Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle

François Galilée

▶ **To cite this version:**

François Galilée. Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1999. Français. NNT : . tel-00004832

HAL Id: tel-00004832
https://theses.hal.science/tel-00004832

Submitted on 18 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse

présentée par

François Galilée

pour obtenir le grade de Docteur

de l'Institut National Polytechnique de Grenoble

(Arrêté ministériel du 30 mars 1992)

Spécialité : Informatique

Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle

Date de soutenance : 22 septembre 1999

Composition du jury :

Jean-Marc Geib, *rapporteur*
François Irigoin, *rapporteur*
Guy Mazaré, *président du jury*
Brigitte Plateau, *directeur de thèse*
Jean-Louis Roch, *responsable de thèse*

Thèse préparée au sein de
l'Unité Informatique et Distribution (ID-IMAG)

Remerciements

À mon époque, la France offrait encore à tous ses jeunes une année entière de vacances. Moi, j'étais plagiste dans la Marine : le pont du mythique bateau gris en fer malmené par la tempête avait été remplacé par une bande de sable bordée par une pinède d'une île baignée de soleil[1]. Dans ces conditions, bien trop occupé à parfaire mon bronzage et ma technique au baby-foot, je n'avais qu'une idée bien vague de ce qui pouvait autant exalter Jean-Louis quand il me disait que le projet avait beaucoup évolué depuis mon passage en DEA. Mais j'ai pu constater, lors de mon retour, combien en effet l'équipe était rodée. Sous la direction de Jean-Louis et la maîtrise de Mathias, elle s'orientait déjà vers une nouvelle définition de l'interface Athapascan-1. J'ai alors eu la chance, grâce et aux côtés de Mathias et Gerson, de participer à l'épopée de son développement.

Mes trois années de thèse se sont déroulées en leur étroite compagnie. Je leur suis donc énormément reconnaissant et redevable. Pour leur accueil, leur patience et leur sympathie : Mathias a su supporter mes questions les plus stupides, Gerson mes râleries et ma mauvaise humeur, et Jean-Louis mes « passagères » démotivations (Dieu que le début a été dur !). Cette thèse, bien que signée de mon seul nom, ne doit donc pas être attribuée à un travail solitaire : elle reflète ces trois années de travail mené ensemble ; de jour, de nuit, de week-end, de jours fériés...

Merci alors à Guy Mazaré, pour m'avoir fait l'immense honneur de présider le jury de ma soutenance. Merci à Jean-Marc Geib et François Irigoin pour leurs précieuses critiques sur les versions préliminaires de ce document. Ces critiques m'ont réellement permis de bien mieux cerner ma rédaction et en ont assurément amélioré la qualité. Merci à Brigitte Plateau, figure de proue de tout le projet APACHE, pour son accueil, son dévouement et son perpétuel sourire. Merci, pour tout, à Jean-Louis.

Merci également à tous les autres habitants du couloir qui constitue notre laboratoire. C'est un environnement de travail exceptionnel où, placé à une extrémité et en criant suffisamment fort, on peut arranguer tous ses locataires. Un merci particulier aux fous séminaristes du mardi pour ces moments d'évasion inoubliables et cette folle compétition avec son règlement si juste[2]. Merci au soutien moral, nocturne et désaltérant des écrivains

[1] Vous avez raison, c'est pas loin de la définition du paradis. Il manquait hélas les vahinées...

[2] Séminaires sponsorisés par Dynastar et les pâtés Hénaff. J'étais en tête jusqu'un fameux 26 janvier où la rotule de mon genou droit décida de partir à gauche. Les deux grands vainqueurs furent alors Jean-Guillaume et Mathias, à égalité avec 9 points. Par équipe, la France bat le Brésil 38 à 22. Désolé Gustavo !

malgré eux : Alexandre, Alfredo, B^{EN}H$_U$R, Gerson, Gregory, Mathias et Olivier. Merci à Yves, Thierry et Nicolas pour les relectures quasi forcées de l'intégralité de mes œuvres. Merci aussi à Ilan pour m'avoir refourgué une moto naze dont les pannes répétées m'ont permis d'aller bosser plutôt que d'écumer les virolos. À Bruno, mon frère, pour avoir financièrement soutenu ce projet fumant. À Marcelo pour avoir dit que si, il trouvait qu'elle ressemblait à une moto. Et à Johan pour les virées du temps où nos étrons respectifs dépassaient encore le 100. Merci à GOX pour son accueil constant lors de mes escapades parisiennes et à Ijleta pour celui des versions strasbourgeoises. Merci également à tous ceux qui resteront ici anonymes mais à qui je dois tout autant et qui sauront sans nul doute se reconnaître.

Je tiens évidemment à remercier, bien que bravant sa plus stricte et formelle interdiction, Stéphanie, Stephanie, Steffi, Steffiline, Steff, Chteff, enfin celle pour qui, chaque jour, je fais de mon mieux pour être à ses yeux un véritable héros.

Enfin, merci à celui qui a su me donner le goût pour la science, la technique, l'honnêteté et un minimum de rigueur et, *a posteriori*, l'envie, la joie et la soif de vivre. C'était il y a de cela une vingtaine d'années, en 6ème, un petit prof de maths. J'étais bien bien moyen et dissipé ; il a su me guider et me convaincre. Mon père, pour cela, l'intégralité de mon travail te sera toujours et à jamais dédiée.

Table des matières

Table des figures

Liste des tableaux

1

Introduction

L'utilité des machines parallèles est aujourd'hui unanimement reconnue par la communauté scientifique. De plus, l'évolution rapide de la technologie en ce qui concerne l'architecture matérielle et les réseaux est telle que les machines symétriques multiprocesseurs[1], tout comme les interconnexions « rapides » de stations de travail[2], sont de plus en plus performantes et répandues.

Ainsi, la programmation de telles machines commence à devenir populaire, comme en témoigne la multiplicité des langages parallèles : parallélisation automatique dans certains langages de programmation logique (Prolog), annotations pour une extraction implicite du parallélisme (HPF [66], Jade [93]), créations de fils d'exécution concurrents dans les langages à base de processus légers (bibliothèque Pthreads [69], Java [85], OpenMP [31]), créations explicites de processus communicants par échanges de messages (PVM [101], MPI [79, 40], PM^2 [81], Athapascan-0 [19]), créations explicites de tâches concurrentes (Cilk [14], NESL [10], Athapascan-1 [28]), réalisation d'une mémoire virtuelle partagée permettant une programmation parallèle des machines à mémoire distribuée (Linda [59, 22]), *etc*.

Si ces langages permettent une description aisée du parallélisme contenu dans l'application, la performance, but de toute programmation parallèle, constitue quant à elle un des plus épineux problèmes. Ce problème est double : d'une part obtenir ces performances, et d'autre part, garantir leur **portabilité**. L'obtention résulte de l'ordonnancement (placement et date d'exécution) des tâches qui sera effectué. La portabilité résulte de la capacité de l'ordonnancement à s'adapter aux conditions particulières de l'exécution, c'est-à-dire essentiellement à la machine hôte. Par exemple, et à l'extrême limite, la stratégie d'ordonnancement doit pouvoir décider d'une exécution purement séquentielle de l'application si l'exploitation du parallélisme sur la machine hôte est trop coûteux.

[1] *SMP, Symmetric Multi-Processors*.

[2] « Réseaux » (*NOW, Network of Workstations*), ou « grappes » (*COW, Cluster of Workstations*) de stations de travail. Ces architectures sont présentées brièvement à la section 2.2 page 25.

Cette thèse se situe dans le cadre des langages parallèles offrant une solution à ce problème de portabilité. Ces langages effectuent une abstraction de la machine et offrent un modèle de coût permettant de garantir, à partir du code source de l'application et des caractéristiques de la machine hôte, les performances en temps et en mémoire de l'exécution. Cette garantie est possible grâce à l'analyse du comportement des stratégies d'ordonnancement utilisées par l'implantation du langage.

La thèse que je soutiens et défends au cours de ce document est alors la suivante :

> *Le **graphe de flot de données** associé à toute exécution parallèle peut constituer l'**élément central** de la définition et de l'implantation de langages parallèles portables et efficaces.*

En effet, la description du flot de données d'une application par l'utilisateur permet de détecter et d'exploiter le parallélisme de l'application (en définissant une sémantique aux accès aux données), de garantir les performances à l'exécution (en utilisant de manière implicite un ordre total d'exécution des tâches) et de fournir une information précise sur l'application à l'ordonnanceur.

Cette thèse a été menée dans le cadre du projet APACHE[3] dont le but est la définition d'un environnement de programmation portable et efficace des machines parallèles : Athapascan. Cet environnement est essentiellement constitué par trois modules :

- **Athapascan-0** [19, 61], le module exécutif, qui permet l'utilisation de la multiprogrammation légère dans un contexte distribué. Ceci est réalisé par un mariage entre une bibliothèque de processus légers et une bibliothèque de communication. Ce module, disponible sur de nombreuses architectures, constitue la couche de portabilité de l'environnement Athapascan. Ce module est brièvement présenté à la section 7.2.1 page 131.
- **Athapascan-1** [28, 55], l'interface applicative, qui permet la programmation d'une application parallèle de façon simple et de haut niveau. Ceci est réalisé par une description indépendante de l'architecture du flot de données d'une application par création explicite de tâches déclarant les accès effectués sur la mémoire virtuelle partagée offerte par le langage. La stratégie d'ordonnancement des tâches est entièrement séparée du code de l'application. Cette séparation, outre le fait qu'elle simplifie la programmation de l'application, permet d'adapter aisément la politique d'ordonnancement à la machine cible et à l'application. L'interface applicative est brièvement décrite à la section 2.8 page 44 et plus complètement dans la thèse de Mathias Doreille [42]. Le module chargé de l'exécution et de l'ordonnancement des tâches est quant à lui succinctement présenté à la section 7.2 page 130 et constitue le cœur de la thèse de Gerson Cavalheiro [25].

[3] Algorithmique Parallèle et pArtage de CHargE [88], projet joint CNRS–INPG–UJF–INRIA (Centre National de la Recherche Scientifique – Institut National Polytechnique de Grenoble – Université Joseph Fourier – Institut National de la Recherche en Informatique et en Automatique). L'*URL* officielle du projet est `http://www-apache.imag.fr`.

– **Paje** [33, 32], le module de visualisation, qui par sa généricité permet de représenter toutes sortes d'événements au niveau d'Athapascan-0, du module d'ordonnancement, d'Athapascan-1 et même au niveau de l'application programmée par l'utilisateur. Cet outil constitue une précieuse aide lors du débogage et de la mise au point des applications.

De nombreuses applications ont été écrites au sein du projet en utilisant cet environnement. Citons entre autres une application de dynamique moléculaire [7], une application de type décomposition de domaine [3], un code de chimie quantique [75], une bibliothèque de calcul symbolique [56] et un code de recherche combinatoire de type *branch and bound* [35].

Mon travail au cours de cette thèse se situe essentiellement au niveau du développement de l'interface applicative Athapascan-1, ainsi, bien qu'à une moindre mesure, au niveau du module d'ordonnancement ayant proposé (et implanté) deux stratégies nouvelles d'ordonnancement. Mes contributions essentielles, au cours de cette thèse, sont alors les suivantes :

– Définition de la sémantique des accès aux données d'Athapascan-1.
– Implantation distribuée et évaluation du graphe de flot de données dans la bibliothèque Athapascan-1. Ce graphe est construit à la volée lors de l'exécution et constitue le noyau central de cette implantation d'Athapascan-1.
– Définition du modèle de coût associé au modèle de programmation d'Athapascan-1 permettant de garantir les performances en temps et en mémoire de toute exécution.
– Implantation et évaluation de deux stratégies d'ordonnancement permettant le contrôle de la consommation mémoire de toute exécution.

Ce document, dont le thème central est la **définition** et **l'exploitation du flot de données** d'une exécution, est structuré en deux parties : la première, constituée des chapitres 3, 4 et 5, concerne l'interprétation du flot de données dans un contexte distribué et la seconde, composée des chapitres 6 et 7, concerne l'utilisation de ce graphe de flot de données dans les modèles de coût permettant de garantir la durée et le volume mémoire nécessaires à toute exécution.

En guise d'introduction le chapitre 2 présente, en étudiant différents langages de programmation parallèle, l'**utilité de la modélisation** de l'exécution par un graphe de flot de données pour la définition de la sémantique et du modèle de coût associé aux langages.

Le chapitre 3 **définit le graphe de flot de données** associé à une exécution et détaille un algorithme permettant sa **construction à la volée** dans le cadre de langages exprimant un parallélisme de type emboîté (*nested parallelism*) où une tâche fille ne peut accéder qu'un sous ensemble des données accédées par sa mère (il n'y a pas d'effet de bord sur la mémoire partagée). À chaque nœud de ce graphe est associé un état qui permet de résoudre les contraintes de précédence entre les tâches. Cette résolution permet de définir la **sémantique des accès aux données** dans l'interface de programmation parallèle Athapascan-1.

Le chapitre 4 étudie le problème de **la gestion distribuée des états** associés aux nœuds du graphe dans un environnement de processeurs faiblement couplés. Cette gestion est basée sur un algorithme de détection de terminaison globale : il faut par exemple détecter la fin des accès répartis en écriture sur un même objet (éventuellement répliqué) avant d'autoriser les lectures sur celui-ci. Ces algorithmes sont couramment employés dans les systèmes de ramasse-miettes distribués mais la contrainte de réactivité est rarement prise en compte : il est important, pour l'utilisation que nous faisons de l'algorithme de terminaison, d'être averti d'une terminaison globale le plus tôt possible, idéalement au moment où elle survient.

Le chapitre 5 détaille l'**implantation et l'évaluation du graphe de flot de données** dans Athapascan-1. Le coût de construction du graphe, création d'une tâche ou d'une donnée partagée, est borné et ne génère aucune communication : les communications ne sont nécessaires que pour la gestion des parties distribuées du graphe, par suite de migration de tâche. Des mesures effectuées sur une machine « classique » permettent d'estimer numériquement le surcoût (en temps et en espace mémoire) de la création et de la gestion de ce graphe.

Le chapitre 6 concerne l'utilisation du graphe de flot de données comme support pour la **définition d'un modèle de coût** permettant de majorer *a priori* la durée et la consommation mémoire de toute exécution à partir de données élémentaires. À partir de ce graphe et d'un ordre total sous-jacent des tâches (l'ordre de « référence »), nous définissons deux stratégies d'ordonnancement permettant de **garantir les performances en temps et en mémoire** de toute exécution d'un programme Athapascan-1.

Le chapitre 7 détaille l'**implantation et l'évaluation**, dans le cadre d'Athapascan-1, des deux **stratégies d'ordonnancement** présentées au cours du chapitre 6. Les évaluations sont effectuées sur une application test de calcul d'une portion de l'ensemble de Mandelbrot et permettent de comparer, en terme de consommation mémoire, les deux stratégies aux politiques d'ordonnancement classiques de type « arbitraire » et « gloutonne ».

Enfin, le chapitre 8 conclut ce document et présente certaines perspectives envisageables à partir de ce travail.

2

Langages de programmation parallèle et flot de données

L'objectif de ce chapitre est de présenter quelques langages parallèles offrant une sémantique et un modèle de coût, c'est-à-dire permettant une majoration *a priori* du coût d'exécution en temps et en mémoire. Les langages présentés sont les langages à base de processus légers, Jade, BSP, NESL, Cilk et Athapascan-1. Il ressort de cette étude que la définition des modèles de coût peut être basée sur la modélisation de l'exécution par un graphe de flot de données.

2.1 Objectif

L'objectif de ce chapitre est de présenter les langages parallèles définissant une sémantique des accès aux données et permettant une majoration *a priori* du coût d'exécution, c'est-à-dire un prédiction des performances en temps et en mémoire de toute exécution : un modèle de coût est alors associé au modèle de programmation. Ce modèle de coût est basé sur une étude théorique du comportement de la politique d'ordonnancement lors de l'exécution, c'est-à-dire sur une estimation du nombre de tops d'inactivité des processeurs et de l'ordre d'exécution des tâches de l'application. Cette étude montre que la notion de graphe permettant de modéliser une exécution est centrale dans la définition de la sémantique et du modèle de coût. Ce graphe, de précédence ou de flot de données, sera le thème central de cette thèse et son apport pour ces deux définitions sera largement discuté.

Nous présentons tout d'abord les **langages à base de processus légers** qui ne permettent pas dans un cas général de prédire ni la sémantique des accès aux données, ni la durée de l'exécution, ni la consommation mémoire d'une exécution à partir d'une analyse simple du code source de l'application. Il est donc nécessaire d'introduire des contraintes supplémentaires sur les tâches afin d'être capable d'envisager un modèle de coût. En restreignant les poss ces contraintes permettent une certaine connaissance sur le comportement de l'application.

Nous présentons ensuite **Jade** [93], qui par une analyse du flot de données de l'application permet d'extraire implicitement le parallélisme d'une application sans modifier la sémantique séquentielle des accès aux données partagées. Ce langage ne prédit pas les performances attendues à l'exécution, mais l'analyse du flot de données, du même esprit que celle d'Athapascan-1, en fait un langage original.

Nous présentons ensuite le modèle de programmation **BSP** qui, par une technique particulière de routage des communications, permet de prédire la durée de toute exécution à partir de mesures simples sur la machine et sur l'application.

Nous présentons enfin **NESL** [10], **Cilk** [14] et **Athapascan-1** [28], langages qui possèdent tous trois une sémantique des accès aux données partagées et un modèle de coût permettant de garantir les performances de toute exécution. Les deux premiers sont basés sur une analyse du graphe de précédence de l'application, tandis que le dernier est basé sur une analyse du graphe de flot de données, comme dans Jade. Un état de l'art plus complet concernant les langages de programmation parallèle peut être trouvé au chapitre 5 de la thèse de Martin C. Rinard [92].

Avant de présenter ces langages, nous précisons dans la section suivante les critères qui vont orienter cette présentation : la génération du parallélisme, la sémantique des accès aux données, l'ordonnancement, le modèle de coût associé au langage et enfin les principaux types de machine visés.

2.2 Quelques problèmes clés en programmation parallèle

L'utilisation d'une machine comportant plusieurs processeurs pour exécuter une application implique l'exécution en parallèle de certaines portions de code, ou tâches[1], de cette application. Les problèmes suivants doivent être résolus :

- La **définition** des différentes tâches de l'application, ainsi que leurs éventuelles contraintes de précédence : classiquement, par exemple, une tâche lecteur doit attendre la fin des tâches écrivains.
- La définition de la **sémantique** des accès à la mémoire, que cette mémoire soit partagée ou distribuée, c'est-à-dire la valeur retournée par tout accès à une donnée lors de toute exécution.
- Le calcul de l'**ordonnancement** des tâches, c'est-à-dire leur placement et leur date d'exécution, sur les processeurs de la machine.
- La **prédiction** de la durée et de la consommation mémoire de l'exécution compte tenu de l'application, de la machine et de l'ordonnancement.

[1] Intuitivement une tâche représente l'exécution d'une portion de code dans un espace d'adressage privé. Les tâches peuvent cependant partager un autre espace mémoire entre elles.

Description du parallélisme Cette description peut être réalisée de manière automatique, implicite ou explicite.

La situation idéale est bien sûr celle où l'expression des tâches est **automatique** : il suffit d'utiliser un compilateur particulier [6, 74]. Cette détection du parallélisme n'est hélas pas toujours possible statiquement. Considérons l'exemple de la boucle présentée dans la figure 2.1 page 23 : l'itération peut être parallélisée seulement si les deux tableaux a et b ne se recouvrent pas. Or, lors de la compilation, il est impossible d'avoir cette information si cette fonction est appelée depuis une unité de compilation autre que celle où est compilée cette itération : il est donc nécessaire, soit de ne pas générer de parallélisme, soit d'effectuer un test lors de l'exécution permettant de décider si les tableaux se recouvrent ou non[2]. Des difficultés similaires se posent lorsque les indices d'accès aux tableaux sont des appels de fonctions par exemple ou avec des indirections (`a[b[i]]` par exemple). Dans un cadre restreint où les indices des tableaux accédés sont des fonctions affines des indices d'itération, il est cependant possible d'effectuer une analyse et une parallélisation statique de ce type de nids de boucles.

```
void f( int n, int*& a, int*& b )
{
   for( int i=0; i<n; i++ )
      a[i] = a[i] + b[i];
}
```

Figure 2.1 *Boucle ne permettant pas une détection automatique aisée du parallélisme. La boucle* for *de cette fonction C++ peut être exécutée en parallèle si les tableaux a et b ne se recouvrent pas. Sinon, sur toute la partie de recouvrement, aucun parallélisme n'est possible si l'on veut conserver le caractère déterministe du résultat de l'itération : les calculs doivent alors être menés en séquence.*

La description **implicite** du parallélisme se fait par une annotation du code source de l'application qui permet de spécifier au compilateur les endroits où du parallélisme peut être généré. C'est le cas par exemple des directives `DOACCROSS` sur certains[3] compilateurs Fortran, `FORALL` en HPF [66] spécifiant un nid de boucles parallélisable, les indications d'accès aux données en Jade ou les directives du standard Open-MP [31, 68].

Enfin, la dernière alternative est la description **explicite** des portions de code pouvant être exécutées en parallèle. L'utilisateur crée lui-même les tâches de son application en utilisant des mots clés du langage : `spawn` en Cilk, l'opérateur `{}` en NESL, la dérivation de la classe `Thread` en Java [85] ou en utilisant des appels à des fonctions de bibliothèque, par exemple `Fork<>()` en Athapascan-1 ou `pthread_create()` pour les processus légers implantant la norme POSIX [69]. Les contraintes de précédence entre les tâches créées seront, soit explicites, soit implicites selon le modèle de programmation du langage.

[2] Il y a, lors de la compilation, génération statique des deux versions de code et d'un test dynamique qui permettra de choisir, lors de l'exécution, entre la version séquentielle et la version parallèle (selon que les tableaux se recouvrent ou non).

[3] Par exemple le compilateur Fortran sur les SGI/Cray Origin-2000.

Graphe de tâches L'exécution, c'est-à-dire l'ensemble des tâches, des données accédées et leurs interactions, peut être représentée par un graphe de précédence ou de flot de données [54]. La forme de ce graphe dépend du parallélisme qui est exploité par le langage :

- Le parallélisme peut être de type **« série-parallèle »** si les synchronisations entre les tâches sont effectuées par fratrie : la tâche mère[4] est seule capable de synchroniser ses filles, et cette synchronisation est globale sur l'ensemble des filles créées. Un exemple typique de tels graphes est celui généré par le langage NESL, figure 2.6 page 38.
- Le parallélisme peut être de type **« emboîté »** lorsque les accès aux données partagées effectués par les filles constituent un sous ensemble des accès effectués par la tâche mère. Un exemple typique de tels graphes est celui généré par l'interface de programmation Athapascan-1, figure 2.9 page 45. Les règles de portées et les passages de paramètres autorisés sont définis par le langage (voir par exemple le tableau 3.2 page 66 en ce qui concerne Athapascan-1).

Sémantique des accès aux données Si il est communément associé aux langages séquentiels une sémantique des accès aux données de type lexicographique, où l'ordre des lectures et des écritures à l'exécution est déterminé par l'ordre des instructions dans le code source de programme, la définition d'une sémantique pour un langage parallèle n'est pas aussi aisée.

En effet, l'exécution en parallèle de différentes tâches accédant toutes une même mémoire commune, partagée ou distribuée, risque d'introduire des situations de concurrence sur certaines zone de cette mémoire (*race-conditions*). Ces conflits peuvent mener à des résultats dépendant des conditions d'exécution, c'est-à-dire de l'ordonnancement des tâches.

Afin de régler ces accès concurrents à la mémoire, les tâches doivent être synchronisées. Ces synchronisations sont, soit **explicites**, comme par exemple pour les langages à base de processus légers ou pour Cilk, soit **implicites** et déterminées à partir de la déclaration par les tâches des accès effectués à la mémoire, comme dans Jade ou Athapascan-1. On parle alors de langages basés sur une analyse du **flot de données**.

Certains langages, tels que Jade, NESL, Cilk ou Athapascan-1, garantissent une sémantique des accès aux données semblable à la sémantique séquentielle : deux exécutions différentes avec les mêmes valeurs en entrée mènent donc aux mêmes résultats[5].

Ordonnancement L'ordonnancement des tâches consiste à affecter à chaque tâche un site et une date d'exécution. Certains langages, tels que PVM [101], MPI [79, 40] ou HPF[6] [66], laissent à l'utilisateur la résolution de ce problème. Nous ne considérons ici que les langages offrant une abstraction de la machine d'exécution et qui résolvent le problème

[4] La notion de tâche mère/tâche fille correspond au contexte de création de la tâche : la mère de la tâche créée est la tâche qui exécute l'instruction de création.

[5] Notons que pour ces langages l'ordre d'exécution séquentiel des tâches est un ordre d'exécution valide.

[6] HPF fournit également un placement par défaut.

de l'ordonnancement. Nous qualifions ces langages de « haut niveau » par opposition à ceux laissant ce travail d'ordonnancement à la charge de l'utilisateur.

Cet ordonnancement doit en général être effectué de manière dynamique, ni les caractéristiques de la machine ni les conditions d'exécution, n'étant connues lors de la compilation. Ainsi les langages à base de processus légers, Jade, NESL et Cilk offrent une ou plusieurs stratégies d'ordonnancement à la volée. Dans Athapascan-1, l'ordonnancement est entièrement séparé du reste de l'implantation ce qui permet de développer un grand nombre de stratégies d'ordonnancement, statiques, dynamiques, ou mixtes.

Il va sans dire que la qualité de l'ordonnancement a une influence directe et cruciale sur les performances de l'exécution.

Modèle de coût Le choix d'utilisation d'une machine parallèle est en général motivé par un désir de performance : soit réduire la durée de calcul d'une application, soit être capable de traiter des problèmes plus « gros » en utilisant les capacités mémoire de plusieurs machines.

Cependant, si la durée d'exécution et le volume mémoire nécessaires à une exécution séquentielle sont en général prévisibles avant l'exécution par une analyse du code source et des données en entrée, il n'en est hélas pas de même pour les exécutions parallèles. En effet, les performances dépendent de l'ordonnancement des tâches qui sera effectué, et le calcul de l'ordonnancement optimal n'est en général pas envisageable. De plus, l'utilisation de parallélisme nécessite une découpe de l'application en tâches et un mécanisme de synchronisation (et éventuellement un mécanisme de communication) entre celles-ci : le travail (*i.e.* le nombre total d'instructions exécutées) est alors plus important que pour une simple exécution séquentielle et la durée d'exécution peut donc être supérieure.

De plus, l'exécution de tâches en parallèle implique également des allocations mémoire en concurrence : la consommation mémoire peut donc être largement supérieure à celle d'une exécution séquentielle. Par exemple, le calcul récursif du n-ième nombre de Fibonacci $F(n)$ nécessite un espace mémoire de n si l'arbre des appels est parcouru en profondeur d'abord et un volume de 2^n si le parcours est effectué en largeur d'abord. Le choix de l'ordonnancement est donc crucial pour permettre de contrôler à la fois les périodes d'inactivités des processeurs et la consommation mémoire lors de l'exécution.

Si la prédiction des performances semble sans espoir dans un cas général, en imposant certaines restrictions sur le parallélisme exploitable par les modèle de programmation il est possible d'estimer durée et consommation mémoire. Ainsi les langages NESL, Cilk et Athapascan-1 associent un modèle de coût au modèle de programmation, modèle qui permet de garantir les performances, en durée et en mémoire, de toute exécution. Ces garanties sont calculées à partir des caractéristiques de l'instance de l'application[7] considérée.

Types de machine Les deux principaux types de machine parallèle que nous considérons sont les machines à mémoire partagée et les machines à mémoire distribuée.

[7]Nous appelons « instance d'application » l'association d'une application et d'un ensemble de ses valeurs d'entrées.

Dans le cas d'une mémoire partagée, nous ne considérons que les machines de type *SMP*, acronyme de *« Symmetric Multi-Processors »*, où tous les processeurs sont identiques et accèdent une même mémoire commune : un processus peut alors migrer librement d'un processeur à un autre. L'accès à la mémoire peut être uniforme (machines de type *UMA, Uniform Memory Access*), lorsque la durée d'accès à une zone mémoire est indépendante du processeur effectuant la requête, ou, dans le cas contraire, non uniforme (machines de type *NUMA, Non Uniform Memory Access*).

Dans le cas d'une mémoire distribuée, la machine est constituée par une interconnexion de stations de travail. On parle alors de *NOW*, acronyme de *« Network Of Workstations »*, ou de *COW*, acronyme de *« Cluster Of Workstations »*. La différence principale réside dans le fait que les *clusters* sont généralement dédiés au calcul et ne sont pas partagés entre plusieurs utilisateurs. Les réseaux de stations peuvent, en général, être utilisés par plusieurs applications simultanément. Ces différentes architectures sont présentées plus en détail, entre autre, dans [20].

Nous ne nous intéressons dans la suite de ce chapitre qu'aux langages parallèles que nous qualifions de « haut niveau », langages offrant une abstraction de la machine d'exécution et un ordonnancement « automatique » permettant d'associer un modèle de coût au modèle de programmation. Nous présentons les langages basés sur des processus légers, Jade, NESL, Cilk et Athapascan-1. Nous donnons pour chacun des langages leurs particularités en fonction des critères de génération du parallélisme, de sémantique des accès aux données, d'ordonnancement, de modèle de coût associé au langage et du type d'architecture visé. Pour chacun de ces langages la génération du parallélisme est illustrée sur un code de calcul du n-ième nombre de Fibonacci, code qui utilise un algorithme récursif prohibitif mais qui a l'avantage de générer de manière simple du parallélisme de type diviser pour paralléliser (*divide and conquer*). De plus, nous donnons le graphe associé à un exemple d'exécution de ce code, les modèles de coût étant basés sur ce graphe.

2.3 Langages à base de processus légers

Un **processus léger**, ou selon Dijkstra [37] un « processus coopérant », constitue un flot d'exécution autonome qui coopère avec ses semblables par partage d'une mémoire commune et par des mécanismes de synchronisation. Les processus légers sont de plus en plus « populaires » (comme en témoignent les nombreuses implantations, celle du langage Java par exemple), leur utilisation permettant une exploitation rapide d'un parallélisme « simple »[8].

Nous présentons dans cette section une utilisation de ces processus légers permettant une programmation d'applications parallèles.

[8]Par exemple, l'interactivité de nombreux logiciels repose sur un mécanisme de processus légers : le principe de base est d'avoir un processus léger dédié à l'écoute de l'utilisateur et un autre dédié à la tâche de calcul.

2.3.1 Génération du parallélisme

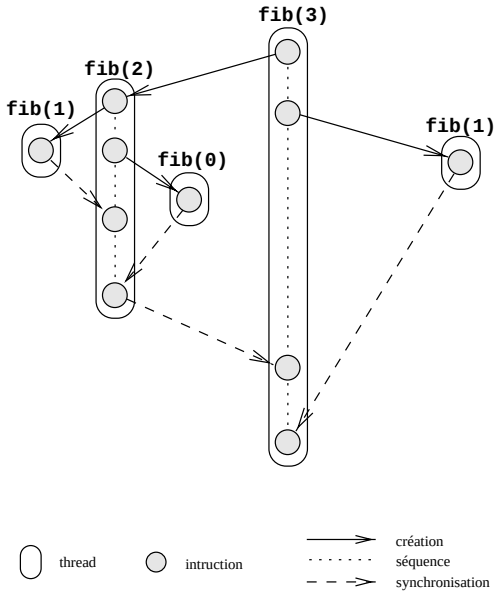
Le parallélisme peut être exprimé en créant un processus léger pour exécuter de manière totalement asynchrone une procédure, comme illustré figure 2.2(a). Un processus léger commence l'exécution de la procédure dès sa création et est détruit lorsque cette procédure arrive à sa dernière instruction.

Entre le moment de sa création et celui de sa destruction, chaque processus léger est entièrement autonome. Les processus légers se partagent les resources du système de manière équitable[9]. Ces processus légers, bien qu'autonomes, peuvent être synchronisés en partie ou dans leur ensemble à l'aide de certaines instructions offertes par les langages. Ces synchronisations sont utilisées pour gérer l'accès à la mémoire partagée, comme présenté dans la section suivante.

```
void *fib( void *n )
{
  void *r= malloc( sizeof( int ) );
  if( *(int*)n < 2 )
    *(int*)r= *(int*)n;
  else {
    int n1= *(int*)n - 1;
    int n2= *(int*)n - 2;
    void *r1, *r2; pthread_t t1, t2;
    pthread_create( &t1, NULL, fib, &n1 );
    pthread_create( &t2, NULL, fib, &n2 );
    pthread_join( t1, &r1 );
    pthread_join( t2, &r2 );
    *(int*)r= *(int*)r1 + *(int*)r2;
    free( (int*)r1 ); free( (int*)r2 );
  }
  pthread_exit( r );
}

int main( int argc, char** argv )
{
  int n= atoi( argv[1] );
  void *r; pthread_t t;
  pthread_create( &t, NULL, fib, &n );
  pthread_join( t, &r );
  printf( "fib(%d)=%d\n", n, *(int*)r );
  free( (int*)r );
  return 0;
}
```

(a)



(b)

Figure 2.2 *Calcul du n-ième nombre de Fibonacci à l'aide de Pthreads.*
En (a), dans le code C utilisant les Pthreads, l'expression du parallélisme se fait en créant un processus léger (`pthread_create`*) auquel est associé une fonction. La fonction* `pthread_join()` *permet d'attendre la terminaison d'un processus léger. En (b), le graphe d'exécution d'un appel à* `fib(3)` *est présenté. Ce graphe est de type série-parallèle dans cet exemple, mais est quelconque dans le cas général.*

[9] La notion d'équité repose entièrement sur la politique d'ordonnancement utilisée.

2.3.2 Flot de données et sémantique

Les processus légers partagent tous un même espace mémoire, et ce sans aucun contrôle. La concurrence sur les accès à cette mémoire doit être maîtrisée par l'utilisateur qui dispose de nombreux moyens pour synchroniser les processus légers entre eux : exclusion mutuelle, sémaphore, barrière, moniteur, *etc*. Ces synchronisations permettent de sérialiser les accès à la mémoire commune. L'utilisation de ces primitives peut engendrer des situations d'interblocage (*dead-lock*) : tout comme pour les accès à la mémoire, c'est à l'utilisateur de garantir la correction de son programme. Il existe cependant des outils [96, 90] permettant de détecter ces situations de concurrence d'accès à la mémoire.

Aucune[10] sémantique des accès aux données n'est donc associée aux bibliothèques de processus légers. Le flot de données entre les processus légers dépend donc entièrement de l'exécution et ne peut être prédit.

2.3.3 Modèle de coût et ordonnancement

Chaque processus léger est ordonnancé suivant la politique qui lui a été associée lors de sa création. Cette politique peut être modifiée en cours d'exécution. Le standard POSIX [69] n'impose aucun ordonnancement particulier mais en suggère trois :

- Un ordonnancement de type FIFO : une liste de processus légers prêts à être exécutés est maintenue et les processus légers sont exécutés dans l'ordre de leur insertion dans cette liste.
- Un ordonnancement de type tourniquet (*Round-Robin*) : les processus légers sont ordonnancés cycliquement avec une certaine tranche de temps à leur disposition.
- Une autre stratégie d'ordonnancement laissée libre à l'implantation.

Par exemple, l'implantation de la bibliothèque fournie avec le système Solaris 2.5 offre soit les deux premières, soit un ordonnancement de type tourniquet avec priorités dynamique (l'ordonnancement classique des processus UNIX). L'implantation fournie avec le système AIX 4.2 n'offre que la politique de type UNIX. Des ordonnancements offrant un système de priorité sont disponibles avec l'implantation des processus légers DCE (processus légers implantant le *draft-4* de la norme POSIX) sur AIX 4.3.

Il n'y a pas de modèle de coût associé à ces bibliothèques car, ni la durée d'exécution, ni la consommation mémoire, ne peuvent être prédites. En effet, des synchronisations pouvant en règle générale intervenir entre n'importe quels processus légers, les politiques d'ordonnancement sont obligées de les exécuter tous en concurrence. Certaines implantations, comme par exemple celle proposée dans [84], utilisent des techniques d'ordonnancement plus évoluées permettant de palier ce problème dans un cadre restreint de parallélisme de type série-parallèle (donc en restreignant les synchronisations possibles entre les processus légers, ce qui est également effectué d'une manière encore plus restrictive, dans le modèle de programmation BSP présenté dans la section 2.5 page 33).

[10] Hormis celle de l'architecture sous-jacente (machine Tera [2] par exemple). Cette sémantique se résume cependant à des notions d'instructions « atomiques » permettant de gérer seulement certaines situations de concurrence.

2.3.4 Bilan et implantation

Les langages basés sur des processus légers permettent de programmer exclusivement[11] les machines à mémoire partagée de manière efficace en laissant une liberté quasi totale à l'utilisateur. Les bibliothèques de processus légers permettent d'exprimer le parallélisme de tâches d'une application avec une grande finesse. Le surcoût introduit par les fonctions de la bibliothèque est en général faible[12], ce qui permet une programmation efficace, dans le cas où le nombre de processus légers créés reste raisonable et le coût d'accès à la mémoire partagée uniforme.

La grande liberté d'expression[13] se paye par une gestion de l'utilisation de la mémoire qui doit être effectuée par l'utilisateur afin d'éviter les situations de concurrence d'accès et les interblocages entre processus légers. Le déboguage, le test et la certification peuvent également devenir très coûteux. Aucune garantie d'exécution ne peut donc être fournie dans ce cadre général.

Une implantation classique est celle des Pthreads qui suit la norme POSIX [69]. Cette implantation, du monde UNIX, consiste à définir plusieurs flots d'exécution au sein d'un même processus. Ces flots ne possèdent de manière privée que leur pile et leurs registres et partagent le tas, les fichiers et le code à exécuter avec leurs semblables, ce qui permet d'avoir des commutations de contexte très rapides[14]. De nombreuses autres implantations sont disponibles, comme par exemple la classe `Thread` dans le langage Java [85].

La grande liberté d'expression permet une programmation efficace d'une très grande variété d'application allant du calcul scientifique (par exemple l'implantation *SunPerf* de la librairie BLAS [38]) aux applications combinatoires de type *Branch and Bound* [34].

2.4 Jade

Le code Jade[15] est un code C séquentiel classique annoté par des instructions Jade permettant de définir les données partagées accédées par les différents blocs d'instructions du code source. Il a été développé au début des années 1990 à l'université de Stanford (le

[11] Des bibliothèques permettant une programmation distribuée à l'aide de processus légers existent [81, 21, 61, 20] mais des références directes aux processeurs sont faites, une des caractéristiques des langages « haut niveau » est donc perdue. Ces langages sont essentiellement basés sur un mariage d'une bibliothèque de processus légers et d'une bibliothèque de communication.

[12] Les performances dépendent évidemment de l'implantation considérée. Une des principales sources de différence est le niveau auquel se situe la bibliothèque : niveau noyau (*kernel-threads* [47]) ou niveau utilisateur (*user-threads*) [76]. Il y a désormais des supports directement matériels de la notion de processus léger, ce qui augmente encore leur efficacité : par exemple la machine Tera [2].

[13] On peut dire que les bibliothèques de processus légers sont à la programmation parallèle ce que les langages assembleurs sont à la programmation structurée.

[14] De l'ordre de la dizaine de micro-secondes, soit classiquement un rapport de 100 comparé aux processus « lourds ». Cette durée de commutation est de l'ordre de la nano-seconde pour machines intégrant un mécanisme de processus légers au niveau matériel, comme la machine Tera [2] par exemple.

[15] L'*URL* officielle du projet est `http://suif.stanford.edu/index.html`. Une information complémentaire peut être trouvée sur la page personnelle d'un des auteurs principaux à l'*URL* suivante : `http://www.cag.lcs.mit.edu/~rinard/jade/`.

projet est terminé, la dernière version date de 1994) et est basé sur une analyse du flot de données d'une application lui permettant d'extraire implicitement du parallélisme.

2.4.1 Génération du parallélisme

Jade définit une mémoire partagée à partir de laquelle sont définies les tâches et la sémantique des accès aux données. Les déclarations des données en mémoire partagée sont effectuées en ajoutant le mot clé `shared` lors de la déclaration de l'objet. Cette spécification permet au langage de construire les fonctions nécessaires à leurs communications dans le cadre d'un machine distribuée. L'utilisation de ces données est identique à celles des données de types standards.

La description du parallélisme se fait au niveau d'un bloc d'instructions en spécifiant les accès, lecture et/ou écriture, qui seront effectués sur les variables en mémoire partagée lors de l'exécution de ce bloc à l'aide des instructions `withonly {...}` ou `with {...} cont`, comme illustré figure 2.3(a). Lors de l'exécution, ces blocs d'instructions sont implicitement transformés en tâches par Jade. Les contraintes de précédence seront déduites de la déclaration des accès effectués par la tâche.

Les accès possibles sur une donnée en mémoire partagée sont construits à partir des accès de base suivants : la lecture (`rd`), l'écriture (`wr`), l'accès commutatif (`cm`)[16] et la destruction (`de`)[17]. D'éventuelles synchronisations entre les tâches seront automatiquement insérées si nécessaire lors de l'exécution afin de garantir la sémantique des accès aux données, comme spécifié dans la section suivante.

2.4.2 Flot de données et sémantique

La sémantique des accès aux données est semblable à la sémantique séquentielle : les données lues sont identiques dans une exécution parallèle et dans une exécution séquentielle[18]. Pour garantir la sémantique des synchronisations sont insérées entre certaines tâches. Ces tâches sont déterminées en analysant le graphe de flot de données qui est construit dynamiquement à partir des déclarations d'accès effectués par les blocs d'instructions, comme illustré figure 2.3(b). Si deux tâches n'accèdent aucune donnée en commun, alors ces deux tâches n'ont aucune contrainte de précédence et peuvent être exécutées dans un ordre quelconque l'une par rapport à l'autre et donc, en particulier, en parallèle. Si, par contre, elles accèdent en commun une même donnée et si l'un des accès est une écriture ou une libération, alors il y a contrainte de précédence entre les tâches : elles doivent s'exécuter séquentiellement et une synchronisation doit être insérée entre ces deux tâches. L'implantation du langage garantit que la tâche qui se serait exécutée en premier lors d'une exécution séquentielle s'exécutera en premier lors de l'exécution parallèle. Cette stratégie d'exécution conserve l'ordre relatif des écritures et des lectures

[16] Cet accès autorise la modification en « concurrence » de la donnée par plusieurs tâches : les tâches sont exécutées en série mais l'ordre au sein de cette série n'est pas défini et peut varier d'une exécution à l'autre.

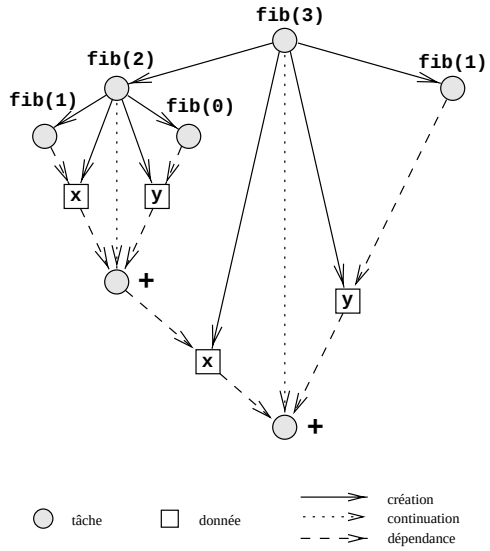
[17] Si la donnée associée à l'objet a été dynamiquement créé à l'aide de `create_object`.

[18] Cette exécution séquentielle correspond à l'exécution du code source dans lequel toutes les annotations Jade sont ignorées.

```
int fib( int n )
{
   if( n<2 )
      return n;
   else {
      int shared x, shared y;
      withonly { wr(x) } do( n, x ) {
         x= fib( n-1 );
      }
      withonly { wr(y) } do( n, y ) {
         y= fib( n-2 );
      }
      with { rd( x ), rd( y ) } cont;
      return( x+y );
   }
}

int main( int argc, char** argv )
{
   int n, r;
   n= atoi( argv[1] );
   r= fib( n );
   printf( "fib(%d)=%d\n", n, r );
   return 0;
}
```

(a)



(b)

Figure 2.3 *Calcul du n-ième nombre de Fibonacci en Jade.*
En (a), dans le code Jade, l'expression du parallélisme est effectuée implicitement en déclarant les accès effectués par certaines parties du programme. L'implantation du langage crée les tâches correspondantes à ces parties et les contraintes de précédence seront déduites de l'analyse des accès annoncés. En (b), le graphe d'exécution d'un appel à `fib(3)` *est représenté. Bien que cela soit le cas dans cet exemple, le graphe n'est pas forcément de type série-parallèle.*

concernant chaque donnée partagée : la sémantique séquentielle des accès aux données est donc conservée puisque les valeurs coïncident.

De plus, Jade contrôle dynamiquement (à l'aide d'une table) tous les accès effectués sur la mémoire partagée afin d'interdire tout effet de bord : une tâche ne peut accéder qu'aux données dont elle a déclaré l'accès.

2.4.3 Modèle de coût et ordonnancement

L'ordonnancement des tâches est effectué dynamiquement par un ordonnanceur centralisé reposant sur un algorithme de liste prenant en compte la localité[19] des objets accédés par une tâche. Cette politique est optimisée pour le cas où la tâche principale (`main`) crée l'intégralité du graphe de flot de données.

Le principe de cet ordonnancement est de maintenir une liste de tâches prêtes, liste triée par rapport aux données accédées : c'est-à-dire que chaque processeur possède une liste d'objets mémoire, et à chacun de ces objets est associée une liste de tâches y accédant. Lorsqu'un processeur termine l'exécution d'une tâche, il choisit la tâche suivante

[19]Chaque objet possède un site (ou processeur) de référence sur lequel les écritures sont effectuées. Ce site, qui constitue la localité de l'objet, peut être modifié en cas de vol de l'objet par un processeur inactif. Les lectures réparties peuvent entraîner des copies de cet objet mais n'en modifie pas la localité.

dans la liste associée à l'objet courant afin d'essayer de tenir compte de la localité. Si cette tâche n'est pas prête, ou si la liste est vide, le processeur passe à l'objet suivant (qui devient alors l'objet courant). Si aucun objet n'est disponible, il vole sur un processeur, choisi au hasard, un objet et sa liste de tâche associée (cette liste devant contenir au moins une tâche prête). L'ordre des objets dans la liste est sans importance.

Aucun modèle de coût n'est associé au modèle de programmation Jade. Cependant, la connaissance du flot de données, et donc d'une description des synchronisations entre les tâches, laisse à penser qu'une telle étude théorique doit être possible sur ce langage. Les résultats devraient, *a priori*, être similaires à ceux obtenus par Cilk, NESL ou Athapascan-1. Il faut cependant s'assurer que le comportement de la politique d'ordonnancement permet d'estimer, dans tous les cas, l'ordre d'exécution des tâches.

2.4.4 Bilan et implantation

Jade permet une expression entièrement implicite du parallélisme d'une application et offre un moyen simple de programmer les machines à mémoire distribuée en offrant une mémoire virtuelle partagée. La sémantique associée à l'utilisation de cette mémoire partagée garantit que les valeurs retournées lors des accès à cette mémoire seront identiques à celles retournées lors d'une exécution séquentielle du programme. L'utilisateur se contente de définir le type des accès effectués par les différentes parties de son programme, l'implantation de Jade se chargeant de créer les tâches, de gérer leurs contraintes de précédence et de migrer les objets (tâches ou données) entre les processeurs. Cependant les performances pour certaines applications sont mauvaises (par exemple pour une factorisation de Cholesky [93]), ce qui peut être expliqué soit par un grain trop faible de l'application (c'est alors le surcoût de Jade qui dicte les performances) soit par l'inadéquation de la stratégie d'ordonnancement au problème (sérialisation des tâches accédant de manière concurrente la même donnée, mauvais placement introduisant des problèmes de localité).

Le parallélisme exprimé est un parallélisme de contrôle, et l'utilisateur choisit le grain des tâches et des données partagées. Compte tenu des choix effectués lors de l'implantation actuelle où la plupart des algorithmes mis en œuvre sont centralisés, Jade est plutôt destiné aux applications de gros grain.

Jade a été conçu pour être portable et utilisable sur une grande variété de machines. Deux versions sont disponibles, l'une pour les machines symétriques à mémoire partagée et l'autre pour les machines à processeurs faiblement couplés. Ces deux versions se distinguent essentiellement par la technique d'ordonnancement mise en œuvre, les contraintes de localités étant différentes.

2.5 Le modèle de programmation BSP

Le modèle de programmation BSP[20] [103, 98] a été proposé par Valiant en 1990, université d'Harvard, et n'est autre qu'une extension du modèle XPRAM présenté dans [104]. Le but de ce modèle est de constituer un intermédiaire entre les architectures parallèles et les applications.

2.5.1 Génération du parallélisme

Dans le modèle BSP le parallélisme est de type *SPMD*[21], et une application est décrite comme l'exécution sur p processeurs du même code applicatif sur des données différentes. Ces p processeurs sont interconnectés par un réseau et chacun peut envoyer des messages (qui peuvent également représenter des écritures à distance), à tout moment, aux autres. Le mode de réception de ces messages est particulier : chaque tâche est découpée en un certain nombre de *super-pas* s_i, deux super-pas étant séparés par une barrière de synchronisation, et tout message émis au cours du super-pas s_i ne sera reçu par le processeur destinataire qu'au début de l'exécution du super-pas s_{i+1}. Le modèle d'exécution est donc une suite d'étapes de calcul, puis communication, puis calcul, puis communication, *etc*. Ces étapes sont illustrées dans la figure 2.4 page 33. Il est à noter que ces synchronisations sont purement conceptuelles, c'est-à-dire que l'implantation peut décider ou non de les réaliser. La seule chose imposée par le modèle de programmation est que les messages émis lors d'un super-pas seront reçus avant le début du super-pas suivant.

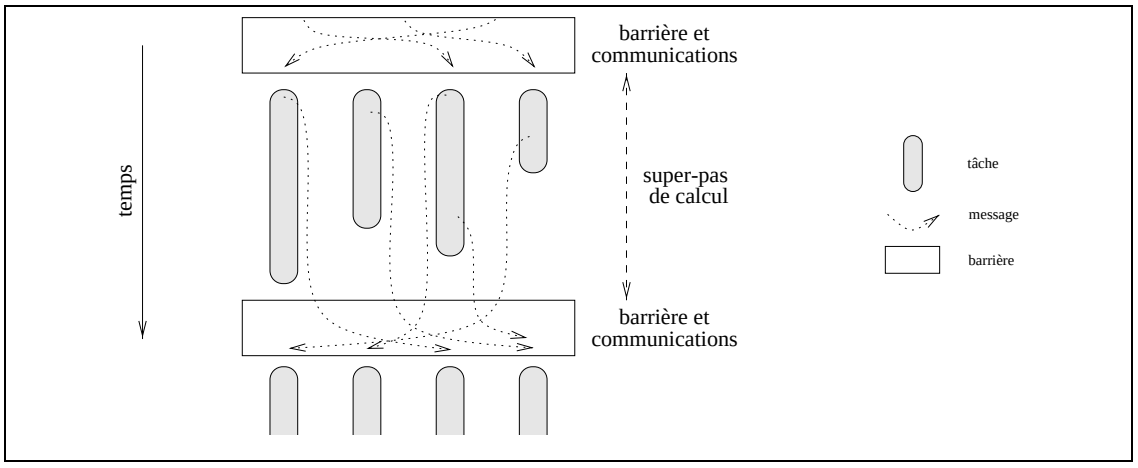


Figure 2.4 *Modèle de programmation BSP.*
L'exécution est découpée en super-pas *séparés par une phase de synchronisation globale durant laquelle toutes les communications de l'étape de calcul précédente sont effectuées. Lors de l'exécution d'une tâche, tous les accès sont locaux car toutes les données ont été amenées lors de la précédente phase de communication.*

Les applications dont l'expression est aisée dans ce modèle sont celles basées sur

[20] BSP est l'acronyme de *Bulk-Synchronous Parallel*. La page officielle de ce projet est maintenue à l'*URL* suivante : `http://www.deas.harvard.edu/cs/research/bsp/bsp.html`.

[21] *« Single Program Multiple Data »*.

un calcul itératif, telles les décompositions de domaine ou les résolutions de systèmes linéaires. Nous donnons cependant une implantation possible du calcul récursif du n-ième nombre de Fibonacci en BSP. Nous utilisons pour cela 2^n processeurs et découpons le calcul de $F(n)$ en n super-pas. Nous nous inspirons pour ce calcul du stockage d'un arbre binaire classiquement effectué lors d'un tri par tas [53] : les deux fils d'un nœud stocké à la place i dans un tableau sont stockés respectivement aux emplacements $2i$ et $2i+1$. Ainsi, appliqué à notre calcul, si le processeur p est responsable du calcul de $F(k)$, alors $F(k-1)$ et $F(k-2)$ seront respectivement calculés[22] par les processeurs $2p$ et $2p+1$. Le but de ce placement est de pouvoir déterminer, à partir du seul numéro de processeur, quand et quelle valeur de la suite de Fibonacci doit être calculée localement.

Soit p un des processeurs impliqués dans le calcul (numérotés de 1 à 2^n) et c le codage en binaire de son identificateur. Nous pouvons alors montrer les deux propositions suivantes :

1. Le processeur p interviendra au super-pas $n - i_p$ avec i_p l'indice du bit de poids fort de c (c'est-à-dire $2^{i_p} \leq p < 2^{i_p+1}$).
2. Le processeur p doit calculer $F(k)$ avec $k = n - i_p - j_p$ avec j_p la somme des bits de poids faibles de c.

Le tableau 2.1 page 34 permet de s'en convaincre (la preuve par récurrence de ces deux propositions est immédiate).

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	...
1	**1**0	**1**1	**1**00	**1**01	**1**10	**1**11	**1**000	**1**001	**1**010	**1**011	**1**100	**1**101	**1**110	**1**111	...
															...
							$n-3$	$n-4$	$n-4$	$n-5$	$n-4$	$n-5$	$n-5$	$n-6$	
			$n-2$	$n-3$	$n-3$	$n-4$									
	$n-1$	$n-2$													
n															

Tableau 2.1 *Placement du calcul de Fibonacci en BSP.*
Il est possible de déterminer pour chaque processeur p_i, à partir des deux propositions présentées page 34, le super-pas où un calcul doit être mené et quel nombre de Fibonacci doit être précisément calculé lors de ce super-pas. Cette détermination se base sur le codage en binaire de l'identificateur du processeur : l'indice du bit de poids fort permet de calculer le super-pas dans lequel un calcul doit être mené et la somme des indices de poids faibles indique quelle valeur doit être effectivement calculée.

La figure 2.5 page 35 représente le code BSP implantant cet algorithme (la syntaxe utilisée est celle proposée dans BSPLib [67]). La version présentée suppose un nombre infini de processeurs, ce qui n'est pas incohérent avec le modèle BSP. C'est à l'implantation du modèle, donc à BSPLib dans notre cas, d'effectuer le repliage et l'ordonnancement du calcul sur les processeurs physiques disponibles.

2.5.2 Flot de données et sémantique

Lors d'un super-pas de calcul chaque tâche ne fait que des accès locaux. De plus, les messages émis (ou les écritures à distance) durant un super-pas de calcul n'arriveront que

[22] Si la découpe est nécessaire, c'est-à-dire si $k \geq 2$.

```
int fib( n )
{
   int p= bsp_pid();
   int ip= log( p ); // indice du poids fort de p
   int jp= ...;      // somme des bits de poids faibles de p
   int k= n-ip-jp;   // valeur de Fibonacci à calculer localement
   int x,y,r;
   for( int s=1; s<=n; s++ ) { // super-pas
      if( p>0 ) {
         if( n-ip==s )
            if( k==0 || k==1 )
               bsp_send( p/2, &k );
            else if( k>1 ) {
               r= x+y;
               bsp_send( p/2, &r );
            }
         else if( ( n-ip==s+1 ) && k>1 ) {
            bsp_receive( 2*p, &x );
            bsp_receive( 2*p+1, &y );
         }
      }
      bsp_sync();
   }
   if( p==0 )
      bsp_receive( 1, &r );
   bsp_sync();
   return r;
}

void main( int argc, char** argv )
{
   int n, r;
   n= atoi( argv[1] );
   begin_bsp( pow( 2, n ) );
   r= fib( n );
   if( bsp_pid()== 0 ) // récupération du résultat sur p0
      printf( "fib(%d)=%d\n", n, r );
   end_bsp();
}
```

Figure 2.5 *Calcul du n-ième nombre de Fibonacci en BSP.*
Le résultat est reçu sur le processeur 0 au bout de $n+1$ super-pas. Les 2^n processeurs impliqués dans le calcul sont numérotés de 1 à 2^n. Les demandes de réception de messages postées lors d'un super-pas s ne seront terminées qu'au début du super-pas suivant (c'est-à-dire juste après la barrière `bsp_sync()`*). Avant ce super-pas $s+1$, tout accès à la valeur est invalide. Les émissions suivent le même principe.*

lors de la phase de communication située au moment de la barrière : il n'y a donc pas de condition de concurrence entre les dates d'arrivée des messages (un des problèmes typique des applications développées sur un paradigme de type échange de messages, par exemple MPI) puisque tous les messages arrivent à la même date. Cependant, les écritures concurrentes à distance sont gérées de manière arbitraire et le résultat de la lecture est donc indéterministe.

Le flot de données est donc constitué uniquement par les tâches et les barrières de synchronisations globales entre chaque super-pas lors desquels les données sont échangées.

2.5.3 Modèle de coût et ordonnancement

La machine d'exécution est modélisée par les trois grandeurs suivantes, normalisée par rapport à la vitesse des processeurs (supposés identiques) :

- p, le nombre de processeurs.
- l, le coût d'une barrière de synchronisation.
- g, le temps de communication d'un mot à travers le réseau. La durée de transfert d'un message de longueur h est supposée être égale à hg.

L'instance de l'application est caractérisée par les grandeurs suivantes :

- $w_{I,j}$, le travail effectué par le processeur j lors du super-pas I.
- $h_{I,j}$, le volume de données émis par le processeur j lors du super-pas I.

La durée s_I du super-pas I et la durée d'exécution T_p sur p processeurs sont alors :

$$s_I = \max_{1 \leq j \leq p}(w_{I,j}) + g \max_{1 \leq j \leq p}(h_{I,j}) + l$$
$$T_p = \sum_I s_I$$

Aucun ordonnancement n'est *a priori* nécessaire étant donné qu'il y a autant de tâches que de processeur. Cependant, en pratique, afin de permettre une adaptation aux conditions d'exécution (et la réduction du paramètre g), il est conseillé [103] à l'utilisateur de programmer son application avec un degré de parallélisme (*parallel slackness*) suffisant, c'est-à-dire en concevant l'application pour un nombre v de processeurs virtuels, avec $v \gg p$ (par exemple $v \approx p \log p$). La manière dont ces v processeurs virtuels seront répartis parmi les p processeurs réels ne dépend absolument pas du modèle. Dans la version[23] 1.4 d'une implantation du modèle proposée par l'université d'Oxford, l'allocation est effectuée en plaçant les processeurs virtuels sur les machines les moins chargées. La version suivante devrait offrir un mécanisme de migration des processeurs virtuels afin de permettre un équilibrage automatique de la charge entre les processeurs physiques. Dans cette version, la barrière conceptuelle de synchronisation est réalisée à l'aide d'une veritable barrière de synchronisation. Dans la version Green BSP [62], l'implantation de cette barrière est effectuée par une attente, par chaque tâche, de la fin d'émission et de réception de toutes les communications annoncées au cours du super-pas. Cette barrière n'est donc pas globale entre toutes les tâches.

2.5.4 Bilan

Le modèle de programmation BSP permet de programmer une application indépendemment de la machine cible en utilisant un concept de processeur virtuel permettant d'exprimer le degré de parallélisme de l'application et un paradigme de type processus communicants. Un modèle de coût associé au modèle BSP permet de prédire, en fonction des caractéristiques de l'application et de la machine, la durée de l'exécution.

[23] *The Oxford BSP Toolset and Profiling system (v 1.4, 30th September 1998)* disponible librement à l'*URL* suivante : `http://www.bsp-worldwide.org/implmnts/oxtool/download.html`.

L'estimation de la durée d'exécution est rendue possible grâce à l'expression des synchronisations sous forme de barrière globale : les synchronisations « imprévisibles » qui rendaient toute prévision impossible dans le cas des langages à base de processus légers ou dans le cas d'une programmation directe par échange de message (de type MPI) sont donc supprimées. BSP peut être vu comme le modèle de programmation associé à un sous ensemble d'un langage de type MPI, les synchronisations et les communications étant globalement regroupées.

Diverses implantations respectant le modèle BSP sont disponibles sous forme de bibliothèque. Nous citerons une des principales, BSPLib [67], qui est disponible sur une grande variété de machines, des machines à mémoire partagée de type *SMP* aux interconnections de stations de travail.

Les applications dont l'expression est aisée dans ce modèle sont celles basées sur un calcul itératif, tels les décompositions de domaine ou les résolutions de systèmes linéaires.

2.6 NESL

NESL[24] est un langage de type fonctionnel qui permet d'exploiter un parallélisme de données de type série-parallèle. Il a été développé au tout début des années 1990 à l'université de Carnegie Mellon (le langage est toujours maintenu mais la dernière version date de novembre 1995). Une approche similaire est celle adoptée par le langage SISAL[25] [78, 48], hormis le fait que ce langage exploite un parallélisme de contrôle. Dans les deux cas un flot de données est construit afin de déterminer les précédences et les communications de données entre les tâches.

2.6.1 Génération du parallélisme

La génération du parallélisme se fait en appliquant une même fonction sur une séquence de valeurs : la fonction sera appliquée en parallèle sur chacun des éléments de la séquence, comme illustré figure 2.6(a). C'est un parallélisme de données de type SPMD. Un autre moyen de générer du parallélisme est d'utiliser des fonctions prédéfinies par le langage, fonctions parallèles qui opèrent sur une séquence dans son ensemble : par exemple le calcul de la somme, du tri ou la recherche du minimum. Tous ces appels sont synchrones, le programme ne passant à l'instruction suivante que lorsque le résultat a été entièrement calculé, c'est-à-dire lorsque toutes les fonctions parallèles générées ont été terminées ; le parallélisme généré est donc de type série-parallèle, comme illustré figure 2.6(b).

[24] La page officielle du projet dans lequel est développé le langage NESL est maintenue à l'*URL* suivante : `http://www.cs.cmu.edu/~scandal/nesl.html`.

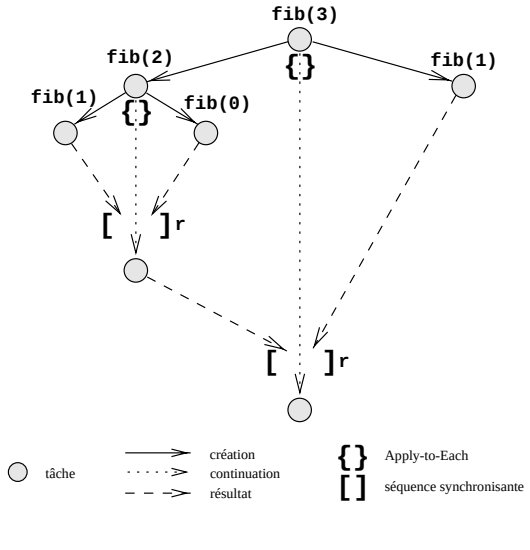
[25] SISAL est l'acronyme de *Streams and Iterations in Single Assignment Language*. Le projet, débuté au milieu des années 80, est abandonné depuis quelques années. Une copie de l'ancien serveur *ftp* est cependant accessible à partir de l'*URL* suivante : `http://www.physics.nmt.edu/raymond/raymondfrontpage.html`.

```
function fib_nesl( n )=
   if n<2 then
      n
   else
      let r={ fib_nesl(x) :
                  x in [n-1, n-2] }
      in elt( r, 0 ) + elt( r, 1 );

function main( n )=
   let r= fib_nesl( n ) in
      print_string( "fib(" ++ @n ++
                    ")=" ++ @r );


function fib_sisal( n )
   if n<2 then
      n
   else
      let r= for i in n-1, n-2
                returns array of
                   fib_sisal( i )
             end for
      in r[1] + r[2];
end function
```

(a)



(b)

Figure 2.6 *Calcul du n-ième nombre de Fibonacci en NESL et SISAL.*
En (a), dans le code NESL, l'expression du parallélisme se fait en appliquant une fonction sur une séquence à l'aide de la construction Apply-to-Each `{` $f(x)$ `:` x `in` seq `}`. *Le résultat de cet opérateur est une nouvelle séquence obtenue en appliquant la fonction* `fib` *à chacun des éléments de la séquence initiale. La fonction* `elt()` *retourne un élément d'une séquence et l'opérateur* `@` *tranforme un nombre en une chaîne de caractères. Dans le code Sisal, la seule différence est l'utilisation d'un parallélisme de tâches au lieu du parallélisme de données. L'itération* for *implique une exécution parallèle du calcul correspondant à chaque pas de la boucle. En (b), le graphe d'exécution d'un appel à* `fib(3)` *est représenté. Ce graphe est forcément de type série-parallèle, les tâches mères étant synchronisées sur les séquences produites par les tâches filles.*

2.6.2 Flot de données et sémantique

La sémantique des accès aux données lors d'une exécution parallèle est identique à celle de l'exécution séquentielle puisqu'il n'y a aucun partage possible des données entre deux tâches concurrentes, chaque tâche opérant sur un élément particulier de la séquence et produisant un élément particulier d'une autre séquence (c'est un modèle fonctionnel). Le flot de données est de type série-parallèle car la tâche qui crée du parallélisme à l'aide d'une séquence est bloquée jusqu'à la terminaison du calcul de tous les éléments la séquence.

Les seuls opérateurs qui permettent un accès concurrent sont les opérateurs *read* et *store*. Ces deux opérateurs prennent deux séquences s_e et s_i en paramètres et retournent une séquence s_s[26]. Dans le cas de la lecture, s_i est une séquence d'entiers et le résultat s_s correspond à la séquence des éléments de s_e d'indice les éléments de s_i, donc $|s_s| = |s_i|$. Dans le cas de l'écriture, s_i est une séquence de couples représentant un indice et une valeur à écrire à cet indice, donc $|s_s| = |s_e|$ par mutation de s_e. Il y a risque de concurrence si

[26] L'utilisation typique de ces opérateurs est la suivante : s_s `=` s_e `->` s_i pour la lecture, ou s_s `=` s_e `<-` s_i pour l'écriture.

un même indice se retrouve plusieurs fois dans e_i et donc un risque d'indéterminisme sur la valeur de la séquence résultat. Si la concurrence en lecture ne pose pas de problème[27] de sémantique, celle en écriture peut engendrer de l'indéterminisme si l'ordre des écritures n'est pas fixé. Afin d'éviter ce problème, NESL ne génère pas de parallélisme sur cette instruction *store* et impose l'ordre des écritures s_i de gauche à droite.

2.6.3 Modèle de coût et ordonnancement

Les tâches créées lors de l'exécution sont insérées dans un graphe de flot de données comme illustré figure 2.6(b) [83]. Un modèle de coût est associé au modèle de programmation, chaque programme est caractérisé par les grandeurs suivantes :

- T_1, le travail, qui correspond au nombre d'instructions exécutées, identique au nombre de nœuds du graphe et à la durée sur un processeur : les tâches sont considérées de durée unitaire et peuvent allouer au plus une quantité bornée de mémoire. Pour l'exemple de Fibonacci, $T_1 \approx 2^n$.
- T_∞, la longueur du chemin critique du graphe, correspondant à la durée de l'exécution sur un nombre infini de processeurs. Pour l'exemple de Fibonacci, $T_\infty \approx n$. La formule récursive suivante permet de calculer cette longueur :

$$T_\infty(t) = T_1(t) + \sum_{s \text{ séquence de } t} \max_{e \in s} T_\infty(e)$$

- S_1, l'espace mémoire requis par une exécution séquentielle, cette exécution correspondant à un parcours du graphe en profondeur d'abord (en parcourant les tâches filles selon leur ordre de création, c'est-à-dire de gauche à droite dans les représentations classiques du graphe). Pour l'exemple de Fibonacci, $S_1 \approx n$.

Les processeurs sont associés aux nœuds de ce graphe dynamiquement et par étapes : régulièrement un nombre fixé ($p \log p$, afin de permettre le recouvrement du coût d'ordonnancement) de tâches prêtes sont affectées aux p processeurs de la machine. Le nombre d'étapes de cet algorithme d'ordonnancement peut être majoré ce qui permet de garantir les résultats suivants pour la durée d'exécution T_p et la consommation mémoire S_p de toute exécution sur une machine à p processeurs [11] :

$$\begin{aligned} T_p &\leq O\left(\frac{T_1}{p} + T_\infty \log p\right) \\ S_p &\leq O(S_1 + T_\infty p \log p) \end{aligned}$$

L'implantation de cet ordonnancement est basé sur un algorithme probabiliste, les performances en temps sont donc obtenues avec une forte probabilité. Pour des programmes générant suffisamment de parallélisme, c'est-à-dire pour lesquels $\frac{T_1}{p} \gg T_\infty$ et $S_1 \gg T_\infty$, les performances obtenues sont à un facteur $1 + o(1)$ des optimales. Le nombre de tâches créées est limité par une technique de création paresseuse [80, 12] ce qui permet d'exprimer des tâches de durée très petite au niveau du code source de l'application sans que cela soit pénalisant pour les performances à l'exécution.

[27] Mis à part d'éventuels problèmes de coût d'implantation et de recopie sur les machines ne tolérant pas les accès concurrents à la mémoire.

2.6.4 Bilan et implantation

Le langage NESL permet d'exploiter aisément le parallélisme de données d'une application par une technique de type SPMD. Le parallélisme exploité est de type série-parallèle, et la sémantique associée à l'accès aux données est identique à la sémantique séquentielle. L'ordonnancement utilisé est basé sur le parcours du graphe des tâches de l'application, graphe qui est construit dynamiquement. Basé sur un algorithme probabiliste, il permet de garantir, asymptotiquement, de bonnes performances en temps et en mémoire avec une grande probabilité.

Des implantations de NESL existent pour une variété de machines à mémoire distribuée et également pour les machines de type *SMP* (SGI Power Challenge et DEC AlphaServer par exemple). Une version reposant sur MPI est disponible, ce qui permet d'utiliser n'importe quelle machine possédant cette librairie de communication. Selon les auteurs, NESL est particulièrement bien adapté à l'enseignement et à l'expérimentation des algorithmes parallèles.

2.7 Cilk

Cilk[28] [70, 14] est un langage destiné à la programmation des machines parallèles à mémoire partagée dont le développement à débuté en 1993 au MIT (le projet est toujours en cours de développement et la prochaine version, Cilk-6, est attendue pour janvier 2000). C'est une extension du langage C qui offre des primitives pour l'expression du parallélisme de contrôle par création explicite de tâches. Un modèle de coût, permettant de garantir les efficacités en temps et en consommation mémoire, est associé au modèle de programmation.

2.7.1 Génération du parallélisme

La description du parallélisme se fait à l'aide du mot clé `spawn` placé devant un appel de fonction, comme illustré figure 2.7(a). Conceptuellement, lors de l'exécution du programme, une tâche sera créée pour évaluer cette fonction. La sémantique de cet appel diffère de celle de l'appel classique d'une fonction au sens où la procédure appelante peut continuer son exécution en parallèle de l'évaluation de la fonction appelée au lieu d'attendre son retour pour continuer. Cette exécution étant asynchrone, la procédure créatrice ne peut pas utiliser le résultat de la fonction appelée sans synchronisation. Cette synchronisation est explicite par utilisation de l'instruction `sync`. Cette instruction a pour effet d'attendre la terminaison de toutes les fonctions appelées en parallèle par la fonction mère avant ce `sync` : le parallélisme exprimé est donc de type série-parallèle (en tenant compte du fait que la tâche mère s'exécute en concurrence avec ses filles, jusqu'à rencontrer l'instruction `sync`), comme illustré figure 2.7(b). Les tâches sœurs créées sont supposées indépendantes : il y a donc risque de concurrence sur les accès à la mémoire partagée, concurrence qui doit être gérée par l'utilisateur. On ne considère dans la suite

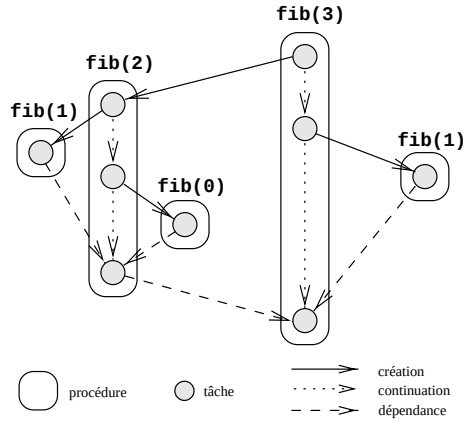
[28] La page officielle du projet est maintenue à l'*URL* suivante : `http://supertech.lcs.mit.edu/cilk/`.

que des programmes **corrects**, c'est-à-dire des programmes pour lesquels tous les conflits d'accès à la mémoire partagée ont été résolus (au moyen de l'instruction `sync` ou de verrous).

```
cilk int fib( int n )
{
   if( n<2 )
      return n;
   else {
      int x, y;
      x= spawn fib( n-1 );
      y= spawn fib( n-2 );
      sync;
      return( x+y );
   }
}

cilk int main( int argc, char** argv )
{
   int n, r;
   n= atoi( argv[1] );
   r= spawn fib( n );
   sync;
   printf( "fib(%d)=%d\n", n, r );
   return 0;
}
```

(a)



(b)

Figure 2.7 *Calcul du n-ième nombre de Fibonacci en Cilk.*
En (a), dans le code Cilk, l'expression du parallélisme se fait en créant une tâche (`spawn`*) à la place d'un classique appel de fonction. L'instruction* `sync` *permet d'attendre la terminaison de toutes les tâches créés par la procédure, ici* `fib(n-1)` *et* `fib(n-2)`*. En (b), le graphe d'exécution d'un appel à* `fib(3)` *est représenté. Ce graphe est forcément de type série-parallèle du fait de la synchronisation globale sur la fratrie effectuée par l'instruction* `sync`.

En plus de cette génération de parallélisme, le langage Cilk permet d'associer lors de la création d'une tâche une fonction « réflexe » ou *callback* qui sera exécutée en exclusion mutuelle lors de la terminaison de cette tâche. Il est possible à l'intérieur de cette fonction de demander la destruction de l'ensemble des tâches créées par la procédure mère et non encore exécutées, possibilité communément utilisée dans les algorithmes parallèles de recherche spéculative.

2.7.2 Flot de données et sémantique

Il est possible, en supprimant tous les mots clés spécifiques du langage, de transformer tout code Cilk en un code C standard ; ce code est nommé la *C-élision* du code initial. Ce code sert de référence dans la définition de la sémantique et dans le modèle de coût. La valeur retournée par chaque accès lors d'une exécution parallèle est identique à celle retournée lors du même accès dans la *C-élision* correspondante : le programme parallèle et la *C-élision* séquentielle voient donc les mêmes valeurs pour chaque accès (pour les programmes dits corrects). La sémantique associée aux accès aux données s'inspire donc directement de la sémantique séquentielle du langage C. C'est cependant à l'utilisateur

de garantir qu'aucune contrainte de précédence n'apparait dans son application[29]. Ceci est fait à l'aide de l'instruction `sync`, du mécanisme des fonctions réflexes ou à l'aide de variables d'exclusion mutuelle (*lock/unlock*). La sémantique des accès aux données n'est bien sûr respectée que si les fonctions réflexes sont commutatives et associatives, de même que les actions effectuées lors des exclusions mutuelles.

Avoir une sémantique des accès aux données semblable à celle du langage C est rendu possible en transformant la pile classique contenant les variables locales d'une procédure en une pile *cactus-stack* [99] permettant le partage et la séparation des données possédées par les tâches, comme illustré figure 2.8 page 42. Ainsi, toutes les tâches qui sont créées dans la portée d'une variable peuvent partager la donnée qui y est associée (en respectant cependant les règles de portée standards, c'est-à-dire que la tâche créée doit posséder un moyen d'accéder cette donnée : un pointeur, une référence, ...). Ce partage est un partage physique (aucune copie de donnée n'est effectuée), la cohérence de cette valeur repose donc sur la gestion de la mémoire partagée opérée par le système.

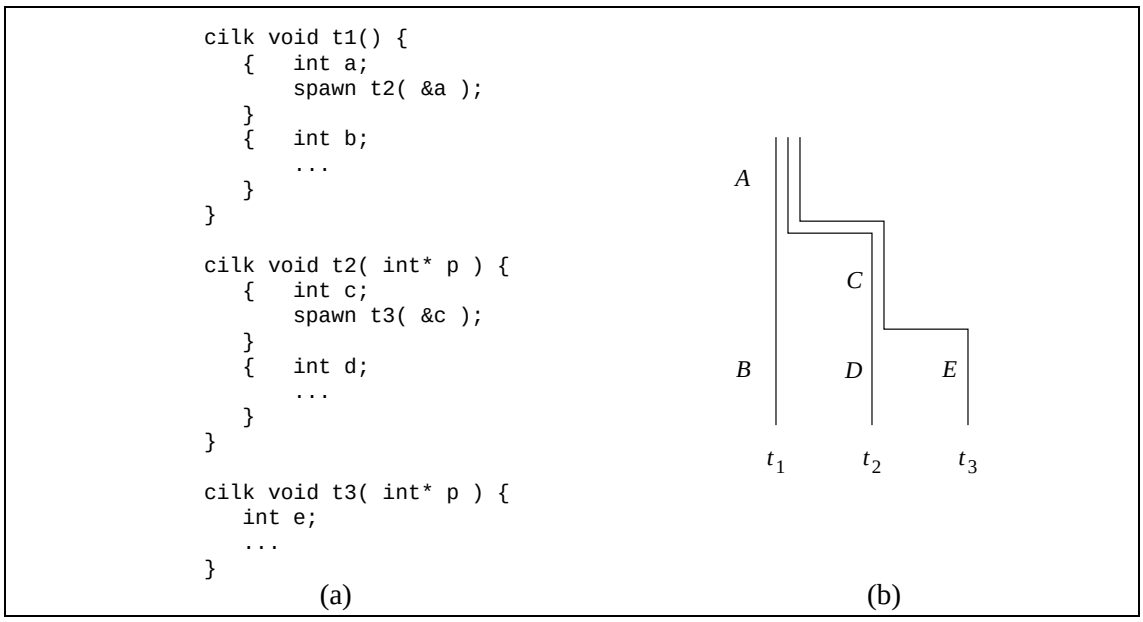


Figure 2.8 *La pile* cactus-stack *de Cilk.*
La classique pile contenant les données locales à un appel de fonction est transformée en une pile cactus-stack *permettant le partage et la séparation des données. Ici, dans le code Cilk donné en (a), la tâche t_1 crée t_2 qui crée t_3. Chaque tâche voit ses données plus celles déclarées par ses ancêtres avant sa création (en respectant les règles de visibilité standards). Ainsi les segments A et C sont partagés tandis que B, D et E sont privés aux tâches les ayant déclarés, comme illustré en (b).*

[29] Ce qui revient à dire que les constructions du langage ne garantissent rien : l'utilisateur est libre d'accèder les données partagées quand bon lui semble. La seule sémantique associée aux données partagées est celle de la *DAG-consistency* [13, 52] garantie à l'aide des piles *cactus-stack*. Cependant, bien que l'utilisateur puisse effectuer les accès qu'il désire, la philosophie du langage est plutôt basée sur un respect par l'utilisateur de la sémantique séquentielle. C'est pour cette raison que nous considérons que Cilk est basé sur cette sémantique.

2.7.3 Modèle de coût et ordonnancement

Au modèle de programmation Cilk est associé un modèle de coût [13]. Comme précédemment pour le langage NESL, chaque programme est caractérisé par les trois grandeurs suivantes :

- T_1, son travail, qui est le nombre total d'instructions. Ceci correspond à la durée de l'exécution sur un seul processeur.
- T_∞, la longueur de son chemin critique, qui est le nombre d'instructions dans un plus long chemin du graphe d'exécution. Ceci correspond à la durée d'exécution sur un nombre infini de processeurs. Il est à noter dans le calcul de cette valeur que, contrairement aux langages NESL et Athapascan-1, la tâche mère peut s'exécuter en concurrence avec les tâches filles[30].
- S_1, la consommation mémoire de l'exécution sur un seul processeur lors d'une exécution en profondeur du graphe de tâches.

Les tâches définies par l'utilisateur sont ordonnancées dynamiquement sur les processeurs par un algorithme de type liste, appelé *work-stealing*, qui fonctionne par vol de travail : lorsqu'un processeur devient inactif, il tire au sort un autre processeur, la victime, à qui il va voler une tâche prête à être exécutée. L'implantation de cet ordonnanceur garantit que la durée d'exécution T_p d'un programme (qui n'utilise pas de variables d'exclusion mutuelle) sur une machine à p processeurs et que la consommation mémoire S_p seront telles que :

$$\begin{aligned} T_p &= \frac{T_1}{p} + O(T_\infty) \\ S_p &\leq pS_1 \end{aligned}$$

Le coût de l'ordonnancement est de l'ordre du nombre de requètes de vol [16], donc de l'ordre de $O(T_\infty)$ qui est considéré faible devant $\frac{T_1}{p}$. Une technique de compilation fine [51] permet de placer tout le surcoût d'ordonnancement lors des vols : le coût de la création de tâche (`spawn`) est donc réduit au maximum[31]. Cette technique est largement utilisée dans le domaine de la compilation de langages fonctionnels, comme par exemple Multilisp [65].

2.7.4 Bilan et implantation

Le langage Cilk offre de manière simple à l'utilisateur le moyen d'exprimer le parallélisme de contrôle de son application. La sémantique associée au langage est identique à celle de la version séquentielle du code. L'utilisateur doit cependant garantir explicitement à l'aide de synchronisations simples qu'aucune contrainte de concurrence n'a lieu

[30] Le calcul de T_∞ doit donc être effectué directement sur le graphe de l'exécution et il ne peut pas être donné de formule récurrente simple en fonction des tâches filles créées. Cette formule ne contiendrait de toute façon pas de terme de type $T_1(t)$, contrairement aux langages tels que NESL et Athapascan-1.

[31] Le coût de l'instruction `spawn` est typiquement de l'ordre de 2 à 6 fois le coût de l'appel d'une fonction [51].

entre les accès aux données. L'implantation de l'ordonnanceur en ligne permet de garantir l'efficacité de l'exécution. Le parallélisme de grain fin peut être exploité grâce à une technique de création paresseuse des tâches [80, 51] ce qui permet de limiter le nombre de tâches coexistantes à un moment donné.

Bien que le langage Cilk ait été conçu et optimisé pour les machines à mémoire partagée, une version du langage Cilk destinée aux interconnections de SMP a été définie et implantée [90]. Cette version se base sur l'implantation d'une mémoire partagée entre les différents SMP, mais les performances obtenues ne sont pas à la hauteur de celles obtenues sur architecture SMP : les raisons principales sont d'une part que l'ordonnanceur de type *work-stealing* n'est pas adapté à la situation et d'autre part que l'absence de connaissance sur le flot de données ne permet pas de gérer convenablement les contraintes de localité sur les accès aux données effectués par les tâches.

2.8 Athapascan-1

La bibliothèque C++ Athapascan-1[32] [28, 55, 42] adopte une position intermédiaire entre les langages Cilk et Jade : le parallélisme de contrôle d'une application est exprimé explicitement par création de tâches, comme dans Cilk, mais les synchronisations entre ces tâches sont implicites, comme dans Jade : chaque tâche déclare les accès effectués sur une mémoire partagée fournie par la bibliothèque. Ces synchronisations implicites permettent de garantir la sémantique des accès aux données de la mémoire partagée. Différentes politiques d'ordonnancement sont disponibles, ce qui permet d'adapter au mieux l'ordonnancement aux caractéristiques de la machine et de l'application. De même que Cilk ou NESL, certaines de ces politiques ont des efficacités garanties.

2.8.1 Génération du parallélisme

La bibliothèque Athapascan-1 définit une mémoire partagée afin de permettre aux tâches de coopérer. Cette mémoire peut contenir des objets de tout type[33], et ceux-ci sont déclarés comme des objets standards mais de type `Shared<T>` où `T` représente le type spécifié par l'utilisateur. Le parallélisme est exprimé par création de tâches représentant l'exécution d'une procédure de manière asynchrone. Une tâche est créée par appel de la procédure générique `Fork<>()` de la bibliothèque instanciée avec le type de la tâche à créer (type représentant une fonction classe [100]) qui prend en paramètre la liste des paramètres de la tâche (la figure 2.9(a) illustre ces créations).

La tâche représente l'unité de calcul en Athapascan-1 et peut être considérée comme une procédure exécutée de manière asynchrone et ne faisant aucun effet de bord, c'est-à-dire dont la seule interaction avec l'environnement est effectuée à travers ses paramètres. Chaque tâche spécifie au moment de sa création les accès qui seront effectués sur la mémoire partagée au cours de son exécution ou lors de l'exécution de toute sa descendance. Les différentes données qui seront accédées sont spécifiées dans la liste des paramètres

[32] L'*URL* de la page officielle du projet est la suivante : `http://www-apache.imag.fr/software/ath1/`.

[33] Ce type doit cependant être « communicable », comme discuté à la section 5.1.1 page 92.

de la tâche et la description de ces droits d'accès (lecture `r`, écriture `w`, accumulation `cw`, modification `r_w`) est faite à l'aide d'un mécanisme de typage.

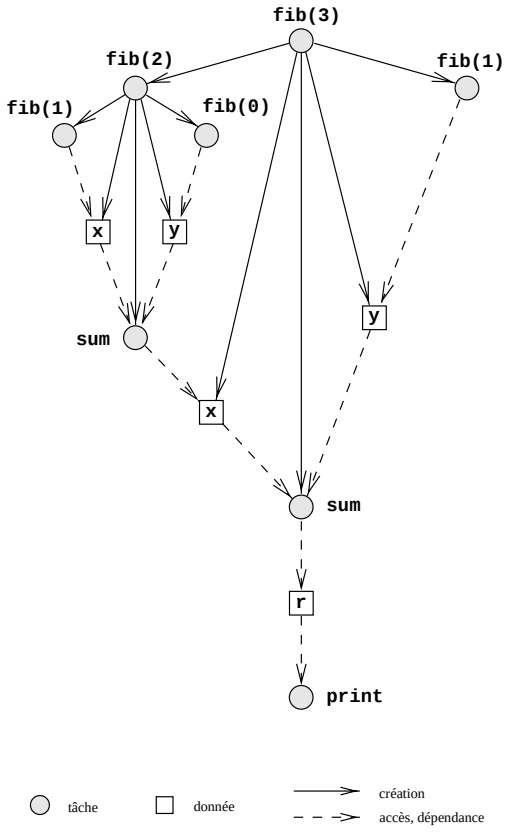
```
struct sum : public a1_task {
   void operator()( Shared_r<int> a,
                    Shared_r<int> b,
                    Shared_w<int> c )
   {
      c.write( a.read() + b.read() );
   }
};

struct fib : public a1_task {
   void operator()( int n,
                    Shared_w<int> r )
   {
      if( n<2 )
         r.write( n );
      else {
         Shared<int> x, y;
         Fork< fib >()( n-1, x );
         Fork< fib >()( n-2, y );
         Fork< sum >()( x, y, r );
      }
   }
};

struct print : public a1_task {
   void operator()( int n,
                    Shared_r<int> r )
   {
      printf("fib(%d)=%d\n", n, r.read());
   }
};

int main( int argc, char** argv )
{
   int n= atoi( argv[1] );
   Shared<int> r;
   Fork< fib >()( n, r );
   Fork< print >()( n, r );
   return 0;
}
```

(a)



(b)

Figure 2.9 *Calcul du n-ième nombre de Fibonacci en Athapascan-1.*
En (a), dans le code C++ utilisant la bibliothèque Athapascan-1, l'expression du parallélisme se fait en créant explicitement des tâches (`Fork`*). Les contraintes de précédence entre ces tâches sont déduites des accès effectués sur la mémoire partagée (objets* `Shared`*). Ces accès sont déclarés dans les prototypes des tâches par typage (*`_r` *pour une lecture,* `_w` *pour une écriture). Les accès effectifs aux données sont effectués par les fonctions membres* `read()` *pour une lecture et* `write()` *pour une écriture. En (b), le graphe d'exécution d'un appel à* `fib(3)`*. Bien que cela soit le cas pour cet exemple, le graphe n'est pas forcément de type série-parallèle. Le code semble plus long que pour les autres langages à cause essentiellement des encapsulations C++, des tâches* `sum` *et de la tâche d'affichage qui est nécessaire (car les synchronisations sont implicites et auront pour effet de retarder l'exécution des tâches sous contraintes de précédence).*

Une tâche a un fonctionement entièrement asynchrone par rapport à la tâche qui l'a créée. Cela implique donc, entre autres, que la tâche mère ne peut accéder les résultats de la tâche fille : ces résultats seront exploités par une tâche nécessairement différente. Le

système exécutif d'Athapascan-1 garantit (afin de permettre à d'éventuels ordonnanceurs de faire des estimations fiables sur la durée d'exécution des tâches) que l'exécution d'une tâche a lieu sans aucune synchronisation. Pour cela, une tâche ne pourra débuter son exécution que si toutes les données accédées en lecture sont prêtes, c'est-à-dire que toutes les tâches qui écrivaient sur cette donnée sont terminées[34].

2.8.2 Flot de données et sémantique

La sémantique des accès aux données de la mémoire partagée est définie, comme dans le cas des langages précédents, NESL, Cilk ou Jade, par rapport à une exécution séquentielle du programme : la valeur retournée par la fonction membre `read()` lors de toute exécution (éventuellement parallèle) est identique à celle retournée lors d'une exécution séquentielle. Bien que cette sémantique lexicographique semble impliquer des contraintes de synchronisation (typiquement l'instruction `sync` de Cilk), l'existence de restrictions sur le passage des droits d'accès aux tâches filles permet de garantir que l'exécution d'une tâche peut avoir lieu sans interruption. Cela implique, sur une machine à un seul processeur, que toutes les tâches filles peuvent être créées *à la fin* de l'exécution de la tâche mère. Cet ordre particulier, dit de « référence » et définit plus formellement définition 7 page 70, est central dans la définition du modèle de coût d'Athapascan-1. C'est en effet à partir de cet ordre que seront définis les valeurs S_1 et T_∞ du modèle de coût.

Athapascan-1 construit dynamiquement un graphe représentant les accès que font les tâches sur les données de la mémoire partagée [45]. Ces accès sont définis lors de la création de la tâche via le type de ses paramètres. Ce graphe de flot de données permet d'une part de définir (puis garantir) la sémantique des accès aux données et d'associer à l'interface de programmation Athapascan-1 un modèle de coût, mais d'autre part de fournir aux politiques d'ordonnancement une information complète sur l'exécution : précédences et localités des tâches et des données.

2.8.3 Modèle de coût et ordonnancement

L'ordonnancement dans Athapascan-1 est assuré par un module séparé de la bibliothèque. Cette séparation permet un développement aisé de nouvelles stratégies d'ordonnancement. De plus, l'accès au graphe de flot de données permet de considérer toutes les politiques d'ordonnancement basées sur sa connaissance, comme par exemple les stratégies de type statique qui permettent de minimiser le volume des communications de données nécessaires (ces stratégies sont appelées en cours d'exécution sur une portion de graphe, graphe qui a été construit dynamiquement).

Pour certaines politiques d'ordonnancement, un modèle de coût est associé au modèle de programmation Athapascan-1, comme pour Cilk ou NESL. Outre les grandeurs définies pour ces langages, la prise en compte des communications entraîne l'introduction de

[34] Il est à noter que les droits d'accès aux données partagées (tableau 3.2 page 66) et la construction du graphe (section 3.4 page 60) sont tels qu'aucun accès me peut être généré « spontanément » : si une tâche accédant un objet est créé alors **nécessairement** la tâche mère possédait un droit d'accès de même type sur cet objet.

nouvelles grandeurs (l'ordre d'exécution des tâches sur un seul processeur, fondamental dans la définition de S_1, est l'ordre de « référence ») :

- q, le nombre de processeurs virtuels émulés sur les p processeurs réels de la machine (afin de pouvoir majorer la latence des communications soit par du calcul soit par d'autres communications [61, 20]).
- h, le délai d'accès à distance d'un bit de donnée. Ce délai peut être borné en utilisant des processeurs virtuels, comme présenté dans [71] (h dépend de p et de q).
- C_1, le volume total d'accès distant. Cette valeur représente la somme sur tout le graphe d'exécution G des tailles des données accédées en lecture directe.
- C_∞, le volume d'accès distant effectué par un plus long chemin dans ce graphe (selon ce critère d'accès).
- σ, la taille du graphe G, c'est-à-dire le nombre de nœuds et d'arêtes.

Le calcul de la valeur T_∞ représentant la longueur d'un chemin critique du graphe peut être effectué dynamiquement lors de la construction du graphe à partr d'une exécution test[35]. Il faut tenir compte dans ce calcul du coût de description du graphe, c'est-à-dire du coût de la tâche mère (la tâche mère étant exécutée intégralement avant l'une quelconque de ses filles : c'est l'ordre de « référence »). La formule récursive suivante donne un moyen de calculer cette valeur (G'_t représente le sous graphe généré par la tâche t) :

$$T_\infty(t) = T_1(t) + T_\infty(G'_t)$$

Il est possible, à l'aide de ce modèle de coût, de garantir la durée et la consommation mémoire de toute exécution en fonction de la stratégie d'ordonnancement utilisée. Par exemple, une politique d'ordonnancement similaire à celle présentée dans [11] et utilisée dans NESL permet de garantir les majorations suivantes, et ce pour toute exécution :

$$\begin{aligned} T_p &\leq \frac{T_1 + hC_1}{p} + \frac{q}{p}(T_\infty + hC_\infty) + h\frac{q}{p}O(\sigma) \\ S_p &\leq S_1 + qO(T_\infty + hC_\infty) \end{aligned}$$

Ce résultat est similaire à celui obtenu par le langage NESL, mais tient compte, en plus, des coûts de communications. Il est possible, en utilisant un autre politique d'ordonnancement, d'obtenir une majoration du même type que celle de Cilk :

$$\begin{aligned} T_p &\leq \frac{T_1 + hC_1}{p} + h\frac{q}{p}O(T_\infty + hC_\infty) \\ S_p &\leq qS_1 \end{aligned}$$

Ces deux politiques sont détaillées dans la section 6.4 page 123.

[35] La proposition 3 page 64 montre en effet que le graphe est caractéristique d'une instance d'application et est donc invariant quelle que soit l'exécution de cette instance.

2.8.4 Bilan et implantation

Le modèle de programmation Athapascan-1, dont la justification complète est détaillée dans la thèse de Mathias Doreille [42], permet d'exprimer le parallélisme de contrôle d'une application par création de tâches. La manière dont sont effectués les accès à la mémoire virtuelle partagée permet de définir la sémantique des accès aux données. L'ordonnancement est confié à un module séparé ce qui permet d'une part d'offrir plusieurs politiques différentes et d'autre part de programmer des stratégies de régulation propres à l'application.

La bibliothèque Athapascan-1 est principalement destinée à la programmation des machines à mémoire distribuée, mais l'implantation étant entièrement portable, les machines à mémoire partagée de type *SMP* peuvent également être exploitées (la portabilité repose sur la couche Athapascan-0 [61] qui repose sur MPI et une bibliothèque de processus légers de type POSIX). L'utilisation d'une mémoire virtuelle partagée dans le modèle de programmation permet de définir une sémantique des accès aux données sans se soucier de l'architecture réelle de la machine.

2.9 Conclusion

Dans le but de dégager ce qui est « nécessaire » pour obtenir un modèle de coût, nous avons présenté dans ce chapitre quelques langages de programmation parallèle en nous focalisant sur les critères suivants : la génération du parallélisme, le type de graphe généré, la sémantique des accès aux données, l'ordonnancement, le modèle de coût associé au langage et le type d'architecture visé. Le tableau 2.2 page 49 synthétise l'analyse de ces langages selon ces différents critères.

Les langages à base de processus légers ne permettent pas de prédire ni la durée ni le volume mémoire nécessaires à une exécution, parce que des synchronisations peuvent intervenir entre deux processus légers quelconques. Jade, en introduisant une modélisation de l'application par un graphe de flot de données, permet de déduire de manière implicite les tâches et leurs précédences. Cependant, aucun modèle de coût n'est associé à ce langage. Le modèle de programmation BSP, en restreignant l'application à une série d'étapes de calcul séparées par des synchronisations globales permet de prédire le coût d'exécution de toute exécution. Si le modèle est d'une simplicité à toute épreuve, la mise en œuvre d'une telle barrière est cependant prohibitive pour une utilisation efficace. Les langages NESL et Cilk relachent cette restriction forte mais se limitent à une description de parallélisme de type « série-parallèle ». Les politiques d'ordonnancement utilisées dans ces deux langages permettent de majorer la durée et la consommation mémoire de toute exécution. Enfin, Athapascan-1 conserve ces propriétés mais, grâce à une analyse du flot de données de l'application permet de relâcher la contrainte imposée sur le graphe : tous les graphes construits de manière « emboîtés » sont admissibles.

Nous retirons de cette brève étude qu'il n'est pas possible dans un cas général de prédire les performances de toute exécution : les synchronisations, non contrôlées, ne permettent pas la mise au point d'un modèle de coût. Des restrictions sur les synchronisations possibles (explicites pour BSP, NESL ou [84], Cilk, ou implicites pour Jade

Langage	Type de graphe	Sémantique	Ordonnancement	Modèle de coût	Architecture visée
à base de processus légers	×	×	quelques choix	×	*SMP*
Jade	flot de données emboîté	type séquentielle	fixé	×	*SMP* et distribuée
Modèle BSP	×	×	×	durée	quelconque
NESL	flot de données série-parallèle	type séquentielle	fixé	durée et mémoire	*SMP* et distribuée
Cilk	précédence série-parallèle	type séquentielle	fixé	durée et mémoire	*SMP*
Athapascan-1	flot de données emboîté	type séquentielle	entièrement adaptable	durée et mémoire	*SMP* et distribuée

Tableau 2.2 *Comparaison de différents langages de programmation parallèle de « haut niveau ».*
Nous entendons par langages de « haut niveau » les langages offrant un ordonnancement automatique des tâches et dans lesquels les applications sont exprimées en ne faisant aucune hypothèse sur la machine sur laquelle aura lieu l'exécution. Nous remarquons que la modélisation de l'exécution d'une application par un graphe permet de définir la sémantique des accès aux données et d'associer un modèle de coût au langage.

et Athapascan-1) permettent, en association avec une politique d'ordonnancement adaptée, de garantir les performances de l'exécution. L'application est alors modélisée par un graphe à partir duquel est décrit aisément la sémantique des accès aux données et le modèle de coût.

Nous étudions, au cours des deux parties suivantes, l'apport de la connaissance du graphe de flot de données d'une application en ce qui concerne la définition et la garantie de la sémantique, l'ordonnancement et la définition d'un modèle de coût associé au langage. Nous étudions également la construction dynamique de ce graphe dans un environnement distribué et appliquons ces résultats à la bibliothèque Athapascan-1 qui constitue le cadre de cette étude.

Partie I

Interprétation distribuée du flot de données

3

Flot de données dynamique et sémantique d'Athapascan-1

Ce chapitre définit le concept de graphe de flot de données qui permet de caractériser une exécution puis, à partir de ce graphe, définit la sémantique d'Athapascan-1. Les principales sections de ce chapitre traitent les points suivants :

- La modélisation d'une application par un **graphe de flot de données** qui décrit les accès effectués par les tâches sur les données (section 3.2 page 54).
- Les contraintes imposées aux tâches permettant une construction dynamique du graphe **indépendante** de l'exécution (section 3.3 page 60).
- Un algorithme de **construction à la volée** du graphe de flot de données (section 3.4 page 60).
- La **sémantique** des accès aux données de l'interface de programmation Athapascan-1 (section 3.5 page 65).

3.1 Objectif

Toute exécution d'une application peut être caractérisée, *a posteriori*, par les « actions » effectuées par les tâches sur les données de la mémoire partagée. Représentées sous forme d'un graphe, ces actions constituent le flot de données de cette exécution particulière. Cependant, dans un cadre général, ce graphe de flot de données est **dépendant** des conditions d'exécution : deux exécutions différentes ayant les mêmes valeurs d'entrées peuvent aboutir à deux graphes différents[1].

Notre objectif est alors de montrer que, en imposant certaines conditions sur les tâches (mais aucune sur l'exécution) il est possible d'associer à toute instance d'application[2] un **unique** graphe de flot de données. Ce graphe sera donc caractéristique de l'instance et

[1] Par exemple dans le cas d'une utilisation directe de bibliothèques de *threads*. Le problème vient des possibilités de concurrence (*race-conditions*) sur les accès à la mémoire.

[2] Nous appelons « instance d'application » l'association d'une application et d'un ensemble de ses valeurs d'entrées.

non plus d'une exécution particulière : il est alors un moyen idéal pour définir la sémantique des accès aux données. Nous montrons alors, dans le cadre d'Athapascan-1, que la connaissance et la construction dynamique du flot de données de l'application permet de **définir** la sémantique du langage et de **garantir** la correction de son implantation.

Nous étudions tout d'abord dans ce chapitre le graphe de flot de données, puis les restrictions ajoutées sur les tâches et un algorithme de construction dynamique du graphe, et enfin la sémantique des accès aux données d'Athapascan-1 définie à partir du graphe de flot de données.

3.2 Modélisation d'une application par un graphe de flot de données

Cette section est consacrée à l'étude du graphe de flot de données associé à une instance d'application. Nous présentons tout d'abord les éléments constitutifs de ce graphe, puis définissons formellement la notion de graphe de flot de données et terminons par la définition et le calcul des états qui sont associés aux nœuds du graphe.

3.2.1 Éléments constitutifs du graphe

Le graphe de flot de données est constitué par les tâches, les versions (qui représentent les données en mémoire partagée) et les accès des tâches sur les versions.

3.2.1.1 Tâches

Une application est caractérisée par les tâches et les données partagées qu'elle crée. Une tâche, lors de son exécution, peut créer de nouvelles tâches ou de nouvelles données partagées. La liste des données partagées qui pourront être accédées par une tâche sont spécifiées dans les paramètres de la tâche lors de sa création.

3.2.1.2 Versions de données partagées

Afin de pouvoir tracer le flot de données concernant les accès aux données partagées, chaque donnée est considérée comme une succession de versions. Ce sont ces versions, représentant les différentes valeurs prises par la donnée au cours de l'exécution, qui sont accédées par les tâches.

Considérons un objet x de la mémoire partagée. À cet identificateur x (et sa portée), traduisant au niveau du langage la liaison à une certaine donnée partagée, correspond une succession de versions $(v_{x,i})_{i\geq 0}$, comme représenté figure 3.1. De manière schématique, la version $v_{x,i}$ traduit la valeur associée à l'identificateur x à l'état d'avancement i du programme. La suite des versions représente donc la suite des valeurs référencées par la donnée partagée ; chaque version constitue alors une sorte de variable à assignation unique [9].

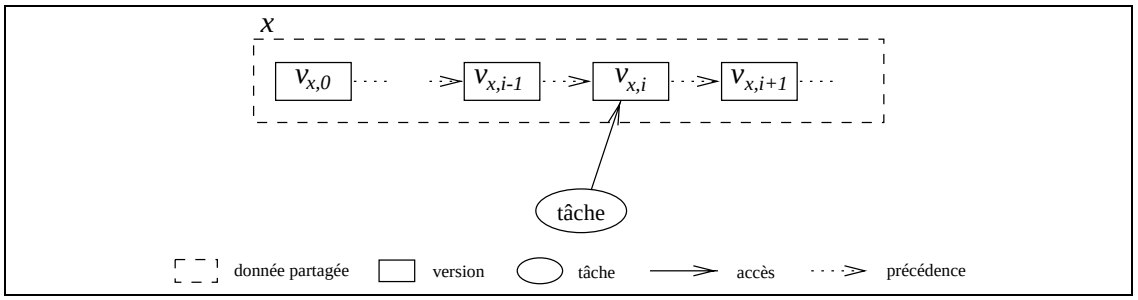


Figure 3.1 *Succession des versions associées à une donnée partagée.*
La donnée partagée x est composée d'une succession de versions, $(v_{x,i})_{i\geq 0}$. Lorsqu'une tâche déclare accéder un objet x au niveau du programme, c'est une version $v_{x,i}$ de la séquence associée à x qui sera retournée. Cette version représente la valeur qui sera manipulée par la tâche lors de l'accès effectif.

Afin de décrire plus aisément la création du graphe et des versions des données partagées, nous associons les trois fonctions suivantes à chaque version $v_{x,i}$ d'une donnée partagée x :

- ref$(v_{x,i})$ qui retourne l'objet partagé auquel fait référence cette version ; ici, x.
- pred$(v_{x,i})$ qui retourne la version précédant $v_{x,i}$ dans la séquence ; ici, $v_{x,i-1}$. Si la version est la première, cette fonction retourne « rien ».
- succ$(v_{x,i})$ qui retourne la version suivant $v_{x,i}$ dans la séquence ; ici, $v_{x,i+1}$. Si la version est la dernière, cette fonction retourne « rien ».

La numérotation $0, ..., i-1, i, ...$ présentée ici n'a qu'une valeur explicative, l'ordre étant déduit du chaînage entre les versions. Ce chaînage représente la succession des écritures qui vont avoir lieu sur la donnée située en mémoire partagée. Nous prouvons dans la suite (proposition 5 page 68) que pour toute exécution parallèle l'ordre de ces versions est invariant et correspond à l'ordre des écritures lors d'une exécution séquentielle.

Lorsqu'une tâche prend pour paramètre une référence sur une donnée en mémoire partagée, c'est une référence sur une des versions associées à la donnée partagée qui lui sera retournée lors de l'exécution. Cette version, ordonnée par chaînage, identifie la valeur qui sera retournée lors de l'accès puisque le chaînage spécifie la série des écritures qui seront vues par cette lecture.

3.2.1.3 Droits d'accès des tâches sur les versions

Les données partagées sont accédées par les tâches avec un **droit d'accès**. Ce droit d'accès, qui peut être lecture, écriture, accumulation ou modification, spécifie le type des accès que la tâche est autorisée à effectuer sur la donnée. Ce droit d'accès est raffiné par un **mode d'accès** qui spécifie si la tâche courante va effectivement utiliser son droit (mode direct) ou si elle va se contenter de le passer à une ou plusieurs de ses tâches filles (mode différé). Le mode d'accès constitue un rafinement du droit qui permet de relacher les contraintes de précédence entre les tâches et donc d'exploiter *a priori* plus de parallélisme de l'application. Ces droits et modes d'accès ainsi que les passages de paramètres autorisés sont illustrés dans le tableau 3.2 page 66.

Dans le graphe, le droit d'accès sera représenté par les arêtes reliant les tâches aux versions. Le mode d'accès n'est pas directement visible dans le graphe, mais peut être implanté (si nécessaire) sous forme d'attribut des arêtes, par exemple.

3.2.2 Graphe de flot de données

Lors de l'exécution d'une application, les tâches générées, les données partagées et les droits d'accès sur ces données constituent un graphe de flot de données.

Définition 1 *Le* **graphe de flot de données** *associé à une exécution est le graphe* $G = (V, E)$ *tel que les tâches (*V_t*) et les versions (*V_v*) forment l'ensemble* $V = V_t \cup V_v$ *des nœuds, et les accès des tâches sur les versions forment l'ensemble* E *des arêtes. Ce graphe est bipartie puisque* $E \subset (V_t \times V_v) \cup (V_v \times V_t)$.

La signification d'une arête est la suivante : soient $t \in V_t$ une tâche et $v \in V_v$ une version, l'arête $(t, v) \in E$ traduit un droit d'accès en écriture de la tâche t sur la version v, et l'arête (v, t) un droit d'accès en lecture de la tâche t sur la version v.

À chaque nœud du graphe est associé un état permettant de résoudre les contraintes de précédence. Ces états sont résumés dans le tableau 3.1 page 58. Un exemple de graphe de flot de données traduisant les accès de 4 tâches sur 3 versions en mémoire partagée est donné figure 3.2 page 56.

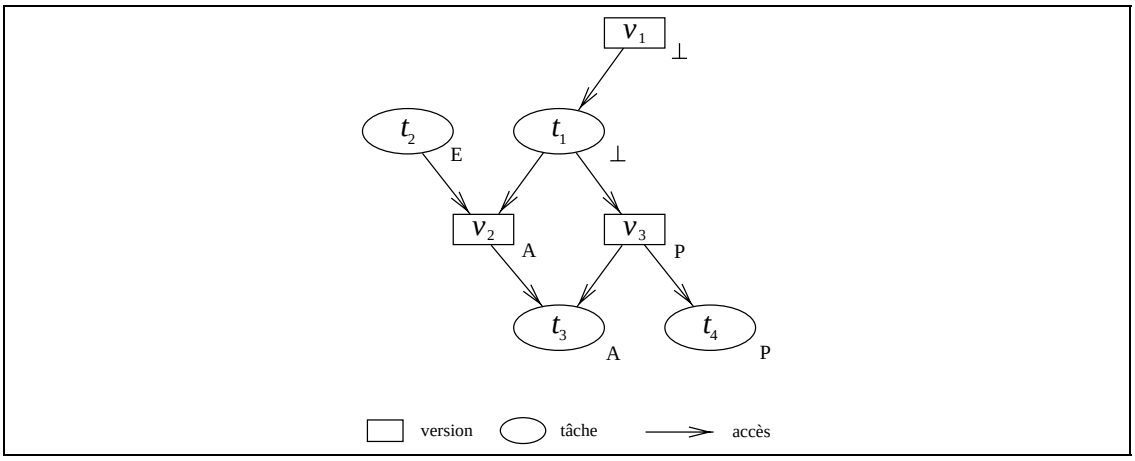


Figure 3.2 *Graphe des accès aux données.*
Les quatre tâches t_1, t_2, t_3, t_4 *accèdent en concurence les trois versions* v_1, v_2 *et* v_3 *associées à trois données différentes. La tâche* t_2 *accède en écriture* v_2 *qui est elle-même accédée en lecture par* t_3 *: il y a donc une contrainte de précédence entre ces deux tâches :* t_2 *doit être exécutée avant* t_3*. La version* v_2 *est en attente (état* A*) car la tâche* t_2 *n'est pas terminée (état* E*). Ces états sont définis section 3.2.3 page 57. Il est à noter que les tâches* t_1 *et* t_2 *peuvent être exécutées en parallèle, l'accès concurrent en écriture sera géré par la mémoire virtuelle partagée afin de respecté la sémantique (compte tenu de la construction du graphe, section 3.4 page 60, il s'agit nécessairement une accumulation). De même pour* t_3 *et* t_4*.*

Il est à noter que certains modèles de programmation[3] ne considèrent pas les accès aux données. Le graphe est donc réduit à un graphe de précédence entre les tâches, ou graphe de tâches.

[3]Comme par exemple Cilk ou NESL.

Définition 2 *Le* **graphe de tâches** *ou* **graphe de précédence** *associé à une exécution est le graphe $G = (V, E)$ tel que V représente l'ensemble des tâches de l'application et E l'ensemble des contraintes de précédence entre les tâches ($E \subset V^2$).*

À partir de ce graphe G, orienté et supposé[4] sans cycle, il est possible de définir un **ordre partiel** sur l'ensemble V des tâches, noté $\prec_G$, qui traduit la **précédence** entre les tâches : $t \prec_G t'$ implique qu'il existe un chemin dans le graphe (c'est-à-dire une succession d'arêtes de E) qui relie t à t'. Autrement dit, deux tâches t et t' ont un lien de précédence, $t \prec_G t'$, si la tâche t doit être exécutée avant t'.

Dans le cas des graphes de flot de données, cette même relation de précédence peut être définie en considérant le graphe de précédence $G' = (V', E')$ tel que $V' = V_t$, et $(t_1, t_2) \in E'$ si et seulement si il existe $v \in V_v$ telle que (t_1, v) et $(v, t_2) \in E$. Ce qui revient à dire que $t \prec_G t'$ si et seulement si t' lit ou modifie une donnée écrite ou modifiée par t.

3.2.3 États associés aux nœuds du graphe

À chaque nœud n du graphe de flot de données (par exemple représenté figure 3.2 page 56), on associe un attribut qui représente son état. Le graphe G des accès aux données précédemment construit est un graphe bipartie orienté et sans cycle : il définit donc un ordre partiel sur les nœuds, donc en particulier sur les tâches (voir le graphe de tâches G' introduit précédemment). Cet ordre traduit les contraintes de précédence et l'état des nœuds va permettre d'implanter cet ordre : l'ordre dans lequel les nœuds tâches passent dans l'état d'exécution E est compatible[5] avec l'ordre défini par le graphe. Les calculs effectués se basent uniquement sur le **droit** d'accès des tâches sur les données partagées, mais il est trivial d'y ajouter le raffinement apporté par le mode d'accès.

3.2.3.1 Définition

Cet état, noté $e(n)$, varie au cours du temps et prend sa valeur dans l'ensemble $\{A, P, E, \perp\}$ pour les tâches et dans $\{A, P, \perp\}$ pour les versions. La signification intuitive de ces états est donnée dans le tableau 3.1 page 58. Ces états permettent de résoudre les contraintes de précédence entre les tâches. La manière dont sont calculés ces états est présentée dans la section suivante.

Il est à noter que d'autres états que ceux présentés ici auraient tout aussi bien pû être définis. Nous ne présentons que ceux qui sont utilisés dans l'implantation du graphe de flot de données de la version distribuée de la bibliothèque Athapascan-1.

Pour tout nœud du graphe, les trois relations suivantes permettent de définir son état, noté $e(t)$ pour l'état d'une tâche et $e(v)$ pour celui d'une version. En supposant les détermination du graphe atomique, les relations peuvent être considérées comme des équivalences.

[4] C'est le modèle de programmation qui doit garantir que le graphe généré par toute application est sans cycle ; c'est le cas en particulier pour tous les langages de « haut niveau » considérés.

[5] L'ordre $\prec_1$ est dit compatible avec $\prec_2$ si, pour tout couple d'éléments (a, b), $(a \prec_2 b) \Rightarrow (a \prec_1 b)$. En d'autres termes, $\prec_2 \subseteq \prec_1$.

État	Tâche	Version
A : attente	Tous les paramètres de la tâche ne sont pas prêts ; elle ne peut donc pas commencer son exécution.	Il existe encore une ou plusieurs tâches possédant un droit en écriture sur la version ; la donnée n'est donc pas disponible pour une lecture.
P : prêt	Tous les paramètres accédés en lecture sont prêts ; la tâche peut être exécutée.	Plus aucune tâche ne peut accéder la version en écriture ; les lectures peuvent démarrer.
E : exécution	La tâche est en cours d'exécution.	$\times$
$\perp$: terminé	L'exécution est terminée et la tâche détruite.	Plus aucune tâche n'accède, ni en lecture, ni en écriture, la version ; la mémoire peut être libérée (ou réutilisée pour une mise à jour en place).

Tableau 3.1 *États des nœuds du graphe.*
Ce tableau définit de manière informelle les différents états possibles des nœuds du graphe. Les relations (3.1) à (3.3) page 58 les définissent formellement.

La relation suivante (3.1) traduit que la tâche $t \in V_t$ est en attente si l'une quelconque des versions v accédées en lecture n'est pas prête (donc en attente car elle ne peut pas être terminée du fait de la présence de t).

$$\Big(\exists (v,t) \in E / e(v) = A\Big) \iff e(t) = A \tag{3.1}$$

La relation suivante (3.2) traduit que la version $v \in V_v$ est prête ou terminée si toutes les tâches t qui possèdent un droit en écriture sur cette version sont dans l'état terminé et si la version antérieure pred(v) est prête ou terminée. Cette relation est équivalente à la suivante : une version non prête implique l'existence d'une tâche non terminée possédant un droit en écriture sur cette version ou bien l'état d'attente de l'éventuelle version antérieure[6].

$$\Big(\forall (t,v) \in E, e(t) = \perp\Big) \wedge \Big(e(\text{pred}(v)) \in \{P, \perp\}\Big) \iff e(v) \in \{P, \perp\} \tag{3.2}$$

La relation suivante (3.3) traduit que la version v est terminée si plus aucune tâche t ne possède de droit d'accès sur elle.

$$\Big(\Big(\forall t \in V_t / (t,v) \vee (v,t) \in E\Big)\ e(t) = \perp\Big) \iff e(v) = \perp \tag{3.3}$$

Proposition 1 *Les invariants sur les états du graphe (les relations définies section 3.2.3.1 page 57) garantissent le respect des contraintes de précédence entre les tâches.*

Preuve. Soit t une tâche et $\mathcal{T}$ l'ensemble des tâches la précédant. Soit $\mathcal{V}$ l'ensemble des versions constituant le lien entre l'ensemble $\mathcal{T}$ et la tâche t : toutes ces versions sont accédées en lecture par t et en écriture par une au moins des tâches de $\mathcal{T}$.

[6] L'attente de la version antérieure est dû à la sémantique associée au langage. Voir la proposition 4 page 67.

Si $e(t) = P$, alors la contraposée de l'équivalence (3.1) implique que toutes les versions de $\mathcal{V}$ sont dans l'état P (la relation (3.3) interdit à ces versions d'être dans l'état $\perp$). Si toutes les versions de $\mathcal{V}$ sont prêtes, alors l'équivalence (3.2) implique que toutes les tâches de $\mathcal{T}$ sont dans l'état terminé. □

3.2.3.2 Calcul des états des nœuds du graphe

L'état d'une version est calculé lors de sa création, puis est recalculé lors d'un changement d'état de la version antérieure ou lors de la terminaison d'une tâche possédant un accès sur cette version. De même, l'état d'une tâche est calculé lors de sa création, puis est recalculé lors du changement d'état de chaque version accédée en lecture jusqu'à passer à l'état P. Il est ensuite modifié par le système[7] lors du début de l'exécution (passage de l'état P à E), ou à la fin d'exécution (passage de l'état E à $\perp$).

Dans le cas d'Athapascan-1, le passage d'une tâche de l'état P à E est effectué sur décision de l'ordonnanceur; le passage de l'état E à $\perp$ est effectué par la tâche après exécution de sa dernière instruction.

La figure 3.3 page 59 représente les changements possibles des états des nœuds du graphe. L'état L d'une version est un état dû au caractère distribué du graphe et sera discuté au cours de la section 4.2 page 74.

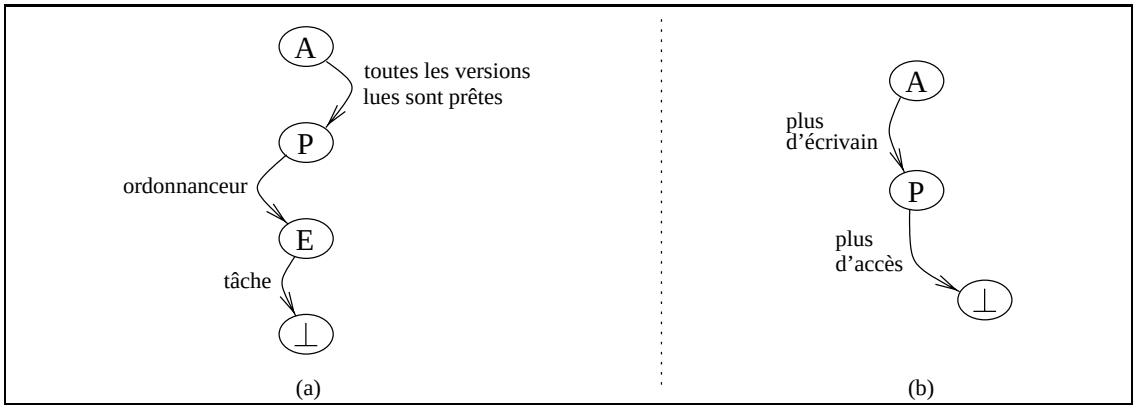


Figure 3.3 *Changements d'états des nœuds du graphe.*
En (a) sont représentées les évolutions possibles d'une tâche, et en (b) celles d'une version. Le changement d'état d'un nœud du graphe est soit dû à une intervention extérieure, le système, soit il est la conséquence de l'évolution des états d'autres nœuds.

Une hypothèse de réactivité sur le calcul des états, qui garantit que le graphe est averti au bout d'un temps fini des modifications d'états des tâches, permet de supposer effectivement que les modifications sont effectuées de manière atomique. Le calcul des états permet donc effectivement de résoudre les contraintes de précédence entre les tâches.

[7] On entends ici par système le module implantant le langage.

3.3 Restrictions imposées sur les tâches

Le graphe est construit dynamiquement au cours de l'exécution et n'est donc connu dans son intégralité qu'une fois l'exécution de l'application terminée. On pourrait alors craindre que la forme de ce graphe dépende des conditions particulières de l'exécution, c'est-à-dire que deux exécutions différentes avec des entrées identiques construisent deux graphes différents. Pour écarter ce problème, l'application est supposée **déterministe**[8]. C'est-à-dire que le résultat dépend exclusivement des entrées de l'application mais en aucun cas des conditions de l'exécution. Ainsi, à un jeu de valeurs en entrée correspond un unique graphe caractéristique de toutes les exécutions prenant en entrée ces mêmes valeurs. En particulier, on suppose que le travail à effectuer, c'est-à-dire l'ensemble des tâches générées au sein du graphe, est indépendant de l'ordre d'exécution de celles-ci, pourvu que cet ordre respecte les contraintes de précédence. Ceci exclut par exemple de fait tous les algorithmes de recherche effectuant des coupures (typiquement les algorithmes de type *branch and bound*) pour lesquels des accélérations tout comme des décélérations super-linéaires peuvent avoir lieu.

Afin de permettre d'une part une description à la volée du graphe de flot de données et d'autre part la définition de la sémantique des accès aux données, les tâches sont supposées **sans effet de bord** : les données partagées effectivement accédées par la tâche ou passées en paramètres aux tâches filles ont été soit déclarées par la tâche, soit reçues dans la liste des paramètres. Ceci implique en particulier que la description du graphe est de type **emboîté**, c'est-à-dire que les tâches créées ne peuvent accéder que les données déjà accédées par la tâche mère.

3.4 Construction dynamique du graphe

Le graphe est créé dynamiquement par adjonction de nœuds et d'arêtes lors des créations de tâches ou des déclarations d'objets en mémoire partagée par une tâche t en cours d'exécution. Nous décrivons dans cette section les évolutions du graphe lors de ces déclarations. Il est à noter que toutes les constructions ne sont pas autorisées pour toutes les tâches : des restrictions sur le passage des paramètres lors de la création des tâches sont introduites pour permettre de garantir la sémantique du langage (voir par exemple le tableau 3.2 page 66).

- (C0) Initialement, le graphe ne contient que la tâche principale (la fonction `main`).

3.4.1 Déclaration d'une donnée en mémoire partagée

Supposons que la tâche t déclare une donnée x en mémoire partagée.

- (C1) Le graphe G est alors modifié en ajoutant deux nouvelles versions v et v' dans V et les dépendances avec t correspondantes (v, t) et (t, v') dans E (figure 3.4 page

[8] Ce qui est vrai dans le cas d'Athapascan-1, comme cela sera établi dans la proposition 3 page 64.

61). v et v' sont définies par :

$$\begin{cases} \text{ref}(v) = \text{ref}(v') = x \\ \text{succ}(v) = v' \\ e(v) = P, e(v') = A \end{cases}$$

v constitue la version courante de la donnée partagée et v' la version future (celle dans laquelle les écritures de la tâche t seront stockées).

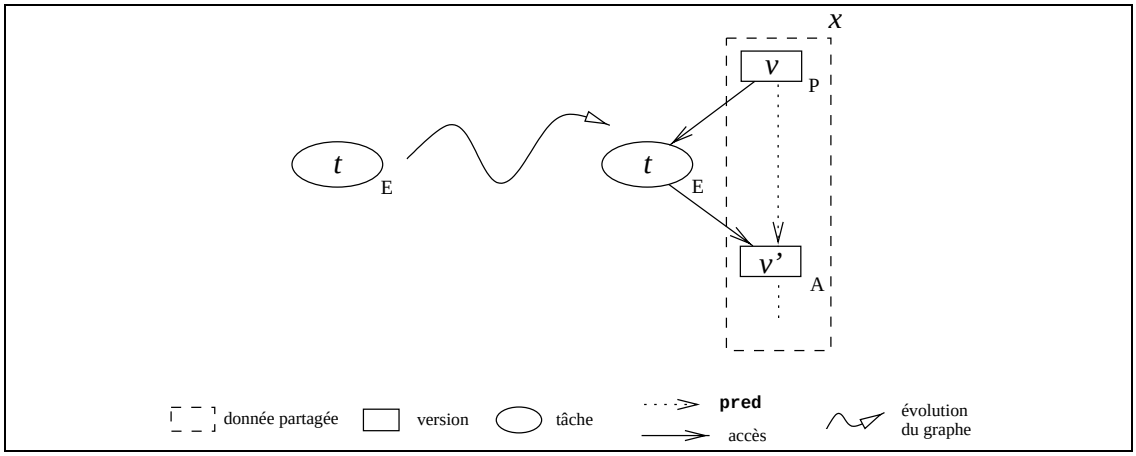


Figure 3.4 *(C1) Déclaration d'une donnée en mémoire partagée.*
Deux versions successives v et v' sont créées. v contient la valeur initiale de la donnée x (version courante) et est accédée en lecture par la tâche t. Les écritures de t seront faites dans la version v' (version future).

3.4.2 Création d'une tâche

Supposons que la tâche t crée une nouvelle tâche t' lors de son exécution. Le graphe G est alors modifié en ajoutant t' dans V. De plus, pour toute donnée partagée x passée en paramètre lors de la création de t', le graphe G est modifié pour décrire les changements de la séquence des versions associées à x (résultant de la création de la tâche t') en fonction des droits d'accès possédé par la mère et ceux requis par la fille.

Les conditions (C2) à (C5) suivantes décrivent les différentes transitions du graphe de flot de données concernant ces créations de tâches :

- (C2) Si t' requiert un droit d'accès en lecture sur x, ajout de (v, t') dans E avec la version $v \in V$ telle que $(v, t) \in E$ et ref$(v) = x$, comme illustré figure 3.5 page 62. Il n'y a pas de changement de version courante. L'état de la tâche dépendra de l'état de v et du mode d'accès requis.
- (C3) Si t' requiert un droit d'accès en écriture avec une sémantique d'accumulation sur x, ajout de (t', v) dans E avec la version $v \in V$ telle que $(t, v) \in E$ et ref$(v) = x$, comme illustré figure 3.6 page 62. Il est à noter que t possédait nécessairement le même droit d'accès en écriture sur la version future de x, il n'y a donc pas de changement de version.

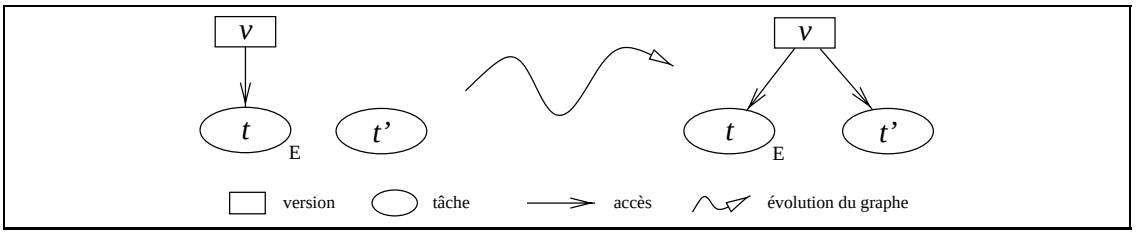


Figure 3.5 *(C2) Ajout d'un lecteur.*
Les deux tâches t et t' accèdent en concurrence la même version v de la donnée partagée. Les valeurs lues sont donc identiques.

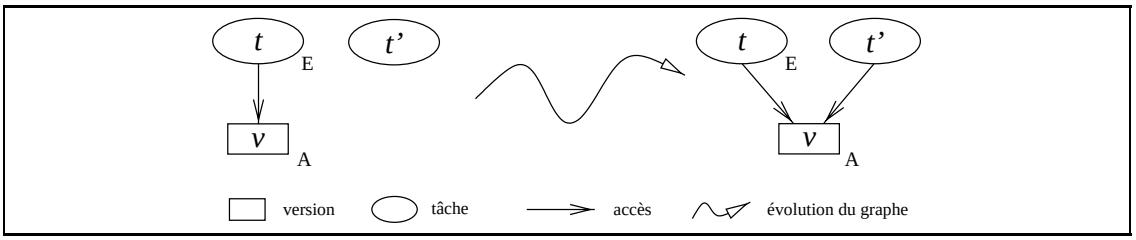


Figure 3.6 *(C3) Ajout d'un écrivain avec sémantique d'accumulation.*
Les deux tâches t et t' accèdent en concurrence la même version v de la donnée partagée.

- (C4) Si t' requiert un droit d'accès en écriture et t ne possède qu'un droit d'accès en écriture sur x, $(t, v) \in E$, ajout de v' dans V et de (t', v') dans E avec v' une nouvelle version telle que (figure 3.7 page 62) :

$$\begin{cases} \text{ref}(v) = \text{ref}(v') = x \\ \text{pred}(v') = \text{pred}(v) \\ \text{succ}(v') = v \\ e(v') = A \end{cases}$$

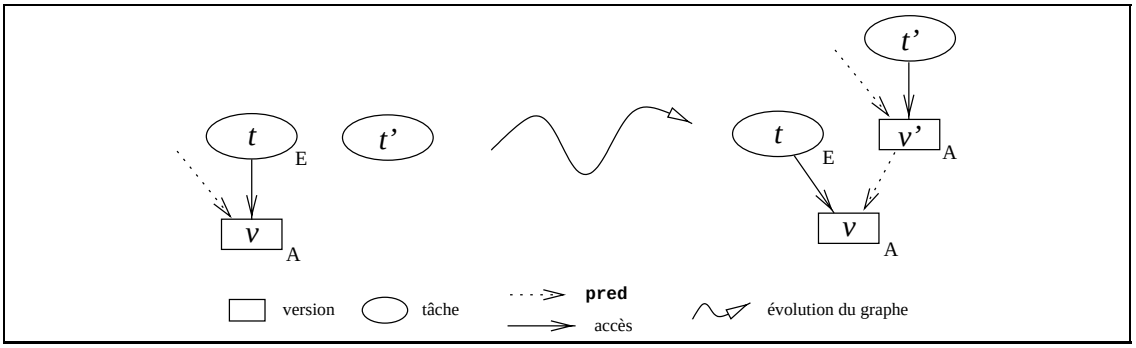


Figure 3.7 *(C4) Ajout d'un écrivain sur une donnée non accessible en lecture.*
Il y a création d'une version v', précédant v, qui contiendra le résultat de l'écriture de t'.

- (C5) Sinon, t' requiert un droit en écriture et t possède un droit d'accès en lecture/écriture sur x. Il est à noter que pour être autorisée à créer une tâche requérant un droit en écriture sur x, la tâche t possède également un droit en écriture sur la

donnée (il n'y a pas d'effet de bord sur les données). Il existe donc $(v_1, t) \in E$ et $(t, v_2) \in E$ tels que ref(v_1) = ref$(v_2) = x$ et succ$(v_1) = v_2$, v_1 étant accédé en lecture, $(v_1, t) \in E$, et v_2 en écriture, $(t, v_2) \in E$. Il y a alors ajout d'une nouvelle version v' dans V(figure 3.8 page 63). De plus, la version future de t' devient la version courante de t[9]. Il y a donc ajout de (t', v') et (v', t) dans E puis retrait de (v_1, t) de E :

$$\begin{cases} \text{ref}(v') = \text{ref}(v_1) = \text{ref}(v_2) = x \\ \text{pred}(v') = v_1 \\ \text{succ}(v') = v_2 \\ \text{succ}(v_1) = \text{pred}(v_2) = v' \\ e(v') = A \end{cases}$$

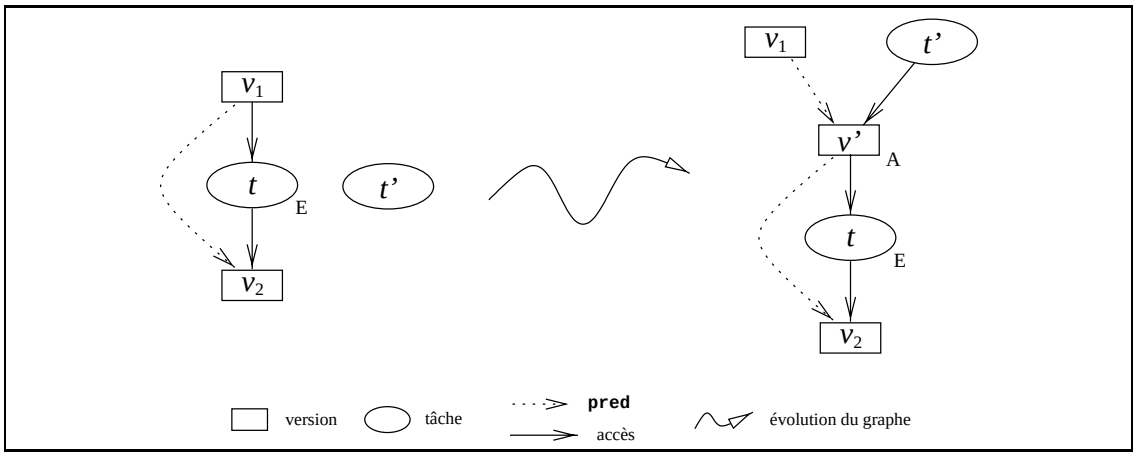


Figure 3.8 *(C5) Ajout d'un écrivain sur une donnée accessible en lecture.*
La tâche t repositionne sa lecture sur la version v' qui contiendra le résultat de l'éventuelle écriture de t'. La version future de t' devient la version courante de t.

Il est à noter que si la tâche t' requiert un droit d'accès en lecture sur x en plus de son droit en écriture, alors l'étape de modification du graphe (C2) doit être effectuée **avant**[10] (C5). La version courante de t' sera donc l'ancienne version courante de t.

3.4.3 Propriétés de la construction

Nous décrivons dans cette section quelques propriétés de cette construction.

[9] En d'autres termes cela signifie que l'écriture de la tâche fille t' sera vue par le reste des instructions de la tâche mère t : ce fonctionnement permettra de garantir de manière aisé la sémantique de type séquentielle associée au langage. Il est à noter que cette pseudo « dépendance » fille-mère n'introduit aucune synchronisation et que la mère peut toujours s'exécuter en concurrence avec sa fille t' (voire avant). Cela est dû aux restrictions de passage des droits d'accès (tableau 3.2 page 66) qui interdit entre autres à la mère d'accéder de manière effective toute donnée produite par l'une quelconque de ses filles.

[10] En effet, si la modification du graphe (C2) était effectuée après (C5), il y aurait formation d'un cycle dans le graphe, la tâche créée t' accédant en lecture **et** en écriture la version v'.

Propriété 1 *Les seules situations où plusieurs tâches accèdent la même version sont les accès en lecture seule ou en écriture seule avec sémantique d'accumulation. Ces accès sont dit concurrents.*

Preuve. Ces situations correspondent aux cas de construction (C2) et (C3). Les constructions (C4) et (C5) créent une nouvelle version qui sera associée à la nouvelle tâche créée et retirent l'accès qui aurait pû être partagé en lecture dans le cas (C5). Le mécanisme de typage des accès aux données garantit que ces modifications (C4) et (C5) ne peuvent être initiées que par des tâches possédant un accès exclusif sur les versions altérées : lecture/écriture ou écriture. □

Proposition 2 *Le graphe $G = (V, E)$ ainsi construit est sans cycle.*

Preuve. Soit un graphe G construit dynamiquement en appliquant les règles de construction précédentes. Supposons qu'il existe une date d à laquelle un cycle est présent à l'intérieur de ce graphe.

Initialement, le graphe est vide et ne contient pas de cycle. Considérons alors la date d' minimale à partir de laquelle un cycle est apparu dans le graphe. Il existe nécessairement une tâche t' du cycle qui a été introduite à la date d'. Soit t la mère de cette tâche. Nous nous proposons de montrer que le cycle était déjà présent au travers de la tâche t avant la création de t', ce qui contredit la minimalité de d et donc l'existence d'une date quelconque à laquelle un cycle est présent dans G.

La tâche t' est dans le cycle au travers de deux versions v_1 et v_2 accédées respectivement en lecture et en écriture. La version v_1 était nécessairement accédée par t avant la création de t', cas (C1) ou (C2). La situation est identique pour v_2 dans le cas d'une écriture avec sémantique d'accumulation, cas (C3). Il reste donc uniquement à montrer que dans les deux schémas d'ajout d'écrivains (C4) et (C5) tout ce qui est atteignable à partir de la nouvelle version v_2 était déjà atteignable à partir de la tâche t. Dans (C4) et (C5) la tâche t accède en écriture la version v qui suit dans l'ordre de précédence des versions la version v_2. Le cycle passant par v_2 passe donc nécessairement par v. Ainsi la tâche t fermait déjà le cycle avant la création de t', ce qui contredit l'existence de d' donc de d. Il est à noter que la preuve reste valide dans le cas où plusieurs tâches intervenants dans le cycle sont créées à la même date d' : il suffit de considérer toutes les tâches mères de toutes les tâches introduites à la date d' pour construire un cycle antérieur à cette date. □

Proposition 3 *Le graphe $G = (V, E)$ obtenu après toutes les créations de tâches est indépendant de l'ordonnancement des tâches qui a été effectué.*

Preuve. Les modifications du graphe par ajout de lecteurs (C2) ou d'écrivains concurrents (C3) ne posent pas de problème, l'ajout étant toujours effectué sur la même version de la donnée, sans altération des propriétés de cette version. Les seules modifications qui ajoutent ou modifient les versions du graphe sont des ajouts d'écrivains (C4) et (C5). Or le mécanisme de typage statique des accès aux données garantit (à la compilation) que ces modifications ne peuvent être initiées que par des tâches possédant un accès exclusif sur les versions altérées : lecture/écriture ou écriture, propriété 1 page 64. Les modifications effectuées par les autres tâches n'affectent donc aucunement l'état des versions

accédées par la tâche courante. Ainsi, quel que soit l'ordonnancement effectué au cours de l'exécution, chaque tâche accèdera toujours les mêmes versions de données partagées. □

3.5 Sémantique d'Athapascan-1

Nous présentons dans cette section le graphe de flot de données construit par Athapascan-1 lors de l'exécution. Ce graphe est construit à partir du typage des paramètres des tâches. À partir de ce graphe nous définissons la sémantique des accès aux données d'Athapascan-1.

3.5.1 Définition du flot de données dans Athapascan-1

Nous présentons dans cette section la sémantique des accès aux données définie à partir du graphe de flot de données.

Athapascan-1 construit dynamiquement un graphe représentant les accès que font les tâches sur les données de la mémoire partagée. Nous présentons dans cette section le mécanisme de typage des références aux données de la mémoire partagée qui traduit les accès qui seront effectués par chaque tâche et toute sa descendance.

Chaque tâche spécifie au moment de sa création les accès qui seront effectués sur la mémoire partagée au cours de son exécution ou lors de l'exécution de toute sa descendance. Les différentes données qui seront accédées sont spécifiées dans la liste des paramètres de la tâche et la description de ces droits d'accès est faite à l'aide d'un mécanisme de typage. La règle générale est qu'une tâche ne peut accéder, soit directement, soit via sa descendance, un objet pour lequel elle n'a pas déclaré l'accès correspondant (restriction des droits d'accès) : les tâches ne font pas d'effet de bord sur la mémoire. De plus, il est interdit[11] de passer deux fois la même donnée en paramètre à une tâche : il y aurait risque de création d'un cycle dans le graphe. Le typage des paramètres permet de vérifier facilement à la compilation la validité des accès effectués sur les données. Les différents types possibles sont répertoriés dans le tableau 3.2 page 66.

Le graphe sera construit en respectant l'algorithme de construction présenté au cours de la section précédente.

3.5.2 Sémantique des accès aux données

La sémantique des accès aux données est définie à partir des valeurs retournées lors des accès en lecture des versions par les tâches. Les versions accédées sont définies dans le graphe de flot de données construit lors de l'exécution, et les valeurs retournées lors des accès en lecture sont définies relativement à une exécution séquentielle de l'application.

[11] Mais la violation de cette règle n'est pas vérifiée par la bibliothèque. L'implantation est possible mais peut être coûteuse (il faut faire un parcours de la portion de graphe généré par la création de la tâche) et doit donc être activable à la demande. Pour ces raisons d'implantation nous avons préféré faire confiance à l'utilisateur.

droit / mode d'accès `Typage`	**Description**
lecture directe `Shared_r`	Accès restreint à la lecture seule (éventuellement concurrente) par la tâche et éventuellement sa descendance. Les tâches créées ne peuvent requérir qu'un accès en lecture seule sur cette donnée, c'est-à-dire les types `Shared_r` ou `Shared_rp`.
lecture différée `Shared_rp`	Accès en lecture seule par la descendance uniquement. La tâche courante ne peut accéder la donnée. Les tâches créées ne peuvent requérir qu'un accès en lecture seule sur cette donnée, c'est-à-dire les types `Shared_r` ou `Shared_rp`.
écriture directe `Shared_w`	Accès en écriture exclusive par la tâche courante uniquement. Les tâches créées ne peuvent requérir aucun accès sur cette donnée.
écriture différée `Shared_wp`	Accès en écriture seule par la descendance uniquement. La tâche courante ne peut accéder la donnée. Les tâches créées ne peuvent requérir qu'un accès en écriture sur cette donnée, c'est-à-dire les types `Shared_w` ou `Shared_wp`.
écriture directe `Shared_cw`	Accès en écriture concurrente (à sémantique d'accumulation) par la tâche et éventuellement sa descendance. Les tâches créées ne peuvent requérir qu'un accès en écriture (avec la même sémantique d'accumulation) sur la donnée, c'est-à-dire les types `Shared_cw` ou `Shared_cwp`.
écriture différée `Shared_cwp`	Accès en écriture seule (à sémantique d'accumulation) par la descendance uniquement. La tâche courante ne peut accéder la donnée. Les tâches créées ne peuvent requérir qu'un accès en écriture (avec la même sémantique d'accumulation) sur la donnée, c'est-à-dire les types `Shared_cw` ou `Shared_cwp`.
modification directe `Shared_r_w`	Accès en lecture et écriture de la donnée par la tâche courante uniquement. Cet accès, qui permet la mise à jour « en place » de la donnée, est exclusif. Les tâches créées ne peuvent requérir aucun accès sur cette donnée.
modification différée `Shared_rp_wp`	Accès en lecture et écriture par la descendance uniquement. Les tâches filles peuvent requérir tout type d'accès sur la donnée.

Tableau 3.2 *Types autorisés lors de la déclaration des paramètres d'une tâche.*
Les types sont composés d'un **droit d'accès** *lecture, écriture, accumulation et modification (*`r`, `w`, `cw`, `r_w`*) qui spécifie le type d'accès qui sera effectué par le sous graphe engendré par la tâche requérant cet accès, et d'un* **mode d'accès** *direct ou différé (*`p` *pour* postponed*) qui spécifie si la tâche requérant un droit d'accès va utiliser ce droit ou se contenter de le passer à sa descendance. Cette information permet de déduire les contraintes de localité des tâches puisque les données réellement accédées sont connues.*

La suite de cette section est organisée comme suit : nous définissons tout d'abord l'ordre d'exécution séquentielle, puis nous donnons la définition de la sémantique d'Athapascan-1 sous forme d'un théorème. Afin de prouver ce théorème, nous définissons la consistance associée à la mémoire partagée implantée dans Athapascan-1 et nous montrons que l'ordre de numérotation des versions est compatible avec l'ordre des écritures dans l'exécution séquentielle.

Définition 3 *L'***exécution séquentielle** *d'un programme Athapascan-1 consiste à remplacer toutes les créations de tâches par un appel direct au bloc d'instruction du programme et à remplacer les attributs* shared *par une référence directe sur la valeur de la donnée partagée.*

Nous montrons dans la proposition 6 page 69 que cet ordre d'exécution des instructions est un ordre valide, c'est-à-dire que les contraintes de précédence entre les tâches ne sont jamais violées.

Théorème 1 *La* **sémantique d'Athapascan-1** *est telle que pour toute exécution (éventuellement parallèle), la valeur retournée lors d'un accès en lecture à une donnée partagée est identique à la valeur lue lors de l'exécution séquentielle de l'application.*

La consistance associée à la mémoire partagée est définie par rapport au graphe d'exécution du programme. Cette approche est généralement appelée *programmer-centric* ou *computation-centric* dans [1, 52, 50] et se différencie de l'approche classique, dite *processor-centric*, où la consistance de la mémoire est définie par rapport aux dates d'exécution des instructions par les processeurs. La consistance associée à notre mémoire partagée est nommée *location-consistency* dans [52, 50] et est une généralisation de la consistance séquentielle définie par Lamport [73, 91] au sens où la consistance de la mémoire est définie pour chaque objet de la mémoire et non pour la mémoire dans son ensemble.

Définition 4 *Soit $G = (V, E)$ un graphe de flot de données correspondant à une exécution. Soit x un objet de la mémoire partagée et $V_x \subset V_v$ l'ensemble des versions associées à cet objet x. Soit $\prec$ l'ordre total sur V_x défini par les relations* prec *et* succ. *La fonction* « **dernière version produite** » *est l'unique fonction $\mathcal{W}_x : V_x \longmapsto V_x$ telle que pour toute version $v \in V_x$:*

1. *Si $\mathcal{W}_x(v) = v'$ alors, soit il existe $(t', v') \in E$ tel que t' a effectivement accédé x en écriture, soit $v' = v_{x,0}$ la valeur initiale associée à x.*
2. *$\mathcal{W}_x(v) \preceq v$.*
3. *Si il existe v' telle que $\mathcal{W}_x(v) \prec v' \preceq v$, alors pour tout $(t', v') \in E$, t' n'a pas accédé x en écriture.*

Définition 5 *La consistance associée à la mémoire partagée est telle que la valeur retournée par tout accès en lecture effectué par une tâche t sur une version v est égale à la valeur associée à la version $\mathcal{W}_{\mathrm{ref}(v)}(v)$. Cette valeur est le résultat de l'écriture des tâches t telles que $\left(t, \mathcal{W}_{\mathrm{ref}(v)}(v)\right) \in E$.*

Les tâches qui accèdent en écriture peuvent être multiples dans le cas d'une écriture avec sémantique d'accumulation. Cela correspond à la construction (C3) du graphe de flot de données.

Proposition 4 *L'implantation de la mémoire virtuelle partagée associée à Athapascan-1 respecte la consistance mémoire de la définition 5 page 67.*

Preuve. Soit une tâche t accédant en lecture une version v. Montrons que la valeur retournée lors de cet accès lecture correspond à la « dernière valeur écrite » $\mathcal{W}_{\mathrm{ref}(v)}(v)$. Si la tâche écrivain était sur la même version que la tâche lecteur t, alors la partie gauche du

« et » $(\wedge)$ de la relation 3.2 page 58 implique que la tâche écrivain est terminée, donc la valeur écrite peut être retournée. Cette valeur correspond bien à $\mathcal{W}_{\mathrm{ref}(v)}(v)$. Si par contre aucune tâche n'a écrit sur la version v, la partie droite du *« et »* de cette même relation implique que la version $\mathrm{pred}(v)$ précédant v est dans l'état prêt ou terminé : la valeur est donc disponible et c'est la valeur de cette version antérieure qui sera retournée. En remontant le long de la suite des versions, la valeur qui sera retournée sera bien celle correspondant à la « dernière valeur écrite ». Cette remontée constitue la détermination dynamique de cette dernière valeur écrite. □

Proposition 5 *La numérotation des versions est compatible avec l'ordre des accès effectués lors d'une exécution séquentielle de l'application.*

Preuve. Considérons une exécution séquentielle de l'application et une donnée partagée x. L'ordre de l'exécution est tel que chaque fois qu'une tâche est créée, son corps est immédiatement exécuté. Montrons que si, suivant cet ordre, un accès quelconque est effectué sur une version v, alors plus aucune version précédant v ne sera ultérieurement accédée. Dans les cas (C2) et (C3), si la tâche t accède la donnée après la création de t', elle le fera au travers de la même version v que t', donc n'accédera pas une version antérieure. Dans les cas (C4) et (C5), la tâche créée accède une version v' qui est placée juste avant celle que pourra accéder la mère. Tous les accès effectués par la mère seront donc effectués sur des versions ultérieures à celles accédées par t'. □

Nous pouvons désormais donner la preuve du théorème 1 page 67.

Preuve du théorème 1. Cette preuve est une conséquence directe de la proposition 5 page 68 et du modèle de consistance de la mémoire partagée, définition 4 page 67. Considérons en effet une tâche t effectuant une lecture sur un objet x au travers de la version v. La consistance de la mémoire partagée implique que la valeur lue est celle de la dernière version produite $\mathcal{W}_x(v)$. Cette dernière version produite correspond également, proposition 5 page 68, à la dernière valeur écrite lors d'une exécution séquentielle, donc à la valeur qui aurait été lue dans cette même exécution. □

3.5.3 Ordonnancement non préemptif et ordre de « référence »

La sémantique d'Athapascan-1 est donc lexicographique[12] (conséquence du théorème 1 page 67). Une telle sémantique semble requérir des synchronisations : une tâche semble devoir être interrompue pour attendre les valeurs produites par ses filles (ce qui est le cas pour le langage Cilk, dans lequel la synchronisation est explicite par utilisation de l'instruction `sync`). En fait, dans Athapascan-1, les restrictions d'accès présentées tableau 3.2 page 66 interdisent ce cas de figure, ce qui sera montré dans la propriété 2 page 69. Bien que lexicographique, la sémantique d'Athapascan-1 autorise donc une exécution non

[12] C'est-à-dire que les valeurs retournées par les accès en lecture peuvent être déterminées à partir d'une lecture du code source du programme.

préemptive des tâches. Parmi ces exécutions, celle ressemblant le plus à l'ordre séquentiel[13] des instructions définit un ordre total sur les tâches : l'ordre de « référence » qui consiste à exécuter le corps de la tâche mère dans son intégralité avant de passer, dans l'ordre où a eu lieu les créations, à l'exécution du corps des filles. Une analyse théorique nous a conduit à considérer des ordonnancements non préemptifs de tâches. Un intérêt pratique de tels ordonnancements est de ne pas requérir l'implantation d'un mécanisme de migration de tâches une fois qu'elles ont débuté leur exécution. En effet, sur une machine distribuée homogène, un tel mécanisme nécessite soit de fermer le contexte, soit de se placer au dessus d'une mémoire globale.

Nous définissons dans cette section comment les contraintes de précédence sont détectée dans Athapascan-1 puis nous prouvons que les tâches peuvent être exécutées sans synchronisation interne : leur exécution peut être vue comme atomique. Nous définissons enfin l'ordre total sur les tâches, l'ordre de « référence ».

Les contraintes de précédence entre tâches sont déduites des accès effectués sur les données partagées :

Définition 6 *Les* **contraintes de précédence** *entre les tâches sont déduites des accès effectués sur les données partagées : une tâche t accédant en lecture directe (paramètre typé* `Shared_r` *ou* `Shared_r_w`*) une donnée partagée au travers de la version v ne peut pas être exécutée avant que la tâche accédant en écriture la version $\mathcal{W}_{\mathrm{ref}(v)}(v)$ ne soit terminée.*

Ces contraintes de précédence définissent un ordre partiel $\prec_{G'}$, section 3.2.2 page 56, qui doit être respecté par toute exécution valide. Le graphe $G' = (V', E') \subset G$ est tel que $V' = V_t$ et $E' \subset E$ avec $(t, t') \in E'$ s'il existe (t, v) et (v, t') dans E avec (v, t') représentant un accès direct en lecture (paramètre typé `Shared_r` ou `Shared_r_w`).

Proposition 6 *L'ordre d'exécution séquentiel des instructions (les créations de tâches sont remplacées par les appels aux blocs d'instructions correspondants) est un ordre d'exécution valide.*

Preuve. Il suffit de montrer que les contraintes de précédence sont toujours résolues pour cet ordre, donc que lors de chaque création de tâche toutes les versions accédées en lecture directe sont prêtes, c'est-à-dire que pour chacune de ces versions v accédées en lecture toutes les tâches accédant $\mathcal{W}_{\mathrm{ref}(v)}(v)$ ont été terminées. Puisque $\mathcal{W}_{\mathrm{ref}(v)}(v) \prec v$, la proposition 5 page 68 et sa preuve impliquent que toutes les tâches accédant des versions antérieures à v ont été terminées. Donc que toutes les versions sont prêtes. Notons que cette exécution implique un mécanisme de préemption de la tâche mère. □

Propriété 2 *Dans le modèle de programmation Athapascan-1, les tâches peuvent être exécutées sans interruption (synchronisation).*

[13] Ordre qui nécessite un mécanisme de préemption car la tâche mère est bloquée jusqu'à la terminaison de l'exécution de la tâche fille qu'elle vient de créer. Cet ordre séquentiel ne peut donc pas convenir pour une exécution non préemtive des tâches.

Preuve. En considérant les créations autorisées présentées dans le tableau 3.2 page 66, une tâche qui possède un accès direct en lecture (le seul qui puisse introduire une contrainte de précédence, définition 6 page 69) possède un droit d'accès typé par `Shared_r` ou `Shared_r_w`. Dans le premier cas, seules des tâches accédant en lecture la donnée peuvent être créées ; comme ces tâches ne pouvent modifier la donnée, aucune contrainte de précédence ne peut être introduite. Dans le second cas, l'accès est exclusif : la tâche ne peut créer aucune tâche accédant cette donnée. Ceci assure que les tâches filles ne peuvent imposer aucune contrainte de précédence sur la tâche mère. Les tâches sont donc sans synchronisation : l'ensemble des instructions constituant leur corps peut être exécuté sans aucune interruption.

□

Définition 7 *L'ordre total $\prec_r$ sur les tâches et leurs instructions, dit* **« ordre de référence »** *est défini de la manière suivante (t, t', t'' et t_i représentent des tâches) :*

1. *Si l'exécution de t crée $t_1, t_2 \ldots, t_n$ dans cet ordre, alors $t_1 \prec_r t_2 \prec_r \ldots \prec_r t_n$ (l'ordre est lexicographique).*
2. *Si t crée t', alors $t \prec_r t'$.*
3. *Soient t_i et t_{i+1} deux tâches sœurs ($t_i \prec_r t_{i+1}$), alors pour toute tâche t' créée lors de l'exécution de la descendance de t_i, $t' \prec_r t_{i+1}$.*

La première clause signifie que l'ordre de création des tâches est conservé (l'ordre de création des tâches filles suit l'ordre d'exécution séquentiel des instructions du corps de la tâche t) ; la seconde clause précise qu'une mère précède toujours ses filles et la troisième qu'une tâche est précédée par la descendance de toute tâche qui la précède.

La figure 3.9 page 71 permet de comparer l'ordre de « référence », l'ordre séquentiel et un ordre d'exécution valide en considérant un exemple simple impliquant quatre tâches indépendantes (hormis les dépendances liées aux créations).

Théorème 2 *L'ordre de « référence » $\prec_r$ est un ordre d'exécution valide.*

Preuve. C'est une conséquence directe de la propriété 2 page 69. □

3.6 Bilan

Nous avons montré, dans ce chapitre, que toute instance d'application pouvait être caractérisée par un graphe de flot de données unique dans le cas où les tâches sont sans effet de bord sur les données en mémoire partagée : il est alors possible de définir la sémantique du langage à partir de ce graphe de flot de données associé à l'instance. Ce graphe peut être construit à la volée par développement au voisinage de la tâche créatrice, ce qui permet de réaliser des implantations du langage garantissant la sémantique.

Nous avons étudié la construction du graphe de flot de données et la sémantique des accès dans le cadre du langage d'Athapascan-1. La réalisation pratique repose sur la définition d'états et la détermination d'états associés aux nœuds du graphe. Si cette détermination est relativement simple dans le cadre d'une machine à mémoire partagée, elle

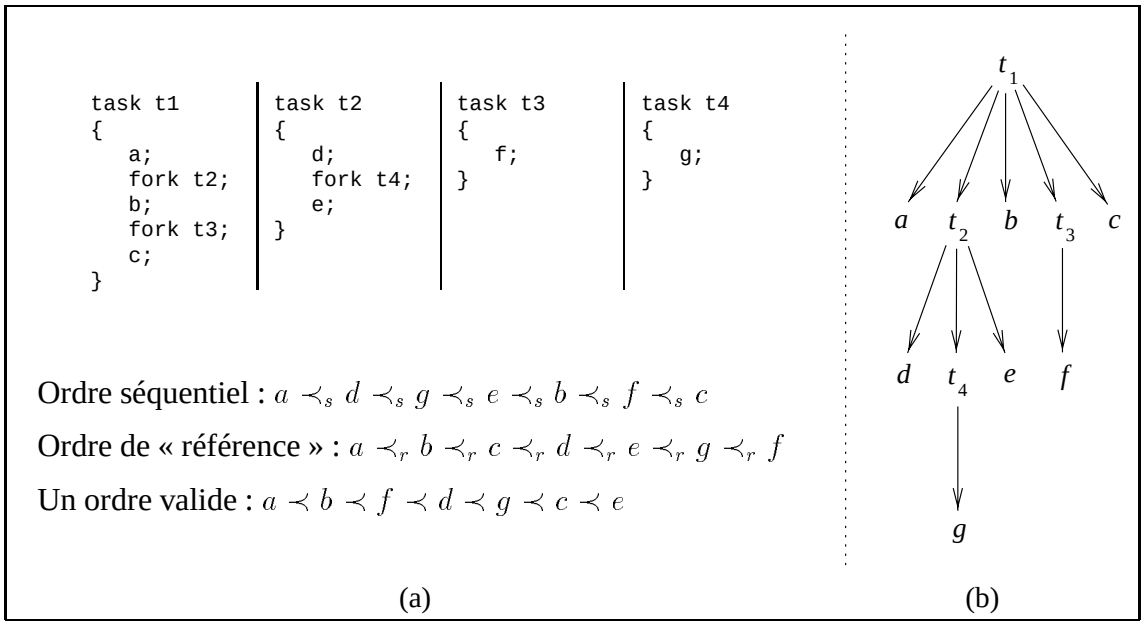


Figure 3.9 *Ordre de « référence ».*
En (a), est représenté le code de 4 tâches. La tâche t_1 crée les tâches t_2 et t_3 et t_2 crée la tâche t_4. Les tâches t_2, t_3 et t_4 sont indépendantes. Les instructions, autres que les créations de tâches sont représentées par les symboles a à g. En (b) est représenté le graphe représentant le contexte d'exécution des différentes instructions. Enfin, en (a) de nouveau, sont représentés trois ordres d'exécution valides des instructions. Le premier est l'ordre séquentiel, le second l'ordre de « référence » et le troisième est un ordre quelconque. Notons que les deux premiers sont définis de manière unique.

nécessite par contre la mise en œuvre d'un algorithme distribué dans le cadre d'une machine à processeurs faiblement couplés. L'étude de la gestion du graphe dans un cadre distribué est l'objet du chapitre suivant.

4

Algorithme distribué d'interprétation du flot de données

Nous étudions dans ce chapitre un algorithme distribué de détection de terminaison permettant la gestion du graphe de flot de données dans un environnement à mémoire distribuée. Les principales sections de ce chapitre traitent les points suivants :

- La **distribution du graphe** de flot de données parmi les différents sites de la machine (section 4.2 page 74).
- Une présentation de différents **algorithmes de terminaison** qui permettent de détecter la fin des références à un objet dans un environnement distribué (section 4.3 page 75).
- La description de l'algorithme **réactif** de terminaison utilisé dans Athapascan-1 (section 4.4 page 80).

4.1 Introduction

Athapascan-1 construit à la volée et de manière répartie un graphe de flot de données : en effet, l'architecture considérée n'est plus réduite à un seul processeur mais à un ensemble de **sites** possédant chacun un espace mémoire propre. Ces différents sites peuvent communiquer par échange de messages[1]. La gestion des états des nœuds de ce graphe nécessite la mise en œuvre d'un algorithme distribué de détection de terminaison, les changements d'états étant liés à des terminaisons du type plus de lecteur (donc possibilité de mise à jour ou de ramassage), plus d'écrivain (donc possibilité de lecture de la valeur résultant de l'écriture). Le calcul de ces états est présenté à la section 3.2.3.2 page 59. Les algorithmes de ce type, souvent appelés algorithmes de terminaison ou de gestion de transitions [4], sont à la base des applications de ramasse-miettes distribué [5]. Nous proposons dans ce chapitre une variante de ces stratégies plus adaptée aux contraintes de

[1]Il est à noter qu'un « site » peut posséder plusieurs processeurs physiques : c'est typiquement le cas pour les machines de type interconnexion de *SMP*. La notion importante est le caractère non partagé de la mémoire entre les sites : un processus au sens UNIX peut alors constituer un site à part entière.

réactivité de notre problème, la levée d'une contrainte de précédence devant être détectée le plus tôt possible.

Ce chapitre est organisé comme suit : il commence par la description de la distribution du graphe parmi les différents sites de la machine, puis il présente les algorithmes classiques de détection de terminaison dans un environnement distribué. Il se termine enfin par la description de l'algorithme réactif de terminaison utilisé dans Athapascan-1.

4.2 Distribution du graphe

La construction du graphe, présentée section 3.4 page 60 est basée sur une extension au voisinage du nœud responsable de l'évolution, ce qui permet d'envisager une construction distribuée de ce graphe. Chaque site contient alors un morceau de graphe. Le lien entre ces différents morceaux est effectué par réplication de certains nœuds du graphe. Les tâches étant exécutées une seule fois et les versions plusieurs (elles sont en effets accédées par plusieurs tâches sur différents sites, en écriture et en lecture), une décision naturelle consiste à répliquer les nœuds versions. Ainsi un nœud version sera répliqué sur tous les sites où un nœud tâche est présent et possède un lien vers cette version, comme illustré figure 4.1 page 74. Cette politique de réplication permet de garantir que tous les accès effectués (lors d'accès en lecture, en écriture ou lors de la création de tâches) par une tâche seront locaux. Les tâches, quant à elles, restent uniques dans le système et migrent (dynamiquement) sur décision de la politique d'ordonnancement. Autrement dit, chaque site qui possède une tâche ayant un lien avec cette version possède un réplicat de la version. Les actions de réplication, et les sites possédant de tels réplicats, sont donc déterminés de manière entièrement dynamique.

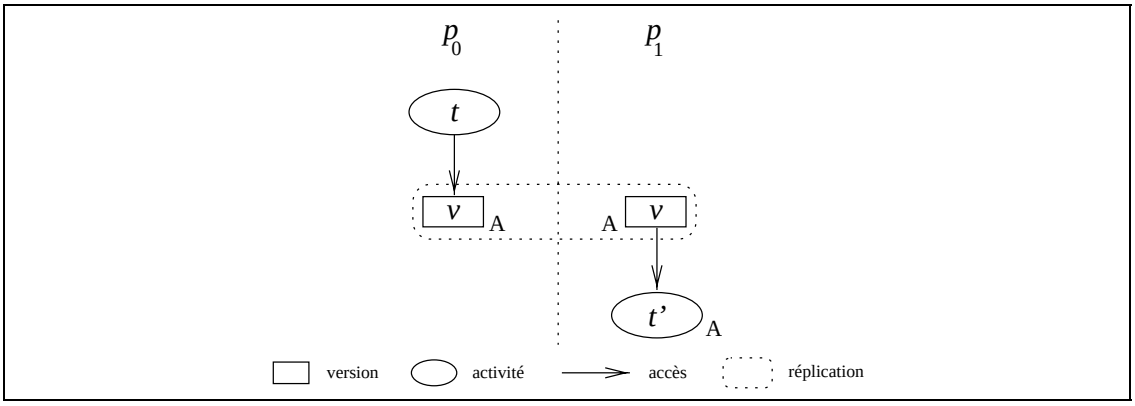


Figure 4.1 *Le graphe est distribué relativement aux nœuds versions.*
La tâche t accède la version v en écriture sur le site p_0, t' l'accède en lecture sur le site p_1. La version v se trouve donc répliquée sur les deux sites et un système de cohérence doit être mis en place pour déterminer son état. Ce système de réplication peut avoir lieu au niveau de chaque nœud de type version du graphe.

Remarque. Pour la décision de migration d'une tâche, un paramètre important de décision peut être l'existence ou non d'une tâche qui référence la donnée sur le site destinataire ; ou, en d'autres termes, si le nœud version est déjà répliqué sur ce site. En rajoutant

dans les nœuds versions des informations sur les sites possédant un *réplicat*, l'ordonnanceur peut alors résoudre les contraintes de localité en parcourant le graphe.

Le problème posé par cette distribution du graphe au niveau des nœuds versions est la détermination de leur état. Nous étudions dans la suite de ce chapitre des algorithmes permettant le calcul des états des nœuds qui sont répliqués.

4.3 Algorithmes de terminaison distribuée

Une même version v peut être référencée sur plusieurs sites distincts. Le calcul de l'état de cette version nécessite la détermination de la fin des références à cette version et devient alors une opération distribuée. Les algorithmes de ramasse-miettes en environnement distribué effectuent cette détermination, mais ces détections soit nécessitent un nombre élevé de messages, soit ne sont pas synchrones avec l'application manipulant les références ce qui induit des problèmes de réactivité.

Nous présentons tout d'abord le problème général de la détection d'une terminaison distribuée pour détailler ensuite l'adaptation de ces techniques à notre situation particulière.

Les algorithmes de détection de la terminaison des accès à une donnée constituent la base des algorithmes de ramasse-miettes (garbage collection). Ces algorithmes sont composés de deux parties : l'une est de décider si un objet est ou n'est pas accessible, l'autre est la destruction de cet objet. La phase de détection consiste à décider pour tout objet s'il existe encore dans le système des tâches pouvant, via une suite de déréférenciations, y accéder. La multiplicité des propositions, dont de nombreuses incomplètes, traduit la complexité de ce problème [87]. Deux techniques sont essentiellement utilisées : le comptage de références et le traçage.

Les méthodes basées sur le comptage de références consistent à mémoriser dans un compteur le nombre de références qui accèdent une version. Elles ont l'avantage d'être réactives, au sens où un objet est déclaré inaccessible lors de la destruction de la dernière référence y accédant (l'instant où le compteur passe à zéro). Ces méthodes sont très sensibles aux pertes et duplications de messages mais leur principal désavantage est que les cycles de références ne sont pas détectés (des objets en pratique inaccessibles ne seront pas identifiés comme tels). Une variante, résistante aux pertes ou inversions de messages est l'énumération de référence : chaque site envoie périodiquement une liste des réfèrences qu'il possède. Par exemple l'algorithme basé sur les SSP chains [97, 86] fonctionne ainsi.

Les techniques basées sur un traçage permettent d'identifier les cycles de références inaccessibles. Ces algorithmes sont composés de deux phases [77] : une première, dite de marquage, où tous les objets atteignables à partir de racines locales sont marqués comme étant valides, puis une seconde, dite de balayage, où tous les objets non marqués valides sont détruits. Le principal désavantage de ce type de technique est la latence introduite entre le moment où un objet devient inaccessible et le moment où il va être effectivement identifié comme tel, la phase de marquage intervenant soit périodiquement, soit sur un signal (lors d'un manque de mémoire par exemple). De plus, une synchronisation est

nécessaire entre la phase de marquage et la phase de balayage afin d'être assuré que tous les messages en transit de la première phase sont parvenus à destination. Sans quoi, certains objets peuvent être détruits à tort.

Nous ne nous intéresserons dans la suite qu'aux techniques basées sur le comptage de références car la réactivité est primordiale dans notre situation.

4.3.1 Modèle

Nous considérons une machine à mémoire distribuée constituée d'espaces mémoire distincts (un par site) interagissant par échanges de messages. Ce système est supposé sûr, c'est-à-dire que les sites ne tombent pas en panne, que les messages ne se perdent pas, ne se dupliquent pas et qu'ils arrivent dans le même ordre que celui de l'émission (FIFO point-à-point).

Nous faisons la distinction entre les références locales (les pointeurs classiques par exemple) qui font référence à un objet situé dans l'espace mémoire du site courant, et les références globales qui font, elles, référence à un objet situé dans l'espace mémoire d'un site distant.

Références globales Comme conséquence des migrations de tâches d'un site à un autre, un même objet peut être utilisé par plusieurs tâches distribuées au sein de la machine. Cet objet, dit public, sera donc référencé dans plusieurs espaces mémoire différents par l'intermédiaire de références globales. Une référence globale pointe localement vers un point de sortie qui fait référence à un point d'entrée distant (*i.e.* qui contient un pointeur vers un objet public, comme illustré figure 4.2 page 77). Le site dont la mémoire contient l'objet public est appelé site propriétaire, ceux possédant une référence globale sur cet objet sont appelés des sites clients.

Chaque espace mémoire possède une racine locale constituée des références (locales ou globales) vers les objets qui sont directement accessibles par les tâches : ce sont les références contenues dans les registres, dans la pile ou dans les zones de mémoire statiques. Un objet est dit inaccessible s'il n'est relié ni à la racine locale par une suite de références, ni pointé localement par un point d'entrée (car un point d'entrée implique l'existence d'un point de sortie, donc l'existence d'une référence globale sur cet objet).

Création Une référence globale est créée lorsque le site propriétaire d'une donnée envoie vers un site récepteur une référence sur cette donnée : le site récepteur devient un client du site propriétaire et la donnée devient publique. La création n'est pas nécessairement unique, c'est-à-dire que plusieurs références globales peuvent référencer le même objet public : il y a alors dans ce cas coexistence de plusieurs points d'entrée. La figure 4.3 page 77 illustre les deux étapes constituant la création d'une référence globale.

Duplication Une référence globale est dupliquée lorsqu'un site client émet cette référence vers un site autre que le site propriétaire. Le fonctionnement, représenté figure 4.4

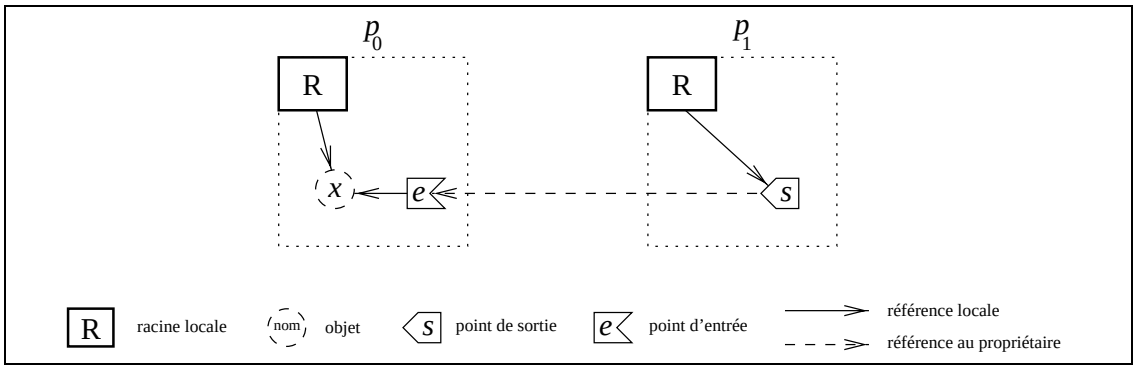


Figure 4.2 *Référence globale sur x.*
p_0 est le site propriétaire, p_1 un site client. Une référence globale est composée d'un pointeur local vers un point de sortie s qui fait référence à un point d'entrée e, situé sur le site propriétaire. Ce point d'entrée contient un pointeur local vers l'objet public x.

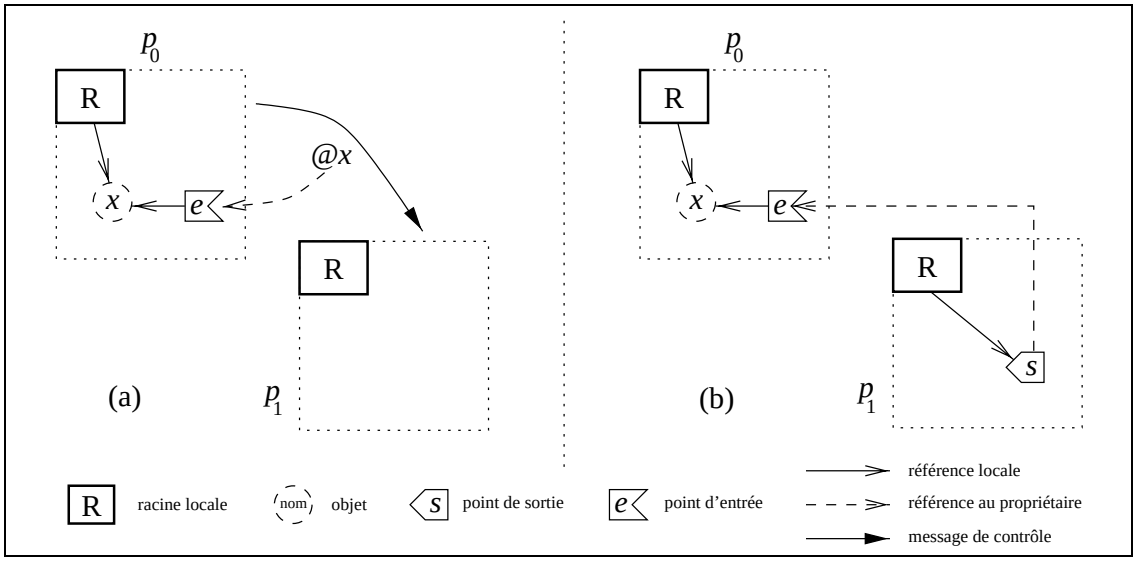


Figure 4.3 *Création d'une référence globale sur x.*
En (a) le site propriétaire p_0 crée un point d'entrée e qui pointe sur x puis émet un message de création de référence au site destinataire p_1. En (b), à la réception de ce message, p_1 crée un point de sortie s qui référence le point d'entrée e.

page 78, est similaire à celui de la création sauf qu'aucun point d'entrée n'est créé. Il est à noter que le site propriétaire n'intervient aucunement dans ce processus.

Destruction Lorsque la référence globale n'est plus accessible sur un site client, le point de sortie est détruit et un message est émis vers le site propriétaire qui pourra alors éventuellement détruire le point d'entrée associé à cette référence globale, ce qui est illustré figure 4.5 page 78. Les dates auxquelles ont lieu ces destructions dépendent des systèmes de ramasse-miettes utilisés : ce peut être immédiatement, ou plus tard, suite à un événement.

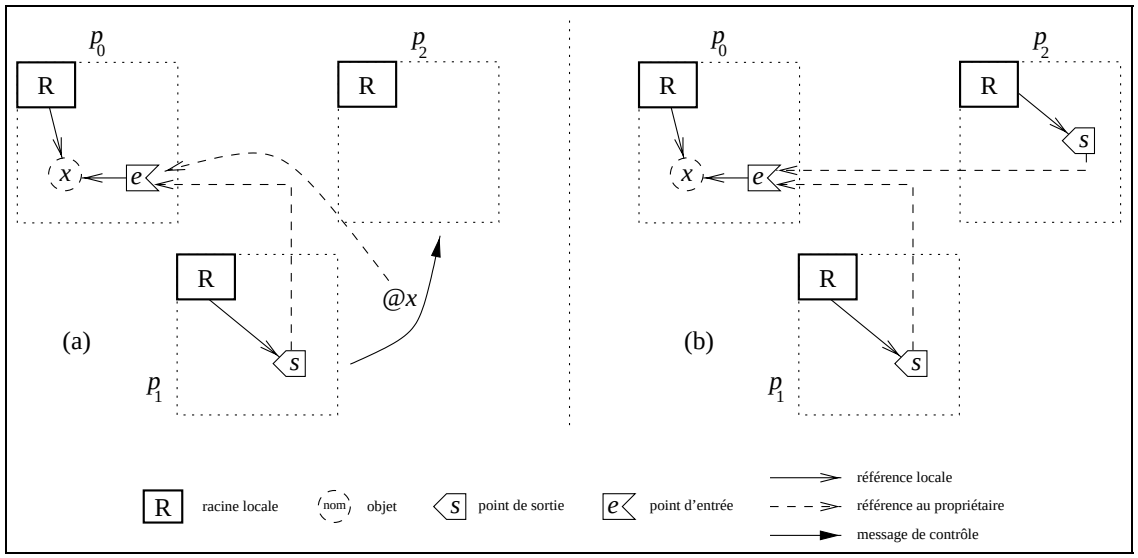


Figure 4.4 *Duplication d'une référence globale sur x.*
En (a) le site client p_1 émet un message de duplication de la référence à p_2. En (b), à la réception de ce message, p_2 crée un point de sortie s qui référence le point d'entrée e ; il devient donc un site client du site propriétaire p_0.

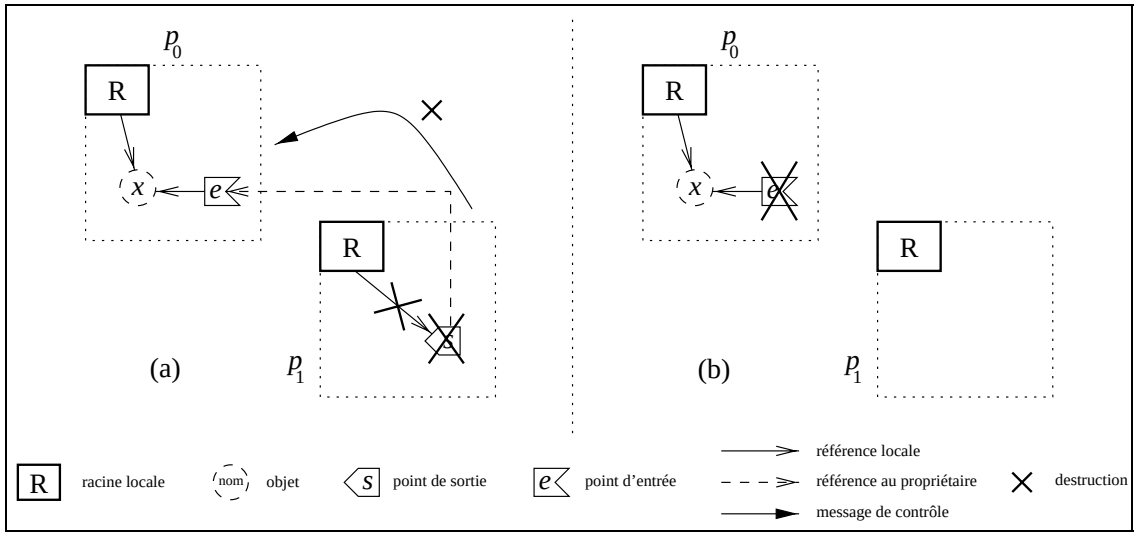


Figure 4.5 *Destruction d'une référence globale sur x.*
En (a) le site client p_1 détruit le point de sortie s puis émet un message de destruction de la référence au site propriétaire. En (b), à la réception de ce message, p_0 peut détruire le point d'entrée e.

4.3.2 Ramasse miettes pour une machine à mémoire distribuée

Comptage de références distribué L'adaptation naïve du comptage de références utilisé pour une mémoire centralisée consiste à émettre un message de contrôle au propriétaire de l'objet chaque fois qu'une référence est dupliquée ou détruite. La suppression des problèmes liés aux accès concurrents au compteur (*race-conditions*) nécessite l'intro-

duction d'un mécanisme d'acquittement qui est pénalisant puisqu'introduisant un surcoût de communication. La figure 4.6 page 79 illustre cette situation de concurrence, entre un message d'incrémentation et un message de décrémentation du compteur. La concurrence entre ces deux messages peut mener le site propriétaire à une conclusion erronée.

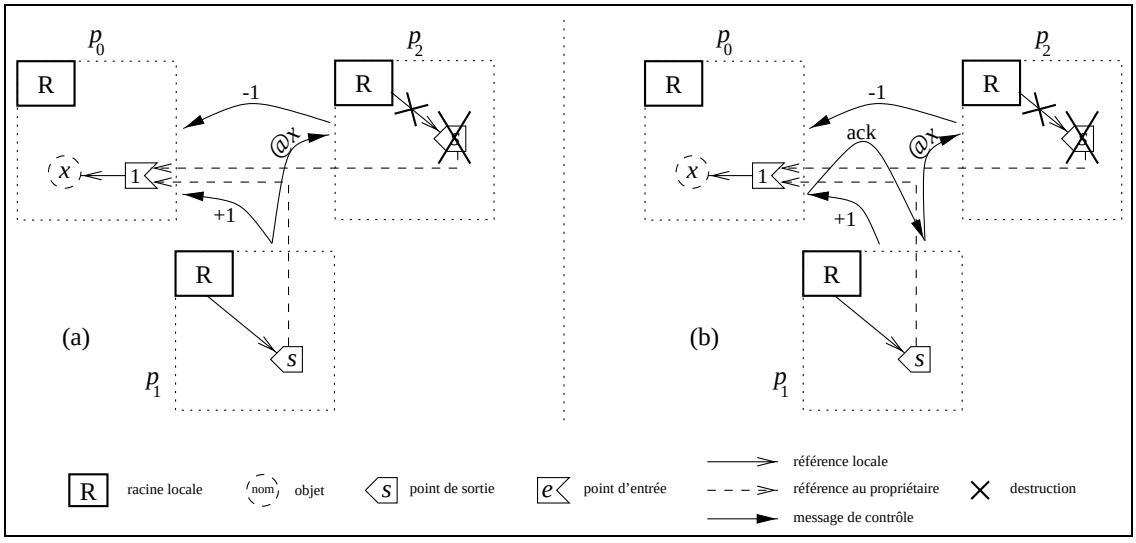


Figure 4.6 *Concurrence entre les messages d'incrémentation et de décrémentation. Le site p_1, client de p_0, duplique la référence globale vers p_2 en prévenant le site propriétaire p_0. Le site p_2, après utilisation de cette référence, la détruit en prévenant p_0. En (a), si le message de décrément arrive avant celui d'incrément, l'objet x sera détruit alors qu'il est toujours référencé sur p_1. En (b), le message de création $@x$ ne sera émis qu'après réception du message d'acquittement de l'incrémentation, ce qui a pour effet de sérialiser les actions sur le compteur.*

Comptage de références pondéré Une alternative permettant d'éviter les messages d'incrémentations et les problèmes de concurrence qui y sont liés est le comptage de références pondéré [8, 105]. Chaque référence globale possède deux poids : un poids total et un poids partiel. Le point d'entrée possède le poids total et chaque point de sortie possède un poids partiel non nul. L'égalité (4.1) suivante est en permanence maintenue pour tout objet x, s_x et e_x désignant respectivement un point de sortie et le point d'entrée associé à x :

$$\sum_{s_x} poids\_partiel(s_x) \quad = \quad poids\_total(e_x) \tag{4.1}$$

Lors de la création de la référence globale, le poids total du point d'entrée et le poids partiel du point de sortie sont initialisés à une même valeur non nulle, figure 4.7(a). L'égalité (4.1) est donc vérifiée.

Lors de la duplication de la référence globale d'un site p_i vers un site p_j, le poids partiel du point de sortie de p_i est diminué de moitié et le point de sortie créé sur p_j prend pour poids partiel la partie retranchée. La figure 4.7(b) illustre cette duplication. L'égalité (4.1) est donc maintenue.

Lors de la destruction d'un point de sortie, un message est émis vers le site propriétaire en spécifiant le poids partiel de cet ancien point de sortie. Cette valeur sera retranchée au poids total du point d'entrée de la référence globale, ce qui est illustré figure 4.7(c). L'égalité (4.1) est donc maintenue.

La terminaison de la référence globale est décidée lorsque le poids total du point d'entrée passe à 0, l'égalité (4.1) impliquant alors l'inexistence de poids partiels (car ils sont tous non nuls).

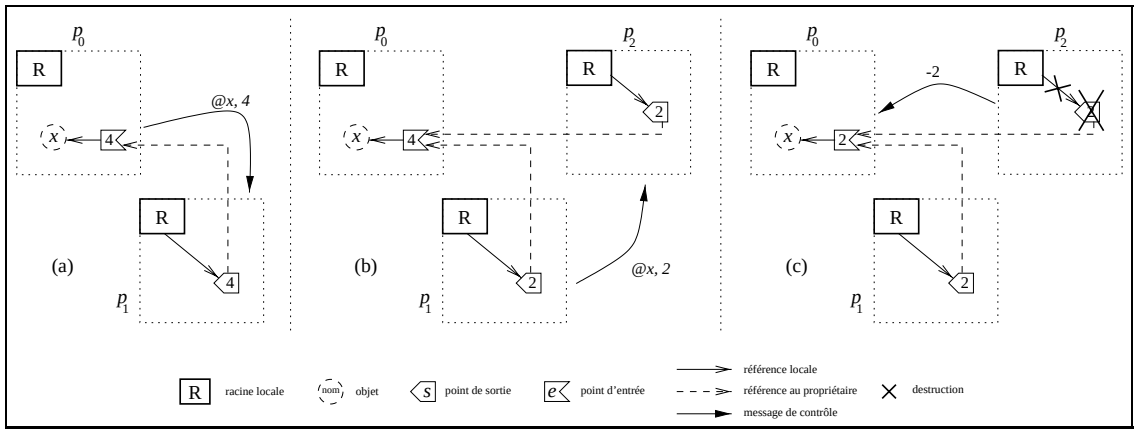


Figure 4.7 *Comptage de référence pondéré.*
En (a), création d'une référence globale avec le poids total et un poids partiel valant 4. En (b), duplication de la référence, le poids partiel de p_1 est réparti équitablement entre p_1 et p_2. En (c), terminaison de la référence sur le site p_2 avec émission du poids partiel détruit ; cette valeur est retranchée du poids total contenu dans le point d'entrée.

Le principal désavantage de cette technique est qu'une référence de poids initial égal à 2^k ne peut être répliquée que k fois. Une première technique levant cette limitation est de demander au site propriétaire, lorsque le poids partiel atteind 1, un nouveau crédit qui sera ajouté et au poids total, et au poids partiel. Une seconde technique [105] consiste en la création d'un nouveau point d'entrée sur le site posant problème. Le poids total de ce point d'entrée sera fourni comme poids partiel de la référence dupliquée. L'inconvénient de cette méthode est l'indirection sur la référence qui est créée. Dans les deux cas le nombre de communications est donc accru.

4.4 Un algorithme réactif de terminaison

Le comptage de références permet de détecter rapidement les terminaisons globales, mais cela se fait en distribué au prix d'un nombre de messages accru, dû soit à des messages d'acquittement, soit à des indirections.

Nous nous proposons de modifier l'adaptation naïve afin de supprimer les messages d'acquittement. Pour cela, nous allons regrouper les messages d'incrémentation et de décrémentation du compteur au niveau de chaque site et faire en sorte que l'ordre de réception des messages de sites différents puisse être quelconque : il n'y a alors plus de contrainte de concurrence sur l'accès au compteur.

4.4.1 Situation

Comme illustré figure 4.8 page 81, nous avons un ensemble $\mathcal{T}$ de tâches qui accèdent en concurrence une même version. Nous devons détecter l'instant où toutes ces tâches concurrentes sont terminées afin de pouvoir « libérer » cette version qui introduit certainement des contraintes de précédence sur d'autres tâches. Il est donc primordial de détecter au plus tôt la terminaison de cet ensemble de tâches. Une tâche peut créer de nouvelles tâches, ses filles, se terminer ou se déplacer vers un autre site. Ni le nombre de migrations, ni le nombre de filles ne sont limités. L'ensemble $\mathcal{T}$ évolue donc dynamiquement ; lors de la création de version il possède au moins une tâche. La terminaison de la version correspond au moment où $\mathcal{T}$ devient vide.

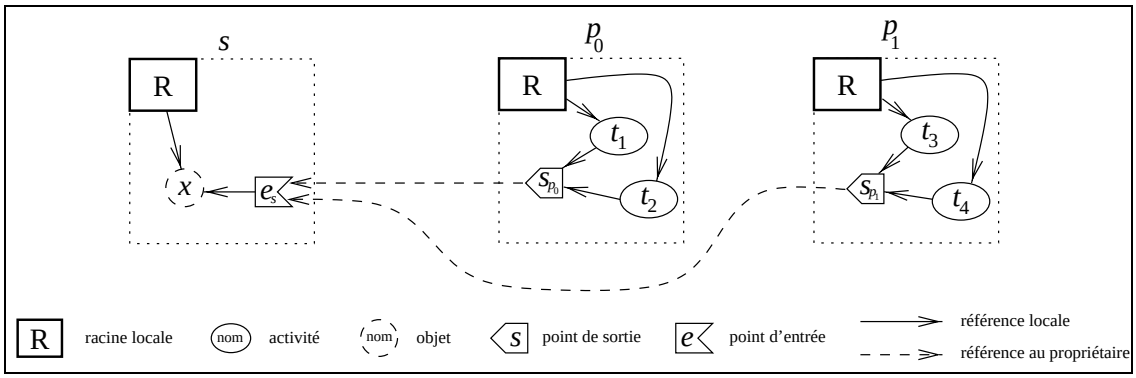


Figure 4.8 *Problème de la terminaison globale.*
Plusieurs tâches (t_1, t_2, t_3, t_4) possèdent une référence globale sur un objet x, propriété d'un site s. Les tâches présentes sur un même site p_i accèdent le même point de sortie s_{p_i}. Le problème est de détecter lorsque le point d'entrée e_s n'est plus référencé par aucun point de sortie, c'est-à-dire la fin des tâches.

Le site propriétaire de la donnée sera nommé s. Ce site peut *a priori* être l'un des sites banalisés p_i, mais pour plus de clarté nous le considérons dans la suite de ce chapitre comme différent.

Localement, sur un site donné p, nous effectuons un comptage classique de références afin de déterminer si le point de sortie s_p est encore référencé. Lorsque le compteur passe à 0, on dit que le site p a atteint une terminaison locale. Cette terminaison locale correspond à la fin de toutes les tâches présentes sur ce site. Un site peut donc être dans deux états différents vis-à-vis d'une donnée x : soit dans un état d'attente A où aucune tâche référençant x n'est présente sur ce site, soit, à l'opposé, un état d'exécution E où des tâches sont présentes. Les évolutions possibles de cet état sont représentées figure 4.9 page 82.

Le problème posé est de détecter, sur le site propriétaire s, la terminaison **globale**, l'ensemble $\mathcal{T}$ devient vide (c'est-à-dire le moment où toutes les tâches t_i ont été terminées).

Nous ne représenterons plus par la suite les références des points de sorties s_p au point d'entrée e_s. L'évolution du temps est traduit verticalement : le temps s'écoule, sur un site donné, de haut en bas.

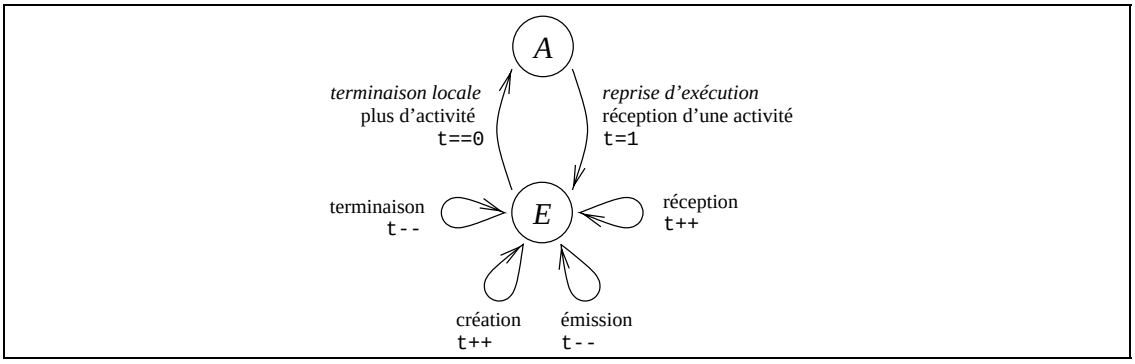


Figure 4.9 *Évolution de l'état d'un site.*
La variable `t` *représente le nombre de tâches présentes sur le site, l'état A un état d'attente et E un état d'exécution. Initialement, tous les sites sont dans l'état d'attente A, sauf le site p_0 sur lequel la référence a été créée et pour lequel* `t=1`*.*

4.4.2 Algorithme

Afin d'illustrer la présentation de cet algorithme nous considérons 4 tâches t_0, t_1, t_2 et t_3 accédant toutes la même donnée x telles que pour tout i, t_{i+1} soit la fille de t_i. L'exécution de ces tâches a lieu sur 2 processeurs, p_0 et p_1. Le site propriétaire s connait initialement l'existence de la tâche t_0 et du point de sortie s_{p_0}.

Nous devons décider d'un état de terminaison globale à partir des états de terminaisons locales : chaque fois qu'une terminaison locale a lieu, un message est envoyé au site propriétaire s.

Nous présentons d'abord de manière intuitive sur l'exemple considéré trois détections erronées : ces exemples permettent d'introduire les informations suffisantes pour assurer une détection correcte de la terminaison globale. Nous décrivons ensuite l'algorithme basé sur ces informations puis nous concluons par une preuve de sa correction.

4.4.2.1 Quelles informations sont suffisantes pour la détection de la terminaison ?

Une première idée est de décider la terminaison globale lorsque le site propriétaire s a reçu un message de terminaison locale de la part de **tous** les sites.

Cette information ne permet pas au site propriétaire s de décider de la terminaison globale. Considérons la situation, illustrée figure 4.10 page 83, où les quatre tâches t_i se terminent sur p_0. Seul p_0 va émettre un signal de terminaison vers s. Le site propriétaire va donc attendre, en vain, un message de p_1.

Exiger de p_1 d'émettre un message de terminaison introduit l'émission d'un message qui n'aurait pas existé dans une programmation directe : il faut donc savoir quels sont les sites qui participent. À chaque indication de terminaison locale est donc associée une liste de sites où la référence globale a été dupliquée, suite à l'émission de tâches vers ces sites.

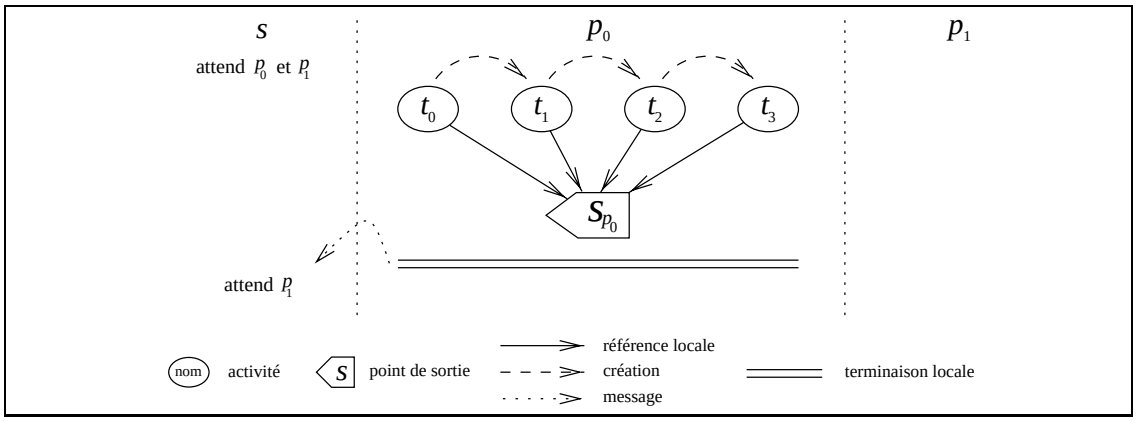


Figure 4.10 *Algorithme : une première idée fausse.*
Le site propriétaire s attend un message de terminaison locale de tous les sites. p_1 ne passant jamais de l'état E à l'état A n'enverra pas de message vers s. La terminaison globale ne sera donc jamais détectée.

Cette information n'est cependant pas suffisante puisque chaque site peut éventuellement émettre plusieurs messages de terminaisons locales. La situation illustrée figure 4.11 page 83 en montre un exemple : t_0 est exécutée sur p_0, t_1 sur p_1 ; t_2, créée sur p_1, est émise pour exécution vers p_0. À la terminaison de t_0, p_0 envoie un message de terminaison locale vers s ; à la terminaison de t_1, p_1 envoie un message de terminaison locale vers s. Ayant reçu deux messages de terminaison locale, le site propriétaire conclut une terminaison globale alors que les tâches t_2 et t_3 sont toujours présentes sur le site p_0.

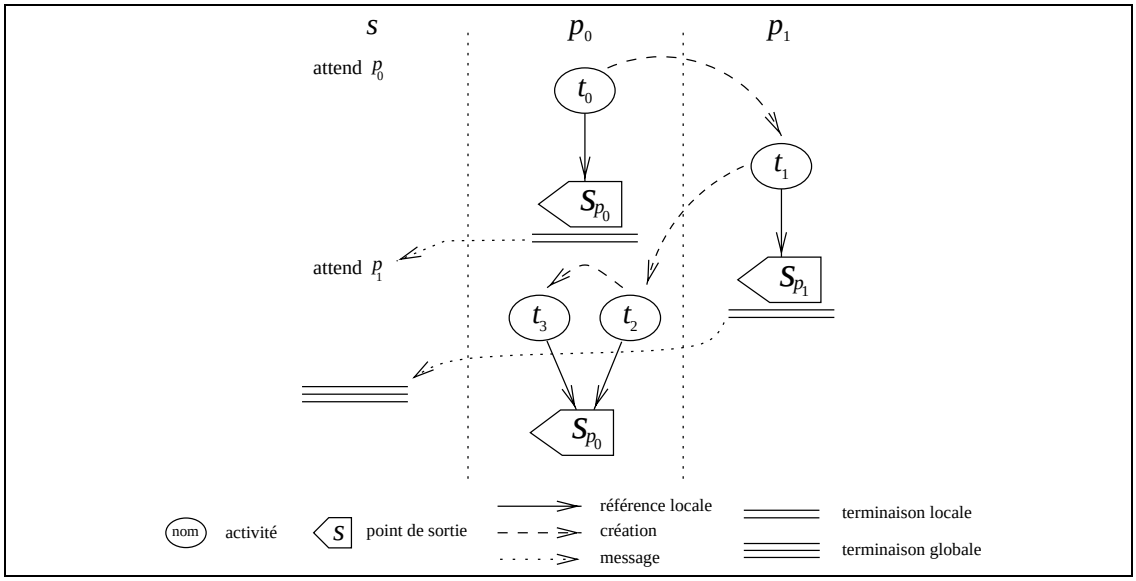


Figure 4.11 *Algorithme : une seconde idée fausse.*
Le site propriétaire s attend un message de terminaison locale de la part de tous les sites clients. Lors d'une terminaison locale, le site émet la liste des sites vers lesquels la référence a été dupliquée. La situation présentée montre qu'un même site peut émettre plusieurs messages de terminaison locale.

Si l'on avait compté les tâches présentes dans le système, le site propriétaire s aurait su lors de la terminaison locale de p_1 que 3 tâches avaient été créées (t_0, t_1, t_2) et seulement

deux terminées (t_0, t_1) : il n'aurait pas décidé de la terminaison globale.

Cependant, ajouter cette information n'est toujours pas suffisant. Considérons en effet la situation illustrée figure 4.12 page 84 : t_0 est exécutée sur p_0, t_1 sur p_1 ; t_2, créée sur p_1, est émise pour exécution vers p_0 ; t_3, créée sur p_0, est émise vers p_1 pour exécution. À la terminaison de t_0, p_0 envoie un message de terminaison locale vers s en indiquant qu'une tâche (t_1) a été créée et une tâche (t_0) a été terminée sur ce site ; à la terminaison de t_1 et t_3, p_1 envoie un message de terminaison vers s en indiquant qu'une tâche a été créée (t_2) et deux tâches (t_1, t_3) ont été terminées. Puisque le nombre de tâches créées égale le nombre de tâches terminées, le site propriétaire détecte une terminaison globale alors que la tâche t_2 est toujours présente sur p_0.

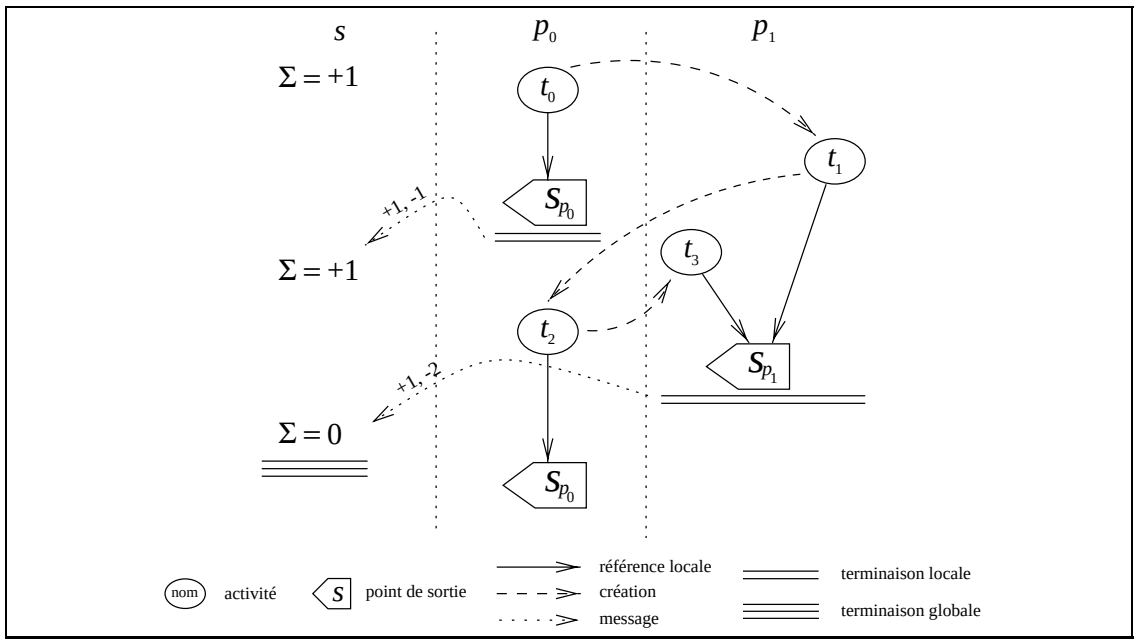


Figure 4.12 *Algorithme : une troisième idée fausse.*
Lors de chaque terminaison locale, le nombre de tâches créées et le nombre de tâches détruites sont envoyés au site propriétaire. Cette information n'est pas suffisante, la terminaison globale étant annoncée alors que t_2 était toujours en tâche sur p_0.

Il est donc nécessaire de distinguer les sites vers lesquels des tâches ont été émises.

4.4.2.2 Algorithme de terminaison proposé

La détection de la terminaison nécessite donc de dénombrer pour chaque site de la machine les tâches émises vers ce site et le nombre de tâches reçues par ce site. La terminaison globale sera prononcée si pour tous les sites ces deux nombres sont égaux. Ces nombres valent en particulier zéro pour les sites n'intervenant pas dans le calcul. L'algorithme que nous proposons est donc le suivant :

– Sur chacun des sites d'exécution :

  - `r` est un entier qui représente le nombre de tâches reçues, `e[]` est un tableau d'entiers où `e[`i`]` représente le nombre de tâches émises vers le site i. Initialement, tout est à 0.
  - Lors de la réception d'une tâche : `r++`
  - Lors de l'émission d'une tâche vers le site i : `e[`i`]++`
  - Lors de la terminaison locale : émission vers s de `r` et `e[]`, puis réinitialisation à 0 de ces deux variables.
- Sur le site propriétaire s :
  - `d[]` est un tableau d'entiers où `d[`i`]` représente la différence entre le nombre de tâches émises vers le site i et le nombre de tâches reçues par ce site i. Initialement, ce tableau est à 0. Le contenu de ce tableau est réactualisé lors de chaque réception de message de terminaison locale.
  - Lors de la réception d'un message de terminaison locale de la part du site i : `d[`i`]-=r`, `d[`j`]+=e[`j`]` pour tout $j \neq i$. Si `d[`i`]` est égal à 0 pour tout i, alors la terminaison globale est annoncée.

Basé sur des compteurs, cet algorithme satisfait les conditions de réactivité souhaitées. La section suivante prouve sa correction.

4.4.3 Preuve de correction

L'algorithme précédent détecte, sur le site propriétaire, l'instant à partir duquel, pour tout site p, le compteur `d[`p`]` associé à la transition est nul.

Prouvons que cet algorithme détecte bien la terminaison globale, c'est-à-dire prouvons que les deux propositions suivantes sont équivalentes :

(1) La terminaison globale est atteinte.

(4) $\forall p, \texttt{d[}p\texttt{]} = 0$.

Afin montrer $(1) \iff (4)$, nous introduisons un second algorithme de détection de terminaison distribué. Cet algorithme est identique à celui présenté à la section 4.4.2.2 page 84, sauf en ce qui concerne le **contenu** des messages de terminaisons locales. En effet, ces messages contiennent la liste des identificateurs de tâches reçues (r'_i par le processeur i) ou émises ($e'_i[j]$ émises par i vers j) au lieu de ne contenir que leur nombre (nous supposons que les tâches sont identifiées de manière unique et globale). Ces listes sont maintenues tant que la terminaison globale n'a pas été détectée, c'est-à-dire que ces listes ne sont pas remises à vide (contrairement aux données de l'algorithme présenté section 4.4.2.2 page 84) lors des événements de terminaison locale. De même le site propriétaire s maintient deux listes r_p et e_p qui contiennent les identificateurs des tâches reçues par p et émises vers p ($r_p = \cup e'_p$ et $e_p = \cup_{i \neq p} e'_i[p]$, union effectuée au niveau du site propriétaire s lors de chaque réception de terminaison locale). Ces listes peuvent contenir plusieurs fois le même identificateur, si la tâche correspondante a été reçue plusieurs fois sur le même site. De même, une tâche reçue par un processeur peut être réémise vers un autre. Il est à noter que l'ordre de réception des messages sur le site propriétaire n'a pas d'importance,

puisque les listes émises ont toujours une taille croissante : tout l'historique du site est donc contenu dans chaque message. Ainsi, lorsqu'un message de terminaison locale arrive sur le site propriétaire s, ce dernier ne tient compte du message que si la taille des listes reçues est supérieure à la taille des listes actuellement possédées.

Nous introduisons alors les deux propriétés suivantes au niveau du site s ($||l||$ représente la longueur de la liste l) :

(2) $\Big(\forall p, r_p = e_p\Big)$ où r_p désigne la liste des tâches reçues par p, et e_p la liste des tâches émises vers p par les autres processeurs.

(3) $\Big(\forall p, ||r_p|| = ||e_p||\Big)$.

Afin de montrer la correction de l'algorithme de la section 4.4.2.2 page 84 nous montrons la suite d'implications suivante : $(1) \Longrightarrow (2)$, $(2) \Longrightarrow (1)$, $(2) \Longrightarrow (3)$, $(3) \Longrightarrow (2)$ et enfin $(3) \iff (4)$ ce qui conclu la démonstration.

Le site propriétaire s considère que la racine[2] de l'arbre de création (ici t_0) a été émise vers le processeur où elle a été créée (ici p_0). Le responsable de cette émission est purement fictif : cela revient à dire que `d[0]=-1` afin de forcer l'attente d'un message de terminaison locale issu du site p_0.

Démonstration de $(1) \Longrightarrow (2)$ Montrons la contraposée $\neg(2) \Longrightarrow \neg(1)$. Si la proposition (2) n'est pas vérifiée, alors il existe un site p sur lequel $r_p \neq e_p$, donc $(r_p \setminus e_p) \cup (e_p \setminus r_p) \neq \emptyset$. S'il existe $t \in (r_p \setminus e_p)$, alors le site q qui a émis cette tâche n'a pas atteint un état de terminaison locale, sinon cette tâche aurait été déclarée émise vers p : la terminaison globale n'est donc pas atteinte. S'il existe $t \in (e_p \setminus r_p)$, alors le processeur p n'a pas encore reçu la tâche t et va donc être réactivé : la terminaison globale n'est pas atteinte. La proposition (1) ne peut donc pas être vérifiée. □

Démonstration de $(2) \Longrightarrow (1)$ La démonstration est effectuée par l'absurde. Supposons que (2) est vérifiée mais que la terminaison globale n'est pas atteinte, c'est-à-dire qu'il existe un site p qui n'est pas dans l'état $\perp$; soit alors t une des tâches qui le maintient dans l'état d'exécution E. Soit $\mathcal{E}$ l'ensemble des tâches qui sont connues du site propriétaire s (c'est l'union des listes des tâches reçues, ou émises). Comme (2) est vraie, toutes les tâches de $\mathcal{E}$ sont terminées puisque le site propriétaire a été prévenu de leur réception après terminaison locale du site récepteur, donc en particulier après exécution de toutes les tâches reçues.

Puisque t est en cours d'exécution, $t \notin \mathcal{E}$; soit alors $t'_0, ..., t'_n$ la généalogie de t telle que $\forall i, t'_i$ soit la mère de t'_{i+1}, $t'_n = t$, $t'_0 \in \mathcal{E}$ et $t'_{i\geq 1} \notin \mathcal{E}$. Cette généalogie existe forcément car la racine t_0 de l'arbre de création appartient forcément à $\mathcal{E}$. Soit q le site de réception puis d'exécution de la tâche t'_0.

Les tâches $t'_{i\geq 1}$ n'ont pas pu être émises à partir du site q, sans quoi elles auraient fait partie de la liste des tâches émises vers un processeur lors de la terminaison locale de q

[2]Cette tâche correspond à la tâche qui a créé la version. Les créations de versions résultent des règles (C1), (C4) et (C5) de construction du graphe présentées à la section 3.4 page 60.

(message qui a précisé que t_0 avait été reçu par q) et donc auraient fait partie de $\mathcal{E}$. Elles ont donc nécessairement toutes été exécutées sur q, ce qui contredit la supposition que t soit encore en cours d'exécution ; la terminaison globale est donc atteinte. □

Démonstration de $(2) \Longrightarrow (3)$ Si sur chaque processeur les listes sont égales, alors nécessairement elles ont la même taille. □

Démonstration de $(3) \Longrightarrow (2)$ La démonstration est par l'absurde en supposant l'existence d'une horloge globale. Pour tout site p, soit d_p la date d'émission de la dernière terminaison locale reçue ($d_p = 0$ si aucun message de terminaison locale n'a été émis).

Si la proposition (2) n'est pas vérifiée, alors il existe un site p tel que p a reçu une tâche t qui n'a pas été déclarée émise (puisque $||r_p|| = ||e_p||$ et $r_p \neq e_p$). Si le message n'a pas été déclaré émis, c'est qu'il a été émis par un processeur q après d_q, avec forcément $d_q < d_p$. Soient $t_0, ..., t_n = t$ les ancêtres de t, avec t_0 la racine de l'arbre de création. Nécessairement, t_0 a été émise avant d_0 et t_n après d_q. Il existe donc deux sites r_1, r_2 et une tâche t_i tels que t_i a été émise par r_1 avant d_{r_1} et reçue par r_2 après d_{r_2}, sinon t_0 aurait été émise après d_0. Donc r_2 possède une tâche déclarée émise mais non reçue. Donc $(\exists p / r_p \neq e_p) \Rightarrow (\exists r / (r_r \neq e_r \wedge d_r < d_p))$. La figure 4.13 page 87 illustre la présence de ce site r. Soit p' tel que $d_{p'}$ soit minimum ; alors $(\exists p / r_p \neq e_p) \Rightarrow (r_{p'} \neq e_{p'})$. Donc p' a reçu un message qui n'a été déclaré émis par aucun autre site. Or tous les autres sites ont déclaré toutes les tâches émises avant leur terminaison locale, donc avant $d_{p'}$; ce qui contredit l'existence de p', donc de p. □

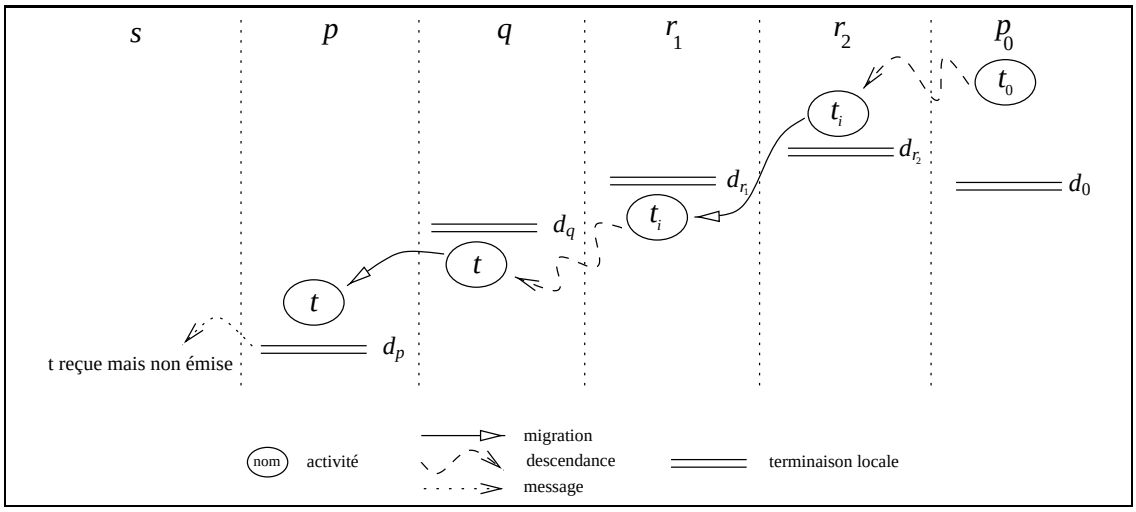


Figure 4.13 *Illustration de la démonstration de* $(3) \Longrightarrow (2)$.
S'il existe une tâche t reçue mais non déclarée émise, alors nécessairement il existe une tâche t_i déclarée émise mais non reçue par r_1 avec $d_{r_1} < d_p$ (les d_i représentent des dates de déterminaisons locales).

Démonstration de $(3) \iff (4)$ `d[`p`]` représente la différence entre le nombre de tâches émises vers p et le nombre de tâches reçues par p. La différence avec $||r_p|| -$

$||e_p||$ est le mode de calcul : $||r_p|| - ||e_p||$ englobe tout le passé du processeur p, tandis que `d[`p`]` est calculé de manière incrémentale au niveau du site propriétaire (lors de chaque réception de message de terminaison locale). L'ordre de réception des messages de terminaison locale ne peut désormais plus être négligé, comme le montre la figure 4.14 page 88.

Supposons donc que l'ordre des messages de terminaison locale vers s est respecté pour tout site. Soit m un message de terminaison locale issu d'un site p. La contribution du message m à l'accroissement des listes r_p et $e_{q \neq p}$ ne peut avoir déjà été prise en compte, car tous les messages reçus par s et provenant de p ont forcément été envoyés avant m : ils ne pouvaient donc contenir son information. La taille de ces contributions égale donc la valeur des variables `r` et `e[]` contenues dans le message m ; ainsi $||r_p|| - ||e_p|| =$ `d[`p`]` pour tout p. □

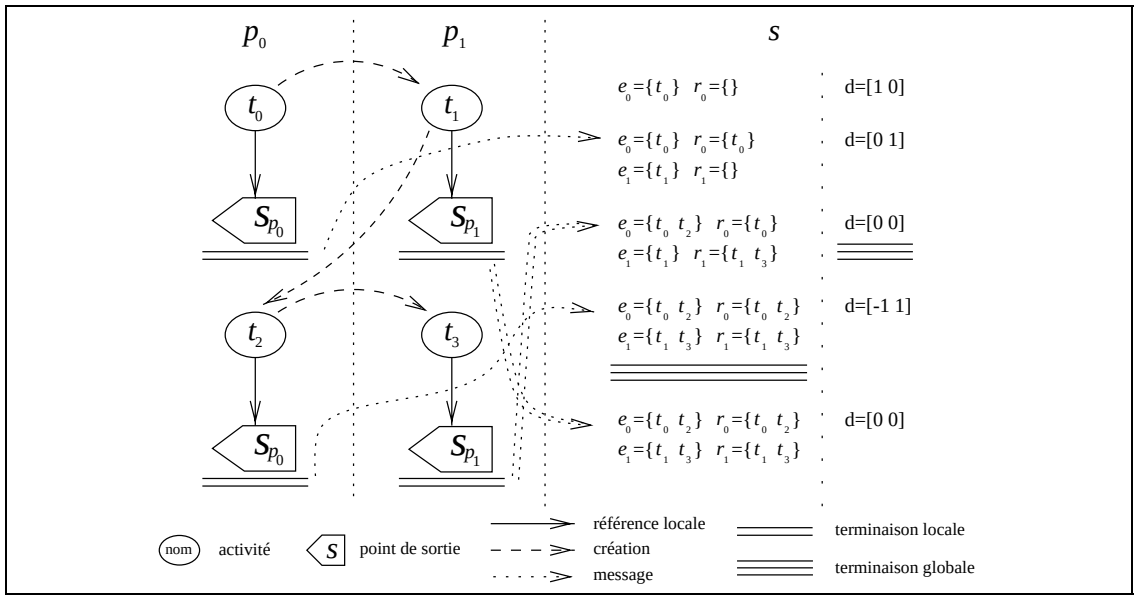


Figure 4.14 *La proposition* (4) *ne tolère pas l'inversion de messages de terminaison locale. L'inversion dans l'ordre d'arrivée des messages de terminaisons locales émis par le site* p_1 *implique une détection de terminaison globale eronnée, la tâche* t_2 *étant dans ce cas potentiellement encore en cours d'exécution sur le site* p_0*. Les deux messages incriminés sont représentés doublés sur la figure.*

L'algorithme proposé est donc correct. Nous le comparons dans la section suivante à l'algorithme de comptage de références pondéré.

4.5 Comparaison

Nous nous proposons dans cette section de comparer brièvement l'algorithme que nous utilisons avec celui de comptage de référence pondéré (présenté à la section 4.3.2 page 79) en ce qui concerne la taille en bit des informations nécessaires à leur mise en œuvre :

– Dans le cas du comptage de référence pondéré, seul le poids est maintenu. Considérons ce poids stocké sur k bits.

– Dans le cas de l'algorithme que nous utilisons, la taille des informations maintenues au maximum[3] est égale à $2pl+l$ bits, p étant le nombre de processeurs de la machine, les entiers étant codés sur l bits.

À première vue, le second algorithme requiert un volume mémoire bien plus important que le premier. Cependant, les performances ne sont pas identiques : dans le premier cas, chaque fois qu'une tâche est migrée le poids du site émetteur est divisé par 2, tandis que dans le second un des entiers est augmenté d'1. Il est donc possible de migrer au maximum k tâches dans le premier cas, 2^l dans le second. Si l'on affecte la même taille en bits aux deux algorithmes, comparer le nombre de tâches qui peuvent être migrées à partir d'un même site revient à comparer $(2p+1)l$ et 2^l, ce qui est à l'avantage du second. En prenant des valeurs classiques pour p et l, $p = l = 32$, il est possible de migrer 2080 tâches dans le premier cas et plus de 4 milliards dans le second. Si l'on veut migrer le même nombre de tâches[4], l'algorithme de comptage pondéré générera 2 millions de messages pour augmenter la valeur de son poids. La taille des messages de contrôle est certes $2p+1$ fois supérieure dans le second cas, mais les messages étant petits, le surcoût n'est pas significatif[5].

Cependant, l'algorithme de comptage de références distribué est plus simple à mettre en œuvre. Une solution meilleure serait sans doute de faire une combinaison de ces deux algorithmes : utiliser le premier tant que les tâches ne migrent pas trop, puis, si les tâches migrent beaucoup, utiliser la seconde stratégie de détection de terminaison.

Nous présentons dans le chapitre suivant une implantation de l'algorithme réactif présenté.

[3] Ce qui correspond au cas où le site de référence coïncide avec un site de calcul, donc quand les tableaux `e[]` et `d[]` sont possédés simultanément en plus de la variable `r`.

[4] Ce qui est assurément irréaliste, les resources mémoires du processeur seront sûrement dépassées depuis longtemps avec un tel nombre de tâches...

[5] L'essentiel du temps de communication étant passé dans la latence du réseau, le débit de celui-ci n'intervenant que pour une faible part. Classiquement, la durée de communication de 100 octets n'est que le double de celle d'1 octet.

5

Implantation du flot de données dans Athapascan-1

Ce chapitre détaille l'implantation de la gestion distribuée du graphe de flot de données dans Athapascan-1. Les principales sections de ce chapitre traitent les points suivants :

- La **représentation locale** des portions de graphe sur chaque site (section 5.1.2 page 93).
- La **gestion distribuée** du flot de données (section 5.1.3 page 95).
- L'**analyse du coût** de cette gestion distribuée et certaines optimisations envisageables (section 5.2 page 98).
- Une **évaluation** du coût de construction de graphe de flot de données (section 5.3 page 105).

5.1 Implantation

Dans ce chapitre nous nous focalisons sur l'implantation distribuée de l'interprétation du flot de données. Cette implantation est basée sur une extension de l'ordre de « référence » décrit précédemment : les accès à la mémoire ont pour conséquence d'introduire des relations de précédence entre les tâches. Ces contraintes sont contrôlées par le système qui retarde l'exécution d'une tâche tant que toutes les versions des données accédées ne sont pas disponibles. L'exécution ayant lieu sur une machine à mémoire distribuée, ces contraintes de précédence doivent être résolues de manière distribuée et les données de la mémoire partagée doivent éventuellement migrer (ou être copiées) vers les sites hébergeant des tâches possédant un droit d'accès en lecture.

Les rôles de l'implantation vis-a-vis de la mémoire sont alors les suivants :

- Détecter puis résoudre les contraintes de précédence entre les tâches. Les tâches étant réparties parmi les processeurs, ces détections sont distribuées.
- Déplacer les données vers les sites où elles vont être accédées (ou générer de nouvelles copies).
- Détruire les données qui ne sont plus référencées.

Ces trois objectifs sont atteints grâce à la construction et l'analyse distribuée du graphe des accès aux données, graphe reliant les tâches aux données partagées. Chaque nœud de ce graphe possède un état propre : il est soit sous contrainte de précédence, soit prêt, soit en cours d'exécution, soit terminé comme résumé dans le tableau 3.1 page 58.

Il est possible de détecter qu'une donnée d n'est plus accessible en examinant l'ensemble des tâches possédant un quelconque droit d'accès (lecture ou écriture) sur cette donnée. Lorsque cet ensemble n'est composé que de tâches terminées, alors la donnée peut être détruite. Cette détection est à la base du mécanisme de ramassage distribué des zones mémoire correspondant à des versions de données qui ne seront plus référencées.

Nous montrons dans ce chapitre que cette gestion peut être effectuée de manière efficace : chaque création de tâche et de donnée peut être faite avec un surcoût borné et sans communication, et le nombre de messages nécessaires à la détermination des états est égal au nombre de messages d'échange de données qu'aurait nécessité, en respectant le même placement, une implantation directement basée sur des processus communicants par échange de messages.

5.1.1 Nommage global des objets

Les objets suivants sont susceptibles de migrer et doivent être identifiés de manière globale :

– Les versions ou leurs réplicats.

– Les tâches.

– Les valeurs des données de la mémoire partagée.

Les tâches et les versions sont identifiées globalement par un couple d'entiers. Le premier membre est le numéro du site de création et le second un numéro, unique, de séquence de création sur ce site. Deux objets différents ont alors nécessairement deux identificateurs différents : soit par le numéro de site s'ils ont été créés initialement sur des sites différents, soit par le numéro de séquence si la création a eu lieu sur le même site.

Les versions sont des objets non typés, leur transmission ne pose donc pas de problème. Par contre, les tâches et les valeurs des données sont typées et cette information de type doit pouvoir être transmise : en effet, il ne suffit pas de savoir que l'on va recevoir une tâche, encore faut-il savoir quels sont les types de ses paramètres, quelle est la fonction associée, *etc*. Cette information de type est un identificateur, unique et global. Cet identificateur est un entier, donnée membre statique[1] du type, qui est déterminé lors de la phase d'initialisation du programme : l'unicité de l'identification repose sur le fait que l'ordre d'initialisation des objets statiques est identique[2] sur tous les sites. On utilise un mécanisme de typage C++ (classes patrons, ou *template*) qui permet d'assurer l'unicité de chaque identificateur pour un type donné.

Outre ces identificateurs, il faut être capable de transmettre les valeurs associées aux versions, dans le cas d'un accès distant en lecture. Tous les types de ces valeurs sont donc

[1] Au sens C++ du terme.

[2] Cela suppose que le **même** code exécutable est utilisé sur tous les processeurs. L'ordre d'initialisation, fixé par le compilateur, sera donc identique sur tous les sites.

supposés **« communicables »**, c'est-à-dire possédant deux opérateurs d'emballage et de déballage présentés dans la figure 5.1 page 93 sur un exemple de type représentant un couple de réels.

```
struct couple { double x, y; };

a1_ostream& operator<<( a1_ostream& out, const couple& c )
{
   out <<~c.x <<~c.y;
   return out;
}

a1_istream& operator>>( a1_istream& in, couple& c )
{
   in~>> c.x~>> c.y;
   return in;
}

...
   Shared< couple > x;
...
```

Figure 5.1 *Définition d'un type communicable en Athapascan-1.*
Un type communicable *est un type possédant les opérateurs d'emballage et de déballage sur les flots* `a1_istream` *et* `a1_ostream` *de la bibliothèque. Ces opérateurs permettent de déplacer les objets de la mémoire partagée d'un processeur à un autre. Seuls ces types peuvent être utilisés lors de la déclaration d'un objet en mémoire partagée. Les types de base C++, ainsi que tous les types de la STL définis sur des types communicables, sont par défaut communicables.*

Ces informations de type permettent de déplacer facilement les objets devant migrer. Nous étudions dans la section suivante les représentations locales de ces objets.

5.1.2 Représentation locale du graphe

Le graphe étant global et distribué, ses nœuds sont représentés par des objets alloués dans les tas des processeurs. Les objets correspondant aux tâches sont appelés clôtures ; les objets représentant les versions sont appelés des transitions. Les arêtes, quant à elles, sont représentées par des pointeurs locaux entres ces deux types d'objets.

5.1.2.1 Nœud tâche : une clôture

Une tâche est représentée par un objet, appelé **clôture**, constituée du nom de la fonction constituant son corps et de la liste de ses paramètres effectifs. Un paramètre est soit de type valeur et dans ce cas la clôture en contient une copie, soit de type référence sur une donnée en mémoire partagée. Une telle référence est un pointeur local vers une transition qui représente la version associée accédée de la donnée partagée. Le terme de clôture traduit le fait qu'un tel objet contient toute l'information nécessaire à son exécution : le corps et les paramètres de la fonction. Cet objet n'est pas répliqué sur les différents sites : quand une tâche migre d'un site vers un autre, la clôture est détruite du site d'origine et reconstruite sur le site destinataire. La figure 5.2(b) contient une illustration de cet objet.

5.1.2.2 Nœud version : une transition

Une version est représentée par un objet, appelé **transition**, qui centralise toutes les références locales que peuvent avoir les tâches sur cette version. Ces références sont séparées en deux listes distinctes : celles représentant un accès en écriture et celles représentant un accès en lecture. Le terme transition a été choisi pour sa connotation de « barrière », *i.e.* de synchronisation liée au passage de son état de A vers E : comme il n'y a plus de tâche écrivain, c'est à ce moment précis que les contraintes de précédence sont levées.

Chaque transition est identifiée de manière unique et globale, ce qui permet d'assurer qu'il n'y aura pas deux réplicats différents d'une même version sur un site donné. Outre les listes des tâches accèdant localement la version, la transition possède les données `r` et `e[]` de l'algorithme présenté section 4.4 page 80, une référence vers le site propriétaire (son numéro de site) et un pointeur, éventuellement nul, vers la donnée physique associée localement à la version. Un des sites constituera le site propriétaire et aura donc en plus la donnée `d[]`. La figure 5.2(b) contient une illustration de cet objet.

5.1.2.3 Arête : deux pointeurs locaux

Les arêtes qui lient les deux types de nœuds et qui représentent les accès des clôtures sur les transitions sont codées par un couple de pointeurs locaux : l'un de la clôture vers la transition, l'autre de la transition vers la clôture. Ce lien bi-directionnel permet un libre parcours de cette portion locale du graphe par l'ordonnanceur ou le système. Ce couple de pointeurs est illustré figure 5.2(b).

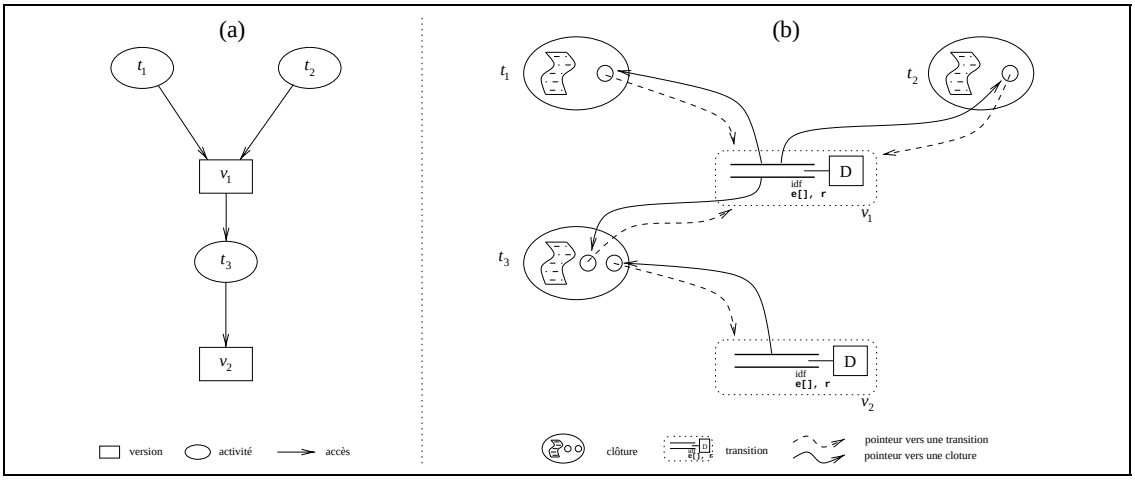


Figure 5.2 *Éléments d'implantation constituant le graphe des accès aux données.*
Cette figure représente, en (a), le graphe des accès aux données de trois tâches t_1, t_2, t_3 sur deux versions v_1, v_2 et, en (b), l'implantation de ce graphe. Les nœuds transitions et clôtures sont des objets alloués dans le tas, tandis que les arêtes sont des couples de pointeurs. Une clôture est constituée du corps de la tâche et des paramètres pointant vers des versions de données partagées. Une transition est constituée des listes de tâches accédant en lecture et en écriture la version représentée, ainsi que des compteurs nécessaires à l'algorithme de terminaison globale. La transition possède de plus un pointeur local éventuellement nul vers une donnée D. Cette donnée représente une des copies de la valeur associée à la version.

5.1.3 Évolution distribuée des états des nœuds du graphe

Les clôtures et les transitions ont un état qu'il faut calculer. Nous présentons dans cette section les méthodes utilisées puis les problèmes spécifiques liés à la distribution : migration d'une tâche, mouvement des données et choix du site propriétaire s. La figure 5.3 page 95 reprend la figure 3.3 page 59 représentant les changements possibles d'états des nœuds du graphe. La différence entre ces deux figures concerne l'état des versions. Dans la figure 5.3 nous représentons l'état d'un **réplicat** et non pas l'état d'une version. Ces états coïncident toujours sauf qu'un réplicat peut être dans l'état local L qui signifie que la donnée est présente localement sur le site de réplication. Alors, nécessairement, si un réplicat est dans l'état L, alors forcément tous les autres sont dans l'état P ou L. Cet état local peut être vu comme un raffinement de l'état P.

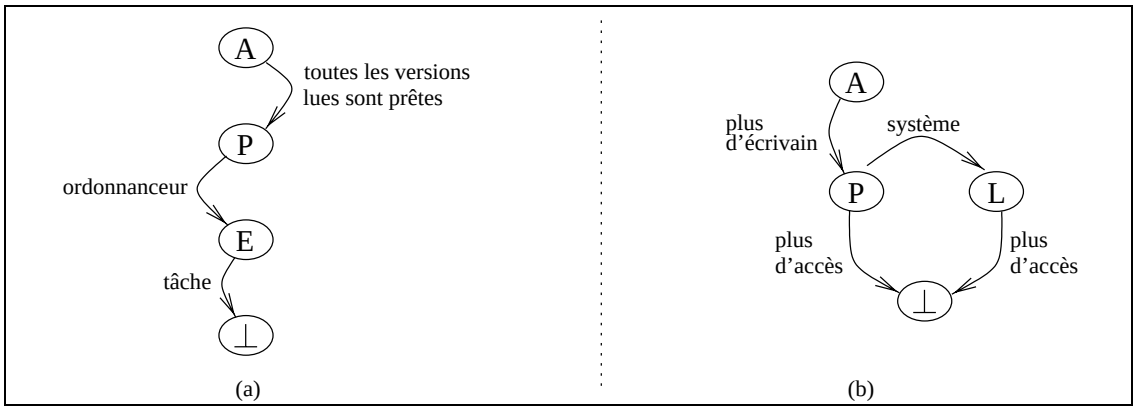


Figure 5.3 *Changements d'états des nœuds du graphe (rappel de la figure 3.3 page 59).*
En (a) sont représentées les évolutions possibles d'une tâche, et en (b) celles d'un réplicat. Le changement d'état d'un nœud du graphe est soit dû à une intervention extérieure, le système, soit la conséquence de l'évolution des états d'autres nœuds.

5.1.3.1 État d'une clôture

L'état d'une clôture est calculé localement :

- À partir des états des transitions qui sont accédées en lecture, à l'aide de la relation (3.1), pour le passage de l'état A à l'état P.
- Sur décision de l'ordonnanceur pour le passage de l'état P à l'état E.
- Sur décision du système pour les passages entre les états E et B.
- Lors de l'exécution de la dernière instruction pour le passage de l'état E à l'état $\perp$.

5.1.3.2 État d'une transition

Le calcul de l'état d'une transition est une opération distribuée basée sur les relations (3.2) et (3.3) définies page 58.

Chaque transition va être le lieu de deux phénomènes de terminaison globale, terminaison détectée à l'aide de l'algorithme réactif présenté section 4.4 page 80 :

– Le premier lors de la terminaison de toutes les tâches écrivains (l'état passe de A à l'état P, relation (3.2)).
– Le second lors de la terminaison des tâches lecteurs (l'état passe de E à l'état $\perp$, relation (3.3)).

Lorsque le site propriétaire a déterminé un nouvel état pour la transition, les réplicats doivent en être avertis. Une solution simple est d'effectuer une diffusion du nouvel état vers tous les sites possédant un réplicat de la transition. Cependant, tous les réplicats n'ont pas forcément besoin[3] d'être avertis de ce changement d'état et de nombreuses optimisations sont envisageables ; quelques-unes seront décrites à la section 5.2.3 page 103.

5.1.3.3 Migration d'une tâche

Lors de la migration d'une tâche t vers un site p, pour toute version v telle que (t, v) ou $(v, t) \in E$, il y a incrémentation de `e[`p`]` puis retrait de t de la liste des tâches référençant localement la version. Si cette liste devient vide, la transition émet un message de terminaison locale vers le site propriétaire s.

Lors de la réception sur le site p, s'il n'existe pas de transition associée à la version, il y a création d'un réplicat (la seule connaissance de l'identificateur unique et du site propriétaire est suffisante). La tâche t est ensuite insérée dans les listes des tâches accédant localement la version et l'entier `r` est incrémenté.

5.1.3.4 Synthèse de la donnée

Lorsque la transition passe à l'état P, l'ensemble des tâches possédant un droit en écriture est nécessairement vide. S'il existe des tâches accédant la donnée en lecture, il convient de la synthétiser.

Dans le cas d'une écriture concurrente par accumulation (droit d'accès `cw`), il faut collecter les valeurs accumulées localement. Chaque site ayant participé à l'écriture doit donc envoyer à un site collecteur la donnée qu'il possède afin que ce site puisse procéder aux accumulations finales. La fonction à utiliser pour cette réduction est codée dans le type de la donnée partagée et fournie par l'utilisateur (voir la section A.4 page 161).

Dans le cas d'une écriture concurrente sans accumulation (droit d'accès `w`), le système doit choisir la valeur associée à la donnée entre celle résultant de l'écriture effective de la tâche, et celle provenant du lien de précédence entre les versions. En effet, conformément à la consistance de la mémoire partagée (définition 4 page 67), si la tâche n'a pas exécuté d'écriture la donnée sera celle provenant de la version précédente ; sinon ce sera le résultat de l'écriture (la donnée précédente sera dans ce cas ignorée). Ces liens de précédence sont considérés comme des écrivains ou des lecteurs.

Dans l'implantation actuelle, à chaque message de terminaison locale est associé la donnée qui a été produite sur le site concerné. Le site collecteur est donc tout simplement le site propriétaire.

[3] Par exemple, un réplicat ne possédant aucune tâche l'accédant en lecture peut, sans que cela pose de problème, ne pas être averti du passage de la transition vers l'état P.

5.1.3.5 Mouvement de la donnée

Lorsqu'une version est dans l'état P, cela signifie que la donnée associée est prête ; les tâches lecteurs peuvent donc débuter leur exécution (on suppose que ces tâches ont déjà été placées sur leur site d'exécution par l'ordonnanceur). Se pose alors le problème de l'utilisation de cette donnée. Faut il :

– Déplacer la donnée vers les sites où vont avoir lieu des lectures ?

– Accéder la donnée à distance ?

L'accès à distance, même s'il offre des avantages (non réplication de la donnée, localisation aisée du site la possédant : le site propriétaire *a priori*) possède le désavantage d'engendrer potentiellement un grand nombre de communications. En effet, si plusieurs tâches lecteurs sont sur un même site, toutes vont faire un appel au site propriétaire lors de leur accès en lecture. Une solution est alors de considérer la transition comme implantant un mécanisme de cache : lors de la première demande de lecture, la donnée est recopiée du site propriétaire vers le site lecteur, puis conservée sur ce site. Les lectures suivantes accèderont ainsi localement la donnée associée à la version.

L'émission, *a priori*, de la donnée vers tous les sites lecteurs permet d'anticiper ce système de cache de la donnée dans le réplicat local et donc d'économiser un aller et retour qui était prévisible. Nous pouvons alors profiter du message annonçant la terminaison globale des écrivains pour envoyer la donnée.

Le problème est de réussir à constituer, au niveau du site propriétaire, l'ensemble des sites possédant des tâches lecteurs. Les messages de terminaisons locales des tâches écrivains ne permettent de constituer qu'un sous ensemble de cette liste : les tâches lecteurs peuvent en effet continuer de se créer, de migrer et donc de créer de nouveaux réplicats durant la phase de décision de terminaison globale des écritures. Il y a donc émission de la donnée vers un premier ensemble de sites lecteurs, puis transmission aux réplicats qui ont été créés entre-temps et n'ont pas bénéficié de l'envoi de la donnée. Ce fonctionnement est illustré figure 5.4 page 98. Ce mécanisme risque de générer plus d'émissions de données que nécessaire. Ces émissions multiples sont étudiées au cours de la section 5.2.2.2 page 102 au paragraphe intitulé « Passage de l'état L à L_d ».

5.1.3.6 Localisation du site propriétaire

L'algorithme présenté section 4.4 page 80 nécessite l'existence d'un site propriétaire s qui centralise les messages de terminaisons locales et décide de l'éventuelle terminaison globale. Durant toute la présentation de l'algorithme, la localisation effective de ce site n'avait aucune importance. Cependant il est évident que le nombre de communications nécessaires à la détection de la terminaison globale dépend de la localisation de ce site propriétaire. Par exemple, considérons une situation où toutes les tâches accédant une version v ont été migrées vers le site p. Si $s \neq p$, des communications vont être nécessaires pour détecter la terminaison globale, alors que cette même décision peut être prise localement si $s = p$.

Le changement de ce site propriétaire n'est cependant pas une opération aisée. En effet, chaque réplicat possède une information sur la localisation de ce site : si cette localisa-

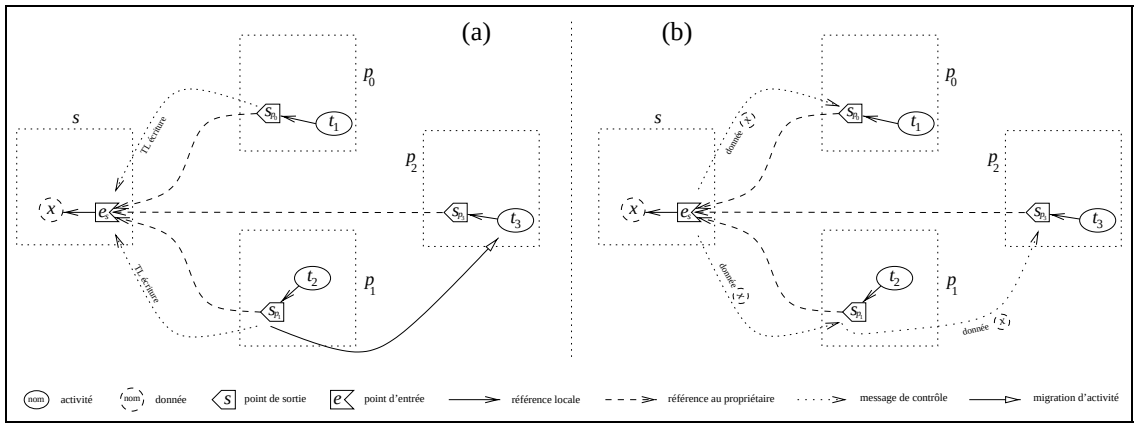


Figure 5.4 *Mécanisme d'émission de la donnée.*
En (a), les deux sites p_0 et p_1 émettent un message de terminaison locale des tâches écrivains vers le site propriétaire s, puis p_1 crée une tâche lecteur t_3 qui migre vers p_2 : il y a création d'un réplicat de la transition sur p_2. Les tâches t_1, t_2 et t_3 accèdent donc toutes la même version en lecture. En (b), le site s décide que la version est prête et émet la donnée synthétisée x vers les sites lecteurs qu'il connaît : p_0 et p_1. Ce dernier, seul à connaître l'existence d'un réplicat sur le site p_2, doit donc retransmettre la donnée. Notons qu'il y a alors un risque d'émissions multiples, par exemple si p_0 a également créé puis migré une tâche vers p_2. Ces émissions multiples sont étudiées dans la section 5.2.2.2 page 102 au paragraphe intitulé « Passage de l'état L à L_d ».

tion change, il faut en avertir tous les réplicats (l'indirection par l'ancien site propriétaire génère beaucoup trop de messages) mais surtout assurer que l'ordre de réception des messages de terminaison locale sera conservé, l'inversion de messages de terminaison locale pouvant mener à une détection erronée de la terminaison globale (voir la figure 4.14 page 88).

Cependant, dans le cas où la transition n'a pas encore été répliquée, le changement du site propriétaire est aisé : c'est une opération purement locale. Dans un tel cas, et lors de la première réplication, l'heuristique[4] simpliste que nous proposons est la suivante : par défaut, le site s est localisé sur le site de création de la version ; puis la première tâche qui migre vers un site p emmène avec elle ce site propriétaire. Cette heuristique permet d'éliminer les messages inutiles de la situation précédente sans avoir à résoudre les problèmes posés par un changement plus général du site propriétaire puisque le changement de localité est effectué lorsque la transition n'est pas encore répliquée.

5.2 Analyse du coût

Nous évaluons dans cette section le coût de l'implantation de la gestion du graphe présentée dans les sections précédentes de ce chapitre. Nous évaluons tout d'abord le coût de la construction de ce graphe, puis le coût de sa gestion dans un environnement

[4]Cette heuristique est **cruciale** dans le cas où toutes les tâches du graphe, donc toutes les données partagées, sont générées sur un même site : TOUS les sites propriétaires se retrouvaient sur ce site ce qui engendrait non seulement un goulot d'étranglement mais encore entrainait des allers et retours inutiles de messages.

distribué, c'est-à-dire le nombre de messages engendrés. Des évaluations quantitatives expérimentales sont présentées à la fin de ce chapitre.

5.2.1 Coût de construction

Proposition 7 *Le **coût** de toute création de tâche ou de donnée partagée **est borné**. De plus, toutes **ces créations sont locales** et ne génèrent aucune communication.*

Nous prouvons cette proposition en décomposant les opérations effectuées lors de ces créations.

Preuve. Comme présenté section 3.4 page 60, le graphe évolue dynamiquement lors des créations de tâches (`Fork<` t `>`) ou de données en mémoire partagée (`Shared` x). Ces créations sont toujours locales. D'une manière générale, une évolution du graphe peut être décomposée comme suit :

- Créations de structures locales.
- Modification du graphe.
- Évolution de l'état des nœuds adjacents.

Le coût de création des structures locales dépend de la modification apportée au graphe, modifications répertoriées à la section 3.4 page 60. Il y a création de deux arêtes et de deux transitions dans le cas d'une déclaration de donnée en mémoire partagée et, dans le cas de l'instanciation d'un paramètre de tâche, création d'au plus une transition et deux arêtes. Le coût de création des structures de base (transitions et arêtes) étant constant, le coût de création d'une donnée en mémoire partagée ou d'un paramètre l'est également. Le coût de création d'un nœud tâche est donc une fonction linéaire du nombre de ses paramètres.

Les modifications apportées au graphe sont effectuées localement. En effet, la tâche ne faisant pas d'effet de bord sur la mémoire partagée, toutes les versions qui peuvent être accédées par la tâche sont connues lors de sa création et sont maintenues présentes, sous forme de réplicats si nécessaire, sur son site d'exécution. Les manipulations du graphe présentées en section 3.4 page 60 n'engendrent donc aucune communication.

Si l'état des nœuds adjacents pouvait évoluer, cela pourrait engendrer des communications (dues à la gestion distribuée) ou une cascade de changement d'état dont le coût ne pourrait être maîtrisé. Ces évolutions ne peuvent avoir lieu. En effet, les ajouts n'activent pas l'algorithme de terminaison et le seul retrait d'élément du graphe, qui concerne le cas de l'ajout d'un écrivain sur une donnée accessible en lecture, est compensé par la création d'un lien de précédence qui fige l'état de la version, ce lien étant considéré comme un accès en lecture.

En considérant que le nombre de paramètres des tâches est borné pour l'application, on obtient alors que le coût de chaque évolution du graphe (dans un sens de construction) est borné et ne génère aucune communication. □

5.2.2 Coût de gestion

Nous présentons dans cette section le coût de gestion du graphe dans un environnement distribué, c'est-à-dire le nombre de messages qui sont générés. Tout d'abord nous détaillons l'ensemble des messages générés par notre algorithme de terminaison, puis celui qu'aurait nécessité une implantation directe de l'application sur une base de processus communicants, et enfin nous montrons que sous certaines conditions notre algorithme génère strictement les mêmes messages que cette implantation directe.

Nous considérons dans la suite une version v et examinons le nombre m de messages relatifs à cette version : ce sont des messages de contrôle ou d'envoi de données. Nous noterons $\mathcal{P}_e$ l'ensemble des sites participant à l'écriture sur la version v, $\mathcal{P}_l$ l'ensemble des sites participant à la lecture de la version v, N_e le nombre de tâches possédant un droit en écriture, N_l celui des tâches possédant un droit en lecture.

5.2.2.1 Implantation directe

Considérons une implantation directe de l'application qui respecte le même placement des tâches, c'est-à-dire que les sites d'exécutions sont identiques. Nous supposons donc que seules les données liées à une synchronisation écrivain-lecteurs entre des tâches situées sur des processeurs différents entraînent une communication. Les messages générés sont alors les suivants :

- Il y a un message de migration par tâche devant s'exécuter à distance.
- À la fin des écritures il faut synthétiser la donnée et prévenir les sites lecteurs. Il faut $|\mathcal{P}_e| - 1$ messages pour synthétiser la donnée sur un des sites écrivains puis $|\mathcal{P}_l|$ pour envoyer cette donnée aux sites lecteurs qui l'attendent. Ce nombre peut être diminué de 1 si l'intersection des deux ensembles n'est pas vide. Le nombre de messages générés est donc :

$$m_{direct} = |\mathcal{P}_e| - 1 + |\mathcal{P}_l| - \varepsilon, \quad \text{avec} \quad \varepsilon = \begin{cases} 1 \text{ si } \mathcal{P}_e \cap \mathcal{P}_l \neq \emptyset \\ 0 \text{ sinon} \end{cases}$$

5.2.2.2 Messages engendrés par l'algorithme de terminaison

L'algorithme présenté à la section 4.4.2 page 82 gère de manière distribuée les changements d'états du graphe. Nous examinons dans la suite le nombre de messages générés par chacune des étapes de cet algorithme, c'est-à-dire chaque évolution d'état (l'état X_d représente l'état X sous une forme distribuée). Nous noterons $m_{X \to Y}$ le nombre de messages engendrés lors du passage de l'état X à l'état Y.

Initialement la version v est dans l'état A.

Passage de l'état A à A_d Un tel passage est lié à la migration d'une tâche qui nécessite la création d'un réplicat. Cela ne nécessite aucune communication autre que celle de migration de la tâche. Il est en effet possible de créer le réplicat sur le site distant

lorsque la tâche qui migre arrive à destination et ne trouve pas de réplicat associé à une des versions qu'elle accède. Ainsi :

$$m_{A \to A_d} = 0$$

Passage des états A, A_d à l'état P Ce changement d'état est dû à la terminaison des tâches qui possèdent un droit d'accès en écriture. Pour cette détection globale, l'algorithme de terminaison distribué se base sur le fait que chaque processeur émet un message lors de chaque terminaison locale. Le nombre de messages ainsi généré est donc au minimum égal au nombre de processeurs participants à l'écriture, soit $|\mathcal{P}_e|$, éventuellement diminué de 1 si le site propriétaire est un des sites écrivains. Dans le cas général, il n'est cependant pas possible de donner une borne supérieure. En effet, considérons une tâche unique accédant en écriture la version v et qui ne cesse de migrer : chaque fois que cette tâche quitte un site, ce site génère un message de terminaison locale. Ainsi :

$$|\mathcal{P}_e \setminus s| \leq m_{A,A_d \to P} \leq \infty$$

Cependant, en supposant que chaque tâche ne peut pas migrer plus d'une fois (ce qui revient à considérer que l'ordonnanceur ne remet jamais en cause son placement), le nombre maximum de messages de terminaison locale peut être ramené à $1 + N_e$, ce qui reste toujours important. Le comportement de la politique d'ordonnancement est donc déterminant dans le nombre de messages générés.

Passage de l'état P à P_d Afin de lever les contraintes de précédence, les réplicats doivent être prévenus du nouvel état. Il y a alors émission de l'état prêt aux sites possédant des tâches accédant la version en lecture. Comme précédemment, le nombre minimum est égal au nombre de processeurs possédant des tâches lecteurs, soit $\mathcal{P}_l$. Cependant, lors de la détection de l'état P, le site propriétaire ne connait qu'un sous ensemble des réplicats : tous ceux qui ont participé à l'écriture. Or il faut atteindre l'ensemble des réplicats possédant des tâches en lecture. Il est alors nécessaire de mettre en œuvre un mécanisme de diffusion pour cet état, comme illustré figure 5.4 page 98. Chaque site veillant à ne pas émettre deux fois vers le même site, il vient que le nombre de messages est majoré par $|\mathcal{P}_l| \left(|\mathcal{P}_l| - 1\right)$. Ainsi :

$$|\mathcal{P}_l \setminus s| \leq m_{P \to P_d} \leq |\mathcal{P}_l| \left(|\mathcal{P}_l| - 1\right)$$

Cette borne supérieure peut certainement être réduite mais pas significativement. Considérons par exemple le cas de $2n$ sites lecteurs tels que le site propriétaire s n'ait connaissance que de la moitié. Un premier message est alors émis vers ces n sites. Ceux-ci peuvent savoir qu'ils n'ont pas à diffuser le message entre eux, mais tous vont le faire passer aux n suivants, ce qui génère un total de n^2 messages (en effet, il ne savent *a priori* pas que d'autres sites vont émettre cette même donnée) ; la borne supérieure reste donc forcément quadratique.

Passage des états P, P_d à l'état L Toutes les écritures étant terminées, la donnée doit être synthétisée. Chaque site où a eu lieu une écriture doit alors envoyer ce qu'il sait de la donnée. Le nombre de messages est donc égal au nombre de processeurs ayant participé à l'écriture, soit $|\mathcal{P}_e| - 1$ (la synthèse est effectuée sur un des sites écrivains). Ainsi :

$$m_{P,P_d \to L} = |\mathcal{P}_e| - 1$$

Passage de l'état L à L_d Tous les processeurs de $\mathcal{P}_l$ doivent posséder une copie de la donnée afin de pouvoir exécuter les tâches qui vont accéder cette version en lecture. Cette situation est identique à celle du passage de l'état P à P_d et entraîne une diffusion. Ainsi :

$$|\mathcal{P}_l - \{s\}| \leq m_{L \to L_d} \leq |\mathcal{P}_l| \left(|\mathcal{P}_l| - 1\right)$$

Passage à l'état $\perp$ La fin de tous les accès, écriture et lecture, doit être détectée afin de pouvoir libérer la mémoire réservée à la donnée partagée. Cette situation est identique à celle de la détection de l'état prêt. Ainsi :

$$|\mathcal{P}_l - \{s\}| \leq m_{P,P_d,L,L_d \to \perp} \leq \infty$$

5.2.2.3 Comparaison

Le nombre de messages générés par notre algorithme dépend de la politique d'ordonnancement choisie. Dans le cadre général, ce nombre ne peut être comparé à celui de l'implantation directe. Pour générer les mêmes communications que celle-ci, il faut satisfaire les conditions suivantes :

- Éviter les migrations successives d'une même tâche pour faire en sorte que chaque site impliqué dans une écriture ne génère qu'un seul et unique message de terminaison. Il faut également regrouper les messages des changements d'états $A, A_d \to P$ et $P \to L$.
- Supprimer le mécanisme de diffusion, et donc diffuser en une étape la donnée valide aux sites lecteurs. Il faut également regrouper les messages des changements d'états $P \to P_d$ et $L \to L_d$.
- Supprimer les messages de détection de l'état terminé ($P, P_d, L, Ld \to \perp$).

La réactivité n'étant pas primordiale pour la détection de l'état terminé, les messages engendrés par cette détection peuvent être supprimés en utilisant une technique de *piggy-packing* qui consiste à greffer sur des messages déjà existants ces messages de contrôle. On peut donc considérer que cette détection ne nécessite aucun message supplémentaire.

Pour supprimer le mécanisme de diffusion il faut que le site propriétaire connaisse l'ensemble des sites lecteurs lorsque la décision de la fin des écritures est prise. Pour éviter l'excès de messages de terminaison locale d'écriture, il faut éviter toutes les migrations de tâches. Un placement statique, où les tâches accédant une même version sont placées dans leur ensemble, permet de satisfaire à ces deux contraintes. De tels ordonnanceurs existent

et procèdent par exemple par phases : une phase de création du graphe, sans placement, puis une phase de placement de la portion de graphe généré. L'exécution des tâches ainsi placées relance une phase de création du graphe. Dans ce cas, les messages générés sont **identiques** à ceux générés par l'implantation directe. La proposition suivante permet de formaliser cette situation :

Proposition 8 *Sous certaines conditions, la gestion dynamique et distribuée du graphe de l'application ne nécessite pas plus de messages que ceux nécessaires aux migrations de données. Ces messages sont alors identiques à ceux d'une implantation directe à l'aide de processus communiquant par échange de messages.*

5.2.3 Quelques optimisations envisageables

Nous traitons dans cette section quelques optimisations possibles concernant l'implantation. Ces optimisations visent à réduire le surcoût de la construction du graphe ou à réduire le nombre de messages nécessités par le mécanisme de terminaison globale. Ces optimisations ne sont pas forcément très difficiles à implanter, mais nécessitent tout de même une certaine quantité de développement et d'évaluation. Pour toutes ces raisons, l'implantation actuelle ne les contient pas.

5.2.3.1 Création du graphe

La création du graphe est systématique. Or, si ce graphe est effectivement nécessaire lors d'une exécution parallèle pour respecter les contraintes de précédence, dans le cas d'une exécution purement séquentielle il est parfaitement inutile. En effet, l'exécution en profondeur d'abord de l'arbre de création des tâches respecte toutes les contraintes de précédence et aucune protection contre une éventuelle concurrence n'est à effectuer.

Cette situation, où les tâches sont exécutées par un seul processeur, séquentiellement, n'est pas une situation rare : lorsque suffisamment de parallélisme a été généré et que tous les processeurs sont actifs, un bon ordonnanceur se doit d'arrêter de distribuer les tâches de manière à limiter le surcoût entrainé par la gestion de cohérence.

L'optimisation consiste à ne créer ce graphe que lorsqu'il est vraiment nécessaire. Ainsi, par défaut, il y a simple empilement d'une information minimale des tâches à créer, sans réelle création des clôtures ou transitions [51, 80] et donc sans mise en œuvre de l'algorithme de terminaison. Ces informations permettent une exécution locale de la tâche sous forme d'un classique appel de fonction. Si du parallélisme doit être généré ultérieurement (sur demande de l'ordonnanceur), les clôtures et les transitions pourront être construites et le graphe retourné à l'ordonnanceur.

5.2.3.2 Synthèse d'une donnée en accumulation

Dans le cas d'une écriture par accumulation, la donnée est actuellement synthétisée sur le site propriétaire, et chaque réplicat ayant participé à l'écriture émet le résultat des accumulations locales. Soit N le nombre de réplicats émettant une donnée. Il va alors y avoir

N accumulations sur le site propriétaire. Ces accumulations seront exécutées séquentiellement. Si la durée de chaque accumulation est importante, il pourrait être avantageux de faire ces accumulations en parallèle sous une forme d'arbre par exemple, ce qui ménerait à une durée d'accumulation de $\log N$ sans accroître le nombre de messages générés.

5.2.3.3 Site de synthèse

Dans l'implantation décrite, la donnée est synthétisée sur le site propriétaire s. Cela implique dans certains cas des allers et retours de la donnée finale, comme illustré figure 5.5 page 104. Ces communications auraient pû être évitées si le site propriétaire s qui décide de la terminaison globale avait été différent du site où la donnée est maintenue. Il suffit pour cela de séparer les phases de détection de terminaison globale et de synthèse de donnée. Le réplicat responsable de la donnée doit être au courant des sites où cette donnée doit être émise pour lecture ; de même les réplicats ayant participé à l'écriture doivent être avertis du site de synthèse. De manière à ne pas augmenter le nombre de messages, ces informations peuvent être transmises par le site propriétaire s en même temps que l'information de changement d'état.

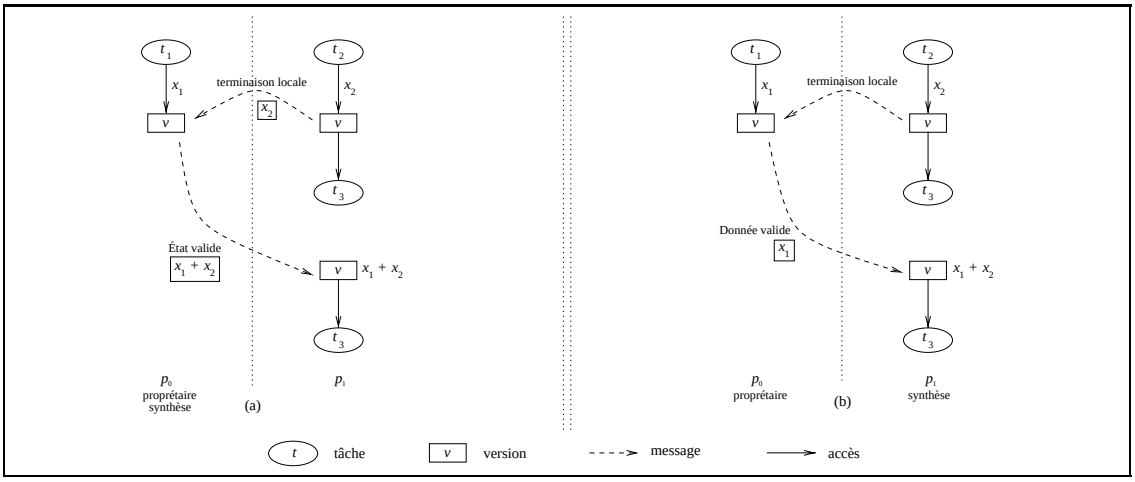


Figure 5.5 *Aller et retour lors de la synthèse de la donnée.*
En (a), le site propriétaire et le site de synthèse sont identiques et égal à p_0. La donnée x_2 est émise vers p_0 pour être accumulée à x_1 puis réémise vers p_1 pour être accédée en lecture par la tâche t_3. En (b), p_0 reste le site propriétaire, mais p_1 devient le site de synthèse de la donnée. Il n'y a dans ce cas qu'une seule émission de donnée, x_1 allant de p_0 à p_1.

5.2.3.4 Gestion quasi-optimale du graphe

Il existe des situations où la gestion du graphe peut être considérablement simplifiée. Considérons par exemple le cas où toutes les tâches de l'application sont générées uniquement par une seule tâche, la tâche principale `main`. Il est alors possible de générer ce graphe sur tous les sites en exécutant la tâche principale sur tous les nœuds de la machine : le graphe n'est alors plus distribué, mais intégralement répliqué. Plus aucune migration de tâches ou réplication de transition n'est donc nécessaire, le graphe étant connu partout.

Si de plus l'ordonnancement des tâches effectué est un ordonnancement statique et déterministe, chaque site peut effectuer ce placement et donc savoir où chaque tâche du graphe va être exécutée. Aucune communication de tâche n'aura lieu, puisque les tâches sont disponibles sur tous les sites ! Sur un site donné, seules les tâches devant être exécutées sur ce site le seront. Les autres seront ignorées.

Lors de la terminaison d'une écriture, le site écrivain sait où auront lieu toutes les lectures ; il peut donc émettre la donnée directement vers ces sites. Cette émission lèvera les contraintes de précédence des tâches lecteurs.

Une telle implantation du graphe et de sa gestion, qui repose sur de sévères restrictions (placement statique et génération du graphe par une tâche unique) permet d'atteindre des performances remarquables : meilleures que celles obtenues lors d'une implantation directe de l'application par échange de messages[5]. Cette optimisation du graphe de flot de données est présentée plus en détail dans [42].

5.3 Évaluations

Les évaluations numériques de cette section ont été réalisées dans les conditions suivantes :

- Utilisation de la version 1.7.7 d'Athapascan-1. Cette version permet l'utilisation de plusieurs nœuds faiblement couplés communicant par échange de messages. Chaque nœud est composé d'un certain nombre de processeurs virtuels. Le système exécutif utilisé, Athapascan-0 [19, 61], est brièvement présenté section 7.2.1 page 131. Chaque nœud peut être vu comme un processus UNIX lourd et chaque processeur virtuel au sein d'un nœud comme un processus léger à l'intérieur de ce processus lourd (il est donc possible d'exploiter les machines multiprocesseurs, soit en utilisant plusieurs nœuds, soit en utilisant plusieurs processeur virtuels au sein d'un même nœud).
- La version d'Athapascan-1 utilisée possède son propre allocateur. En effet, l'utilisation de l'allocateur par défaut est particulièrement inefficace compte tenu du fait que nous allouons et libérons un grand nombre de « petites » structures (clôtures, transitions, *etc.*). Cet allocateur mémoire est basé sur une mécanique classique de *free list*. Il permet de gagner 10 à 15% sur les temps de créations et destructions des structures internes de la bibliothèque.
- Afin de permettre au compilateur de remplacer certains appels de fonction par leur corps (*inlining*), une seule unité de compilation est utilisée pour générer l'application[6]. Dans toutes les applications que nous avons expérimentées cela apporte un gain situé entre 15 et 20%.

[5] Car bien que les messages soient rigoureusement identiques dans les deux implantations, leur gestion est différente : elle est entièrement asynchrone dans le cas d'Athapascan-1. Cela revient à dire que l'implantation directe en MPI, bien que naturelle et soignée, n'était pas la meilleure possible.

[6] Nous avons pour cela inclus tout le code de la bibliothèque dans le code de l'application, en incluant les fichiers d'entêtes et les fichiers de corps.

– Machine Sparc bi-processeurs 300 MHz sous Solaris. Dans ces conditions d'expérience, la durée d'un appel de fonction sans argument est de $9.6ns$, ce qui correspond à 3 cycles d'horloge du processeur.

– Optimisation `-O5` (la plus haute) du compilateur C++ utilisé[7].

5.3.1 Structures utilisées

La seconde colonne du tableau 5.1 page 106 indique la taille mémoire des objets utilisés dans l'implantation. Ces valeurs doivent être considérées comme des maxima, peu d'optimisations ayant été apportées lors de leur conception. En particulier, les parties constantes des tailles des clôtures et des transitions doivent pouvoir être considérablement réduites. Les durées de création d'un réplicat ou d'un site propriétaire sont identiques bien que plus de structures doivent être allouées dans le second cas. C'est l'utilisation d'un allocateur propre qui permet de rendre ces allocations quasiment indétectables.

objet	**taille**	**création**		**destruction**	
	octets	μs	appels	μs	appels
transition, réplicat	$156 + 8p$	5.03	524	5.30	552
site propriétaire	$156 + 16p$	5.03	524	5.30	552
clôture	$128 + t_p$	0.35	36	1.30	135
donnée partagée	20	0.059	6	0.057	6
évolution	72	0.01	1	0.01	1

Tableau 5.1 *Coût en temps et en mémoire des principaux objets utilisés dans l'implantation du graphe de flot de données.*
La deuxième colonne représente la taille mémoire nécessitée pour chacun des objets de l'implantation. Les temps de création et de destruction sont donnés en μs et en équivalent en nombre d'appels de fonction. p représente le nombre de sites et t_p la taille des paramètres de la clôture : cette taille vaut 20 pour chacune des références à une donnée partagée, et la taille de chaque objet $+4$ dans le cas d'un paramètre par valeur (plus 4 octets car cet objet est encapsulé dans une classe Athapascan-1 permettant d'abstraire son type).

Chaque mesure compte 10000 créations ou destructions d'objet (réalisées à l'aide d'un tableau C++). La valeur reportée est la moyenne de 200 telles mesures auquelles on a écarté les 10 valeurs extrêmes (les 5 plus grandes et les 5 plus petites). La prise de temps est une mesure de temps réel, mais a été réalisé seul sur la machine (les chances d'être perturbé sont donc très faibles, comme en témoigne la variance des résultats).

Une évolution est une structure « pont » qui est associée à toute donnée partagée accédée par une tâche et qui pointe vers deux transitions : la version courante (celle accédée en lecture) et la version future (celle accédée en écriture). Cet objet, qui représente la succession de deux arêtes du graphe, contient les droits et modes d'accès des tâches sur les versions accédées et les champs nécessaires à la formation des listes utilisées pour le stockage des clôtures accédant les transitions et pour le stockage des paramètres de type

[7] `CC : WorkShop Compilers 4.2 30 Oct 1996 C++ 4.2.`

données partagées pour les tâches. Ceci explique cette taille de 72 octets qui peut paraître, à première vue, bien trop conséquente pour seulement deux pointeurs.

5.3.2 Surcoût de gestion du graphe

Nous présentons dans cette section le surcoût imposé par la création locale du graphe. Nous donnons figure 5.6 page 107 la durée de création et d'insertion dans le graphe d'une tâche en fonction du nombre et du type de ses paramètres. Nous étudions ensuite, figure 5.7 page 109 l'influence du surcoût de gestion imposé par le graphe sur la durée moyenne que doivent avoir les tâches pour permettre son amortissement. Enfin nous illustrons le comportement de l'algorithme dans le cas d'une gestion distribuée.

Création de tâches L'expérience consiste à mesurer la durée de création d'une tâche en fonction du nombre de ses paramètres. La durée de création, selon l'algorithme de construction du graphe utilisée, doit être fonction du nombre et du type des paramètres. Nous traçons donc, pour les cinq types de paramètres possibles (par valeur, `Shared_r`, `Shared_w`, `Shared_r_w`, `Shared_cw`). Les mesures sont présentées dans la figure 5.6 page 107. Un point sur la courbe représente la durée moyenne de création, moyenne effectuée sur 200 tâches. Chaque mesure (un type de tâche, un nombre de paramètres) a été exécutée 50 fois et tous ces points sont représentés.

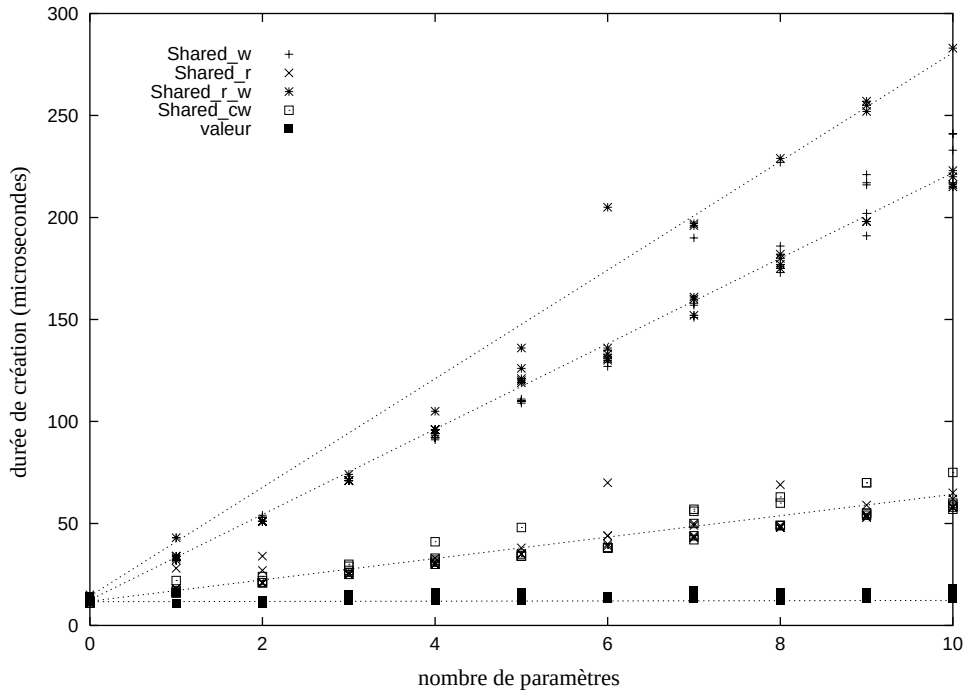


Figure 5.6 *Durée de création d'une tâche en fonction du nombre de ses paramètres..*
La durée de création d'une tâche est linéaire en fonction du nombre de ses paramètres (tous de même type).

La durée de création d'une tâche est donc linéaire en fonction du nombre de ses paramètres. Il est à noter que dans le cas des paramètres par valeurs, ce paramètre est recopié : la durée de création est donc fonction de la durée de recopie de ces éléments. Dans l'expérience, le type considéré était les entiers.

On remarque de plus que la durée de création de la tâche lorsque ses paramètres sont en lecture ou en accumulation est identique, les cas de constructions (C2) et (C3) page 60 étant similaires. La durée pour des paramètres effectuant des écritures est plus importante car ces paramètres nécessitent la création d'une transition, cas de la construction (C4). On remarque enfin que la durée de création de la tâche, lors de l'utilisation de paramètres en modification, est égale à la somme des durées pour des paramètres en écriture et en lecture : ce sont les schémas de construction (C2) et (C5) qui sont utilisés.

Gestion locale du graphe Le graphe est manipulé lors des créations et des destructions de tâches ou de données partagées. Si les tâches créées n'effectuent aucun calcul, il est évident que c'est ce surcoût de gestion qui sera prédominent dans la durée d'exécution. *A contrario*, si les tâches sont longues, le surcoût (qui est constant) ne représentera qu'une partie infime de la durée d'exécution. Nous évaluons ici la durée « moyenne » minimale que doivent avoir les tâches pour pouvoir masquer la gestion du graphe (il peut y avoir beaucoup de tâches très petites, pour peu qu'il y en ait suffisamment de plus longues). L'exemple considéré est celui du calcul du nombre de Fibonnacci $F(n)$ pour $n = 35$. Le calcul est effectué récursivement de la manière suivante :

- Si n est inférieur à un certain seuil, nous calculons $F(n)$ en séquentiel (de manière récursive) ce qui permet de faire varier la durée des tâches « terminales » du calcul.
- Sinon, si n est supérieur à ce seuil, nous évaluons en parallèle $F(n-1)$ et $F(n-2)$. Ces deux tâches partagent en accumulation une même donnée en mémoire partagée dans laquelle elles iront stocker leur résultat.

La courbe 5.7 page 109 a été tracée en comparaison avec le langage Cilk en utilisant le compilateur g++[8] afin de coller au plus près aux exigences de Cilk qui impose l'utilisation de gcc-2.7.x. Cette courbe représente la durée du calcul de $F(35)$ en fonction du seuil d'arrêt. Les courbes sont tracées pour 1 et 2 processeurs pour les langages Cilk-5.2 et Athapascan-1[9].

Nous pouvons remarquer tout d'abord que les courbes sont identiques, pour Athapascan-1 et pour Cilk, en ce qui concerne la partie droite, c'est-à-dire pour un seuil supérieur à 21. Dans la partie gauche, les courbes ont une forme similaire mais celle d'Athapascan-1 est décalée vers la droite : cela est dû au surcoût plus important de la gestion du parallélisme dans Athapascan-1 par rapport à Cilk. On peut, graphiquement, noter le seuil à partir duquel le surcoût du langage devient visible : c'est $F(21)$ pour Athapascan-1 et $F(15)$ pour Cilk, ce qui correspond respectivement à $2120\mu s$ et $384\mu s$. Il est à noter que, s'il n'y avait aucun surcoût de création et de gestion des tâches, les courbes seraient des droites horizontales (aux alentours de $2s$).

[8] `gcc version gcc-2.95 19990602 (prerelease)`.

[9] Nous utilisons 1 nœud comportant 2 processeurs virtuels : la création et la gestion du graphe est donc locale et protégée par un verrou entre les deux processeurs virtuels.

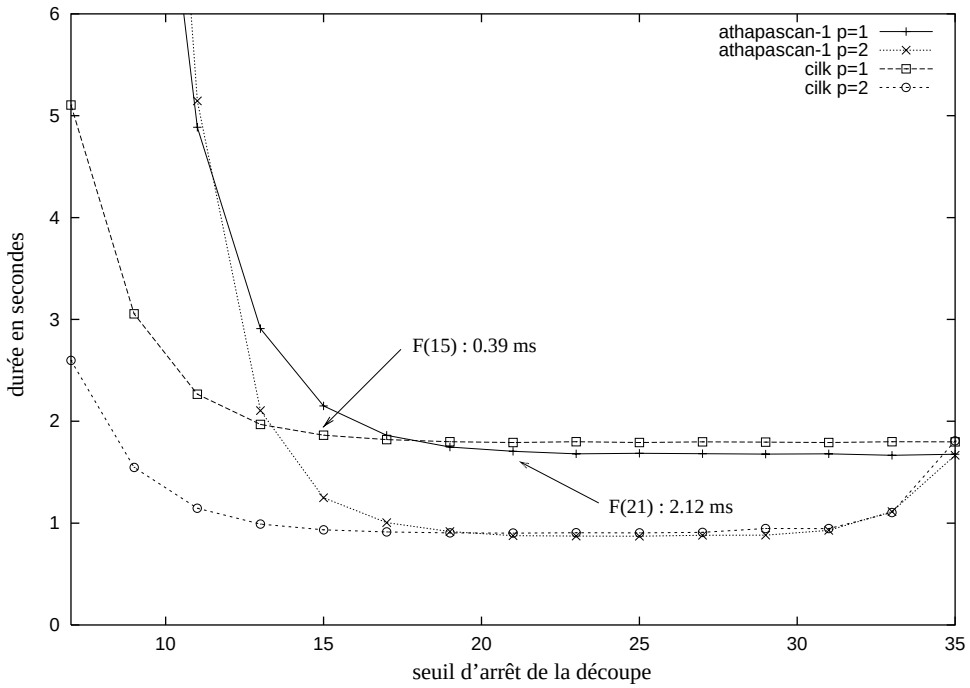


Figure 5.7 *Influence du seuil d'arrêt de la découpe sur la durée d'exécution..*
Cette courbe représente, pour 1 et 2 processeurs, la durée de calcul de $F(35)$ en fonction du seuil d'arrêt de la découpe (en deça de ce seuil le calcul est effectué de manière itérative en séquentiel). Les points particuliers $F(21)$ et $F(15)$ représentent les seuils à partir desquels les surcoûts de gestion ne peuvent plus être négligés.

Une autre remarque est le fait que la courbe pour 2 processeurs, dans le cas d'Athapascan-1 et pour des valeurs faibles du seuil, croit plus vite que celle de Cilk dans les mêmes conditions. Ceci est illustré par la courbe figure 5.8 page 110 traçant l'accélération $a = \frac{T_1}{T_2}$ en fonction du seuil.

De plus, en deça d'un certain seuil, l'accélération s'effondre pour Athapascan-1 tandis que la version Cilk conserve une accélération de 2. Deux raisons sont avancées :

– La première concerne la présence d'un verrou global dans Athapascan-1 qui sérialise tous les accès au graphe, et plus généralement aux fonctionnalités de la bibliothèque. Lorsque le seuil devient trop faible, l'application passe l'intégralité de son temps dans des appels à la bibliothèque : comme ces appels sont sérialisés, il est alors normal que l'accélération chute et tombe proche de 1. Depuis la version Cilk-5.1, Cilk a supprimé ce verrou.

– Cependant, l'accélération passe en dessous de la barre de 1, barre correspondant à une sérialisation totale des accès au graphe ou aux fonctions de la bibliothèque. Cette anomalie s'explique par des invalidations de cache trop importantes entre les processeurs. Afin de s'en convaincre nous avons implanté, en utilisant directement une bibliothèque de processus légers, 2 processus légers se disputant le même verrou : nous obtenons une décélération de l'ordre de 3.5 (quelque soit le type de verrou utilisé : standard ou *spin*) sur 2 processeurs. La durée de la prise de ver-

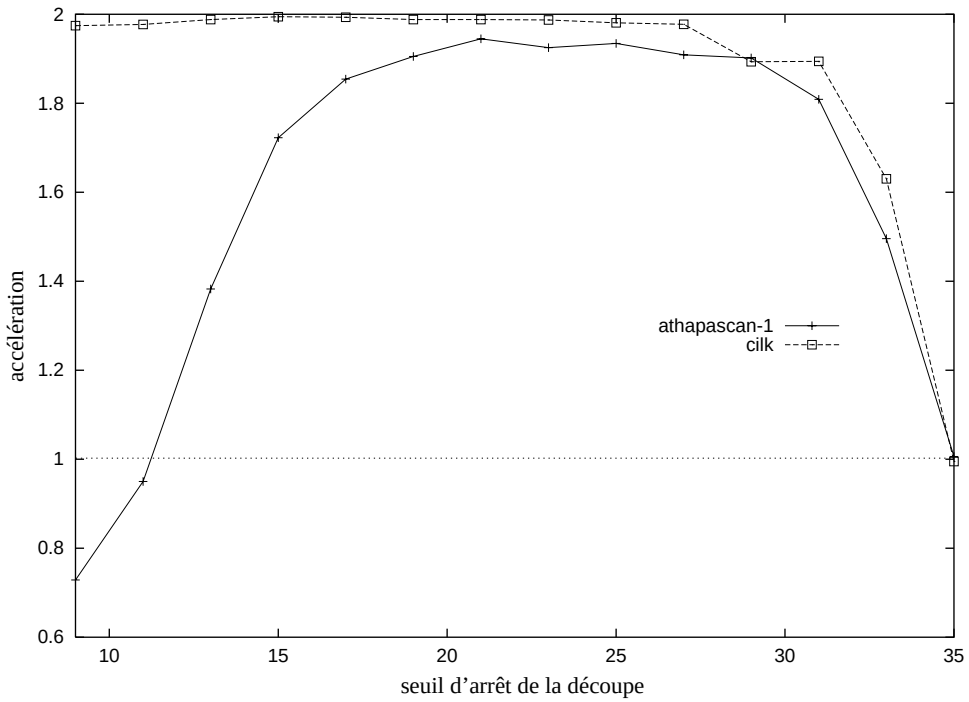


Figure 5.8 *Influence du seuil d'arrêt de la découpe sur l'accélération* $\left(a = \frac{T_1}{T_2}\right)$ *de l'exécution sur* 2 *processeurs.*
Le ralentissement d'Athapascan-1 pour des seuils d'arrêt faibles s'explique par une sérialisation des accès au graphe et des surcoûts dûs aux invalidations de cache.

rou est donc multipliée par près de 4 lors d'une exécution concurrente, ce qui peut être expliqué soit par le surcoût introduit par la gestion du verrou, soit par des phénomènes d'invalidation de cache. Une expérience similaire permettant de montrer l'influence de la gestion du cache consiste, au lieu de se disputer un verrou, de se disputer l'accès à un entier (lecture et écriture) ce qui implique une invalidation de cache lors de chaque accès. Dans cette situation, la durée d'exécution est multipliée par 3.2 lors d'une exécution sur 2 processeurs. C'est donc assurément cette durée d'invalidation qui constitue la plus grande part de la durée de prise de verrou. C'est donc toute forme de partage qu'il faut éviter entre les processeurs virtuels.

Notons que la version optimisée de la gestion du graphe, présentée section 5.2.3.4 page 104 mais modifiée afin d'autoriser, sur une machine *SMP*, les créations récursives de tâches, obtient les mêmes performances que Cilk sur un processeur. Cette version présente le même phénomène que la version distribuée d'Athapascan-1 en ce qui concerne l'anomalie d'accélération compte tenu de la présence d'un verrou global partagé entre les 2 processeurs virtuels. La même expérience, menée à l'aide de 2 nœuds Athapascan-0 communicant par échange de messages, en utilisant la gestion distribuée du graphe de flot de données ne présente pas ce phénomène de décélération et conserve une efficacité supérieure à 1.9 même pour les seuils faibles (le test a été mené jusqu'à un seuil de 6).

Gestion distribuée du graphe Nous illustrons dans cette expérience le fait que la manipulation est effectuée lors de la création et lors de la terminaison des tâches.

Considérons le cas de 4 tâches, créées par la tâche principale, qui se partagent en modification la même donnée partagée : ces tâches sont donc sérialisées. L'exécution a lieu sur deux nœuds possédant chacun un unique processeur virtuel : la première et troisième tâches sont placées sur le nœud 1, la seconde et la dernière sur le nœud 0. La figure 5.9 page 111, tracée à l'aide de l'outil de visualisation Paje[10] [33, 32], illustre les échanges de messages entre les deux nœuds de la machine.

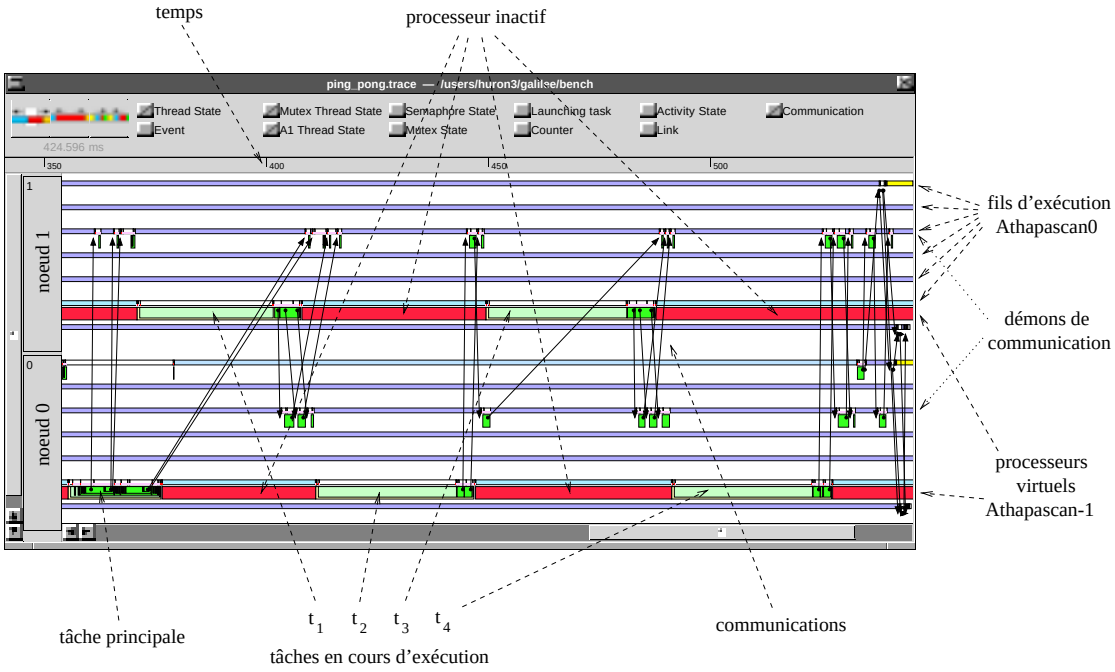


Figure 5.9 *Visualisation des communications impliquées par la gestion distribuée du graphe de flot de données.*
L'application consiste en l'exécution de 4 tâches $t_1 \prec t_2 \prec t_3 \prec t_4$ sur 2 nœuds de la machine. Les tâches t_1 et t_3 sont exécutées sur le nœud 0 et les tâches t_2 et t_4 sur le nœud 1. Nous remarquons que les communications entre processeurs ont lieu lors des terminaisons de tâches. Ces communications, permettant de gérer les états des nœuds du graphe, sont les mêmes que celles qui auraient eu lieu lors d'une programmation directe à l'aide de processus communicants. Cette visualisation a été effectuée à l'aide du logiciel Paje et de la version tracée du système exécutif Athapascan-0.

Il y a donc moyen de regrouper les communications afin de ne pas en faire plus qu'une implantation directe à l'aide de processus communicants par échange de messages, dans le cas où il n'y a pas d'aller retour sur ces messages. Les raisons de ces allers et retours sont exposées dans la section 5.2.3.3 page 104 (ce qui est le cas ici si l'on exclut les messages de destruction des réplicats de transition lorsque la version associée de la donnée partagée n'est plus accédée).

[10] L'*URL* de la page officielle du projet est la suivante : `http://www-apache.imag.fr/software/paje/`.

5.4 Bilan et critiques

Nous avons montré dans ce chapitre qu'il était possible de construire dynamiquement avec un **surcoût borné** et **sans ajout de communication**[11] le graphe de flot de données associé à toute exécution. La gestion distribuée de ce graphe est effectuée à l'aide d'un algorithme réactif de détection de terminaison. Le nombre de messages nécessaires à cette gestion distribuée peut, sous certaines conditions, être égal au nombre de messages qu'aurait nécessité une implantation directe de l'application sur une base de processus communicants.

Les évaluations de cette construction sont conformes aux prédictions de surcoût borné, mais le nombre de messages est en général important et limitant dans certaines applications. De plus, le surcoût, bien que borné, est relativement élevé ce qui limite l'utilisation de la bibliothèque à des tâches ayant, en moyenne, une durée assez grande. Le partage du graphe sur un nœud de la machine pose, lorsque les tâches sont de faible durée (de l'ordre de la centaine de microsecondes dans la version présentée), des problèmes de gestion de cache qui écroulent les performances : il est donc nécessaire, pour être capable de supporter localement des tâches de faible durée, d'éviter tout partage entre les processeurs virtuels.

De plus, la taille mémoire requise par ce graphe peut être considérablement réduite.

Nous étudions dans la partie suivante l'utilisation du graphe de flot de données dans la définition d'un modèle de coût (permettant de prédire la durée et la consommation mémoire de toute exécution) associé au modèle de programmation. il permet un contrôle de la consommation mémoire de toute exécution dans Athapascan-1.

[11] Par rapport à une programmation basée sur des processus communicants par échange de messages.

Partie II

Flot de données et contrôle de la mémoire

6

Flot de données et modèle de coût

Ce chapitre étudie l'apport de la modélisation des exécutions d'une application par un graphe, qu'il soit de précédence ou de flot de données, dans la définition d'un modèle de coût permettant de garantir les performances de toute exécution. Les principales sections de ce chapitre traitent les points suivants :

- Les **grandeurs caractéristiques** de l'application, grandeurs déterminées à partir d'une analyse théorique du graphe (section 6.2 page 116).
- Le **modèle de coût** qui permet de prédire, *a priori*, les performances de toute exécution (section 6.3 page 118).
- Les **garanties de performances** dans le cas d'Athapascan-1. Ces garanties sont liées à la politique d'ordonnancement utilisée (section 6.4 page 123).

6.1 Introduction

Comme vu au chapitre 2 page 21, la plupart des langages parallèles de « haut niveau » permettent de décrire le parallélisme d'une application à un degré bien supérieur à celui du nombre de processeurs. L'implantation du langage est alors responsable de l'exploitation du parallélisme exprimé sur la machine cible : l'efficacité de l'exécution reposera entièrement sur l'ordonnancement des tâches qui sera effectué par le système. Cette efficacité est mesurée en terme de durée d'exécution et de volume mémoire requis.

Afin de permettre une analyse formelle et théorique de l'exécution d'une application, toute exécution est modélisée par un graphe. C'est à partir de ce graphe, donc de cette modélisation, que les performances en temps et en mémoire de cette application seront évaluées dans le modèle de coût associé au langage. La constitution de ce graphe dépend des besoins du modèle : il peut être réduit à un graphe de précédence (le cas de Cilk par exemple) ou contenir une information sur les données accédées et devenir un graphe de flot de données (le cas d'Athapascan-1 ou de Jade par exemple).

Un ordonnancement glouton peut être effectué dynamiquement et à la volée et permet un exécution efficace en temps : les processeurs prennent, chaque fois qu'ils deviennent inactifs, une tâche à exécuter dans une liste de tâches prêtes. Ces techniques d'ordonnan-

cement exploitent le parallélisme de l'application à un degré permettant de maintenir au maximum les processeurs en activité. Ainsi sur un modèle de machine simple (processeurs identiques et absence de communication) un tel ordonnancement permet d'obtenir un temps d'exécution effectif à un facteur 2 de l'optimum. Cependant, ceci peut mener à une utilisation abusive de la mémoire : soit parce qu'un nombre trop important de tâches est maintenu à un instant donné par le système, soit parce que des tâches allouant de la mémoire sont exécutées avant d'autres tâches qui en libèrent. Il existe en effet, dans un cadre de graphe général, des applications parallèles pour lesquelles tout gain en temps, même minime, ne peut s'effectuer qu'au prix d'une consommation mémoire prohibitive [16], comme présenté section 6.3.2.1 page 119

Cependant, si l'on impose des restrictions sur le graphe d'exécution de l'application, il est possible de majorer simultanément la durée et la consommation mémoire de toute exécution. Par exemple, le langage Cilk impose un graphe de tâches série-parallèle et son implantation [51] garantit, en majorant à tout instant le nombre de *threads* en concurrence par p, une consommation en mémoire inférieure à pS_1, p étant le nombre de processeurs et S_1 l'espace mémoire requis pour une exécution séquentielle sur un processeur. De même l'implantation proposée de NESL dans [12] majore l'utilisation mémoire par $S_1 + p \log pT_\infty$, avec T_∞ la longueur en temps du plus long chemin du graphe. Avec une technique d'ordonnancement similaire le langage basé sur un graphe d'exécution série-parallèle proposé dans [83, 82] limite l'utilisation de la mémoire à $S_1 + pT_\infty$. La bibliothèque Athapascan-1 construit de manière dynamique un graphe orienté et sans cycle des accès aux données et aucune limitation n'est imposée sur ce graphe. Les performances de l'exécution dépendent de la politique d'ordonnancement choisie mais certaines garantissent de manière asymptotique des efficacités en mémoire de l'ordre de pS_1 ou $S_1 + O(pT_\infty + hC_\infty)$ (C_∞ représente la longueur en accès distants d'un plus long chemin du graphe).

Nous présentons tout d'abord les grandeurs caractéristiques associées à une exécution, puis la prédiction des performances en temps et en mémoire de ces exécutions à l'aide des modèles de coût associés aux langages de programmation. Nous présentons enfin le cas particulier d'Athapascan-1 et donnons deux politiques d'ordonnancement permettant de contrôler la consommation mémoire des exécutions tout en exploitant le parallèlisme de l'application (afin d'obtenir un gain en temps).

6.2 Grandeurs caractéristiques

Dans les langages offrant un modèle de coût, les performances de l'exécution peuvent être déduites du code de l'application. Ici, ces performances sont exprimées en fonction du graphe caractérisant l'instance de l'application et de la machine à l'aide des grandeurs suivantes :

– T_s, la durée d'exécution d'une implantation séquentielle de l'algorithme. Cette grandeur traduit le coût de la méthode de calcul utilisée par l'application : il ne contient aucun surcoût de parallélisme.

– T_1, la durée d'exécution du code parallèle de l'application sur un seul processeur. Cette grandeur représente le **travail de l'application**. Cette grandeur contient le coût de la méthode de calcul T_s et le surcoût introduit par la description du parallélisme. En particulier nous supposerons :

$$T_1 \geq T_s$$

De plus, cette valeur de T_1 est supposée indépendante de l'ordonnancement effectué des tâches[1].

– S_1, la consommation mémoire de l'exécution sur un seul processeur. L'exécution considérée est celle décrite dans la définition de T_1 et l'ordre d'exécution des instructions est celui calculé par l'ordonnanceur. Si cet ordre n'avait que peu d'influence sur la valeur de T_1, il est désormais primordial pour la définition de S_1.

– T_∞, la durée d'exécution théorique sur une infinité de processeurs : c'est à dire le meilleur temps d'exécution possible sur une machine. Cette grandeur est une borne inférieure pour la durée d'exécution sur toute machine parallèle et représente le maximum des durées d'exécution des ensembles d'instructions qui doivent être exécutées séquentiellement (suite à des contraintes de précédence). T_∞ représente la durée d'un plus long chemin d'exécution dans le graphe de tâches. Cette valeur ne contient pas le surcoût d'ordonnancement. Nécessairement :

$$T_\infty \leq T_1$$

– p, le nombre de processeurs physiques de la machine parallèle. Ces p processeurs sont considérés comme identiques et l'accès à la mémoire est uniforme : la machine considérée est de type *SMP (symmetric multiprocessors)*.

– T_p, la durée d'exécution sur p processeurs. En particulier, comme T_1 et T_∞ sont des invariants :

$$\max\left(\frac{T_1}{p}, T_\infty\right) \leq T_p$$

– S_p, la consommation mémoire de l'exécution sur p processeurs.

– $\bar{p} = \frac{T_1}{T_\infty}$, le degré de parallélisme de l'application, ou accélération maximale possible puisque nécessairement $\frac{T_1}{T_p} \leq \frac{T_1}{T_\infty} = \bar{p}$. On définit également le degré de liberté du parallélisme, ou *parallel slackness* [103], comme étant le rapport $\frac{\bar{p}}{p}$.

– $a_p = \frac{T_1}{T_p}$, l'accélération de l'exécution. Nécessairement :

$$a_p \leq \min(p, \bar{p})$$

[1]Ce qui est faux en toute rigueur : même si l'application est entièrement déterministe, la durée d'exécution de certaines instructions peut dépendre du passé de l'exécution. C'est typiquement le cas des actions sur la mémoire (`new`, `delete`) dont la durée n'est en général pas constante mais amortie.

6.3 Modèle de coût

Le modèle de coût associé à un langage permet de déterminer *a priori* la performance de l'exécution effective d'une application, et ce à partir de la seule connaissance du graphe caractéristique de l'exécution. La performance d'une exécution est caractérisée par sa durée et le volume mémoire qui lui a été nécessaire.

Nous présentons tout d'abord une technique simple permettant d'atteindre une accélération linéaire quasi optimale en temps puis les problèmes liés à la consommation mémoire. Le but des langages est de permettre[2] une accélération linéaire en temps, $T_p = O\left(\frac{T_1}{p}\right)$, sans pour autant pénaliser l'exécution par une consommation incontrôlée de mémoire.

6.3.1 Performances en temps

L'efficacité en temps est facilement obtenue par des techniques simples d'ordonnancement en ligne permettant d'atteindre des efficacités quasi optimales. Le surcoût d'ordonnancement doit cependant être maîtrisé afin de ne pas constituer l'essentiel de la durée d'exécution.

6.3.1.1 Algorithmes gloutons

Les algorithmes gloutons d'ordonnancement permettent d'obtenir des temps d'exécution situées à un facteur 2 de l'optimum [63, 64, 17, 94]. Le principe de base de ces algorithmes est de maintenir au maximum les processeurs en activité : chaque fois qu'un processeur devient inactif, une tâche prête à être exécutée (s'il en existe) lui est donnée. Ces algorithmes sont tels qu'à chaque étape de l'exécution, si il existe au moins p tâches prêtes alors aucun processeur n'est inactif, sinon toutes les tâches prêtes sont en cours d'exécution. Le théorème suivant ne tient compte ni des communications ni du surcoût d'ordonnancement.

Théorème 3 *[16] Quel que soit le nombre p de processeurs de la machine, toute application de travail T_1 et de profondeur T_∞ ordonnancée par un algorithme d'ordonnancement glouton s'exécute en un temps T_p tel que $T_p \leq \frac{T_1}{p} + T_\infty$.*

Nous rappelons la preuve donnée dans [63, 72] car son schéma intervient dans la démonstration de la proposition 10 page 125.

Preuve. Soit t_1 une tâche qui s'est terminée à la date T_p, et soit d_1 la date du début de cette tâche. Nous nous intéressons à ce qui s'est passé *avant* la date d_1. Deux cas peuvent être distingués :

1. Soit aucun processeur n'a été inactif avant d_1.
2. Soit au contraire il existe une date $d < d_1$ à laquelle au moins un processeur était inactif. Soit d' la plus grande de ces dates. L'algorithme étant glouton, si à la date d' la tâche t_1 avait été prête alors elle aurait débuté son exécution. Il existe donc une

[2] Au minimum de manière asymptotique.

tâche t_2 en cours d'exécution à la date d' telle que $t_2 \prec_G t_1$. Soit d_2 la date de début d'exécution de cette tâche.

L'application récursive de ce schéma permet de construire une séquence $t_k \prec_G \cdots \prec_G t_2 \prec_G t_1$ de tâches telles qu'à tout instant de l'exécution, soit tous les processeurs sont actifs, soit un des processeurs est en train d'exécuter une des tâches de cette séquence. La durée de la première situation est majorée par $\frac{T_1}{p}$ et celle de la seconde par la durée d'un chemin critique du graphe, c'est-à-dire par T_∞. Ainsi $T_p \leq \frac{T_1}{p} + T_\infty$. □

Il est à noter que ce résultat ne tient pas compte du surcoût dû à l'implantation de l'algorithme permettant d'assurer le caractère glouton de l'ordonnancement.

6.3.1.2 Surcoût de l'ordonnancement

Le coût d'ordonnancement comprend le maintien de l'ensemble des tâches prêtes ainsi que l'attribution des tâches de cet ensemble aux processeurs.

Dans le cas du langage Cilk [14] pour des graphes de type série-parallèle, ce coût peut être majoré par $O(T_\infty)$ en utilisant un ordonnancement glouton basé sur une technique de vol de travail [16, 13].

Dans le cas de NESL, l'ordonnancement est effectué par étapes [12]. Le calcul du placement pour une étape est effectué [11] en un temps $\log p$. Le nombre d'étape est de l'ordre de $\frac{T_1}{p \log p} + T_\infty$, le coût de l'ordonnancement est donc majoré par $O\left(\frac{T_1}{p} + (\log p) T_\infty\right)$.

6.3.2 Performance en mémoire

L'obtention d'une efficacité linéaire en temps, c'est-à-dire $T_p = O\left(\frac{T_1}{p}\right)$ sur une machine à p processeurs, nécessite de maintenir actif les processeurs la majeure partie du temps. L'exécution en parallèle de tâches nécessite en général un espace mémoire plus important qu'une exécution séquentielle car, outre le fait qu'il faille maintenir les structures de données caractérisant les tâches, des allocations mémoire peuvent avoir lieu en concurrence (tandis que lors d'une exécution séquentielle chaque allocation aurait pu être suivie d'une libération). Nous montrons dans la suite de cette section, à travers l'étude d'un exemple, qu'il n'est pas possible dans un cas général d'obtenir une accélération linéaire en temps d'exécution tout en limitant l'espace mémoire nécessaire à cette exécution [16]. Cependant, des limitations sur le parallélisme exploité, et donc une restriction de la classe des graphes d'exécution possibles, permettent de contrôler cet espace mémoire nécessaire.

6.3.2.1 Cas général

Nous montrons dans cette section qu'il n'est pas possible, dans un cas général, d'obtenir à la fois une efficacité linéaire en temps et une consommation raisonnable de mémoire.

Proposition 9 *Pour toute durée T_1 et tout espace mémoire S_1 il existe des applications de durée d'exécution T_1 et de consommation mémoire S_1 sur 1 processeur telles que*

$T_\infty \approx \sqrt{T_1}$ et pour tout p :

$$T_p \leq T_1/2 \Longrightarrow S_p \geq \frac{\sqrt{T_1}}{2} S_1.$$

Les propriétés de cette application sont telles que $\bar{p} \approx \sqrt{T_1}$, ce qui laisse l'utilisateur en droit d'attendre une accélération raisonnable pour toute exécution parallèle.
Preuve. Étant donnés T_1 et S_1, nous exhibons une application possédant les caractéristiques exposées dans la proposition. Cette application est une version simplifiée de celle présentée dans [16]. Cette application, représentée figure 6.1 page 121, est composée de :

- $\frac{\sqrt{T_1}}{2}$ tâches W chacune de durée $\sqrt{T_1}$ et allouant un espace mémoire de taille 0.
- $\frac{T_1}{4}$ tâches A de durée 1 et allouant chacune un espace mémoire de taille S_1.
- $\frac{T_1}{4}$ tâches L de durée 1 et chacune libérant un espace mémoire de taille S_1.

Les tâches A et L sont appariées, ces paires étant elles-mêmes groupées par tranches de $\frac{\sqrt{T_1}}{2}$: il y a donc $\frac{\sqrt{T_1}}{2}$ telles tranches. À chacune de ces tranches est associée une tâche de type W. Les dépendances entre les tâches, représentées figure 6.1 page 121 sont les suivantes :

- Les tâches A n'ont aucune contrainte de précédence.
- Chaque tâche W doit attendre l'exécution des $\frac{\sqrt{T_1}}{2}$ tâches A de la tranche qui la précède avant de débuter son exécution. La tâche W de la première tranche n'a aucune contrainte de précédence.
- Chaque tâche L doit attendre la terminaison de la tâche W associée à la tranche et la terminaison de la tâche A qui lui est appariée.

Il est aisé de vérifier que l'exécution sur un processeur unique peut être effectuée en temps T_1 dans un espace mémoire S_1 : il suffit pour cela d'exécuter les tranches les unes après les autres, et pour chaque tranche d'abord la tâche W puis pour chaque paire la tâche d'allocation A puis la tâche de libération L.

En remarquant que les tâches A, L et W sont indépendantes entre-elles, que les tâches A n'ont aucune contrainte de précédence, que les tâches W n'ont de dépendance qu'avec les tâches A et enfin que les tâches L sont dépendantes des tâches A et W, il vient que $T_\infty = \sqrt{T_1} + 2$. Il est donc *a priori* possible d'atteindre des accélérations allant jusqu'à $\bar{p} = \frac{T_1}{T_\infty} = \frac{1}{1+\frac{2}{\sqrt{T_1}}}\sqrt{T_1}$.

Considérons l'exécution de cette application sur une machine possédant p processeurs. Si aucune des tâches W n'est exécutée en concurence, alors nécessairement $T_p \geq \frac{\sqrt{T_1}}{2} \times \sqrt{T_1}$, et donc $\frac{T_1}{T_p} \leq 2$. Obtenir une meilleure performance implique donc l'exécution en concurrence d'au moins deux tâches W. Considérons dans la suite les tâches W numérotées, W_i avec $1 \leq i \leq \frac{\sqrt{T_1}}{2}$, selon l'ordre de gauche à droite de la figure 6.1 page 121.

Plaçons nous à un instant t où deux tâches W_i et W_j, $i < j$, s'exécutent en concurrence. Comme W_j est en cours d'exécution, les $\frac{\sqrt{T_1}}{2}$ tâches A qui précèdent son exécution sont terminées. Étudions l'état de la tâche W_{j-1}. Si cette tâche n'a pas encore commencé

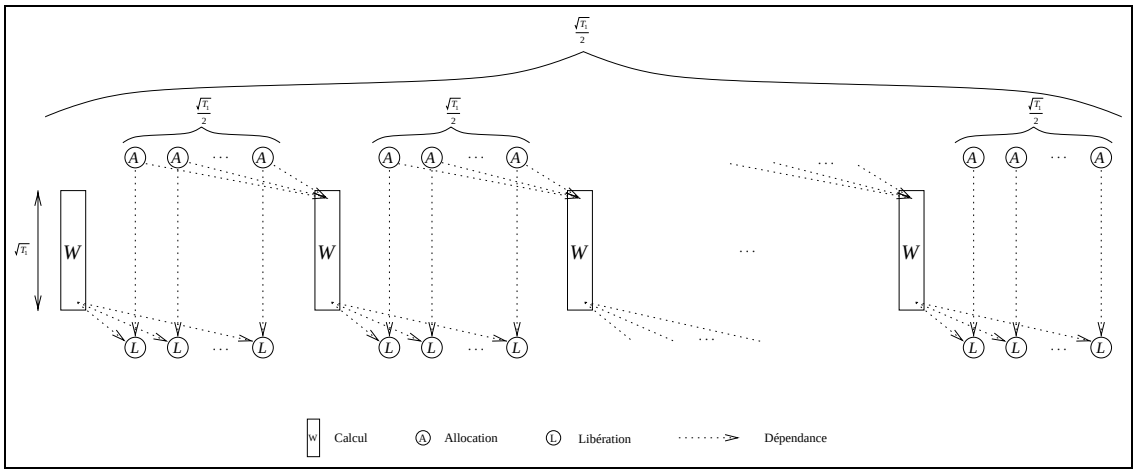


Figure 6.1 *Application consommatrice de mémoire en cas d'accélération lors d'une exécution parallèle.*
Cette application, s'exécutant sur un processeur en temps T_1 et consommant un volume mémoire S_1, est constituée de $\frac{\sqrt{T_1}}{2}$ tâches W de travail, de $\frac{T_1}{4}$ tâches d'allocation et de $\frac{T_1}{4}$ tâches de libération. Les contraintes de précédence entre ces tâches sont telles que toute exécution parallèle vérifiant $T_p < \frac{T_1}{2}$ implique une consommation mémoire $S_p \geq \frac{\sqrt{T_1}}{2} S_1$.

son exécution ou est en cours d'exécution (par exemple si $j = i + 1$), alors les tâches L correspondant aux allocations n'ont pas pu être exécutées : le volume mémoire correspondant à ces allocations est $\frac{\sqrt{T_1}}{2} S_1$. Si par contre cette tâche W_{j-1} est terminée, on peut supposer que les tâches L ont été exécutées. Mais la terminaison de cette tâche implique que les $\frac{\sqrt{T_1}}{2}$ tâches A qui la précède ont été terminées. On se retrouve donc dans la situation précédente mais cette fois avec la tâche d'indice $j - 1$. L'itération de cette analyse se terminera nécessairement puisque W_i n'est pas terminée mais en cours d'exécution.

Pour cette application, toute efficacité supérieure à 2 implique donc une consommation mémoire lors de l'exécution parallèle importante : $S_p \geq \frac{\sqrt{T_1}}{2} S_1$. □

L'application réelle doit créer l'ensemble de ces tâches. Le travail associé à cette création est égal au nombre de tâches, c'est-à-dire à $\frac{T_1}{2} + \frac{\sqrt{T_1}}{2}$. Cette création peut par exemple être menée en parallèle pour chaque tranche et de manière séquentielle à l'intérieur de chaque tranche, la profondeur de la création est alors de l'ordre de $\sqrt{T_1}$. Cette phase de création ne modifie donc pas les caractéristiques de l'application (T_1 et T_∞ restent du même ordre).

6.3.2.2 Restriction sur la classe des graphes d'exécution générables

En limitant les possibilités de synchronisations entre les tâches, il est possible d'interdire la construction d'applications telles que celle présentée dans la section 6.3.2.1 page 119. Un ordre d'exécution particulier des tâches du graphe permet alors la prédiction de l'espace mémoire nécessaire à l'exécution.

Le langage Cilk n'autorise que la programmation d'applications *strictes*, applications

dans lesquelles toute tâche ne peut imposer une contrainte de précédence que sur la continuation de la tâche mère qui l'a créée. L'application précédente ne peut donc plus être codée dans un tel modèle. Avec de telles restrictions il est possible de construire un algorithme d'ordonnancement [15] tel que l'espace mémoire nécessaire à l'exécution de l'application sur p processeurs avec une accélération linéaire $\frac{T_1}{T_\infty} = \Omega(p)$ est telle que $S_p \leq pS_1$. Ce résultat est obtenu en effectuant un parcours de type profondeur d'abord du graphe de tâches : lorsque sur un processeur une tâche crée une nouvelle tâche, alors le processeur continue l'exécution avec la tâche nouvellement créée. À tout instant de l'exécution le graphe en cours de développement possède donc au plus p feuilles et chacune de ces feuilles est en cours d'exécution sur un processeur. Chaque feuille nécessitant un espace mémoire inférieur à S_1 pour s'exécuter, il vient que $S_p \leq pS_1$.

Le langage NESL impose également un type de graphe série-parallèle et l'ordre d'exécution des tâches est un parcours en profondeur du graphe (parallélisme emboîté), parcours exécuté par p processeurs [82]. Un tel ordre d'exécution permet de garantir [12] que l'ordonnancement sur p processeurs de toute application est tel que $S_p \leq O(S_1 + p \log pT_\infty)$ avec $T_p \leq O\left(\frac{T_1}{p} + \log pT_\infty\right)$.

6.3.2.3 Utilisation d'un ordre séquentiel implicite

Comme nous l'avons vu précédemment, la consommation mémoire S_p d'une exécution parallèle est comparée à la consommation mémoire d'une exécution séquentielle sur un processeur de la même instance d'application.

La consommation mémoire d'une application dépend énormément de l'ordre d'exécution des tâches. Considérons par exemple l'application possédant n tâches a_i allouant 1 bit et n tâches l_i effectuant les libérations correspondantes, telles que les seules contraintes de précédence soient du type $a_i \prec l_i$. Considérons les deux ordres d'exécution séquentiels suivants :

$$a_1 < b_1 < \ldots < a_i < b_i < \ldots < a_n < b_n \quad (1)$$
$$a_1 < \ldots < a_i < \ldots < a_n < b_1 < \ldots < b_i < \ldots < b_n \quad (2)$$

Le volume mémoire requis pour l'exécution correspondant à l'ordre (1) est de 1 bit, tandis que celui correspondant à l'ordre (2) est de n bits. Le volume mémoire requis dépendant fortement de l'ordre, le principe pour contrôler cette consommation lors d'une exécution parallèle est de **suivre l'ordre séquentiel** qui a été utilisé pour définir S_1 [83]. Il est alors possible de comparer S_p à S_1.

Nous présentons ici la stratégie d'ordonnancement présentée dans [83, 9] et qui permet de limiter la consommation mémoire de toute exécution parallèle par $S_1 + pKT_\infty$. Le parallélisme est de type série-parallèle avec une description emboîtée du graphe. De plus chaque tâche est supposée de durée unitaire et alloue au maximum un volume K de mémoire.

Cette stratégie maintient une liste de tâches prêtes ordonnées par priorité. La priorité de chaque tâche correspond à son numéro dans l'ordre d'exécution séquentiel (le

parcours en profondeur d'abord du graphe)[3]. Les tâches les plus vieilles dans l'ordre séquentiel sont donc plus prioritaires que les autres. Il est montré [83] qu'au plus T_∞ tâches par processeur peuvent être exécutées simultanément en avance sur leur numéro d'ordre. Chaque tâche allouant au plus un volume K de mémoire, le volume consommé en plus de l'exécution séquentielle est donc majoré par pKT_∞.

Il est possible d'étendre cette preuve aux tâches allouant une volume m arbitraire de mémoire : il suffit d'introduire $\frac{m}{K}$ tâches fictives avant cette tâche d'allocation. Les tâches effectuant de grosses allocations seront alors précédées par un ensemble de ces tâches fictives qui vont avoir pour effet de retarder leur exécution. Ainsi, ces allocations étant retardées, elles n'auront que peu de chance d'être effectuées en parallèle. Ainsi, dans le cas de l'exemple de la section 6.3.2.1 page 119, chacune des tâches A seraient précédées par un ensemble de tâches fictives, ce qui aurait pour effet de retarder leur exécution jusqu'à, par exemple, la terminaison de la tâche W. Une tâche de libération L pourra donc être exécutée après chaque exécution de tâche d'allocation.

6.3.3 Bilan

La durée de l'exécution d'une application peut facilement être obtenues et, moyennant certaines restrictions sur le graphe d'exécution et en se basant sur un ordre séquentiel implicite des tâches, la consommation mémoire controlée. Ces garanties permettent donc d'associer un modèle de coût au modèle de programmation du langage.

Nous présentons dans la suite le cas de l'interface applicative Athapascan-1 pour laquelle un modèle de coût, basé sur la description de l'exécution par un graphe de flot de données, est défini. Les performances dépendant de l'ordonnancement et l'ordonnancement étant séparé de la bibliothèque, les garanties offertes par le modèle de coût dépendent donc de la politique d'ordonnancement choisie lors de l'exécution.

6.4 Contrôle de la consommation mémoire d'un programme Athapascan-1

Les performances de l'exécution des applications dépendent directement de la politique d'ordonnancement utilisée. L'ordonnancement est effectué dans Athapascan-1 par un module séparé de la bibliothèque. Ce module est présenté section 7.2 page 130. Nous décrivons dans cette section deux politiques d'ordonnancement qui permettent de garantir les performances de l'exécution au niveau de la consommation mémoire.

L'exécution est supposée être effectuée sur une machine à mémoire distribuée possédant p processeurs identiques. Afin de permettre la majoration des temps d'accès aux mémoires distantes, les algorithmes proposés dans [71] sont utilisés pour simuler une mémoire globale pour les p processeurs à l'aide de fonctions de hachage universel. Afin d'obtenir une simulation quasi-optimale (le délai h ne dépendant quasiment pas de p) une

[3]Le numéro n'est pas connu en réalité, mais l'ordre est maintenu par une technique de chaînage de la liste : les tâches filles sont insérées à l'ancienne position de la mère dans la liste.

technique de multiprogrammation légère permettant d'exploiter le degré de parallélisme, *parallel slackness* [89, 103], est utilisée. Cette technique consiste en l'émulation préemptive de plusieurs processeurs virtuels sur chaque processeur physique de la machine.

Nous reprenons les grandeurs caractéristiques de l'exécution présentées section 6.2 page 116. L'ordre séquentiel considéré pour la définition de ces grandeurs est l'ordre de référence de la définition 7 page 70.

À ces grandeurs caractéristiques nous ajoutons les grandeurs suivantes afin de tenir compte de la particularité de la machine qui est à mémoire distribuée et des communications nécessaires :

- q, le nombre de processeurs virtuels émulés sur les p processeurs réels de la machine. Nécessairement :
$$p \leq q$$
- h, le délai d'accès à distance d'un bit de donnée.
- C_1, le volume total d'accès distant. Cette valeur représente la somme sur tout le graphe d'exécution G des tailles des données accédées en lecture directe (mode d'accès `r` ou `r_w`).
- C_∞, le volume d'accès distant effectué par un plus long chemin dans ce graphe (selon ce critère d'accès).
- σ, la taille du graphe G.
$$\sigma = |V| + |E|$$
Le graphe nécessite donc σ opérations pour être construit. La gestion des états associés aux nœuds permettant de résoudre les contraintes de précédence est basée sur un mécanisme de terminaison et correspond donc à une « dé-construction » du graphe : il y a donc également σ telles actions.

La première technique, notée $\mathcal{O}_1$ et présentée dans la définition 8 page 124, est l'utilisation directe de l'algorithme glouton présenté section 6.3.1.1 page 118. Les performances de l'exécution sont celles de la proposition 10 page 125. La seconde, notée $\mathcal{O}_2$ et présentée dans la définition 9 page 126, est une restriction du modèle de programmation au cadre des graphes de type série-parallèle, ce qui permet d'utiliser une politique d'ordonnancement basée sur le vol de travail similaire à celle utilisée dans le langage Cilk [51]. Les performances de l'exécution sont celles de la proposition 11 page 126.

6.4.1 Politique d'ordonnancement $\mathcal{O}_1$

Nous donnons dans cette section la définition de cette politique et évaluons théoriquement ses performances. Cet algorithme est un algorithme de type liste centralisée ordonnée avec une priorité donnée par l'ordre de « référence » $\prec_r$.

Définition 8 *La **politique d'ordonnancement** $\mathcal{O}_1$ consiste à exécuter de manière gloutonne les tâches prêtes en suivant au plus prêt l'ordre de « référence » $\prec_r$.*

L'implantation maintient de manière centralisée un graphe $G' \subset G$ représentant la portion créée et non encore terminée de G. Une liste P_I de processeurs inactifs, et une

liste $R_{\prec_r}$ de tâches prêtes, triées selon $\prec_r$, sont également maintenues. La cohérence de ces structures est assurée par un verrou global qui sera pris avant chaque modification.

L'ordonnancement est effectué de la manière suivante :

- *Chaque fois qu'un processeur devient inactif, il prend la première tâche de la liste $R_{\prec_r}$ et l'exécute. Si cette liste est vide il s'insère dans la liste P_I des processeurs inactifs.*
- *Chaque fois qu'une tâche devient prête, elle est affectée à l'un des processeurs de la liste P_I. Si cette liste est vide, la tâche est insérée (en temps constant) dans la liste $R_{\prec_r}$ de tâches prêtes.*

Proposition 10 *[72] L'exécution de toute application ordonnancée par la politique d'ordonnancement $\mathcal{O}_1$ est telle que (les coûts d'ordonnancement et de communication sont pris en compte, chaque tâche est supposée allouer un volume borné de mémoire) :*

$$\begin{aligned} T_p &\leq \frac{T_1 + hC_1}{p} + \frac{q}{p}(T_\infty + hC_\infty) + h\frac{q}{p}O(\sigma) \\ S_p &\leq S_1 + qO(T_\infty + hC_\infty) \end{aligned}$$

Preuve. En reprenant la démarche utilisée dans [64, 72] et dans la preuve du théorème 3 page 118, l'ensemble des tops de l'exécution peut être partitionné en trois sous-ensembles :

1. A (activité) : pour tout top de A, tous les q processeurs de la machine sont en train soit d'exécuter une instruction de l'application soit d'effectuer un accès à distance. Nécessairement :
$$|A| \leq \frac{T_1 + hC_1}{q}$$
2. I (inactivité) : pour tout top de I, un au moins des q processeurs est inactif ($|P_I| \neq 0$). En utilisant la même technique que celle utilisée dans la preuve du théorème 3 page 118 il vient :
$$|I| \leq \text{durée sur une infinité de processeurs} \leq T_\infty + hC_\infty$$
3. Q (ordonnancement) : pour tout top de Q, un au moins des q processeurs est en train de manipuler les structures maintenues globalement. Les accès concurrents sont sérialisés en utilisant le verrou global. La durée de chaque manipulation est majorée par $O(h)$. Il y a au plus 2σ manipulations des structures, donc :
$$|Q| \leq O(h\sigma)$$

La durée sur q processeurs $T_q = |A| + |Q| + |I|$ est donc majorée par $\frac{T_1+hC_1}{q} + T_\infty + hC_\infty + O(h\sigma)$. En remarquant que la durée d'exécution sur les p processeurs réels de la machine est telle que $T_p \leq \left\lceil \frac{q}{p} \right\rceil T_q$, on obtient la borne annoncée.

L'ordonnancement précédent exécute les tâches en suivant au plus près l'ordre de « référence » : il y a donc, à chaque top, au plus $q-1$ tâches qui sont exécutées en avance sur cet ordre, donc, à tout instant, un maximum de $(T_\infty + hC_\infty)(q-1)$. Chaque tâche étant supposée allouer une quantité bornée de mémoire, on obtient la majoration de l'espace mémoire annoncée. □

6.4.2 Politique d'ordonnancement $\mathcal{O}_2$

Nous donnons dans cette section la définition de cette politique et évaluons théoriquement ses performances. Cet algorithme est, à la base, un algorithme glouton auquel on impose aux vols de « suivre » *a priori* l'ordre de « référence ». Cette technique est utilisée dans le langage Cilk et entièrement détaillée dans [16].

Définition 9 *La **politique d'ordonnancement** $\mathcal{O}_2$ consiste à exécuter de manière gloutonne les tâches prêtes en suivant au plus prêt l'ordre séquentiel de création des tâches en élargissant les contraintes de précédence (définition 6 page 69) à tous les accès en lecture, (directs et différés).*

L'implantation maintient de manière distribuée un graphe $G' \subset G$ représentant la portion créée et non encore terminée de G. Chacun des q processeurs possède un ensemble de branches du graphe qui a été développé localement. Les tâches sont ordonnées dans chaque branche selon l'ordre d'un parcours en profondeur d'abord du graphe.

L'ordonnancement est effectué comme suit. Chaque fois qu'un processeur p termine une tâche t, il peut être dans l'une des deux situations suivantes :

- *Une tâche t' suit t dans la branche et t' est prête.*
- *La tâche qui suivait t a été volée par le processeur p'.*

Dans la première situation, p prend t' et l'exécute. Dans la seconde, il va voir sur p' si le reste de la branche qui a été volé est exécutable (c'est-à-dire si la première tâche de cette branche est prête et si cette branche n'est pas celle en cours d'exécution sur p'). Si oui, toute cette branche est volée par p qui continue son exécution avec la première tâche de son vol. Dans tous les autres cas, p vole à un processeur quelconque une tâche prête (et tout le reste de la branche) puis l'exécute.

Les nouvelles contraintes de précédence imposées sur les tâches sont telles que si une tâche est prête, alors une exécution séquentielle de toute la descendance de cette tâche est possible sans synchronisation. C'est ce qui permet l'exécution en profondeur de toute portion du graphe.

Proposition 11 *L'exécution de toute application ordonnancée par la politique d'ordonnancement $\mathcal{O}_2$ est telle que (les coûts d'ordonnancement et de communication sont pris en compte) :*

$$\begin{aligned} T_p &\leq \frac{T_1 + hC_1}{p} + h\frac{q}{p}O(T_\infty + hC_\infty) \\ S_p &\leq qS_1 \end{aligned}$$

Preuve. Cet ordonnancement étant glouton, les performances en temps sont données par le théorème 3 page 118. Le coût de l'implantation de cet ordonnancement est de l'ordre du nombre de tâches volées par chaque processeur. Or chaque processeur ne peut effectuer plus de $T_\infty + hC_\infty$ requêtes de vol et la durée de chaque vol est constante avec un grande probabilité. Nous ne détaillons pas ici le calcul de cette probabilité, calcul qui est effectué dans [15].

L'ordonnancement précédent exécute les tâches en suivant au plus près l'ordre séquentiel de création des tâches. L'exécution de toute la descendance d'une tâche requiert donc un volume mémoire majoré par S_1. On vérifie aisément qu'il y a au plus q descendances qui sont exécutées en concurrence, un processeur arrêtant l'exécution d'une branche uniquement dans le cas où un autre processeur est en cours d'exécution sur la continuation de cette même branche. L'espace mémoire nécessaire à l'exécution peut donc être majoré par qS_1. Il est à noter que cette borne est globale et ne garantit absolument pas que la mémoire utilisée par chacun des q processeurs est majorée par S_1. □

6.5 Bilan

L'exécution d'une application sur une machine parallèle est modélisée par un graphe de flot de données. Ce graphe décrit les contraintes de précédence entre les tâches de calcul et les accès aux données. Étant donné un graphe, les politiques d'ordonnancement utilisées dans les langages de « haut niveau » garantissent les performances en temps et en mémoire de toute exécution parallèle de ce graphe. Le tableau 6.1 page 127 récapitule les garanties d'efficacité d'ordonnancement fournies par les modèles de coût associés aux langages étudiés.

Langage	**Restriction sur le graphe**	T_p **majoré par**	S_p **majoré par**
Cilk	série-parallèle	$\frac{T_1}{p} + O\left(T_\infty\right)$	pS_1
NESL	série-parallèle emboîté	$O\left(\frac{T_1}{p} + \log pT_\infty\right)$	$O(S_1 + p \log pT_\infty)$
Athapascan-1	description emboîté	$\frac{T_1+hC_1}{p} + \frac{q}{p}(T_\infty + hC_\infty) + h\frac{q}{p}O\left(\sigma\right)$	$S_1 + qO(T_\infty + hC_\infty)$
	série-parallèle emboîté	$\frac{T_1+hC_1}{p} + h\frac{q}{p}O(T_\infty + hC_\infty)$	qS_1

Tableau 6.1 *Garanties d'efficacité de Cilk, NESL et Athapascan-1.*
Les architectures visées sont des machines à mémoire partagée pour Cilk et NESL et les machines à mémoire distribuée pour Athapascan-1. Dans le cas des machines à mémoire distribuée, le délai d'accès à un bit de donnée distant est supposé majoré par h : ceci est effectué en émulant q processeurs virtuels sur les p processeurs réels de la machine. Notons que seul Athapascan-1 tient compte des communications dans la formule de coût : Cilk est conçu pour les machines de type SMP *et NESL les ignore.*

Les techniques d'ordonnancement en ligne de type « glouton » où les processeurs sont maintenus au maximum en activité permettent d'atteindre des efficacités en temps quasi-optimales, de l'ordre de $\frac{T_1}{p} + T_\infty$. Si l'efficacité en temps peut être obtenue relativement facilement, il n'en est pas de même pour l'efficacité en mémoire. Différentes variantes des algorithmes gloutons permettent de garantir une majoration de la consommation mémoire, le principe de base étant de suivre au maximum l'ordre séquentiel d'exécution. Le respect de cet ordre sur une machine à plusieurs processeurs introduit cependant un coût

supplémentaire d'ordonnancement. NESL par exemple suit de plus près l'ordre séquentiel que Cilk, ce qui lui permet de majorer l'utilisation mémoire par $S_1 + p \log p T_\infty$ tandis que Cilk majore par pS_1. Le gain est substantiel pour des applications pour lesquelles $T_\infty \leq S_1$, ce qui est le cas par exemple pour le produit de deux matrices $n \times n$ pour lequel $T_\infty = \log n$ et $S_1 = n^2$. Cet ordonnancement a cependant un coût $\log p$ fois plus important.

Dans le cas d'Athapascan-1, le coût de cette technique peut être de l'ordre de la taille σ du graphe, taille qui peut être de l'ordre de T_1 pour certaines applications (le calcul de Fibonacci à un grain fin par exemple). Il est possible, en imposant des contraintes plus fortes de précédence entre les tâches, d'atteindre les mêmes performances que celles garanties par le langage Cilk ($\mathcal{O}_2$).

Nous présentons dans le chapitre suivant l'implantation dans Athapascan-1 des deux politiques d'ordonnancement $\mathcal{O}_1$ et $\mathcal{O}_2$ permettant de contrôler la mémoire requise pour toute exécution.

7

Implantation et évaluation du contrôle de la consommation mémoire dans Athapascan-1

Ce chapitre étudie l'ordonnancement dans la version distribuée de la bibliothèque Athapascan-1. Les principales sections de ce chapitre traitent les points suivants :

- Le **module d'ordonnancement** d'Athapascan-1 qui est entièrement séparé du reste ce qui permet une programmation aisée de nouvelles stratégies (section 7.2 page 130).
- L'**implantation** des stratégies d'ordonnancement $\mathcal{O}_1$ et $\mathcal{O}_2$ (section 7.3 page 137).
- L'**évaluation** de ces deux stratégies sur un exemple simple de calcul d'une portion de l'ensemble de Mandelbrot (section 7.4 page 139).

7.1 Le problème de l'ordonnancement

Les applications s'exécutant sur une machine parallèle sont décomposées en tâches qui s'exécutent simultanément sur les différents processeurs de la machine. Une fois les tâches déterminées et créées, leur ordonnancement est généralement un des problèmes majeurs posés par l'utilisation d'une machine parallèle. La qualité de l'ordonnancement a une influence cruciale sur les performances de l'exécution, mais le calcul d'un « bon » ordonnancement des tâches implique un surcoût. En effet, notons W_1 le travail de l'application et W_o celui effectué par l'ordonnancement. Alors nécesssairement $T_p \geq \frac{W_1+W_o}{p}$ et donc supérieur à $T_p^{opt} + \frac{W_o}{p}$.

Face à l'importance de ce problème de nombreuses stratégies d'ordonnancement ont été développées. Deux principaux types de stratégies sont à distinguer [24] : les stratégies de type **statique** où l'ordonnancement est calculé **avant** l'exécution et les stratégies de type **dynamique** où l'ordonnancement est cette fois calculé **au cours** de l'exécution. Les stratégies de type statique [60, 108] ont l'avantage de n'introduire aucun surcoût d'ordonnancement lors de l'exécution mais ne sont véritablement efficaces que dans le cadre des

applications régulières [58, 95] où le graphe des tâches de l'application ne dépend pas des conditions d'exécution. À l'opposé, les stratégies de type dynamique, où l'ordonnancement des tâches est calculé à la volée, permettent une bonne adaptation aux conditions d'exécution. Ces stratégies n'ont cependant qu'une vue partielle de l'application qui doit être ordonnancée et introduisent un surcoût à l'exécution.

La technique d'ordonnancement la plus efficace dépend hélas de l'application considérée [58, 18], et l'obtention d'une efficacité maximale passe donc par le codage de la stratégie d'ordonnancement dans l'algorithme de calcul. Cette programmation est cependant en général délicate et peut dans certains cas accroître significativement la complexité de la programmation. De plus, les stratégies simples et adaptatives, telles que les algorithmes gloutons présentés section 6.3.1.1 page 118 risquent d'être programmées et reprogrammées de nombreuses fois. C'est pourquoi les langages parallèles de « haut niveau » tentent d'offrir des outils permettant un ordonnancement efficace des tâches sans être obligé de se poser le problème de la programmation de la stratégie. Par exemple l'environnement Pyrros [60] pour un ordonnancement statique, ou les langages Cilk, NESL, Jade, Sisal [48] qui intègrent dans leur noyau une politique d'ordonnancement. Cette solution, si elle permet d'affranchir l'utilisateur du problème d'ordonnancement, s'avère limitée si les conditions d'exécution ne sont pas celles de prédilection de l'algorithme choisi pour faire la régulation : c'est entre autres le problème dont souffre la version distribuée [90] du langage Cilk. Une autre alternative est de séparer entièrement la politique d'ordonnancement de l'application [107, 106, 58, 34] et de proposer non pas une seule et unique stratégie d'ordonnancement, mais tout un ensemble : l'utilisateur décidera, éventuellement expérimentalement, quelle est la stratégie la mieux adaptée à sa situation. C'est la solution retenue par les environnements de programmation tels que Dynamo [102], PAC++-Givaro [57, 56] ou Athapascan-1.

Nous présentons dans la suite de ce chapitre tout d'abord l'environnement d'ordonnancement défini dans [29, 26, 25] constituant le module d'ordonnancement utilisé au sein de la bibliothèque Athapascan-1, puis l'implantation à l'aide de cet environnement des trois stratégies d'ordonnancement présentées au cours du chapitre 6 page 115 : une stratégie gloutonne permettant d'obtenir une durée d'exécution théoriquement quasi-optimale et les deux stratégies introduites au chapitre précédent, $\mathcal{O}_1$ et $\mathcal{O}_2$, permettant de contrôler théoriquement l'espace mémoire nécessaire à l'exécution. Ces stratégies seront comparées expérimentalement en fonction du volume mémoire nécessaire à l'exécution de l'application sur plusieurs processeurs.

7.2 Implantation des politiques d'ordonnancement en Athapascan-1

L'ordonnancement est assuré dans Athapascan-1 par un module séparé de la bibliothèque. Nous détaillons dans cette section les interfaces des quatre différents composants du module d'ordonnancement :

– Le générateur de tâches, constitué par la bibliothèque Athapascan-1.

– Le module d'ordonnancement, chargé d'affecter un site d'exécution à chaque tâche.
– Le module d'exécution, chargé d'exécuter les tâches qui lui sont confiées.
– Le module d'information de charge, chargé de fournir des informations relatives à l'état d'activité de la machine.

La figure 7.1 page 131 représente l'interaction de ces différents modules au sein d'une architecture distribuée. Le principe de base de cette distribution repose sur une réplication de chacun des modules sur chacun des nœuds de la machine. Chaque version locale d'un module n'interagit qu'avec les réplicats locaux des autres modules (par appels de fonctions) et les autres réplicats de son module situés sur d'autres sites (par échange de messages).

Nous présentons dans la suite tout d'abord le noyau d'exécution parallèle Athapascan-0 sur lequel reposent les implantations d'Athapascan-1 et du module d'ordonnancement, puis nous détaillons les quatre composants de ce module ainsi que leurs interactions.

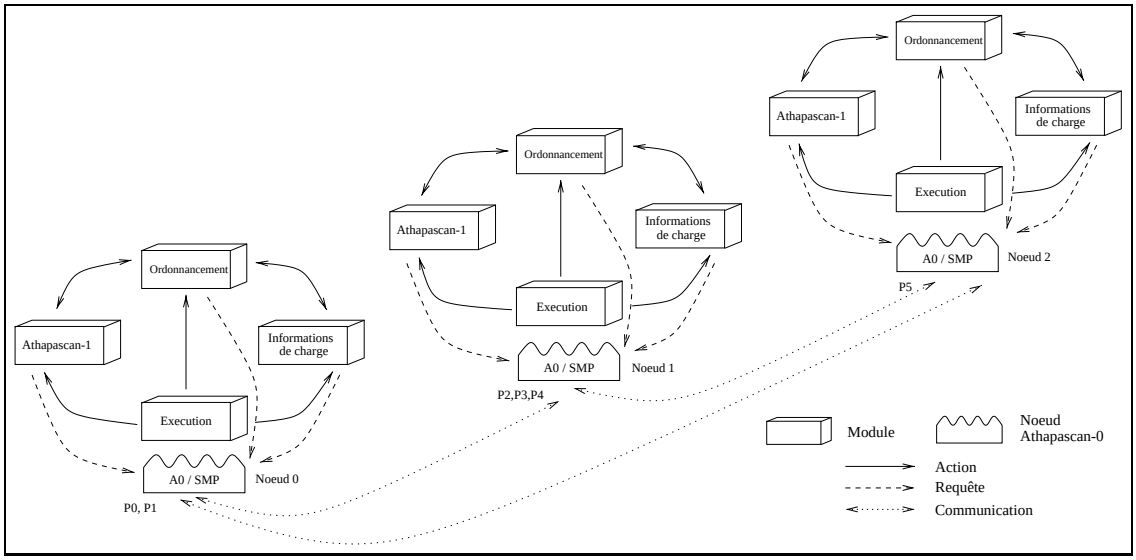


Figure 7.1 *Interaction des différents modules dans le cadre d'une machine distribuée. Les différents nœuds constituants la machine parallèle sont gérés par la bibliothèque Athapascan-0. Caque nœud peut contenir plusieurs processeurs physiques (ce qui permet d'exploiter les machines de type* SMP*). Chaque module est présent sur chacun des nœuds et les différents réplicats d'un même module peuvent communiquer au moyen des primitives de communications offertes par Athapascan-0. Le réplicat d'un module n'interagit qu'avec les représentants locaux des autres modules.*

7.2.1 Athapascan-0

Athapascan-0[1] [19, 61] est un noyau exécutif pour machines parallèles supportant la multiprogrammation légère (ou *multi-threading*). L'abstraction de la machine parallèle offerte par Athapascan-0 consiste en l'interconnexion par un réseau d'un ensemble de nœuds de calculs. Les primitives offertes par la bibliothèque permettent de créer des fils

[1] L'*URL* de la page officielle du projet est la suivante : `http://www-apache.imag.fr/software/ath0/`.

d'exécution de calcul (*threads*) sur chaque nœud et d'échanger des messages contenant des données entre des threads situés sur des nœuds différents.

Ce noyau exécutif permet d'exploiter quatre types de parallélisme pouvant être présents sur une machine parallèle :

- Le parallélisme inter-nœuds, chaque nœud exécutant une copie du noyau d'exécution Athapascan-0 et de l'application. Les nœuds progressent donc dans leurs exécutions indépendamment les uns des autres (mises à part les synchronisations dues aux communications).
- Le parallélisme intra-nœud, plusieurs fils d'exécution étant présents sur un même nœud. En effet, si ce nœud[2] possède plusieurs processeurs, ces fils d'exécution pourront être répartis entre ces processeurs par le système d'exploitation gérant ce nœud.
- Le parallélisme entre calculs et communications, en offrant des primitives asynchrones de communication. Ce type de parallélisme est également exploitable en utilisant des communications bloquantes et en s'appuyant sur le parallélisme intra-nœud (en utilisant alors plusieurs fils, certains calculant, d'autres étant synchronisés sur des communications).
- Le parallélisme entre les communications, si un fil d'exécution initie plusieurs communications non bloquantes ou si plusieurs fils d'exécution communiquent à partir du même nœud.

Cette bibliothèque constitue, en offrant exactement les primitives de multiprogrammation légère et de communication nécessaires, la couche de portabilité d'Athapascan-1. Il peut donc être installé sur toute machine possédant Athapascan-0 et un compilateur C++ supportant le mécanisme de classes « patrons »[3]. Athapascan-0 est implanté de telle manière que sa couche de portabilité est réduite à l'utilisation d'une bibliothèque de processus légers (généralement des threads POSIX [69]) et d'une bibliothèque de communications (généralement MPI [40]).

7.2.2 Le module générant les tâches : Athapascan-1

Au sein du système exécutif, le rôle de la bibliothèque Athapascan-1 est de créer des tâches. Ces tâches sont générées localement et font partie d'un graphe les reliant aux données partagées : ce graphe traduit les contraintes de précédence et de localité.

Bien que les tâches soient toujours générées localement, le graphe est réparti. Les tâches sont en effet placées sur les différents nœuds de la machine, selon les décisions du module d'ordonnancement. La gestion de la cohérence du graphe, le calcul des états de ses nœuds et la synthèse des données deviennent de ce fait des opérations distribuées.

[2] Un nœud Athapascan-0 est actuellement implanté comme un processus lourd UNIX. Si plusieurs processeurs physiques sont à la dispositions du système, le cas d'un SMP par exemple, les processus légers contenus dans le nœud seront répartis par le système. C'est la situation des nœuds 0 et 1 représentés figure 7.1 page 131.

[3] L'implantation d'Athapascan-1 utilise de manière intensive les classes « patrons », ou *template* [100], afin d'effectuer le maximum de l'analyse lors de la passe de compilation. Ces classes sont parfois mal supportées par les compilateurs. Les dernières versions publiques d'`egcs` semblent pourtant les supporter convenablement.

La gestion distribuée de ce graphe nécessite la coopération des différents réplicats du module de la bibliothèque. Cette coopération est effectuée au moyen des primitives de communications offertes par le noyau Athapascan-0.

7.2.3 Le module d'exécution

Le module d'exécution est composé d'un pool de processeurs virtuels. Ces processeurs sont implantés sous forme de fils d'exécution (*threads* Athapascan-0), ce qui permet d'exploiter le parallélisme intra-nœud et le recouvrement des communications par du calcul. Ces processeurs virtuels seront responsables de l'exécution effective des tâches.

7.2.3.1 Fonctionnement

Chaque processeur virtuel, autonome, exécute une boucle infinie, présentée figure 7.2 page 133, qui consiste à demander une tâche, l'exécuter puis... recommencer.

```
while( ! ended ) {
   t= ordo.get_task();
   if( t )
      t.execute();
   else
      ordo.wait();
}
```

Figure 7.2 *Boucle exécutée par tout processeur virtuel.*
Chaque processeur virtuel demande une tâche à exécuter, l'exécute, puis retourne au point de départ. Si aucune tâche n'est disponible le processeur se bloque en attente passive et sera réveillé lorsque le module d'ordonnancement aura de nouveau du travail à donner ou, au plus tard, à la fin de l'application.

L'existence de ce module d'exécution est due aux enseignements tirés d'un premier prototype de la bibliothèque Athapascan-1a [43, 27]. Dans ce prototype, dès qu'une tâche était prête et sur son site d'exécution, un fils Athapascan-0 était créé pour son exécution. Ce fonctionnement, simple et intuitif, impliquait cependant la création d'un grand nombre de fils d'exécution dont le coût était important. La gestion d'un nombre trop important de fils par Athapascan-0 était également problématique.

7.2.3.2 Interface avec la bibliothèque Athapascan-1

Il n'y a pas, à proprement parler, d'interaction directe entre ce module d'exécution et la bibliothèque. Cependant, les tâches ou les objets en mémoire partagée sont créés lorsque certaines instructions (`Fork`, `Shared`) du corps des tâches sont exécutées par les processeurs virtuels. C'est uniquement pour traduire ces créations d'objets qu'un lien a été représenté dans la figure 7.1 page 131.

7.2.3.3 Interface avec le module d'ordonnancement

L'interaction entre le module d'exécution et le module d'ordonnancement est de type *receiver initiated* : c'est lorsqu'un des processeurs virtuels devient inactif que le module

d'ordonnancement est contacté et doit fournir du travail, et non le module d'ordonnancement qui affecte les tâches dès qu'elles sont prêtes aux processeurs virtuels.

Dans la boucle représentée figure 7.2 page 133 l'interaction avec le module d'ordonnancement se fait à travers les deux fonctions `ordo.get_task()` et `ordo.wait()`. La première demande une nouvelle tâche à exécuter lorsque le processeur vient de finir l'exécution d'une tâche et la seconde est une synchronisation qui intervient lorsqu'aucune tâche n'est prête à être exécutée. Le fil d'exécution sera réveillé soit par le module d'ordonnancement si une tâche vient à être disponible, soit par le système si l'application est terminée.

7.2.4 Module d'ordonnancement

Le module d'ordonnancement est constitué d'un certain nombre de groupes, chacun d'entre eux implantant une politique d'ordonnancement. Un groupe représente donc cette politique particulière au sein du module d'ordonnancement. Chaque groupe est autonome et seul responsable de l'ordonnancement des tâches qu'il contient. Les interactions avec les autres modules, représentés figure 7.3 page 135, sont décrites dans la suite de cette section.

7.2.4.1 Fonctionnement

Les tâches sont fournies aux groupes par la bibliothèque Athapascan-1 dès leur création. Par défaut toute tâche fille est donnée au même groupe que sa mère. Conformément à la politique qui lui est associée, chaque groupe décide du nœud sur lequel chaque tâche doit être exécutée et, au niveau de chaque nœud, quelle tâche doit être donnée au module d'exécution lorsqu'un processeur virtuel demande du travail. Le module d'ordonnancement se comporte donc comme un filtre entre le module de génération du graphe constitué par la bibliothèque Athapascan-1 et le module d'exécution.

7.2.4.2 Interface avec la bibliothèque Athapascan-1

Les interactions possibles entre un groupe d'ordonnancement et la bibliothèque sont les suivantes :

- Lorsqu'une tâche est créée au niveau de la bibliothèque, elle est confiée au groupe responsable de son ordonnancement (`new_task`). Le choix du groupe est effectué par l'utilisateur, comme illustré figure 7.4 page 135. Par défaut, la tâche créée est placée dans le même groupe que la tâche créatrice.
- Chaque fois que l'état d'une tâche évolue, le groupe auquel elle appartient est averti (`new_state`). Cette fonction est appelée en particulier lors de la terminaison de la tâche. Voir le tableau 3.1 page 58 pour une définition des différents états et la figure 3.3(a) page 59 pour leurs possibles évolutions.
- Le groupe peut explorer la portion locale du graphe des tâches. Cette exploration se fait à partir d'une des tâches dont il est possible d'obtenir les données partagées

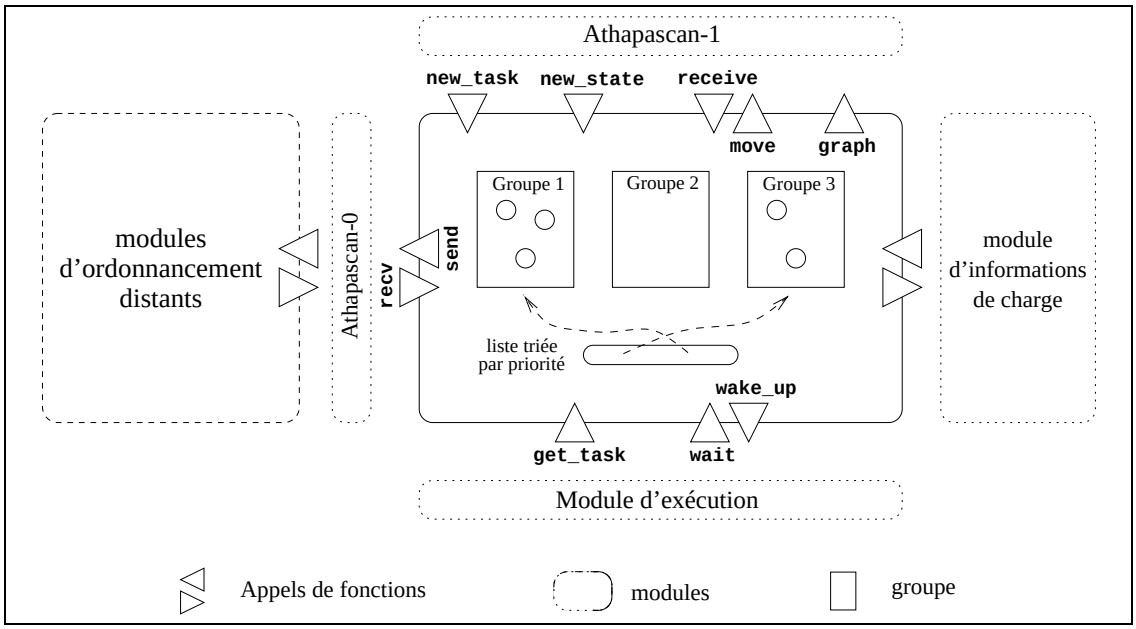


Figure 7.3 *Interfaces du module d'ordonnancement.*
Le module d'ordonnancement est en interaction locale avec la bibliothèque Athapascan-1, le module d'exécution et le module d'informations de charge. Il est également en relation avec les autres réplicats du module d'ordonnancement grâce aux primitives de communications offertes par Athapascan-0.

```
{
   ...
   policy::group grp;
   a1_set_default_group( grp );
   Fork< t >( <parametres> );
   ...
}
```

Figure 7.4 *Changement du groupe d'ordonnancement des tâches créées.*
Toutes les tâches créées après le changement du groupe par défaut, donc en particulier t, seront placées dans le groupe associé à la politique d'ordonnancement $policy$. L'utilisateur peut changer autant de fois le groupe par défaut de la tâche créatrice qu'il le désire.

accédées. Les contraintes de localité peuvent donc être analysées par la politique d'ordonnancement. À partir de ces données, il est également possible d'obtenir la liste des tâches locales les accédant ainsi qu'une liste des sites sur lesquels d'autres tâches les accèdent également (le graphe étant distribué par rapport aux données, section 4.2). Une série de fonctions est associée à toutes ces actions possibles et est représentée sous la dénomination générique `graph` dans la figure 7.3 page 135.

– Si le groupe décide de déplacer une tâche vers un autre site, il soumet une requête de déplacement (`move`) à la bibliothèque. La tâche sera déplacée, ainsi que ses données, vers le site destinataire. Lors de la réception sur le site distant, le réplicat du groupe sera averti de l'arrivée de cette tâche (`receive`). Le groupe ne peut gérer tout seul le déplacement des tâches car dans ce cas la cohérence du graphe doit être maintenue.

7.2.4.3 Interface avec le module d'exécution

Lorsque le module d'exécution demande une tâche à exécuter (`get_task`), le module d'ordonnancement transmet cette requête au groupe de plus forte priorité. Si la liste des groupes possédant au moins une tâche exécutable est vide, le module retourne une tâche vide (ce qui aura pour effet de placer le processeur virtuel dans un état d'attente, comme spécifié figure 7.2 page 133). Lorsqu'un des groupes qui ne possédait pas de tâche exécutable en possède soudain une (suite à la création ou la réception d'une tâche, l'évolution d'un état), ce groupe est inséré dans la liste en respectant sa priorité. Si un processeur virtuel s'était stoppé dans le module d'ordonnancement (`wait`), alors ce processeur est réveillé (`wake_up`) ce qui lui permet de tenter à nouveau d'obtenir une tâche à exécuter.

7.2.4.4 Interface avec les autres réplicats

Un groupe peut communiquer avec son homologue situé dans un autre réplicat du module d'ordonnancement au moyen des primitives de communications offertes par Athapascan-0. Les interactions entre ces groupes dépendent donc de la politique d'ordonnancement implantée : pour un placement de type aléatoire où les tâches sont affectées aux nœuds arbitrairement au moment de leur création aucune interaction n'est nécessaire ; pour d'autres, comme les algorithmes gloutons basés sur une technique de vol de travail, des requêtes de vol doivent être émises.

Il est à noter que deux groupes d'ordonnancement différents n'ont aucune interaction directe entre eux. Le seul lien possible est à travers le module d'information de charge qui permet à un groupe d'avoir accès à des informations fournies par un autre groupe.

7.2.5 Module d'information de charge

Le module d'information maintient une série d'indices permettant de caractériser l'état de charge de la machine à un instant donné. Les données maintenues par ce module sont classiquement fournies par le module d'information en ce qui concerne les données « systèmes » telles que la charge de la machine, la durée d'exécution des tâches, ou par le module d'ordonnancement en ce qui concerne les données « applicatives » telles que le nombre de tâches restant à exécuter.

Ce module peut, à partir de ces données, synthétiser des informations qui seront utilisées par les politiques du module d'ordonnancement [46, 49] : par exemple, l'estimation de la durée d'exécution d'une tâche sur laquelle sont basés les algorithmes développés dans [36].

L'interaction de ce module avec les autres dépend de l'implantation de la politique d'ordonnancement qui définira les informations dont elle a besoin. Il n'est en effet pas raisonable de maintenir *a priori* un ensemble d'informations si elles ne sont pas utilisées vu le coût de leur mise à jour. De plus deux politiques d'ordonnancement ne requièrent que rarement des informations similaires.

Il est à noter que pour les stratégies d'ordonnancement que nous proposons ce module n'est pas utilisé.

7.3 Implantations de quatre algorithmes d'ordonnancement dans Athapascan-1

Nous présentons dans cette section l'implantation, au sein du module d'ordonnancement présenté dans le début de ce chapitre, de quatre stratégies d'ordonnancement :

- Un algorithme de placement arbitraire.
- Un algorithme glouton présenté section 6.3.1.1 page 118.
- L'algorithme $\mathcal{O}_1$ présenté section 6.4.1 page 124.
- L'algorithme $\mathcal{O}_2$ présenté section 6.4.2 page 126.

7.3.1 Algorithme de placement arbitraire

Chaque processeur maintient une liste locale de tâches prêtes gérée de manière LIFO. Chaque fois qu'une nouvelle tâche est créée, elle est arbitrairement envoyée à un processeur qui sera le site de son exécution. L'implantation de cet algorithme au sein du module d'ordonnancement est la suivante :

- `new_task` : la tâche est envoyée arbitrairement à un processeur de la machine.
- `new_state` : si une tâche passe à l'état prêt, elle est insérée en tête de la liste locale et les processeurs virtuels bloqués dans le module d'ordonnancement sont réveillés. Toutes les autres modifications d'état sont ignorées.
- `receive` : si la tâche reçue est prête à être exécutée alors elle est insérée en tête de la liste locale. Sinon, elle est ignorée (on attend qu'elle soit prête pour la prendre en considération).
- `get_task` : la tâche située en tête de la liste est retournée. Si la liste était vide, une tâche vide est retournée (ce qui aura pour effet de bloquer le processeur virtuel).
- De plus, chaque fois qu'une tâche est ajoutée dans une liste vide (`new_state` ou `receive`) tous les processeurs virtuels qui se sont bloqués dans le module d'ordonnancement (`wait`) sont réveillés (`wake_up`).

7.3.2 Algorithme glouton

La stratégie implantée est celle présentée section 6.3.1.1 page 118 et qui consiste à maintenir les processeurs au maximum en activité. Le principe de base est de maintenir une liste de tâches prêtes, liste dans laquelle les processeurs virtuels iront chercher du travail lorsqu'ils seront inactifs. Cette liste est gérée de manière distribuée : chaque nœud maintient une liste locale dans laquelle il insère les tâches prêtes qui ont été créées sur le nœud. Lorsqu'un processeur virtuel demande une tâche à exécuter, cette tâche est prise dans la liste locale. Si cette liste locale est vide, une requête de vol est émise vers un nœud choisi arbitrairement. Ce nœud émettra alors une tâche prête ou retransmettra la requête vers un autre nœud si sa liste est également vide. L'implantation de cet algorithme au sein du module d'ordonnancement est la suivante :

- `new_task`, `receive` : si la tâche est prête à être exécutée alors elle est insérée en tête de la liste locale. Sinon, elle est ignorée.
- `new_state` : si une tâche passe à l'état prêt, elle est insérée en tête de la liste locale et les processeurs virtuels bloqués dans le module d'ordonnancement sont réveillés. Toutes les autres modifications d'état sont ignorées.
- `get_task` : la tâche située en tête de la liste est retournée (la liste est de type LIFO). Si la liste était vide, une tâche vide est retournée et une requête de vol est émise (`send`) vers un autre nœud choisi au hasard.
- `recv` : Lors de la réception d'une requête de vol, une tâche prise arbitrairement dans la liste des tâches prêtes est envoyée (`move`). Si la liste était vide, la requête est retransmise (`send`) vers un autre nœud.
- De plus, chaque fois qu'une tâche est ajoutée dans une liste vide tous les processeurs virtuels qui se sont bloqués dans le module d'ordonnancement (`wait`) sont réveillés (`wake_up`).

7.3.3 Algorithme $\mathcal{O}_1$

L'algorithme $\mathcal{O}_1$, présenté section 6.4.1 page 124, consiste à suivre au plus près l'ordre de « référence » défini sur l'ensemble des tâches. L'implantation de cet ordre est effectué de manière centralisée par le groupe qui ne prend en compte les nouvelles tâches créées qu'à la terminaison de la tâche créatrice : chaque tâche maintient donc la liste de ses tâches filles. L'implantation de cet algorithme au sein du module d'ordonnancement est la suivante :

- `new_task` : la tâche est insérée en queue de la liste des filles de la tâche créatrice.
- `new_state` : si une tâche passe à l'état terminé sur le processeur maître, la liste de ses filles vient remplacer dans la liste locale la tâche créatrice : la liste reste donc triée. Si une tâche passe à l'état terminé sur un autre processeur, la liste des filles est émise (`send`) vers le processeur maître. Si une de ces tâches est prête, les processeurs virtuels éventuellement bloqués sur le processeur maître sont réveillés. Si des processeurs avaient émis des requêtes de demande de travail qui n'avaient pas été honorées, le processeur maître tente alors de les satisfaire. De même si une tâche déjà insérée dans la liste locale sur le processeur maître passe à l'état prêt.
- `receive` : Seuls les processeurs autres que le maître peuvent recevoir des tâches. La tâche est conservée pour exécution. Cette tâche reçue est la conséquence d'une requête faite au processeur maître.
- `get_task` : sur le processeur maître, la première tâche prête de la liste locale est retournée. Sur les autres processeurs, si une tâche a été reçue (`receive`) elle est retournée ; sinon une requête de demande de travail est émise (`send`) vers le processeur maître.
- `recv` : seul le processeur maître reçoit des messages venant des autres réplicats du module. Si le message représente la fin d'une tâche, les tâches filles sont insérées

dans la liste locale. Si c'est une demande de travail, la première tâche prête de la liste est émise.

7.3.4 Algorithme $\mathcal{O}_2$

L'algorithme $\mathcal{O}_2$, présenté section 6.4.2 page 126, constitue un intermédiaire entre l'algorithme $\mathcal{O}_1$ et la stratégie gloutonne basée sur une technique de vol de travail. Cet algorithme peut être vu comme l'utilisation locale sur chaque nœud de la stratégie $\mathcal{O}_1$, les nœuds étant reliés conformément à la stratégie gloutonne par des requêtes de vol de travail. Il est à noter qu'il y a toujours un des processeurs qui suit l'ordre de « référence » $\prec_r$: à tout instant t, en notant t_i la tâche en cours d'exécution sur p_i à l'instant t, toutes les tâches inférieures à $\min_{\prec_r}(t_i)$ ont été exécutées.

Chaque processeur maintient une liste locale triée selon l'ordre de « référence » dans laquelle il puise en tête les tâches à exécuter. Lorsque qu'il rencontre une tâche non prête il abandonne cette liste et part voler la fin d'une autre liste. Lorsque la dernière tâche d'une liste est terminée, le processeur tente de poursuivre son exécution avec ce qui lui reste de la portion de liste qui lui avait été volée (et qui constitue donc les tâches suivant cette dernière tâche exécutée selon l'ordre de « référence »). L'implantation de cet algorithme au sein du module d'ordonnancement est la suivante :

- `new_task` : la tâche est insérée en queue de la liste des filles de la tâche créatrice.
- `new_state` : si une tâche passe à l'état terminé la liste de ses filles vient remplacer dans la liste locale la tâche créatrice : la liste locale reste donc triée.
- `receive` : il y a réception de toute une portion de liste de tâches suite à une requête de vol : ces tâches constituent la nouvelle liste locale.
- `get_task` : la première tâche de la liste locale est retournée. Si cette tâche n'est pas prête, il y a émission d'une requête de vol vers un processeur choisit arbitrairement. Si la liste est vide, le processeur émet une requête de vol en essayant d'abord de récupérer la fin (qui lui avait été nécessairement volée) de cette liste afin de continuer l'exécution (c'est ce qui permet de majorer le volume de mémoire nécessaire à l'exécution).
- `recv` : la réception d'une requête de récupération de liste volée est effectuée en retournant ce qui reste de la liste qui avait été volée, sauf si cette liste est celle en cours d'utilisation. Une requête de vol est traitée en retournant une portion de liste : une tâche prête est choisie arbitrairement parmi toutes les listes possédées par le site, puis toutes les tâches suivant cette tâche prête sont envoyées au processeur inactif.

7.4 Évaluations

Nous étudions dans cette section l'évaluation des quatre stratégies d'ordonnancement présentées dans la section précédente. Les performances de ces stratégies sont comparées en fonction du volume de mémoire nécessaire à l'exécution sur une machine parallèle.

Cette section est organisée comme suit : nous présentons tout d'abord l'application qui nous sert à comparer les quatre stratégies, puis nous présentons les paramètres des évaluations, puis la collecte et la synthèse des résultats et enfin nous étudions, pour chaque stratégie, le volume mémoire consommé lors d'une exécution particulière.

7.4.1 Expérimentation

L'application considérée pour comparer les différentes stratégies d'ordonnancement implantées consiste à calculer une portion de l'image de l'ensemble de Mandelbrot. Cette application a été choisie d'une part pour la simplicité du parallélisme généré (de type diviser pour paralléliser) et d'autre part pour son côté « visuel ».

Cette application, bien qu'extrêmement simple et d'intérêt restreint, nous a cependant permis d'évaluer rapidement et très simplement le comportement vis-à-vis de l'ordre d'exécution des tâches de toutes les politiques d'ordonnancement que nous implantions.

7.4.1.1 Algorithme

L'algorithme consiste en une découpe récursive de la zone à calculer. Une fois un certain seuil atteint, une tâche de visualisation, destinée à afficher la portion de l'image calculée, sera créée par zone ; le calcul des valeurs de chaque zone sera effectué également par une découpe du calcul récursive, chaque portion de zone calculée étant accumulée (après détermination de la couleur à associer à chaque pixel) dans la donnée qui sera lue par la tâche d'affichage. Le graphe de tâches correspondant à cette application est illustré figure 7.5 page 140.

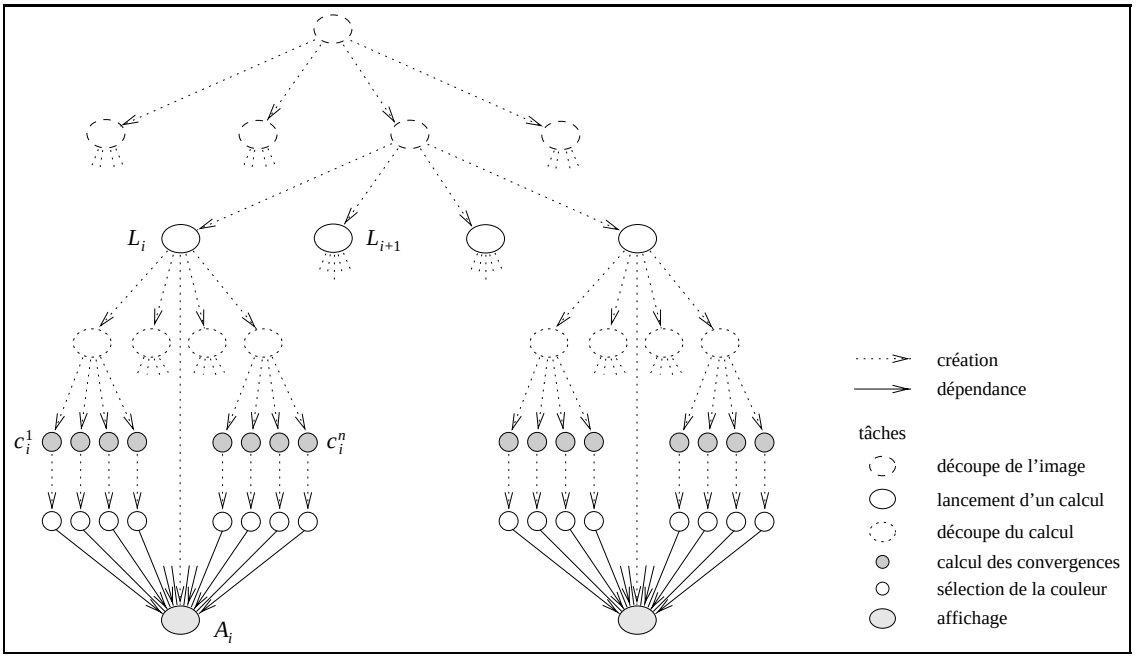


Figure 7.5 *Graphe de tâches du calcul et de la visualisation de l'ensemble de Mandelbrot.*

Nous notons dans la suite $L_1, \ldots, L_N$ les tâches de lancement de calcul, $A_1, \ldots, A_N$ les tâches d'affichages et $c_i^1, \ldots, c_i^n$ les tâches de calcul qui seront associées à la tâche A_i. Les contraintes de précédence entre ces tâches et l'ordre de « référence » sont les suivants :

$$\begin{aligned} \forall i \in [1..N], \forall j \in [1..n] \quad &: \quad L_i \prec c_i^j \prec A_i \\ \forall i \in [1..N] \quad &: \quad L_i \prec_r c_i^1 \prec_r \ldots \prec_r c_i^n \prec_r A_i \prec_r L_{i+1} \end{aligned}$$

L'espace mémoire S_1 est la somme des espaces mémoire alloués par tout ensemble $\{c_i^1, \ldots, c_i^n\}$ de tâches de calcul, cet espace mémoire étant désalloué par la tâche d'affichage A_i. Puisqu'il y a au plus N groupes de tâches de calculs :

$$S_1 \leq S_p \leq N S_1$$

7.4.1.2 Conditions d'évaluations

Toutes les évaluations effectuées dans cette section ont été effectuées sur un quadri-processeurs de type SMP (ceci permet de communiquer via la mémoire par appels de fonction, ce qui facilite considérablement l'implantation des stratégies). Les caractéristiques précises des exécutions sont rassemblées dans le tableau 7.1 page 141. La zone calculée est $(-0.07, 0.74) \times (-0.03, 0.68)$: ce choix est essentiellement le fruit du hasard, nous recherchions juste une portion de l'ensemble qui présentait certaines zones de non convergence (représentées en noir) afin d'introduire une certaine irrégularité dans la durée des tâches de calcul.

Caractéristiques des exécutions	
taille de l'image (pixels)	500×500
zone calculée	$(-0.07, 0.74) \times (-0.03, 0.68)$
nombre d'itérations maximum	100
nombre total de tâches	2405
dont tâches de découpe de l'image	$1 + 4 = 5$
dont tâches d'affichage	$4 * 4 = 16$
dont tâches de découpe de calcul	$16 * (4 * (1 + 4)) = 320$
dont tâches de calcul	$16 * 16 * 4 = 1024$
mémoire requise pour une zone	$\approx (500 * 500/1024) * 4$ o ≈ 976 o
T_1	≈ 29 s
T_∞	≈ 0.04 s
S_1	≈ 78 ko
p	de 1 à 4 IBM SP 604e 332 MHz

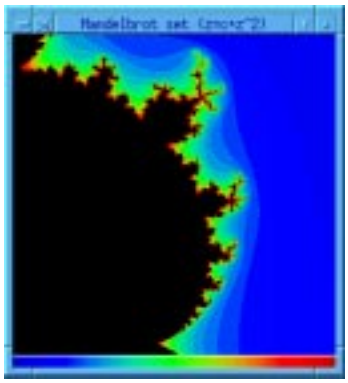


Tableau 7.1 *Conditions d'évaluation des différentes stratégies d'ordonnancement.*
Le tableau à gauche contient les caractéristiques de l'application utilisée pour comparer les différentes stratégies. Pour indication, l'image à droite représente le résultat d'une exécution.

Les courbes présentées dans les figures 7.8 page 145, 7.9 page 146, 7.10 page 147, 7.11 page 148 et 7.13 page 149 ont toutes la même légende : l'ordonnée représente le volume mémoire réservé[4] par l'application (somme des volumes alloués non encore désalloués) après chaque nouvelle action (allocation ou libération) sur cette mémoire. Ces actions sont numérotées et représentées en abscisse.

[4]Ces données ont été obtenues en définissant une version tracée des opérateurs C++ `new`, `new[]`, `delete` et `delete[]` et représentent donc toutes les allocations mémoire effectuées par Athapascan-1 et

7.4.1.3 Résultats

Synthèse Les résultats présentés dans le tableau 7.2 page 142 représentent la durée, le volume mémoire nécessaire et la nombre de messages effectués pour chacune des quatre politiques d'ordonnancement arbitraire, gloutonne, $\mathcal{O}_1$ et $\mathcal{O}_2$. Les mesures de temps ont été effectuées en désactivant le mécanisme de traçage des opérations mémoire. Le nombre de messages représente le nombre d'appels de fonction d'un processeur vers un autre (les implantations et les exécutions ont été effectuées sur un *SMP* à mémoire partagée). Les durées sont des moyennes sur 20 exécutions dont on a retiré les 2 extrêmes (les mesures sont stables). Les courbes de consommation mémoire présentées, quant à elles, ne concernent qu'une exécution particulière.

p	S_p (ko)	$\frac{S_p}{S_1}$	T_p (s)	$\frac{T_1}{T_p}$	m	S_p (ko)	$\frac{S_p}{S_1}$	T_p (s)	$\frac{T_1}{T_p}$	m
	arbitraire					**glouton**				
1	77.6	1	29.2	1	–	77.5	1	29.2	1	–
2	548.7	7.07	14.8	1.97	1200	146.0	1.88	14.6	1.99	80
3	871.7	11.23	10.0	2.91	1600	209.2	2.70	10.1	2.90	180
4	808.9	10.42	9.5	3.08	1800	284.8	3.68	9.5	3.07	240
	$\mathcal{O}_1$					$\mathcal{O}_2$				
1	78.0	1	29.2	1	–	78.2	1	29.3	1	–
2	84.5	1.08	14.9	1.96	2400	143.2	1.83	14.7	1.99	80
3	100.1	1.28	10.1	2.90	3200	189.7	2.43	10.0	2.91	180
4	105.3	1.35	9.7	3.02	3700	238.7	3.05	9.2	3.17	240

Tableau 7.2 *Évaluation numérique de différentes stratégies d'ordonnancement pour le calcul de Mandelbrot.*
Cette table est une synthèse chiffrée des différentes courbes présentées. On prend pour valeur de S_1 le volume mémoire nécessaire à l'exécution sur un processeur. Le nombre de messages engendrés m est à considérer comme un ordre de grandeur, ce nombre dépendant de l'exécution ; c'est un moyenne sur une vingtaine exécutions. $p = q$ représente le nombre de processeurs physiques (qui coïncide ici avec le nombre de processeurs virtuels d'Athapascan-1) de la machine.

Nous pouvons noter que l'accélération est proche de p pour $p \leq 3$, mais pas pour $p = 4$. Cela est dû à la présence d'un démon de communication activé par Athapascan-0 et qui vole du temps de calcul à l'application. Cela revient à dire, lorsque nous n'activons que 3 processeurs virtuels pour le module d'exécution d'Athapascan-1 ($p = 3$) qu'en fait 4 processeurs physiques peuvent être exploités du fait de la présence de ce démon. Lorsque $p = 4$ le système doit ordonnancer 5 processus légers sur les 4 processeurs de la machine, d'où la perte d'efficacité. Ce problème est largement étudié dans [20].

Nous remarquons également que les stratégies gloutonne et $\mathcal{O}_2$ semblent être légèrement plus rapide (au moins sur 2 processeurs). Ce phénomène est masqué par le fait que l'implantation a été effectuée sur une machine de type *SMP*, mais il est certainement accentué aur une machine à mémoire distribuée où la durée des messages est bien plus importante : les stratégies arbitraire et $\mathcal{O}_1$ seront donc largement désavantagées.

l'application (l'allocateur propre d'Athapascan-1 était désactivé). Les allocations effectuées par Athapascan-0 (utilisant `malloc`) ne sont donc pas comptées (mais sont constantes). De même la STL (*Standard Template Library*) n'a pas été configurée pour utiliser les opérateurs C++ `new` et `delete`. Les structures de gestion des stratégies d'ordonnancement ne sont donc pas prises en compte. Mais ce volume mémoire est fonction du nombre de tâches (la consommation mémoire de ces objets est comptée) donc ne modifie pas l'allure des courbes présentées.

Forme des courbes Les courbes présentées dans les figures 7.8 page 145, 7.9 page 146, 7.10 page 147, 7.11 page 148 et 7.13 page 149 présentent toutes un phénomème de « dents ». Ce phénomène s'explique de la manière suivante (nous considérons pour simplifier l'exécution sur un processeur d'un ordonnancement arbitraire, figure 7.8 page 145 par exemple).

Remarquons tout d'abord qu'il y a 16 dents : chacune de ces dents correspond à une tâche d'affichage A. Cette tâche d'affichage a été créée par une tâche de lancement de calcul L qui a créé 4 tâches de découpe de calcul (disons d_1, d_2, d_3, d_4). Chacune de ces tâches va donner naissance à 4 nouvelles tâches de découpe qui vont créer chacune 4 tâches de calcul de convergence (disons c_1, c_2, c_3, c_4) et 4 tâches de sélection de la couleur (disons s_1, s_2, s_3, s_4). Les contraintes de précédence impliquent que la tâche A sera exécutée après toutes ces tâches de découpe, de calcul de convergence et de sélection de couleur. En général, toutes ces tâches sont exécutées en respectant l'ordre séquentiel de création ce qui explique la régularité dans les formes des courbes. Les figures 7.6 page 143 et 7.7 page 144 illustrent ce phénomène.

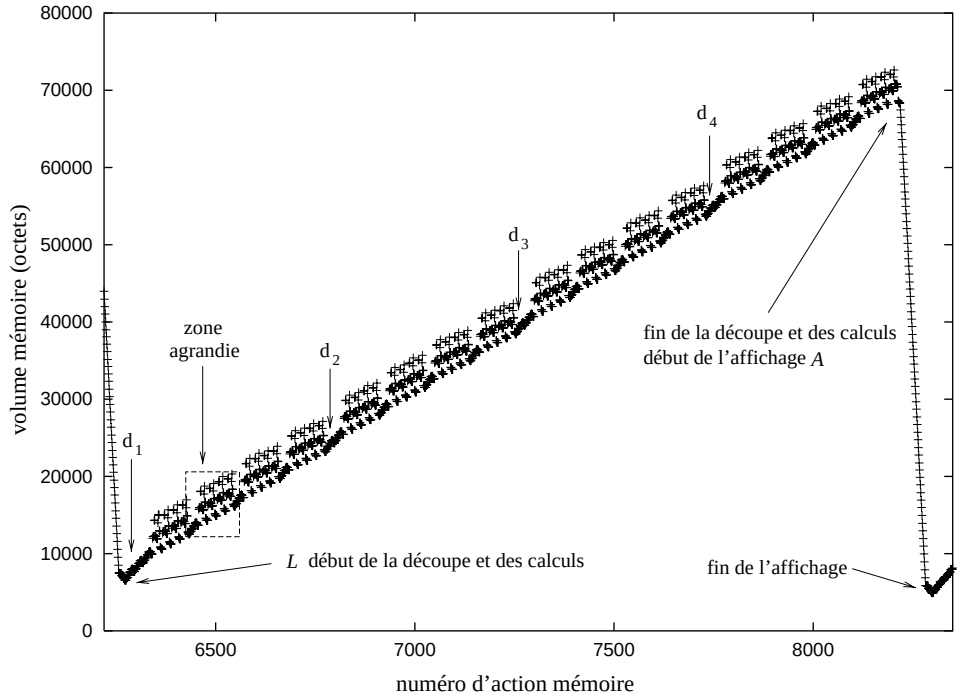


Figure 7.6 *Une « dent » de consommation mémoire.*
Cette dent correspond à l'évolution du volume mémoire nécessaire à l'application entre le début d'exécution d'une tâche de lancement de calcul et la tâche d'affichage A correspondant à ce lancement. On peut voir les 4 tâches principales de découpe et, pour chacune des tâches finales (créées par les principales), on peut distinguer les 4 tâches de calcul de convergence. La figure 7.7 page 144 est un agrandissement d'une de ces tâches finales.

La différence de pente entre les deux pans de la « dent » est due au fait que la mémoire est beaucoup plus manipulée lors de la phase de calcul que lors de la phase d'affichage : pour l'affichage, toutes les zones de calculs sont libérées les unes après les autres, tandis

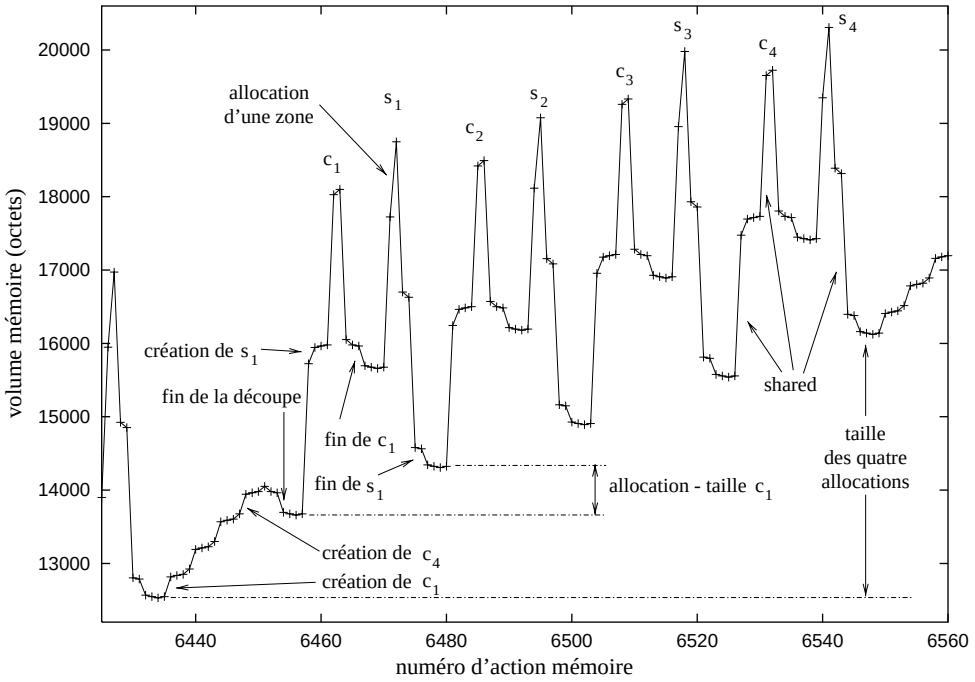


Figure 7.7 *Agrandissement de la figure 7.6 page 143 sur une tâche finale de découpe de calcul. On peut apercevoir les 4 tâches c_i de calcul de convergence et les 4 tâches s_i de sélection de couleur qui leurs sont associées.*

que dans la phase de calcul les zones sont allouées, des tâches sont créées, des tâches se terminent, des données partagées sont instanciées, *etc*., ce qui ralentit la vitesse d'évolution de la mémoire (vitesse par rapport au numéro d'action sur la mémoire).

7.4.2 Algorithme de placement arbitraire

La figure 7.8 page 145 illustre le volume mémoire nécessaire à l'exécution de l'application considérée section 7.4.1 page 140. La durée d'exécution et le nombre de messages générés sont transcrits dans le tableau 7.2 page 142. On peut remarquer que l'utilisation de plusieurs processeurs accroît dramatiquement le volume mémoire nécessaire. Cela est dû aux exécutions tardives des tâches d'affichage qui ont pour effet de libérer la zone allouée.

Les piètres performances concernant le volume mémoire nécessaire à l'exécution s'expliquent par le fait que l'exécution des tâches d'affichage, celles qui libèrent la mémoire, peuvent être exécutées très tardivement. Par exemple, le placement suivant peut apparaître sur deux processeurs p_0 et p_1 : supposons que p_0 possède toutes les tâches de découpe de l'image, c'est donc lui qui décide du placement des tâches de lancement L_i. Il peut décider de les placer toutes sur p_1, mais à un rythme tel qu'entre deux tâches L_i reçues p_1 ait le temps d'exécuter toutes les tâches c_n^i mais pas de commencer A_i. Alors L_{i+1} sera mise en tête de la liste des tâches prêtes et A_i ne pourra être exécutée qu'après l'exécution de toutes les tâches engendrées par L_{i+1}. L'ordre d'exécution des tâches sur

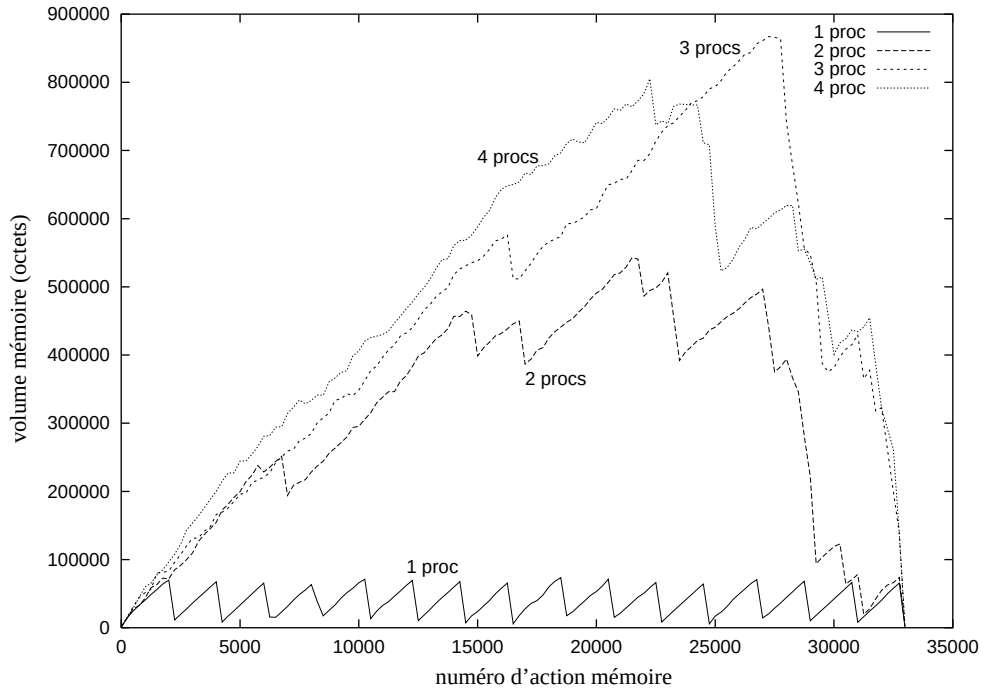


Figure 7.8 *Volume mémoire nécessaire à l'exécution selon un placement arbitraire des tâches. Les conditions d'expérimentation sont celles présentées à la section 7.4.1.2 page 141.*

p_1 est alors :

$$L_1 < c_1^1 < \ldots < c_1^n < L_2 < \ldots < L_N < c_N^1 < \ldots < c_N^n < A_N < A_{N-1} < \ldots < A_1$$

Cet ordre d'exécution mène à une consommation mémoire $S_p = NS_1$ qui est le maximum que cette application puisse consommer.

Le nombre de messages engendrés est également important : un message par tâche avec une probabilité de $1 - \frac{1}{p}$. De plus, aucune notion de localité entre les tâches est respectée, comme en témoigne la figure 7.14 page 150.

7.4.3 Algorithme glouton

La figure 7.9 page 146 illustre le volume mémoire nécessaire à l'exécution de l'application présentée section 7.4.1 page 140. La durée d'exécution et le nombre de messages générés sont transcrits dans le tableau 7.2 page 142. On peut remarquer que l'utilisation de plusieurs processeurs accroît le volume mémoire nécessaire d'une manière semble-t-il linéaire. Le nombre de messages engendrés est faible, ce qui traduit une bonne propriété de localité de l'algorithme.

Cependant, cette stratégie peut mener à un ordonnancement des tâches tel que la consommation mémoire soit maximale, c'est-à-dire $Sp = NS_1$. En effet, soit une exécution sur 3 processeurs, p_0, p_1 et p_2 telle qu'au moins une phase de découpage du calcul intervienne entre les tâches L_i et les tâches c_i^j. Supposons que p_0 possède toutes les tâches L_i dans sa liste de tâches prêtes. Il exécute en premier L_N (la liste est gérée de manière

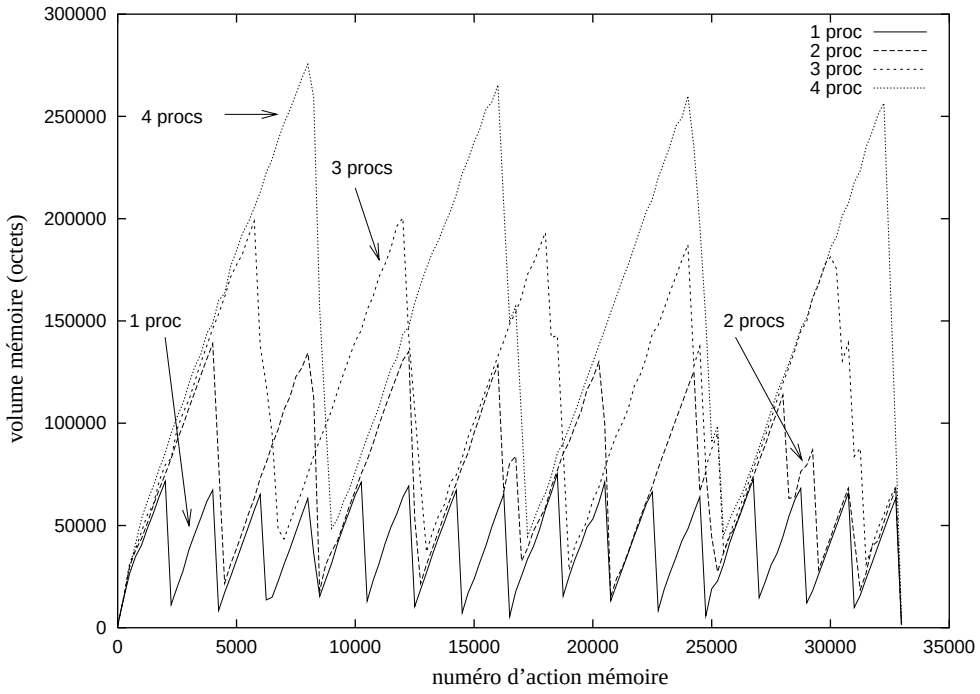


Figure 7.9 *Volume mémoire nécessaire à l'exécution selon un algorithme glouton. Les conditions d'expérimentation sont celles présentées à la section 7.4.1.2 page 141.*

LIFO). p_1 vole une tâche de découpe de calcul à p_0 et p_2 vole une tâche de calcul, disons c_1^1 à p_1. Compte tenu des contraintes de précédence entre les tâches, la tâche A_N ne pourra s'exécuter que lorsque l'exécution de c_1^1 aura été terminée et que p_0 en aura été averti. Or il se peut que p_2 ait plus de mal à contacter p_0 qu'à voler des tâches à p_1 (les liens de communications ne sont pas forcément les mêmes). Ce scénario pouvant être répété pour toute tâche L_i, il vient que p_2 va bloquer l'exécution sur p_0 de toutes les tâches A_i en possédant toutes les tâches c_i^1. Le volume mémoire nécessaire à cette exécution est donc égal à NS_1.

7.4.4 Algorithme $\mathcal{O}_1$

La figure 7.10 page 147 illustre le volume mémoire nécessaire à l'exécution de l'application présentée section 7.4.1 page 140. La durée d'exécution et le nombre de messages générés sont transcrits dans le tableau 7.2 page 142. On peut remarquer que l'utilisation de plusieurs processeurs n'accroît que très peu le volume mémoire nécessaire. Cela est dû au fait que l'ordre d'exécution des tâches suit au plus près l'ordre de « référence ». L'implantation centralisée de cet ordre sur un processeur maître génère un nombre important de messages : deux messages (un pour l'exécution distante et un pour notifier la terminaison) par tâche avec une probabilité de $1 - \frac{1}{p}$.

Compte tenu du goulot d'étranglement que représente la centralisation du calcul de l'ordre de « référence » sur un seul processeur maître, cette stratégie ne peut être envisagée que sur un nombre relativement restreint de processeurs. Une autre solution est de

distribuer le calcul de l'ordre. C'est ce que fait la stratégie $\mathcal{O}_2$ suivante.

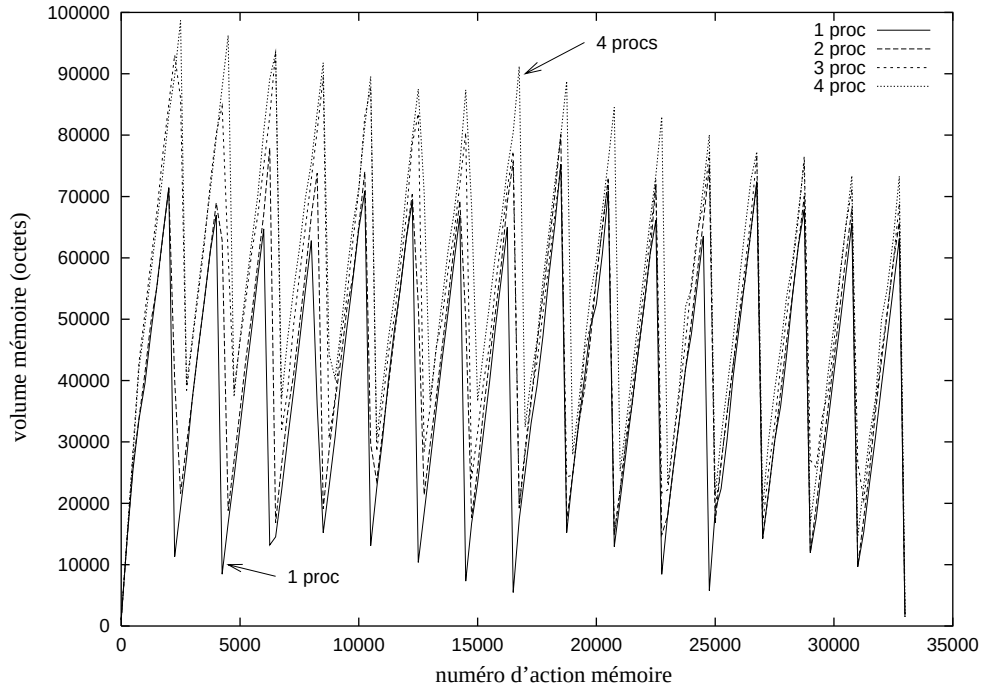


Figure 7.10 *Volume mémoire nécessaire à l'exécution selon l'algorithme $\mathcal{O}_1$.*
Les conditions d'expérimentation sont celles présentées à la section 7.4.1.2 page 141.

7.4.5 Algorithme $\mathcal{O}_2$

La figure 7.11 page 148 illustre le volume mémoire nécessaire à l'exécution de l'application présentée section 7.4.1 page 140. La durée d'exécution et le nombre de messages générés sont transcrits dans le tableau 7.2 page 142. On peut remarquer que l'utilisation de plusieurs processeurs accroît le volume mémoire nécessaire mais dans la limite de pS_1. Le nombre de messages engendrés est faible, ce qui traduit une bonne propriété de localité de l'algorithme.

7.4.6 Comparaison

La figure 7.12 page 148 trace la mémoire consommée par chacune des stratégies en fonction du nombre de processeurs. On peut remarquer que les stratégies $\mathcal{O}_1$ et $\mathcal{O}_2$ respectent les bornes théoriques préditent, à savoir $S_1 + ...$ et pS_1.

La figure 7.13 page 149 retrace, avec cette fois une même échelle pour les ordonnées, le volume mémoire utilisé lors de l'exécution sur 3 processeurs suivant la stratégie d'ordonnancement utilisée. Cette courbe permet de constater à quel point l'ordre d'exécution des tâches (calculé par la stratégie d'ordonnancement) influence le volume de mémoire nécessaire à l'exécution.

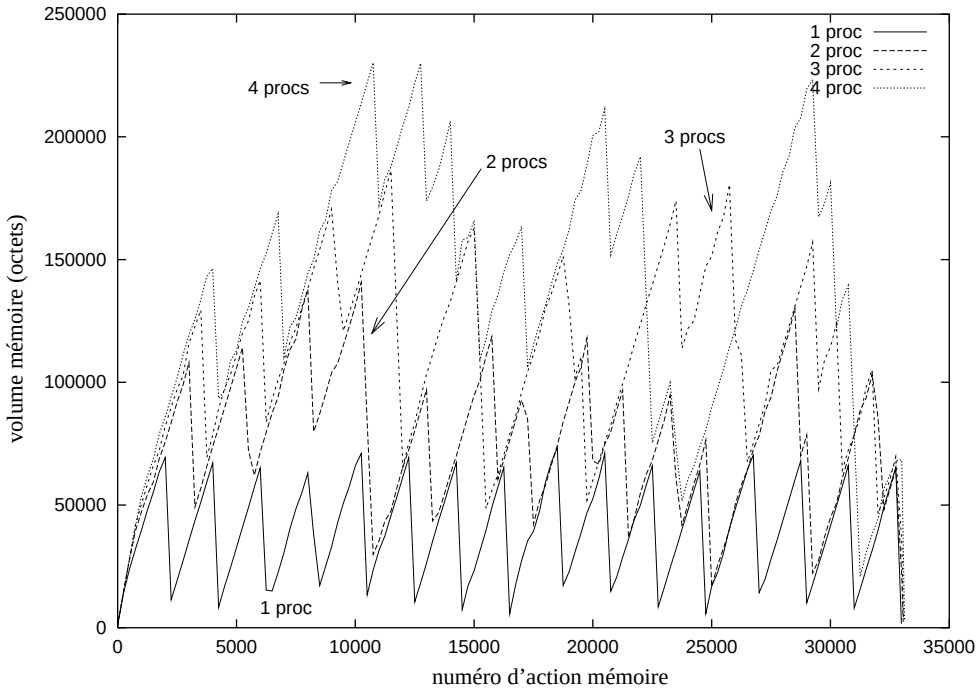


Figure 7.11 *Volume mémoire nécessaire à l'exécution selon l'algorithme $\mathcal{O}_2$.*
Les conditions d'expérimentation sont celles présentées à la section 7.4.1.2 page 141.

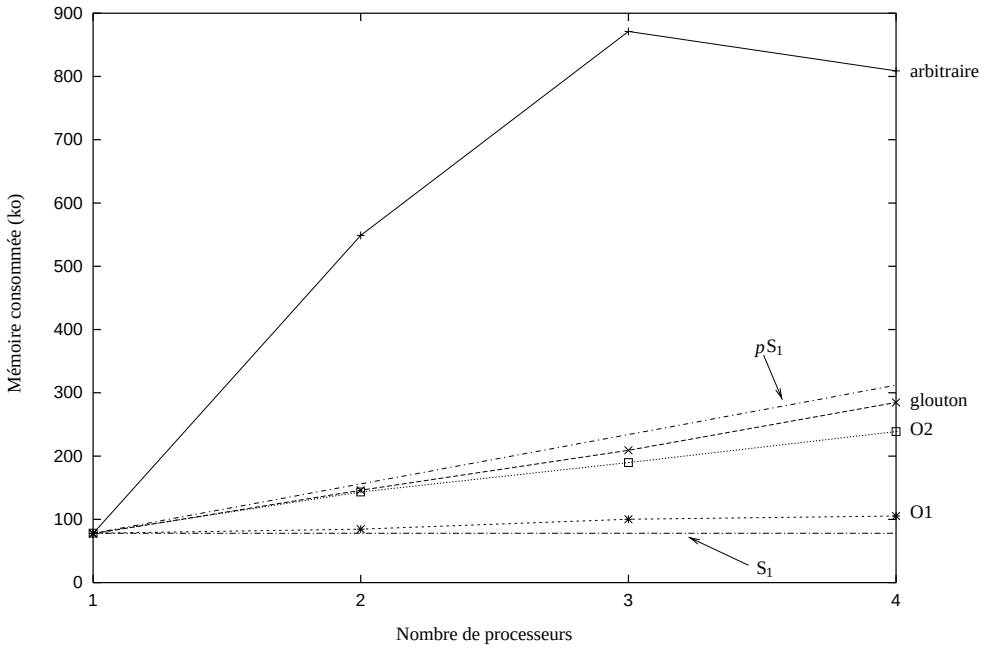


Figure 7.12 *Consommation mémoire en fonction du nombre de processeurs et de la stratégie d'ordonnancement utilisée.*
Les courbes S_1 et pS_1 sont particulières et utilisées dans les majorations théoriques.

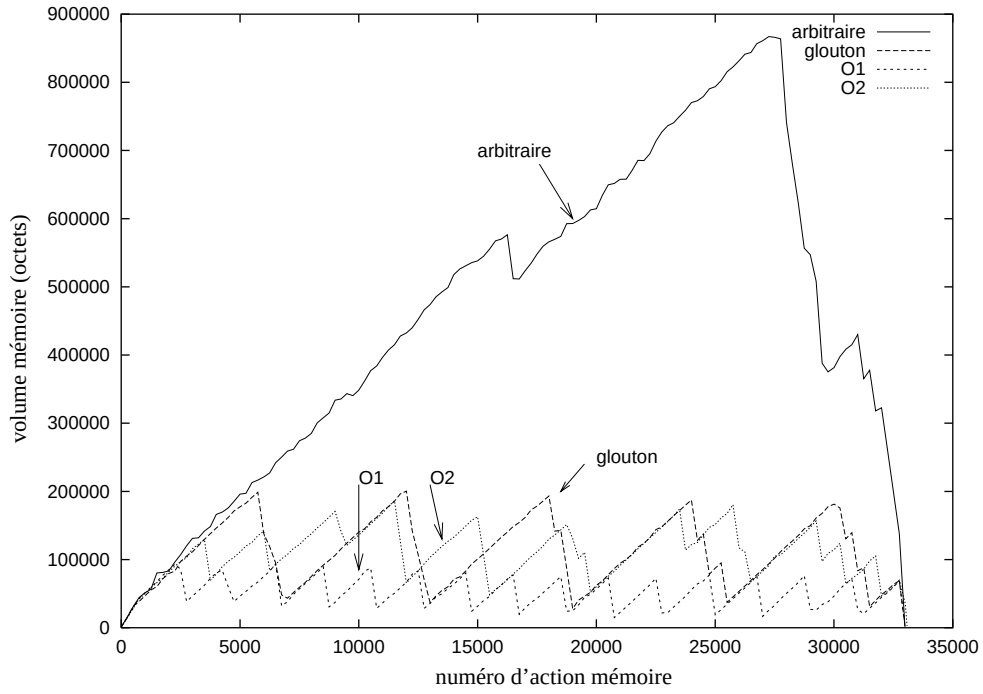


Figure 7.13 *Comparaison du volume mémoire nécessaire à l'exécution sur 3 processeurs selon la politique d'ordonnancement.*

La figure 7.14 page 150 illustre pour chacune des politiques d'ordonnancement, sur une exécution particulière effectuée sur 2 processeurs, l'ordre de calcul des différentes zones de l'image.

La première chose que l'on remarque est la répartition des couleurs : pour les stratégies arbitraires et $\mathcal{O}_1$ les zones de couleurs uniformes sont petites, tandis que pour les deux autres stratégies ces zones sont de plus grosse taille. Cela est dû à la propriété de localité de ces deux dernières : lorsqu'une tâche est exécutée sur un nœud, il y a de forte chance pour que toute sa fratrie soit également exécutée sur ce même nœud (à moins qu'il y ait eu vol). Cette propriété n'est pas vraie pour les stratégies arbitraires et $\mathcal{O}_1$. Ceci est directement corrélé au nombre de messages qui est plus important, tableau 7.2 page 142. En effet, une tâche correspond à un message dans le cas de ces stratégies tandis que pour les deux autres un message correspond à un gros bloc.

Une seconde chose à remarquer est la forme de la zone blanche, c'est-à-dire la zone non encore calculée : dans le cas arbitraire cette zone est relativement décousue, tandis que pour les trois autres cette zone reste compacte : la consommation mémoire est directement liée à ce critère.

Une dernière remarque concerne la différence entre la stratégie gloutonne et la stratégie $\mathcal{O}_2$. Ces deux stratégies se ressemblent fortement, mais la différence fondamentale est la suivante : dans le cas de la politique $\mathcal{O}_2$ il y a toujours un processeur virtuel qui exécute une tâche selon l'ordre de référence, ce qui n'est pas le cas pour la première. C'est cette différence qui permet de majorer le volume mémoire nécessaire à l'exécution dans le cas

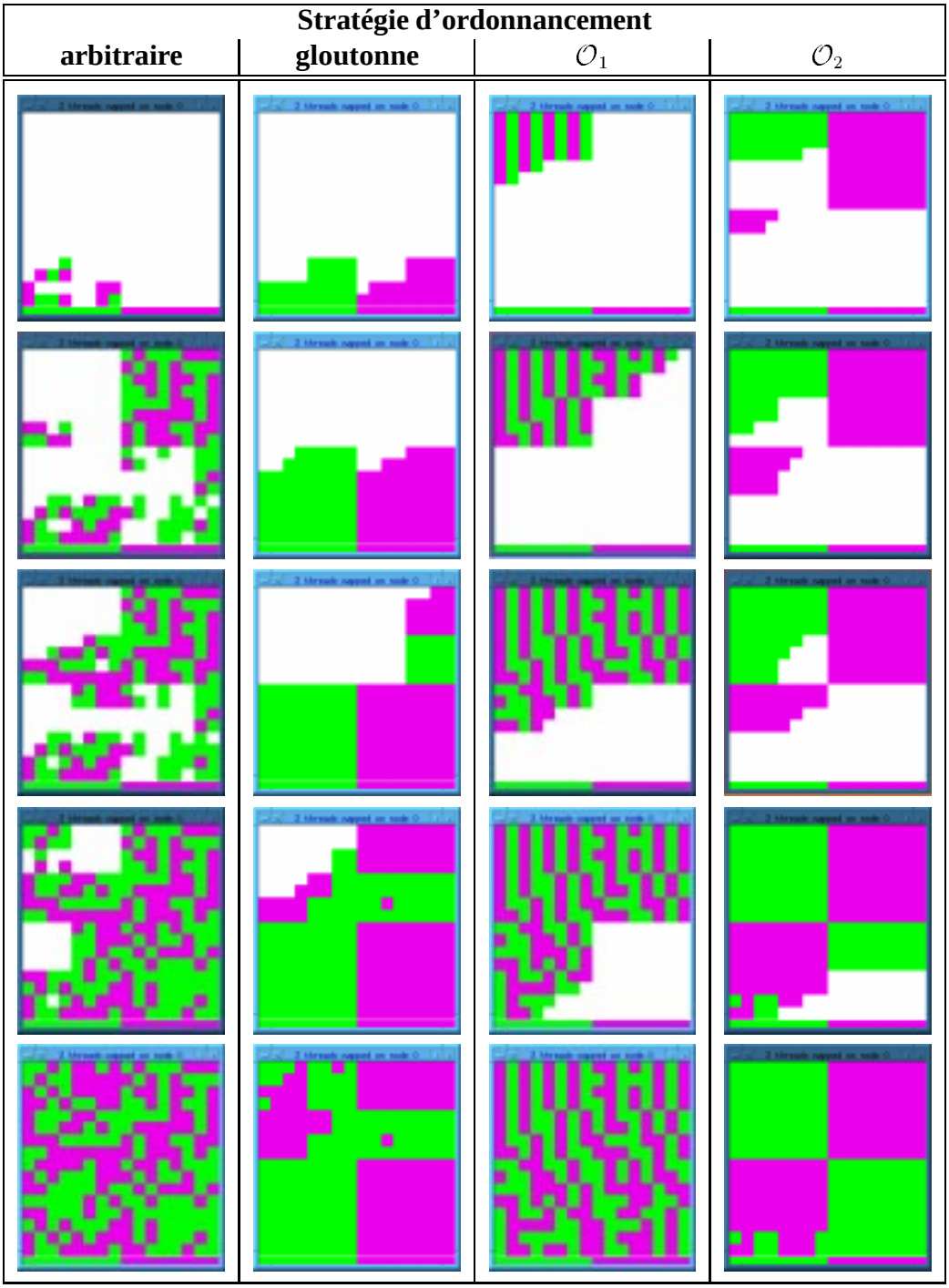


Figure 7.14 *Visualisation de l'ordre d'évaluation des tâches pour les différentes stratégies d'ordonnancement.*
L'exécution a eu lieu sur deux processeurs. Sont colorées les zones de l'image qui ont été calculées (une couleur par processeur) : le temps s'écoule de haut en bas. Les figures sur une même horizontale n'ont aucun lien particulier.

d'$\mathcal{O}_2$ contrairement à la stratégie gloutonne.

7.5 Conclusion

Nous avons étudié dans ce chapitre l'implantation et l'évaluation dans Athapascan-1 de deux politiques d'ordonnancement permettant un contrôle de la consommation mémoire.

L'ordonnancement est assuré par un module séparé de la bibliothèque Athapascan-1. Ce module facilite l'implantation d'une grande variété de stratégies, ce qui permet d'adapter la politique d'ordonnancement à la situation traitée (application, machine, contraintes mémoire, *etc.*).

Les évaluations effectuées en comparaison avec deux autres stratégies, simples mais ne contrôlant pas l'utilisation mémoire, qui sont un placement arbitraire des tâches et une stratégie gloutonne, montrent que la consommation mémoire des deux stratégies proposées sont conformes aux prédictions annoncées, comme illustré figure 7.12 page 148. Cependant, le nombre de messages généré (dû au caractère centralisé de l'algorithme) par la politique $\mathcal{O}_1$ qui est la plus performante en mémoire restreint son utilisation à une machine possédant peu de processeurs et un réseau rapide. En relâchant certaines contraintes sur l'ordre d'exécution des tâches, c'est-à-dire en ne suivant plus d'aussi près l'ordre de « référence », il est possible d'implanter une version distribuée de cet algorithme : $\mathcal{O}_2$.

8

Conclusion et perspectives

Nous avons présenté dans ce document l'**intérêt** que représente la modélisation de l'exécution d'une application par un **graphe de flot de données** dans la définition et l'implantation de langages parallèles. Les points suivants ont été abordés :

- Nous avons proposé un algorithme de **construction dynamique** du graphe de flot de données d'une exécution. Le graphe construit est caractéristique de l'instance de l'application[1] et est orienté et sans cycle. Il permet donc, entre autres, de définir implicitement un ordre total sur les tâches : l'ordre de « référence ».
- Nous avons défini, à partir de ce graphe, la **sémantique des accès aux données partagées** dans le cadre du langage Athapascan-1. Cette sémantique est lexicographique (toute lecture voit la dernière écriture selon l'ordre total de « référence »).
- Nous avons proposé et évalué un algorithme de **gestion distribuée** des états associés aux nœuds de ce graphe. Cet algorithme est réactif et ne génère, dans un certain nombre de cas, pas plus de communication qu'une implantation directe de l'application par processus communicants.
- L'**implantation** de ce graphe et de cet algorithme de gestion répartie constitue, dans le cas d'Athapascan-1, le cœur de la version distribuée.
- Nous avons proposé un **modèle de coût** associé à Athapascan-1, modèle permettant de garantir la durée et la consommation mémoire de toute exécution en fonction des caractéristiques de la machine hôte et de l'application.
- Enfin, nous avons implanté et évalué deux stratégies d'ordonnancement permettant de **contrôler la consommation mémoire** dans Athapascan-1.

[1] Une instance représente l'association d'une application et des données en entrées. Deux exécutions différentes de la même application avec les mêmes données correspondent donc à une même instance. Les tâches sont supposées sans effet de bord et la sémantique des accès aux données est telle que les valeurs résultats ne dépendent que des données en entrée et aucunement des conditions de l'exécution.

Le travail mené au cours de cette thèse est entièrement intégré dans la définition et l'implantation de l'interface de programmation **Athapascan-1**, supervisées par Jean-Louis Roch, et a été réalisé conjointement aux travaux de thèse de Gerson Cavalheiro [25] et Mathias Doreille [42]. La version distribuée de cette interface, basée sur la construction et la gestion distribuée du graphe de flot de données, est impliquée dans plusieurs travaux de recherche concernant entre autres une application de chimie quantique [75], des problèmes de traitement de données [23], un code d'optimisation combinatoire [35], une bibliothèque de calcul formel [57] et d'algèbre linéaire creuse [44]. Ces utilisations ont notamment motivé l'écriture d'un manuel d'utilisation et de référence et le développement d'un outil de visualisation du graphe de flot de données ainsi que l'interfaçage avec le visualisateur Paje [33] permettant d'analyser plus finement les performances d'Athapascan-0 et d'Athapascan-1 sur les applications.

Il ressort de ces travaux de recherche que :

– De part une description naturelle du flot de données et une sémantique lexicographique intuitive, la **simplicité du langage** permet un développement rapide et aisé des applications parallèles. De plus, l'existence d'une version purement séquentielle (dont les résultats seront rigoureusement identiques à toute exécution parallèle) facilite le déverminage des erreurs de codage de l'application.

– Le **modèle de coût** associé au langage permet de majorer en temps et en mémoire, selon la politique d'ordonnancement choisie, les ressources nécessaires à l'exécution d'une application sur une machine donnée. Cela permet à l'utilisateur de déterminer *a priori* si une stratégie d'ordonnancement peut convenir à sa situation (en offrant des garanties de performances qui entrent dans la limite des resources disponibles sur la machine). Les **performances** de l'application sont alors **portables** puisque garanties par le modèle de coût.

L'étude et les résultats présentés au cours de cette thèse ne doivent évidemment pas être vus comme un aboutissement mais bien plus comme une tentative de considérer le **flot de données** caractérisant une application comme **l'élément central** d'une interface et d'un modèle de programmation. Les continuations de ce travail sont nombreuses. Tout d'abord, à court terme, un travail complémentaire doit être réalisé pour minimiser plus finement le coût de gestion du graphe de flot de données. De plus, il serait également intéressant de valider le modèle sur des applications de grande envergure, d'effectuer une abstraction du modèle mémoire permettant de découpler la gestion du graphe de celle de ses états ou de permettre la dégénérescence séquentielle des tâches afin de pouvoir adapter la taille des tâches aux spécificités de la machine hôte. Mais, à plus long terme, deux interrogations fondamentales sont, à mon avis, posées par cette thèse.

La première concerne les restrictions imposées sur le langage afin d'être capable d'y associer un modèle de coût. Nous avons pu, en restreignant les droits d'accès des tâches sur la mémoire partagée, garantir la performance de certains ordonnanceurs en temps et en mémoire. Ces restrictions ont essentiellement pour but de définir un ordre séquentiel total sur les tâches, ordre par rapport auquel sont définies les performances. Se pose alors

la question suivante : « Est-il nécessaire que cet ordre soit total ? ». La réponse est assurément non, car il est possible par exemple d'imaginer un droit d'accès permettant à la fois la lecture et l'écriture concurrente[2] tout en étant capable de garantir la durée et la consommation mémoire de toute exécution. Ce nouveau droit permet d'accroître les possibilités d'expression du langage tout en conservant l'avantage de posséder un modèle de coût. Alors, jusqu'où pouvons nous aller ? Est-il possible de garantir les performances d'applications écrites, par exemple, à l'aide d'une bibliothèque de processus légers ? Si non, où est la limite ?

La seconde concerne le contrôle de la consommation mémoire. Le volume mémoire requis par toute exécution est, dans le modèle de coût, comparé à celui requis par une exécution de référence. Ici, cette exécution particulière est celle qui respecte l'ordre total de « référence » et définit un volume mémoire S_1. Il est clair que ce volume est différent du volume requis par une exécution purement séquentielle (où toutes les tâches sont remplacées, dans le code source de l'application, par leur corps). Les machines parallèles sont souvent utilisées pour leur capacité de stockage supérieure à celle d'une machine séquentielle. L'ordre d'exécution des tâches influençant directement la consommation mémoire de l'application, est-il possible de calculer, à la volée, un ordre minimisant cette consommation ? Quelles informations, ou restrictions, supplémentaires doivent être fournies par les tâches afin de permettre cette minimisation ?

[2]Ce type de droit est particulièrement adapté aux applications itératives, par exemple les itérations de type produit matrices creuses-vecteurs. Le modèle de fonctionnement d'un tel droit peut être rapproché du modèle de programmation BSP fonctionnant par *super-pas*.

Annexes

Annexe A

Une bibliothèque C++ pour l'interface de programmation Athapascan-1

Une bibliothèque de fonctions C++ implantant l'interface de programmation Athapascan-1 a été développée. Cette annexe présente les principales fonctionnalités et l'utilisation pratique de cette bibliothèque.

Nous présentons dans cette annexe l'utilisation pratique de la bibliothèque C++ qui implante l'interface de programmation Athapascan-1, cadre de cette thèse.

Ces quelques pages constituent un résumé de la documentation de référence de l'interface de programmation Athapascan-1 disponible électroniquement dans les pages du projet[1]. Seuls les aspects pratiques, tels la syntaxe par exemple, sont abordés dans cette annexe. Nous renvoyons le lecteur aux différents chapitres de cette thèse, ou plus particulièrement de celle de Mathias Doreille [42], pour une description plus approfondie du modèle et de l'interface de programmation Athapascan-1.

A.1 Machine d'exécution

La machine d'exécution au niveau de l'interface Athapascan-1 est la machine virtualisée par Athapascan-0 (voir section 7.2.1 page 131). Cette machine est composée d'un ensemble de **nœuds** de calculs **interconnectés** par un réseau. Chacun des nœuds héberge un pool de processeurs virtuels Athapascan-1 (voir section 7.2.3 page 133). Notons que chaque nœud de calcul peut contenir plusieurs processeurs physiques.

Cette machine est dotée d'une **mémoire virtuelle partagée** qui peut être manipulée par les tâches de l'application. Les tâches ne sont pas autorisées à faire des effets de bord

[1] L'*URL* est la suivante : `http://www-apache.imag.fr/software/ath1/`.

sur cette mémoire : toutes les données accédées doivent avoir été déclarées dans la liste des paramètres de la tâche.

A.2 Application

Tâche principale Initialement, le code exécutable de l'application est lancé sur chacun des nœuds de la machine. L'utilisateur doit, sur tous les nœuds, initialiser la bibliothèque Athapascan-1, ce qui est effectué classiquement dans le corps de la fonction `main`. Ensuite, il convient de n'exécuter la première tâche, dite principale, que sur **un seul** des nœuds de calcul. Le schéma typique de la fonction principale (`main`) est représenté dans la figure A.1. L'initialisation est effectuée en deux étapes afin de permettre à l'utilisateur d'initialiser, sur chaque nœud, ses propres modules (entre l'`init` et l'`init_commit`). Athapascan-1, à proprement parler, n'a besoin que d'une étape.

```
int main( int argc, char** argv )
{
   a1_system::init( argc, argv );
   a1_system::init_commit();
   if( a1_system::self_node() == 0 ) {
      // corps de la tâche principale
      ...
   } else {
      // ne rien faire
   }
   a1_system::terminate();
}
```

Figure A.1 *Schéma typique du corps de la fonction principale (*`main`*) d'une application Athapascan-1.*
Chacun des nœuds de calcul doit initialiser la bibliothèque Athapascan-1, mais un seul (ici le nœud 0) va exécuter la tâche pricipale. Les autres attendrons la terminaison globale. Les trois appels `init`*,* `init_commit` *et* `terminate` *constituent des barrières de synchronisation globales.*

Lancement Le lancement s'effectue au moyen d'une commande fournie par la bibliothèque Athapascan-0. Cette commande, `a0run`, prend en paramètre le nom de l'exécutable de l'application puis la liste des paramètres propres à l'application. Une option incontournable de cette commande est `-a0n` qui permet de préciser le nombre de nœuds qui doivent être utilisés sur la machine. Il est également possible de fixer la taille du pool d'exécution Athapascan-1 en utilisant l'option `-a1_pool`. Par exemple le lancement de l'application illustrée figure 2.9 pour calculer le $5^{\text{ème}}$ nombre de Fibonacci sur 2 nœuds avec 3 processeurs virtuels par nœud est effectué de la manière suivante :

```
a0run fibo -a0n 2 -a1_pool 3     5
```

A.3 Tâches

Définition du type d'une tâche Le type d'une tâche est défini comme une fonction classe, c'est-à-dire une classe définissant l'opérateur C++ `()`, comme illustré figure A.2. Le corps de cet opérateur représente le corps de la tâche.

```
class user_task {
public:
   void operator()( <liste des paramètres de la tâche> )
   {
      // corps de la tâche
      ...
   }
};
```

Figure A.2 *Définition d'une tâche.*
Une tâche est définie à l'aide d'une fonction classe. Chacun des paramètres doit être de type communicable (section A.4 page 162).

Création La création d'une tâche se fait en appelant la fonction `Fork` au lieu d'appeler directement le corps de la tâche. Cette fonction est générique et prend en paramètre le type de la tâche à créer. L'instruction suivante a pour effet de créer une tâche du type `user_task` défini précédemment :

```
Fork< user_task >() ( <paramètres effectifs> );
```

Exécution L'exécution, tout comme la création, d'une tâche sont deux opérations asynchrones : la tâche mère n'est pas interrompue dans son exécution. Les contraintes de précédence entre les tâches sont résolues par le système qui analyse les accès aux données partagées effectués par les tâches. La date d'exécution dépend donc de l'ordonnancement qui sera effectué. La seule garantie offerte par le système est que toutes les tâches seront exécutées et que les contraintes de précédence entre celles-ci seront respectées.

A.4 Paramètres formels des tâches

Types des paramètres autorisés Un paramètre est soit un paramètre par valeur soit une référence sur une donnée partagée.

- Dans le cas des paramètres par valeur, tous les types C++ sont autorisés à la condition qu'ils soient en plus communicables.
- Dans le cas d'une référence sur une donnée partagée le type dépend des accès qui seront effectués par la tâche (et toute sa descendance). Ces types sont énumérés dans le tableau 3.2 page 66. Les types sont génériques et doivent être instanciés par le type (forcément communicable) de la donnée partagée. Ainsi, dans le cas d'un accès en écriture directe sur une donnée partagée de type entière, le type du paramètre formel sera `Shared_r< int >`. Les écritures concurrentes prennent en

plus une fonction classe définissant la manière dont sera effectuée l'accumulation : `Shared_cw< F, T >` avec la classe `F` définie comme suit :

```
class F {
public:
   void operator()( T& x, const T& y ) {
      // accumulation de y dans x
      ...
   }
};
```

Types communicables Afin de pouvoir transmettre les données entre les différents nœuds de la machine, les types doivent posséder certaines propriétés (voir section 5.1.1 page 92). Les cinq fonctions suivantes sont requises :

- `T()`, le constructeur vide.
- `T( const T& )`, le constructeur de recopie.
- `~T()`, le destructeur.
- `a1_ostream& operator<<( a1_ostream& out, const T& x )`, l'opérateur d'emballage.
- `a1_istream& operator>>( a1_istream& in, T& x )`, l'opérateur de déballage.

La figure 5.1 page 93 représente un exemple de définition d'un type communicable. Les types de base C++ (`int`, `char`, ...), ainsi que tous les types de la STL[2] définis sur des types communicables, sont par défaut communicables.

A.5 Objets partagés

Chaque tâche doit déclarer les données partagées qui seront manipulées par son corps (les effets de bord sur la mémoire partagée sont en effet interdits).

Déclaration Un objet en mémoire partagée est déclaré de l'une des manières suivantes :

- `shared< T > x;` la référence `x` doit être affectée avant toute utilisation.
- `shared< T > x( 0 );` la référence `x` peut être utilisée comme paramètre effectif de tâche mais aucune valeur n'est associée à la donnée (il faut donc nécessairement une écriture avant toute tentative de lecture).
- `shared< T > x( (T*) p );` la référence `x` peut être utilisée et la donnée partagée possède une valeur initiale (celle pointée par `p`).

[2]La *Standard Template Library* est une bibliothèque générique de fonctions et de structures de données faisant partie des bibliothèques standards du langage C++.

Accès L'accès à la valeur d'une donnée partagée se fait à l'aide des fonctions membres suivantes (si le mode et le droit d'accès l'autorisent) :

- `const T& read() const;`
  Si la lecture est autorisée (`Shared_r< T > x`) l'appel `x.read()` retourne une référence constante sur la valeur contenue dans l'objet partagé.
- `void write( T* );`
  Si l'écriture est autorisée (`Shared_w< T > x`) l'appel `x.write( p )` stocke la valeur pointée par `p` dans la donnée partagée.
- `void cumul( const T& );`
  Si l'accumulation est autorisée (`Shared_cw< F, T > x`) l'appel `x.cumul( v )` accumule `v` dans la valeur de la donnée partagée `x` (à l'aide de la fonction `F`). Dans le cas où la donnée partagée ne contient aucune valeur lors de l'accumulation, une copie de `v` est effectuée à la donnée.
- `T& access();`
  Si la modification est autorisée (`Shared_r_w< T > x`) l'appel `x.access()` retourne une référence sur la valeur contenue dans la donnée partagée.

Règles de conversion lors du passage de paramètres Le tableau A.1 énumère la compatibilité, lors de la création d'une tâche, entre les types formels et effectifs des références sur les données partagées. Ces règles de conversion ne sont valables que lors des créations de tâches : lors des appels classiques de fonction ce sont les règles standards C++ qui sont appliquées.

Type du **paramètre formel**	Type (requis) du **paramètre effectif**

`Shared_r[p]< T >`	`Shared_r[p]< T >`
	`Shared_rp_wp< T >`
	`Shared< T >`

`Shared_w[p]< T >`	`Shared_w[p]< T >`
	`Shared_rp_wp< T >`
	`Shared< T >`

`Shared_cw[p]< F,T >`	`Shared_cw[p]< F,T >`
	`Shared_rp_wp< T >`
	`Shared< T >`

`Shared_rp_wp< T >` `Shared_r_w< T >`	`Shared_rp_wp< T >` `Shared< T >`

Tableau A.1 *Règles de conversion des types de paramètres lors de la création d'une tâche. Si une tâche requiert un paramètre d'un certain type formel, alors la tâche mère, lors de la création de la tâche fille, doit nécessairement posséder la référence sur la donnée partagée avec un type compatible lui permettant la création. Les types acceptables sont situés dans la colonne de droite du tableau.*

A.6 Ordonnancement des tâches

L'ordonnancement des tâches est assuré dans Athapascan-1 par un module séparé de la bibliothèque (voir section 7.2 page 130) ce qui permet d'adapter la politique d'ordonnancement à la situation traitée. Le choix de la politique d'ordonnancement peut être effectué lors de la compilation ou en cours d'exécution.

Politiques d'ordonnancement prédéfinies Le tableau A.2 contient les différentes politiques d'ordonnancement actuellement prédéfinies dans la bibliothèque Athapascan-1.

Type du groupe	Description de la politique d'ordonnancement associée
`a1_mapping::fixed`	Les tâches sont placées sur le site correspondant à leur localité. Si cette localité n'est pas définie, un site est choisi arbitrairement.
`a1_mapping::random`	Les tâches sont placées sur un nœud choisi arbitrairement.
`a1_mapping::cyclic`	Les tâches sont placées sur un nœud choisi de manière cyclique.
`a1_mapping::block_cyclic`	Les tâches sont placées cycliquement par groupes de n tâches. La valeur de n est fixée lors de la création du groupe d'ordonnancement.
`a1_work_stealing::basic`	Les tâches sont placées localement mais une technique de vol de travail permet à un nœud inactif de trouver une tâche à exécuter. Le nœud volé est choisi arbitrairement.
`a1_work_stealing::cyclic`	Le choix du nœud volé est effectué cycliquement.

Tableau A.2 *Politiques d'ordonnancement prédéfinies en Athapascan-1.*

Utilisation des politiques d'ordonnancement Il est nécessaire de créer un groupe associé à une certaine politique d'ordonnancement (voir section 7.2.4 page 134). Par exemple `a1_work_stealing::basic my_grp;` déclare un groupe géré par une politique de type gloutonne basée sur une stratégie de vol de travail. Ce groupe est ensuite utilisé comme suit :

- `a1_set_default_group( my_grp );` Toutes les tâches créées dans la suite de l'exécution de la tâche seront ordonnancées conformément à la politique associée au groupe `my_grp`. Ce groupe sera également le groupe par défaut pour tout le reste de la descendance de la tâche mère.
- `Fork< user_task >( my_group ) ( <paramètres effectifs> );` Dans ce cas, seule la nouvelle tâche créée sera ordonnancée conformément à la politique associée au groupe `my_grp`.

Ajout d'information aux tâches Il est possible de donner quatre informations à chaque tâche créée. Ces informations sont destinées aux politiques d'ordonnancement :

– Le **coût** d'exécution de la tâche (un `double` C++).
– La **localité** de la tâche (un `int` C++).
– La **priorité** de la tâche au sein du groupe (un `int` C++).
– Un **extra** dont le sens dépend de la politique d'ordonnancement utilisée (un `double` C++).

Ces informations sont précisées lors de la création des tâches de la manière suivante :

```
Fork< user_task >( SchedAttribute( <infos> ) ) ( <paramètres effectifs> );
```

L'utilisation qui sera faite de ces informations dépend de la stratégie d'ordonnancement utilisée.

Annexe B

Utilisation et performances d'Athapascan-1

Nous présentons dans ce chapitre quelques utilisations d'Athapascan-1 qui ont été faites au sein du projet. Ces utilisations vont des applications récursives générant énormément de parallélisme (les calculs de Fibonacci ou le placement des n-reines section B.1 page 167) à la résolution de problèmes d'algèbre linéaire dense (section B.3 page 169) en passant par une application de traitement de données (parallélisation de l'outil de compression `gzip`, section B.2 page 168).

B.1 Placement des n-reines

Le problème consiste à placer n reines sur un échiquier de taille $n \times n$ de sorte qu'aucune ne soit en prise. L'application calcule le nombre de telles positions. Les tâches sont récursivement créées afin de parcourir la totalité de l'arbre de recherche. Chaque tâche prend en entrée un échiquier avec les reines qui ont été précédemment placées.

Le tableau B.1 page 167 représente la durée d'exécution de cette application sur une machine parallèle IBM-SP1 sous AIX-4.2, contenant 32 processeurs[1] RS-6000 cadencés à 120 MHz et possédant 64 Mo de mémoire. L'accélération est proche de l'optimale, c'est-à-dire proche du nombre de processeurs utilisé.

taille	# tâches	T_s	T_1	T_2	T_4	T_8	T_{12}	T_{16}
13	1178	36.66	38.04	20.03	9.28	5.06	2.92	2.40
14	1537	223.13	225.34	113.95	57.01	28.44	18.65	14.65
15	1964	1451.92	1450.74	779.39	364.84	188.41	130.80	91.60

Tableau B.1 *Placement des n-reines sur une architecture parallèle.*
Ce tableau représente la durée d'exécution (en secondes) d'une application recherchant le nombre de positions valides en fonction de la taille de l'échiquier et du nombre de processeurs utilisés. T_s représente la durée d'exécution en utilisant la version séquentielle de la bibliothèque Athapascan-1.

[1] Seuls 21 noeuds étaient en service au moment des expérimentations.

B.2 Outil de compression `gzip` parallèle

La parallélisation de l'application `gzip` permettant la compression de fichier a été réalisée par Bertrand Carton et Fabien Giquel. Le rapport [23] contient une description complète des résultats.

Le point de départ de cette parallélisation est le code séquentiel de `gzip`. Une rapide encapsulation en C++ a permis de le rendre exécutable en tant que programme Athapascan-1. `gzip` possède la particularité de pouvoir décompresser les fichiers concaténés. La parallélisation de la compression reposait donc sur une technique de type « diviser pour reigner » : le fichier en entrée était découpé en blocs. Chaque bloc était compressé en parallèle puis les résultats de ces compressions étaient concaténés afin d'obtenir la compression du fichier total. En pratique, le programme `a1_gzip` est implanté par un ensemble de tâches indépendantes, chaque tâche prenant en entrée une portion d'un fichier (déjà stocké en mémoire) et fournit en sortie une zone de mémoire contenant la portion du fichier compresse. D'autres tâches prennent en charge la lecture du fichier source et l'écriture du fichier sortie.

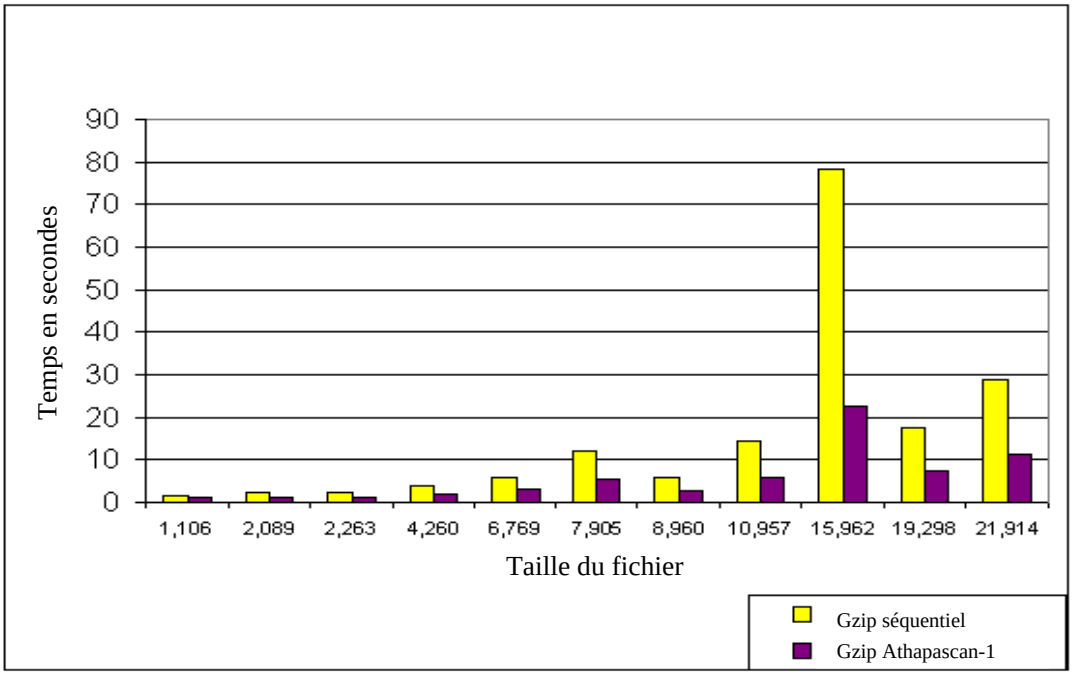


Figure B.1 *Performance d'un algorithme de compression de fichiers.*
Ces histogrammes représentent la durée d'exécution (en seconde) de l'application pour différentes tailles et types de fichiers sur une architecture de type SMP *possédant* 4 *processeurs.*

Le graphe de la figure B.1 présente les temps (en secondes) obtenus pour l'exécution d'`a1_gzip` pour la compression de différents fichiers de différents tailles et de différentes natures (textes, images, exécutables, données, *etc*). L'architecture utilisée est de type *SMP* possédant 4 processeurs pentiumPro cadencés à 200MHz avec 256 Mo de mémoire vive. Le système d'exploitation utilisé est SunOs 5.6.

B.3 Algèbre linéaire dense

Les résultats présentés dans cette section ont été obtenus par Mathias Doreille dans le cadre de sa thèse et sont présentés de manière plus complète dans [42]. La version d'Athapascan-1 utilisée pour réaliser ces tests est une version optimisée, mais restreinte, tirant profit d'une description « statique » du graphe de flot de données de l'application. Cette version, étudiée dans [42], est brièvement présentée à la section 5.2.3.4 page 104.

Cette section présente la parallélisation en Athapascan-1 de trois algorithmes de base d'algèbre linéaire :

- Le calcul du produit C de deux matrices denses A et B.
- La factorisation LU d'une matrice dense A (avec L triangulaire inférieure et U triangulaire supérieure à diagonale unitaire). Cette factorisation est réalisée par une élimination de Gauss.
- La factorisation LL^t d'une matrice dense A, symétrique, tel que $A = LL^t$ avec L triangulaire inférieure. Cette factorisation est réalisée par une élimination de Cholesky.

Les trois algorithmes utilisés ici sont des variantes des algorithmes classiques dans lesquels les calculs sont restructurés de manière à exprimer les algorithmes sous la forme d'opération sur des sous matrices (des blocs) plutôt que par des opérations sur les éléments scalaires des matrices. Ceci permet d'utiliser les BLAS de niveau 3 [39] dont l'efficacité n'est plus à prouver.

Les expérimentations ont été réalisées sur les deux machines parallèles suivantes :

- Un réseau de 20 stations SUN sous Solaris 7, chacune contenant quatre processeurs Ultra Sparc II cadencés à 295 MHz et possédant 512 Mo de mémoire. La puissance de crête d'un processeur de cette machine est de 390 Mflops (million d'opérations flottantes par seconde)[2]. Les stations sont reliées entre elles par un réseau Myrinet. Les communications sont réalisées à l'aide de la bibliothèque MPICH-GM.
- Une machine IBM-SP1 sous AIX-4.2, contenant 32 processeurs[3] RS-6000 cadencés à 120 MHz et possédant 64 Mo de mémoire. La puissance de crête d'un processeur de cette machine est de 100 Mflops[4]. Pour les communications, nous avons utilisé la bibliothèque MPI d'IBM qui utilise la bibliothèque de communication native MPL du SP1. Les communications sont réalisées à l'aide de la bibliothèque MPI.

B.3.0.1 Placement cyclique bidimensionel des tâches

La figure B.2 page 170 montre les performances obtenues sur 16 noeuds (64 processeurs) du réseau de stations SMP avec une partitions des matrices en blocs de taille

[2] Puissance estimée sur le produit de deux matrices de tailles 100×100 à l'aide de la fonction `dgemm` de la bibliothèque BLAS fournie par SUN (Sun Performance Library 2.0). Le nombre d'opérations flottantes d'un produit de matrice de taille n est estimé égal à $2n^3$.

[3] Seuls 21 noeuds étaient en service au moment des expérimentations.

[4] Puissance estimée sur le produit de deux matrices de tailles 150×150 à l'aide de la fonction `dgemm` de la bibliothèque BLAS fournie par IBM.

100×100. L'exécution du programme est ici réalisée en lançant sur chaque noeud *SMP* un processus UNIX, contenant lui même 4 processus légers pour exploiter les 4 processeurs du noeud *SMP*. Avec cette configuration, la factorisation LU exploite jusqu'à 11 Gflops, ce qui se ramène à 172 Mflops par processeur, à comparer aux 390 Mflops de puissance de crête mesurée sur un processeur.

La figure B.3 page 171 montre les performances obtenues sur 16 noeuds du SP1 avec une partition des matrices en blocs de taille 100×100. L'exécution du programme est ici réalisée en lançant un processus UNIX sur chaque noeud mono-processeur de la machine. Le produit de matrice exploite alors jusqu'à $1,1$ Gflops, ce qui donne 68 Mflops par processeur, à comparer aux 100 Mflops de puissance de crête mesurée sur un processeur. La chute des performances pour le produit de matrice de taille 2800×2800 s'explique par un dépassement de la capacité mémoire sur un noeud du SP1 (limitée à 30 Mo de données résidant en mémoire centrale).

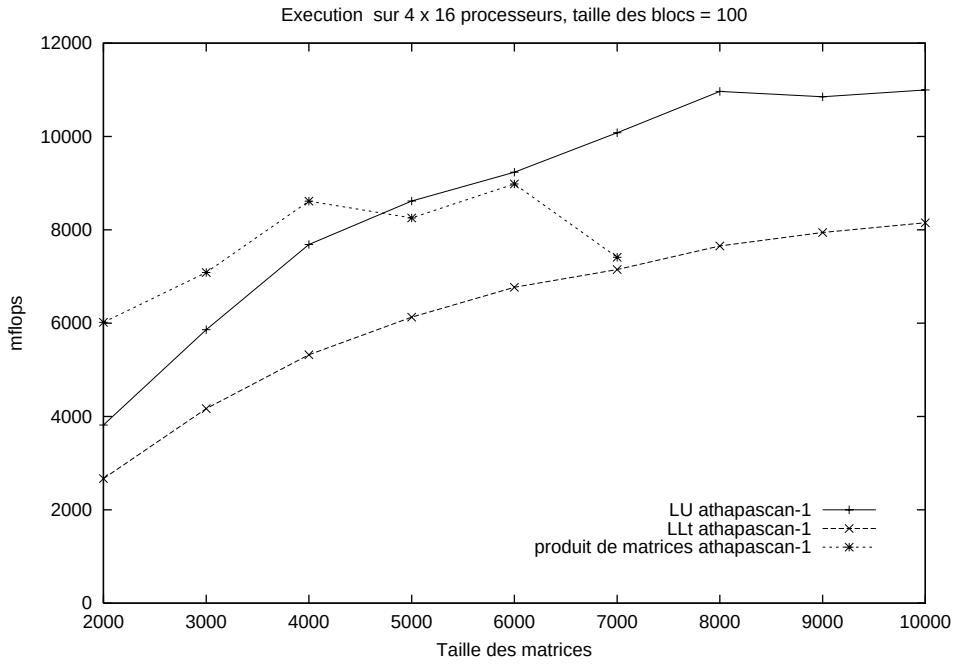


Figure B.2 *Problèmes d'algèbre linéaire dense sur un réseau de 16 stations SUN Sparc* SMP *à 4 processeurs.*

B.3.0.2 Comparaison avec ScaLapack

Nous comparons ici la factorisation de Cholesky dense écrite en Athapascan-1 avec la procédure de factorisation de Cholesky dense `pdpotrf` fournie par la bibliothèque ScaLapack [30]. La partition de la matrice, en bloc de taille 100×100 ainsi que le placement cyclique bidimensionnel des blocs sur les noeuds sont identiques pour les deux versions.

La figure B.4 page 172 montre les performances obtenues par Athapascan-1 et ScaLapack sur 16 noeuds (soit un total de 64 processeurs) du réseau de stations *SMP*. Les

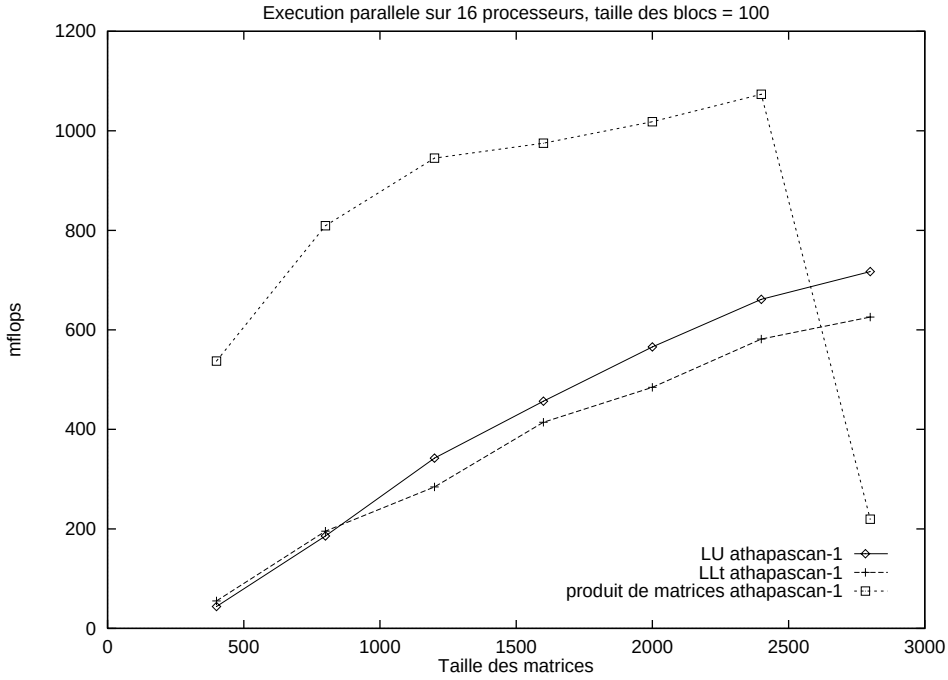


Figure B.3 *Trois problèmes d'algèbre linéaire dense sur un IBM SP1 à 16 processeurs. Ces trois applications sont une élimination de Gauss (LU) une factorisation de Cholesky (LL^t) et un produit de matrices.*

performances obtenus par Athapascan-1 sont jusqu'à un tiers meilleures que celles obtenues avec ScaLapack. Ces résultats s'expliquent par l'utilisation dans Athapascan-1 de processus légers pour exploiter les noeuds multi-processeurs tandis que ScaLapack utilise quatre processus UNIX communiquant par échange de messages.

La figure B.5 page 172 montre les performances obtenues par Athapascan-1 et ScaLapack sur un noeud *SMP* de la même machine. La courbe « LL^t Athapascan-1 » a été obtenue en utilisant Athapascan-1 avec un processus UNIX et quatre processus légers, la courbe « LL^t Athapascan-1 sans *threads* » a été obtenue en utilisant Athapascan-1 avec quatre processus UNIX et la courbe « LL^t ScaLapack » a été obtenue en utilisant Scalapack et quatre processus UNIX. On peut alors remarquer que l'utilisation de processus légers plutôt que de processus UNIX pour exploiter un noeud *SMP* améliore les performances obtenues jusqu'à un facteur deux.

La figure B.6 page 173 montre les performances obtenues pas Athapascan-1 et ScaLapack sur 16 noeuds de la machine SP1. Sur cette machine composée de noeuds monoprocesseurs, les résultats sont pratiquement similaires avec cependant un léger avantage pour Athapascan-1 lorsque la taille du problème augmente. Cela peut s'expliquer par la façon dont sont réalisées les communications dans les deux versions. En effet, dans ScaLapack, les communications sont réalisées de manière synchrone (seul type de communications fourni par la bibliothèque de communication BLACS [41]), tandis qu'au contraire, dans Athapascan-1, les communications sont toutes réalisées de manière asynchrones.

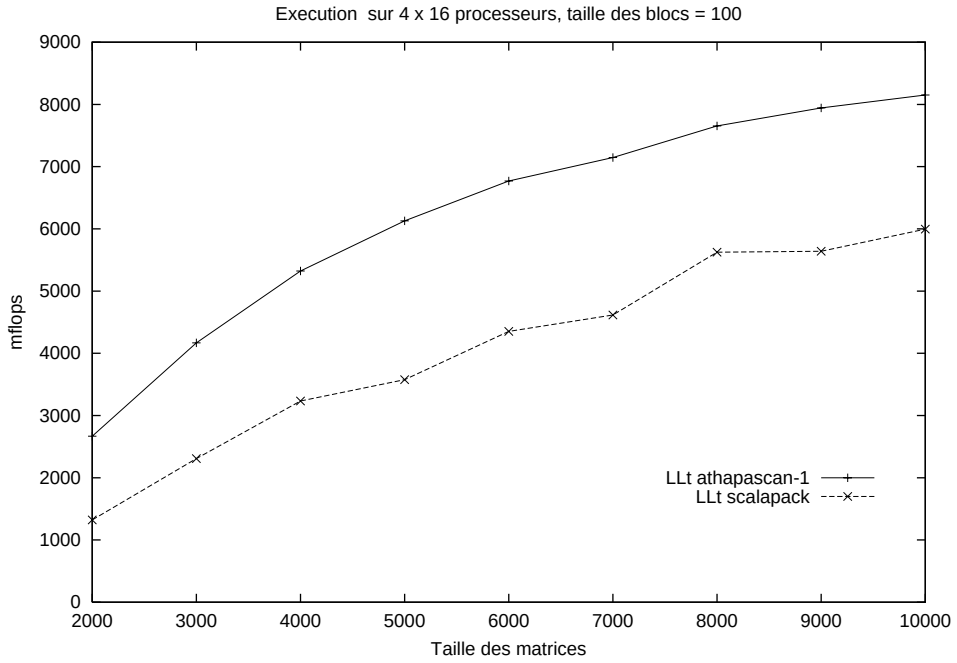


Figure B.4 *Comparaison Athapascan-1/ScaLapack sur un réseau de 16 stations SUN.*

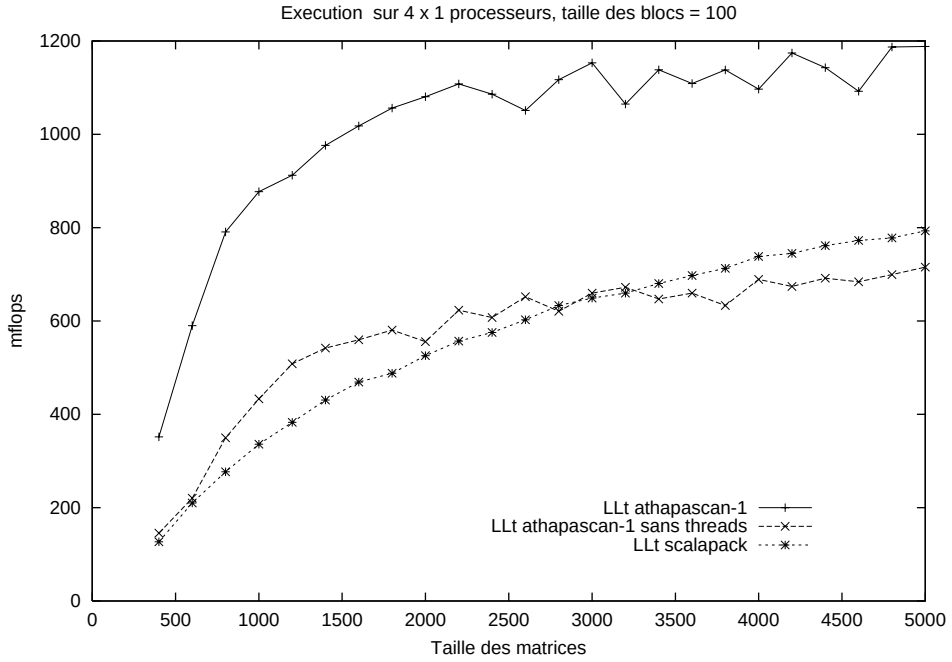


Figure B.5 *Comparaison Athapascan-1/ScaLapack sur une machine* SMP.
La courbe « LL^t Athapascan-1 » a été obtenue en utilisant Athapascan-1 avec un processus UNIX et quatre processus légers, la courbe « LL^t Athapascan-1 sans threads » a été obtenue en utilisant Athapascan-1 avec quatre processus UNIX et la courbe « LL^t ScaLapack » a été obtenue en utilisant Scalapack et quatre processus UNIX.

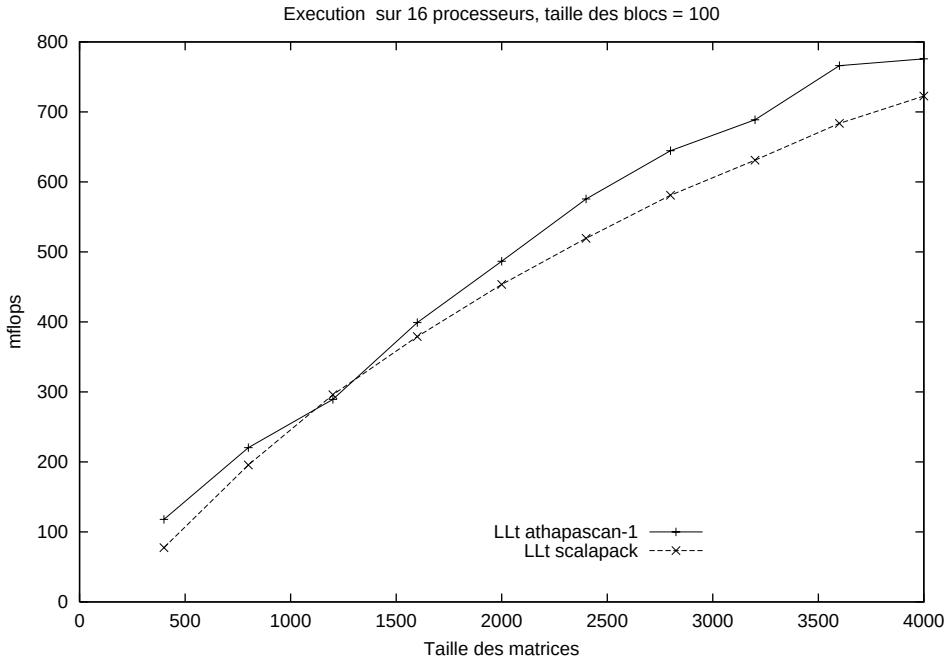


Figure B.6 *Comparaison Athapascan-1/ScaLapack sur un IBM SP1 à 16 processeurs.*

Bibliographie

[1] Adve (S. V.) et Gharachorloo (K.). – Shared memory consistency models : A tutorial. *IEEE Computer*, vol. 29, n12, Déc. 1996, pp. 66–76.

[2] Alverson (R.), Callahan (D.), Cummings (D.), Koblenz (B.), Porterfield (A.) et Smith (B.). – The Tera computer system. *In : Proceedings of the 1990 International Conference on Supercomputing*, pp. 1–6.

[3] ao (A. S. C.), Charpentier (I.) et Plateau (B.). – Un environnement modulaire pour l'exploitation des processus légers dans les méthodes de décomposition de domaine. *In : RenPar'11 – 11èmes rencontres francophones du parallélisme des architectures et des systèmes*.

[4] Balter (R.), Banâtre (J.-P.) et Krakowiak (S.) (édité par). – *Construction des systèmes d'exploitation répartis*, chap. 8 Gestion répartie des transactions (Balter (R.)), pp. 8.1–8.47. – INRIA, 1991.

[5] Balter (R.), Banâtre (J.-P.) et Krakowiak (S.) (édité par). – *Construction des systèmes d'exploitation répartis*, chap. 7 - Gestion répartie d'objets (Shapiro (M.)), pp. 7.1–7.36. – INRIA, 1991.

[6] Banerjee (U.), Eigenmann (R.), Nicolau (A.) et Padua (D. A.). – Automatic program parallelization. *Proceedings of the IEEE*, vol. 81, n2, Fév. 1993, pp. 211–243.

[7] Bernard (P.-E.), Gautier (T.) et Trystram (D.). – Large scale simulation of parallel molecular dynamics. *In : Proceedings of Second Merged Symposium IPPS/SPDP 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. – San Juan, Puerto Rico, Avr. 1999.

[8] Bevan (D. I.). – Distributed garbage collection using reference counting. *In : Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Volume II : Parallel Languages*, éd. par de Bakker (A. J. N. J. W.) et Treleaven (P. C.), pp. 176–187. – Eindhoven, The Netherlands, Juin 1987.

[9] Blelloch (G.), Gibbons (P.), Matias (Y.) et Narlikar (G.). – Space-efficient scheduling of parallelism with synchronization variables. *In : Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 12–23. – Newport, RI, Juin 1997.

[10] Blelloch (G. E.). – *NESL : A Nested Data-Parallel Language (Version 3.1)*. – Rapport technique nCMU-CS-95-170, Computer Science Department, Carnegie Mellon University, Sept. 1995.

[11] Blelloch (G. E.), Gibbons (P. B.) et Matias (Y.). – Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, vol. 46, n2, Mars 1999, pp. 281–321.

[12] Blelloch (G. E.) et Greiner (J.). – A provable time and space efficient implementation of NESL. *ACM SIGPLAN Notices*, vol. 31, n6, Juin 1996, pp. 213–225.

[13] Blumofe (R. D.). – *Executing multithreaded programs efficiently*. – Cambridge, MA, USA, Phd thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1995, 145p.

[14] Blumofe (R. D.), Joerg (C. F.), Kuszmaul (B. C.), Leiserson (C. E.), Randall (K. H.) et Zhou (Y. C. E.). – Cilk : an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, vol. 30, n8, Août. 1995, pp. 207–216.

[15] Blumofe (R. D.) et Leiserson (C. E.). – Scheduling multithreaded computations by work stealing. *In : Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pp. 356–368. – Santa Fe, NM, USA, Nov. 1994.

[16] Blumofe (R. D.) et Leiserson (C. E.). – Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, vol. 27, n1, Fév. 1998, pp. 202–229.

[17] Brent (R. P.). – The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, vol. 21, n2, Avr. 1974, pp. 201–206.

[18] Briat (J.), Gautier (T.) et Roch (J.-L.). – On-line scheduling. *In : Proc. of ESPPE'96, Parallel Programming Environments for High Performance Computing*, pp. 95–108.

[19] Briat (J.), Ginzburg (I.), Pasin (M.) et Plateau (B.). – Athapascan runtime : Efficiency for irregular problems. *Lecture Notes in Computer Science*, vol. 1300, 1997, pp. 591–599.

[20] Carissimi (A.). – *Le noyau exécutif Athapascan-0 et l'exploitation de la multiprogrammation légère sur les grappes de stations multiprocesseurs*. – Thèse de doctorat, Laboratoire de Modélisation et de Calcul, Institut National Polytechnique de Grenoble, France, Novembre 1999. À paraître.

[21] Carissimi (A.) et Pasin (M.). – Athapascan : An experience on mixing MPI communications and threads. *Lecture Notes in Computer Science*, vol. 1497, 1998, pp. 137–144.

[22] Carriero (N.) et Gelernter (D.). – How to write parallel programs : A guide to the perplexed. *ACM Computing Surveys*, vol. 21, n3, Sept. 1989, pp. 323–357.

[23] Carton (B.) et Giquel (F.). – *Data Processing using Athapascan-1*. – 3rd year engineering school project report, ENSIMAG, Grenoble, Juin 1999.

[24] Casavant (T. L.) et Kuhl (J. G.). – A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, vol. 14, n2, Fév. 1988, pp. 141–154.

[25] Cavalheiro (G. G. H.). – *Athapascan-1 : un mécanisme de régulation de charge pour les programmes parallèles*. – Thèse de doctorat, Laboratoire de Modélisation

et de Calcul, Institut National Polytechnique de Grenoble, France, Novembre 1999. À paraître.

[26] Cavalheiro (G. G. H.), Denneulin (Y.) et Roch (J.-L.). – A general modular specification for distributed schedulers. *Lecture Notes in Computer Science*, vol. 1470, 1998, pp. 373–376.

[27] Cavalheiro (G. G. H.) et Doreille (M.). – Athapascan : A C++ library for parallel programming. *In : Stratagem'96*. INRIA, pp. 75–76. – Sophia Antipolis, France, Juillet 1996.

[28] Cavalheiro (G. G. H.), Galilée (F.) et Roch (J.-L.). – Athapascan-1 : Parallel Programming with Asynchronous Tasks. *In : Proceedings of the Yale Multithreaded Programming Workshop*. – Yale, USA, Juin 1998.

[29] Cavalheiro (G. G. H.) et Roch (J.-L.). – Un schéma modulaire pour l'écriture des ordonnanceurs. *In : RenPar'98*. – Strasbourg, France, Juin 1998.

[30] Choi (J.), Dongarra (J.), Ostrouchov (L. S.), Petitet (A.), P. (A.), Walker (D. W.) et Whaley (R. C.). – Design and implementation of the ScaLAPACK LU, QR, and cholesky factorization routines. *Scientific Programming*, vol. 5, n3, Fall 1996, pp. 173–184.

[31] Dagum (L.) et Menon (R.). – OpenMP : An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, vol. 5, n1, Jan.–Mar. 1998, pp. 46–55.

[32] de Oliveira Stein (B.). – *Visualisation interactive et extensible de programmes parallèles à base de processus légers*. – Thèse de doctorat, Université Joseph Fourier, France, Octobre 1999. À paraître.

[33] de Oliveira Stein (B.) et Chassin de Kergommeaux (J.). – Interactive visualisation environment of multi-threaded parallel programs. *In : Parallel Computing : Fundamentals, Applications and New Directions*. pp. 311–318. – Elsevier.

[34] Denneulin (Y.). – *Conception et ordonnancement des applications hautement irrégulières dans un contexte de parallélisme à grain fin*. – Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, France, Janvier 1998.

[35] Denneulin (Y.). – Granularity and on-line scheduling of branch&bound tasks. *In : 12th conference of the European Chapter on Combinatorial Optimization*. – Bandol, France, Mai 1999.

[36] Devarakonda (M. V.) et Iyer (R. K.). – Predictability of Process Resource Usage : A Measurement-Based Study on UNIX. *IEEE Transactions on Software Engineering*, vol. 15, n12, Déc. 1989, pp. 1579–1586.

[37] Dijkstra (E. W.). – Cooperating sequential processes. *In : Programming Languages*, éd. par Genuys (F.), pp. 43–112. – Academic Press, 1968.

[38] Dongarra (J.), Du Croz (J.), Hammarling (S.) et Duff (I.). – A Set of Level-3 Basic Linear Algebra Subprograms. *ACM Trans.Math.Software*, vol. 16, 1990, pp. 1–17,18–28.

[39] Dongarra (J.), Du Croz (J.), Hammarling (S.) et Duff (I.). – A Set of Level-3 Basic Linear Algebra Subprograms. *ACM Trans.Math.Software*, vol. 16, 1990, pp. 1–17,18–28.

[40] Dongarra (J. J.), Otto (S. W.), Snir (M.) et Walker (D.). – *An Introduction to the MPI Standard*. – Rapport technique n UT-CS-95-274, Department of Computer Science, University of Tennessee, Jan. 1995.

[41] Dongarra (J. J.) et Whaley (R. C.). – *A Users' Guide to the BLACS v1.0*. – Rapport technique, University of Tennessee, 1995.

[42] Doreille (M.). – *Athapascan-1 : vers un modèle de programmation parallèle pour le calcul scientifique*. – Thèse de doctorat, Laboratoire de Modélisation et de Calcul, Institut National Polytechnique de Grenoble, France, Novembre 1999. À paraître.

[43] Doreille (M.), Cavalheiro (G. G. H.) et Roch (J.-L.). – Régulation dynamique en athapascan : Exemple d'un tri parallèle probabiliste optimal. *In : Huitièmes Rencontres du Parallélisme*. LaBRI, pp. 181–184. – Bordeaux, France, Mai 1996.

[44] Doreille (M.), Dumitrescu (B.), Roch (J.-L.) et Trystram (D.). – Two-dimensional block partitionings for the parallel sparse Cholesky factorization. *Numerical Algorithms*, vol. 16, n1, Feb 1998, p. 17.

[45] Doreille (M.), Galilée (F.) et Roch (J.-L.). – Graphe de flot de données ATHAPASCAN et ordonnancement. *In : RenPar'9 — 9èmes rencontres francophones du parallélisme*, éd. par André Schiper (D. T.). – Lausanne, Suisse, Mai 1997.

[46] Elleuch (A.), Kanawati (R.), Muntean (T.) et ghazali Talbi (E.). – Dynamic load balancing mechanisms for a parallel operating system kernel. *In : Int. Conf. on Parallel and Vector processing CONPAR 94*. pp. 864–877. – Linz, Austria, Sept. 1994.

[47] Eykholt (J. R.), Kleiman (S. R.), Barton (S.), Faulkner (R.), Shivalingiah (A.), Smith (M.), Stein (D.), Voll (J.), Weeks (M.) et Williams (D.). – Beyond multiprocessing : Multithreading the SunOS kernel. *In : Proceedings of the Usenix Summer 1992 Technical Conference*. pp. 11–18. – Berkeley, CA, USA, Juin 1992.

[48] Feo (J. T.), Cann (D. C.) et Oldehoeft (R. R.). – A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, vol. 10, n4, Déc. 1990, pp. 349–366.

[49] Fonlupt (C.). – *Distribution dynamique de données sur machines SIMD*. – Thèse de doctorat, UFR d'IEEA de Lille, Décembre 1994.

[50] Frigo (M.). – *The weakest memory model*. – Master's thesis, Massachusetts Institute of Technology, Jan. 1998.

[51] Frigo (M.), Leiserson (C. E.) et Randall (K. H.). – The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, vol. 33, n5, Mai 1998, pp. 212–223.

[52] Frigo (M.) et Luncangco (V.). – Computation-centric memory models. *In : Proc. of the 10th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'98)*, pp. 240–249.

[53] Froidevaux (C.), Gaudel (M.-C.) et Soria (M.). – *Types de données et algorithmes*. – Paris, Ediscience international, 1993.

[54] Galilée (F.) et Roch (J.-L.). – Langages pour l'expression dynamique de parallélisme et graphe de tâches. *In : ICaRE'97 : conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*, éd. par Barth (D.), Chassin de Kergommeaux (J.), Roch (J.-L.) et Roman (J.). – CNRS, Déc. 1997.

[55] Galilée (F.), Roch (J.-L.), Cavalheiro (G. G. H.) et Doreille (M.). – Athapascan-1 : On-line building data flow graph in a parallel language. *In : Pact'98*. – Paris, France, Oct. 1998.

[56] Gautier (T.). – *Calcul formel et parallélisme : Conception du Système Givaro et Applications au Calcul dans les Extensions Algébriques*. – Thèse de doctorat, Institut National Polytechnique de Grenoble, France, Juin 1996.

[57] Gautier (T.) et Roch (J.-L.). – PAC++ System and Parallel Algebraic Numbers Computation. *In : First International Symposium on Parallel Symbolic Computation (PASCO'94)*, éd. par Kong (H.), pp. 145–153.

[58] Gautier (T.), Roch (J.-L.) et Villard (G.). – Regular versus irregular problems and algorithms. *Lecture Notes in Computer Science*, vol. 980, 1995, pp. 1–26.

[59] Gelernter (D.). – Generative communication in linda. *ACM Trans. on Programming Languages and Systems*, vol. 7, n1, Jan. 1985, pp. 80–112.

[60] Gerasoulis (A.) et Yang (T.). – Pyrros : Mapping and compiling programs on scalable parallel architectures. *IEEE parallel and distributed technology : systems and applications*, vol. 1, n3, Août. 1993, pp. 81–82.

[61] Ginzburg (I.). – *Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications*. – Thèse de doctorat, Institut National Polytechnique de Grenoble, France, Sept. 1997.

[62] Goudreau (M.), Lang (K.), Rao (S.) et Tsantilas (T.). – *The Green BSP library*. – Technical Report nCR-TR-95-11, University of Central Florida, 1995.

[63] Graham (R. L.). – Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, vol. 45, 1966, pp. 1563–1581.

[64] Graham (R. L.). – Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, vol. 17, Mars 1969, pp. 416–429.

[65] Halstead (R. H.). – Multilisp : A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, vol. 7, n4, Oct. 1985, pp. 501–538.

[66] High Performance Fortran Forum. – High Performance Fortran language specification. *Scientific Programming*, vol. 2, n1-2, 1993, pp. 1–170.

[67] Hill (J. M. D.), McColl (B.), Stefanescu (D. C.), Goudreau (M. W.), Lang (K.), Rao (S. B.), Suel (T.), Tsantilas (T.) et Bisseling (R.). – *BSPlib : The BSP Programming Library*. – Rapport technique nMay, Oxford University Computing Laboratory, 1997.

[68] Hu (Y. C.), Lu (H.), Cox (A.) et Zwaenepoel (W.). – OpenMP for networks of SMPs. *In : Proc. of the Second Merged Symp. IPPS/SPDP 1999)*.

[69] IEEE. – *IEEE 1003.1c-1995 : Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2 : Threads Extension (C Language)*. – 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, IEEE Computer Society Press, 1995.

[70] Joerg (C.). – *The Cilk system for parallel multithreaded computing*. – Thèse de PhD, Massachussets Institute of Technology, january 1996.

[71] Karp (R. K.), Luby (M.) et auf der Heide (F. M.). – Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, vol. 16, n4/5, Oct./Nov. 1996, pp. 517–542.

[72] Konig (J.-C.) et Roch (J.-L.). – Machines virtuelles et techniques d'ordonnancement. *In : ICaRE'97 : conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*, éd. par Barth (D.), Chassin de Kergommeaux (J.), Roch (J.-L.) et Roman (J.). – CNRS, Déc. 1997.

[73] Lamport (L.). – How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, vol. C-28, n9, Sept. 1979, pp. 690–691.

[74] Levesque (J. M.) et Friedman (R.). – The state of the art in automatic parallelisation. *In : Supercomputing Europe '93*. pp. 95–107. – Utrecht, Netherlands, 1993.

[75] Maillard (N.), Roch (J.-L.) et Valiron (P.). – Parallélisation du calcul ab-initio de l'énergie de corrélation électronique. *In : RenPar'9 – 9èmes rencontres francophones du parallélisme*, pp. 45–48. – Lausanne, Suisse, Mai 1997.

[76] Marsh (B. D.), Scott (M. L.), LeBlanc (T. J.) et Markatos (E. P.). – First-class user-level threads. *In : Proceedings of the 13th ACM Symposium on Operating Systems Principle*, pp. 110–121. – Pacific Grove, CA, Oct. 1991. Published in ACM Operating Systems Review Vol.25, No.5, 1991.

[77] Mc-Carthy (J.). – Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, vol. 4, 1960, pp. 184–195.

[78] McGraw (J.). – *SISAL : Streams and Iterations in a Sigle-Assignment Language – Reference Manual*. – Rapport technique nManual M-146, Lawrence Livermore National Lab., 1985.

[79] Message Passing Interface Forum. – MPI : A Message Passing Interface. *In : Proceedings, Supercomputing '93 : Portland, Oregon, November 15–19, 1993*, éd. par IEEE. pp. 878–883. – IEEE Computer Society Press.

[80] Mohr (E.), Kranz (D. A.) et Halstead, Jr. (R. H.). – Lazy task creation : A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, n3, Juil. 1991, pp. 264–280.

[81] Namyst (R.). – PM^2 *: un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* – Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, France, Septembre 1997.

[82] Narlikar (G. J.) et Blelloch (G. E.). – *A Framework for Space and Time Efficient Scheduling of Parallelism*. – Rapport technique nCMU-CS-96-197, Computer Science Department, Carnegie Mellon University, Déc. 1996.

[83] Narlikar (G. J.) et Blelloch (G. E.). – Space-efficient implementation of nested parallelism. *In : Proceedings of the ACM SIGPLAN Symposium on Principles and Practice od Parallel Programming (PPOPP-97)*. pp. 25–36. – New York, Juin 18–21 1997.

[84] Narlikar (G. J.) et Blelloch (G. E.). – *Pthreads for Dynamic Parallelism*. – Rapport technique nCMU-CS-98-114, Computer Science Department, Carnegie Mellon University, Avr. 1998.

[85] Oaks (S.) et Wong (H.). – *Java Threads*. – 981 Chestnut Street, Newton, MA 02164, USA, O'Reilly & Associates, Inc., 1999, second édition, 344p.

[86] Plainfossé (D.). – *Distributed Garbage Collection and Reference Management in the Soul Object Support System*. – Paris (France), Thèse de PhD, Université Paris-6, Pierre-et-Marie-Curie, Juin 1994. Available from INRIA as TU-281, ISBN-2-7261-0849-0.

[87] Plainfossé (D.) et Shapiro (M.). – A survey of distributed garbage collection techniques. *Lecture Notes in Computer Science*, vol. 986, 1995, pp. 211– ? ?

[88] Plateau (B.) et al. – *Présentation d'*APACHE. – Rapport APACHE n 1, Grenoble, IMAG, Oct. 1993.

[89] Ranade (A. G.). – How to emulate shared memory. *In : Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, éd. par Chandra (A. K.). pp. 185–194. – Los Angeles, CA, Oct. 1987.

[90] Randall (K. E.). – *Cilk : Efficient Multithreaded Computing*. – Massaschussetts Institute of Technology, Thèse de PhD, Department of Electrical Engineering and Computer Science, Juin 1998.

[91] Raynal (M.). – Critères de cohérence pour les systèmes à mémoire partagée. *In : ICaRE'97 : conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*, éd. par Barth (D.), Chassin de Kergommeaux (J.), Roch (J.-L.) et Roman (J.), pp. 309–323. – CNRS, Déc. 1997.

[92] Rinard (M. C.). – *The Design, Implementation, and Evaluation of Jade : a portable implicitly parallel programming language*. – Thèse de PhD, Stanford University, 1994.

[93] Rinard (M. C.) et Lam (M. S.). – The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, vol. 20, n3, Mai 1998, pp. 483–545.

[94] Roch (J.-L.) et Villard (G.). – *Parallel Computer Algebra*. – Rapport technique, Lecture notes for a tutorial, 1997. ISSAC'97.

[95] Roch (J.-L.), Villard (G.) et Roucairol (C.). – Algorithmes irréguliers et ordonnancement. *In : Algorithmes parallèles, analyse et conception II*, éd. par Authié (G. e. a.), chap. 10. – Hermès, 1995.

[96] Ronsse (M.) et Bosschere (K. D.). – JiTI : Tracing Memory References for Data Race Detection. *In : Parallel Computing : Fundamentals, Applications and New Directions*. – Elsevier.

[97] Shapiro (M.), Dickman (P.) et Plainfossé (D.). – *SSP Chains : Robust, Distributed References Supporting Acyclic Garbage Collection*. – Rapports de Recherche n1799, Institut National de la Recherche en Informatique et Automatique, Nov. 1992. Also available as Broadcast Technical Report 1.

[98] Skillicorn (D.), Hill (J. M. D.) et McColl (W. F.). – Questions and answers about BSP. *Scientific Programming*, vol. 6, n3, Fall 1997, pp. 249–274.

[99] Stenström (P.). – VLSI support for a cactus stack oriented memory organization. *In : Twenty-First Annual Hawaii International Conference on System Sciences*, pp. 211–220.

[100] Stroustrup (B.). – *The C++ Programming Language : Third Edition*. – Reading, Mass., Addison-Wesley Publishing Co., 1997.

[101] Sunderam (V. S.). – PVM : a framework for parallel distributed computing. *Concurrency, practice and experience*, vol. 2, n4, Déc. 1990, pp. 315–339.

[102] Tarnvik (E.). – Dynamo — A portable tool for dynamic load balancing on distributed memory multicomputers. *Lecture Notes in Computer Science*, vol. 634, 1992, p. 485.

[103] Valiant (L. G.). – A bridging model for parallel computation. *Communications of the ACM*, vol. 33, n8, Août. 1990, pp. 103–111.

[104] Valiant (L. G.). – General purpose parallel architectures. *In : Handbook of Theoretical Computer Science*. – Elsevier Science Publishers and MIT Press, 1990.

[105] Watson (P.) et Watson (I.). – An efficient garbage collection scheme for parallel computer architecture. *In : Parallel Architectures and Languages Europe*, éd. par de Bakker (J. W.), Nijman (L.) et Treleaven (P. C.), pp. 432–443. – Eindhoven, The Netherlands, sv, Juin 1987.

[106] Willebeek-LeMair (M. H.) et Reeves (A. P.). – Local vs. global strategies for dynamic load balancing. *In : Proceedings of the 1990 International Conference on Parallel Processing. Volume 1 : Architecture*, éd. par Wah (B. W.). pp. 569–570. – Urbana-Champaign, IL, Août. 1990.

[107] Willebeek-LeMair (M. H.) et Reeves (A. P.). – Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, n9, Sept. 1993, pp. 979–993.

[108] Yang (T.) et Gerasoulis (A.). – DSC : Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, n9, Sept. 1994, pp. 951–967.

Résumé

Athapascan-1 : Interprétation distribuée du flot de données d'un programme parallèle

Cette thèse est centrée sur la modélisation de l'exécution d'une application parallèle par un graphe de flot de données. Ce graphe, qui relie les tâches aux données partagées, est construit de manière dynamique. Cette construction, indépendante de l'ordonnancement des tâches effectué, permet de définir la sémantique des accès aux données et de contrôler la consommation mémoire de toute exécution.

Nous étudions dans une première partie les algorithmes permettant la construction et la gestion d'un tel graphe de flot de données dans un environnement distribué. Un point crucial de ces algorithmes est la détection de terminaison des accès des tâches sur les données partagées. Nous proposons un algorithme réactif réalisant cette détection.

L'implantation de cet algorithme est au centre de l'implantation distribuée de l'interface de programmation parallèle Athapascan-1. Cette interface permet la description du parallélisme d'une application par création de tâches asynchrones. La sémantique (de type lexicographique) de cette interface est également définie à partir du graphe de flot de données.

Nous montrons dans une deuxième partie que la connaissance du flot de données d'une application permet de contrôler de manière théorique la durée et, surtout, la consommation mémoire de toute exécution. Ce contrôle est effectué à partir d'un ordonnancement séquentiel implicite des tâches. Nous proposons, implantons dans Athapascan-1 et évaluons deux algorithmes d'ordonnancement distribués permettant de limiter le volume de mémoire requis par toute exécution. Ces expérimentations permettent de valider les résultats théoriques obtenus.

Mots clés : Langage parallèle, graphe de flot de données, terminaison distribuée, ordonnancement à la volée, modèle de coût en temps et en mémoire.

Abstract

Athapascan-1 : distributed interpretation of parallel programs based on data flow analysis.

The topic of this thesis is the modelisation by a data-flow graph of any execution of a parallel application. This graph, that links tasks and data, is dynamically built. This construction is independant from the effective tasks' scheduling. This independance enables the definition of a data access semantic and the control of memory consumption.

We study in the first part the distributed algorithms enabling the construction and the management of this kind of graph. The central point of this management is the detection of the end of access of tasks on shared data. We propose a reactive algorithm performing this detection efficiently.

The implementation of this algorithm is the kernel of the distributed implementation of the Athapascan-1 interface for parallel programming. The semantic of data access in this programming interface is lexicographic and its definition is based on the data-flow graph of the application.

We show in a second part that the knowledge of the data-flow of an application enables a theoretical bound of the time and space of any execution. We propose, implement in Athapascan-1 and evaluate two distributed sheduling algorithms that limit the memory space used by any parallel execution. These experiments validate the theoretical results claimed by the two policies.

Keywords : parallel language, data-flow graph, distributed termination detection , online scheduling, cost model for time and space.