu efficaces et inversement des transformations de complexité rédhibitoires sont en pratique utilisables comme nous avons pu le vérifier.

En effet, la technique "de base" (transformation instruction par instruction) est très peu efficace et les optimisations simples voire naïves que nous avions mises en

œuvre dans notre premier prototype ne suffisaient pas pour obtenir des performances raisonnables. De même, le schéma de compilation optimisé ne permet pas d'obtenir de meilleures performances (en temps et en espace) s'il n'est pas associé à une gestion de la mémoire elle aussi optimisée (certains compilateurs proposent même des schémas "optimisés" qui ont pour effet d'allouer les tableaux répartis en totalité sur chaque nœud!).

Cela veut-il dire qu'il soit nécessaire de réaliser un compilateur complet pour évaluer un schéma de compilation?

Je crois qu'il est effectivement nécessaire d'intégrer ses idées dans un compilateur (que l'on réalise, ou que d'autres on réalisé), mais se pose alors le choix du langage que l'on veut transformer. Si une argumentation de nature théorique (preuve d'un schéma) peut se contenter de se référer à un langage "jouet" comme le langage $\mathcal{L}$ [Bougé et Cachera 95], ceci ne peut convaincre de l'applicabilité de techniques de compilation comme celles que nous avons proposées dans des cas réalistes. A l'autre extrême, réaliser un compilateur de niveau industriel pour un langage existant comme Fortran 90 est irréaliste dans le cadre d'un projet de recherche. Il faut donc rechercher un compromis entre ces deux extrêmes; c'est ce que nous avons fait.

5.2 Perspectives

La direction de recherche que je me suis fixée consiste à étudier et mettre en œuvre des techniques de transformation de programmes permettant de masquer le parallélisme d'exécution et la répartition. L'étude de ces transformations se doit d'inclure le choix, voire la définition, de modèles de programmation ainsi que le choix ou la mise en œuvre de supports d'exécution.

Pour concentrer mes efforts, j'ai maintenant choisi comme cadre privilégié de mes travaux l'utilisation des réseaux de stations de travail hétérogènes (aussi bien du point de vue du réseau que des machines); c'est je pense le type d'architecture qui sera le plus répandu dans les prochaines années et c'est donc pour ce type d'architecture qu'un grand nombre d'applications devront être développées.

De plus, il me paraît incontournable d'étudier des transformations de programmes qui visent les langages à objets car ceux-ci permettent de concevoir de nouvelles applications de taille conséquente avec des critères de qualité qui ne peuvent être atteints par les langages procéduraux comme Pascal, C ou Fortran. Je pense que ce seront les langages à objets qui seront les plus utilisés dans les années à venir; il me paraît donc crucial de développer du savoir faire et des technologies autour de ces langages.

5.2.1 Grappes de stations

Les calculateurs parallèles à mémoire distribuée étaient la cible initiale des travaux que j'ai présenté ici mais l'usage de ces "machines" [1] reste confiné au calcul hautes performances dans de grands laboratoires et pour des raisons de coût il semble que peu d'autres utilisations soient envisagées. Par contre les NOWs (Networks of Workstations) et COWS (Clusters of workstations) qui sont réalisés par l'interconnexion de stations de travail par des réseaux haut débit se développent et sont d'un coût bien moindre (aussi bien à l'achat que pour leur maintenance). L'utilisation croissante de ces architectures que nous appellerons "grappes" de stations (ou simplement "grappes") est non seulement dû à leur relativement faible coût et à leur facilité d'installation mais aussi aux possibilités d'usage non spécialisé de ces architectures comme stations de travail ordinaires.

Les grappes sont des cas particuliers d'architectures à mémoire distribuée et donc les travaux que j'ai présentés peuvent s'appliquer à ce cadre (nous avons d'ailleurs porté l'exécutif PANDORE sur un réseau de Suns inter-connectés par un réseau Myrinet) mais l'utilisation croissante des grappes est en train de changer le type d'utilisation et d'utilisateurs de ces architectures à mémoire distribuée.

- Du fait du relativement faible coût de ces architectures, on ne peut espérer que celles-ci seront programmées par des "experts" comme c'était le cas pour les premières architectures parallèles à mémoire distribuée dont l'utilisation quasi-confidentielle était surtout le fait de grands laboratoires de recherche en informatique ou de laboratoires étroitement associés à des centres de recherche en informatique. Le besoin d'outils d'aide à la programmation est donc très important dans ce cadre.

- Le calcul scientifique "modérément performant", c'est-à-dire n'utilisant pas de ressources extrêmement coûteuses, va se développer en particulier dans le domaine de la simulation (dans l'industrie automobile on se préoccupe maintenant même d'étudier la climatisation de l'habitacle d'un véhicule par simulation numérique) ou d'applications dites de réalité virtuelle (simulation de milieux urbains, d'impacts des tracés d'autoroute sur le paysage par exemple). De nombreuses applications scientifiques devenant économiquement rentables face à la réalisation de maquettes ou de prototypes matériels, le calcul parallèle et distribué devra s'intégrer au reste de l'informatique de l'entreprise.

- D'autres applications mixant les caractères (géographiquement) réparti et performance pourront également se développer. Je pense au calcul réparti sur les réseaux hétérogènes à grande échelle (*metacomputing*) mais aussi aux outils d'indexation automatique du Web et plus généralement de traitement et de gestion de données (*data mining*) ainsi qu'aux applications de travail coopératif. Les outils développés pour les "machines" à mémoire distribuée n'étant pas

1. Le terme "machine" est utilisé ici pour signifier que celles-ci se trouvent dans une seule "armoire" et sont composées d'éléments (réseau et nœuds de calcul) conçus spécifiquement pour cet usage).

adaptés à ces nouveaux types d'applications, il est nécessaire d'en concevoir de nouveaux.

5.2.2 Difficultés

Bien que ces machines soient relativement faciles à installer, les difficultés d'utilisation des grappes ne doivent pas être sous-estimées. On retrouve en effet toutes les difficultés d'utilisation des calculateurs parallèles à mémoire distribuée sur la plupart des grappes auxquelles s'ajoutent de nouvelles contraintes qui doivent être prises en compte. Parmi ces difficultés je m'intéresserai en particulier à traiter :

de l'hétérogénéité.

Les stations inter-connectées sont différentes aussi bien au niveau matériel car leur architecture peut être très spécifique (éventuellement parallèle comme sur les multiprocesseurs symétriques ce qui introduit un niveau de hiérarchie mémoire) ; les processeurs et les systèmes d'exploitation tournant sur chaque nœud peuvent être différents (plusieurs UNIX, Windows NT et quelques rares systèmes propriétaires devront cohabiter) rendant difficile, voire impossible l'exécution directe de code SPMD.

Les réseaux permettant l'interconnexion de différentes plate-formes (Ethernet, Myrinet, ATM, SCI,etc.) et les protocoles de communication (TCP, protocoles propriétaires sur Myrinet,etc.) doivent également inter-opérer pour permettre l'exécution des programmes répartis sur ces architectures.

Cette hétérogénéité doit pouvoir être masquée au programmeur lorsqu'il le souhaite et doit donc pouvoir être prise en compte soit au niveau d'un exécutif, soit au niveau des transformations de programmes.

de la reconfiguration.

La reconfiguration d'un réseau de stations de travail est fréquente. En effet l'ajout ou le changement d'une station sur un réseau n'est ni exceptionnel ni difficile à réaliser. De plus la reconfiguration peut être dynamique : il faut envisager la probabilité de panne d'une station ou d'une partie du réseau ainsi que la réservation et la libération de stations ou d'un sous-réseau qui interviennent lorsqu'une application ou un groupe d'applications ont des besoins spécifiques.

Ces reconfigurations ne doivent pas induire de modifications du programme de l'utilisateur car elles sont trop fréquentes. Elles doivent donc pouvoir être prises en compte au niveau des transformations de programme ou au niveau de l'exécutif lorsque celles-ci sont dynamiques.

de la diversité des applications.

L'utilisation des grappes étant a priori plus large que celles des "machines" à mémoire distribuée, il est difficile de définir un modèle de programmation qui soit satisfaisant pour tous les utilisateurs. D'un autre coté, il ne paraît pas réaliste de

définir un langage de programmation spécifique à chaque classe d'application. Il faut donc trouver le moyen d'adapter le modèle de programmation sans remettre en cause le langage de programmation ni l'ensemble des transformations de programme.

du partage.

Une grappe doit pouvoir être partagé entre diverses applications dont certaines sont interactives pour que leur coût puisse être amorti. Ce problème sort du cadre que je me suis fixé car il s'agit ici surtout de développer des outils de gestion de grappes en tant que machine. Mais le lien avec les problèmes de placement de tâches que j'ai étudiés me sensibilisent à ce problème que je vois comme une activité annexe de mon projet de recherche.

5.2.3 Vers des solutions

Pour que l'utilisation des grappes soit viable, il nous faut concevoir et réaliser des outils masquant ou à défaut séparant les problèmes d'implantation que j'ai cités des problèmes de conception. En un mot, il faut remonter et adapter le niveau d'abstraction des grappes qui est présenté au programmeur.

Ces outils doivent permettre d'exprimer et de mettre en œuvre des modèles de programmation adaptés aux applications développées pour les grappes. Les logiciels réalisés grâce à ces outils doivent "supporter" et si possible exploiter l'hétérogénéité des supports d'exécution ; ils doivent "survivre" aux reconfigurations du réseau, et même en tirer parti lors de leur exécution.

Modèle de programmation

Quelle que soit l'application développée, je cherche à bâtir des solutions qui permettent au programmeur d'exprimer le parallélisme et la distribution correspondants à la logique de son application.

- En ce qui concerne le parallélisme, l'utilisation d'un langage séquentiel me paraît insuffisante car elle peut conduire à une sur-spécification inutile, voire confuse (la séquentialisation d'actions indépendantes dans la logique du programme). Il serait donc souhaitable que le programmeur puisse exprimer le parallélisme lié à son application en laissant à un outil automatique le soin d'adapter le parallélisme à la machine d'exécution.

- De même, la spécification pour la distribution et le placement de certaines entités du programme (tâches ou données) peut être soit liée à la logique du programme, soit à des impératifs liés à une architecture donnée. Il serait idéal que le programmeur spécifie le placement et la distribution qui font partie de son problème et qu'il puisse encore laisser à un outil automatique le soin de calculer les distributions manquantes.

Si l'on ne veut pas fixer plus précisément le modèle de programmation tout en cherchant à offrir des services adaptés aux applications, il me semble judicieux de

séparer les spécificités d'un modèle de programmation du langage lui-même. Ceci peut être réalisé de manière élégante en utilisant des *frameworks* qui s'intègrent particulièrement bien à la programmation objet [Jézéquel 97]. Un *framework* est une collection de classes qui sont liées entre elles par de multiples schémas de collaboration (*patterns*) statiques et dynamiques. Il fournit un ensemble intégré de fonctionnalités spécifiques à un domaine plus ou moins spécialisé. Dans notre cas le framework est simplement le modèle de programmation parallèle qui servira de base aux techniques de transformation de programmes que je compte développer.

Supports d'exécution

Masquer l'hétérogénéité est un élément essentiel pour faciliter l'utilisation des grappes. Des solutions partielles existent pour la communication, (par exemple PVM, MPI, l'appel de méthode à distance de JAVA, ou CORBA) bien que leur mise en œuvre ne soit pas toujours efficace et fiable. Du point de vue de l'exécution sur un nœud, l'utilisation de machines virtuelles mise en œuvre par des interprètes (comme la machine virtuelle JAVA) est une solution dont l'efficacité encore insuffisante peut être améliorée par l'utilisation de techniques de compilation "à la volée" (*Just In Time compilers* [Muller et al. 96]).

Des support d'exécutions optimisés pour certains types d'applications comme les applications de calcul irrégulières (Athapascan [Christaller 96] et PM^2 [Namyst et Méhaut 95]) ou pour la coopération entre applications de calcul scientifique réparties comme l'exécutif d'Opus [Haines 95]), ont d'ores et déjà été développées.

J'ai l'intention d'utiliser prioritairement des supports d'exécution les plus généraux possibles en évitant de m'écarter inutilement des standards. Néanmoins lorsque des supports d'exécution non standards existent et fournissent des mécanismes utiles ils seront également considérés comme des cibles intéressantes des transformations de programmes que je développerai.

Transformation de programmes

Définir de nouvelles transformations de programmes tout en en maîtrisant finement leur sémantique est l'élément essentiel pour passer du modèle de programmation présenté au concepteur au modèle d'exécution fourni par le support d'exécution.

Ces transformations peuvent être automatiques et mises en œuvre par des techniques de compilation ou de spécialisation, mais peuvent aussi être manuelles ou semi-automatiques, par exemple par écriture et utilisation de bibliothèques masquant la parallélisme et la distribution (comme dans le cas de EPEE [Jézéquel et al. 97]). Il y a sans doute matière dans la recherche d'un équilibre entre les transformations de programmes automatiques et l'écriture pénible et surtout peu sûre de plusieurs versions d'une même bibliothèque pour différentes machines.

C'est parce que la préservation d'une sémantique était assurée[2] par la règle

2. L'équivalence de résultat qui est suffisante dans le cadre du calcul scientifique.

des calculs locaux que HPF a pu se développer. Dans EPEE qui utilise la technologie objet, le même type de transformation a également pu être implanté et utilisé avec succès pour les mêmes raisons, mais cette fois-ci de manière manuelle (quoique réutilisable). Dans tous les cas un même problème se pose : que me garantit exactement la transformation du point de vue sémantique? L'équivalence de résultat qui était bien agréable dans le cadre du calcul scientifique s'avère insuffisante pour de nombreuses applications.

Mes travaux ont pour but de trouver et de mettre en œuvre d'autres types de transformations assurant des équivalences éventuellement plus fines que l'équivalence de résultat. Ces transformations s'appuient sur l'utilisation de *frameworks* dans des langages à objets.

5.2.4 Quelques étapes

Je me propose d'organiser la suite de mes travaux selon les étapes suivantes.

Étude d'une transformation parallèle vers réparti

Pour effectuer une première généralisation de l'approche de transformation de programmes, je propose de réaliser des transformations de programmes explicitement parallèles en des programmes parallèles et répartis [3]. Par rapport à l'approche du projet PANDORE, nous prenons en compte dans un même cadre la génération de code réparti par distribution de contrôle et par distribution de données. Il faut noter que nous aurions pu faire ce travail en nous limitant à Fortran et en définissant la distribution de boucles parallèles de la même manière que la distribution de tableaux mais ceci a déjà été réalisé dans le compilateur HPF de NEC, pour des espaces d'itération "rectangulaires".

Pour ce projet, nous avons choisi d'utiliser le langage à objets JAVA sans en modifier la syntaxe mais en "encadrant" la programmation par un *framework* qui permet de restreindre le parallélisme exprimable.

L'intérêt de ce langage est d'abord d'assez bien masquer l'hétérogénéité grâce à la machine virtuelle d'exécution (JVM) et au protocole d'appel de méthode à distance (RMI) ; de plus ce langage est d'apprentissage assez facile et est assez répandu même s'il souffre encore de quelques défauts de jeunesse et de limitations désagréables (pas de généricité).

L'idée est de pouvoir réaliser un premier prototype dans le temps d'une thèse (travail de Pascale Launay) qui a pour but de démontrer la faisabilité de l'approche comme cela à été fait pour PANDORE. De plus, nous espérons que ce prototype permettra de tester la pertinence de l'approche sur des exemples réalistes.

Dans ce cadre nous avons défini des contraintes à imposer au parallélisme afin d'être capable de générer du code réparti "maîtrisable" [Launay et Pazat 97]. En effet, la transformation de code séquentiel en code réparti produisait un code SPMD ayant de "bonnes" propriétés sémantiques (propriété du losange [Bareau 95]) que l'on risque de perdre lorsque l'on transforme du code parallèle en code réparti. Il faut

3. Au sens défini en 1.3.3

également définir quelles sont les transformations de programme légales et donc définir des équivalences entre programmes parallèles répartis et non répartis. Il faudra donc définir avec soin la sémantique que l'on désire préserver (nous commencerons par la sémantique de résultat).

Calcul de distributions

Dans ce même cadre, il est important de pouvoir calculer la distribution d'un certain nombre d'objets du programme (ici au sens des langages à objets) d'une part pour permettre au programmeur de n'exprimer que la répartition qui l'intéresse et d'autre part pour permettre au logiciel de résister et même de tirer parti des reconfigurations de la machine (lorsque celles-ci ne sont pas dynamiques). Ceci peut se faire de deux façons différentes :

- par simple placement (un outil automatique devant calculer le meilleur placement de cet objet en fonction de ces relations avec les autres objets du programme),
- par "fragmentation" ou plus précisément découpage des données qui le composent lorsque celles-ci possèdent une structure (un outil automatique devant alors calculer à la fois la fragmentation et le placement des fragments).

Si le premier cas semble relativement facile à traiter (certes de façon non optimale), car il se rapproche des problèmes de placement de tâches, le second cas par contre mérite une étude approfondie. En effet les seules structures de données pour lesquelles la fragmentation à été étudiée dans le cadre de HPF sont les tableaux. L'aspect très dynamique introduit par l'utilisation de langages à objets limitera certainement les calculs de distribution réalisables et je ne fais ici que poser le problème.

5.2.5 A plus long terme

C'est toujours un exercice périlleux que de donner un plan de travail à long terme sur un sujet de nature exploratoire. Je pense qu'il faut confronter ces premières idées à l'expérience avant d'aller plus avant. Je me souviens qu'au début du projet PANDORE nous pensions "régler le problème" des tableaux pour passer très vite à des structures de données plus intéressantes comme les ensembles, les listes et les arbres (le travail de DEA d'Olivier Chéron portait d'ailleurs sur des schémas d'exécution distribués adaptés à ces structures de données). Nous avons vu au cours de la lecture de ce document que les problèmes de performances étaient tels que l'approche ne pouvait être validée que si nous montrions que l'efficacité des codes générés pouvait être améliorée.

Néanmoins, l'étude de la transformation "parallèle vers réparti" que nous avons commencée doit nous fournir des pistes :

- sur le modèle de programmation qu'il est intéressant de fournir au programmeur, et donc sur la sémantique qu'il est intéressant de préserver ;

– sur les moyens d'assurer que cette sémantique est préservée;

– sur l'intérêt des transformations mises en œuvre du point de vue de l'efficacité, autrement dit le "prix à payer" en performance est-il compatible avec les applications développées et est-il amorti par la facilité de programmation qui découlera de l'utilisation de notre outil?

Bibliographie

[Allen et Kennedy 87] Randy Allen et Ken Kennedy, « Automatic Translation of Fortran Programs to Vector Form », *ACM Transactions on Programming Languages*, vol. 9, n° 4, October 1987, p. 491–542.

[Ancourt et Irigoin 91] C. Ancourt et F. Irigoin, « Scanning Polyhedra with DO Loops », *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1991.

[André et al. 89] F. André, J.-L. Pazat et T. Priol, « Experiments with Mapping Algorithms on a Hypercube », *HCCA4*, 1989, Monterey (USA).

[André et Mahéo 96] F. André et Y. Mahéo, « CIDRE: Programming with Distributed Shared Arrays », *Proc. 3rd International Conference on High Performance Computing, Trivandrum India*, IEEE, décembre 1996.

[André et Pazat 88] F. André et J.-L. Pazat, « Le placement de tâches sur des architectures parallèles », *TSI*, vol. 7, n° 4, 1988, p. 385–401.

[André et Pazat 96] F. André et J.-L. Pazat, *A Multi-threads Runtime for The Pandore Data-Parallel Compiler*, HPCN'96, avril 1996, poster.

[André 96] F. André, *A Multi-threads Runtime for The Pandore Data-Parallel Compiler*, rapport technique n° 986, France, IRISA, février 1996.

[Banerjee 90] U. Banerjee, « Unimodular Loop Transformations of Double Loops », *Third Workshop on Languages and Compilers for Parallel Computing, Irivine, Cal.*, août 1990.

[Bareau et al. 93] C. Bareau, B. Caillaud, C. Jard et R. Thoraval, « Correctness of Automated Distribution of Sequential Programs », *PARLE'93, Parallel Architectures and Languages Europe*, A. Bode, M. Reeve et G. Wolf ed., p. 517–528, LNCS 694, Springer Verlag, juin 1993.

[Bareau 95] C. Bareau, *Distribution automatique de programmes séquentiels : étude structurelle et expérimentale*, Thèse de doctorat, IFSIC / Université de Rennes I, juillet 1995.

[Barreteau et Feautrier 95] M. BARRETEAU et P. FEAUTRIER, « Automatic mapping of scans and reductions », *High Performance Computing Symposium'95, Montreal (Canada)*, 1995.

[Benkner 94] S. BENKNER, *Vienna Fortran 90 and its Compilation*, thèse, Technical University of Vienna, 1994.

[Bodin et al. 93] F. BODIN, L. KERVELLA et T. PRIOL, « Fortran-S: a Fortran interface for Shared Virtual Memory Architectures », *Supercomputing'93*, 1993.

[Bodin 97] F. BODIN, *Transformation de programmes pour l'amélioration de performances*, Habilitation à diriger les recherches, IFSIC/Université de Rennes I, 1997.

[Bougé et Cachera 95] L. BOUGÉ et D. CACHERA, « On the completeness of a proof system for a simple data-parallel programming language », *Euro-Par'95*, Springer Verlag, 1995.

[Bougé 96] L. BOUGÉ, *The Data Parallel Programming Model*, *LNCS*, chapitre The Data Programming Model: A Semantic Perspective, p. 4–26, Number 1132 in LNCS, Springer Verlag, 1996.

[Brandes et Zimmermann 94] T. BRANDES et F. ZIMMERMANN, « ADAPTOR - A Transformation Tool for HPF Programs », *Programming Environments for Massively Parallel Distributed Systems, Birkhäuser Verlag*, K. M. Decker et R. M. Rehmann ed., p. 91–96, 1994.

[Brunie et Levèvre 94] L. BRUNIE et L. LEVÈVRE, « DOSMOS : A distributed shared memory based on PVM », *First european PVM users group meeting, Universita di Roma*, octobre 1994.

[Cabillic et al. 94] G. CABILLIC, T. PRIOL et I. PUAUT, *Myoan: an implementation of the Koan shared virtual memory on the intel paragon*, rapport technique n° 812, IRISA, avril 1994.

[Caillaud 94] B. CAILLAUD, *Contribution à la modélisation du* SPMD *: distribution asynchrone d'automates*, Thèse de doctorat, IFSIC/Université de Rennes 1, juin 1994.

[Callahan et Kennedy 88] D. CALLAHAN et K. KENNEDY, « Compiling Programs for Distributed-Memory Multiprocessors », *Journal of Supercomputing*, vol. 2, 1988, p. 151–169.

[Carriero et al. 86] N. CARRIERO, D. GELERNTER et J. LEICHTER, « Distributed data structures in Linda », *Proceedings of the 13th ACM Symposium on Pinciples of Programming Languages-St Petersbourg*, p. 236–242, janvier 1986.

[Cavalheiro et Doreille 96] G. CAVALHEIRO et M. DOREILLE, « A C++ library for parallel programming », *Stratagem'96, Inria Sophia Antipolis*, juillet 1996.

[Chandy et Misra 88] K. M. Chandy et J. Misra, *Parallel Program Design: A Foundation*, Addison Wesley, 1988.

[Chapman et al. 91] B. Chapman, P. Mehrotra et H. Zima, *Vienna Fortran: A Fortran Language Extension for Distributed Memory Multiprocessors*, rapport technique n° 91-72, ICASE, septembre 1991.

[Chapman et Zima 90] B. Chapman et H. Zima, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley, 1990, *Frontier Series*.

[Chaterjee et al. 93] S. Chaterjee, J. R. Gilber, F. J. E. Schreiber et S. H. Teng, « Generating Local Addresses and Communication Sets for Data Parallel Programs », *4th ACM Sigplan Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1993.

[Chéron 93] O. Chéron, *Pandore II : un compilateur dirigé par la distribution des données*, Thèse de doctorat, IFSIC/Université de Rennes I, juillet 1993.

[Christaller 96] M. Christaller, *Vers un support d'exécution portable pour applications parallèles irrégulières: ATHAPASCAN-0*, Thèse de doctorat, Université Joseph Fourier - Grenoble I, France, novembre 1996.

[Coelho et al. 96] F. Coelho, C. Germain et J.-L. Pazat, *Spring School on Data Parallelism*, *LNCS 1132*, chapitre State of the art in compiling HPF, p. 104–133, LNCS 1132, Springer Verlag, 1996.

[Coelho 96] F. Coelho, *Contribution à la compilation du High Performance Fortran*, Thèse de doctorat, Ecole des Mines de Paris, octobre 1996.

[Darte et Vivien 94] A. Darte et F. Vivien, *Automatic parallelization based on multidimensional scheduling*, rapport technique n° 94-24, LIP, septembre 1994.

[Darte et Vivien 95] A. Darte et F. Vivien, *A comparison of nested loops parallelization algorithms*, rapport technique n° RR95-11, LIP, mai 1995.

[Feautrier 89] P. Feautrier, *Parallel and Distributed Algorithms*, chapitre Semantical Analysis and Mathematical Programming: Application to Parallelization and Vectorization de , Elsevier Science Publishers B.V. (North Holland), 1989.

[Feautrier 91] P. Feautrier, « Dataflow Analysis of Scalar and Array References », *International Journal of Parallel Programming*, vol. 20, n° 1, 1991.

[Feautrier 92a] P. Feautrier, « Some Efficient Solutions to the Affine Scheduling Problem, Part I, One-Dimensional Time », *International Journal of Parallel Programming*, vol. 21, n° 5, 1992.

[Feautrier 92b] P. Feautrier, « Some Efficient Solutions to the Affine Scheduling Problem, part II, Multidimensional Time », *International Journal of Parallel Programming*, vol. 21, n° 6, 1992.

[Feautrier 96a] P. FEAUTRIER, *Automatic Parallelization in the Polytope Model*, rapport technique n° RR-96-08, PRiSM, 1996.

[Feautrier 96b] P. FEAUTRIER, « Placement automatique des données et des calculs », *TSI*, n° 15, 1996, p. 529–558.

[Folliot et Sens 94] B. FOLLIOT et P. SENS, « GATOSTAR: A Fault-tolerant Load Sharing Facility for Parallel Applications », *First European Dependable Computing Conference, Berlin, Allemagne*, Springer-Verlag, 1994.

[Forum 88] THE PARALLEL COMPUTING FORUM, *PCF Fortran: Language Definition*, Kuck and Associates, Champaign IL 61820, 1988.

[Geist et al. 94] A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK et V. SUNDERAM, *PVM: Parallel Virtual Machine-A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.

[Genaud et al. 95] S. GENAUD, E. VIOLARD et G.-R. PERRIN, « Transformation techniques in PEI », *EUROPAR*, LNCS, Springer Verlag, Stockholm, Sweden, août 1995.

[Germain et Delaplace 95] C. GERMAIN et F. DELAPLACE, « Automatic Vectorization of Communications for Data Parallel Programs », *Euro-Par'95*, Springer Verlag, 1995.

[Gerndt 90] M. GERNDT, « Updating Distributed Variables in Local Computations », *Concurrency Practice and Experience*, 1990.

[Guidec et Mahéo 95a] F. GUIDEC et Y. MAHÉO, « POM : une machine virtuelle parallèle incorporant des mécanismes d'observation », *Calculateurs Parallèles*, vol. 7, n° 2, 1995, p. 101–118, numéro spécial consacré aux environnements d'exécution de programmes parallèles.

[Guidec et Mahéo 95b] F. GUIDEC et Y. MAHÉO, *POM: a Virtual Parallel Machine Featuring Observation Mechanisms*, PI n° 902, IRISA, janvier 1995.

[Haines et al. 94] M. HAINES, D. CRONK et P. MEHROTRA, « On the Design of Chant: A Talking Threads Package », *Supercomputing '94, Washington D.C.*, p. 350–359, novembre 1994.

[Haines 95] M. HAINES, « Runtime Support for Data Parallel Tasks », *Fifth Symposium on the Frontiers of Massively Parallel Computation, McLean VA*, p. 432–439, février 1995.

[Hiranandani et al. 91] S. HIRANANDANI, K. KENNEDY, C. KOELBEL, U. KREMER et C.-W. TSENG, *An Overview of Fortran D Programming System*, rapport technique n° TR91121, RICE University, CRPC, mars 1991.

[HPFF 97] HIGH PERFORMANCE FORTRAN FORUM, *High Performance Fortran Language Specification*, rapport technique n° Version 2.0, Rice University, janvier 1997.

[Irigoin et al. 93] F. Irigoin, C. Ancourt, F. Coelho et R. Keryell, « A linear algebra framework for static HPF code distribution », *International Workshop on Compilers for Parallel Computers*, décembre 1993.

[Jézéquel et al. 97] J.-M. Jézéquel, T. Leseney, S. Matsuoka, J.-M. Pacherie, J.-L. Pazat et M. Sato, « Operators for Object-Oriented Meta-Computing », *OBPDC'97*, octobre 1997.

[Jézéquel 93] J.-M. Jézéquel, « EPEE: an Eiffel Environment to Program Distributed Memory Parallel Computers », *Journal of Object Oriented Programming*, vol. 6, n° 2, mai 1993, p. 48–54.

[Jézéquel 97] J.-M. Jézéquel, *Programmation fiable et efficace des architectures parallèles et distribuées*, Habilitation à diriger les recherches, IFSIC/Université de Rennes I, 1997, A paraître.

[Koelbel et Mehrotra 90] C. Koelbel et P. Mehrotra, *Supporting Shared Data Structures on Distributed Memory Architectures*, rapport technique n° csd-tr 915, Department of Computer Science, Purdue University, 1990.

[Lahjomri et Priol 92] Z. Lahjomri et T. Priol, « KOAN: a Shared Virtual Memory for the iPSC/2 hypercube », *CONPAR/VAPP92*, septembre 1992.

[Launay et Pazat 97] P. Launay et J.-L. Pazat, « Generation of distributed object-oriented programs », *ParCo 97*, septembre 1997.

[Lazure 95] D. Lazure, *Programmation géométrique à parallélisme de données : modèle, langage et compilation*, Thèse de doctorat, Université des sciences et technologies de Lille, janvier 1995.

[Le Fur et al. 95] M. Le Fur, J.-L. Pazat et F. André, « An Array Partitioning Analysis for Parallel Loop Distribution », *EUROPAR*, LNCS, Springer Verlag, Stockholm, Sweden, August 1995.

[Le Fur et Mahéo 95] M. Le Fur et Y. Mahéo, « Efficient Communications in Parallel Loop Distribution », *ParCo 95*, Gent, Belgium, septembre 1995.

[Le Fur 95a] M. Le Fur, *Compilation de boucles dirigée par la distribution des données*, Thèse de doctorat, IFSIC / Université de Rennes I, juillet 1995.

[Le Fur 95b] M. Le Fur, « Scanning Parameterized Polyhedron using Fourier-Motzkin Elimination », *High Performance Computing Symposium*, Montréal, Canada, juillet 1995.

[Le Fur 96] M. Le Fur, « Scanning Parameterized Polyhedron Using Fourier-Motzkin Elimination », *Concurrency Practice and Experience*, vol. 8, n° 6, juillet 1996, p. 445 – 460.

[Le Verge et al. 94] H. Le Verge, V. Van Dongen et D. K. Wilde, *Loop Nest Synthesis Using the Polyhedral Library*, rapport technique n° 2288, INRIA, mai 1994.

[Loechner et Mongenet 96] V. LOECHNER et C. MONGENET, « OPERA : A Toolbox for Loop Parallelization », *First International Workshop on Software Engineering for Parallel and Distributed Systems, Berlin*, mars 1996.

[Mahéo 95] Y. MAHÉO, *Environnement pour la compilation dirigée par les données : supports d'exécution et expérimentations*, Thèse de doctorat, IFSIC / Université de Rennes 1, juillet 1995.

[Muller et al. 96] G. MULLER, B. MOURA, F. BELLARD et C. CONSEL, *JIT vs Offline Compilers: Limits and Benefits of Bytecode Compilation*, rapport technique n° 1063, France, IRISA, 1996.

[Namyst et Méhaut 95] R. NAMYST et J.-F. MÉHAUT, « Parallel Multithreaded Machine. A computing environment for distributed architectures », *ParCo'95, Gent, Belgium*, 1995.

[Nieplocha et al. 94] J. NIEPLOCHA, R. J. HARRISON et R. J. LITTLEFIELD, « Global Arrays: A portable 'shared-memory' programming model for distributed memory computers », *Supercomputing'94*, 1994.

[Ning et al. 95] Q. NING, V. VAN DONGEN et G. R. GAO, « Automatic Data and Computation Decomposition for Distributed Memory Machines », *Proc. of the 28th Hawaii International Conference on System Sciences*, Wailea, Hawaii, janvier 1995.

[Pazat 89a] J.-L. PAZAT, « A control Replay Scheme for Distributed Computers », *1st European Workshop on Hypercubes and Distributed Computers*, F. André et J.P. Verjus ed., p. 105–116, North-Holland, 1989, Rennes (France).

[Pazat 89b] J.-L. PAZAT, *Outils pour la programmation d'un multiprocesseur à mémoires distribuées*, Thèse de doctorat, Université de Bordeaux I (France), février 1989.

[Pazat 91] J.-L. PAZAT, « Code Generation for Data Parallel Programs on DMPCs », *European Distributed Memory Computers Conference*, avril 1991.

[Pellegrini 95] F. PELLEGRINI, *Application de méthodes de partition à la résolution de problèmes de graphes issus du parallélisme*, Thèse de doctorat, Université de Bordeaux I, janvier 1995.

[Quinton et al. 94] P. QUINTON, S. RAJOPADHYE et D. WILDE, *Using static analysis to derive imperative code from alpha*, rapport technique n° 828, IRISA, mai 1994.

[Ramanujam 92] J. RAMANUJAM, « Non-unimodular transformations of nested loops », *IEEE Computer*, vol. 16, n° 20, novembre 1992.

[Reffay 96] C. REFFAY, *Génération de code paralèle à partir d'équations récurrentes*, Thèse de doctorat, Université de Franche-Comté, janvier 1996.

[Snir et al. 95] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker et J. Dongarra, *MPI: The Complete Reference*, MIT Press, 1995, ISBN 95-80471.

[Thomas 91] H. Thomas, *Une approche de la compilation de programmes séquentiels pour machines à mémoire distribuée*, Thèse de doctorat, IFSIC/Université de Rennes I, juin 1991.

[Tseng 93] C.-W. Tseng, *An Optimizing Fortran D Compiler for MIMD Distributed Memory Machines*, thèse, RICE University, 1993.

[Wolf et Lam 87] M. E. Wolf et M. Lam, « A Loop Transformation Theory and an Algorithm to Maximize Parallelism », *IEEE Transactions on Parallel and Distributed Systems*, octobre 1987.

[Wolfe 89] Michael Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman Publishing, 1989, MIT Press édition, *Research Monographs in Parallel and Distributed Computing*.

[Zima et al. 88] H. P. Zima, H.-J. Bast et M. Gerndt, « SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization », *Parallel Computing*, n° 6, 1988, p. 1–18.

[Zima et Chapman 93] H. Zima et B. Chapman, « Compiling for Distributed-Memory Systems », *Proc. of the IEEE*, vol. 81, n° 2, 1993, p. 264–287.