Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique

Mathias Doreille

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

T H E S E

pour obtenir le titre de

DOCTEUR DE L'INPG

Spécialité : "MATHEMATIQUES APPLIQUEES"

présentée et soutenue publiquement

par

Mathias Doreille

le 14 Décembre 1999

Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique

Directeur de thèse :

Jean-Louis Roch
Denis Trystram

Composition du jury :

Président :	Jean-Pierre Verjus
Rapporteurs :	Patrick R. Amestoy
	François Bodin
Examinateurs :	Brigitte Plateau
	Jean-Louis Roch
	Jean Roman
	Denis Trystram

préparée au **Laboratoire de Modelisation et Calcul**
et soutenue au **Laboratoire Informatique et Distribution**
dans le cadre de l'**Ecole Doctorale "Mathématiques et Informatique"**

Remerciements

Je voudrais tout d'abord remercier Jean-Pierre Verjus pour m'avoir fait l'honneur de présider le jury de cette thèse. J'adresse également tous mes remerciements à Patrick Amestoy et François Bodin pour avoir bien voulu consacrer une part de leur temps à rapporter sur ces travaux. Leurs remarques ont certainement permis d'améliorer la qualité de ce document. Merci également à Jean Roman d'avoir accepté de participer au jury de cette thèse.

Je remercie également Brigitte Plateau, responsable du projet Apache pour avoir bien voulu participer au jury et pour l'intérêt qu'elle a accordé à Athapascan-1. Merci à Denis Trystram pour m'avoir accueilli au sein de l'équipe parallélisme.

Et surtout je remercie Jean-Louis Roch qui m'a encadré depuis le DEA et qui m'a permis de poursuivre en thèse. Je ne saurai jamais lui exprimer toute ma gratitude pour son aide et son soutien durant ces quatre années.

Je tiens également à remercier Bogdan Dumitrescu pour l'aide qu'il m'a apportée dans la partie numérique de cette thèse ainsi que sa famille pour son accueil chaleureux en Roumanie.

Merci à tous les relecteurs de ce document qui ont eu à subir mon orthographe souvent fantaisiste et qui se sont livrés à une chasse aux fautes des plus fructueuses.

Je tiens également à remercier tous les membres de l'équipe parallélisme qui ont fait de ces années de thèse des années inoubliables. Merci à François (pas de doute, l'adresse que tu m'a refilée pour le SN n'était effectivement pas loin du paradis sur terre) et Gerson qui m'a beaucoup appris sur le Brésil (notamment la signification du mot minhoca). J'ai eu beaucoup de plaisir à travailler avec eux. Merci également à Paulo pour tous les moments inoubliables passés ensemble et à son cousin Gustavo qui m'a supporté pendant la difficile phase de rédaction. Merci à mes anciens collèges de bureau Alexandre et Benhur, à Jean-Guillaume participant le plus assidu au séminaire du mardi, à Nicolas, au gang des poseurs de bombes Alfredo, Gregory, Olivier et Christophe, à Titou qui m'a recueilli sous son toit, m'évitant ainsi de finir sous les ponts, à Andréa, Ekbel, Gilles, Ilan, Marcelo, Marta, Pierre-Eric, Renaud, Roberta, Yves. J'en oublie sûrement, je les prie de bien vouloir m'en excuser.

Enfin je remercie ma famille qui m'a soutenu et encouragé de loin durant ces années de thèse, et particulièrement mes parents qui ont bien voulu assister à ma soutenance.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

1

Introduction

Un grand nombre d'applications de calcul scientifique nécessitent des ressources de calcul et de mémoire sans cesse croissantes. Par exemple, dans le domaine de la simulation numérique, la complexité des modèles ainsi que la précision recherchée sont en constante augmentation (par exemple l'évolution des prévisions météorologiques). Les problèmes à traiter sont donc de taille croissante. Leurs résolutions en temps raisonnable nécessitent alors l'utilisation de machines plus puissantes.

Bien que les progrès réalisés en architecture des processeurs ont permis aux ordinateurs séquentiels de voir leur puissance de calcul doubler environ tous les deux ans, le parallélisme offre une alternative permettant d'augmenter encore d'un facteur la puissance de calcul. Le principe de base du parallélisme est simple. Il consiste à faire travailler ensemble plusieurs processeurs sur un même problème afin de le résoudre plus rapidement. Ce principe permet d'atteindre aujourd'hui des puissances de calcul dépassant le teraflops (10^{12} opérations flottantes par seconde) sur des machines constituées d'une dizaine de milliers de processeurs standards.

Cependant la programmation d'application sur ces machines parallèles est beaucoup plus complexe que sur des machines séquentielles. Les problèmes rencontrés sont multiples et de différentes natures : algorithmique et extraction du parallélisme de l'application, ordonnancement et placement des calculs sur les processeurs, gestion des synchronisations entre les activités parallèles pour les accès concurrents aux données de la mémoire partagée, gestion des communications entre processeurs, *etc*. De plus, de part la diversité et la constante évolution des architectures des machines parallèles, les applications écrites pour un type de machine donné sont difficilement portables.

Une solution à ce problème est alors de fournir des mécanismes permettant d'exprimer à un haut niveau le parallélisme de l'application. L'objectif est de décharger le

programmeur au maximum des problèmes rencontrés lors de la programmation parallèle, problèmes qui peuvent être résolus de manière automatique. Cette solution conduit également à une meilleure portabilité des applications puisque les programmes font complètement abstraction de la machine parallèle qui sera utilisée.

Ainsi beaucoup de modèles de programmation ont été proposés dans ce sens. Par exemple, les environnements fournissant une mémoire virtuellement partagée comme TreadMarks [3] ou bien certains langages supportant des mécanismes de partage de données comme LINDA [17] et ORCA [9, 8] libèrent le programmeur du placement et des communications des données de l'application sur une machine à mémoire distribuée. Les processus légers de la norme POSIX [75] ou du langage Java [96], en permettant une abstraction du nombre de processeurs, libèrent le programmeur du placement des calculs et de la régulation de la charge des processeurs. Les modèles de programmation exploitant le parallélisme de données comme par exemple HPF [65] offrent, pour les applications régulières, un niveau d'abstraction supplémentaire puisque le programmeur est libéré des problèmes de synchronisation. La parallélisation automatique à la compilation va encore plus loin dans l'abstraction du parallélisme puisque le programme est alors exprimé dans un langage séquentiel.

Cependant, plus le niveau d'abstraction dans la description du parallélisme est élevé plus les performances obtenues par l'application sont tributaires du système sous-jacent, compilateur et/ou support d'exécution. Le problème difficile est alors de trouver un bon compromis entre le niveau d'abstraction du modèle de programmation et sa capacité à garantir une exécution parallèle efficace (théoriquement et expérimentalement).

Dans ce cadre des langages et environnements de programmation parallèle haut niveau, *i.e.* qui font une totale abstraction de la machine parallèle sous-jacente, nous présentons dans cette thèse l'interface de programmation parallèle Athapascan-1. Cette interface de programmation vise aussi bien les applications régulières que les applications irrégulières et les machines cibles sont à la fois les architectures à mémoire partagée et les architectures à mémoire distribuées.

Cette thèse a été menée dans le cadre du projet APACHE[1] dont le but est la réalisation d'un environnement de programmation portable et efficace pour les machines parallèles. Deux modules complémentaires composent cet environnement :

– Athapascan-0 [47, 16, 15], le module exécutif, qui permet l'utilisation de la multiprogrammation légère dans un contexte distribué. Ce module, basé sur un couplage entre différentes bibliothèques de processus légers et de communication standards, constitue le noyau exécutif de l'environnement et donc la couche de portabilité de l'environnement Athapascan.

1. Algorithmique Parallèle et pArtage de CHargE [101], 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* du serveur du projet est `http://www-apache.imag.fr`.

– Athapascan-1, l'interface applicative qui fait l'objet de cette thèse. Ce module est l'interface de programmation des applications. Il implémente un modèle de programmation haut niveau basé sur un mécanisme de création de tâches communiquant entre elles par l'intermédiaire d'une mémoire partagée.

Outre ces deux modules, l'environnement Athapascan comprend également différents outils d'aide au développement adaptés à la multiprogrammation dans un contexte distribué, notamment un module de prise de traces, un module de réexécution déterministe et un module de visualisation de trace et de performances [28].

Les applications visées par l'interface applicative Athapascan-1 sont en particulier les applications irrégulières pour lesquelles l'obtention de performances passe par la mise en œuvre à l'exécution de mécanismes de régulation de charge et d'ordonnancement souvent complexes. Cependant, ces mécanismes de régulation de charge et d'ordonnancement sont souvent identiques d'une application à une autre. Pour réutiliser facilement de tels mécanismes, et ainsi faciliter la programmation de telles applications, il est alors nécessaire de séparer la description de l'application de son ordonnancement. L'idée directrice qui a conduit à la conception du modèle de programmation Athapascan-1 est d'utiliser le graphe de flot de données de l'application comme élément central pour séparer la description de l'application de son ordonnancement [111, 50, 14].

Les contributions de cette thèse sont les suivantes :

– Nous proposons (chapitre 3 page 41) une interface de programmation effective implantant ce modèle théorique. La caractéristique principale de cette interface est de permettre la description implicite du flot de données de l'application.

– Nous proposons (chapitre 5 page 75) une extension des résultats théoriques existants pour l'ordonnancement de graphe de flot de données avec communications. Une adaptation de l'algorithme ETF permettant un ordonnancement à la volée a ainsi été proposée.

– Nous validons expérimentalement cet environnement sur plusieurs applications (chapitre 7 page 113) et comparons les performances obtenues (sur des machines jusqu'à 64 processeurs) avec des bibliothèques parallèles standards (ScaLapack notamment).

– Nous illustrons la puissance et la simplicité de programmation de ce modèle en étudiant la parallélisation fine sur une application complexe : la factorisation de Cholesky d'un matrice creuse par un algorithme utilisant un partitionnement bidimensionnel de la matrice (chapitre 9 page 153). Les résultats expérimentaux obtenus montrent que le modèle de programmation peut être efficacement mis en œuvre à la fois sur des architectures à mémoire partagée et distribuée.

Organisation du document

Ce document est structuré en deux parties : la première (chapitres 3-6) est consacrée à l'interface de programmation Athapascan-1 et la seconde (chapitres 7- 9) est consacrée à sa validation par des applications de calcul scientifique et principalement par la factorisation de Cholesky d'une matrice creuse.

En introduction à la thèse, le chapitre 2 est consacré aux modèles de programmation parallèle permettant d'exprimer à un haut niveau le parallélisme de l'application. Nous présenterons tout d'abord les architectures des machines parallèles actuelles puis nous présenterons quelques modèles de programmation parallèle de haut niveau.

Nous débuterons ensuite la première partie du document par le chapitre 3 dans lequel le modèle de programmation d'Athapascan-1 sera présenté en détail et illustré sur deux exemples simples.

Le chapitre 4 présente le modèle d'exécution utilisé par Athapascan-1, c'est-à-dire la manière dont un programme écrit en Athapascan-1 est exécuté sur une machine parallèle. Nous montrerons comment le parallélisme implicite de l'application est extrait durant l'exécution à partir d'une analyse des dépendances de données entre les tâches. Nous montrerons ensuite comment ces dépendances de données sont explicitement représentées sous la forme d'un graphe de flot de données qui est utilisé pour l'ordonnancement et l'exécution des tâches de l'application. Nous verrons également comment l'algorithme d'ordonnancement utilisé, qu'il soit statique ou dynamique, peut être découplé des autres mécanismes nécessaires à l'exécution de l'application.

Le chapitre 5 s'intéresse aux algorithmes d'ordonnancement de graphe de flot de données et à leur application à Athapascan-1. Nous présenterons tout d'abord les algorithmes d'ordonnancement existants, aussi bien statiques que dynamiques, pour l'ordonnancement de graphes de précédence ou de graphes de flot de données. Puis nous proposerons un algorithme original d'ordonnancement dynamique adapté aux graphes de flot de données. Finalement ces algorithmes d'ordonnancement seront appliqués à l'exécution d'un programme Athapascan-1 : ils permettent de donner différents modèles de coût permettant d'évaluer le temps d'exécution d'applications Athapascan-1 pour différentes modélisations de machines parallèles.

Le chapitre 6 sera consacré à la bibliothèque C++ Athapascan-1, bibliothèque dont l'interface offre un support pour l'écriture d'applications dans le modèle de programmation décrit dans le chapitre 3 et dont l'implantation suit également de près le modèle d'exécution décrit dans le chapitre 4. Cette bibliothèque offre une interface pour l'écriture de politiques d'ordonnancement qui pourront ensuite être utilisées par l'application. Nous présenterons alors des implantations sur cette interface de certains des algorithmes d'ordonnancement présentés dans le chapitre 5.

La deuxième partie du document est ensuite consacrée à la validation de cette interface de programmation parallèle Athapascan-1 sur différentes applications, l'application principale étant la factorisation creuse de Cholesky.

Le chapitre 7 présente trois algorithmes d'algèbre linéaire dense qui ont été implantés en Athapascan-1. Nous montrerons les résultats expérimentaux obtenus sur différentes machines parallèles et pour différentes politiques d'ordonnancement des tâches. Ces résultats seront également comparés avec ceux obtenus par la bibliothèque ScaLapack.

Le chapitre 8 est consacré à la factorisation creuse de Cholesky en général et à sa parallélisation. Nous présenterons plus particulièrement un algorithme de factorisation numérique parallèle avec partitionnement bidimensionnel de la matrice, développé en collaboration avec Bogdan Dumitrescu de l'Institut Polytechnique de Bucarest [34].

Le chapitre 9 sera finalement consacré à l'implantation de cet algorithme de factorisation creuse de Cholesky en Athapascan-1. Nous présenterons tout d'abord des implantations de cette application réalisées dans d'autres environnements de programmation parallèle haut niveau ; puis nous présenterons l'implantation réalisée en Athapascan-1. Enfin, cette implantation sera évaluée expérimentalement et comparée à celles réalisées dans les autres environnements de programmation.

2

Modèles de programmation parallèle haut niveau

Nous présentons dans ce chapitre les principales caractéristiques des machines parallèles et les langages de programmation dits de haut niveau qui permettent de les abstraire, éventuellement avec des garanties de performances, dans le but de faciliter leur programmation.

2.1 Les architectures de machines parallèles

Malgré leur diversité, on peut regrouper la majorité des architectures parallèles actuelles en trois catégories.

2.1.1 Les architectures symétriques à mémoire partagée

Les machines de cette catégorie, appelées SMP (*Symmetric Multi-Processors*), sont composées d'un ensemble de processeurs identiques qui accèdent à une même mémoire, appelée mémoire partagée. Un réseau d'interconnection ou bien un bus est utilisé pour relier les processeurs à cette mémoire partagée. Le temps d'accès à un mot de la mémoire peut être uniforme pour les machine de type UMA (*Uniform Memory Access*), ou non uniforme pour les machines de type NUMA (*Non Uniform Memory Access*).

Le fait de disposer d'une mémoire commune entre les processeurs facilite grandement la mise en œuvre des applications parallèles, pour autant que les problèmes de cohérence soient pris en compte. Par ailleurs, certaines limites technologiques font que ces machines

sont peu extensibles et limitées aujourd'hui à quelques dizaines de processeurs pour les architectures de type UMA.

2.1.2 Les architectures à mémoire distribuée

Les machines de cette catégorie sont constituées d'un ensemble de nœuds. Chaque nœud possède un processeur et une mémoire appelée mémoire locale. Un réseau de communication relie entre eux les nœuds qui peuvent alors s'échanger des données. Ces communications entre nœuds se font soit par échange de messages entre les processeurs soit directement par accès à une mémoire distante. Dans les deux cas, le temps d'accès à un mot situé sur un nœud distant est très nettement supérieur au temps d'accès à un mot situé sur la mémoire locale du nœud.

La mise en œuvre d'applications parallèles sur ces machines est donc plus délicate puisque le programmeur doit distribuer et gérer ses données sur les mémoires locales en prenant en compte les différences de temps d'accès entre mémoire locale et distante. Par contre, ce type de machine est facilement extensible à la fois en nombre de processeurs et en capacité mémoire.

2.1.3 Les réseaux de multiprocesseurs à mémoire partagée

Cette catégorie de machines qui combine les deux architectures précédentes est la plus répandue actuellement. Une telle machine est constituée d'un ensemble de nœuds, reliés entre eux par un réseau de communication. Chaque nœud est constitué d'une machine SMP. Le temps d'accès à un mot situé sur un nœud distant est, comme pour les machines à mémoire distribuée, très supérieur au temps d'accès à un mot situé sur la mémoire locale du nœud. L'intérêt évident de ce type d'architecture est qu'à puissance de calcul égale, un réseau de multiprocesseurs à mémoire partagée a moins de nœuds qu'une machine à mémoire distribuée et donc une puissance de communication supérieure.

Ce type de machine a tendance à devenir prédominant. Il peut être vu comme une réponse au problème d'extensibilité des machines à mémoire partagée. Cependant la mise en œuvre d'une application parallèle exploitant efficacement l'ensemble de la machine reste délicate puisqu'il y a toujours des différences importantes de temps d'accès entre mémoire locale et distante. De plus le programmeur peut être amené à utiliser simultanément deux modèles de programmation différents. Si les processeurs sont sur des nœuds différents, le modèle par échange de messages est utilisé, mais par contre, si les processeurs sont sur un même nœud, le modèle par mémoire partagée est préférable pour éviter des copies inutiles.

2.2 Programmation bas niveau des architectures parallèles

La programmation de ces architectures a conduit au développement de noyaux d'exécution permettant de tirer parti de leurs caractéristiques.

Sur machine à mémoire partagée, des mécanismes sont fournis pour créer des processus, le programmeur gère ensuite la synchronisation entre ces processus en utilisant des primitives bas niveau comme les verrous ou les variables de condition. La norme Posix [75] est un standard proposant de tels mécanismes.

Sur machine à mémoire distribuée, il faut pouvoir communiquer entre deux nœuds. Ainsi des bibliothèques de communication ont été développées. Parmi celles-ci, les implantations du standard MPI [119] sont les plus utilisées notamment en calcul scientifique. Enfin sur les réseaux de machine SMP, des noyaux exécutifs ont été développés permettant d'allier processus légers et communications. Parmi ceux-ci, on peut citer Nexus [42], PM2 [93] et Athapascan-0 [16].

En fournissant au programmeur de tels mécanismes de contrôle fin de l'exécution parallèle, celui-ci peut implanter efficacement sur une machine donnée ses applications. Cependant ceci n'est obtenu qu'au prix d'un important effort de programmation (avec parfois des problèmes de vérification de la correction du programme) et d'une perte de portabilité.

2.3 Modèles de programmation parallèle haut niveau

Dans le but de faciliter la programmation parallèle ainsi que pour augmenter la portabilité des applications, différents modèles de programmation plus abstraits ont été proposés, que nous désignerons par « modèles de programmation parallèle haut niveau ». Ces modèles de programmations proposent différents paradigmes permettant d'abstraire la machine parallèle : mémoire partagée distribuée pour abstraire le réseau de communication, parallélisme applicatif pour abstraire les processeurs.

Le parallélisme de l'application peut également être explicite ou bien implicite. Dans le premier cas, les activités parallèles (les tâches) et les synchronisations entre ces activités sont explicitement décrites par le programme alors que dans le second cas, ce parallélisme est extrait de l'application à la compilation ou à l'exécution.

Nous détaillons dans la suite séparément chacun de ces paradigmes.

2.3.1 Abstraction du réseau de communication : mémoire partagée distribuée

Ce paradigme consiste à fournir des mécanismes de partage de données entre processeurs et ainsi cacher au programmeur les communications nécessaires à une exécution sur les architectures distribuées. Ce paradigme regroupe deux mécanismes de partage de données qui seront détaillés dans la suite : l'un, que nous désignerons par mémoire virtuellement partagée (abrégée par MVP) utilise la pagination et l'autre, que nous désignerons par mémoire partagée distribuée de niveau objet (abrégé par DSM pour *Distributed Shared Memory*) fourni des mécanismes de partage de données au niveau d'objets définis par l'utilisateur.

L'intérêt de ce paradigme de mémoire partagée est multiple. Tout d'abord, il simplifie la programmation sur machine distribuée en libérant le programmeur de la distribution et de la communication des données. Chaque processeur peut alors accéder une donnée quelconque de cette mémoire partagée, sans que le programmeur ait à se préoccuper du lieu où elle se situe (sur quelle mémoire locale ou distante) ni comment la ramener localement.

Par ailleurs les applications écrites selon ce paradigme sont bien mieux adaptées aux machines SMP que celles écrites dans un modèle de programmation par échange de messages. Lorsqu'une donnée est échangée entre deux processus, cette donnée est copiée de l'espace d'adressage d'un processus à l'autre dans un modèle de programmation par échange de messages comme MPI alors que dans un modèle de programmation à mémoire partagée la donnée est directement accédée (donc sans copie). De même, lorsqu'une même donnée est accédée par plusieurs processus, elle est dupliquée sur chacun des espaces d'adressage des processus dans un modèle de programmation par échange de messages alors que sur un modèle de programmation à mémoire partagée la donnée n'est pas copiée et n'est présente qu'une fois en mémoire physique. Ainsi, lors de l'exécution sur architecture SMP, une application programmée sur un modèle à mémoire partagée nécessitera moins de mémoire et s'exécutera généralement plus vite que la même application écrite dans un modèle de programmation par échange de messages.

L'approche mémoire virtuellement partagée consiste à implanter un espace d'adressage unique de la mémoire de manière à fournir au programmeur l'illusion d'une mémoire physiquement partagée. Cette approche est généralement mise en œuvre par un mécanisme de pagination sur les différents nœuds de la machine distribuée. Le système le plus connu actuellement fournissant une telle mémoire virtuellement partagée est le système TreadMarks [3]. Ce type de mémoire partagée distribuée se rencontre également dans les implantations distribuées de langages ou environnements de programmation parallèle initialement développés pour des architectures SMP. C'est le cas par exemple de l'implantation distribuée du langage Cilk [104] où encore des implantations distribuées d'OpenMP [87, 67].

L'approche mémoire partagée distribuée de niveau objet consiste à définir et accé-

der les données partagées au niveau des types de données utilisateurs. Ainsi, alors que sur une mémoire virtuellement partagée un accès à la mémoire consiste en une opération de lecture ou d'écriture sur un mot mémoire élémentaire, sur une telle mémoire partagée distribuée, un accès à la mémoire consiste en une opération définie par l'utilisateur sur un objet partagé qui encapsule une structure de données. Cette approche est appelée mémoire partagée distribuée de niveau objet (*object-based distributed shared memory*). Les langages ORCA [8], Jade [108] et Cid [95] sont basés sur ce type de mémoire partagée. LINDA [17] fournit également, par l'intermédiaire d'une mémoire associative (*Tuple Space*), un mécanisme de partage de données de niveau objet. Le système de mémoire partagée distribuée SAM [115] implante également un mécanisme de partage de données au niveau objet. Nous verrons également qu'Athapascan-1 offre une telle mémoire partagée.

L'intérêt d'une mémoire partagée distribuée de niveau objet par rapport à une mémoire virtuellement partagée réside dans sa capacité à être plus efficacement mise en œuvre comme le montrent les comparaisons réalisées entre ORCA et TreadMarks [9][1]. Ceci s'explique en partie par la granularité des données partagées et des accès à ces données qui conditionnent la quantité de messages qui seront générés sur le réseau pour maintenir la cohérence de cette mémoire partagée distribuée. Sur une mémoire partagée distribuée de niveau objet, cette granularité est explicitement définie par l'utilisateur lors de la création des objets partagés qui encapsulent les données partagées du programme. Par contre, sur une mémoire virtuellement partagée la granularité des données partagées est le mot mémoire élémentaire (bien que les mécanismes de pagination permettent de ramener cette granularité à la page lorsque les données du programmes sont bien structurées) et la granularité des accès à ces données partagées est la lecture ou l'écriture d'un mot mémoire élémentaire (bien que différents protocoles de cohérence relâchée, *i.e. lazy release consistency*, permettent à l'utilisateur de regrouper plusieurs accès en mémoire partagée et ainsi de regrouper les communications nécessaires au maintien de la cohérence de la mémoire). Ainsi, sur une mémoire virtuellement partagée la granularité des données partagées et des accès à ces données est fixe et indépendante de l'application alors que sur une mémoire partagée distribuée cette granularité est adaptée à l'application.

L'inconvénient d'une mémoire partagée distribuée de niveau objet est cependant de forcer la restructuration des programmes écrits pour des machines à mémoire partagée. Les données partagées de ces programmes doivent être explicitement encapsulées dans des objets partagés et les accès aux données partagées doivent être regroupés en opérations sur ces objets partagés. Il faut cependant noter que l'obtention de performances sur mémoire virtuellement partagée passe également par une restructuration similaire. Il est par exemple conseillé de regrouper sur une même page les données accédées simultanément pour réduire le nombre de pages communiquées. Ainsi l'inconvénient énoncé pour

1. Les difficultés à implanter efficacement Linda dans un environnement distribué sont principalement dues au mode d'adressage associatif des objets présents dans le *Tuple Space* et ne remet donc pas en cause le principe du partage des données au niveau objet.

l'utilisation d'une mémoire partagée distribuée n'en est pas vraiment un si les performances sont avant tout recherchées.

2.3.2 Abstraction des processeurs : parallélisme applicatif

Dans un modèle de programmation parallèle proche de la machine, le parallélisme de l'application est généralement exprimé sous la forme de processus explicitement créés sur une ressource (processeurs). C'est alors à la charge du programmeur d'organiser les calculs de son application pour s'adapter au parallélisme de la machine. Lorsque l'application est simple et régulière, cette opération peut être réalisée sans trop de problèmes, mais ce n'est plus le cas pour les applications irrégulières pour lesquelles l'obtention de performances passe par la mise en œuvre de mécanismes de régulation de charge adaptés.

L'abstraction des processeurs consiste alors à offrir au programmeur des mécanismes lui permettant d'exprimer le parallélisme de son application indépendamment du nombre de processeurs. Ces mécanismes permettent de créer dynamiquement des activités (encore appelées tâches de calcul du point de vue applicatif) sans préciser ni où ni quand ces activités seront exécutées. C'est alors le système d'exécution qui est chargé de réguler la charge de calcul entre les processeurs en contrôlant la localisation et l'entrelacement de ces activités et éventuellement en les migrant d'un processeur à l'autre.

Sur architecture SMP, les processus UNIX offrent déjà un tel mécanisme d'abstraction des processeurs. Cependant le coût important de leur gestion fait qu'ils ne sont pas utilisés pour exprimer le parallélisme à un grain fin. Les processus légers de la norme POSIX [75] permettent de s'affranchir de cette lourdeur. Cilk [11, 10, 104, 44], par différentes techniques (détaillées dans la section 2.4.1) permet d'alléger encore plus le surcoût de création d'activité parallèle[2] et donc d'exprimer le parallélisme à un grain encore plus fin. Sur ces systèmes ou langages pour architecture SMP, les communications entre les activités sont naturellement réalisées par lecture et écriture en mémoire partagée et par des mécanismes de synchronisation entre ces activités (par exemple les verrous, les variables de condition et le *join* pour les processus légers de la norme POSIX ou le `sync` du langage Cilk qui permet une synchronisation sur la fin de plusieurs activités).

Sur architecture distribuée, des mécanismes similaires de création dynamique d'activité permettent également une abstraction des processeurs. La communication entre les activités, est généralement obtenue par une mémoire partagée distribué du type de celles présentées dans la section précédente et par une extension au cadre distribué des mécanismes de synchronisations. Ainsi l'environnement de programmation parallèle MILLIPEDE [43] implémente une mémoire virtuellement partagée et offre divers mécanismes de synchronisation entre activités tel que les sémaphores. L'implantation distribuée de Cilk [104] implémente également une mémoire virtuellement partagée et offre un mé-

2. Le coût de création d'une activité dans la dernière version de Cilk est annoncé comme étant seulement six fois supérieur à un appel de fonction normale [44].

canisme de synchronisation sur l'attente de la terminaison d'un groupe d'activité. Le langage Cid [94, 95] implémente une DSM de niveau objet et offre un mécanisme de synchronisation proche de Cilk permettant l'attente de la fin de plusieurs activités.

L'ordonnancement de ces activités dans un environnement distribué est cependant plus complexe que sur architecture SMP et pose plusieurs problèmes : tout d'abord, pour réaliser cette régulation de charge, le système d'exécution doit être capable de créer des activités à distance et éventuellement de migrer les activités d'un espace d'adressage physique à un autre. Si sur architecture SMP les mécanismes de migration d'activité sont simples à mettre en œuvre (généralement réalisés par le système d'exploitation de la machine), il en va tout autrement sur les architectures distribuées, notamment lorsqu'elles sont hétérogènes. MILLIPEDE [43] offre de tels mécanismes de migration d'activité. L'implantation du langage Cilk fournit également des mécanismes de migration d'activité.

Un autre problème rencontré dans l'ordonnancement des activités dans un environnement distribué concerne la prise en compte de la localité des accès aux données partagées pour réduire les communications entre nœuds (*i.e.* les défauts de pages sur une mémoire virtuellement partagée ou les communications d'objets partagés sur une DSM de niveau objet). Différentes heuristiques peuvent être utilisées. Par exemple dans MILLIPEDE un historique des accès réalisés sur les pages de la mémoire virtuellement partagée par chacune des activités est maintenu [116]. Ces informations sont exploitées lors des phases de régulation de charge pour migrer en priorité les activités ayant peu accédé à des pages locales.

2.3.3 Description implicite du parallélisme applicatif

Dans la section précédente, nous avons présenté des langages ou environnements de programmation permettant de décrire explicitement le parallélisme de l'application par création dynamique d'activité (ou création de tâche lorsque l'on se place du point de vue de l'application) et par synchronisation explicite de ces activités.

Une autre méthode de programmation parallèle consiste à laisser à un compilateur et/ou à un système d'exécution le soin de découvrir le parallélisme de l'application. On parle alors de parallélisme implicite. Cette méthode s'applique aussi bien à des langages sans séquencement comme les langages fonctionnels ou logiques qu'aux langages impératifs classiques tel que Fortran ou C. La détection du parallélisme de l'application peut être réalisée par un compilateur paralléliseur à partir du code de l'application mais elle peut également être réalisée dynamiquement en cours d'exécution (c'est le cas pour beaucoup de langages fonctionnels ou logiques). Dans les deux cas la détection du parallélisme est obtenue par analyse des dépendances de données du programme.

Les caractéristiques des langages dans lesquels aucun ordre d'exécution des instructions n'est spécifié, permettent d'en extraire naturellement les dépendances de données. Ainsi, parmi les langages fonctionnels Sisal [13], Multilisp [61] et Haskell [71, 121] offrent des compilateurs ou des supports exécutifs permettant l'exécution parallèle.

Nous nous intéresserons principalement dans la suite à l'extraction du parallélisme des langages séquentiels.

2.3.3.1 Extraction du parallélisme à la compilation

La parallélisation automatique d'un code séquentiel est un problème difficile et non encore complètement résolu à ce jour [40]. Cette technique est actuellement surtout adaptée à la parallélisation des nids de boucles. En effet ces nids de boucles permettent de représenter de manière compacte une quantité importante de parallélisme potentiel, l'analyse en est donc facilitée.

Des directives de compilation peuvent également être rajoutées dans le programme source pour faciliter l'extraction du parallélisme ainsi que pour aider au placement des calculs et des données sur les processeurs. Cette approche est par exemple utilisée dans HPF [65] dans lequel des directives de partitionnement et de placement des tableaux sont fournies par le programmeur conduisant à un placement des calculs sur les processeurs.

Finalement, une limitation importante des compilateurs paralléliseurs est qu'il n'y a pas toujours suffisamment d'information lors de la compilation pour générer un code parallèle. Par exemple lorsque le parallélisme de l'application dépend des données en entrée, le compilateur n'est pas capable de paralléliser le programme.

2.3.3.2 Extraction du parallélisme à l'exécution

Contrairement aux techniques de parallélisation à la compilation qui analysent les sources du programme pour extraire le parallélisme, les techniques présentées maintenant effectuent cette analyse à l'exécution. Elles permettent ainsi d'extraire le parallélisme du programme qui dépend des données en entrée du programme (par exemple la factorisation creuse de Cholesky).

Cependant cette analyse à l'exécution va naturellement engendrer un surcoût lors de l'exécution. Dans le but de réduire ce surcoût, l'analyse n'est pas réalisée sur des instructions élémentaires comme dans les compilateurs paralléliseurs mais sur des tâches qui correspondent à des séquences d'instructions du programme. L'exécution du programme est alors explicitement décomposée en tâches de calcul par le programmeur. L'analyse à l'exécution consiste alors essentiellement à extraire le parallélisme présent entre ces tâches.

Plusieurs langages ou environnements de programmation ont ainsi été proposés, qui permettent d'extraire à l'exécution le parallélisme présent au niveau des tâches. C'est le cas par exemple de Jade [108, 107] et de Mentat [59]. Athapascan-1 entre également dans ce cadre.

Dans ces langages, les tâches sont explicitement définies et créées par le programmeur. Ainsi, dans Jade, une tâche correspond à un bloc d'instructions inséré dans une construction spécifique du langage (pour le détail, voir la section 2.4.2 page 34). Une

tâche est alors créée à chaque fois que le flot d'exécution rencontre ce bloc. Dans le langage Mentat les tâches correspondent à l'exécution de certaines fonctions identifiées dans le programme grâce à une construction du langage[3]. Les tâches sont alors créées lors de l'appel de l'une de ces fonctions. En Athapascan-1, les tâches correspondent à l'exécution de procédures et sont explicitement créées en ajoutant un mot clé devant l'appel de la procédure. Notons également que pour ces trois langages les tâches peuvent être créées récursivement.

L'extraction du parallélisme entre les tâches nécessite d'analyser les dépendances de données entre les tâches. Plusieurs solutions ont été proposées pour permettre cette analyse des dépendances de données. Dans Jade et Athapascan-1 des mécanismes permettent d'indiquer au système d'exécution, lors de la création d'une tâche, l'ensemble des effets de bords pouvant éventuellement être réalisés par celle-ci. Dans Mentat les tâches sont soit des fonctions pures donc sans effet de bord soit des fonctions qui réalisent des effets de bord sur un objet bien identifié[4].

Grâce à cette analyse des dépendances de données, le système peut maintenir pendant l'exécution un graphe de flot de données qui contient les tâches à exécuter ainsi que les dépendances de données entre ces tâches. Le système d'exécution repose alors sur des mécanismes d'abstraction de la machine parallèle similaires à ceux présentés dans les deux sections précédentes pour exécuter ces tâches tout en respectant les contraintes de précédences induites par les dépendances de données. Ainsi dans Jade, Mentat et Athapascan-1 une mémoire partagée distribuée de niveau objet est proposée et les effets de bord des tâches ne peuvent être réalisés que sur des objets présents dans cette mémoire partagée. De même pour l'exécution des tâches, des mécanismes de régulation de charge sont mis en œuvre.

En conclusion, l'intérêt de cette approche est essentiellement de faciliter la programmation parallèle puisque le programmeur ne gère pas explicitement les synchronisations entre les tâches (il n'y a plus de problème liés à la cohérence des accès en mémoire partagée ou à l'indéterminisme de l'exécution). Cependant un autre intérêt de cette approche est d'exhiber, à l'exécution, une représentation fine de l'exécution future sous la forme du graphe de flot de données (la précision ou granularité de cette représentation est contrôlée par le programmeur). Pour certaines applications, dont le parallélisme peut dépendre des données en entrée, il est même possible de dérouler en début d'exécution l'ensemble du graphe de flot de données de l'exécution à venir : c'est le cas par exemple de la factorisation numérique creuse de Cholesky présentée dans les chapitres 8 et 9 et dont le parallélisme dépend fortement de la structure de la matrice creuse. L'intérêt de disposer de ce graphe est essentiel tant du point de vue de l'ordonnancement de l'application (des techniques d'ordonnancement statiques de ce graphe sont détaillées dans le chapitre 5 et

3. Plus exactement Mentat est une extension de C++ possédant trois « types » de classes étendant les classes du langage C++. Les tâches correspondent alors aux fonctions membres de ces classes et sont créées lors de l'appel de l'une de ces fonctions membres.

4. C'est en fait l'objet C++ sur lequel la fonction membre est appelée.

peuvent être mises en œuvre) que du point de vue de la gestion de la mémoire partagée distribuée. Notons que les langages qui supportent un modèle de programmation parallèle explicite comme Cilk ou Cid n'offrent que des mécanismes simples de synchronisation qui ne permettent pas une description aussi précise du futur de l'exécution. Ainsi, dans Cilk et Cid seul un mécanisme de synchronisation sur la fin d'activité est offert. La représentation, à un instant donnée, du futur de l'exécution est alors restreinte à un graphe *join* (*i.e.* un arbre inverse) représentant les synchronisations entre les activités.

2.4 Exemples de langages de programmation parallèle haut niveau

Nous présentons dans ce chapitre plus en détail deux des langages de programmation parallèle haut niveau précédemment cités, Cilk et Jade.

Pour chacun de ces langages nous présentons tout d'abord le modèle de programmation utilisé permettant d'abstraire les caractéristiques de la machine sous-jacente. Nous présentons ensuite l'implantation de ce modèle de programmation sur les architectures SMP et distribuées en insistant plus particulièrement sur l'algorithme d'ordonnancement utilisé. Nous présenterons finalement le modèle de coût associé lorsque qu'il existe et les évaluations expérimentales réalisées.

2.4.1 Cilk

Cilk[11, 104] est un langage destiné à la programmation des architectures SMP (une version distribuée existe, basée sur une mémoire virtuellement partagée) dont le développement à débuté en 1993 au MIT ; il est actuellement disponible dans sa cinquième version.

2.4.1.1 Modèle de programmation

Le langage Cilk est une extension du langage C qui offre des primitives pour l'expression du parallélisme de contrôle par création explicite de tâches.

La description du parallélisme se fait à l'aide du mot clé `spawn` placé devant un appel de fonction. 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 sa terminaison pour continuer. Cette exécution étant asynchrone, la procédure créatrice ne peut utiliser le résultat de la fonction appelée qu'avec une synchronisation 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 (et sa descendance) 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 et leurs descendances, jusqu'à rencontrer l'instruction `sync`). 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.

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.

2.4.1.2 Implantation

La base de l'implantation du langage est un algorithme d'ordonnancement de type liste, appelé *work-stealing*, basé sur le « 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. Plusieurs techniques d'optimisation [44] permettent de placer le maximum du surcoût d'ordonnancement lors des vols et d'avoir ainsi un coût de création de tâche (`spawn`) réduit au maximum. Parmi ces techniques, une création paresseuse des tâches [91] permet de ne créer effectivement en mémoire les tâches que lorsqu'elles sont volées. Un protocole d'exclusion mutuelle entre les processeurs permet une prise de verrou uniquement lors d'un vol de tâche [30, 44].

Une version distribuée de Cilk pour les réseaux de SMP est proposée dans [104]. Elle est basée sur une mémoire virtuellement partagée implantée par un mécanisme de pagination. Dans cette version, le principe du vol de travail a été maintenu, ainsi que le déterminisme de l'exécution. Cependant pour réduire les communications entre différents nœuds SMP, l'ordonnancement prend en considération une notion de «localité». En effet, pour effectuer le vol, le choix du processeur victime n'est pas totalement aléatoire, les processeurs sur le même nœud SMP qui partagent la même mémoire physique que le voleur ont une plus grande probabilité d'être choisis que tout autre processeur distant.

2.4.1.3 Évaluations théoriques et expérimentales

Au modèle de programmation Cilk est associé un modèle de coût. 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.

– S_1, la consommation mémoire de l'exécution sur un seul processeur lors d'une exécution en profondeur du graphe de tâches.

L'implantation de ce modèle de programmation garantit que la durée d'exécution T_p d'un programme sur une machine à p processeurs et que la consommation mémoire S_p seront telles que :

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

Plusieurs applications ont été portées et expérimentées sur la version SMP de Cilk. Les expérimentations réalisées [104] donnent de très bonnes accélérations, comprises entre 5 et 8 sur une architecture SMP à 8 processeurs, pour toutes les applications testées. Nous verrons cependant dans le chapitre 9.1.1 que l'implantation proposée pour la factorisation de Cholesky creuse est très inefficace en terme de puissance de calcul (au moins d'un facteur 50) par rapport à notre implantation séquentielle.

Sur l'implantation distribuée de Cilk, seule une évaluation d'un programme calculant récursivement un terme de la suite de Fibonacci est proposée. L'accélération obtenue sur une machine constituée de trois triprocesseurs SMP est de 5.8. Cependant ce programme utilise très peu la mémoire virtuellement partagée et ne permet donc pas de bien évaluer l'implantation distribuée de Cilk.

2.4.1.4 Bilan

Le langage Cilk offre de manière simple à l'utilisateur le moyen d'exprimer le parallélisme de contrôle de son application. L'utilisateur doit cependant contrôler explicitement les synchronisations entre les tâches qui sont nécessaires au maintien de la cohérence des données partagées.

Les performances obtenues sur architectures SMP permettent d'exploiter efficacement le parallélisme de l'applications à un grain fin. Cependant sur les architectures distribuées, le peu de résultats expérimentaux proposés laisse à penser que les performances ne sont pas à la hauteur de celles obtenues sur architecture SMP.

2.4.2 Jade

Jade [107, 108] est un langage pour la programmation parallèle sur machine à mémoire partagée et distribuée développé à l'Université de Standford ; la dernière version de Jade date de 1994.

2.4.2.1 Modèle de programmation

Un programme Jade est un programme C annoté par des instructions Jade définissant des objets partagés, des tâches, ainsi que les accès réalisés par les tâches sur les objets partagés. Les annotations ne modifient pas (strictement parlant) la sémantique du programme C. En ce sens, Jade est un modèle de programmation implicite.

Le modèle de programmation de Jade est basé sur trois concepts : des objets partagés (*i.e.* une mémoire partagée distribuée de niveau objet), des tâches et des mécanismes permettant de spécifier les accès réalisés par les tâches sur les données partagées (*i.e.* les effets de bords des tâches sur la mémoire partagée distribuée). Les objets partagés et les tâches permettent au programmeur de spécifier respectivement la granularité des données et la granularité des calculs. La spécification des accès réalisés par les tâches sur les données partagées permet au système d'exécution, par analyse des dépendances de données entre tâches, d'extraire le parallélisme entre ces tâches.

La création d'un objet partagé est effectuée de deux manières différentes : création statique au démarrage du programme par l'ajout du mot clé **shared** dans une déclaration d'une variable globale C ; création dynamique en cours d'exécution par appel de la construction **create_objet** avec en paramètre le type de l'objet partagé à créer. La destruction d'un objet partagé est effectuée par l'appel de la fonction **destroy_objet** avec en paramètre un pointeur sur un objet partagé.

La définition d'une tâche est effectuée par insertion d'un bloc d'instructions dans la construction **withonly** :

```
1: int fonction_C( ... )
2:     ...
3:     withonly {[list-spécifications-accès]} do ([list-paramètres]) {
4:         [corps-de-la-tâche]
5:     }
6:     ...
7: }
```

La liste des paramètres contient les données qui seront copiées dans le contexte de la tâche lors de sa création. La liste de spécification des accès contient la liste des objets partagés accédés par la tâche lors de son exécution, chaque objet `p` de cette liste étant annoté par le type d'accès réalisé. Les types d'accès de bases possibles sont les suivants : la lecture **rd**`(p)`, l'écriture **wr**`(p)`, la modification **rd_wr**`(p)`, l'accumulation **cm**`(p)` et la destruction **de**`(p)`.

2.4.2.2 Implantation

Une tâche est créée lors de chaque exécution d'une instruction **withonly**. L'extraction du parallélisme du programme (*i.e.* le parallélisme entre les tâches) est réalisée à l'exécution par analyse des dépendances de données entre tâches lors de leurs créations. Cette

analyse des dépendances est rendue possible grâce à la spécification des accès aux objets partagés réalisés par les tâches. Cette analyse des dépendances conduit alors à la génération d'un graphe de flot de données contenant les tâches à exécuter et les synchronisations nécessaires entre ces tâches pour respecter la sémantique séquentielle du programme.

L'ordonnancement des tâches ainsi créées est réalisé dynamiquement de manière centralisée. Notons que cet ordonnancement est optimisé pour le cas où la tâche racine `main` est la seule à créer d'autres tâches. L'algorithme utilisé sur machine à mémoire distribuée est un algorithme de liste prenant en compte des critères de localité. Plus exactement chaque tâche est associée au premier objet partagé qu'elle déclare accéder dans sa liste de spécifications des accès et chaque objet partagé est associé au processeur sur lequel la dernière modification de l'objet a été réalisée. Ainsi à chaque tâche est associé un processeur qui contient une version valide du premier objet partagé accédé. L'algorithme d'ordonnancement essaye alors d'exécuter chaque tâche sur ce processeur en évitant ainsi la communication de cet objet.

2.4.2.3 Évaluations théoriques et expérimentales

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 et similaire à celle donnée dans cette thèse pour Athapascan-1 (chapitres 4 et 5).

Différentes applications ont été codées en Jade [107]. Nous détaillerons dans la section 9.1.2 page 155 une implantation de la factorisation de Cholesky creuse. Trois de ces applications ont un parallélisme à gros grain avec une granularité moyenne des tâches comprise entre 5 et 42 secondes (rapportée à la puissance de crête d'un processeur de la machine utilisée, cela correspond à une granularité des tâches comprise entre 150 Mflop à 1260 Mflop). Deux autres applications, dont la factorisation creuse de Cholesky, ont un parallélisme à grain fin avec une granularité moyenne des tâches comprise entre 20 et 30 millisecondes (de la même manière, cela correspond à une granularité des tâches comprise entre 0.6 Mflop à 0.9 Mflop). Les résultats obtenus sont relativement bons sur machine à mémoire partagée (DASH) avec des accélérations allant de 9 à 27 sur 32 processeurs et sur différentes applications[5]. Cependant, sur machine à mémoire distribuée (iPSC/860), aucune accélération n'est obtenue sur les applications à grain fin.

2.4.2.4 Bilan

Jade permet une programmation simple des applications parallèles en offrant un modèle de programmation parallèle implicite basé sur un langage de programmation connu. Cependant, compte tenu des choix effectués lors de l'implantation actuelle où la plupart

5. Parmi celles-ci, les applications *Wather*, *Volume Rendering* et *Ocean* de la batterie de teste SPLASH [118] ainsi qu'une factorisation creuse de Cholesky avec partitionnement monodimensionnel

des algorithmes mis en œuvre sont centralisés, Jade est plutôt destiné aux applications de gros grain.

2.5 Conclusion

Nous avons présenté dans ce chapitre des langages de programmation parallèle de haut niveau qui utilisent différents paradigmes de programmation afin d'abstraire la machine parallèle sous-jacente, éventuellement avec des garanties de performances, dans le but de faciliter la programmation d'application parallèle et d'augmenter la portabilité de ces applications.

Le premier de ces paradigmes consiste à fournir une mémoire partagée distribuée qui libère le programmeur du placement des données sur les différents nœuds de la mémoire distribuée et de la communication de ces données entre nœuds. Ce paradigme permet ainsi d'abstraire le réseau de communication. Le deuxième paradigme consiste à fournir des mécanismes de création dynamique d'activités (de tâches) pour décrire le parallélisme applicatif, le placement et l'entrelacement de celles-ci étant à la charge du système d'exécution. Ce paradigme permet ainsi d'abstraire les processeurs de la machine parallèle.

Nous avons également présenté les modèles de programmations parallèles implicites et notamment ceux dont le parallélisme de l'application est découvert dynamiquement pendant l'exécution. Ces derniers permettent l'extraction du parallélisme dépendant des données en entrée du programme et offrent en cours d'exécution une représentation fine du futur de l'exécution sous la forme d'un graphe de flot de données.

La première partie de ce document sera consacrée à Athapascan-1 qui offre et utilise de tels mécanismes d'abstraction de la machine parallèle, une décomposition explicite de l'application en tâches de calcul et une description implicite du parallélisme entre ces tâches. Par ailleurs Athapascan-1 offre des garanties théoriques de performances par l'intermédiaire d'un modèle de coût qui sera développé dans les chapitres 4 et 5.

I

Athapascan-1

3

Athapascan-1 : modèle de programmation

3.1 Présentation du modèle de programmation

Athapascan-1 supporte un modèle de programmation parallèle haut niveau, au sens défini dans le chapitre précédent. Un programme Athapascan-1 se compose de tâches explicitement créés en cours d'exécution, ainsi que d'une mémoire partagée distribuée contenant des objets créés et accédés par les tâches. En ce sens, Athapascan-1 permet d'abstraire la machine parallèle. Les synchronisations entre les tâches sont cependant définies implicitement de manière à respecter (en partie) la sémantique séquentielle du langage impératif sur lequel le modèle est construit (C ou C++). Pour permettre au système chargé de l'exécution de déterminer ces synchronisations, les tâches déclarent l'ensemble des accès qu'elles réaliserons sur les objets en mémoire partagée. Il est alors possible d'analyser les dépendances de données entre les tâches et ainsi de déterminer les synchronisations nécessaires au respect de la sémantique du langage. Athapascan-1 est donc un modèle de programmation parallèle implicite puisque le parallélisme présent entre les tâches n'est pas explicitement fournie par le programmeur.

3.1.1 Tâches et communications par objets partagés

Le modèle de programmation d'Athapascan-1 est une extension du modèle de programmation procédural. Dans ce dernier modèle un programme est une séquence ordonnée d'instructions et d'appels de procédure qui est évaluée séquentiellement, les instructions interagissant et communiquant entre elles par l'intermédiaire de variables. Les lan-

gages de programmation C, Pascal ou FORTRAN sont des exemples de langage utilisant le modèle de programmation procédural.

Pour exprimer la concurrence, ce modèle de programmation procédural est étendu par le concept de tâche. Une tâche diffère d'une procédure uniquement par la manière dont elle est appelée. Tandis que l'appel d'une procédure est synchrone, c'est-à-dire que l'appelant est bloqué jusqu'à la terminaison de la procédure appelée, la création d'une tâche est non bloquante, l'appelant pouvant continuer son exécution immédiatement après la création sans attendre la fin de l'exécution de la tâche.

Les tâches communiquent entre elles soit lors de l'appel, une tâche recevant des paramètres de la tâche qui l'a invoquée, soit par l'intermédiaire d'une mémoire partagée distribuée de niveau objet.

Le support pour partager des données entre tâches est alors l'objet partagé qui est une instance d'un type de données abstrait. Cet objet partagé peut être n'importe quelle structure de données comme par exemple une zone contiguë en mémoire, une liste chaînée, un arbre ou même une matrice (l'objet contient alors le nombre de ligne et de colonne de la matrice ainsi que le tableau des éléments de la matrice). De plus, ces objets sont alloués dynamiquement et leur structure peut évoluer au fil des modifications qui leurs sont apportées.

3.1.2 Sémantique des accès aux objets partagés

Quatre types d'accès peuvent être réalisés sur un objet partagé :

- Accès en lecture de l'objet.
- Accès en modification de l'objet, l'objet peut à la fois être lu et modifié.
- Écriture par affectation d'une nouvelle valeur sur l'objet.
- Écriture par accumulation d'une valeur sur l'objet. Cet accès en accumulation est caractérisé par la fonction utilisée pour réaliser l'accumulation, cette fonction étant définie par l'utilisateur.

Dans la suite, nous définissons l'exécution séquentielle d'un programme Athapascan-1, comme l'exécution obtenue en remplaçant chaque création de tâche par l'exécution de la procédure associée à la tâche. Cet exécution séquentielle définit alors naturellement un ordre total (appelé «ordre séquentiel») des accès réalisés sur les objets partagés.

La sémantique des accès aux objets partagés est alors telle que tout accès en lecture ou modification d'un objet partagé voit la dernière modification, affectation ou accumulation réalisée sur cet objet relativement à l'ordre séquentiel des accès aux objets partagés.

Par ailleurs le système chargé de l'exécution prend comme hypothèse que les fonctions utilisées pour réaliser les accès en accumulation sont des fonctions associatives et

commutatives. Donc si a_1 et a_2 sont deux accès sur un même objet partagé, consécutifs relativement à l'ordre séquentiel des accès réalisés sur cet objet et si ces deux accès sont des accumulations effectuées avec la même fonction d'accumulation alors aucune hypothèse ne peut être fait sur l'ordre dans lequel ces accumulations seront appliquées sur l'objet partagé lors d'une exécution parallèle.

Cette sémantique des accès aux objets partagés sera appelée, par abus de langage, « sémantique séquentielle ». Il est à noter que l'accès en accumulation fait qu'Athapascan-1 ne peut être considéré comme un langage ayant une sémantique séquentiel, malgré cette appellation, puisque l'ordre séquentiel des accès à une même donnée partagée peut ne pas être respecté dans certains cas.

Caractéristique 1 *Les synchronisations entre les tâches sont définies implicitement de manière à respecter la sémantique définie sur les accès aux objets partagés. Le modèle de programmation ne fournit aucun des mécanismes classiques de synchronisation entre tâches que l'on retrouve dans d'autres modèles de programmations parallèles comme le* join*, les variables de condition et les sections critiques des processus légers de la norme POSIX [75] ou le sync de Cilk [11].*

Le choix d'une sémantique séquentielle permet de simplifier grandement la programmation d'une application parallèle. En effet, la plupart des difficultés rencontrées lors de l'utilisation d'un modèle de programmation parallèle explicite, comme par exemple l'indéterminisme de l'exécution, la gestion des accès conflictuels à une même donnée et la gestion des synchronisations sont éliminés dans un modèle utilisant une sémantique séquentielle. Il faut cependant noter que cette simplification est obtenue en contrepartie d'une certaine diminution de l'expressivité du parallélisme. Le système doit de plus extraire automatiquement le parallélisme de l'application. Nous verrons dans la suite de ce chapitre, ainsi que dans le chapitre suivant, comment ceci peut être réalisée à un faible coût.

3.1.3 Référence contrainte sur un objet partagé : droit d'accès

Pour déterminer automatiquement, pendant l'exécution, les synchronisations entre les tâches nécessaires au respect de la sémantique séquentielle, il est nécessaire de rajouter des information dans le programme et d'imposer certaines règles sur l'utilisation des objets partagés. Ces informations et règles d'utilisations permettront alors au système chargé de l'exécution de connaître, lors de la création d'une tâche, l'ensemble des effets de bord sur les objets partagés pouvant être réalisé par cette tâche (en incluant son éventuelle descendance). À partir de cette information le système pourra alors analyser les dépendances de données entre les tâches et ainsi déterminer les synchronisations entre les tâches qui sont nécessaires au respect de la sémantique séquentielle.

Dans le modèle de programmation d'Athapascan-1, les objets partagés sont accédés uniquement par l'intermédiaire de références contraintes. Une référence contrainte impose des restrictions sur les accès pouvant être réalisés sur l'objet référencé. Quatre types différents de restrictions d'accès, encore appelées droits d'accès, peuvent être associés à une référence contrainte :

- Lecture-écriture : droit d'accès en modification sur l'objet (donc aucune contrainte d'accès).
- lecture : droit d'accès en lecture seule sur l'objet. Seule une lecture de l'objet référencé est autorisée.
- Écriture : droit d'accès en écriture seule sur l'objet. Seule une affectation d'une nouvelle valeur sur l'objet référencé est autorisée.
- Accumulation : droit d'accès en accumulation sur l'objet. Seule une opération d'accumulation sur l'objet référencé est autorisée. La fonction d'accumulation est associée à la référence.

Un deuxième niveau d'information peut être ajouté sur la référence contrainte, précisant si l'objet référencé sera effectivement accédé par la tâche ou uniquement par son éventuelle descendance. Dans le second cas, la référence contrainte est dite différée (*postponed*) et aucun accès n'est autorisé sur l'objet référence.

Par ailleurs, deux règles sur l'utilisation des ces références contraintes sont imposées par le modèle de programmation :

- Une tâche ne peut obtenir une référence contrainte sur un objet partagé que de deux manières : soit cette référence est un paramètre de la tâche, soit elle est obtenue lors de la création d'un nouvel objet partagé par cette tâche. Toute autre manière de récupérer une référence sur un objet partagé est interdite : par exemple insérer cette référence dans un objet partagé où dans un paramètre passé par valeur est interdit.
- Les droits d'accès sur une référence contrainte ne peuvent être modifiés que lors du passage de cette référence en paramètre d'une tâche. De plus cette modification ne peut aller que dans le sens d'une diminution des droits d'accès accordés sur l'objet. Le tableau 3.1 précise les modifications des droit d'accès qui sont autorisées lors de la création d'une tâche.

Caractéristique 2 *Grâce à ces règles d'utilisation des objets partagés, l'analyse du prototype d'une tâche permet de déterminer l'ensemble des accès pouvant être réalisés par cette tâche (et par sa descendance éventuelle) sur les objets partagés qui lui sont passés en paramètres. Les synchronisations entre les tâches, nécessaires pour respecter la sémantique séquentielle, peuvent alors être calculées lors de l'exécution.*

droit d'accès de la tâche mère	droit d'accès autorisée de la tâche fille
lecture-écriture [différée]	lecture-écriture [différée] lecture [différée] écriture [différée] accumulation [différée]
lecture [différée]	lecture [différée]
écriture [différée]	écriture [différée]
accumulation [différée]	accumulation [différée]

Tableau 3.1 – *Modification autorisée des droits d'accès d'une référence contrainte.*

Nous montrerons, dans le chapitre suivant 4.3.4 page 64, que la complexité du calcul de ces synchronisations est bornée par la somme du nombre de tâches créées et du nombre de références contraintes pris en paramètre par ces tâches.

Le principe de fournir des informations sur la manière dont les données seront utilisées par le programme, dans le but de faciliter l'extraction du parallélisme de l'application n'est pas nouveau. Ce principe se retrouve par exemple dans le langage Jade bien qu'il diffère d'Athapascan-1 dans la manière de définir les tâches.

3.1.4 Synchronisations entre tâches créatrices et tâches créées

Lorsqu'une tâche se synchronise en cours d'exécution, par exemple sur l'attente d'un résultat produit par une autre tâche, le système chargé de l'exécution doit être capable, pour assurer un ordonnancement efficace des tâches, de reprendre l'exécution de cette tâche sur un autre processeur. Ce mécanisme de migration d'une tâche en cours d'exécution est complexe, notamment sur les architectures hétérogènes. Par ailleurs, ce coût de migration n'est pas prévisible par le programmeur : il dépend du coût de fermeture du contexte (migration de la pile) et donc est *a priori* difficilement prévisible dans le programme. Ceci limite alors le développement de modèle de coût avec communication. Nous avons donc choisi de supprimer les synchronisations pouvant apparaître en cours d'exécution d'une tâche.

Pour supprimer ces éventuelles synchronisations, la restriction suivante sur l'utilisation des références contraintes est rajoutée :

– Une référence contrainte de type lecture-écriture ne peut pas être passée en paramètre d'une autre tâche.

Il est alors clair qu'il ne peut y avoir de synchronisation entre une tâche mère et une tâche fille puisque, pour tout objet partagé accessible à la fois par la tâche mère et la tâche fille, soit la tâche mère n'a pas de droit de lecture sur cet objet soit la tâche fille n'a pas de

droit d'écriture sur cet objet. Nous verrons dans la section 3.3.1 page 51 comment cette restriction peut être contournée du point de vue de l'utilisateur.

Caractéristique 3 *Une tâche s'exécute jusqu'à la fin de son exécution sans synchronisation. La création de tâche est non bloquante et une tâche créatrice ne peut attendre les résultats d'une tâche créée par elle.*

Le tableau 3.2 précise les modifications des droits d'accès permis lors de la création d'une tâche, après l'ajout de cette dernière règle dans le modèle de programmation d'Athapascan-1.

droit d'accès de la tâche mère	droit d'accès autorisée de la tâche fille
lecture-écriture	
lecture-écriture différée	lecture-écriture [différée] lecture [différée] écriture [différée] accumulation [différée]
lecture [différée]	lecture [différée]
écriture [différée]	écriture [différée]
accumulation [différée]	accumulation [différée]

Tableau 3.2 – *Modification autorisée des droits d'accès d'une référence contrainte permettant de supprimer les synchronisations en cours d'exécution d'une tâche.*

3.1.5 Ramasse-miettes sur les objets partagés

De part le caractère non bloquant des créations de tâche, la gestion de la destruction des objets partagés ne peut être laissée à la charge de l'utilisateur. En effet la durée de vie de la tâche qui a créé un objet partagé est généralement plus courte que la durée de vie de cet objet. La destruction de l'objet ne peut donc pas être réalisée par la tâche qui a créé l'objet. Pour cette raison, le modèle de programmation d'Athapascan-1 fournit un ramasse-miettes pour la destruction des objets partagés. Un objet partagé est donc détruit lorsque plus aucune référence contrainte sur cet objet n'existe dans le système.

3.1.6 Informations pour le système d'exécution

Le système d'exécution peut être aidé pour réaliser l'ordonnancement du programme si des informations supplémentaires lui sont fournies. Ces informations peuvent être par exemple une estimation du coût d'exécution des tâches, une estimation de la taille des objets partagés ou encore une estimation du volume maximum des données locales nécessaires à l'exécution d'une tâche.

Le programme peut également indiquer une politique d'ordonnancement préférentielle à utiliser sur l'ensemble ou une partie du programme suivant le type de l'application.

Nous verrons dans le chapitre 5 comment ces informations peuvent être exploitées efficacement lors de l'exécution du programme.

3.2 Exemples

Pour illustrer le modèle de programmation, nous présentons deux exemples didactiques de programmes écrits en Athapascan-1 : un algorithme de calcul récursif du $n^{ème}$ terme de la suite de Fibonacci ainsi qu'un algorithme d'élimination de Gauss par colonne.

La syntaxe complète et exacte qui est utilisée par la bibliothèque C++ Athapascan-1 implantant ce modèle de programmation sera présentée dans le chapitre 6. Cependant, cette syntaxe utilise des constructions du langage C++ (classe, patrons), et nécessite une certaine connaissance du langage C++. Pour faciliter la présentation des exemples, nous utiliserons donc ici une syntaxe simplifiée basée sur une extension du langage C par les mots clés suivant :

- Le mot clé **Fork** placé devant un appel de fonction indique une création de tâche.
- Les mots clés **Shared_r_w**, **Shared_r**, **Shared_w** et **Shared_cw(**[fonction-accumulation]) placés devant un nom de type quelconque `T` indiquent une référence contrainte sur un objet partagé de type `T`. Le type d'accès possible à la référence contrainte est donc précisé dans le suffixe du mot **Shared** :
  - **r_w** pour un accès en lecture-écriture.
  - **r** pour un accès en lecture.
  - **w** pour un accès en écriture.
  - **cw(**[fonction-accumulation]**)** pour un accès en accumulation avec la fonction [fonction-accumulation].

  Par ailleurs les accès différés correspondants sont précisés par les suffixes **rp_wp**, **rp**, **wp** et **cwp**.
- L'accès à l'objet partagé contenu dans une référence contrainte est implicite. Pour le programmeur, la référence s'utilise donc comme si elle était l'objet. Cependant, seul les accès autorisés par la référence contrainte sont permis :
  - Aucune modification ne peut être réalisée sur un objet référencé avec un droit d'accès en lecture.
  - Seule une affectation d'une nouvelle valeur peut être réalisée sur un objet référencé avec un droit d'accès en écriture.

– Seule une accumulation d'une valeur (par l'intermédiaire de la fonction d'accumulation associée à la référence) peut être réalisée sur un objet référencé avec un droit d'accès en accumulation.

3.2.1 Fibonacci

La figure 3.1 page 48 montre un programme calculant récursivement le $n^{ème}$ terme de la suite de Fibonacci.

```
1:  void add( Shared_cw(add) int res, int b )  {
2:     res += b;
3:  }
4:  void fibo( int n, Shared_cw(add) int res ) {
5:     if( n < 2 )
6:        add( res, n );
7:     else {
8:        Fork fibo( n-1, res );
9:        Fork fibo( n-2, res );
10:    }
11: }
12: void print_int( Shared_r int i ) {
13:    printf("%d",i);
14: }
15: main() {
16:    Shared_rp_wp int res(0);
17:    Fork fibo( 20, res );
18:    Fork print_int( res );
19: }
```

Figure 3.1 – *Calcul récursif du $n^{ème}$ terme de la suite de Fibonacci, écrit dans le modèle de programmation Athapascan-1*

La ligne 4 déclare et définit la fonction (*i.e.* le type de tâche) `fibo` qui prend un paramètre par valeur (le terme de la suite à calculer) ainsi qu'une référence `res` sur un objet partagé ayant un droit d'accès en accumulation avec la fonction `add` (là où stocker le résultat). Les seuls accès possibles à l'objet référencé par la référence contrainte `res` sont des accumulations réalisées avec la fonction `add`. Une accumulation de ce type est par exemple réalisée ligne 6. Cette fonction crée également d'autre tâche exécutant la fonction `fibo` (ligne 8 et 9).

À la ligne 16, un objet partagé référencé par `res` est créé et initialisé à zéro. Notons que les droits d'accès permis sur cet objets sont différés (**Shared_rp_wp**), c'est-à-dire

que la fonction `main` (*i.e.* la tâche racine du programme) ne peut pas accéder directement en lecture ou écriture à cet objet. Seules les tâches filles pourrons réaliser des accès sur cet objet.

À la ligne 17 la tâche calculant ce $n^{ème}$ terme est invoquée avec un paramètre par valeur (ici la constante `20`) et la référence `res` sur l'objet partagé. Pour afficher le résultat, il est ensuite nécessaire de créer une tâche prenant en paramètre `res` (ligne 18) puisque la tâche `main` n'a pas le droit de lecture sur l'objet partagé référencé par `res`. Ainsi la tâche `main` peut s'exécuter sans synchronisation. Par contre la tâche `print_int` ne pourra elle s'exécuter que lorsque toutes les tâches `fibo` modifiant l'objet référencé par `res` seront terminées.

3.2.2 Élimination de Gauss par colonnes

L'élimination de Gauss par colonnes, est un algorithme de base de l'algèbre linéaire. L'algorithme 1 page 49 présente la forme séquentielle et parallèle de l'élimination de Gauss par colonne sans pivot[1].

Algorithme 1 : Élimination de Gauss par colonnes.

pour $k = 1$ **jusqu'à** $n - 1$ **faire**
  pour $i = k + 1$ **jusqu'à** n **faire**
    $A(i,k) = A(i,k)/A(k,k)$
  pour $j = k + 1$ **jusqu'à** n **faire [en parallèle]**
    pour $i = k + 1$ **jusqu'à** n **faire**
      $A(i,j) = A(i,j) - A(i,k) * A(k,j)$

La figure 3.2 montre l'écriture de cet algorithme dans le modèle de programmation dAthapascan-1.

Le parallélisme de cet algorithme est défini sur les opérations sur les colonnes de la matrice. Deux types de tâches `scal` et `axpy`, réalisant les deux types d'opération sur des colonnes de la matrice, sont alors définies lignes 2 et 7. Le type de tâche `axpy` ne modifie pas la colonne référencée par le paramètre `x`. Pour cette raison, seul un droit d'accès en lecture sur ce paramètre `x` est indiqué.

Aux lignes 14 et 15, le type de la matrice (de taille $n \times n$) est défini. Une matrice est alors constituée d'un tableau de n références **Shared_rp_wp** sur des objets partagés. Chaque objet partagé de type `column`, constitué d'un tableau `vector` de n éléments, représente alors une colonne de la matrice.

1. L'écriture de l'élimination de Gauss sous cette forme n'est pas la plus efficace. Une écriture dans le modèle de programmation d'Athapascan-1 d'une factorisation LU par élimination de Gauss, utilisant un partitionnement bidimensionnel de la matrice est présentée dans le chapitre 7.

```
1:  // x ← a * x
2:  void scal(int k, int n, double a, Shared_r_w column x ) {
3:      for( int i = k+1; i<n ; i++ )
4:          x[i] = a * x[i];
5:  }
6:  // y = a * x + y
7:  void axpy(int k, int n, double a,
8:            Shared_r column x, Shared_r_w column y) {
9:      for( int i = k+1; i<n ; i++ )
10:         y[i] = a * x[i] + y[i];
11: }
12: main() {
13:     int n = 100;
14:     int j,k;
15:     typedef double column[n];              // type des colonnes de la matrice
16:     typedef Shared_rp_wp column matrix[n]; // type de la matrice
17:
18:     matrix A;
19:     // ici, initialisation de A
20:
21:     for( k = 0; k < n − 1; k++ )
22:         Fork scal(k, n, 1/A[k][k], A[k]);
23:         for( j = k+1; j<n; j++ )
24:             Fork axpy(k, n, −A[j][k], A[k], A[j]);
25: }
```

Figure 3.2 – *Élimination de Gauss par colonnes écrite dans le modèle de programmation Athapascan-1*

L'algorithme d'élimination de Gauss par colonnes est ensuite implanté de la ligne 20 à la ligne 23 ; chaque opération sur une colonne étant remplacée par la création de la tâche correspondante.

3.3 Discussion sur le modèle de programmation

Dans cette section, nous étudions les principales contraintes imposées par le modèle de programmation et leur influence sur l'écriture d'un programme. Le premier point concerne les créations de tâches, et plus particulièrement le fait qu'une tâche ne peut pas directement accéder aux résultats d'une tâche qu'elle a invoquée. Le deuxième point traite de l'influence de la granularité des objets partagés sur la granularité des tâches du

programme.

3.3.1 Programmation par continuation explicite

Les restrictions imposées pour éviter les synchronisations au cours de l'exécution d'une tâche font qu'il n'est pas possible pour une tâche mère d'accéder directement aux résultats de l'une de ces tâches filles. La seule façon pour elle d'y accéder est de créer une seconde tâche prenant en paramètre les références les objets devant être accédés (programmation par continuation). C'est par exemple le cas dans le programme de la figure 3.1 page 48 calculant récursivement le $n^{ème}$ de la suite de Fibonacci. Dans ce programme, à la ligne 18, une tâche doit être invoquée pour accéder au résultat précédemment calculé (ici pour l'affichage).

Cependant, dans beaucoup de cas une tâche n'a pas besoin de connaître le résultat d'une tâche qu'elle a invoquée pour continuer à décrire la suite de l'exécution parallèle. Par exemple, dans le cas de l'élimination de Gauss par colonnes de la figure 3.2 page 50, la tâche `main` n'a pas besoin de connaître le résultat de l'exécution de la tâche `scal` mettant à jour la première colonne pivot pour continuer à décrire la suite de l'exécution. Elle peut donc créer les tâches suivantes qui, elles, devront attendre le résultat de cette tâche pour commencer leurs exécutions. Pour ce programme, l'exécution des tâches peut même ne commencer qu'après la terminaison de la tâche `main` et donc l'ensemble des tâches à exécuter peut être connu avant leurs exécutions. Il faut bien noter que ceci n'est possible que si le modèle de programmation permet de créer des tâches pouvant attendre la terminaison d'autres tâches avant de s'exécuter. Le modèle de programmation de Cilk [11], par exemple, ne permet pas de créer ce type de tâche[2]. Dans ce dernier modèle, la synchronisation après la création de chaque tâche pivot est obligatoire, le système n'a donc qu'une connaissance partiel de l'ensemble des tâches qui seront à exécuter.

3.3.2 Structuration des données du programme

La granularité des objets partagés influence directement le degré de parallélisme des tâches qui seront créées. Par exemple, invoquer plusieurs tâches modifiant des parties distinctes d'un même objet partagé ne générera pas de parallélisme supplémentaire puisque, dans le modèle de programmation, les contraintes d'accès sur un objet partagé s'appliquent à l'ensemble des données de cet objet. Le système d'exécution va alors considérer que chacune de ces tâches accède à l'ensemble de l'objet partagé, il va donc les exécuter les unes après les autres. De plus, si ces tâches sont exécutées sur différens processeurs d'une machine à mémoire distribuée, des copies inutiles de l'objet seront réalisées entre les différents nœuds.

2. Plus exactement seule la continuation d'une tâche située après un mot clef `sync` peut se synchroniser sur la terminaison d'autres tâches.

Le programmeur doit également éviter pour la même raison de regrouper sur un même objet partagé des données qui seront éventuellement accédées différemment par une tâche. Par exemple, considérons une tâche accédant un objet partagé, contenant deux données A et B respectivement lue et modifiée par cette tâche. L'objet contenant A et B doit donc être pris en paramètre par la tâche avec un droit d'accès en lecture-écriture même si A n'est accédé qu'en lecture.

Cette structuration des données suivant la granularité choisie pour exprimer le parallélisme est peut être ce qui est le plus contraignant lors de l'écriture d'un programme Athapascan-1. Cependant cette structuration des données d'un programme en vue de son exécution sur une machine à mémoire distribuée est généralement indispensable sur la plupart des modèles de programmations offrant une mémoire partagée distribuée. La solution consistant à utiliser une mémoire virtuellement partagée du type de celle fournie par TreadMarks [3], et à synchroniser les tâches explicitement, pourrait être considérée comme une solution permettant d'éviter cette structuration des données suivant la granularité choisie pour exprimer le parallélisme. Mais en pratique, cette simulation d'un espace d'adressage unique est réalisée au niveau de la page du système d'exploitation. Le programmeur doit alors également structurer ces données pour les aligner sur les pages du système s'il veut obtenir de bonnes performances. Il doit éviter par exemple de placer sur une même page deux données qui seront accédées en parallèle par deux tâches différentes[3]. De plus le surcoût induit par la simulation d'un espace d'adressage unique est généralement important, comparé à celui induit par la gestion d'une mémoire partagée au niveau objet [9]. Ainsi, même sur une mémoire virtuellement partagée, les données doivent être structurées en fonction de la granularité du parallélisme si les performances sont recherchées.

Par contre, spécifier le type des accès qui seront réalisés sur les objets partagés n'est pas un problème. Beaucoup de langages de programmation supportent des notions similaires. Par exemple, en C ou C++ le mot clé `const` est utilisé devant un nom de type pour préciser que la variable ou l'objet est constant donc qu'il ne pourra être modifié. En ADA et en Fortran 90 les mots clefs `in` et `out` permettent d'indiquer lors de la déclaration des paramètres formels d'une procédure, le type d'accès pouvant être réalisé sur ces paramètres ; lecture pour un paramètre passée en mode `in`, affectation pour un paramètre passée en mode `out` et lecture-écriture pour un paramètre passée en mode `in out`.

3.4 Conclusion

Nous avons présenté dans ce chapitre le modèle de programmation parallèle d'Athapascan-1. Ce modèle permet une programmation parallèle simple et de haut niveau. Ceci

3. Des protocoles d'écritures multiples sur une même page par différents processeurs ont cependant été développés et permettent de résoudre en partie ce problème [3].

est obtenu par trois mécanismes offerts par le modèle :

– Une mémoire partagée distribuée permet de libérer le programmeur de la distribution et de la communication des données.

– La décomposition de l'exécution sous la forme de tâches conduit à une abstraction du nombre de processeurs et libère ainsi le programmeur de l'ordonnancement et de la régulation de charge. Seul le choix de la granularité des tâches et donc du degré de parallélisme de l'application, reste à la charge du programmeur.

– Les synchronisations entre les tâches sont implicitement définies à partir de la sémantique du modèle de programmation qui est une sémantique « séquentielle » et donc naturelle pour le programmeur.

Par ailleurs, dans le but de permettre une mise en œuvre efficace de ce modèle de programmation, plusieurs choix ont été pris :

– Les accès à la mémoire partagée distribuée se font au niveau des objets et non au niveau du mot mémoire élémentaire.

– Chaque tâche déclare dans son prototype les accès qu'elle (et sa descendance éventuelle) pourra réaliser sur les objets en mémoire partagée. Ainsi le système connaît l'ensemble des effets de bord pouvant être réalisés par les tâches sur les objets partagés. Il peut alors facilement analyser les dépendances de données entre les tâches et ainsi définir les synchronisations nécessaires au respect de la sémantique séquentielle.

– Aucune dépendance de données entre une tâche mère et une tâche fille n'est autorisée. Ainsi une tâche peut toujours s'exécuter sans synchronisation. Il est alors possible de mettre en œuvre des mécanismes d'ordonnancement et de régulation de charge prouvé théoriquement efficace.

Ces choix seront justifiés plus en détail dans la suite de cette thèse et plus particulièrement dans les deux chapitres suivants dans lesquels un modèle de coût de l'exécution d'un programme Athapascan-1 sera proposé.

4

Modèle d'exécution d'un programme Athapascan-1

4.1 Introduction

Le modèle de programmation Athapascan-1 introduit dans le chapitre précédent permet d'exprimer le parallélisme indépendemment de l'architecture. L'exécution d'un programme repose alors sur un système d'exécution qui contrôle la répartition (ou plutôt le repliement) d'un programme sur l'architecture cible. Ce chapitre est consacré à la présentation de ce système, dont comme nous le verrons, l'implémentation dépend de l'architecture cible.

Le rôle du système d'exécution associé à Athapascan-1 est double :

– D'une part, détecter les synchronisations qui sont implicites dans le langage Athapascan-1. Dans ce chapitre nous proposerons une méthode basée sur la construction d'un graphe de flot de données. Ce graphe, construit au cours de l'exécution, représente à tout instant les dépendances de données entre les différentes tâches déjà créées mais non encore exécutées.

– D'autre part, l'ordonnancement du programme, à savoir le placement et l'entrelacement des tâches de calculs et la localisation des objets partagés. Le graphe de flot de données construit en cours d'exécution fournit alors un bon support pour le calcul de cet ordonnancement. Par exemple, une grande partie des algorithmes d'ordonnancement statique utilisent ce type de graphe.

Nous montrerons alors dans ce chapitre trois modèles d'exécution d'Athapascan-1, dans lesquels la politique d'ordonnancement utilisée est clairement séparée des différents

mécanismes nécessaires à l'exécution (*i.e.* interprétation du flot de données, gestion des objets distribués, ...). Les différents algorithmes d'ordonnancement pouvant être utilisés dans ces modèles d'exécution seront quant à eux étudiés dans le chapitre 5.

La structure du chapitre est la suivante. Dans une première partie, nous modélisons l'exécution d'un programme Athapascan-1 par un graphe de flot de données. Nous montrons ensuite comment ce graphe de flot de données est implicitement construit et utilisé pendant l'exécution d'un programme Athapascan-1. Finalement, nous introduisons trois modèles d'exécution d'un programme Athapascan-1, qui tirent plus ou moins parti de la connaissance de ce graphe.

4.2 Flot de données comme représentation de l'exécution

Nous montrons dans cette section comment l'exécution d'un programme Athapascan-1 peut être représentée par un graphe de flot de données, dans lequel les nœuds correspondent à des tâches du programme et le flot de données correspond à des dépendances entre ces tâches induites par l'accès à un même objet partagé.

Un graphe de flot de données ou *macro data-flow graph* est composé de noeuds correspondants à des activités séquentielles encore appelé tâches (*i.e.* une séquence d'instructions exécutable sans interruption et donc atomique) et d'arcs entre ces noeuds représentant des dépendances de données entre ces tâches : écriture par une première tâche suivie d'une lecture par une seconde tâche.

Cependant, comme plusieurs tâches peuvent accéder en concurrence à un même objet partagé (lecture concurrente mais également écriture concurrente par accumulation), le flot de données peut être plus facilement représenté par un graphe composé de deux ensembles de nœuds : l'ensemble de nœuds $T = \{t_1, \ldots, t_n\}$ correspondant aux tâches et l'ensemble de nœuds $A = \{a_1, \ldots, a_m\}$ représentant un accès, par une ou plusieurs tâches en concurrence, sur un même objet partagé. Les nœuds tâches sont alors associés à des nœuds accès par un ensemble d'arêtes reliant une tâche à un accès. Le flot de données est lui représenté par un ensemble d'arcs reliant entre eux les nœuds accès. La figure 4.1 montre un exemple de graphe de flot de données représentant sept tâches accédant à un même objet partagé.

Dans le cas d'un programme Athapascan-1, de par les restrictions imposées par le langage (*i.e.* la règle définie dans la section 3.1.4 page 45 qui interdit de passer une référence contrainte avec un droit de modification non différé à une tâche fille), une tâche du langage est atomique puisque constituée d'une séquence d'instructions pouvant être exécutée sans synchronisation. Elle est donc directement associée à un noeud du graphe de flot de données.

Par ailleurs, un autre type d'arc est également présent dans le graphe permettant de

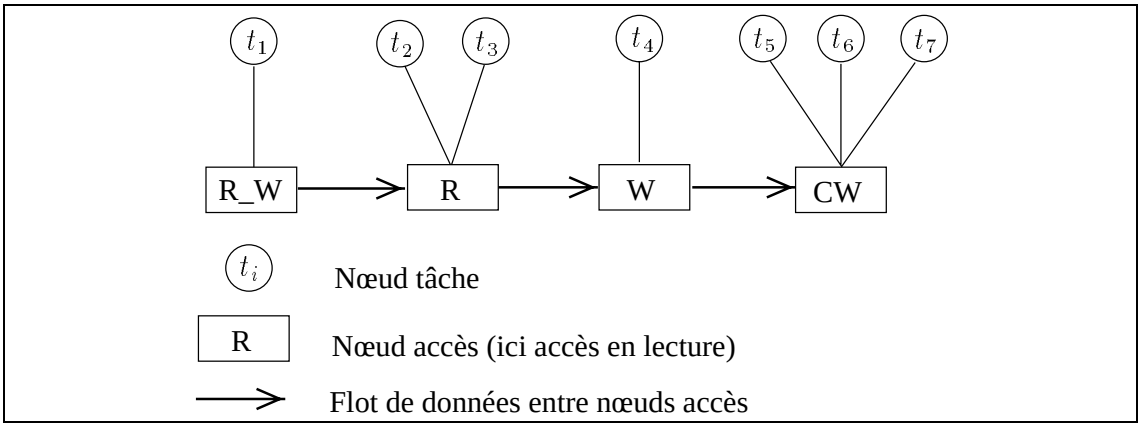

Figure 4.1 – *Les quatre types d'accès du graphe de flot de données. Ce graphe représente 7 tâches accédant à un même objet partagé. La tâche t_1 accède l'objet en lecture et écriture (R_W) puis les tâches t_2 et t_3 accèdent l'objet en lecture (R), puis la tâche t_4 accède l'objet en écriture (W) et enfin les tâches t_5, t_6 et t_7 accèdent l'objet en accumulation (CW).*

représenter la précédence entre une tâche fille et la tâche mère qui l'a invoquée : Ce type d'arc représente le flot de données correspondant aux paramètres passés par valeur lors des créations de tâches.

Les figures 4.2 et 4.3 montrent le graphe de flot de données ainsi obtenu après l'exécution des deux programmes présentés dans le chapitre précédant figure 3.1 et 3.2, calculant respectivement le $n^{ème}$ terme de la suite de Fibonacci et l'élimination de Gauss par colonne.

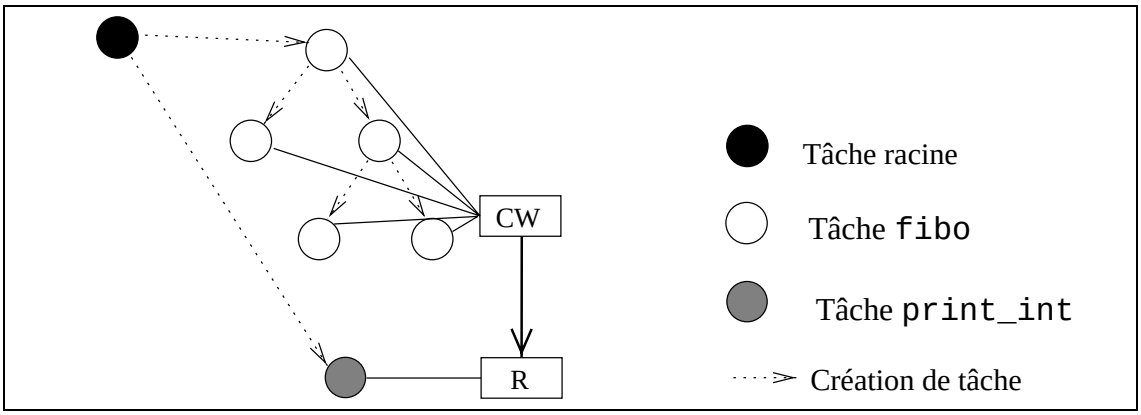

Figure 4.2 – *Le graphe de flot de données du programme calculant le $4^{ème}$ terme de la suite de Fibonacci présentée dans la figure 3.1 page 48.*

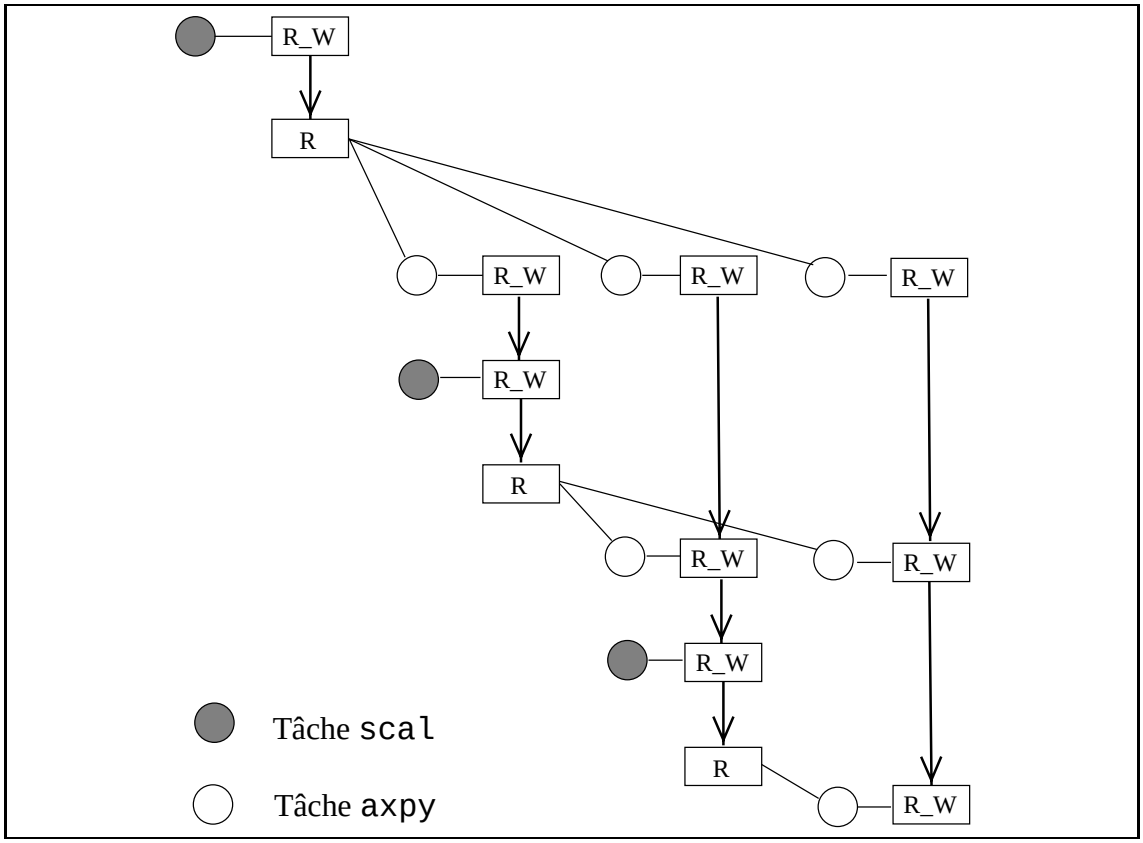

Figure 4.3 – *Le graphe de flot de données du programme présenté dans la figure 3.2 page 50 réalisant l'élimination de Gauss par colonnes d'une matrice 4×4. Pour la clarté de la figure la tâche racine ainsi que les précédences induites par les créations de tâche ont été omises (la tâche racine est ici la seule à créeer d'autres tâches).*

4.3 Modèle d'exécution et flot de données

Le graphe de flot de données, présenté précédemment et représentant l'exécution d'une application a également une place centrale dans l'exécution d'un programme Athapascan-1. Le modèle d'exécution d'un programme Athapascan-1 est basé sur la construction de ce graphe à la volée. Ainsi, lors de chaque opération **Fork**, la tâche créée est rajoutée dans le graphe. Ce graphe représente alors à tout instant les tâches à exécuter ainsi que les contraintes de flot entre ces tâches (*i.e.* les contraintes de précédences induites par l'accès aux objets partagés). Simultanément à la construction du graphe, le système doit également décider des tâches qui peuvent être exécutées immédiatement sans violer les contraintes de flot décrites dans le graphe.

Nous présentons dans cette section tout d'abord la méthode utilisée pour calculer à

la volée les contraintes de flot et donc construire le graphe de flot de données. Nous présenterons ensuite l'algorithme de contrôle des contraintes de flot permettant de déterminer, en cours d'exécution, les tâches de ce graphe pouvant être exécutées sans violer les contraintes de flot décrites par ce graphe. Nous présenterons ensuite rapidement le problème de l'ordonnancement des tâches du graphe de flot de données. Enfin, nous conclurons cette section par la présentation du schéma général d'exécution d'un programme Athapascan-1, regroupant les trois modules, calcul des contraintes de flot, contrôle de ces contraintes et ordonnancement des tâches du graphe.

4.3.1 Calcul des contraintes de flot

Le calcul des contraintes de flot de données et donc du parallélisme se ramène au calcul du graphe de flot de données. Le but de cette section est de montrer comment le graphe peut être calculé et d'exhiber le coût de sa construction.

Notons que la seule tâche connue avant l'exécution est la tâche racine qui sera exécutée lors du lancement du programme. Le graphe de flot de données n'est donc découvert qu'en cours d'exécution, au fur et à mesure des créations de tâches. Le graphe qui représente le flot de données est donc construit dynamiquement pendant l'exécution.

Lorsqu'une tâche est exécutée, sa présence dans le graphe n'est plus utile, elle peut être retirée de celui-ci. Le graphe maintenu pendant l'exécution ne contiendra donc que les tâches créées et non encore exécutées. Les arcs représentant la précédence entre une tâche fille et la tâche mère qui l'a invoquée peuvent également être ignorés puisque la tâche fille n'apparaît dans le graphe qu'après sa création.

À chaque opération **Fork** (*i.e.* à chaque création de tâche), un nœud tâche est rajouté dans le graphe de flot de données. Ce nœud tâche représente alors dans le graphe une tâche à exécuter. Par ailleurs, cette tâche peut avoir des dépendances de données avec d'autres tâches non encore exécutées et donc présentes dans le graphe. Des arcs (et éventuellement des nœuds accès) sont alors rajoutés dans le graphe pour représenter le flot des données (*i.e.* les dépendances de données) entre la tâche nouvellement créée et ces tâches.

Pour calculer directement, lors d'une création de tâche, les arcs (et éventuellement les nœuds accès) à rajouter dans le graphe, chaque référence contrainte (rappelons qu'une référence contrainte est une variable dont le type est préfixé par l'un des mots clefs commençant par **Shared**) contient un pointeur sur l'un des nœuds accès du graphe de flot de données. Ce nœud accès représente d'une certaine manière la « position » dans le graphe de l'accès potentiellement réalisable à partir de cette référence contrainte.

Lors de la création d'une tâche fille t_f par la tâche mère t_m les opérations suivantes sont alors réalisées :

– La tâche fille t_f est insérée dans le graphe.

– Pour chaque référence partagée r_k passée en paramètre à la tâche fille :

  – Si la tâche peut accéder en concurrence au nœud accès a pointé par la référence r_k alors une arête est rajouté entre la tâche t_f et ce nœud accès (deux accès peuvent être réalisés en concurrence uniquement si les deux accès sont du même type ; lecture ou accumulation avec la même fonction).

  – Sinon un nouveau nœud accès a' est inséré immédiatement après le nœud a ; une arête est rajoutée entre la tâche t_f et ce nœud accès et le pointeur de la référence contrainte r_k est déplacé sur ce nouveau nœud.

La figure 4.5 page 61 illustre ce mécanisme de création dynamique du graphe de flot de données sur le programme présenté dans la figure 4.4 page 60. La figure 4.5 montre alors l'état du graphe après l'exécution de chacune des lignes 8 à 13 du programme.

```
1:  struct lecture {
2:    void operator()(Shared_r<int> a) { . . . }
3:  };
4:  struct modification {
5:    void operator()(Shared_r_w<int> a) { . . . }
6:  };
7:  main() {  // tâche t0
8:    Shared_r_w<int> a(1);     // création d'un objet partagé (étape a)
9:    Fork<lecture>()(a);         // création de la tâche t 1     (étape b)
10:   Fork<lecture>()(a);         // création de la tâche t 2     (étape c)
11:   Fork<lecture>()(a);         // création de la tâche t 3     (étape d)
12:   Fork<modification>()(a);    // création de la tâche t 4     (étape e)
13:   Fork<modification>()(a);    // création de la tâche t 5     (étape f)
14: }
```

Figure 4.4 – *Exemple d'un programme simple illustrant la création dynamique du graphe présenté dans la figure4.5.*

Chaque opération **Fork** a un coût proportionnel au nombre de paramètres pris par la tâche. Le coût de construction du graphe de flot de données peut alors être borné en fonction du nombre de tâche créées et du nombre total de paramètres de chacune des tâches. Nous obtenons alors la proposition suivante :

Proposition 1 *La construction à la volée du graphe de flot de données d'un programme Athapascan-1 est bornée en nombre d'opération par :*

$$O(n + n_d)$$

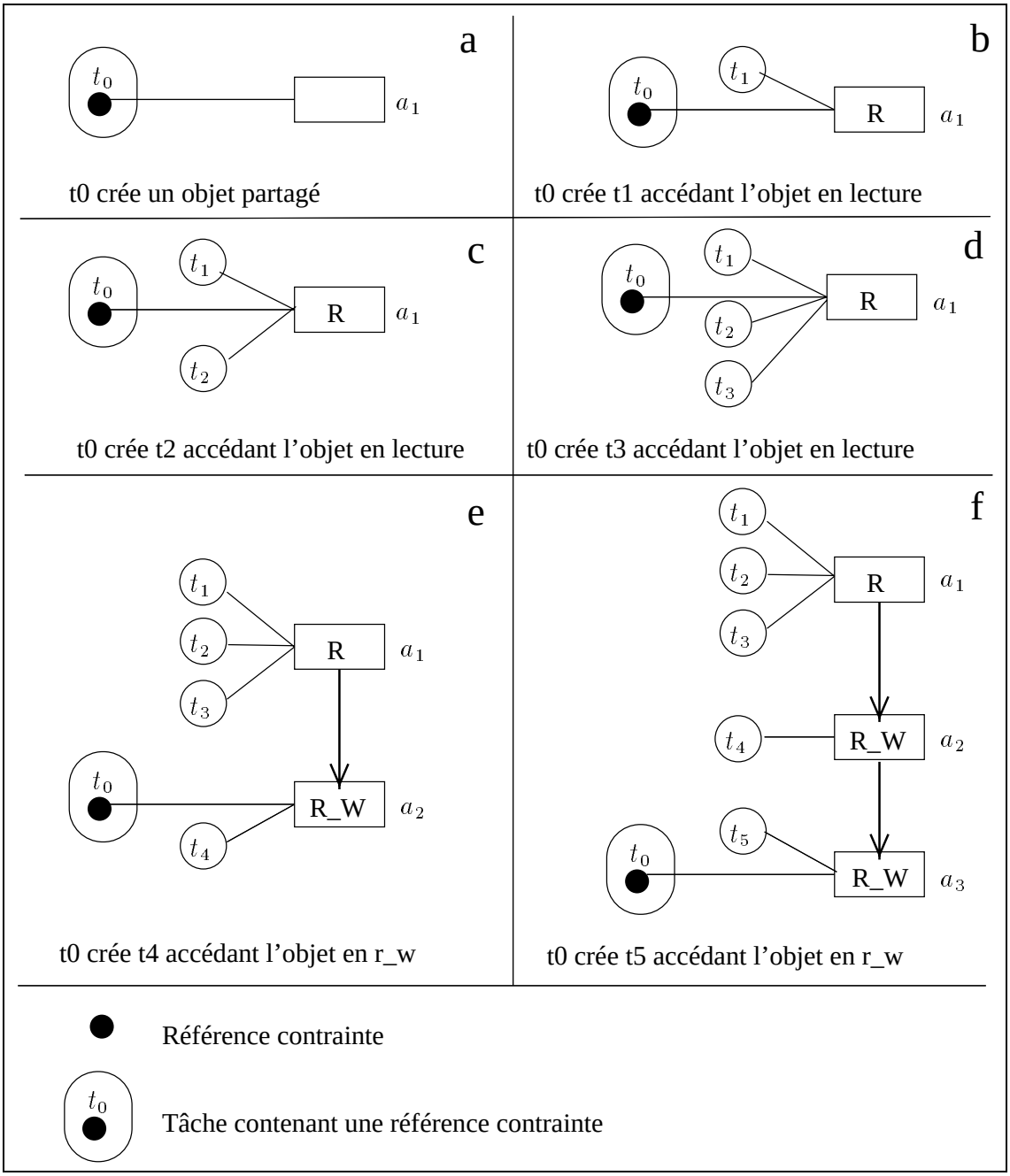

Figure 4.5 – *Construction dynamique du graphe de flot de données lors de l'exécution du programme de la figure 4.4. On peut remarquer que les tâches t_1 à t_3 sont reliées au même nœud accès que celui pointé par la référence car deux accès en lecture peuvent être réalisés en concurrence. Par contre lors de la construction des tâches t_4 et t_5, accédant l'objet en lecture-écriture un nouveau nœud accès est créé, la tâche créée est reliée à ce nœud et la référence contrainte de la tâche t_0 pointe alors sur ce nouveau nœud.*

où n est le nombre de tâches créées et n_d est le nombre de références contraintes passées en paramètre des tâches.

Preuve. Triviale. □

4.3.2 Contrôle des contraintes de flot

Le contrôle des contraintes de flot consiste à déterminer, à un instant donné de l'exécution, l'ensemble des tâches prêtes, c'est-à-dire l'ensemble des tâches du graphe pouvant être exécutées immédiatement sans violer les contraintes de précédence décrites par le graphe de flot de données (nous rappelons que ces contraintes de précédences ont été définies lors de la construction du graphe de manière à respecter la sémantique séquentielle du programme).

Pour maintenir à tout instant l'ensemble des tâches prêtes, il suffit de calculer, lors de la terminaison d'une tâche, les nouvelles tâches devenant prêtes. Pour cela, un état est associé à chaque nœud tâche et chaque nœud accès du graphe. Le tableau 4.1 précise ces différent états.

L'algorithme calculant lors de la terminaison d'une tâche les nouvelles tâches prêtes est alors le suivant. Lorsqu'une tâche se termine, chacun des nœuds accès auquel elle est reliée est consulté pour éventuellement passer à l'état terminé. Si l'un de ces noeuds passe à l'état terminé alors le nœuds accès suivant dans le flot de données passe à l'état prêt et chacune des tâches qui lui sont reliées est consultée pour déterminer si elle peut passer à l'état prêt. L'ensemble des tâches devenant prêtes après la terminaison d'une tâche peut ainsi être calculé. L'algorithme 2 détaille plus précisément ce mécanisme.

État	Noeud accès	Noeud Tâche
Attente	Le nœud accès prédécesseur n'est pas terminé.	Un des nœuds accès de la tâche n'est pas prêt.
Prêt	Le nœud accès prédécesseur est terminé.	Tout les nœuds accès de la tâche sont prêts.
Exécution	x	La tâche est en cours d'exécution.
Terminé	Toutes les tâches reliées au nœud accès sont terminées.	L'exécution de la tâche est terminée.

Tableau 4.1 – *Les différents états possibles des nœuds du graphe de flot de données.*

Une optimisation peut être apportée dans le calcul de l'ensemble des tâches prêtes à un instant donné de l'exécution en considérant les références contraintes différées. Nous rappelons qu'une référence contrainte est dite différée pour indiquer que l'objet référencé ne sera pas accédé par la tâche mais uniquement passé à des tâches filles. Une tâche,

Algorithme 2 : Calcul des nouvelles tâches prêtes.

Entrée : Le graphe de flot de données composé de nœuds tâche et de nœuds accès.
Entrée : Une tâche t_i venant de terminer son exécution.
Sortie : L'ensemble R des nouvelles tâches prêtes.

```
pour tout a_j, nœud accès de la tâche t_i faire
    Retirer l'arête entre t_i et a_j.
    si a_j n'a plus d'arrête (i.e. a_j est terminée) alors
        si a_j a un nœud successeur immédiat a_k alors
            a_k reçoit l'objet partagé de a_j et passe à l'état prêt.
            pour tout t_l, tâche accédant au nœud accès a_k faire
                si t_l a tous ses nœuds accès prêts alors
                    t_l passe à l'état prêt et R = R ∪ {t_l}.
        sinon
            L'objet partagé associé au nœud accès a_j est détruit.
        Suppression du nœud a_j du graphe.
    Suppression du nœud t_i du graphe.
```

accédant un objet de manière différé, peut donc être considérée comme prête, même si cet objet n'est pas immédiatement accessible. Par contre, une tâche fille qui accède cet objet de manière non différée, devra attendre que l'objet soit accessible pour devenir prête.

Le contrôle des contraintes de flot consiste, par des exécutions successives de l'algorithme 2, à parcourir une et une seule fois ce graphe. Nous pouvons alors borner le coût induit par cette opération :

Proposition 2 *Le nombre d'opération nécessaire au contrôle des contraintes de flot d'un programme Athapascan-1, est bornée par :*

$$O(n + n_d)$$

où n est le nombre de tâches créés et n_d est le nombre de références contraintes passées en paramètre des tâches.

Preuve. L'algorithme 2 est exécuté n fois. Une itération de la boucle externe est réalisée pour chaque arête du graphe reliant les tâches aux nœuds accès donc le nombre d'itérations de cette boucle est n_d. Le code situé à l'intérieur du premier test est lui exécuté lors de chaque passage d'un nœud accès à l'état prêt. Ce code est donc exécuté au plus n_d fois (Tout nœud accès est relié à au moins une tâche donc le nombre de nœuds accès est borné par le nombre d'arêtes). Une itération de la boucle interne est réalisée pour chaque arête du graphe donc le nombre d'itérations de la boucle interne est n_d. □

4.3.3 Ordonnancement des tâches du graphe de flot de données

Le nombre de processeur étant limité, les nouvelles tâches prêtes générées par l'algorithme de contrôle précédent ne peuvent généralement pas être exécutées immédiatement. Se pose alors le problème de l'ordonnancement de ces tâches sur un nombre borné de ressources. Ce problème consiste à choisir pour chaque tâche un processeur et une date d'exécution en cherchant généralement à minimiser un ou plusieurs critères. Ces critères sont principalement le temps total d'exécution et l'espace mémoire utilisé pendant l'exécution[1]. Nous n'aborderons ici que rapidement le problème de l'ordonnancement. Ce problème sera traité en détail dans le chapitre suivant.

Dans le cas d'un programme Athapascan-1, le graphe n'est connu qu'en cours d'exécution. Les techniques d'ordonnancement à mettre en œuvre seront donc du type ordonnancement à la volée (*on-line scheduling*). Cependant, pour certains programmes, les techniques d'ordonnancements statiques (*off-line scheduling*) peuvent aussi être utilisées. En effet, à tout instant, le graphe de flot de données décrit le futur de l'exécution. Cette connaissance est partielle puisqu'une tâche peut *a priori* créer de nouvelle tâches lors de son exécution. Cependant cette connaissance peut être utilisée pour calculer un ordonnancement statique adapté, capable de prendre en compte par exemple les contraintes de localité contenues dans le graphe.

4.3.4 Conclusion : modèle d'exécution général d'Athapascan-1

Le modèle d'exécution général d'un programme Athapascan-1 est composé de trois modules distincts : calcul des contraintes de flot génération, contrôle de ces contraintes et ordonnancement des tâches. Ces modules communiquent entre eux, soit directement, soit par l'intermédiaire du graphe de flot de données. La figure 4.6 présente le diagramme du modèle d'exécution d'un programme Athapascan-1, valable aussi bien pour un ordonnancement à la volée que pour un ordonnancement statique des tâches.

À chaque création de tâche (instruction **Fork**) le module de création du graphe insère une nouvelle tâche à exécuter dans le graphe de flot de données. À chaque terminaison de tâche, le module d'interprétation du graphe calcule les nouvelles tâches prêtes et envoie ces tâches au module d'ordonnancement. À chaque fois qu'un processeur devient disponible (*i.e.* après la terminaison d'une tâche), le module d'ordonnancement choisit une tâche à exécuter parmi les tâches prêtes, ce choix dépendant de la politique d'ordonnancement utilisée.

1. Dans le cas d'une mémoire bornée, l'ordonnancement des tâches doit de plus être réalisable sur la mémoire disponible.

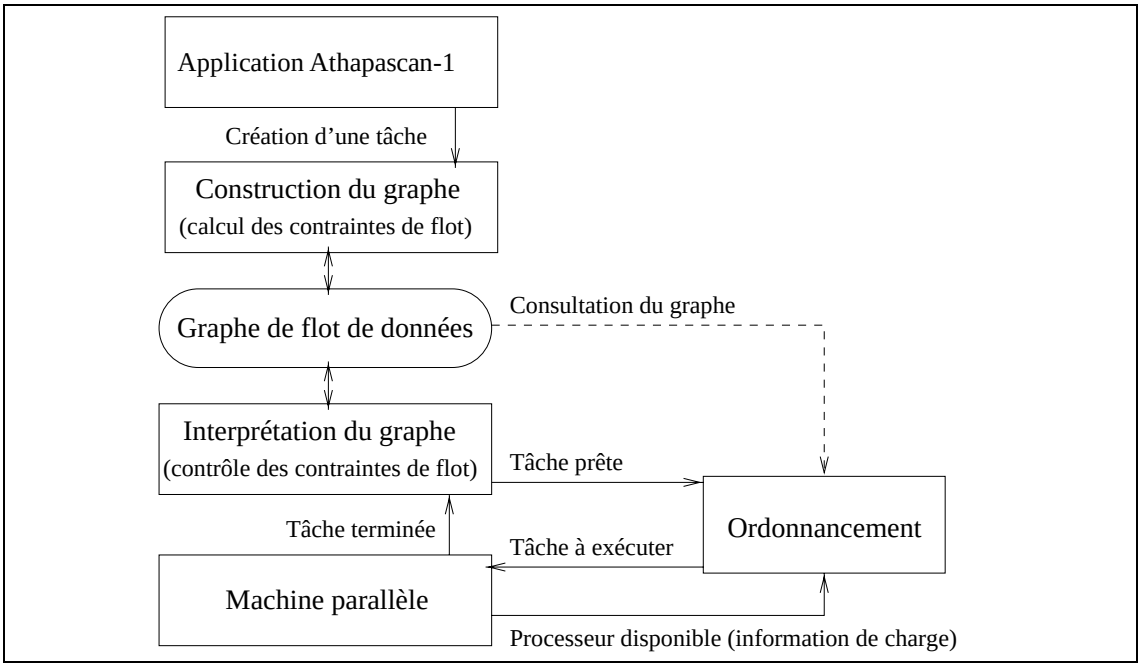

Figure 4.6 – *Modèle d'exécution d'un programme Athapascan-1.*

4.4 Analyse du coût du modèle d'exécution

Nous présentons dans cette section une analyse du coût du modèle d'exécution sur différentes architectures et pour différents types d'ordonnancement. Pour cela le modèle d'exécution général présenté précédemment sera spécialisé pour ces différentes architectures et types d'ordonnancement. Trois modèles d'exécution seront alors présentés. Un modèle valable pour les architectures à mémoire partagée et deux modèles pour les architectures à mémoire distribuée.

Pour analyser le coût d'exécution d'un programme Athapascan-1, cette exécution est représentée par son graphe de flot de données G composé de n tâches prenant en paramètres au total n_d références contraintes (donc n_d est la somme des durées des noeuds tâches).

Dans la suite de cette section, nous chercherons à borner le temps d'exécution T_{exec} d'un programme Athapascan-1 modélisé par le graphe G par :

$$T_{exec} \leq \omega_m(G) + \rho_m(G) + \pi_m(G)$$

où $\omega_m(G)$, $\rho_m(G)$ et $\pi_m(G)$ sont définies de la manière suivante :

– $\rho_m(G)$ correspond au temps passé dans le calcul d'un ordonnancement du graphe G pour une exécution sur m processeurs. Ce temps correspond au temps passé dans

le module d'ordonnancement du modèle d'exécution d'Athapascan-1. Ce temps ne dépend que de l'algorithme d'ordonnancement utilisé.

– $\omega_m(G)$ correspond au temps d'exécution sur m processeurs du programme modélisé par le graphe G. Ce temps ne considère que le coût d'exécution des tâches (sans compter le coût des opérations **Fork**) ainsi que le coût de communication des données par le réseau (sans compter le coût de contrôle de la mémoire partagée distribuée) lorsque l'architecture est distribuée. Ce temps dépend uniquement du graphe G.

– $\pi_m(G)$ correspondant au surcoût d'Athapascan-1. Ce surcoût comprend entre autre le temps passé au calcul des contraintes de flot (construction du graphe), au contrôle de ces contraintes et, sur une architecture distribuée, à la gestion de la mémoire partagée distribuée.

Dans la suite, nous nous intéresserons exclusivement à $\pi_m(G)$ correspondant au surcoût d'Athapascan-1. Les valeurs $\omega_m(G)$ et $\rho_m(G)$ seront quant à elles bornées dans le chapitre suivant consacré à l'ordonnancement de graphe de flot de données.

4.4.1 Analyse du coût sur une machine à mémoire partagée

Le modèle d'exécution décrit dans la figure 4.6 page 65 s'applique directement à l'exécution sur une machine à mémoire partagée : en effet, le graphe de flot de données ainsi que les objets partagés peuvent être accédés depuis l'ensemble des processeurs.

Ainsi, lorsqu'une instruction **Fork** est rencontrée, le processeur envoit au module de construction du graphe la nouvelle tâche créée pour l'insérer dans le graphe. Lorsqu'un processeur termine l'exécution d'une tâche, il prévient le module d'interprétation du graphe pour calculer les nouvelles tâches prêtes puis ensuite le module d'ordonnancement pour obtenir une nouvelle tâche à exécuter (dans le cas ou il n'y à plus de tâche prête, le processeur se met alors en attente). Par ailleurs, un verrou global protège l'accès aux modules de construction, d'interprétation et d'ordonnancement du graphe.

Des techniques existent pour supprimer la prise de ce verrou global à chaque création ou terminaison de tâche dans le but de réduire les surcoûts de gestion et d'ordonnancement du graphes (voir par exemple l'implantation de Cilk présentée dans la section 2.4.1.2 page 33 qui utilise une telle technique). Cependant, cette optimisation n'a pas d'influence dans l'analyse du coût que nous proposons pour une application quelconque[2]. Nous ne la considérerons donc pas dans l'analyse théorique.

2. Plus exactement lorsque le graphe est généré séquentiellement par une seul tâche (par la tâche racine), le coût de la prise du verrou est du même ordre que le coût de construction du graphe. Le gain de cette optimisation n'apparait que pour les applications qui créent récursivement des tâches. Dans ce cas, la construction, l'analyse et l'ordonnancement du graphe peuvent être réalisés en parallèle.

La proposition suivante borne le temps d'exécution d'un programme Athapascan-1 dont l'exécution est représentée par le graphe de flot de données G :

Proposition 3 *Le temps d'exécution sur m processeurs à mémoire partagée, d'un programme Athapascan-1, représenté par le graphe de flot de données G, est borné par :*

$$T_{exec} \leq \omega_m(G) + \rho_m(G) + O(n + n_d)$$

Preuve. Soit T le temps total d'exécution. L'ensemble des tops $\{0, \ldots, T-1\}$ d'exécution est recouvert par les trois sous ensembles suivants :

- C (surcoût d'Athapascan-1) : une opération de construction ou d'interprétation du graphe est en cours sur un processeur.
- O (surcoût d'ordonnancement) : une opération d'ordonnancement est en cours sur un processeur.
- $E = T \setminus (C \cup O)$: aucun processeur n'exécute une des trois opérations de construction, d'interprétation ou d'ordonnancement du graphe. Donc chacun des processeurs, soit exécute une tâche, soit est en attente.

On a alors $\#C \leq O(n + n_d)$ de par les propositions 1 page 60 et 2 page 63, $\#O = \rho_m(G)$ de par la définition de $\rho_m(G)$ et de O et $\#E \leq \omega_m(G)$ de par la définition de $\omega_m(G)$. □

4.4.2 Analyse du coût sur une machine à mémoire distribuée

Par rapport à l'exécution sur une machine à mémoire partagée vue précédemment, des opérations supplémentaires sont à réaliser lors de l'exécution sur une machine à mémoire distribuée :

- Le graphe de flot de données est créé de manière distribuée par les tâches s'exécutant sur différents nœuds. Ce graphe ainsi que sa construction, son interprétation et son ordonnancement sont alors distribués. Par ailleurs, l'interprétation et l'ordonnancement distribués peuvent faire intervenir des communications entre processeurs.
- Une copie valide d'un objet partagé accédé par une tâche doit être présente sur un nœud pour que celui-ci puisse exécuter cette tâche. Lorsque cet objet est accédé en concurrence par plusieurs tâches s'exécutant sur différents nœuds, une copie valide de cet objet partagé doit être présente localement sur chacun de ces nœuds. Le système doit donc gérer la cohérence de ces différentes copies. Pour cela, les valeurs des objets valides sont communiquées entre processeurs, par l'intermédiaire du réseau.

  Remarque 1 *Ce problème est similaire au problème de maintien de la cohérence des pages dans les systèmes simulant une mémoire partagée [3]. Cependant, nous*

verrons dans la suite, qu'il est possible de combiner la gestion de la cohérence des copies des objets partagés avec l'interprétation distribuée du graphe de flot de données.

– L'ordonnancement des tâches doit alors prendre en compte les délais de communication nécessaires au maintien de la cohérence des différentes copies des objets partagés. Les algorithmes d'ordonnancement de graphe de flot de données avec délai de communication seront abordés dans le chapitre suivant.

Le modèle d'exécution distribué général, utilisant un graphe distribué est basé sur un algorithme de terminaison distribué réactif. Ce modèle d'exécution est détaillé dans la thèse de François Galilée [48].

Dans la suite de ce chapitre, nous présentons deux modèles d'exécution spécifique pour l'exécution d'un programme Athapascan-1 sur machine à mémoire distribuée. Le premier, basé sur une centralisation du graphe sur un processeur, est valable pour une application Athapascan-1 quelconque. Le second, basé sur un ordonnancement statique du graphe, n'est valable que pour la classe des applications Athapascan-1 dont le graphe de flot de données est entièrement connu après l'exécution de la tâche racine.

4.4.2.1 Graphe et ordonnancement centralisés

Nous décrivons dans cette section un modèle d'exécution d'Athapascan-1 pour machine à mémoire distribuée valable pour tout type de programme Athapascan-1 ; l'ordonnancement et la gestion du graphe et des objets partagées sont centralisées. L'intérêt de ce modèle d'exécution centralisé se justifie en partie par l'existence de certaines politiques d'ordonnancement centralisées. En effet pour prendre une décision d'ordonnancement, ce type de politique a besoin de consulter l'ensemble des tâches prêtes et non encore exécutées ainsi que la charge de l'ensemble des processeurs. Notons cependant que ce modèle d'exécution centralisé peut être pénalisant sur un grand nombre de processeurs pour des applications dans lesquelles les tâches sont de faible granularité.

La figure 4.7 page 69 montre le diagramme du modèle d'exécution centralisé. La gestion du graphe et de l'ordonnancement est réalisée sur un processeur, appelé ci-après maître. Les autres processeurs sont dédiés à l'execution des tâches et sont appelés esclaves. Nous détaillons dans la suite les diverses opérations réalisées par le processeur maître et les processeurs esclaves.

Le processeur maître gère les événements suivants arrivant du réseau :

– Création de tâche : la tâche est alors insérée dans le graphe grâce au module de construction du graphe.

– Fin de tâche : les nouvelles tâche prêtes sont alors calculées par le module d'interprétation du graphe.

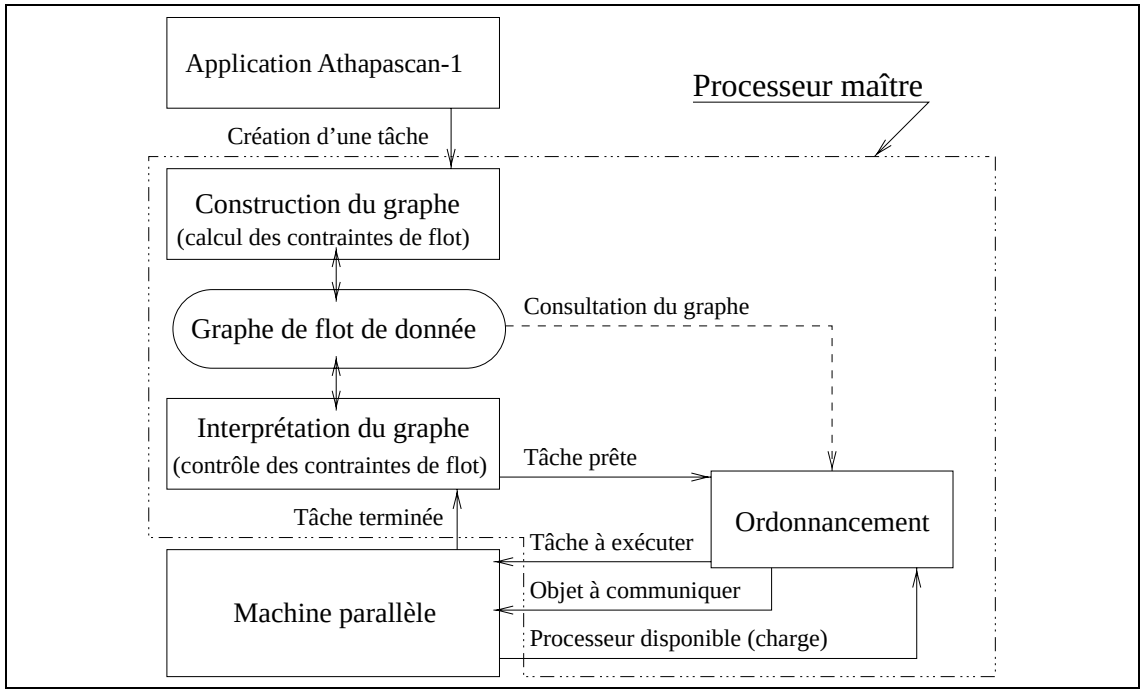

Figure 4.7 – *Modèle d'exécution d'un programme Athapascan-1 centralisé. Les opérations réalisées par le processeur maître se situe à l'intérieur du cadre en pointillé.*

Par ailleurs, à la fin du traitement chacun de ces événements, le processeur maître peut décider d'allouer une tâche prête (si elle existe) sur un des processeurs esclaves en vue de son exécution. Cette tâche est alors envoyée sur le processeur disponible et des messages de contrôle sont éventuellement envoyés à certains processeurs esclaves pour gérer la cohérence de la mémoire partagée distribuée. Ces messages consistent à demander à un processeur esclave l'envoi par le réseau d'un objet partagé sur un autre processeur esclave.

Un processeur esclave peut *a priori* disposer simultanément de plusieurs tâches à exécuter qui lui ont été allouées par le processeur maître. Chaque processeur esclave dispose alors de deux listes de tâches. L'une contient les tâches prêtes à être exécutée et l'autre contient les tâches en attente de données nécessaires à leur exécution (*i.e.* les versions valide des objets partagés). Un processeur esclave gère alors les événements suivant :

– Terminaison d'une tâche. L'identification de la tâche est alors envoyée au processeur maître.

– Réception d'une tâche allouée par le processeur maître. Cette tâche est alors mise dans l'une des deux listes de tâches.

– Réception d'un message venant du processeur maître demandant l'envoi d'un objet partagé à un autre processeur esclave. L'objet est alors envoyé immédiatement.

– Réception d'un objet partagée envoyé par un autre processeur esclave. Les deux listes de tâches sont alors mises à jour (*i.e.* les tâches devenant prêtes à être exécuter sont mises dans l'autre liste).

Par ailleurs, à la fin du traitement chacun de ces événements, si des tâches prêtes à exécuter sont présentes et si le processeur est disponible (*i.e.* il n'exécute aucune tâche), alors il démarre l'exécution d'une nouvelle tâche.

Ce schéma d'exécution suppose qu'un processeur esclave doit être capable, lors de la réception d'une communication, d'interrompre l'exécution d'une tâche pour traiter l'événement associé à la communication. Nous verrons dans le chapitre 6 comment un tel mécanisme de réactivité face aux communications peut en pratique être obtenu grâce à la bibliothèque Athapascan-0. Pour l'analyse du coût de ce modèle d'exécution, nous supposerons que l'exécution d'une tâche est interrompue immédiatement après l'arrivée d'un message venant du réseau.

La proposition suivante borne alors le temps de l'exécution d'un programme Athapascan-1, obtenu en suivant ce modèle centralisé d'exécution.

Proposition 4 *L'exécution d'un programme Athapascan-1 sur une machine à mémoire distribuée comportant $m+1$ processeurs identiques peut être réalisée en suivant le modèle d'exécution centralisé précédemment présenté. Le temps d'exécution est alors majoré par :*

$$T_{exec} \leq \omega_m(G) + \rho_m(G) + O(n + n_d) + 3\tau(n + n_d)$$

où G est le graphe généré par l'exécution et τ est le temps de communication d'un identificateur d'objet partagé ou de tâche.

Preuve. Nous définissons comme dans la section 4.4.1 page 66, les sous ensembles disjoints C (surcoût d'Athapascan-1), O (surcoût d'ordonnancement) et $E = T \setminus (C \cup O)$ qui partitionnent l'ensemble T des tops d'exécutions. Cependant C contient aussi maintenant, outre les tops pour lequel une opération de construction ou d'interprétation du graphe est en cours sur un des processeurs, les tops pendant lesquels une communication autre qu'une version d'un objet partagé est en cours (*i.e.* une communication entre le processeur maître et un processeur esclave).

La construction et l'interprétation du graphe sur le processeur maître sont, d'après les propositions 1 et 2, bornées par $O(n + n_d)$ donc $\#E \leq O(n + n_d)$. $\#O = \rho_m(G)$ de par la définition de $\rho_m(G)$ et de O et $\#E \leq \omega_m(G)$ de par la définition de $\omega_m(G)$.

Les communications générées pour contrôler l'exécution sont les suivantes :

– Lors de la création d'une tâche, les données communiquées au processeur maître sont un identificateur de la tâche et un identificateur de chaque objet partagé accédé par cette tâche. $n + n_d$ identificateurs sont au total communiqués ici.

– Lors de la terminaison d'une tâche, l'identificateur de la tâche est communiquée au processeur maître . n identificateurs sont au total communiqués ici.

– Lors du placement, par le processeur maître, d'une tâche prête sur un processeur esclave pour son exécution, les données communiquées au processeur esclave sont un identificateur de la tâche et un identificateur de chaque objet partagé accédé par cette tâche. $n + n_d$ identificateurs sont au total communiqués ici.

– Pour chaque objet partagé communiqué entre deux processeurs esclaves, un identificateur d'objet partagé a été envoyé du processeur maître vers un processeur esclave. Au plus n_d identificateurs sont au total communiqués ici.

Le nombre d'identificateurs communiqués est donc de $3(n + n_d)$. □

4.4.2.2 Ordonnancement pseudo-statique du graphe

Nous décrivons dans cette section un modèle d'exécution d'Athapascan-1 valable seulement pour la classe des programmes Athapascan-1 où seule la tâche racine crée d'autres tâches (les autres créations de tâches sont alors dégénérées en exécution séquentielle). Un ordonnancement statique peut alors être calculé à partir du graphe généré après l'exécution de la tâche racine. Cet ordonnancement fourni un placement et un ordre d'exécution de toutes les tâches du graphe. Ainsi le placement des tâches peut être connu par tous les processeurs en début d'exécution. Nous verrons que cette informations permet alors de supprimer totalement le surcoût induit par la gestion de la mémoire partagée distribuée.

Ce modèle d'exécution est surtout adapté aux applications pour lesquelles le graphe de flot de données (et donc le parallélisme de l'application) peut être décrit de manière fine à partir de l'exécution d'une petite partie du programme sur les données en entrée. Notons cependant qu'il peut être facilement étendu aux applications dans lesquelles le parallélisme est décrit par phases (succession de phases de description du parallélisme séparées par des phases d'exécution).

L'exécution du programme est alors divisée en quatre étapes distinctes :

1. Exécution de la tâche racine t_1, qui va générer le graphe de flot de données comme décrit dans la section 4.3.1 page 59.

2. Calcul d'un ordonnancement statique de ce graphe G, fournissant un placement des tâches sur les m processeurs et un temps d'exécution du graphe $\omega_m(G)$.

3. Diffusion de ce graphe et du placement des tâches sur les processeurs. Une copie du graphe est alors présente sur chaque processeur.

4. Interprétation du graphe et exécution des tâches de ce graphe.

La dernière étape utilise activement le fait que le graphe est dupliqué sur chaque processeur. La figure 4.8 montre un exemple simple de l'exécution sur deux processeurs

d'un graphe contenant 5 tâches et un objet partagé. Sur le processeur 1, la tâche t_1 est prête mais comme elle a été placée sur le processeur 2, elle est retirée du graphe sans être exécutée. Le flot de données arrive alors sur le nœud a_2 qui attend la version de l'objet modifiée par t_1 avant de devenir prêt. Sur le processeur 2, la tâche t_1 est exécutée puis retirée du graphe. le flot de données arrive alors sur le nœud a_2 qui devient prêt. Il est alors possible en regardant les sites d'exécution des tâches branchées sur a_2 de voir que le processeur 1 à besoin de la version de l'objet partagé pour exécuter t_3. Le processeur 2 peut alors immédiatement après la terminaison de t_1 envoyer les valeurs de la version de l'objet sur le processeur 1. Le même mécanisme est appliqué lors de la terminaison de t_4. Le processeur 1 peut alors immédiatement envoyer au processeur 2 la valeur de la version de l'objet.

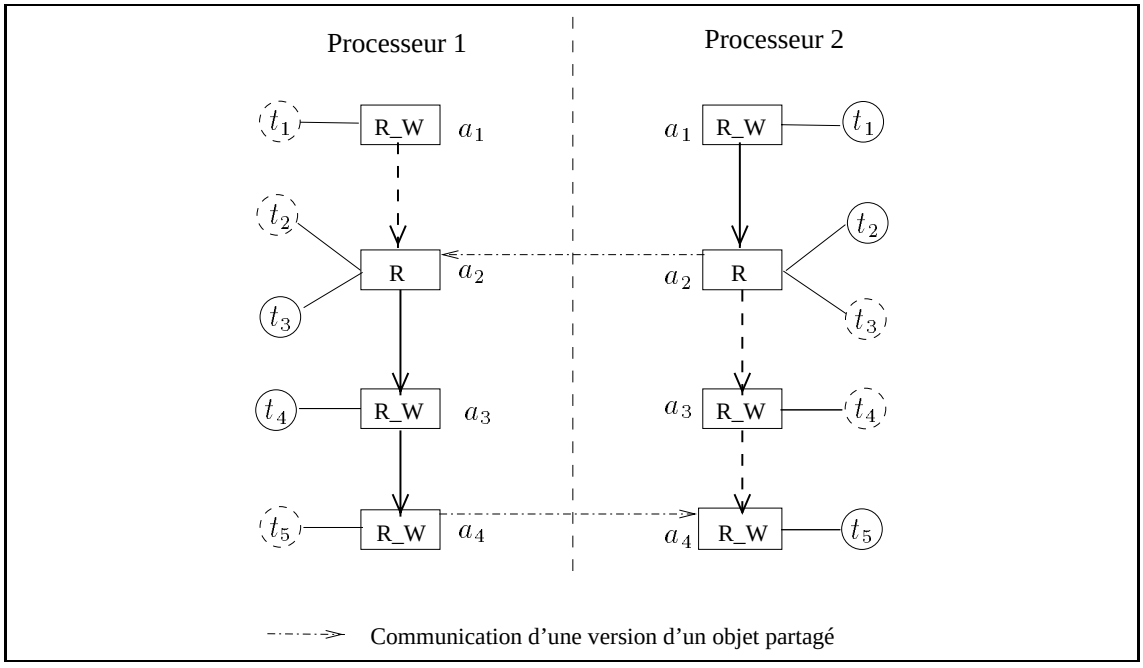

Figure 4.8 – *Exécution d'un graphe sur deux processeurs. Les cercles en pointillés représentent les tâches qui ont été placées sur un autre processeur.*

En suivant ce modèle d'exécution, on obtient alors la borne suivant sur le temps d'exécution :

Proposition 5 *Le temps d'exécution sur m processeurs d'une machine à mémoire distribuée, d'un programme Athapascan-1 dans lequel seule la tâche racine crée d'autres tâches est bornée par :*

$$T_{exec} \leq \omega_m(G) + \rho_m(G) + O(n + n_d) + D(n + n_d)$$

où G est le graphe généré lors de l'exécution de cette tâche racine, $D(n + n_d)$ est le temps de diffusion de G sur chacun des processeurs, $\omega_m(G)$ est le temps d'exécution fourni par un ordonnancement statique du graphe G et $\rho_m(G)$ est le coût de calcul de cet ordonnancement.

Preuve. Le coût de création du graphe est $O(n + n_d)$ d'après la proposition 1 page 60. Le coût de la première étape est alors $\mu(t_0) + O(n + n_d)$ où $\mu(t_0)$ est le temps d'exécution de la tâche racine dans lequel le coût des créations de tâches est ignoré. Le coût de la seconde et de la troisième étape est par définition $\rho_m(G) + D(n + n_d)$. Le coût de la quatrième étape est borné par le temps d'exécution fourni par l'ordonnancement statique $\omega_m(G)$, moins le coût de l'exécution de la tâche racine $\mu(t_0)$ déjà exécutée lors de la première étape, plus le surcoût lié à la réalisation de l'ordonnancement ; ce surcoût est borné par $O(n + n_d)$ (proposition 2 page 63). □

Une optimisation peut être obtenue en dupliquant l'exécution des deux premières étapes sur tous les processeurs. Le graphe ainsi que le placement des tâches sont alors connus de tous les processeurs. Ce nouveau schéma d'exécution n'est bien sûr possible que si l'algorithme d'ordonnancement statique est déterministe.

Corollaire 1 *Le temps d'exécution sur m processeurs d'une machine à mémoire distribuée, d'un programme Athapascan-1 dans lequel seule la tâche racine crée d'autres tâches est bornée par :*

$$T_{exec} \leq \omega_m(G) + \rho(G) + O(n + n_d).$$

où G est le graphe généré lors de l'exécution de cette tâche racine, $\omega_m(G)$ est le temps d'exécution fourni par un ordonnancement statique déterministe du graphe G et $\rho_m(G)$ est le coût de calcul de cet ordonnancement.

4.5 Conclusion

Nous avons montré dans ce chapitre le modèle général d'exécution d'un programme Athapascan-1. Ce modèle est basé sur la construction et l'interprétation à la volée du graphe qui représente les tâches à exécuter ainsi que le flot de données entre ces tâches.

Trois variantes de ce modèle général d'exécution ont ensuite été proposées suivant le type de machine parallèle et le type d'ordonnancement utilisé. Nous verrons dans le chapitre 6 que la bibliothèque C++ Athapascan-1 suit étroitement ces modèles d'exécutions.

Une analyse de complexité a de plus été réalisée pour chacun de ces modèles d'exécutions. Cet analyse montre que le surcoût lié à la génération et l'interprétation du graphe de flot de données ainsi que le surcoût lié à la gestion des objets partagées est borné par $O(n + n_d)$ où n est le nombre de tâches créées et n_d est le nombre total de références contraintes passées en paramètre des tâches. Ce coût peut donc être facilement amorti par

le choix d'une granularité des tâches adaptée (les évaluations expérimentales réalisées dans la section 6.5 du chapitre 6 consacré à l'implantation du modèle de programmation permettrons de préciser la constante présente dans le $O(n + n_d)$ et donc la granularité «minimale» des tâches permettant d'amortir ce surcoût).

Ce chapitre à également permis d'illustrer la capacité qu'a Athapascan-1, de par son modèle de programmation[3] et d'exécution[4] à pouvoir utiliser divers algorithmes d'ordonnancement, aussi bien dynamique que statique. Comme le montre les analyses de complexité des modèles d'executions présentés dans ce chapitre, les performances d'une application Athapascan-1 seront alors principalement tributaire de l'algorithme d'ordonnancement utilisé.

3. Les applications supportées vont des applications dont le graphe est connu en début d'exécution et annotée jusqu'aux applications dont le graphe n'est découvert que progressivement et dont les coûts sont inconnus.

4. Le modèle d'exécution permet à l'algorithme d'ordonnancement d'accéder au graphe de flot de données.

5

Algorithmes d'ordonnancement d'un graphe de flot de données

5.1 Introduction

L'exécution, sur une architecture parallèle donnée, d'une application parallèle représentée sous la forme d'un graphe de flot de données, mobilise différentes ressources : des ressources de calcul, des ressources de mémoire et, lorsque l'architecture est distribuée, des ressources de communication entre les nœuds. Le problème de l'ordonnancement d'une application parallèle consiste alors à contrôler l'exécution du programme afin d'optimiser l'utilisation des différentes ressources disponibles en fonction de certains critères.

Bien que de nombreux critères puissent être considérés, nous ne nous intéresserons ici qu'au problème de la minimisation du temps total d'exécution, qui est le critère le plus généralement considéré. De plus, nous supposerons que les ressources mémoire sont infinies. Une étude, dans le cadre d'Athapascan-1, de l'ordonnancement d'un graphe de flot de données prenant en compte, outre le temps total d'exécution, la mémoire nécessaire à l'exécution, est réalisée dans [48].

Lorsque le graphe de flot de données est connu avant l'exécution (*i.e.* la structure du graphe de flot de données, les coûts d'exécution des tâches et le volume de données transitant entre les tâches sont connus), un ordonnancement de ce graphe de flot de données peut être calculé avant l'exécution : on parle dans ce cas d'ordonnancement statique. Le calcul de cet ordonnancement est généralement basé sur une simulation, sur un modèle de machine parallèle, de l'exécution du graphe de flot de données qui modélise l'exécution de l'application. La performance effective de l'ordonnancement obtenu sur la machine réelle est alors fonction non seulement de l'exactitude des coûts fournis sur le graphe de

flot de données mais aussi de la pertinence du modèle de machine utilisé par rapport à la machine réelle.

À l'opposé des ordonnancements statiques, les ordonnancements à la volée (ou ordonnancements dynamiques) ne supposent qu'une connaissance partielle du graphe de flot de données (voir aucune) et sont moins (voir pas du tout) tributaires d'un modèle de machine puisque l'ordonnancement est directement calculé à la volée pendant l'exécution.

D'autre techniques d'ordonnancement viennent s'insérer entre ces deux techniques d'ordonnancements extrêmes, statique et à la volée. On parle d'ordonnancement mixte ou hybride [19, 50]. Un tel ordonnancement est particulièrement adapté lorsque l'exécution est composée d'alternance de phase de construction du graphe suivies de phase d'exécution. Différentes applications de ce type, appelées semi-prévisibles, ont été étudiées dans le cadre du projet inter-PRC Stratagème [70]. Par exemple, lorsque le graphe de flot de données est semi-prévisible, un ordonnancement statique peut être utilisé après chaque phase de construction du graphe. Dans ce cadre, nous proposerons à la section 5.5 page 86 un algorithme d'ordonnancement à la volée prenant en compte la connaissance du coût des tâches prêtes et du volume de données qu'elles accèdent en entrées.

Le chapitre est organisé de la manière suivante. Tout d'abord, nous introduisons les différents modèles de machines qui seront utilisés pour l'analyse des performances des algorithmes d'ordonnancement mais également par la plupart des algorithmes d'ordonnancement statiques. Nous caractérisons ensuite le graphe de flot de données utilisé pour représenter l'application. Dans une troisième partie, nous abordons différents algorithmes d'ordonnancement statiques et dynamiques pour des machines à mémoire partagée ou distribuée. Ces algorithmes d'ordonnancement seront finalement appliqués sur le modéle d'exécution d'Athapascan-1 introduit dans le chapitre précédent ; ils permettront de donner, d'un point de vue théorique, des garanties sur le temps d'exécution d'un programme écrit en Athapascan-1.

5.2 Modélisation de la machine parallèle

Aborder l'analyse de l'exécution d'un programme parallèle sur une machine réelle est une tâche très difficile voire impossible de par la complexité du comportement des machines actuelles. Ceci est encore plus vrai pour les machines parallèles et distribuées.

La solution classique est alors de recourir à un modèle d'exécution [88]. Un tel modèle définit une machine abstraite ainsi que le comportement de cette machine lors de l'exécution d'un programme. Le modèle d'exécution doit être suffisamment simple pour rendre abordable l'analyse théorique d'une exécution. Il doit également être réaliste pour rendre les prédictions aussi proches que possible d'une exécution réelle. Il doit de plus représenter une large classe de machines pour rendre l'analyse théorique portable. Ces trois caractéristiques sont contradictoires et définir un modèle d'exécution passe par des compromis.

Plusieurs modèles d'exécution ont ainsi été définis pour représenter l'exécution d'un programme sur une machine parallèle. Ils peuvent être séparés en deux classes : les modèles à mémoire partagée et les modèles à mémoire distribuée.

5.2.1 Modèles d'exécution à mémoire partagée

Dans ces modèles, les processeurs communiquent entre eux par l'intermédiaire d'une mémoire partagée. Ces modèles peuvent représenter les machines à mémoire partagée mais également les machines à mémoire distribuée dans lesquels la mémoire est virtuellement partagée.

Le plus connu de ces modèles est le modèle PRAM (*Parallel Random Access Machine*) [41]. Une machine PRAM est composée de p processeurs et d'une mémoire partagée. Les processeurs sont synchronisés entre eux. À chaque top d'exécution, chaque processeur peut lire deux données, réaliser une opération et écrire une donnée. Ainsi, une opération arithmétique élémentaire peut être réalisée à chaque top d'exécution. Plusieurs variantes et extensions de ce modèle ont été proposées permettant de prendre en compte les conflits d'accès à la mémoire partagée. La latence et les délais d'accès à cette mémoire peuvent être considérés : par exemple le modèle Local-PRAM [78] associe à chaque processeur une mémoire locale. On peut citer par exemple le modèle QRQW (*Queue-Read Queue-Write*) [57].

Différents travaux ont étudié la simulation du modèle d'exécution PRAM sur une machine à mémoire distribuée. Dans [72], une telle simulation est basée sur des fonctions de hachage permettant de distribuer les cellules de la mémoire partagée sur les différents modules mémoires. Le délai h de la simulation est alors le temps nécessaire pour accéder un mot mémoire (une cellule), ce temps incluant d'une part le coût de l'évaluation d'une fonction de hachage universelle et d'autre part le coût lié à la latence du réseau. Dans [72], une simulation de délai $O(\log\log p \log^* p)$ est donnée sur une machine distribuée à p processeurs avec un réseau d'interconnexion complet ; les processeurs étant supposés pouvoir émettre et recevoir un mot mémoire par cycle avec une latence d'un cycle.

Pour obtenir une simulation optimale offrant un accès en un cycle de la PRAM simulée, la technique classique de virtualisation (*Parallel slackness*) est utilisée [122, 72]. Elle consiste à simuler q processeurs virtuels sur les p processeurs de la machine à mémoire distribuée pour recouvrir les délais d'accès à la mémoire partagée simulée. Ainsi, pour obtenir une simulation optimale à partir de la simulation précédente, on pose $q = \log\log p \log^* p$.

5.2.2 Modèles d'exécution à mémoire distribuée

Dans ces modèles, les processeurs communiquent entre eux par l'intermédiaire d'un réseau de communication. Ces modèles d'exécution cherchent alors principalement à mo-

déliser les communications entre processeurs. On parle souvent de modèle de communication plutôt que de modèle d'exécution.

Le modèle délai est, jusqu'à récemment, le modèle de communication le plus utilisé pour le problème de l'ordonnancement d'applications représentées par un graphe de tâches [106, 97, 69, 128]. Dans ce modèle le réseau de communication est caractérisé par une latence ou délai, correspondant au temps τ nécessaire à la transition d'un mot entre deux processeurs. Nous utiliserons principalement dans ce chapitre le modèle de Hwang et al [69] dans lequel le temps de communication d'un message de taille n entre deux processeurs prend τn unités de temps.

Ce modèle délai est très imparfait puisqu'il considère une bande passante infinie en entrée et en sortie de chaque processeur : un processeur peut en effet envoyer ou recevoir un nombre illimité de messages à un instant donné. Pour remédier à ce problème, des modèles plus proches des machines réelles ont été proposés. On peut citer parmi ceux-ci le modèle BSP proposé par Valiant [122] et le modèle LogP proposé par Culler et al. [27, 26]. Cependant, ces modèles sont beaucoup plus complexes à utiliser que les modèles délais, tant du point de vue de la construction d'algorithmes d'ordonnancement statiques que pour l'analyse théorique d'une exécution. C'est pourquoi très peu d'algorithmes d'ordonnancement ont été proposés sur le modèle LogP [90, 12, 77] ; de plus, la plupart de ces algorithmes sont restreints à des classes de graphes de tâches spécifiques comme les arbres, ce qui limite leur intérêt pour des applications réelles.

5.3 Modélisation du programme parallèle : précédence versus flot de données

Le modèle de programme le plus couramment employé par les algorithmes d'ordonnancement est le graphe de précédence $G = (V, E)$. Les nœuds de ce graphe sont des tâches (une tâche est une séquence d'instructions élémentaires) et les arcs de ce graphe représentent les précédences entre ces tâches. Généralement, une précédence (u, v) indique que l'exécution de la tâche v nécessite des résultats produits par la tâche u. Les tâches de ce graphe peuvent être annotées par leur coût d'exécution. De même les arcs peuvent être annotés par une valeur ; la valeur de l'arc (u, v) peut correspondre par exemple selon le modèle de machine considéré au volume de données communiqué entre u et v, ou encore au délai nécessaire pour communiquer ces données entre u et v.

Ce graphe de précédence, une fois annoté par des coûts sur les arcs, peut être considéré comme un graphe de flot de données. Cependant, il ne permet pas de représenter complètement les diffusions et les réductions de données entre tâches. Par exemple, une diffusion d'une donnée produite par la tâche u vers les tâches $v_1, \ldots, v_l$ est représenté par l arcs. Même si les tâches $v_1, \ldots, v_l$ sont exécutées sur un même nœud, ce graphe modélise toujours l communications de la donnée produite par u. Or cela n'est pas néces-

sairement le cas si le modèle d'exécution utilise un mécanisme de cache des données sur chaque processeur comme le fait par exemple le modèle d'exécution d'Athapascan-1.

Au contraire, le graphe de flot de données présenté dans la section 4.2 page 56 permet de représenter plus précisément les opérations de diffusion et de réduction des données. Il est donc plus précis pour modéliser un programme Athapascan-1.

Par ailleurs, le graphe de précédence se déduit directement du graphe de flot de données en considérant le sous graphe constitué des noeuds tâches et des arcs entre les noeuds tâches t_i et t_j si un ou plusieurs flot de données vont de t_i à t_j.

5.3.1 Grandeurs caractéristiques d'un graphe de flot de données

Les notations suivantes seront utilisées dans la suite du chapitre pour caractériser le flot de données G défini dans la section 4.2 page 56 et qui modélise l'exécution d'un programme parallèle [110] :

n : désigne le nombre de tâches créées, donc le nombre de tâches du graphe.

n_d : désigne le nombre total d'objets partagés passés en paramètre de tâches, autrement dit la somme des degrés des nœuds tâches du graphe (en ne comptant pas les arcs représentant les créations de tâches).

T_1 : désigne le temps séquentiel d'exécution des tâches du graphe, c'est-à-dire la somme des coûts d'exécution de chaque tâche.

T_∞ : désigne le temps d'exécution d'un plus long chemin du graphe de flot de données dans lequel le coût des communications est ignoré. C'est aussi une borne inférieure sur le temps d'exécution du graphe, sur un nombre infini de processeurs en ignorant les communications (modèle PRAM).

C_1 : désigne la somme des volumes de données accédées par chacune des tâches.

C_{max} : désigne le nombre maximum de données communiquées sur un chemin du graphe.

C_{max^*} : désigne le nombre maximum de données communiquées sur un chemin du graphe G' identique au graphe G si ce n'est que pour chaque tâche, le volume de données entrant par un arc dans cette tâche est aligné sur le plus grand volume de données entrant dans cette tâche. Cette caractéristique du graphe sera utilisée à la place de C_{max} dans les algorithmes d'ordonnancement à la volée.

5.3.2 Degré de connaissance du graphe de flot de données à l'exécution

À un instant donné de l'exécution d'une application, le système chargé de l'exécution dispose du graphe de flot de données décrivant les tâches créées et non encore exécutées. Le graphe représentant l'exécution *a posteriori* n'est donc pas entièrement connu avant la fin de l'exécution. Ainsi, suivant le type d'application, une partie plus ou moins grande de ce graphe peut être connue à un instant donné.

Par ailleurs, le graphe connu par le système à un instant donné de l'exécution peut être annoté par des informations fournies par le programmeur. Par exemple, dans le cas d'un programme Athapascan-1, nous avons vu dans la section 3.1.6 page 46 que l'utilisateur pouvait indiquer, lors de la création d'une tâche, une estimation de son coût ou bien une fonction permettant d'évaluer son coût en fonction de la taille de ces paramètres.

Le degré de connaissance du graphe, à un instant donné de l'exécution, est alors très variable :

1. La structure du graphe :
   - *Structure connue* : Si seule la tâche racine génère d'autres tâches, alors la structure du graphe de flot de données est entièrement connue à la fin de l'exécution de la tâche racine. C'est par exemple le cas du programme d'élimination de Gauss par colonnes vu dans la section 3.2.2 page 49.
   - *Structure inconnue* : L'ensemble des tâches est créé au fur et à mesure du déroulement de l'exécution. C'est par exemple le cas d'un programme créant récursivement des tâches comme l'exemple du calcul du $n^{ème}$ terme de la suite de Fibonacci vu dans la section 3.2.1 page 48.
   - *Structure caractérisée* : L'utilisateur fournit, en début d'exécution, une information caractérisant la structure du graphe. La structure peut par exemple être un arbre, un arbre binaire, un graphe de type série-parallèle.
2. Une estimation du coût des tâches :
   - *Coût connu à la création* : Une estimation du coût des tâches est donnée lors de leurs créations.
   - *Coût connu lorsque la tâche est prête* : Une estimation du coût des tâches est obtenue lorsqu'elles deviennent prêtes, via un calcul sur leurs paramètres effectifs.
   - *Coût inconnu* : Aucune estimation du coût des tâches n'est donnée.
3. Une estimation de la taille des objets partagés :
   - *Taille connue à la création de l'objet* : Une estimation de la taille des objets partagés est donnée lors de leur création. Cette taille étant supposée constante

pendant l'exécution ; *i.e.* les tâches modifiant un objet partagé ne modifient pas sa taille.

– *Taille connue après chaque modification de l'objet* : Une estimation de la taille est donnée pour chaque valeur prise par un objet partagé. Une telle estimation peut par exemple être fournie lors de chaque modification de l'objet partagé.

– *Taille inconnue* : La taille d'une valeur d'un objet partagé ne peut pas être estimée (où plutôt, le programmeur n'a pas fourni cette estimation lors de la modification de l'objet).

Ce degré d'information connue à l'exécution sur le graphe est très variable et dépend du programme ; nous allons cependant voir que cette connaissance influence la manière dont le programme pourra être ordonnancé.

5.4 Algorithmes d'ordonnancement

Différentes classes d'algorithmes peuvent être utilisées pour calculer un ordonnancement de tâches. La première consiste à chercher une solution optimale : le problème étant difficile (NP-dur même pour des instances simples lorsque les communications sont prises en compte [24]), on procède par énumération (généralement par séparation-évaluation). La seconde consiste à calculer, en temps raisonnable (en temps polynômial en fonction de la taille du graphe), une approximation de la solution optimale. Cette approximation peut être garantie, c'est-à-dire que le temps d'exécution de cette solution, sur un modèle d'exécution donné, est borné à un facteur près du temps d'exécution de l'ordonnancement optimal sur ce même modèle d'exécution. Enfin, de par les incertitudes liées à la fois au modèle d'exécution et à la connaissance du graphe, une troisième classe consiste à considérer des méthodes heuristiques adaptées à des cadres restreints d'utilisation.

Nous présentons dans la suite de ce chapitre quelques un de ces algorithmes d'ordonnancement, l'objectif étant ensuite de les appliquer pour l'ordonnancement d'un programme Athapascan-1. Cependant, les algorithmes d'ordonnancement de complexité non polynômiale semblent difficilement utilisables dans le cadre d'un modèle de programmation comme Athapascan-1 où l'ordonnancement est calculé à l'exécution et où le parallélisme exprimé à un grain fin conduit à un graphe de flot de données de taille importante. Nous ne considérerons donc dans la suite de ce chapitre que les algorithmes calculant un ordonnancement au plus en temps polynômial.

5.4.1 Ordonnancement sans délai de communication

Parmi les algorithmes d'ordonnancement garantis lorsque les communications ne sont pas prises en compte, le plus connu est sans doute l'algorithme proposé par Graham [58]. Cet algorithme est appelé « glouton » : dès qu'un processeur devient inactif, une tâche

prête lui est allouée s'il en existe. Une liste des tâches prêtes doit donc être maintenue à tout instant de l'exécution. Pour cette raison, un tel algorithme est couramment appelé algorithme de liste.

Plusieurs stratégies peuvent être considérées pour le choix de la tâche à allouer parmi les tâches prêtes. Cependant, Graham a établi un résultat valable pour tous les algorithmes de listes :

Théorème 1 *[58] Tout algorithme d'ordonnancement de liste, appliqué sur un graphe de tâche avec précédences et sur une machine à m processeurs identiques sans délai de communication, fournit un temps d'exécution des tâches borné par*

$$T_m \leq \frac{T_1}{m} + (1 - \frac{1}{m})T_\infty$$

L'ordonnancement est alors à un facteur $2 - \frac{1}{m}$ de l'optimal.

Les algorithmes de liste ont l'avantage d'être directement utilisables pour ordonnancer à la volée un graphe quelconque puisque seule la liste des tâches prêtes et les processeurs inactifs doivent être connus à un instant donné de l'exécution. De plus, Graham montre qu'aucun algorithme à la volée, déterministe et non préemptif, ne peut améliorer la borne ainsi obtenue dans le cas où le coût des tâches est inconnu.

Dans le cas où les machines ne sont plus identiques, cette borne ne tient plus. Shmoys et *al.* ont proposé [66, 82, 117] divers algorithmes d'ordonnancements statiques et dynamiques d'approximation garantie pour des machines non identiques[1]. Par exemple, une technique proposée pour l'ordonnancement de tâches indépendantes sur machine uniforme est basée sur un algorithme de décision relâché. Cet algorithme prend en entrée un temps d'exécution des tâches d et soit affirme qu'un ordonnancement des tâches en temps d n'est pas réalisable soit retourne un ordonnancement des tâches en temps au plus $2d$. Un ordonnancement des tâches à un facteur 2 de l'optimal est alors obtenu en réalisant, grâce à cet algorithme, une recherche dichotomique du temps d'exécution d pour lequel l'algorithme de décision ne retourne pas de solution pour un temps d'exécution $d - 1$ et une solution en temps au plus $2d$ si le temps d'exécution est d. L'ordonnancement obtenu est alors clairement à un facteur 2 de l'optimal. Une adaptation de cet algorithme, pour le problème de l'ordonnancement dynamique de tâches indépendantes sur des machines uniforme, est également fournie dans [117] : un ordonnancement est ainsi construit avec une performance garantie à un facteur $8(\log(2R))$ de l'optimal, où R est le rapport des vitesses entre le plus rapide et le plus lent des processeurs.

1. Les machines peuvent alors être uniformes ou non uniformes. Dans le premier cas la vitesse d'un nœud de la machine parallèle est constante pour toutes les tâches alors que dans le second cas cette vitesse dépend de la tâche exécutée.

5.4.2 Ordonnancement avec délai de communication

La prise en compte des délais de communication dans le modèle d'exécution rajoute une difficulté importante au problème de l'ordonnancement.

Une première approche consiste à considérer comme unité d'ordonnancement à la fois la tâche et les délais de communication pour ramener les données nécessaires à l'exécution de cette tâche et à appliquer alors un algorithme glouton sur ces unités d'ordonnancement. Ainsi lorsqu'un processeur devient inactif, une tâche prête lui est allouée. Les données nécessaires à l'exécution de la tâche sont alors communiquées à partir de l'instant où la tâche est allouée, celle ci ne commençant son exécution qu'après que ces données soit arrivées localement.

Le théorème suivant borne le temps d'exécution de l'ordonnancement des tâches ainsi obtenu :

Théorème 2 *En négligant le surcoût lié au calcul et à la réalisation de l'ordonnancement, l'algorithme précédant qui consiste à allouer de façon gloutonne une tâche prête sur un processeur disponible calcule un ordonnancement sur m processeurs identiques borné en temps par :*

$$T_m \leq \frac{T_1}{m} + (1 - \frac{1}{m})T_\infty + \tau \left(\frac{C_1}{m} + \left(1 - \frac{1}{m}\right) C_{max^*} \right)$$

où τ est le temps de transition d'un mot mémoire entre deux processeurs.

Preuve. La preuve est identique à celle de Graham. Soit $\tilde{G}$ le graphe de tâches de structure identique au graphe G et annoté de la manière suivante : le temps d'exécution d'une tâche t'_i de $\tilde{G}$ est égal à la somme du temps d'exécution de la tâche correspondante t_i de G et du temps de communication des données nécessaires à l'exécution de cette tâche t_i. Les temps de communication associé aux arcs de $\tilde{G}$ sont nuls. Alors l'ordonnancement du graphe G par l'algorithme est par définition l'ordonnancement du graphe $\tilde{G}$ par l'algorithme glouton de Graham.

De par la définition du temps d'exécution des tâches de $\tilde{G}$, le temps d'exécution séquentiel $T_1(\tilde{G})$ du graphe $\tilde{G}$ est égale à $T_1(G) + \tau C_1(G)$ et de même on a $T_\infty(\tilde{G}) = T_\infty(G) + \tau C_{max^*}(G)$. En appliquant la borne du théorème 1 page 82 sur $\tilde{G}$, on borne alors le temps d'exécution de $\tilde{G}$ par $\frac{T_1+\tau C_1}{m} + (1 - \frac{1}{m})(T_\infty + \tau C_{max^*})$, ce qui termine la démonstration. □

Comme pour l'algorithme glouton proposé par Graham, cet algorithme peut directement être utilisé pour ordonnancer un graphe quelconque à la volée puisque seule la liste des tâches prêtes et les processeurs inactifs doivent être connus à tout instant.

Le surcoût induit par les communications dans l'ordonnancement produit par cet algorithme est $\frac{\tau C_1}{m} + (1 - \frac{1}{m})\tau C_{max^*}$. Nous présentons dans la suite d'autres algorithmes d'ordonnancement qui permettent de réduire ce surcoût induit par les communications. La réduction est obtenue soit en recouvrant des communications par des calculs soit en minimisant les communications.

5.4.2.1 Recouvrement des communications par des calculs

La première technique pour réduire le surcoût induit par les communications consiste à recouvrir les délais de communications par des calculs.

ETF et ERT. Parmi les algorithmes utilisant cette approche, on peut citer ETF *Earliest Task First* [69], et ERT *Earliest Ready First* [81] qui sont deux algorithmes d'ordonnancement statiques de type glouton.

Le principe de ces deux algorithmes est d'ordonnancer, parmi l'ensemble des tâches prêtes à un instant donné $R = \{r_1, \ldots, r_k\}$ (une tâche est prête lorsque tous ses prédécesseurs sont terminés), la tâche qui pourra commencer son exécution au plus tôt sur l'un des processeurs disponibles (*i.e.* inactifs), en considérant les délais de communications. Soit $P = \{p_1, \ldots, p_m\}$ l'ensemble des m processeurs de la machine. L'algorithme évalue donc, pour chaque couple $(r_i, p_j) \in (R, P)$, la date de démarrage de la tâche r_i sur le processeur p_j (en considérant les délais de communications ainsi que la date de disponibilité de p_j). Le couple (r_i, p_j) obtenant la plus petite date de démarrage est alors choisi conduisant à l'ordonnancement de la tâche r_i sur le processeur p_j.

Ces deux algorithmes calculent un ordonnancement statique : donc les communications des données générées par une tâche u peuvent démarrer dès la terminaison de u puisque le placement des tâches successeurs de u est connu avant l'exécution. Cette propriété est considérée lors du calcul de l'ordonnancement ainsi que dans la preuve sur la borne du théorème 3. Ces algorithmes ne peuvent donc pas être utilisés à la volée.

La borne suivante est valable pour tout algorithme d'ordonnancement statique assignant de façon gloutonne une tâche prête sur le processeur qui démarrera son exécution au plus tôt. Cette borne est donc valable pour ETF et ERT. Par ailleurs, elle suppose comme modèle de communication le modèle de Hwang et *al.* introduit dans la section 5.2.2.

Théorème 3 *[86] Tout algorithme d'ordonnancement statique, consistant à assigner de façon gloutonne une tâche prête sur le processeur qui démarrera son exécution au plus tôt, appliqué sur un graphe de tâche avec délais de communication et sur une machine distribuée à m processeurs identiques, fournit un temps d'exécution des tâches borné par :* $T_m \leq \frac{T_1}{m} + (1 - \frac{1}{m})T_\infty + \tau C_{max}$.

Par ailleurs, la complexité du calcul de l'ordonnancement statique par ETF ou ERT est bornée par $O(mn^2)$ [69].

La technique de virtualisation des processeurs [122] (*parallel slackness*), consistant à simuler q processeurs virtuels sur p processeurs physiques permet également de recouvrir les délais de communication par les calculs. D'un point de vue pratique, les techniques de multiprogrammation légère (*multithreading*) permettent de réaliser une telle virtualisation.

5.4.2.2 Réduction des communications

Une autre manière de réduire le surcoût induit par les délais de communication est naturellement de chercher à réduire les communications effectuées lors de l'exécution.

Regroupement de tâches Une première technique pour réduire les communications consiste à effectuer un regroupement préliminaire des tâches. Ce regroupement des tâches est généralement obtenu en calculant un ordonnancement des tâches sur un nombre non borné de processeurs ; cette hypothèse d'un nombre non borné de processeurs permet de se focaliser sur la minimisation des coûts dus aux délais de communication. Les groupes ainsi obtenus (un groupe correspondant aux tâches allouées sur un même processeur) sont ensuite répartis sur les processeurs disponibles. Intuitivement, cette approche cherche à regrouper sur un même processeur les tâches qui communiquent beaucoup entre elles. Pyrros [128] utilise par exemple cette approche grâce à l'algorithme d'ordonnancement statique DSC *Dominant Sequence Cluster*) [127] qui est utilisé pour réaliser le regroupement des tâches.

Les techniques de partitionnement de graphe de dépendance de données[2] [63, 74, 98] peuvent être incluses dans la catégorie des algorithmes cherchant à réduire les communications par regroupement de tâches. Cependant cette technique n'est pas directement applicable au cadre de l'ordonnancement de graphe de précédence ou de flot de données. Elle ne peut intervenir qu'après une première phase de regroupement de tâches ramenant le problème de l'ordonnancement temporel et spatial des calculs sur les ressources (ordonnancement de graphe de précédence ou de flot de donnée) en un problème d'ordonnancement spatial (ordonnancement de graphe de dépendance).

Duplication de tâche La duplication de tâche, c'est-à-dire l'exécution d'une même tâche sur plusieurs processeurs, permet également de réduire les communications nécessaires à l'exécution. Papadimitriou et Yannakakis proposent dans [97] un algorithme statique basé sur ce principe. Cet algorithme calcule un ordonnancement statique à un facteur 2 de l'optimal sur un nombre non borné de processeurs identiques. Pour le problème de l'ordonnancement avec petit temps de communication[3] Colin et Chrétienne ont proposé dans [25], toujours sur un nombre non borné de processeurs, un algorithme optimal.

Cependant replier l'ordonnancement obtenu sur un nombre de processeurs fini comme pour les algorithme de regroupements de tâches vus précédemment ne permet pas d'aboutir à un ordonnancement efficace de par le travail redondant généré par les duplications

2. Nous rappelons qu'un graphe de dépendance est un graphe non orienté dont les nœuds correspondant à des tâches communicantes, et les arêtes représentent les communications entre ces tâches. Ce graphe ne définit donc pas de précédence entre les tâches.

3. Un graphe est à petits temps de communications lorsque le rapport entre le plus grand temps de communication d'un arc du graphe par le plus petit temps de calcul d'une tâche du graphe est inférieur à un.

de tâches. Bien qu'intéressant d'un point de vue théorique, ces algorithmes ne sont donc pas utilisables en pratique sur un nombre borné de processeur. Pour le problème de l'ordonnancement sur un nombre borné de processeur et à petit temps de communication, un algorithme est proposé par Rapine dans [105] conduisant à un ordonnancement à un facteur deux de l'optimal.

5.4.2.3 Ordonnancement à la volée avec délai de communication

Les algorithmes d'ordonnancement précédents avec délai de communication ont tous en commun de réaliser un ordonnancement statique. Ils supposent une connaissance complète du graphe de flot de données.

Le seul algorithme connu d'ordonnancement à la volée qui a des performances garanties sur un modèle avec délai de communication a été proposé par Deng et *al.* dans [29]. Cet algorithme est une adaptation de l'algorithme avec duplication des tâches proposé par Papadimitriou et Yannakakis dans [97]. Il prend en entrée la structure du graphe mais suppose le coût des tâches et les délais de communication inconnus. Cependant, comme l'algorithme [97], cet algorithme relâche la contrainte sur le nombre de processeurs : le nombre de processeurs utilisés pour l'exécution de l'ordonnancement est égal à la largeur du graphe de précédence. Bien qu'intéressant d'un point de vue théorique, il n'est donc pas utilisable en pratique.

5.5 Un algorithme d'ordonnancement à la volée avec délai de communication

Nous présentons ici un algorithme d'ordonnancement d'un graphe de précédence dans le modèle d'exécution de Hwang et *al.* On suppose donc qu'il n'y a pas de contention sur le réseau et qu'une communication est réalisée sans surcoût pour les processeurs impliqués.

Le graphe est constitué de n tâches notées $t_1, \ldots, t_n$, chacune avec un temps de calcul $\mu(t_i)$. Le volume de communication associé à un arc du graphe reliant entre eux deux tâches du graphe t_i et t_j est noté $\eta(t_i, t_j)$. L'ensemble des tâches qui sont des prédécesseurs immédiats de la tâche t_i est noté $pred(t_i)$.

La machine est supposée constituée de m processeurs identiques notés $p_i, \ldots, p_m$; le temps de communication entre deux tâches t_i et t_j exécutées respectivement sur les processeurs p_k et p_l est défini par :

$$C(t_i, t_j, p_k, p_l) = \begin{cases} \tau\eta(t_i, t_j), & \text{si } k \neq l \\ 0, & \text{si } k = l \end{cases}$$

Nous proposons un algorithme original d'ordonnancement à la volée avec prise en

compte des communications. Cet algorithme appelé « algorithme ERT à la volée » est présenté page 87.

Algorithme 3 : Algorithme d'ordonnancement ERT à la volée.

Notations

T	: Temps courant de l'algorithme.
R	: Liste des tâches prêtes et non allouées à l'instant T.
$DISPO(p_i)$	: Date à laquelle le processeur p_i sera disponible.
$ALLOC(t_i)$	: Processeur choisi pour exécuter la tâche t_i.

Étape 0 $T \leftarrow 0$,
pour tout $j = 1, \ldots m$, $DISPO(p_j) \leftarrow 0$.

Étape 1 Mettre à jour l'ensemble R des tâches prêtes et non allouées à l'instant T.

Étape 2 Pour chaque tâche t_i de l'ensemble R et pour l'ensemble des processeurs $p_j, j = 1, \ldots, m$, calculer

$$r(t_i, p_j) = max\{DISPO(p_j), T + \max_{t_k \in pred(t_i)}\{C(t_k, t_i, ALLOC(t_k), p_j)\}\}$$

qui correspond à la date de démarrage au plus tôt de la tâche t_i sur le processeur p_j lorsque les communications nécessaires à cette exécution sont démarrées à l'instant T.

Étape 3 Sélectionner alors une tâche t_i et un processeur p_j vérifiant

$$r(t_i, p_j) = \min_{t_k \in R, 1 \leq l \leq m} r(t_k, p_l)$$

On assigne alors la tâche t_i sur le processeur p_j ; on démarre les communications nécessaires à cette tâche et on met à jour les variables suivantes :

$$\begin{aligned} &ALLOC(t_i) \leftarrow p_j \\ &DISPO(p_i) \leftarrow r(t_i, p_j) + \mu(t_i) \\ &R \leftarrow R - \{t_i\} \end{aligned}$$

Si R est non vide alors retourner à l'étape 2.

Étape 4 Mise à jour de la date T correspondant au prochain moment où au moins une tâche deviendra prête. Si cette date n'existe pas alors l'exécution est terminée, sinon retour à l'étape 1.

Dans cet algorithme, les communications nécessaires à l'exécution d'une tâche ne

sont ordonnancées que lorsque cette tâche devient prête. La décision de placement d'une tâche est également prise lorsque cette tâche devient prête. Cet algorithme peut donc être exécuté à la volée.

En comparaison, l'algorithme ERT « classique » ordonnance toujours les communications au plus tôt, c'est-à-dire à l'instant où les données à communiquer sont générées. Ceci n'est possible que si l'on connaît à cet instant les processeurs qui auront besoin de ces données. Cela présuppose donc que le placement des tâches est connu à l'avance. Pour cette raison, cet algorithme ne peut pas être exécuté à la volée.

Théorème 4 *L'algorithme 3 «ERT à la volée» appliqué sur un graphe de précédence avec temps de communication G, calcule un ordonnancement sur m processeurs identiques borné en temps par :*

$$T_m \leq \frac{T_1}{m} + (1 - \frac{1}{m})T_\infty + \tau C_{max^*}$$

où C_{max^*} est le nombre maximum de données communiquées sur un chemin du graphe G' identique au graphe G si ce n'est que pour chaque tâche, le volume de données entrant par un arc dans cette tâche est aligné sur le plus grand volume de données entrant dans cette tâche.

$$C_{max^*} = \max_{(t_{i(1)},\ldots,t_{i(s)}) \textit{chaîne de } G} \left\{ \sum_{k=2}^{s} \max_{t_j \in pred(t_{i(k)})} \eta(t_j, t_{i(k)}) \right\}$$

Cette borne ne prend pas en compte le surcoût de la mise en œuvre de l'ordonnancement.

Preuve. La preuve, similaire à celle de Graham [58], est basée sur le recouvrement des temps d'inactivité par un chemin critique du graphe. Elle est fournie dans l'annexe A. □

La borne sur le temps d'exécution obtenu avec un ordonnancement calculé par l'algorithme «ERT à la volée» est donc identique à celle obtenue avec les algorithmes d'ordonnancements statiques ETF et ERT lorsque C_{max^*} est égale à C_{max}.

5.6 Application des algorithmes d'ordonnancement à l'exécution d'un programme Athapascan-1

Dans cette section, nous appliquons quelques-uns des algorithmes d'ordonnancement vus précédemment à l'exécution d'un programme Athapascan-1. Les différents modèles d'exécution d'Athapascan-1 introduits dans le chapitre 4 n'autorisent pas la duplication ou la migration de tâches ; nous ne considérerons donc ici que des algorithmes d'ordonnancement non préemptifs sans duplication de tâches.

5.6.1 Machine à mémoire partagée

Nous utilisons ici l'algorithme d'ordonnancement de Graham présenté dans la section 5.4.1 page 81 et le modèle d'exécution d'Athapascan-1 pour mémoire partagée, présenté dans la section 4.4.1 page 66. Nous obtenons alors la borne suivante sur le temps d'exécution d'un programme Athapascan-1, valable sur une machine PRAM :

Corollaire 2 *[35] Le temps d'exécution d'un programme Athapascan-1 sur une machine PRAM à mémoire partagée à m processeurs identiques est bornée par*

$$T_m \leq \frac{T_1}{m} + \left(1 - \frac{1}{m}\right) T_\infty + O(n + n_d)$$

Preuve. Elle découle directement de la proposition 3 page 67, du théorème 1 page 82 et du coût $O(n)$ de la gestion par l'ordonnanceur de la liste des tâches prêtes et de la liste des processeurs disponibles. □

5.6.2 Machine à mémoire distribuée : mémoire partagée simulée

Le résultat précédent, obtenu sur le modèle de machine PRAM peut facilement être étendu à un modèle de machine à mémoire partagée simulée [76]. La seule différence est alors le délai h intervenant pour tout accès d'un mot de la mémoire partagée, c'est-à-dire pour tout accès à un objet partagé par une tâche et pour tout accès au graphe de flot de données par le système chargé de l'exécution.

Corollaire 3 *[35] Le temps d'exécution d'un programme Athapascan-1, sur une machine simulant m processeurs identiques et accédant avec un délai h à une mémoire virtuellement partagée, est borné par :*

$$T_m \leq \frac{T_1}{m} + (1 - \frac{1}{m})T_\infty + h\left(\frac{C_1}{m} + (1 - \frac{1}{m})C_{max^*} + O(n + n_d)\right)$$

Preuve. La preuve est identique à celle utilisée pour démontrer le théorème 2 page 83. Au temps d'exécution de chaque tâche est ajouté le temps passé pour accéder les données nécessaires à cette tâche. L'ordonnancement réalisé est alors équivalent à l'ordonnancement glouton d'un graphe G' de temps d'exécution séquentielle $T_1(G') = T_1(G) + hC_1(G)$ et de chemin critique $T_\infty + hC_{max^*}$. □

5.6.3 Machine à mémoire distribuée : modèle de communication de Hwang et *al.*

Nous présentons maintenant trois bornes sur le temps d'exécution d'un programme Athapascan-1 valable sur machine à mémoire distribuée en utilisant le modèle de communication de Hwang et *al.*

5.6.3.1 Graphe inconnu, coût inconnu

En utilisant comme algorithme d'ordonnancement l'algorithme glouton proposée dans la section 5.4.2 page 83 et le modèle d'exécution centralisé d'un programme Athapascan-1, proposé dans la section 4.4.2.1 page 68, on obtient alors la borne suivante :

Corollaire 4 *Soit l'exécution d'un programme Athapascan-1 sur m processeurs identiques à mémoire distribuée, réalisée en utilisant l'algorithme d'ordonnancement glouton présenté dans la section 5.4.2 page 83 et le modèle d'exécution centralisé exposé dans la section 4.4.2.1 page 68. Le temps d'exécution est borné par :*

$$T_m \leq \frac{T_1}{m} + (1 - \frac{1}{m})T_\infty + \frac{\tau C_1}{m} + (1 - \frac{1}{m})\tau C_{max^*} + O(n + n_d) + 3\tau(n + n_d)$$

Preuve. Elle découle directement de la proposition 4 page 70, du théorème 2 page 83 et du coût $O(n)$ de la gestion par l'ordonnanceur de la liste des tâches prêtes et de la liste des processeurs disponibles. □

5.6.3.2 Graphe inconnu, coût connu

En utilisant comme ordonnancement l'algorithme 3 «ERT à la volée» proposé dans la section 5.5, et le modèle d'exécution centralisée d'un programme Athapascan-1, proposé dans la section 4.4.2.1, on obtient alors la borne suivante :

Corollaire 5 *Soit l'exécution sur m processeurs identiques, d'un programme Athapascan-1 dont le coût des tâches est connu lorsque elles deviennent prêtes et la taille des versions des objets partagés est connue lorsque ces versions sont disponibles. Soit G le graphe généré, alors le temps d'exécution est bornée par :*

$$T_m \leq \frac{T_1}{m} + (1 - \frac{1}{m})T_\infty + \tau C_{max^*} + O(mn^2) + O(n + n_d) + \tau(3n + 4n_d)$$

.

Preuve. Elle découle directement de la proposition 4 page 70, du théorème 4 page 88 et du coût $O(mn^2)$ nécessaire au calcul de l'ordonnancement par l'algorithme 3. □

5.6.3.3 Graphe connu, coût connu

En utilisant comme ordonnancement statique l'algorithme ETF, qui est déterministe et le modèle d'exécution avec ordonnancement statique d'un programme Athapascan-1, proposé dans la section 4.4.2.2, on obtient la borne suivante :

Corollaire 6 *Soit un programme Athapascan-1 dans lequel seule la tâche racine crée d'autres tâches et génère le graphe G et dans lequel les coûts des tâches et les tailles des*

objets partagés sont fournis par l'utilisateur. Alors l'ordonnancement de ce graphe sur m processeurs identiques conduit à un temps d'exécution T_m majoré par :

$$T_m \leq \frac{T_1}{m} + (1 - \frac{1}{m})T_\infty + \tau C_{max} + O(mn^2) + O(n + n_d)$$

.

Preuve. Elle découle directement du corollaire 1 page 73, du théorème 3 page 84 et du coût $O(mn^2)$ nécessaire au calcul de l'ordonnancement par l'algorithme ETF [69]. □

5.7 Conclusion

Divers algorithmes d'ordonnancement d'un graphe de flot de données ont été présentés et proposés dans ce chapitre. Ces algorithmes d'ordonnancement, une fois appliqués au modèle d'exécution d'Athapascan-1 proposé dans le chapitre précédant, permettent d'aboutir à des majorations du temps nécessaire à l'exécution d'un programme quelconque écrit en Athapascan-1.

Cette analyse des algorithmes d'ordonnancement et des bornes associées amène à la constatation suivante. Sur machine à mémoire partagée, les performances obtenues avec un ordonnancement glouton sont difficilement améliorables lorsque le graphe de flot de données de l'application est quelconque (sur machine identique). Fournir alors explicitement le graphe de flot de données à l'ordonnanceur n'apporte donc pas de gain significatif. Par contre, il en va tout autrement sur machine à mémoire distribuée : en effet, dans ce cas nous avons vu qu'il existait différent algorithmes d'ordonnancement de complexité polynômiale qui, grâce à la connaissance (éventuellement partielle) du graphe de flot de données, améliorent les performances en comparaison de l'algorithme glouton. Cela justifie donc en partie le modèle d'exécution d'Athapascan-1 qui offre à l'ordonnanceur la possibilité d'explorer le graphe de flot de données même dans le cas d'applications pour lesquelles ce graphe n'est pas totalement connu avant l'exécution.

6

Interface de programmation et support d'exécution Athapascan-1

6.1 Introduction

Ce chapitre présente l'implantation effective de l'interface de l'interface de programmation Athapascan-1. Après une discussion sur le choix du langage séquentiel de base, en l'occurrence C++, ce chapitre se compose de deux sections. Dans la première, nous décrivons l'interface de programmation (*API*) de la bibliothèque, supportant le modèle de programmation décrit au chapitre 3. Dans la seconde nous décrivons les composants et les mécanismes de cette bibliothèque permettant l'exécution du programme suivant les modèles d'exécutions décrits dans le chapitre 4. La particularité est ici la possibilité d'utiliser plusieurs algorithmes d'ordonnancement tels que ceux présentés au chapitre 5. Le choix de l'ordonnancement le mieux adapté dépendant non seulement de la connaissance du graphe et donc de l'application mais aussi de l'architecture.

6.2 Choix du langage séquentiel de base

La sémantique d'Athapascan-1, de type séquentielles et basée sur un modèle de programmation procédural, motive son implantation au dessus d'un langage séquentiel impératif classique. Ceci permet entre autre de pouvoir réutiliser les implantations efficaces de ces langages.

Athapascan-1 est implanté par une bibliothèque écrite en C++ [120]. Ce choix d'une bibliothèque C++ comme support du modèle de programmation d'Athapascan-1, plutôt

qu'un compilateur prenant en entrée une extension du langage C++, a été dicté pour des raisons de simplicité : en effet les mécanismes de généricité et de typage fort du langage C++ permettent une mise en œuvre et une vérification syntaxique efficace. En contrepartie la syntaxe est plus complexe que celle introduite dans le chapitre 3 et peut paraître obscure, au premier abord, pour une personne ne connaissant pas le langage C++.

Bien que le langage C++ n'intègre aucun support pour la programmation parallèle (contrairement à des langages comme Modula-2, Ada ou Java par exemple), beaucoup d'environnements de programmation parallèles ainsi que des extensions « parallèles » de langages ont été construit sur C++. On peut se reporter au livre [124] pour un survol des différents environnements et extensions parallèles possibles réalisées sur C++.

Cependant, on peut se poser la question de savoir si ce choix du langage C++ est judicieux du point de vue des applications de calcul scientifique. Le langage Fortran est *a priori* la référence pour ces applications basées sur des manipulations de tableaux. Cependant, Fortran est de plus en plus concurrencé par d'autre langages de programmation plus récents, principalement C++, et ceci pour plusieurs raisons. Tout d'abord C++ offre maintenant, grâce aux améliorations des compilateurs et à certaines techniques de programmations (notamment la STL [92] et les classes numériques `valarray` de la bibliothèques standard C++), des performances pratiquement équivalentes à Fortran sur de nombreuses applications de calcul scientifique [109, 123, 36]. Par ailleurs, l'avantage qu'a Fortran d'être un langage spécialisé peut se transformer en handicap puisqu'il devient naturellement moins utilisé que des langages d'usage plus général. Le langage C++ apporte en outre des fonctionnalités utiles pour la programmation d'applications en calcul scientifique comme la programmation orientée objet, l'écriture de code générique ou la surcharge des opérateurs du langage. Le langage Fortran a évolué pour intégrer certaines notions de programmation orientée objet mais il n'autorise pas toutes les capacités offertes par C++ [18]. Pour ces raisons, de plus en plus d'applications et de bibliothèques de calcul scientifique sont maintenant développées en C++.

6.3 Interface de programmation

Nous décrivons dans cette section les différentes fonctions et classes de la bibliothèque C++ Athapascan-1 qui implémentent le modèle de programmation d'Athapascan-1. Ces classes permettent la création des tâches et des objets partagés, la définition des références contraintes avec leurs droits d'accès ainsi que les informations passées au système d'exécution pour aider à l'ordonnancement du programme.

6.3.1 Objet communicable

Le langage C++ n'offrant pas de support pour communiquer un objet C++ entre plusieurs nœuds, l'interface de programmation Athapascan-1 introduit la notion d'objets

communicable. Un objet communicable est un objet C++ disposant de fonctions permettant de manipuler cet objet pour le transmettre sur un autre nœud. Pour qu'un objet de type `T` soit communicable, les fonctions suivantes sont requises :

- Trois fonctions membres permettant de copier et de détruire l'objet :
  - Un constructeur vide : `T()`.
  - Un constructeur de recopie : `T( const T& )`.
  - Un destructeur : `~T()`.
- Deux fonctions, dites fonctions d'emballage et de déballage, permettant de parcourir le contenu de l'objet, sur un principe identique aux fonctions d'entrée-sortie par flot de la librairie standard C++ :

```
1: a1_ostream& operator<<( a1_ostream& out, const T& t );
2: a1_istream& operator>>( a1_istream& in, T& t );
```

Notons que ces fonctions sont prédéfinies par la bibliothèque pour certains types : les types de base de C++ (`char`, `int`, `float`, `double`, ...) ainsi que les containers de la STL (`vector<E>`, `list<E>`,...), sous réserve que ceux-ci soient utilisés avec un type `E` communicable. Par ailleurs, les constructeurs (vide et de recopie) et le destructeur sont implicitement fournis par le compilateur si la classe ne les définit pas.

La figure 6.1 page 96 montre un exemple d'une classe C++ définissant un type d'objets communicable. L'objet est ici une structure composée d'un entier et d'un double.

6.3.2 Objet partagé et références contraintes sur ces objets

Les objets partagés du modèle de programmation sont, dans l'interface de programmation, des objets C++ communicables.

Les références contraintes du modèle de programmation sont également des objets C++, typés à la fois par le type de l'objet partagé et par les droits d'accès sur l'objet partagé. La «syntaxe» d'un type de référence contrainte est la suivante :

```
1: Shared_[droit-accès]<[type-objet-communicable]>
2: Shared_[droit-accès]<[fonction-accumulation],
3:                      [type-objet-communicable]>
```

Les droits d'accès [droit-accès] de base sont **r** pour un droit de lecture, **w** pour un droit d'écriture, **cw** pour un droit d'accumulation (uniquement avec la fonction d'accumulation fournie) et **r_w** pour un droit de lecture et modification. En outre, les droits d'accès différés associés à chacun de ces accès de base sont respectivement **rp**, **wp**, **rp_wp**, **cwp**.

```
1: class nom_type {
2: public:
3:     // constructeur vide, opérateur de copie et destructeur fournie implicitement.
4:
5:     // les données de l'objet
6:     int i;
7:     double j;
8: };
9: // fonction d'embalage
10: a1_ostream& operator<<(a1_ostream& out, const nom_type& o) {
11:     out << o.i << o.j;
12: }
13: // fonction de débalage
14: a1_istream& operator>>(a1_istream& in, nom_type& o) {
15:     in >> o.i >> o.j;
16: }
```

Figure 6.1 – *Exemple d'une classe C++ définissant un type d'objets communicable.*

De manière analogue au choix standard effectué pour la STL, les fonctions d'accumulation [fonction-accumulation] sont des fonctions classe C++, définies par l'utilisateur de la manière suivante :

```
1: struct [fonction-accumulation] {
2:     void operator()( [type-objet-communicable] & a,
3:                      const [type-objet-communicable] & b) {
4:         // ici,  une opération d'accumulation de  b dans a
5:     }
6: };
```

Notons que, comme dans le modèle de programmation, la bibliothèque suppose que l'opération associé à cette fonction d'accumulation est commutative et associative.

Déclaration d'un objet partagé Un objet partagé de type `T` est déclaré en construisant une référence contrainte `x` pouvant éventuellement être initialisée par une valeur initiale mais ne pouvant pas être directement modifiée (choix d'accès en lecture-écriture différée) :

```
1: Shared< T > x;
2: Shared< T > x( t );
```

Accès à un objet partagé Les accès sur un objet partagé de type `T` sont réalisés par les appels de méthodes suivantes définies sur les références contraintes. Chacun des quatre types de références contraintes fournit la méthode suivante pour l'accès à l'objet référencé :

- si x est du type **Shared_r_w**<`T`> alors `x.access()` retourne une référence C++ sur l'objet partagé référencé par `x`. L'objet peut alors être lu et modifié dans la suite de l'exécution de la tâche.
- si x est du type **Shared_r**<`T`> alors `x.read()` retourne une référence constante C++ sur l'objet partagé référencé par `x`. L'objet partagé ne peut alors qu'être lu dans la suite de l'exécution de la tâche.
- si x est du type **Shared_w**<`T`> alors `x.write( t )` affecte la valeur `t` sur l'objet partagé référencé par `x`.
- si x est du type **Shared_cw**<`f,T`> alors `x.cumul( t )` accumule la valeur `t` sur l'objet partagé référencé par `x` avec la fonction classe `f`.

6.3.3 Type de tâche et création de tâche

La définition d'un type de tâche est réalisé par l'écriture d'une fonction classe de la manière suivante :

```
1: struct [type-tâche] {
2:     void operator()( [liste-paramètres-formels]          ) {
3:         [corps-tâche]
4:     }
5: };
```

où [type-tâche] est le nom du type de tâche, [liste-paramètres-formels] est la liste des paramètres formels de la tâche et [corps-tâche] est le corps de la tâche, c'est-à-dire les instructions qui seront exécutées par cette tâche.

La création d'une tâche est alors réalisée de la manière suivante :

```
1: Fork< [type-tâche] >()([liste-paramètres-effectifs] );
```

où [type-tâche] est un type de tâche identifiant la procédure à exécutée par la tâche et où [list-paramètres-effectifs] est la liste des paramètres effectifs passés à la tâche.

Les paramètres formels et effectifs peuvent être soit des types d'objets communicables (passage par valeur) soit des références contraintes sur des objets partagés (passage par référence). Si le paramètre effectif est un objet communicable alors ce type doit être identique au type correspondant du paramètre formel. Si le paramètre effectif est une référence

type effectif		type formel autorisé	
Shared_rp_wp	<T>	**Shared_rp_wp**	<T>
Shared	<T>	**Shared_rp**	<T>
		Shared_r	<T>
		Shared_wp	<T>
		Shared_w	<T>
		Shared_cwp	<f,T>
		Shared_cw	<f,T>
Shared_r_w	<T>	x	
Shared_r	<T>	**Shared_r**	<T>
Shared_rp	<T>	**Shared_rp**	<T>
Shared_w	<T>	**Shared_w**	<T>
Shared_wp	<T>	**Shared_wp**	<T>
Shared_cw	<f,T>	**Shared_cw**	<f,T>
Shared_cwp	<f,T>	**Shared_cwp**	<f,T>

Tableau 6.1 – *Règles de conversion autorisées sur une référence contrainte. Ce changement de droit d'accès ne peut intervenir que lors du passage de la référence en paramètre d'une tâche.*

contrainte alors les type formels et effectifs doivent respecter les règles de conversion du tableau 6.1.

Ces règles de conversion sont l'implémentation effective des règles présentées dans le chapitre 3 qui permettent l'analyse à la volée des dépendances de données entre les tâches et évitent la synchronisation lors de l'exécution d'un corps de tâche. Il est à noter que ces règles de conversion sont vérifiées sans exception par le compilateur C++ lors de la compilation de l'application.

6.3.4 Informations pour le système d'exécution

Quatre attributs d'informations peuvent être associés à chaque tâche créée. Ces informations sont enregistrées dans le graphe de flot de données et pourront être consultées par les ordonnanceurs :

- Le coût d'exécution de la tâche.
- La localité de la tâche (*i.e.* le processeur sur lequel elle doit préférenciellement s'exécuter).
- La priorité de la tâche.
- Un attribut supplémentaire dont le sens dépend de la politique d'ordonnancement utilisée.

Ces informations sont passées lors de la création de la tâche de la manière suivante :

```
1 : Fork<[type-tâche]>(SchedAttribute([infos]))([liste-paramètres-effec.])
```

Par défaut, une tâche est ordonnancée avec la même politique d'ordonnancement que celle utilisée pour ordonnancer la tâche qui l'a créée. En outre par défaut, la tâche racine est ordonnancée avec une politique d'ordonnancement à la volée de type glouton. Cependant une autre politique d'ordonnancement, choisie parmi une de celles disponibles dans la bibliothèque Athapascan-1, peut être spécifiée. Le tableau 6.2 page 104 donne la liste de quelques une des politiques d'ordonnancement fournies en standard dans la bibliothèque. Cette nouvelle politique [politique] peut être précisée soit lors de la création de la tâche soit par l'appel de la fonction `a1_set_scheduling` qui modifie la politique prise par défaut. Ces deux méthodes sont les suivantes :

```
1 : Fork<[type-tâche]>([politique])([liste-paramètres-effec.]);
2 : a1_set_scheduling([politique]);
```

6.3.5 Exemple : élimination de Gauss par colonnes

On reprend ici comme exemple l'élimination de Gauss par colonnes, déjà utilisé pour illustrer le modèle de programmation dans le chapitre 3. L'écriture de cet algorithme dans le modèle de programmation d'Athapascan-1 avait alors été présenté à l'aide d'une syntaxe simplifiée. Nous présentons maintenant dans la figure 6.2 page 100 l'écriture de cet algorithme dans l'interface de la bibliothèque Athapascan-1. On peut noter ici l'utilisation du type vector de la STL qui remplace ici le tableau C. La bibliothèque Athapascan-1 prédéfinit les fonctions d'emballage et de déballage pour tous les conteneurs de la STL (*Standard Template Library*) donc ces fonctions n'ont pas besoin d'être réécrites pour le type `column` du programme.

À la ligne 1 et 7, les deux types de tâches utilisés dans le programme sont définis. Ces types de tâches seront ensuite invoqués aux lignes 26 et 28. Aux lignes 3, 10 et 11, les références contraintes prises en paramètre des tâches sont déréférencées pour accéder en place (*i.e.* sans copie) à l'objet en mémoire partagée. Nous pouvons également noter diverses informations liées à l'ordonnancement : à la ligne 22 la politique d'ordonnancement prise par défaut est modifiée pour devenir une politique de vol de tâche aléatoire ; aux lignes 26 et 28, les créations de tâches sont attribuées par une information de coût (valeur de la variable `sa` initialisée à la ligne 25).

6.4 Implantation du support d'exécution

Le fonctionnement interne de la bibliothèque Athapascan-1 suit de très près le modèle d'exécution décrit dans le chapitre 4. Nous ne détaillerons ici que quelques points non abordés dans ce chapitre 4.

```
1:  typedef vector<double>  column;
2:  typedef vector<Shared_rp_wp<column> > matrice;
3:
4:  struct scal { // x ← a * x
5:     void operator()(int k, double a, Shared_r_w<column> x) {
6:        vecteur& xx = x.access();
7:        for( int i = k+1; i<xx.size() ; i++ )
8:           xx[i] = a * xx[i];
9:     }
10: };
11: struct axpy { // y ← a * x + y
12:    void operator()(int k, double a, Shared_r<column> x,
13:                    Shared_r_w<vecteur> y) {
14:       const vecteur& xx = x.read();
15:       vecteur& yy = y.acces();
16:       for( int i = k+1; i<yy.size() ; i++ )
17:          yy[i] = a * xx[i] + yy[i];
18:    }
19: };
20: main() {
21:    matrice A;
22:    /* ici, initialisation de A */
23:
24:    // Choix de la politique d'ordonnancement
25:    a1_set_scheduling(a1_work_stealing::basic());
26:
27:    for( int k = 0; k < A.size() - 1; k++ ) {
28:      SchedAttribute sa(2*(A.size()-k)); // 2(n - k)
29:       Fork<scal>(sa) ( k, 1/A[k][k], A[k]);
30:       for( int j = k+1; j<A[k].size(); j++ )
31:          Fork<axpy>(sa) ( k, - A[j][k], A[k], A[j]);
32:    }
33: }
```

Figure 6.2 – *Élimination de Gauss par colonnes dans l'interface de programmation Athapascan-1*

Notons tout d'abord l'existence de deux versions distinctes de la bibliothèque, offrant la même interface de programmation présentée dans la section précédente. La première de ces versions est très générale et peut à la fois utiliser des algorithmes d'ordonnancement statiques et dynamiques. Elle peut donc exécuter tous les programmes Athapascan-1. Nous l'appellerons dans la suite «version générale». Une deuxième version a ensuite été développée, spécialisée pour les application dont le graphe de flot de données est entièrement généré en début d'exécution avec une estimation des coûts des tâches et de la taille des objets partagés. Cette version est basée sur le modèle d'exécution spécialisé proposé dans la section 4.4.2.2 page 71 : un ordonnancement statique du graphe est calculé en début d'exécution qui conduit à un placement des tâches sur les processeurs. Nous appellerons cette version de la bibliothèque «version spécialisée».

6.4.1 Architecture logicielle d'Athapascan-1

La bibliothèque Athapascan-1 repose sur un noyau exécutif portable, Athapascan-0 [47] [15]. Athapascan-0 est une bibliothèque qui fournit des primitives de création de processus légers, localement et à distance, ainsi que de communication (synchrone et asynchrone). Le modèle de programmation offert par cette bibliothèque est celui d'un réseau dynamique de processus légers communicants. Cette notion de processus légers communicants fournit naturellement un mécanisme de réactivité face aux communications. Sur un même processeur, un processus léger peut réaliser des calculs pendant qu'un autre attend l'arrivée d'une communication. Un processeur peut alors continuer à travailler tout en étant capable de répondre activement à des demandes venant d'autres processeurs. Les différentes méthodes utilisées par la bibliothèque Athapascan-0 pour assurer cette réactivité, au sens où un processus bloqué sur l'arrivée d'une communication est réveillé au plus tôt, sont complexes et délicates à régler, chaque machine parallèles étant un cas particulier. Elles consistent principalement en une scrutation périodique du réseau de communication. Cette bibliothèque fournit ainsi à Athapascan-1 un support portable et efficace pour la réactivité face aux communications.

La bibliothèque Athapascan-0 utilise elle même une bibliothèque de processus légers (Posix Thread [75] ou Marcel [89]) et une bibliothèque de communication (généralement MPI [119] ou éventuellement Nexus [42]). L'architecture logicielle de l'ensemble de ces composants est représentée dans la figure 6.3 page 102.

6.4.2 Exécution des tâches et processus légers

Pour des raisons d'efficacité, un processus léger n'est pas créé pour chaque tâche à exécuter. De par le modèle de programmation, une tâche prête peut toujours s'exécuter sans synchronisation. Si la machine parallèle a m processeurs, il n'est alors pas nécessaire d'avoir plus de m tâches s'exécutant simultanément sur la machine pour exploiter

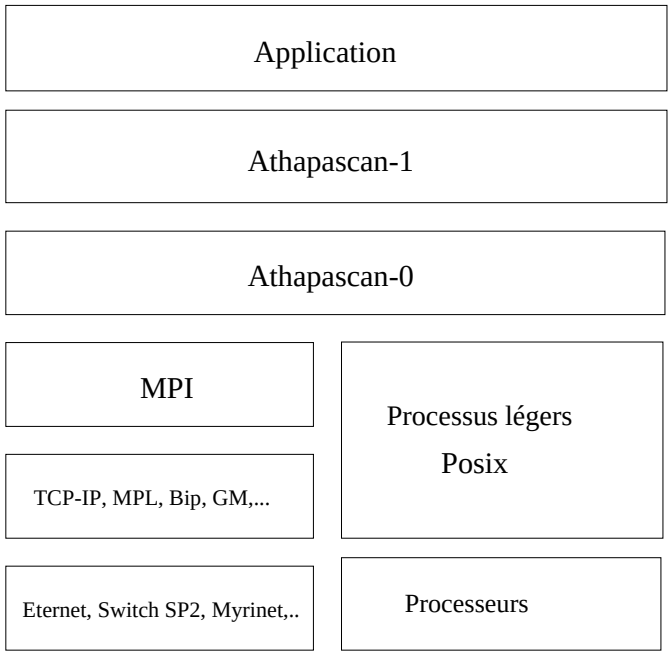

Figure 6.3 – *Architecture logicielle d'Athapascan-1*

toutes les ressources de calcul. Un plus grand nombre de tâches s'exécutant simultanément pourrait cependant permettre le recouvrement les latences systèmes comme les défauts de page.

Pour exécuter le programme sur m processeurs, m processus légers de niveau système sont alors créés au lancement du programme. Lorsqu'un de ces processus légers devient disponible, il demande une nouvelle tâche à exécuter à l'ordonnanceur, l'exécute, et prévient l'ordonnanceur de la fin de l'exécution.

6.4.3 Communication d'objets

La communication d'objets C++ à travers le réseau pose différents problèmes. Les primitives de base offertes par la bibliothèque Athapascan-0 pour la communication sont l'envoi et la réception de zones mémoire contiguës, ainsi que des mécanismes permettant de communiquer des tableaux sur une machine hétérogène. Cependant, un objet C++ est généralement beaucoup plus complexe.

Le point le plus délicat est celui posé par le typage dynamique des objets C++. L'information sur le type de l'objet, et donc sur la structure associée au type (taille et champs de la classe) n'est pas nécessairement connue à la compilation mais est contenue dans l'objet lui même. Lorsque l'on doit copier un objet d'un nœud A sur le nœud B, le nœud B doit connaître le type exact de l'objet. Si l'objet utilise un typage dynamique, l'informa-

tion du type de l'objet doit être communiquée au préalable par le réseau. La bibliothèque Athapascan-1 implémente pour cela un mécanisme d'identification automatique de tous les objets C++ pouvant être éventuellement communiqués [1]. A chaque type C++ est alors associé un identificateur (un entier) pouvant être communiqué par le réseau, et qui permet de reconstruire à distance un objet du même type.

Le second point à traiter est la manière dont l'objet est décomposé en zones contiguës de mémoire pour être passé à la bibliothèque Athapascan-0. Si l'objet est par exemple une liste d'entiers, les éléments de la liste devrons être intégralement recopiés dans une zone mémoire contiguë. Par contre si c'est une liste de tableau, seules les tailles des tableaux seront recopiées sur une zone mémoire contiguë, les tableaux eux mêmes seront directement envoyés sans copie intermédiaire [2] s'ils sont de taille suffisante.

6.4.4 Interface pour l'implantation d'algorithmes d'ordonnancement

La bibliothèque Athapascan-1 fournit une interface de programmation permettant de définir aisément de nouvelles politiques d'ordonnancement. Cette interface de programmation est détaillée dans [19] [21] [20]. Les fonctionnalités offertes sont cependant similaires à celles exposées dans le chapitre 4 page 55 consacré au modèle d'exécution. L'ordonnanceur dispose de la liste des tâches prêtes, de la liste des processeurs disponibles et peut consulter le graphe de flot de données constitué des tâches créées et non encore exécutées et décrivant les dépendances de données entre ces tâches. Notons cependant que dans un environnement distribué, seules les informations locales sont disponibles.

Le tableau 6.2 page 104 contient quelques unes des différentes politiques d'ordonnancement déjà implantées dans la bibliothèque Athapascan-1. Certaines de ces politiques d'ordonnancement seront utilisées expérimentalement dans la partie II de ce document.

6.5 Évaluation expérimentale

Nous présentons dans cette section une première évaluation des primitives de base de la bibliothèque Athapascan-1. Cette évaluation sera complétée dans la deuxième partie de la thèse par des applications de calcul numérique programmées en Athapascan-1.

1. Le langage C++ fournit également un identificateur pour chaque type, appelé *rtti*, mais cet identificateur est un objet opaque qui ne peut donc pas être transmis par le réseau. De plus, la manière dont cet identificateur est codé n'est pas normalisée. Ce codage peut donc être différent suivant les compilateurs, ce qui est gênant dans le cas d'une exécution sur un environnement hétérogène.

2. Sous l'hypothèse que les bibliothèques de communications sous-jacentes ne réalisent pas de copie. C'est par exemple le cas de Bip [103] sur Myrinet.

Type de la politique	Description de la politique d'ordonnancement
Politiques de type placement	
`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::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.
Politiques de type ordonnancements statiques	
`a1_static_mapping::dsc`	L'ordonnancement des tâches est réalisé par la bibliothèque Pyrros [56] qui met en œuvre l'algorithme DSC pour regrouper les tâches.
`a1_static_mapping::etf`	L'ordonnancement des tâches est réalisé par l'algorithme ERT [81] variante de l'algorithme d'ETF.
Politiques de type ordonnancements à la volée	
`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 6.2 – *Quelques une des politiques d'ordonnancement prédéfinies en Athapascan-1*

6.5.1 Espace mémoire utilisé pour la gestion du graphe

Le graphe de flot de données est construit et interprété dynamiquement. Lorsqu'une tâche est créée, elle est insérée dans le graphe. Lorsqu'elle se termine, elle est retirée de ce graphe. Donc seules les tâches créées et non encore exécutées sont présentes en mémoire. Il en est de même pour tous les autres objets utilisés pour représenter ce graphe. Nous donnons ici une borne sur la place mémoire utilisée par la bibliothèque Athapascan-1 pour stocker pendant l'exécution cette partie du graphe présente en mémoire.

Cette borne a été calculée à partir de la taille de chacun des objets utilisés pour représenter le graphe. La taille de ces objets à été évaluée sur une machine à espace d'adressage 32 bits, pour laquelle un pointeur est stocké sur 4 octets.

Pour exprimer cette borne, le graphe de flot de données G_t, présent en mémoire à un instant donné t de l'exécution, est caractérisé par les paramètres suivants :

- n, le nombre de tâches de G_t.
- n_d la somme des degrés des nœuds tâches de G_t, c'est-à-dire le nombre total d'objets partagés passés en paramètres d'une tâche.

– n_h le nombre de nœuds accès de G_t. Notons que $n_h \leq n_d$ puisque des accès concurrents en lecture (ou en accumulation) correspondent à un seul noeud accès.

– v le nombre total de paramètres passés par valeur aux tâches de G_t.

– v_s la somme des tailles, en octets, des paramètres passés par valeur aux tâches de G_t.

L'espace mémoire, utilisé pour représenter le graphe G_t dans la version générale de la bibliothèque est [48] :

$$S_t(G_t) = 128n + 164n_d + (156 + 2p)n_h + v_s + 4v \text{ octets}$$

où p est le nombre de processeurs utilisés pour interpréter le graphe.

Par contre, dans la version spécialisée, l'espace mémoire utilisé pour représenter le graphe G_t est :

$$S_l(G_t) = 12n + 20n_d + v_s \text{ octets}$$

soit environ inférieur d'un facteur 10.

Notons que la représentation du graphe de flot de données sous la forme d'un graphe distribué de tâche et de transitions n'a pratiquement pas été optimisé. L'espace mémoire utilisé peut être considérablement réduit.

6.5.2 Coûts de création et d'interprétation du graphe de flot de données

Le coût de création et d'insertion dans le graphe d'une tâche à été évalué sur une station SUN à quatre processeurs Ultra Sparc II cadencé à 250 MHz, utilisant le système Solaris 7. Sur cette machine le coût d'un appel de fonction est de $9.6 \times 10^{-3} \mu s$ et la puissance de référence est de 390 Mflops[3] (million d'opérations flottantes par seconde).

Nous donnons dans la suite une évaluation du coût de création et d'insertion dans le graphe d'une tâche prenant en paramètre k objets partagés. Ce coût a été évalué par l'exécution, sur un processeur, d'un programme créant 200 tâches. La figure 6.4 donne les résultats obtenus en fonction du type et du nombre de paramètre pris par la tâche.

Le coût de création et d'insertion d'une tâche prenant en paramètre k objets partagés est donc

– pour la version standard d'environ $(50 + 19k)\mu s$ (*i.e.* $12500 + 4750k$ cycles du processeur) pour des paramètres avec un mode d'accès en écriture ou lecture écriture et d'environ $(30 + 5k)\mu s$ (*i.e.* $7500 + 1250k$ cycles du processeur) pour les autres paramètres.

3. Obtenue sur le produit de deux matrices de taille 100×100 par la fonction dgemm de la bibliothèque BLAS fournie SUN (Sun Performance Library 2.0). Le nombre d'opérations flottantes d'un produit de matrice de taille n est évalué à $2n^3$.

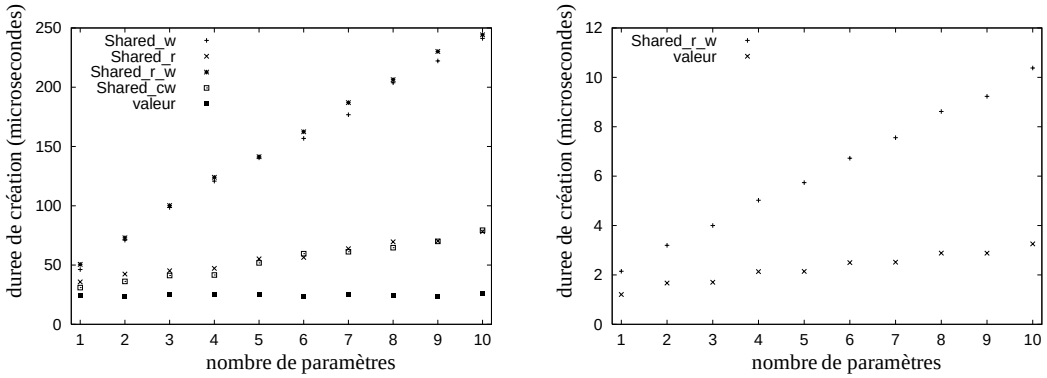

Figure 6.4 – *Temps de création et d'insertion d'une tâche dans le graphe en fonction du type et du nombre de paramètres. Le tableau de gauche est pour la version standard de la librairie et le tableau de droite pour la version spécialisée.*

– pour la version spécialisée de $(2.2+0.95k)\mu s$ (*i.e.* $550+237k$ cycles du processeur) indépendamment du mode d'accès des paramètres.

Pour estimer le coût d'interprétation du graphe, nous avons utilisé l'exemple de calcul récursif d'un terme de la suite de Fibonacci présenté dans la figure 3.1 page 48. Ce programme à cependant été modifié pour que le calcul récursif d'un terme inférieur à un seuil fixé soit réalisé en séquentiel sans faire intervenir de création de tâche.

La figure 6.5 page 107 montre les résultats obtenus en fonction du seuil et du nombre de processeurs utilisés (plus exactement en fonction du nombre de processus légers de niveau système utilisés). Par ailleurs, nous avons également ajouté dans cette figure les résultats obtenus sur le même programme en utilisant le langage Cilk [11].

Le surcoût de création et de gestion du graphe est donc amorti jusqu'à un seuil de 15 qui correspond à une granularité maximum des tâches générées de 8 millisecondes et une accélération du programme est obtenue jusqu'à un seuil de 11 qui correspond à une granularité maximum des tâches générées de 1 milliseconde.

Pour un seuil raisonnable, Athapascan-1 et Cilk ont des performances très similaires. La plus faible granularité de Cilk s'explique d'une part parce que le langage est plus simple[4] et d'autre part du fait d'une implantation optimisée pour ce type de calcul récursif qui génère un grand nombre de tâches (voir la section 2.4.1.2 page 33).

6.5.3 Coût de gestion de la mémoire partagée distribuée

Le coût de gestion de la mémoire partagée distribuée est évaluée par un programme de type « ping-pong ». Ce programme consiste en une suite de tâches accédant toutes

4. Cilk ne permet que la synchronisation d'une tâche mère sur la terminaison de tâches filles, le graphe de dépendance entre les tâches est donc de type série-parallèle

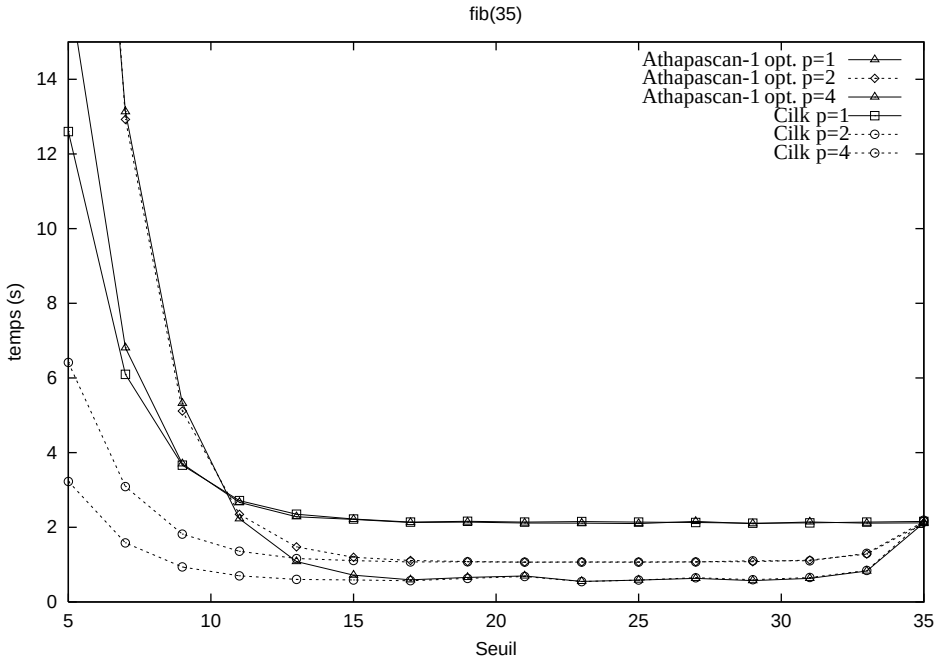

Figure 6.5 – *Durée de calcul du terme 35 de la suite de Fibonacci en fonction du seuil d'arrêt. Chacune des courbes correspond à un nombre différent de processeur utilisé.*

le même objet partagé en modification. Cet objet partagé est constituée d'un tableau de taille fixé et connue par les deux processeurs. Une politique d'ordonnancement spécifique, `a1_mapping::fixed`, permet de forcer l'exécution d'une tâche sur un processeur choisi lors de la création de la tâche. Cet ordonnancement est alors utilisé de manière à ce que deux tâches consécutives soient exécutées sur deux processeurs différents. Le tableau contenu dans l'objet accédé par les tâches est alors transmis d'un processeur à l'autre durant l'exécution.

Ce programme ne permet pas d'évaluer exactement le coût de gestion de la mémoire partagée puisque les surcoûts liés à la création du graphe de flot de données et à la gestion des synchronisations entre les tâches sont également pris en compte. Les résultats obtenus permettent cependant de donner une bonne approximation du coût de gestion de la mémoire partagée distribuée.

Les expérimentations ont été réalisées sur deux nœuds d'un IBM-SP1. Les résultats obtenus sont donnés dans la figure 6.6 page 108. À titre de comparaison, nous fournissons les résultats obtenus sur un programme MPI réalisant un « ping-pong » d'un tableau de taille équivalente. Comparé à MPI, les résultats sont les suivants :

– Version générale : le coût d'un « ping-pong » est d'un facteur 100 supérieur à MPI pour un message petit et d'un facteur 4.5 pour un message de 100 kilo-octets. Ce

facteur prohibitif s'explique par l'implantation plus complexe de l'algorithme distribué de détection de terminaison conduisant à un code peu efficace sur machine SMP.

– Version spécialisée : le surcoût obtenu est d'environ 50 μs indépendamment de la taille du tableau accédé en mémoire partagée. Ce surcoût peut donc être facilement amorti en choisissant une granularité adaptée des objets partagés.

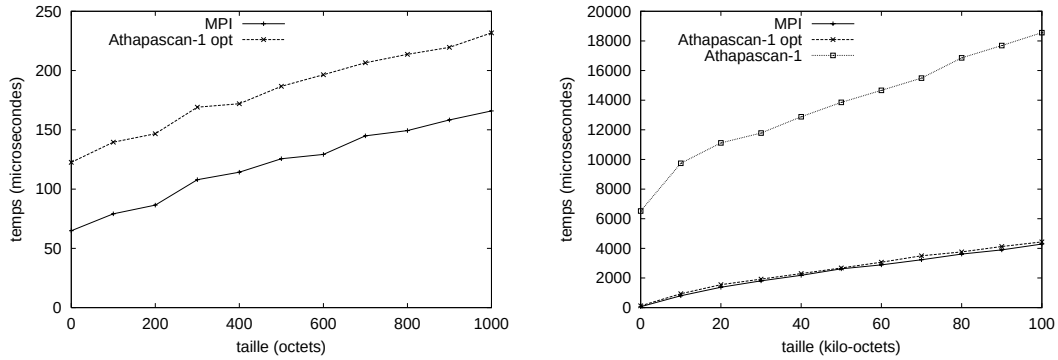

Figure 6.6 – *Résultats de l'exécution d'un programme générant des allers-retours en mémoire partagée distribuée.*

6.6 Conclusion

Nous avons montré dans ce chapitre l'intégration dans une bibliothèque C++ du modèle de programmation Athapascan-1 présenté dans le chapitre 3. Il est à noter que cette intégration n'a nécessité aucune phase de précompilation des programmes. Ceci à été rendu possible par l'utilisation intensive des mécanismes de généricité offerts par le langage C++.

Nous avons également montré dans ce chapitre l'implantation dans cette bibliothèque du modèle d'exécution présenté dans le chapitre 4. Les premières expérimentations réalisées ont permis d'évaluer les coûts de création et d'interprétation du graphe ainsi que les coûts de gestion de la mémoire partagée distribuée.

Deux versions d'Athapascan-1 ont été implantées :

– Une générale. Bien que de surcoût important, elle permet de réagir dynamiquement à tout type de comportement du programme. Cette version doit cependant pouvoir être facilement améliorée pour réduire son surcoût.

– Une spécialisée qui réalise une dégénérescence séquentielle systématique des tâches. Le surcoût de cette version est très faible. Bien que restreint à des applications dont

le graphe de flot de données peut être calculé en début d'exécution, cette bibliothèque spécialisée est bien adaptée aux applications d'algèbre linéaire (dense et creuse) étudiées dans la seconde partie de cette thèse.

II

Applications d’algèbre linéaire en Athapascan-1

7

Évaluation d'Athapascan-1 sur des problèmes de base d'algèbre linéaire

7.1 Introduction

Dans ce chapitre, nous nous intéressons à l'écriture 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, tel que $A = LU$ 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, telle que $A = LL^t$ avec L triangulaire inférieure. Cette factorisation est réalisée par une élimination de Cholesky qui est une variante de l'élimination de Gauss pour les matrices symétriques.

Il est à noter que les factorisations de Gauss et de Cholesky sont sans pivot. Sur des matrices quelconques, elle ne sont généralement pas stables numériquement en calcul flottant, du fait des erreurs d'arrondis. Elles n'ont donc d'intérêt pratique que pour certaines classes de matrices comme les matrices à diagonales dominantes ou symétriques définies positives. On peut se référer au livre [80] pour le détail de ces deux factorisations.

7.2 Algorithmes parallèles avec partitionnement bidimensionnel de la matrice

Les trois algorithmes utilisés ici sont des variantes des algorithmes classiques, dans lesquels les calculs sont restructurés de manière à exprimer des opérations sur des sous-matrices blocs plutôt que sur des scalaires.

L'intérêt de réorganiser les calculs des algorithmes d'algèbre linéaires denses mais également creux (voir le chapitre 8), de manière à utiliser les BLAS de niveau 3 [33], n'est plus à démontrer. Cette technique est largement utilisée en séquentiel comme par exemple dans la librairie LAPACK [4] mais également en parallèle [49]. Elle permet d'exploiter au mieux les mémoires à plusieurs niveaux ou mémoires caches des machines actuelles. Sur ces machines, seul le niveau de cache le plus élevé est capable de fournir les données aux unités de calculs à la fréquence du processeur. Sur une opération BLAS de niveau 3 utilisant n^2 mots mémoire, il y a $O(n^3)$ calculs à réaliser. En choisissant la taille des blocs de manière à ce que les n^2 éléments logent dans le cache mémoire de niveau le plus élevé, les $O(n^3)$ calculs peuvent être réalisés en exploitant pleinement la puissance du processeur.

Cette réorganisation des calculs pour se ramener à des opérations sur des sous-matrices est également utile dans les versions distribuées de ces algorithmes. Une certaine distribution des données par bloc, appelée placement par bloc cyclique bidimensionnel [79], tend à s'imposer, de par ses propriétés d'extensibilité, d'équilibrage de charge et de minimisation des communications, comme méthode générale de distribution des données pour les algorithmes distribués d'algèbre linéaire dense. Ce type de placement est par exemple utilisé dans les bibliothèques parallèles d'algèbre linéaire ScaLapack [22] et PLAPACK [1].

Pour réaliser ce placement cyclique bidimensionnel, la ou les matrices de tailles $n \times n$ à traiter sont partitionnées dans les deux dimensions en $N \times N$ blocs de dimensions $k \times k$. Pour des raisons de simplicité, on supposera dans la suite que k est un diviseur de n.

En réorganisant les boucles des algorithmes on peut alors réécrire ceux-ci, non plus en terme d'opérations sur des scalaires mais en opérations sur les blocs de la matrice partitionnée. Les algorithmes obtenus pour le produit de matrices et les factorisations LU et LL^t sont décrites dans les figures 6, 5 et 6. Le parallélisme de ces trois algorithmes, exprimé au niveau des opérations sur les blocs, est aisément identifiable.

Pour réaliser le placement par blocs cycliques bidimensionnels les blocs des matrices sont distribués de la manière suivante. En supposant que $q^2 = p$ processeurs sont disponibles, le blocs d'indice (i, j) est placé sur le processeur d'indice $(i \bmod q)q + (j \bmod q)$. La figure 7.1 montre un exemple de placement cyclique bidimmensionel des blocs sur 4 processeurs. Les opérations sur les blocs de matrice, sont alors réalisées sur le site où est localisé le bloc modifié par l'opération. On montre alors que le volume total de communication de ces trois algorithmes parallèles avec ce placement des données est en $O(n^2\sqrt{p})$ [79].

Algorithme 4 : Produit de matrices, avec partitionnement bidimensionnel.

Entrée : deux matrices A et B de taille $n \times n$, partitionnées dans les deux dimensions en $N \times N$ blocs ; $A_{i,j}$ référence un bloc de la matrice A, $B_{i,j}$ référence un bloc de la matrice B.

Sortie : $C = AB$ ou $C_{i,j}$ référence un bloc de la matrice C.

pour $k = 1$ **jusqu'à** N **faire [en parallèle]**
  pour $i = 1$ **jusqu'à** N **faire [en parallèle]**
    pour $j = 1$ **jusqu'à** N **faire [en parallèle]**
      $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$

Algorithme 5 : Factorisation LU d'une matrice dense par élimination de Gauss, avec partitionnement bidimensionnel de la matrice.

Entrée : matrice A de taille $n \times n$ partitionnée dans les deux dimensions en $N \times N$ blocs ; $A_{i,j}$ référence un bloc de la matrice.

Sortie : L et U tel que $LU = A$ (L et U sont stockés en place dans A).

pour $k = 1$ **jusqu'à** N **faire**
  Factorisation LU en place du bloc $A_{k,k}$
  pour $i = k + 1$ **jusqu'à** N **faire**
    $A_{i,k} = A_{i,k}A_{k,k}^{-1}$
  pour $j = k + 1$ **jusqu'à** N **faire [en parallèle]**
    $A_{k,j} = A_{k,j}A_{k,k}^{-1}$
  pour $j = k + 1$ **jusqu'à** N **faire [en parallèle]**
    pour $i = k + 1$ **jusqu'à** N **faire [en parallèle]**
      $A_{i,j} = A_{i,j} - A_{i,k}A_{k,j}$

Algorithme 6 : Factorisation de Cholesky LL^t d'une matrice dense symétrique définie positive, avec partitionnement bidimensionnel de la matrice.

Entrée : matrice A de taille $n \times n$ symétrique, partitionnée dans les deux dimensions en $N \times N$ blocs ; $A_{i,j}$ référence un bloc de la matrice ; les blocs situés au dessus de la diagonale de sont pas stockés (A est symétrique).

Sortie : LL^t tel que $LL^t = A$ (stocké en place dans A).

pour $k = 1$ **jusqu'à** N **faire**
  Factorisation LL^t en place du bloc $A_{k,k}$
  pour $i = k + 1$ **jusqu'à** N **faire [en parallèle]**
    $A_{i,k} = A_{i,k}A_{k,k}^{-1}$
  pour $j = k + 1$ **jusqu'à** N **faire [en parallèle]**
    pour $i = j$ **jusqu'à** N **faire [en parallèle]**
      $A_{i,j} = A_{i,j} - A_{i,k}A_{j,k}^{t}$

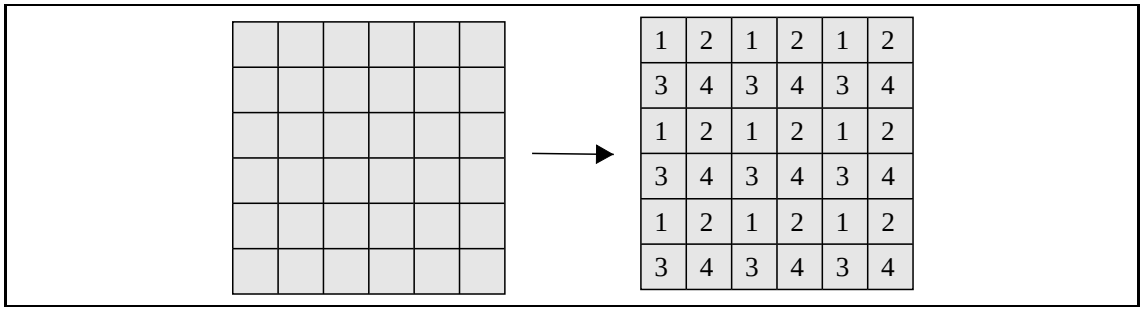

Figure 7.1 – *Placement cyclique bidimmensionel sur* 4 *processeurs des blocs d'une matrice partitionnée en* 6×6 *blocs.*

7.3 Écriture en Athapascan-1

Nous présentons dans cette section l'écriture en Athapascan-1 des trois algorithmes précédemment présentés. Seul le code du programme Athapascan-1 réalisant l'algorithme 5 page 115 de factorisation LU est fourni dans la figure 7.2. Les trois autres programmes sont très similaires.

L'écriture de ces algorithmes se compose de trois étapes. La première consiste à choisir la granularité des tâches et des objets partagés. Ici, cette granularité est naturellement définie au niveau des blocs de la matrice partitionnée. Les blocs de la matrice seront alors des objets partagés et les tâches seront associées aux opérations sur ces blocs. Notons que la taille des blocs peut être choisie à l'exécution.

Une fois que la granularité des tâches et des objets partagés est choisie, la seconde étape consiste à définir les types d'objets partagés et les types de tâches. Ici un seul type d'objet partagé est utilisé dans les programmes. Ce type noté `bloc` et représentant un bloc de matrice, contient un tableau des éléments d'un bloc de la matrice partitionnée.

Pour chacun des algorithmes, un type de tâche est ensuite défini pour chaque type d'opération réalisée sur les blocs. Par exemple, pour la factorisation LU cinq opérations sont utilisées donc cinq types de tâches sont définis dans le programme. Les 25 premières lignes du programme de la figure 7.1 définissent ces cinq types de tâches.

Chaque type de tâche prend en paramètre des références contraintes sur des objets partagés de type `bloc`. Les contraintes d'accès sur ces références sont ajoutées en fonction du type d'accès réalisé dans la tâche sur les objets référencés.

On peut noter l'utilisation d'une référence avec un droit d'accès en accumulation **Shared_cw**<`add,bloc`> pour le paramètre `c` du type de tâche `gemm` à la ligne 23 du programme. Ceci indique qu'une accumulation avec la fonction d'accumulation `add` (réalisant la somme de deux blocs) est réalisée sur l'objet référencé par le paramètre `c`. Cette accumulation est effectuée à la ligne 24 par l'appel de la méthode `cumul` sur la référence contrainte `c`.

```
1 : struct Factorization_LU { // Factorisation LU séquentiel du bloc a
2 :    void operator()( Shared_r_w<bloc> a ) {
3 :       a.access().FactorisationLU();
4 : }};
5 : struct trsm_sup { // b ← b * sup(a)^-1
6 :    voit operator()( Shared_r<bloc> a,
7 :                     Shared_r_w<bloc> b ) {
8 :       b.access().dtrsm_sup( a.read() );
9 : }};
10 : struct trsm_inf { // b ← b * inf(a)^-1
11 :    void operator()( Shared_r<bloc> a,
12 :                     Shared_r_w<bloc> b ) {
13 :       b.access().dtrsm_inf( a.read() );
14 : }};
15 : struct add { // a ← a + b
16 :    void operator()(bloc& a, const bloc& b) {
17 :       a += b;
18 : }};
19 : struct gemm { // c ← c - a * b
20 :    void operator()( Shared_r<bloc> a,
21 :                     Shared_r<bloc> b,
22 :                     Shared_cw<add,bloc> c) {
23 :       c.cumul( - a.read() * b.read() );
24 : }};
25 : Factorization_LU_Par(matrix_bloc<Shared_rp_wp<bloc> >& A, int n ) {
26 :    int i, j, k;
27 :    for( k = 0; k < n; k++ ) {
28 :       Fork<Factorization_LU>()( A(k,k) );
29 :       for( i = k+1; i < n; i++ )
30 :          Fork<trsm_sup>()( A(k,k), A(i,k) );
31 :       for( j = k+1; j < n; j++ )
32 :          Fork<trsm_inf>()( A(k,k), A(k,j) );
33 :       for(i = k+1; i < n; i++ )
34 :          for( j = k+1; j < n; j++ )
35 :             Fork<gemm>()( A(i,k), A(k,j), A(i,j) );
36 :    }
37 : }
```

Figure 7.2 – *Écriture de la factorisation LU par bloc en Athapascan-1.*

Pour réaliser cette accumulation, il est nécessaire d'effectuer au préalable le produit – `a * b` sur une matrice temporaire. On pourrait penser qu'il est plus efficace d'effectuer en place sur le bloc `c` le calcul – `a * b + c`, comme le permet la fonction `xgemm` de la librairie BLAS. Cependant, dans ce cas, lors de l'exécution sur une machine à mémoire partagée, les tâches d'accumulation sur une même matrice ne pourraient pas être exécutées en parallèle.

La troisième étape consiste à écrire l'algorithme proprement dit en remplaçant les opérations sur les blocs par la création des tâches correspondantes. Pour la factorisation LU, l'écriture de cet algorithme est fournie à partir de la ligne 26 du programme de la figure 7.1.

En conclusion, nous pouvons noter que, de par la sémantique séquentielle du modèle de programmation d'Athapascan-1, l'écriture de l'algorithme en Athapascan-1 est identique à l'écriture séquentielle de l'algorithme 5 page 115. De plus, cette écriture particulièrement naturelle permet de décrire très précisément le parallélisme de ce programme.

7.4 Analyse dans le modèle d'exécution d'Athapascan-1

Dans cette section, nous présentons une analyse théorique de ces trois algorithmes dans le modèle d'exécution d'Athapascan-1 introduit au chapitre 4. Le tableau 7.1 décrit les différents paramètres caractérisant le graphe de flot de donnés généré par les trois algorithmes en fonction de la taille n de la matrice et de la taille k des blocs. Ces différents paramètres ont été précédemment définis dans la section 5.3.1 page 79.

	Produit de matrice	Factorisation LU et LL^t
T_1	$O(n^3)$	
T_∞	$O(k^3 + nk)$	$O(nk^2)$
C_1	$O(\frac{n^3}{k})$	
C_{max}	$O(k^2)$	$O(nk)$
n	$O((\frac{n}{k})^3)$	
n_d	$O((\frac{n}{k})^3)$	

Tableau 7.1 – *Caractérisation du graphe de flot de données produit par les algorithmes de produit de matrices et de factorisation. n désigne la taille des matrices et k désigne la tailles des blocs.*

Ces trois algorithmes peuvent être exécutés avec un ordonnancement statique puisque le graphe est entièrement connu après l'exécution de la tâche racine. L'exécution de ces trois programmes entre alors dans le cadre du modèle d'exécution spécialisé présenté dans la section 4.4.2.2.

Différents algorithmes d'ordonnancement statiques peuvent être utilisés pour exécuter ces programmes Athapascan-1. Si l'algorithme d'ordonnancement glouton ETF (ou ERT) est utilisé, alors l'exécution sur m processeurs identiques à mémoire distribuée est bornée en temps d'après le corollaire 6 page 90 par :

$$O\left(\left(\frac{n}{k}\right)^6 m\right) + O\left(\frac{n^3}{m}\right) + O(k^3 + nk)$$

pour le produit de matrice et par :

$$O\left(\left(\frac{n}{k}\right)^6 m\right) + O\left(\frac{n^3}{m}\right) + O(nk^2)$$

pour les deux factorisations.

Cependant le graphe généré par ces algorithmes possède une structure régulière. L'utilisation d'un algorithme d'ordonnancement statique de type glouton comme ETF n'est donc peut être pas la meilleure solution pour calculer un ordonnancement statique de ce graphe. Bien qu'ETF permette, d'après l'analyse de complexité, de recouvrir complètement les communications par les calculs sur une machine à mémoire distribuée sans contention ni surcoût induit par les communications, le volume total de communication produit par cet ordonnancement n'est borné que par $C_1 = O(\frac{n^3}{k})$. Comparé au travail $T_1 = O(n^3)$ de ces algorithmes, il est clair que les communications seront difficilement recouvertes sur un réseau de communication avec contention.

Un algorithme d'ordonnancement spécifique à ce type de graphe, réalisant un placement cyclique bidimmensionel des tâches, similaire au placement cyclique bidimmensionel des données présenté dans la section précédente, peut avantageusement être utilisé. Pour cela, chaque tâche est indexée par le couple d'indice (i, j) correspondant à l'indice, dans la matrice de blocs, du bloc modifié par la tâche. Si $p = q^2$ est le nombre de nœuds de la machine, alors les tâches sont allouées au processeur $(i \bmod q)q + (j \bmod q)$. Le volume total de communications obtenu avec cet ordonnancement est alors, comme pour le placement par bloc cyclique bidimensionel, $O(n^2 q)$.

L'ordonnancement de l'algorithme du produit de matrice pose également un problème si l'on cherche à minimiser l'espace mémoire utilisé. Par exemple, si les matrices de taille $n \times n$ sont partitionnées en $\sqrt{p} \times \sqrt{p}$ blocs, l'espace mémoire utilisé par processeur et sur p processeurs peut aller jusqu'à $O(\frac{n^2}{\sqrt{p}})$. L'espace total nécessaire est alors $O(n^2\sqrt{p})$ donc $\sqrt{p}$ fois l'espace mémoire nécessaire en séquentiel. Un ordonnancement des calculs et des communications efficace de cet algorithme, appelé algorithme de Cannon [79], permet de se ramener à un espace mémoire total en $O(n^2)$.

7.5 Expérimentation

Nous présentons dans cette section une évaluation expérimentale des trois programmes Athapascan-1 précédemment présentés, multiplication de matrice, factorisation LU et LL^t.

Les expérimentations ont été réalisées sur les deux machines parallèles suivantes :

- Un réseau de 20 stations SUN sous Solaris 7, (localisé à l'Universitée du Delaware), chacune contenant quatre processeurs Ultra Sparc II cadencé à 250 MHz et 512 méga octets de mémoire. La puissance de référence d'un processeur de cette machine est de 390 Mflops (million d'opérations flottantes par seconde) [1]. Les stations sont reliés entre elles par un réseau Myrinet. Pour les communications, nous avons utilisé MPICH-GM qui est une version de MPICH utilisant la bibliothèque de communication native GM de Myrinet [2]. Les performances mesurées sur MPICH-GM sont de 30 méga octets par seconde de bande passante pour un message de 80000 octets et d'environ 30 μs de latence pour des messages inférieurs à 500 octets.

- Une machine IBM-SP1 (localisée à l'IMAG de Grenoble) sous AIX-4.2, contenant 32 processeurs [3] RS-6000 cadencés à 120 MHz et 64 méga octets de mémoire. La puissance de référence 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 performances du réseau sur MPI sont alors de 30 méga octets par seconde de bande passante et de 60 μs de latence.

Il est à noter que les rapports calcul/communication de ces deux machines sont donc relativement différent puisque les bandes passantes sont pratiquement similaire alors que la puissance de calculs disponible sur chaque nœud est 100 Mflops pour le SP1 et de 4×390 Mflops sur le réseau de station.

Les différentes applications Athapascan-1 et ScaLapack ont été compilées sur les deux machines par `gcc -O3` (version 2.95) et utilisent les bibliothèques BLAS fournies par les constructeurs.

Les résultats sont exprimés en millions d'opérations flottantes réalisées par seconde (Mflops). Si t est le temps d'exécution parallèle obtenu sur un problème nécessitant a

1. Obtenue sur le produit de deux matrices de tailles 100×100 par la fonction dgemm de la bibliothèque BLAS fournie par SUN (Sun Performance Library 2.0). Le nombre d'opérations flottantes comptées pour un produit de matrice de taille n est $2n^3$.

2. GM est une bibliothèque de communication par échange de messages exploitant l'interface Myrinet au niveau utilisateur (*i.e.* communication directe à travers la carte réseau sans passer par le système d'exploitation).

3. Seuls 21 nœuds étaient en service à la date des expérimentations.

4. Obtenue sur le produit de deux matrices de tailles 150×150 par la fonction dgemm de la bibliothèque BLAS fournie par IBM.

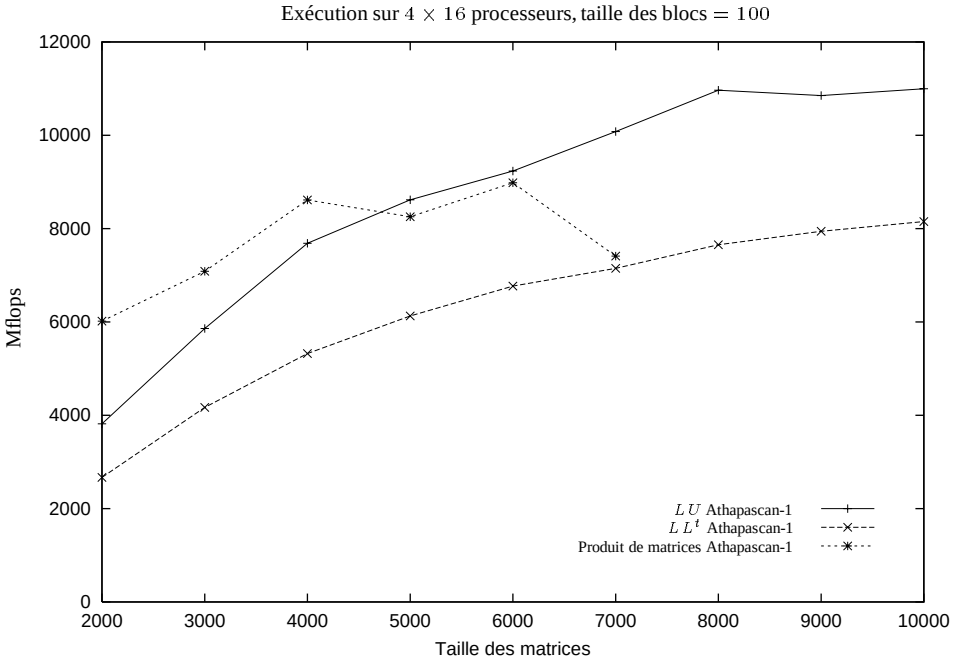

Figure 7.3 – *Exécution sur un réseau de 16 stations SUN Sparc SMP à 4 processeurs.*

opérations flottantes, alors le nombre de Mflops obtenu est $\frac{a}{t}10^{-6}$. Une approximation du nombre d'opérations flottantes utilisées par les trois algorithmes, multiplication de matrice, factorisation LU et LL^t a été obtenue avec les fonctions `dopbl3` et `dopla` de la bibliothèque LAPACK (*LAPACK timing routine*).

7.5.1 Placement cyclique bidimensionel des tâches

Nous présentons ici les résultats obtenus avec la meilleure configuration, c'est-à-dire avec le mode d'exécution statique de la bibliothèque Athapascan-1 ainsi qu'un algorithme d'ordonnancement réalisant un placement cyclique bidimensionel des tâches comme présenté dans la section précédente.

La figure 7.3 page 121 montre les performances obtenues sur 16 nœuds (64 processeurs) du réseau de stations SMP avec une partition des matrices en blocs de taille 100. L'exécution du programme est ici réalisée en lançant sur chaque nœud SMP un processus UNIX, contenant lui même 4 processus légers pour exploiter les 4 processeurs du nœud SMP. Avec cette configuration, la factorisation LU exploite jusqu'à 11000 Mflops, ce qui donne 172 Mflops par processeur, à comparer aux 390 Mflops de puissance de référence obtenue sur un produit de matrices.

La figure 7.4 page 122 montre les performances obtenues sur 16 nœuds du SP1 avec

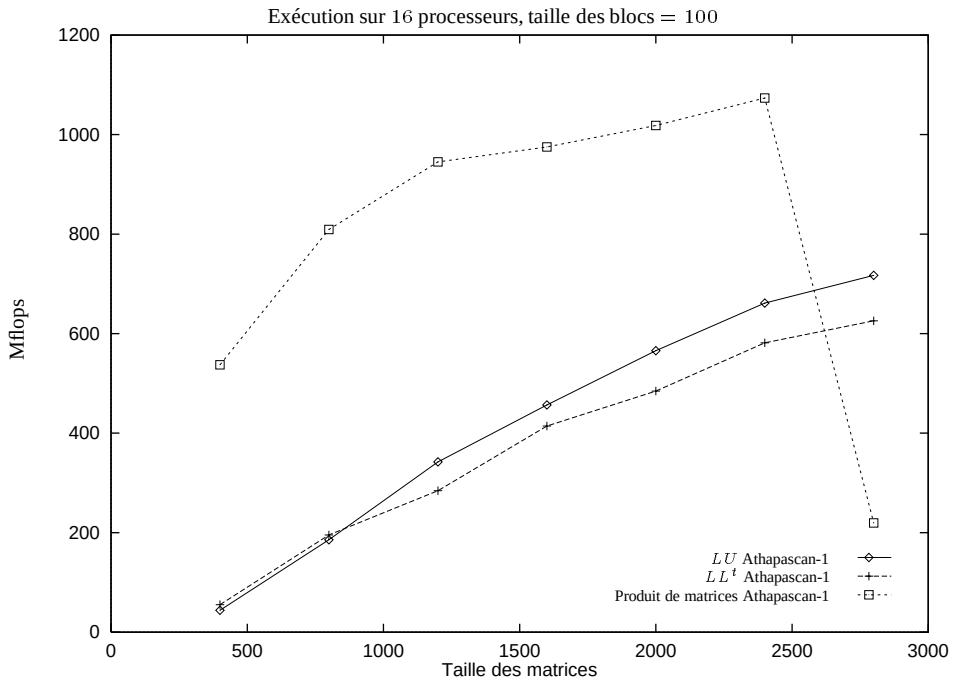

Figure 7.4 – *Exécution sur un IBM SP1 à 16 processeurs.*

une partition des matrices en blocs de taille 100. L'exécution du programme est ici réalisée en lançant un processus UNIX sur chaque nœud mono-processeur de la machine. Le produit de matrice exploite alors jusqu'à 1100 Mflops, ce qui donne 68 Mflops par processeurs, à comparer aux 100 Mflops de puissance de référence mesurée sur un processeur avec le produit de matrices. La chute des performances pour le produit de matrices de taille 2800×2800 s'explique par un dépassement de la capacité mémoire sur un nœud du SP1 (limité à 30 méga octets de données utilisateurs résidant en mémoire centrale[5]). Les factorisations sont moins consommatrices de mémoire (une seul matrice à stocker contre trois pour le produit de matrice), le phénomène n'apparaît donc pas pour les tailles de matrice utilisées.

7.5.2 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 [23].

Les deux algorithmes Athapascan-1 et ScaLapack sont identiques : La factorisation

5. Au delà, la mémoire est systématiquement transférée sur disque, le système AIX se réservant les 30 méga octets restants.

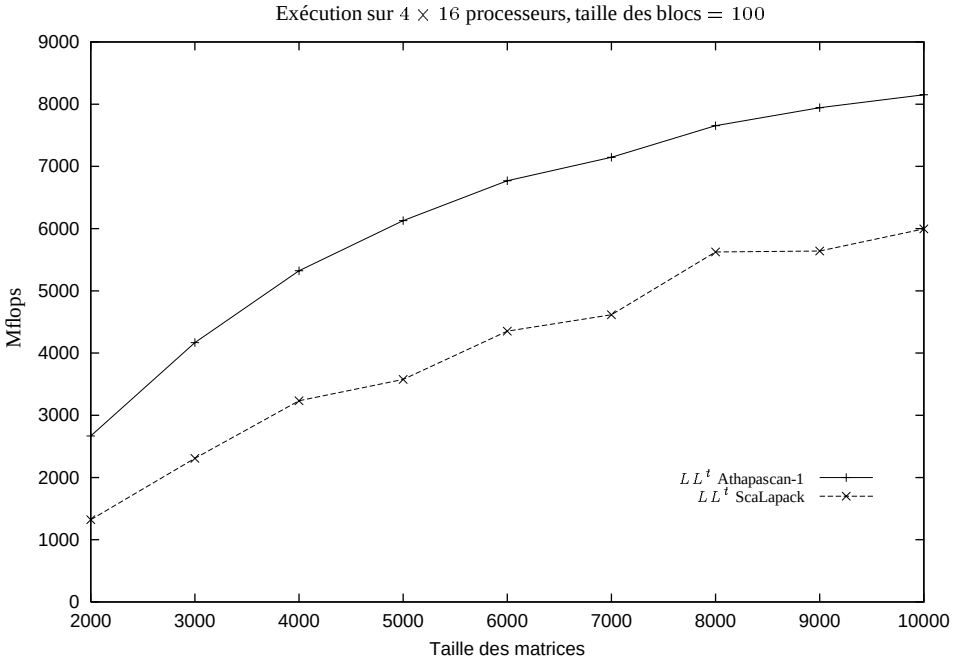

Figure 7.5 – *Comparaison Athapascan-1/ScaLapack sur un réseau de 16 stations SUN.*

est réalisée sans pivotage, la matrice est partitionnée en blocs de même taille 100 et le même placement cyclique bidimensionel des blocs sur les nœuds est effectué.

La figure 7.5 page 123 montre une comparaison des performances Athapascan-1 et ScaLapack obtenues sur 16 nœuds (64 processeurs) du réseau de stations SMP. Les performances obtenu par Athapascan-1 sont jusqu'à deux fois meilleures que celles obtenues avec ScaLapack. Ces résultats s'expliquent par l'utilisation dans Athapascan-1 de processus légers pour exploiter les nœuds SMP alors que ScaLapack utilise quatre processus UNIX communiquant par échange de messages.

La figure 7.6 page 124 montre une comparaison des performances Athapascan-1 et ScaLapack obtenues sur un nœud 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 des processus UNIX pour exploiter un nœud SMP améliore jusqu'à deux fois les performances obtenues. Ceci explique en partie les écarts de performances mesurés entre Athapascan-1 et ScaLapack sur réseau de SMP.

La figure 7.7 page 125 montre une comparaison des performances Athapascan-1 et ScaLapack obtenues sur 16 nœuds de la machine SP1. Sur cette machine composée de

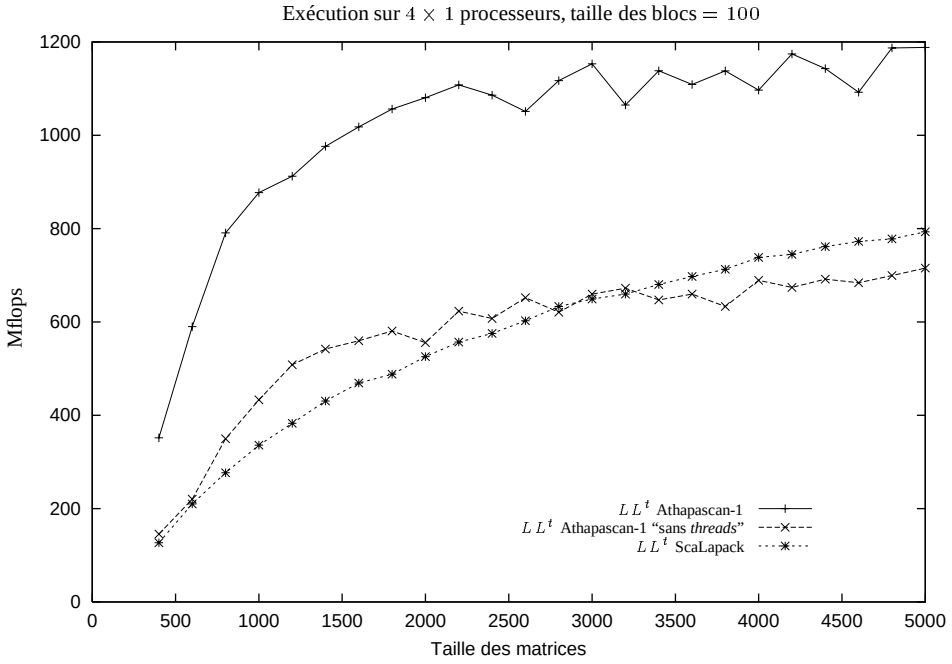

Figure 7.6 – *Comparaison Athapascan-1/ScaLapack sur une machine SMP.*

nœuds mono-processeur, 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. Dans ScaLapack, les communications sont réalisées par la bibliothèque de communication BLACS [31] qui ne fournit que des mécanismes de communication synchrones (communication point à point ou collectives). Au contraire, dans Athapascan-1, les communications sont toutes asynchrones.

7.5.3 Évaluation d'Athapascan-1 sur d'autres politiques d'ordonnancement

La bibliothèque Athapascan-1 intègre plusieurs politiques d'ordonnancement différentes, pouvant être utilisées sur une même application sans modification de celle-ci (mise à par bien sûr, l'indication de la politique d'ordonnancement à utiliser). Dans cette section, nous comparons expérimentalement, sur machine à mémoire distribuée, différentes politiques d'ordonnancements sur la factorisation dense de Cholesky.

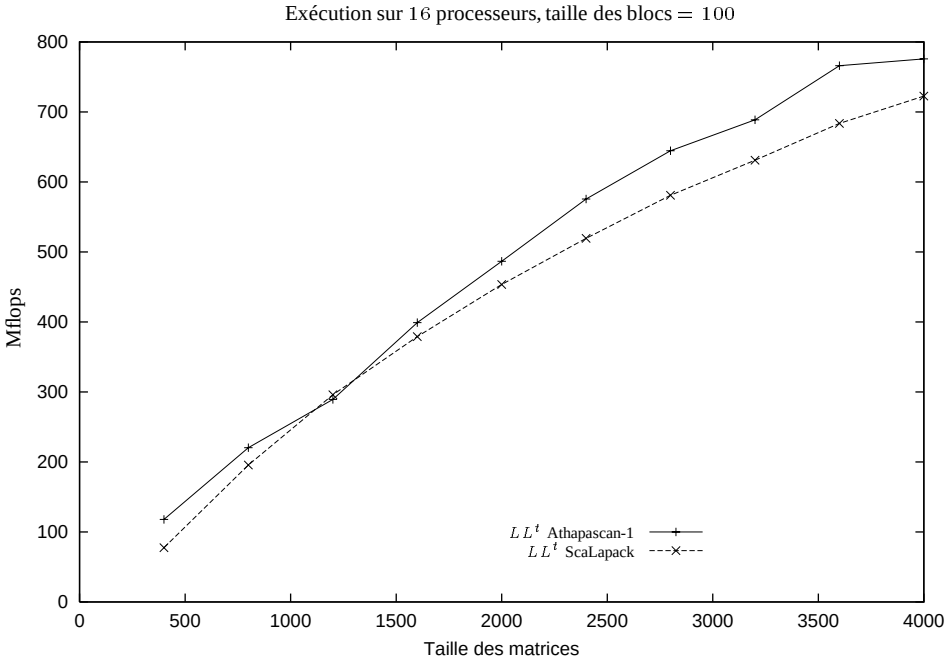

Figure 7.7 – *Comparaison Athapascan-1/ScaLapack sur un IBM SP1 à 16 processeurs.*

7.5.3.1 Ordonnancement statique

Tout d'abord, nous présentons les résultats obtenus avec deux politiques d'ordonnancement statiques : ETF et DSC. Ces deux algorithmes d'ordonnancement statique nécessitent en entrée une modélisation de la machine et plus précisément une modélisation du réseau de communication. Le modèle de communication de ces deux algorithmes d'ordonnancement statique est caractérisé par trois paramètres : ω, le nombre d'opérations par seconde réalisées sur un nœud ainsi que β et τ la latence et le débit du réseau. Notons que ce modèle de communication ne prend pas en compte les éventuelles contentions du réseau de communication. Il ne peut donc être valide que si la bande passante du réseau est suffisante par rapport au volume de communication généré par l'application. Dans le cas contraire, on peut s'attendre à obtenir un ordonnancement statique des tâches conduisant à une exécution réelle peu performante.

La figure 7.8 page 126 compare les performances obtenues sur les deux algorithmes d'ordonnancement généraux ETF et DSC par rapport aux performances produites par l'ordonnancement cyclique bidimensionnel déjà évalué précédemment. L'application consiste en la factorisation de Cholesky dense d'une matrice de taille 2400×2400 et les valeurs prises pour modéliser le SP1 sont $\omega = 100 Mflops$ pour la puissance de calcul d'un processeur, $beta = 60 \mu s$ pour la latence et $\tau = 10 Mbyte/s$ pour la bande passante. On peut

alors voir que les performances obtenus avec l'ordonnancement ETF sont meilleures que celle obtenue avec l'ordonnancement cyclique bidimensionnel. Par contre, l'ordonnancement fournie par DSC se dégrade fortement à partir de 10 processeurs (pas d'explication si ce n'est une erreur possible dans le code, directement extrait de PYROS).

Les temps affichés dans la figue 7.8 page 126 ne prennent cependant pas en compte le temps de calcul de l'ordonnancement. Pour l'algorithme ETF, dont la complexité est en $O(np \log n)$, ce temps de calcul varie entre $1s$ pour 4 processeurs et jusqu'à $1.5s$ pour 16 processeurs alors que pour l'algorithme DSC, qui à une complexité en $O((n+n_d)\log n)$[6] ce temps de calcul, varie de $1.6s$ à $2.2s$. Ce coût d'ordonnancement intervient donc au maximum dans 16% du temps total d'exécution pour l'algorithme ETF sur 16 processeurs.

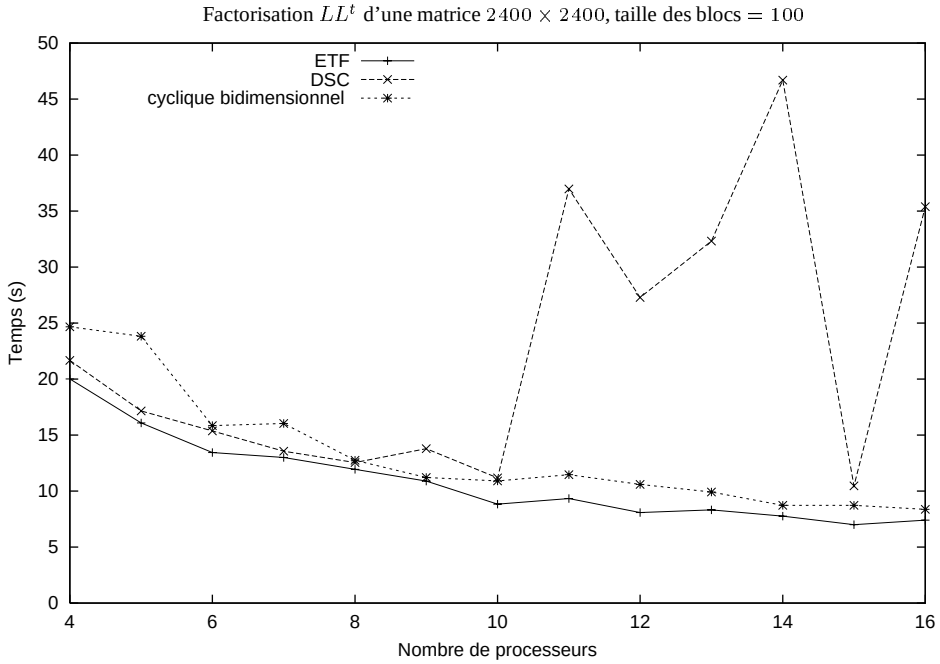

Figure 7.8 – *Exécution sur le SP1 avec la version spécialisée de la bibliothèque Athapascan-1 avec trois politiques d'ordonnancement statiques de la factorisation de Cholesky LL^t.*

6. Cette complexité de l'algorithme ETF diffère de celle donnée dans [69] et dans la section 5.4.2.1 page 84. Elle est obtenue en utilisant des arbres de recherches binaires équilibrés pour calculer à chaque itération de l'algorithme la tâche qui pourra s'exécuter au plus tôt sur un processeur.

7.5.3.2 Ordonnancement dynamique

Nous présentons maintenant une expérimentation de la factorisation dense de Cholesky en utilisant la version générale de la bibliothèque Athapascan-1, pouvant à la fois supporter des politiques d'ordonnancement statiques et à la volée. Deux politiques d'ordonnancement sont expérimentées ici. Un ordonnancement à la volée de type glouton avec vole de tâche (*i.e.* lorsqu'un processeur n'a plus de tâche prête, il vole une tâche prête à un processeur choisi au hasard) et l'ordonnancement cyclique bidimensionnel utilisé précédemment.

La figure 7.9 page 128 montre les résultats obtenus sur le SP1 pour la factorisation de Cholesky sur différentes tailles de matrice. Deux problèmes peuvent alors être observés. Tout d'abord, les résultats obtenus avec l'ordonnancement cyclique bidimensionnel sont bien moins performants que ceux obtenus avec un ordonnancement identique mais sur la version spécialisée de la bibliothèque (voire figure 7.7 page 125). Ceci peut s'expliquer par le surcoût plus important de gestion et d'interprétation du graphe dans le mode d'exécution dynamique, comme nous l'avons déjà souligné dans la section 6.5 page 103. Le second problème concerne l'écart de performance obtenu entre les deux politiques d'ordonnancement sur le même mode d'exécution dynamique de la bibliothèque. Ceci s'explique par le fait que l'algorithme glouton utilisé ne tient aucun compte de la localité des données et génère alors beaucoup de communications de données entre processeurs.

7.6 Conclusion

Nous avons présenté dans ce chapitre trois applications de bases de l'algèbre linéaire dense implantées en Athapascan-1 : le produit de matrice, la factorisation LU et la factorisation LL^t. Ces trois applications ont permis d'illustrer certains point d'Athapascan-1 :

- La facilité de programmation de ce type d'application dans l'interface de programmation d'Athapascan-1. Le programmeur est libéré de l'analyse du parallélisme de son programme mais surtout des problèmes de distribution et de communications des données. Il est également libéré des problèmes d'ordonnancement des calculs, bien que dans le cas des applications régulières traitées ici, ce problème ne soit pas aussi difficile que dans le cas d'application irrégulière (le chapitre 9 traitera un exemple d'application irrégulière en Athapascan-1).

- La portabilité apportée par l'abstraction de la machine parallèle dans l'écriture du programme. Le système chargé de l'exécution peut alors tirer pleinement partie des différentes architectures parallèles. Par exemple sur les machines de types SMP ou réseau de SMP, le système d'exécution peut avantageusement utiliser des processus légers et la mémoire partagée de la machine, ce qui n'est par exemple pas possible pour une application écrite dans un modèle de programmation par échange

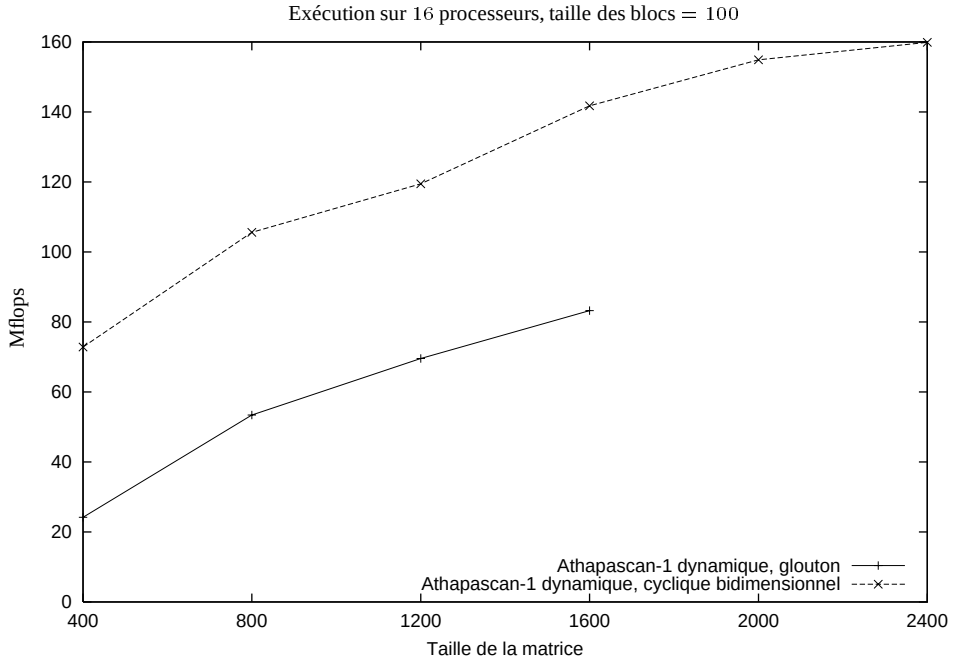

Figure 7.9 – *Exécution sur le SP1 avec la version générale de la bibliothèque Athapascan-1 avec deux politiques d'ordonnancement, de la factorisation LL^t.*

de messages entre processus lourds. Cet avantage a été illustré dans la section 7.5.2 page 122 où l'application de factorisation de Cholesky dense en Athapascan-1 améliore jusqu'à 100% les performances obtenue par ScaLapack sur une machine SMP et améliore jusqu'à 100% sur une machine de type réseau de SMP.

– La possibilité offerte par Athapascan-1 d'utiliser différentes politiques d'ordonnancement. L'ordonnancement cyclique bidimensionnel spécifique à ce type d'application fournit souvent de bons résultats ; mais nous avons vu par exemple dans la section 7.5.3 page 124 que sur certaines machines et pour certaines tailles de problèmes un autre ordonnancement comme ETF peut permettre d'obtenir une amélioration significative des performances.

8

Factorisation parallèle creuse de Cholesky

Ce chapitre est consacré à la factorisation parallèle creuse de Cholesky. Les phases préparatoires à la factorisation numérique proprement dite, renumérotation et factorisation symbolique, nécessaires pour obtenir un algorithme efficace sont présentées dans un cadre général. Nous présentons ensuite (section 8.5.4 page 142) un algorithme de factorisation numérique parallèle pour machines à mémoire distribuée, de type fan-in, avec partitionnement par blocs bidimensionnel de la matrice. Cet algorithme est décrit dans un modèle de programmation par échange de messages. Un ordonnancement des calculs par placement initial des données de la matrice est ensuite proposé pour cet algorithme.

8.1 Introduction

La résolution de systèmes linéaires creux symétriques définis positifs intervient dans beaucoup de problèmes comme par exemple les méthodes d'éléments finis, l'analyse des structures, la simulation de semi-conducteurs... De plus, pour ces applications, une partie importante du temps d'exécution est passée dans la résolution de ces systèmes linéaires.

Pour résoudre un tel système $Ax = b$, deux classes de méthodes, directes et itératives, peuvent être utilisées. La méthode directe consiste à factoriser la matrice initiale A pour obtenir une matrice L triangulaire inférieure telle que $A = LL^T$. Cette factorisation est obtenue en utilisant une variante creuse de l'algorithme de factorisation de Cholesky présentée au chapitre précédent. La solution du système est ensuite obtenue en résolvant les deux systèmes triangulaires $Ly = b$ puis $L^T x = y$. Les méthodes itératives consistent quant à elles à obtenir une approximation de la solution par des itérations faisant intervenir

des produits de type matrice creuse par un vecteur.

Bien que les méthodes directes soient généralement plus coûteuses en mémoire et en temps d'exécution que les méthodes itératives, elles ont l'avantage d'être plus robustes et générales que ces dernières. En effet, pour résoudre efficacement un système linéaire par une méthode itérative il faut choisir le schéma itératif le mieux adapté au système. Ensuite, il faut généralement préconditionner la matrice pour se ramener à un système sur lequel le schéma itératif sera plus efficace. La encore, pour cette opération appelée préconditionnement, il existe différentes méthodes. De plus, pour des raisons de stabilité, il existe certains systèmes sur lesquels seules les méthodes directes sont utilisables.

8.2 Présentation de la factorisation creuse de Cholesky

L'algorithme de la factorisation creuse de Cholesky est une élimination de Gauss adaptée au cas symétrique dans lequel les opérations sur les éléments nuls sont supprimées. L'algorithme 7 présente la forme kij de l'algorithme où $col(k)$ est l'ensemble des indices de ligne des éléments non nuls de la colonne k de la matrice L située sous la diagonale, c'est-à-dire. $col(k) = \{i, L_{i,k} \neq 0, i > k\}$.

Algorithme 7 : Algorithme de factorisation creuse de Cholesky.

Entrée : matrice A symétrique définie positive de taille n
Sortie : $LL^T = A$

1: L est initialisée avec A
2: **pour** $k = 1$ **jusqu'à** n **faire**
3: $\quad L_{k,k} \leftarrow \sqrt{L_{k,k}}$
4: $\quad$ **pour** $i \in col(k)$ **faire**
5: $\quad\quad L_{i,k} \leftarrow \frac{L_{i,k}}{L_{k,k}}$
6: $\quad$ **pour** $j \in col(k)$ **faire**
7: $\quad\quad$ **pour** $i \in col(k)$ et $i \leq j$ **faire**
8: $\quad\quad\quad L_{i,j} \leftarrow L_{i,j} - L_{i,k} L_{j,k}^t$

Une propriété bien connue et importante des matrices symétriques définies positives est l'existence et l'unicité de la décomposition $A = LL^T$ avec $L_{i,i} > 0$. Cette propriété assure de toujours obtenir un pivot strictement positif $L_{k,k}$ à la ligne 3 de l'algorithme. On montre également [80] qu'au cours de la factorisation, les valeurs des coefficients de la matrice L sont bornées par le plus grand coefficient de la matrice initiale. Il n'est donc pas nécessaire de réaliser des permutations de lignes et de colonnes sur la matrice à factoriser pour réduire les erreurs d'arrondi. Pour la même raison, il est possible d'effectuer des

permutations sur la matrice initiale sans engendrer d'erreur d'arrondi supplémentaire au cours de la factorisation. Cette propriété va être utilisée dans le paragraphe suivant.

La factorisation creuse de Cholesky a pour conséquence un remplissage de la matrice factorisée L qui contient alors plus d'éléments non nuls que la matrice initiale A. Lors de l'affectation de la ligne 8 de l'algorithme, si $L_{i,j}$ avant l'affectation est nul alors un nouvel élément non nul est introduit dans la matrice factorisée L. Un exemple de matrice creuse A et du remplissage du facteur L obtenu est donné dans la figure 8.1. Le nombre d'éléments non nuls introduits est fortement influencé par une permutation des lignes et des colonnes appliquée à la matrice. Une première étape de la factorisation creuse de Cholesky consiste donc à calculer une permutation P qui minimise le remplissage lors de la factorisation de la matrice PAP^T, dans le but de réduire l'espace mémoire et le nombre d'opérations utilisées pour la factorisation. Cette permutation correspond à une renumérotation des inconnues du système linéaire associé à la matrice A ; c'est pour cette raison que cette opération est habituellement dénommée étape de renumérotation. Plusieurs méthodes existent pour réaliser cette renumérotation. Elles seront présentées dans la section 8.3.

Comme seuls les éléments non nuls sont stockés, le remplissage de la matrice L en cours de factorisation peut impliquer une gestion dynamique coûteuse de la structure stockant cette matrice s'il n'est pas bien géré. Cependant, dans le cadre de la factorisation creuse de Cholesky, il existe des algorithmes permettant, pour un faible coût, de précalculer la structure finale de la matrice L, c'est-à-dire la position dans la matrice factorisée des éléments non nuls (voir la section 8.4). Le principe de ces algorithmes est de réaliser l'algorithme 7, mais uniquement sur la structure creuse de la matrice, sans considérer les valeurs. Pour cette raison, cette opération se nomme «factorisation symbolique». C'est également grâce à cette opération de factorisation symbolique qu'il est possible, lors de la parallélisation de la factorisation creuse de Cholesky, de réaliser un ordonnancement statique des calculs de l'étape suivante. La section 8.4 sera consacrée à cette phase de la factorisation.

La dernière étape, la factorisation numérique peut alors être réalisée en appliquant l'algorithme 7 sur une matrice ayant la structure creuse de la matrice factorisée L et ayant pour valeur PAP^T. Cette étape est généralement beaucoup plus coûteuse en nombre d'opérations que les étapes précédentes. Nous traiterons en détail cette factorisation numérique dans la section 8.5.

Un concept important pour la compréhension de la factorisation creuse de Cholesky est l'arbre d'élimination associé à la factorisation [84]. À chaque colonne de la matrice à factoriser est associé un nœud de l'arbre. Un nœud i a pour père le nœud j si et seulement si l'indice du premier élément non nul situé en dessous de la diagonale de la colonne i de la matrice factorisée L est égal à j. La figure 8.1 page 132 donne un exemple de facteur L obtenu après factorisation et l'arbre d'élimination qui lui est associé.

L'arbre d'élimination exhibe directement une partie importante du parallélisme de la factorisation creuse de Cholesky. Plus exactement, il exprime les dépendances entre deux

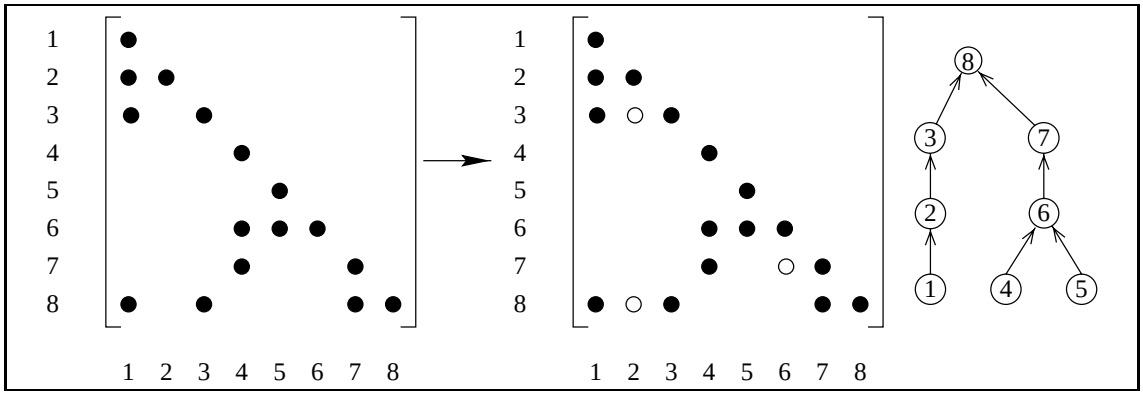

Figure 8.1 – *Structure d'une matrice creuse symétrique A, du facteur L obtenu après factorisation et l'arbre d'élimination associé. Les cercles blancs représentent les éléments rajoutés dans la structure du facteur.*

itérations différentes de la boucle externe (*i.e.* la boucle sur l'indice k de la ligne 2) de l'algorithme 7. L'itération d'indice i ne peut commencer que lorsque l'ensemble des itérations d'indice j où j appartient au sous arbre du nœud i dans l'arbre d'élimination. Il faut noter que ce parallélisme est uniquement induit par le creux de la matrice. Lorsque la matrice est dense, l'arbre d'élimination est une chaîne et les itérations sur la boucle externe de l'algorithme (ligne 2 de l'algorithme 7 page 130) doivent alors être sérialisées. Le parallélisme de la factorisation dense de Cholesky est alors uniquement tiré des itérations sur les deux boucles internes (ligne 6 et 7 de l'algorithme 7 page 130).

Un super-nœud est une autre notion importante dans la factorisation creuse de Cholesky. Un super-nœud est un ensemble de colonnes adjacentes du facteur L ayant la même structure creuse pour les éléments situés en dessous de la diagonale. Plus précisément, l'ensemble des colonnes d'indice $j, \ldots, j+t$ constitue un super nœud si, pour tout $j \leq k < j+t$, l'ensemble des indices de ligne des éléments non nuls de la colonne k est égale à l'ensemble des indices de ligne des éléments non nuls de la colonne $k+1$ augmenté de l'indice k. La matrice de la figure 8.2 a, par exemple, 5 super-nœuds, composés des ensembles de colonnes d'indices $\{1,2,3\}$, $\{4\}$, $\{5\}$, $\{6\}$ et $\{7,8\}$.

Une technique majeure pour optimiser la factorisation numérique de Cholesky est de réorganiser les calculs en fonction de ces super-nœuds de manière à remplacer les opérations réalisées sur la structure creuse de la matrice par des opérations réalisées sur des blocs denses. En effet, il est ainsi possible d'utiliser les BLAS de niveau 3 [33] pour réaliser efficacement ces opérations. Cette technique sera abordée dans la section 8.5.

On peut également se reporter à l'article [62] et au livre [32] pour une étude complète de la factorisation parallèle creuse de Cholesky.

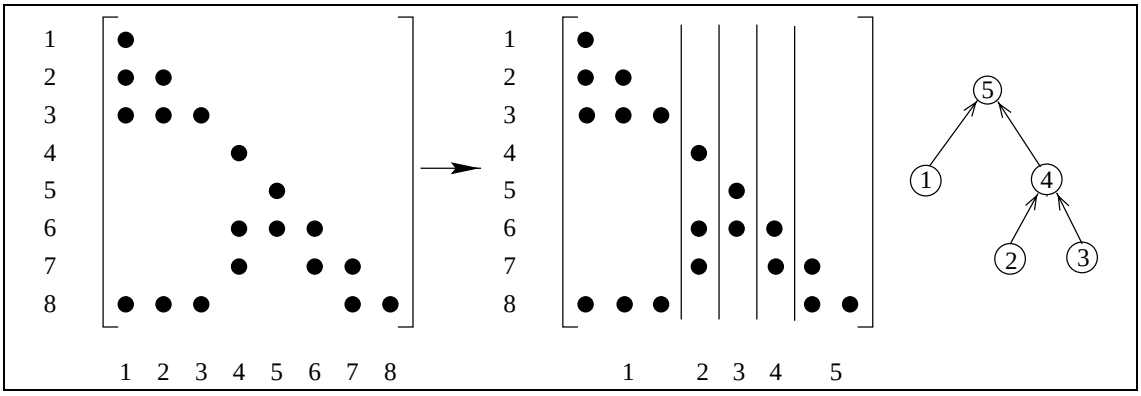

Figure 8.2 – *Les super-nœuds obtenus sur la structure du facteur de Cholesky de la figure 8.1 et l'arbre d'élimination des super-nœuds associés.*

8.3 Renumérotation des inconnues

Le calcul d'une renumérotation des inconnues d'un système linéaire creux symétrique qui minimise le remplissage de la matrice lors de la factorisation de Cholesky, est NP-difficile [129]. Ainsi des heuristiques ont été proposées pour calculer une bonne renumérotation. On distingue deux classes principales : degré minimum et dissection emboîtée. Toutes deux sont basées sur le graphe d'adjacence associé à la matrice à factoriser.

Le graphe d'adjacence d'une matrice symétrique A de taille $n \times n$, notée $G(A)$ est un graphe non orienté, contenant n nœuds et dans lequel les nœuds distincts i et j sont reliés par une arête si et seulement si $A_{i,j}$ est non nul.

8.3.1 Méthode de degré minimum et de remplissage local minimum

Le remplissage de la matrice intervient lors de l'opération de mise à jour de la ligne 8 de l'algorithme 7 page 130. Les méthodes considérées ici cherchent alors à réduire le remplissage introduit entre deux itérations sur la boucle externe de l'algorithme (*i.e.* la boucle sur l'indice k). Ces méthodes sont donc des méthodes locales, c'est-à-dire qu'elles ramènent le problème de minimisation du remplissage entre la matrice initiale et la matrice factorisée à un problème de minimisation du remplissage entre deux itérations de l'algorithme de factorisation.

Pour minimiser ce remplissage (appelé dans la suite remplissage local) qui intervient lors de la $k^{ème}$ itération sur la boucle externe de l'algorithme de factorisation, ces méthodes appliquent une permutation sur les colonnes d'indice supérieur ou égal à k (la même permutation est également appliquée sur les lignes). En fait, seule une permutation de la colonne d'indice k avec une des colonnes d'indice supérieur à k est nécessaire : en

effet, les permutations entre colonnes d'indices strictement supérieurs à k ne modifient pas le remplissage local. Notons que la permutation de deux colonnes correspond à une permutation de deux inconnues du système.

La méthode de remplissage locale minimum calcule la solution exacte au problème de minimisation locale : elle cherche, parmi l'ensemble des $n-k$ permutations possibles, la permutation produisant le remplissage local minimum. Par contre, la méthode de degré minimum réalise une approximation du problème de minimisation locale. Dans cette méthode, c'est la colonne ayant le moins d'éléments non nuls parmi les éléments d'indice de ligne supérieur ou égal à k qui est choisie pour être permutée avec la colonne d'indice k. La méthode de remplissage local minimum conduit à une solution globale (*i.e.* une renumérotation des inconnues) réduisant d'environ 15% le nombre d'éléments non nuls par rapport à la méthode de degré minimum [114] ; cependant cette dernière est préférée pour son moindre coût de calcul.

En pratique ces deux algorithmes ne travaillent pas directement sur la matrice à factoriser mais sur le graphe d'élimination correspondant au graphe d'adjacence de la sous matrice restant à factoriser. L'algorithme de degré minimum se ramène alors à une recherche dans ce graphe du nœud de degré minimum (le degré d'un nœud est le nombre d'arrêts reliées à ce nœud). Plusieurs optimisations de ces deux algorithmes, basées sur la théorie des graphes, ont été proposées. L'article [53] fait un survol des différentes optimisations proposées pour la méthode de degré minimum. L'article [2] propose également une variante de l'algorithme de degré minimum. Cet algorithme conduit à une renumérotation d'aussi bonne qualité que celle produite par l'algorithme de degré minimum mais avec un coût de calcul plus faible.

8.3.2 Méthode de dissection emboîtée

Le premier algorithme de dissection emboîtée a été proposé par Alan George [52]. Cet algorithme, initialement valable pour les matrices dont le graphe d'adjacence est une grille bidimensionnelle, a ensuite été étendu pour des matrices symétriques quelconques [83].

Cet algorithme est basé sur la notion de séparateur du graphe d'adjacence $G(A)$ de la matrice. Un séparateur d'un graphe G est un ensemble de nœuds S tel que si on enlève de G le sous ensemble de nœuds S et les arêtes qui y sont reliées, alors on obtient deux sous graphes non connectés. Le principe de l'algorithme de dissection emboîtée est le suivant : On extrait sur le graphe $G(A)$ un séparateur S séparant le graphe en deux parties G_1 et G_2 et l'on renumérote les inconnues de manière à avoir en premier les nœuds de G_1 puis ceux de G_2 et en dernier les nœuds du séparateur S. On applique ensuite récursivement cette méthode sur les sous graphes G_1 et G_2 pour obtenir la renumérotation finale. L'idée de cet algorithme est d'éviter, lors des itérations sur les nœuds de G_1, le remplissage sur les nœuds de G_2 et lors des itérations sur les nœuds de G_2, le remplissage sur les nœuds de G_1.

La qualité de la renumérotation obtenue est directement liée à la taille des séparateurs.

Si le graphe est de structure régulière, il est relativement facile d'obtenir de bons séparateurs. Par exemple l'algorithme initial de Alan George [52] sur une grille à 2 dimensions conduit à une renumérotation optimale. Mais dans le cas où le graphe est irrégulier (*i.e.* de structure quelconque), obtenir de bons séparateurs du graphe est un problème difficile. Les techniques de partitionnement de graphes irréguliers sont généralement utilisées dans ce cas. Le partitionnement du graphe fournit un séparateur des arêtes de ce graphe, c'est-à-dire un ensemble d'arêtes qui sépare le graphe en deux parties. Il est alors possible d'extraire, parmi les nœuds connectés à ce séparateur, un sous-ensemble de nœuds qui soit un séparateur du graphe.

Pour le partitionnement d'un graphe quelconque, il existe plusieurs méthodes mais la plus employée est la méthode de type multi-niveaux [74]. Utilisée dans le cadre de la renumérotation d'une matrice creuse, cette méthode de partitionnement donne souvent les meilleurs résultats comparée à d'autres algorithmes de partitionnement [74]. Une implantation de cette dernière méthode de renumérotation a, par exemple, été réalisée dans les bibliothèques METIS [73] et SCOTCH [99].

8.3.3 Méthode mixte

Les méthodes de dissection emboîtée génèrent généralement plus de parallélisme pour la factorisation numérique que les méthodes de degré minimum. Ceci se remarque sur l'arbre d'élimination de la factorisation qui est plus large et mieux équilibré.

L'exemple classique d'une matrice tri-diagonale de taille n illustre l'avantage des méthodes de dissection emboîtée par rapport aux méthodes de degré minimum sur ce point. Sur cette matrice, l'arbre d'élimination obtenu à partir d'une renumérotation par un algorithme de degré minimum est une chaîne. Il n'y a donc pas de parallélisme et le temps de factorisation parallèle est en $O(n)$. Par contre, l'arbre d'élimination obtenu à partir d'une renumérotation par une dissection emboîtée est un arbre binaire équilibré. Le temps de factorisation parallèle est alors en $O(\log n)$.

Cependant les méthodes de degré minimum sont connues pour être très efficaces tant en terme de remplissage qu'en coût de calcul de la renumérotation. Pour ces raisons, plusieurs méthodes mixtes combinant une dissection emboîtée avec un algorithme de degré minimum ont récemment été proposées [100] [64]. Le principe est alors de réaliser une dissection emboîtée jusqu'à un certain seuil, puis de continuer la renumérotation de chacune des partitions indépendantes obtenues avec un algorithme de degré minimum. Cette technique permet alors de calculer efficacement une renumérotation offrant de bonnes propriétés pour exploiter le parallélisme de la factorisation numérique.

8.4 Factorisation symbolique

Comme nous l'avons vu en introduction, l'objectif de cette étape est de pré-calculer la structure creuse du facteur L qui sera obtenu après la factorisation de façon à éviter de gérer des structures de données dynamiques pendant la factorisation. Cette étape n'est donc utile que si le nombre d'opérations nécessaires pour l'effectuer est faible devant le nombre d'opérations de la factorisation numérique.

Dans la suite on définit pour une matrice creuse A l'ensemble $Struct(A_{*,j})$ comme l'ensemble des indices de lignes des éléments non nuls de la colonne j de A et situés en dessous de la diagonale :

$$Struct(A_{*,j}) = \{i \text{ tel que } i > j \text{ et } A_{i,j} \neq 0\}$$

Nous définissons également la fonction p,

$$p(j) = \begin{cases} \min Struct(L_{*,j}) & \text{si } Struct(L_{*,j}) \neq \emptyset, \\ j & \text{sinon.} \end{cases}$$

et l'ensemble R_i des indices des colonnes j pour lesquels l'élément $L_{i,j}$ est le premier élément non nul situé en dessous de la diagonale :

$$R_i = \{j \text{ tel que } 1 < j < i \text{ et } p(j) = i\}$$

On montre alors relativement facilement [54] que, si L est la matrice obtenue après factorisation de A, alors :

$$Struct(L_{*,j}) = Struct(A_{*,j}) \cup \left(\bigcup_{j' \in R_j} Struct(L_{*,j'}) \right) - \{j\}$$

Autrement dit, la structure de la colonne j de L est l'union des structures de la colonne j de A et des colonnes j' de L pour lesquelles l'élément $L_{i,j'}$ est le premier élément non nul situé en dessous de la diagonale.

De plus on montre que les ensembles R_j sont une représentation de l'arbre d'élimination associé à la factorisation. Les indices de l'ensemble R_j sont en effet les indices des fils du nœud j dans l'arbre d'élimination.

On obtient alors, à partir de ces propriétés, un algorithme de factorisation symbolique de Cholesky, présenté dans la figure 8. La complexité en nombre d'opérations de cet algorithme est bornée par $O(d_{max}\eta(L))$ où $\eta(L)$ est le nombre d'éléments non nuls de la matrice L et d_{max} est le nombre maximum de fils d'un nœud de l'arbre d'élimination. Lorsque la matrice est renumérotée par une dissection emboîtée d_{max} est égal à 2. Dans les deux cas, le volume mémoire nécessaire pour réaliser la factorisation est $O(\eta(L))$[1].

1. En utilisant la notion de graphe quotient [55] le volume mémoire nécessaire à la factorisation symbolique peut être ramené à $O(\eta(A))$.

Algorithme 8 : Algorithme de factorisation symbolique de Cholesky.

Entrée : Matrice A symétrique définie positive

Sortie : Structure du facteur L obtenu après factorisation de la matrice A : $Struct(L_{*,j})$ contenant la structure creuse de la colonne j de L.

Sortie : Arbre d'élimination R associé à la factorisation : R_j contenant la liste des fils du nœud j.

```
1:  pour j = 1 jusqu'à n faire
2:     R_j = ∅
3:  pour j = 1 jusqu'à n faire
4:     S = Struct(A_{*,j})
5:     pour j' ∈ R_j faire
6:        S = S ∪ Struct(L_{*,j'}) − {j}
7:     Struct(L_{*,j}) = S
8:     si Struct(L_{*,j}) ≠ ∅ alors
9:        i = min Struct(L_{*,j})
10:       R_i = R_i ∪ {j}
```

Une amélioration de cet algorithme consiste à introduire le calcul des super-nœuds en cours de factorisation. Si R_j ne contient qu'un élément j', les colonnes j' et j sont contiguës. Si $Struct(A_{*,j}) \subseteq Struct(L_{*,j'})$ alors $Struct(A_{*,j}) = Struct(A_{*,j'}) - \{j\}$. Les colonnes i et j appartiennent donc à un même super-nœud. Ceci permet de ne stocker que la structure des super-nœuds pendant la factorisation. Une autre amélioration de cet algorithme peut également être obtenue, par introduction de la notion de graphe quotient [55].

Les super-nœuds de petites tailles posent un problème lors de la factorisation numérique car les blocs denses, sur lesquels sont effectués les opérations, sont alors trop petits pour bien exploiter l'unité de calcul du processeur. Il est cependant possible, en rajoutant des éléments nuls dans la structure de la matrice creuse d'amalgamer des super-nœuds pour augmenter leurs tailles, au prix cependant d'un accroissement du nombre d'opérations effectuées lors de la factorisation numérique. La figure 8.4 montre un exemple d'amalgame. La difficulté est alors de trouver le bon compromis entre le nombre d'éléments nuls ajoutés dans la structure et la taille des super-nœuds obtenus. Ashcraft et Grimes [6] ont proposé un algorithme pour amalgamer les super-nœuds dépendant uniquement du nombre maximum d'éléments nuls ajoutés dans la structure d'un super-nœud après le traitement.

La complexité des algorithmes de factorisation symbolique et de regroupement de super-nœud est faible devant les autres étapes de la factorisation de Cholesky. À titre d'exemple, les temps obtenus sur une machine SUN Ultra Sparc II pour la factorisation séquentielle d'une matrice contenant 1029655 éléments non nul (matrice BCSSTK32 de la base Harwell-Boeing) sont de $0.18s$ pour la factorisation symbolique alors que le temps

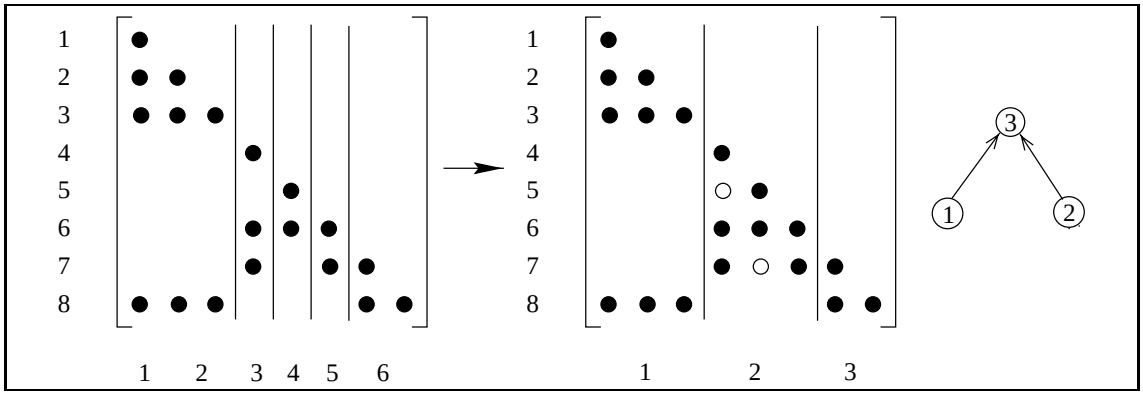

Figure 8.3 – *Les super-nœuds obtenus avant et après amalgame, sur la structure du facteur de Cholesky de la figure 8.1 page 132 et l'arbre d'élimination des super-nœuds associés après amalgame. Les cercles blancs représentent les éléments rajoutés sur la structure du facteur après amalgame.*

total (renumérotation, factorisation symbolique, factorisation numérique) de factorisation est de $20.18s$.

8.5 Factorisation numérique

Cette étape est la plus coûteuse des trois. C'est également celle qui a le plus de parallélisme potentiel.

8.5.1 Factorisation numérique séquentielle

Plusieurs organisations des calculs existent en séquentiel. Dans la suite nous nous intéresserons uniquement aux algorithmes de type *right-looking* pour lesquels la mise à jour de la sous matrice située à droite de la colonne pivot est faite immédiatement après le traitement de cette colonne pivot. L'algorithme 7 est de type *right-looking*.

Une première variante de cet algorithme, consiste à exprimer les calculs en terme d'opérations sur des blocs de matrice, en utilisant la structure des super nœuds. La matrice à factoriser A est partitionnée en N sous matrices, chaque sous matrice correspondant à un super-nœud de la matrice. L'algorithme 9 présente cette version. Comme les colonnes d'une sous matrice ont la même structure creuse, la factorisation LL^t d'une sous matrice est une factorisation séquentielle. De la même manière la mise à jour de la sous matrice L_j par la sous matrice L_k est réalisée par un produit de deux sous matrices denses de la matrice L_k suivie d'une addition du résultat dans la matrice L_j. Cette addition est la seule opération creuse intervenant dans l'algorithme.

Algorithme 9 : Algorithme de factorisation creuse de Cholesky par bloc de colonnes.

Entrée : matrice A symétrique définie positive partitionnée en N blocs de colonnes notés $L_j, 1 \leq j \leq q$

Sortie : $LL^T = A$

1: **pour** $k = 1$ **jusqu'à** N **faire**
2: Factorisation de Cholesky en place du bloc L_k
3: **pour** $j \in Struct(L_{*,k})$ **faire**
4: Mise à jour du bloc L_j par le bloc L_k

Une variante de cet algorithme, appelée méthode multifrontale [85], diffère dans la manière dont est réalisée l'opération de mise à jour de la sous matrice L_k par la sous matrice L_j. Dans cette méthode l'opération de mise à jour de la dernière ligne de l'algorithme 9 n'est pas accumulée directement sur le bloc L_j mais sur un front d'élimination qui est ensuite utilisé pour les mises à jour suivantes. Cette méthode est utile lorsque la matrice à factoriser ne tient pas entièrement en mémoire. La matrice à factoriser peut être laissée sur le disque et seul le front d'élimination est gardé en permanence en mémoire.

Dans la suite, nous nous focaliserons sur la parallélisation de l'algorithme 9 sans méthode frontale. On peut se référer à l'article [60] pour une étude de la parallélisation de la méthode frontale.

8.5.2 Partitionnement par bloc bidimensionnel pour la factorisation numérique parallèle

Deux niveaux de parallélisme sont explorés lors de la factorisation numérique de Cholesky. Le premier niveau consiste à traiter en parallèle les super-nœuds appartenant à deux sous arbres distincts de l'arbre d'élimination. Plusieurs itérations sur la boucle d'indice k de l'algorithme 9 sont alors réalisées en parallèle. Mais au fur et à mesure que l'on avance dans la factorisation, le nombre de super-nœuds pouvant être traités en parallèle diminue et de plus la taille des super-nœuds croît. Il est alors nécessaire d'exploiter un autre niveau de parallélisme. La factorisation de Cholesky dense de la ligne 2 ainsi que la mise à jour du bloc L_j par le bloc L_k de la ligne 4 qui est principalement un produit de matrices denses peuvent être parallélisées. Pour cela, on applique la même technique que celle utilisée dans la section 7.1, qui consiste à partitionner dans les deux dimensions les matrices pour réduire le volume de communication.

L'arbre d'élimination peut alors est divisé en deux parties, une partie contenant les nœuds qui seront traités sur un seul processeur et l'autre les nœuds dont le traitement sera parallélisé. La figure 8.4 montre cette séparation de l'arbre par une ligne en pointillés. Le problème qui consiste à fixer cette séparation sera étudié dans la section 8.5.5.

Pour pouvoir exprimer simplement l'algorithme, la partition appliquée sur les lignes de la matrice est la même partition que celle appliquée sur les colonnes. Cette partition est

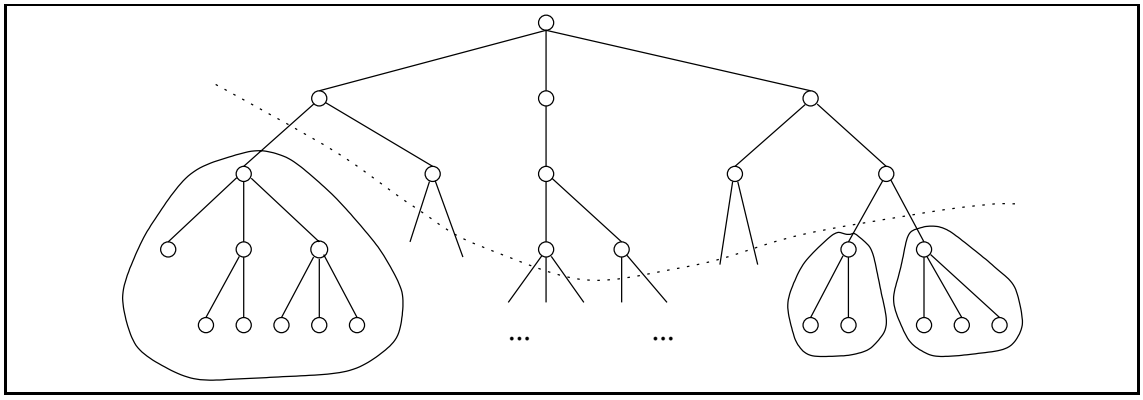

Figure 8.4 – *Sous arbres locaux de l'arbre d'élimination.*

obtenue à partir de la partition en super-nœuds des colonnes où les super nœuds dont la taille est trop grande sont redécoupés. Un exemple de partition bidimensionnelle sur une matrice creuse est donné dans la figue 8.5 page 140. Comme cet exemple est petit, il n'est pas nécessaire de redécouper les super-nœuds de grande taille.

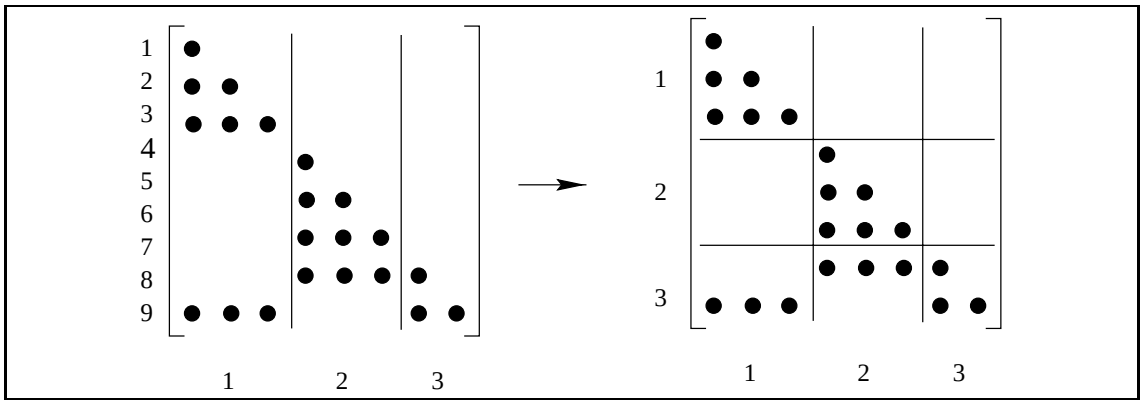

Figure 8.5 – *Partitionnement bidimensionnel de la matrice*

L'algorithme 10 page 141 de factorisation avec partitionnement par bloc bidimensionnel peut alors être écrit. La factorisation de Cholesky de la ligne 2 peut être réalisée par un appel à la routine DPOTRF de la librairie LAPACK ; l'opération de la ligne 5 peut être réalisée par un appel à la routine DTRSM de la librairie BLAS ; le produit de matrices de la ligne 8 est réalisé par un appel à la routine DGEMM de la librairie BLAS : enfin la seule opération creuse est l'accumulation de ce produit sur la matrice $L_{i,j}$. Pour plus de détails sur cet algorithme, on peut se référer à l'article [112].

Dans la suite nous noterons par $bdiv(k,i)$ et $bmod(k,i,j)$ respectivement l'opération

Algorithme 10 : Algorithme de factorisation creuse de Cholesky avec partitionnement bidimensionnel de la matrice.

Entrée : matrice A symétrique définie positive partitionnée dans les deux dimensions en $N \times N$ blocs notés $L_{i,j}$ (certains blocs peuvent être nuls).

Sortie : $LL^T = A$

1: **pour** $k = 1$ **jusqu'à** N **faire**
2: Factorisation en place du bloc $L_{k,k}$
3: **pour** $i \in Struct(L_{*,k})$ **faire**
4: {opération $bdiv(k,i)$}
5: $L_{i,k} = L_{i,k} L_{k,k}^{-1}$
6: **pour** $j \in Struct(L_{*,k})$ **faire**
7: **pour** $i \in Struct(L_{*,k})$ et $i \leq j$ **faire**
8: {opération $bmod(k,i,j)$}
9: $L_{i,j} = L_{i,j} - L_{i,k} L_{j,k}^t$

de la ligne 5 et 8 de l'algorithme 10 et nous désignerons par le terme « panneau » une colonne de blocs de la matrice partitionnée.

8.5.3 Méthode fan-out et fan-in

Dans cette section, nous allons nous intéresser aux diverses organisations des calculs de l'algorithme 10 sur une machine à mémoire distribuée en utilisant un modèle de programmation par échange de messages.

On considère ici que l'ordonnancement des calculs est dirigé par le placement initial des blocs de la matrice sur la mémoire distribuée. L'opération de factorisation du bloc de la ligne 2 de l'algorithme 10 est alors exécutée sur le processeur contenant ce bloc et l'opération $bdiv(k,i)$ est exécutée sur le processeur contenant le bloc i. Par contre pour l'opération $bmod(k,i,j)$, il existe plusieurs possibilités.

Dans la méthode appelée fan-out, l'opération $bmod(k,i,j)$ est réalisée sur le processeur contenant le bloc $L_{i,j}$. Pour cela les blocs $L_{i,k}$ et $L_{j,k}$ doivent être communiqués au processeur réalisant l'opération $bmod(k,i,j)$.

Dans la méthode appelée fan-in, l'opération $bmod(k,i,j)$ est réalisée sur l'un des processeurs contenant le bloc $L_{i,k}$ ou le bloc $L_{j,k}$, et le résultat est accumulé localement sur ce processeur. Ce résultat ne sera envoyé sur le processeur contenant le bloc $L_{i,j}$ qu'après que toutes les contributions de ce processeur sur le bloc $L_{i,j}$ auront été calculées. Cette méthode permet ainsi une réduction du nombre de communications et du volume de données transmises.

La méthode fan-in a initialement été proposée dans le cadre de la factorisation creuse de Cholesky par colonnes, comme remplacement de la méthode fan-out [5] [68] ; cette méthode permettant en effet une réduction du volume de communication. Différentes

adaptations de cette méthode ont été réalisées pour la factorisation de Cholesky creuse par bloc de colonnes ou partitionnement 1D [39] [38].

Pour la factorisation creuse de Cholesky avec partitionnement bidimensionnel de la matrice, le seul code existant est celui proposé par Rothberg et Gupta [112]. Cet algorithme utilise la méthode fan-in uniquement dans la première phase de la factorisation, là où seul le premier niveau de parallélisme de la factorisation creuse de Cholesky est exploité, c'est-à-dire pour les nœuds de l'arbre situés en dessous de la ligne en pointillé de la figure 8.4 page 140.

Nous proposerons dans la suite, un algorithme utilisant la méthode fan-in sur l'ensemble des nœuds du graphe et un placement particulier des blocs de la matrice, appelé placement proportionnel, améliorant substantiellement l'algorithme proposé par Rothberg.

8.5.4 Algorithme fan-in

Nous présentons dans cette section le détail de l'algorithme fan-in, mis à part le placement des blocs de la matrice sur les processeurs, qui sera traité dans la section suivante. L'algorithme prend donc en entrée ce placement par l'intermédiaire de la variable $owner(i,j)$ qui retourne le processeur sur lequel le bloc $L_{i,j}$ est placé. L'exécution est alors entièrement dirigée par ce placement des données. La structure principale du programme est présentée dans l'algorithme 11 page 144 et deux importantes fonctions sont présentées dans les algorithmes 12 page 145 et 13 page 145. Cet algorithme est de type SIMD, c'est-à-dire que chaque processeur exécute les instructions de l'algorithme 11 page 144.

Les structures de données de cet algorithme sont les suivantes : $wait(K)$ est un ensemble contenant l'indice des lignes des blocs locaux de la colonne K prêts pour l'opération $bdiv$; $finished(K)$ est un ensemble contenant l'indice des lignes des blocs locaux ou reçus de la colonne K du facteur de Cholesky (*i.e.* les blocs sur lesquels toutes les opérations ont été effectuées); $diag(K)$ est un drapeau indiquant si le bloc $L_{K,K}$ est factorisé et présent en mémoire locale; $queue$ est une liste contenant les blocs locaux sur lesquels une opération peut être réalisée immédiatement (*i.e.* sans attendre une donnée du réseau).

Le schéma de calcul et de communication de cet algorithme est le suivant. Lorsque un bloc $L_{I,K}$ du facteur de Cholesky est calculé, il est diffusé à l'ensemble des processeurs contenant des blocs de la colonne K, *i.e.* l'ensemble $group(K)$ (lignes 4 et 12 de l'algorithme 13). Par ailleurs, chaque mise à jour d'un bloc $L_{I,J}$ par le produit des blocs $L_{I,K}$ et $L_{J,K}$ est calculée et accumulée localement sur le processeur contenant le bloc $L_{I,K}$ (ligne 1 de l'algorithme 12). L'ensemble des contributions sur le bloc $L_{I,J}$ calculées par un processeur, est alors envoyé en une seule fois sur le processeur contenant le bloc $L_{I,J}$. Pour cela, le nombre total de contributions réalisées par un processeur sur le bloc $L_{I,J}$ est pré-calculé (lors de la phase de factorisation symbolique) et enregistré dans $nmod(I,J)$. Cette variable est ensuite décrémentée lors de chaque calcul local d'une nouvelle contri-

bution. La contribution locale sera alors envoyée lorsque ce compteur deviendra nul (voir l'algorithme 12 $bmod\_send$ page 145). De la même façon, le nombre de contributions que doit recevoir un bloc est pré-calculé et enregistré dans $nmod(I, J)$. Après chaque réception d'une contribution sur le bloc $L_{I,J}$ (ou bien lors d'un calcul local d'une contribution), $nmod(I, J)$ est décrémenté et lorsqu'il devient nul le bloc $B_{I,J}$ devient prêt pour une opération $bdiv$ (ou une opération de factorisation si $I = J$).

Un point reste à clarifier dans cet algorithme. Les appels à la fonction $bmod\_send$ à la ligne 31 de l'algorithme 11 page 144 ainsi qu'aux lignes 9 et 15 de l'algorithme 13 page 145 ne sont pas tout à fait exactes. Lorsque l'ensemble $finished(K)$ est parcouru, il y a trois situations où le processeur courant appelle la fonction $bmod\_send(K, I, J)$: lorsque $L_{I,K}$ et $L_{J,K}$ sont tous les deux locaux à un même processeur ; lorsque $L_{I,K}$ est local et que $L_{J,K}$ est reçu par le réseau ; lorsque $L_{J,K}$ est local et que $L_{I,K}$ est reçu par le réseau.

Un autre point important de cet algorithme concerne la manière dont les communications sont émises (ligne 10 de l'algorithme 12 page 145 ainsi que lignes 4 et 7 de l'algorithme 13 page 145) et reçus (ligne 17 de l'algorithme 11 page 144). Les émissions peuvent être réalisées par une communication asynchrone puisque un bloc qui est communiqué est un bloc qui ne sera plus modifié localement. Pour les réceptions, le problème est un peu plus difficile puisque la taille des blocs à recevoir n'est pas connue à l'avance. Une solution est alors d'envoyer, lors de l'émission, un premier message contenant la taille du bloc qui va être communiqué. Le bloc lui même ne sera envoyé qu'après. À la réception, la taille du message est d'abord reçue puis l'espace nécessaire pour recevoir le bloc est alloué et enfin le bloc lui même est reçu.[2]

8.5.5 Ordonnancement des calculs

Nous étudions dans cette section les méthodes utilisées pour calculer un placement des blocs de la matrice sur les processeurs. C'est ce placement, réalisé avant la factorisation numérique, qui dirigera ensuite l'ordonnancement des calculs comme défini dans la section 8.5.3. L'objectif est donc de calculer un placement des données conduisant à un ordonnancement des calculs minimisant le temps d'exécution. Pour cela un bon compromis entre équilibrage de la charge de calcul et minimisation des communications est recherché.

Plusieurs heuristiques différentes existent pour réaliser ce placement [51, 102, 113, 60, 39]. Nous présentons dans la suite deux méthodes pour calculer ce placement des blocs, méthodes proposées et plus largement détaillées dans l'article [34].

2. Notons qu'en MPI, la fonction MPI_PROBE permet à la réception, de récupérer la taille d'un message arrivant sur le processeur avant d'avoir réservé l'espace mémoire. Il n'est donc pas nécessaire d'envoyer explicitement les tailles des blocs communiqués.

Algorithme 11 : Factorisation fan-in avec partitionnement bidimensionnel.

Entrée : id, le numéro du processeur
Entrée : N le nombre de colonnes (*i.e.* de panneaux) de la matrice par blocs L
Entrée : $L_{i,j}$ les blocs de la matrice (*i.e.* les blocs non nuls et locaux au nœud id)
Entrée : $owner(i,j)$, le processeur sur lequel le bloc $L_{i,j}$ est placé
Entrée : $group(j)$, l'ensemble des processeurs contenant des blocs de la colonne j
Entrée : $nmod(i,j)$, le nombre de mises à jour sur le bloc $L_{i,j}$

1: {Étape locale}
2: **pour** $K = 1$ **jusqu'à** N **faire**
3: **si** L_K est un panneau local (*i.e.* $\#group(k) = 1$) **alors**
4: factoriser le panneau L_K
5: **pour tout** $I, J \in Struct(L_{*,K})$ tel que $I \geq J$ **faire**
6: $bmod\_send(K, I, J)$
7: **sinon si** $id \in group(K)$ **alors**
8: réaliser toutes les opérations possibles sur les blocs locaux du panneau K
9: initialiser les ensembles $wait(K)$ et $finished(K)$
10: {Étape distribuée}
11: **boucler**
12: **tant que** l'ensemble $queue$ est non vide **faire**
13: retirer un bloc $L_{I,J}$ de l'ensemble $queue$
14: $last\_block\_op(I, J)$
15: **si** il n'y a plus de bloc à recevoir du réseau **alors**
16: fin de l'algorithme
17: ATTENDRE un bloc $L_{I,J}$ du réseau
18: **si** $L_{I,J}$ est un bloc de mise à jours **alors**
19: accumulation du bloc reçu sur le bloc local $L_{I,J}$
20: soustraire à $nmod(I, J)$ le nombre de contributions apportées par le bloc
21: **si** $nmod(I, J) = 0$ **alors**
22: $last\_block\_op(I, J)$
23: **sinon** {$L_{I,J}$ est un bloc du facteur de Cholesky}
24: **si** $I = J$ **alors**
25: $diag(J) = 1$
26: **pour tout** $I' \in wait(J)$ **faire**
27: $bdiv(I', J)$
28: **sinon**
29: insérer I dans l'ensemble $finished(J)$
30: **pour tout** $I' \in finished(J)$ **faire**
31: $bmod\_send(J, I, I')$

Algorithme 12 : Fonction $bmod\_send$.

Entrée : Les indices K, I et J

1: $bmod(K, I, J)$
2: $nmod(I, J) = nmod(I, J) - 1$
3: **si** $nmod(I, J) = 0$ **alors**
4: **si** $id = owner(I, J)$ **alors**
5: **si** $I = J$ ou $diag(J) = 1$ **alors**
6: insérer $L_{I,J}$ dans l'ensemble $queue$
7: **sinon**
8: insérer I dans l'ensemble $wait(J)$
9: **sinon**
10: ENVOYER $L_{I,J}$ sur le processeur $owner(I, J)$

Algorithme 13 : Fonction $last\_block\_op$.

Entrée : Les indices I et J

1: **si** $I = K$ **alors**
2: calculer en place le facteur de Cholesky du bloc $L_{K,K}$
3: $diag(K) = 1$
4: DIFFUSER $L_{K,K}$ à tous les processeurs dans $group(K)$
5: **pour tout** $I' \in wait(K)$ **faire**
6: $bdiv(I', K)$
7: insérer I' dans l'ensemble $finished(K)$
8: **pour tout** $J \in finished(K)$ **faire**
9: $bmod\_send(K, I', J)$
10: **sinon si** $diag(K) = 1$ **alors**
11: $bdiv(I, K)$
12: DIFFUSER $L_{I,K}$ à tous les processeurs dans $group(K)$
13: insérer I dans l'ensemble $finished(K)$
14: **pour tout** $J \in finished(K)$ **faire**
15: $bmod\_send(K, I, J)$
16: **sinon**
17: insérer I dans l'ensemble $wait(J)$

8.5.5.1 Placement sur une grille

L'algorithme proposé par Geist et Ng [51] est utilisé pour le placement des sous arbres locaux. Cet algorithme prend en entrée une liste de sous-arbres, initialisée avec la racine de l'arbre d'élimination, et calcule un placement de ces sous arbres sur les processeurs par un algorithme de *bin-packing* (*i.e.* chacun des sous arbres, pris dans l'ordre décroisant de leurs poids, est placé sur le processeur le moins chargé). Si le déséquilibre de charge est trop important, le sous arbre de plus grand poids est retiré de l'arbre et remplacé par ses fils.

Le placement des blocs des panneaux distribués est ensuite réalisé en suivant l'idée de Rothberg et Gupta [112] utilisée pour la méthode fan-out. Les p processeurs sont disposés sur une grille $p_l \times p_c$ et chaque bloc d'une colonne (respectivement ligne) de la matrice par bloc est placé sur une colonne (respectivement ligne) de la grille de processeurs. L'équilibrage de la charge entre les lignes (respectivement les colonnes) de la grille de processeurs est réalisé en adaptant l'algorithme proposé dans [113] à la méthode fan-in. À chaque colonne (respectivement ligne) de la matrice par bloc est associée un coût correspondant à la somme des coûts correspondants aux blocs de la colonne (respectivement ligne). Chacune des colonnes (respectivement des lignes) de la matrice, pris dans l'ordre décroisant de leur coût est alors placée sur la colonne (respectivement la ligne) la moins chargée de la grille de processeur.

Ce placement des blocs sur une grille permet alors de réduire les communications, de manière analogue aux algorithmes parallèles d'algèbre linéaire dense vus dans le chapitre II. En effet, les diffusions réalisées aux lignes 4 et 12 de l'algorithme 13 page 145 ne font intervenir qu'au maximum $p_c - 1$ processeurs : l'ensemble $group(K)$ contient les processeurs ayant un bloc de la colonne K. De la même manière, le processeur possédant le bloc $L_{I,J}$ ne recevra de contribution sur ce bloc que des processeurs situés sur la même ligne de la grille puisque l'opération $bmod(K, I, J)$ est réalisée sur le processeur contenant le bloc $L_{I,K}$.

8.5.5.2 Placement proportionnel

Lorsque l'on analyse, par rapport à l'arbre d'élimination, les communications réalisées lors de la factorisation par la méthode fan-in, on remarque que les blocs de deux nœuds appartenant à deux sous arbres différents de l'arbre d'élimination, n'interviennent pas dans une même communication. Donc les calculs réalisés sur deux sous arbres distincts, au sens où l'un n'est pas un sous arbre de l'autre, ne feront jamais intervenir de communication entre ces deux sous arbres.

L'idée du placement proportionnel est alors de partitionner les processeurs de manière à ce que les calculs de deux sous arbres distincts soient toujours réalisés sur deux partitions différentes des processeurs, dans le but de réduire les communications. Pour cela la méthode suivante est utilisée. À chaque nœud de l'arbre est associé un groupe de pro-

cesseurs. Le nœud racine de l'arbre reçoit l'ensemble des processeurs et chaque nœud de l'arbre partitionne ce groupe de processeurs pour le répartir sur ses nœuds fils. Cette partition est appliquée récursivement jusqu'à obtenir des partitions ne contenant qu'un processeur. La partie gauche de la figure 8.6 montre un exemple d'une telle partition des nœuds sur le graphe.

La manière dont un nœud répartit son groupe de processeurs sur ses sous arbres est primordiale pour la régulation de la charge de calcul entre les processeurs. Chaque sous arbre implique des calculs indépendants. L'idée du placement proportionnel est alors de répartir les processeurs sur les sous arbres proportionnellement à la charge de calcul de chaque sous arbre. Cette répartition proportionnelle n'est cependant pas triviale car l'ensemble des ressources à partitionner, c'est-à-dire le nombre de processeurs, doit rester de cardinal entier.

Une fois que l'assignement des processeurs sur les nœuds du graphe est réalisé, il faut encore placer chacun des blocs d'un nœud sur un des processeurs assignés au nœud. Cette seconde étape est représentée sur la partie droite de la figure 8.6. Là encore plusieurs méthodes sont utilisables.

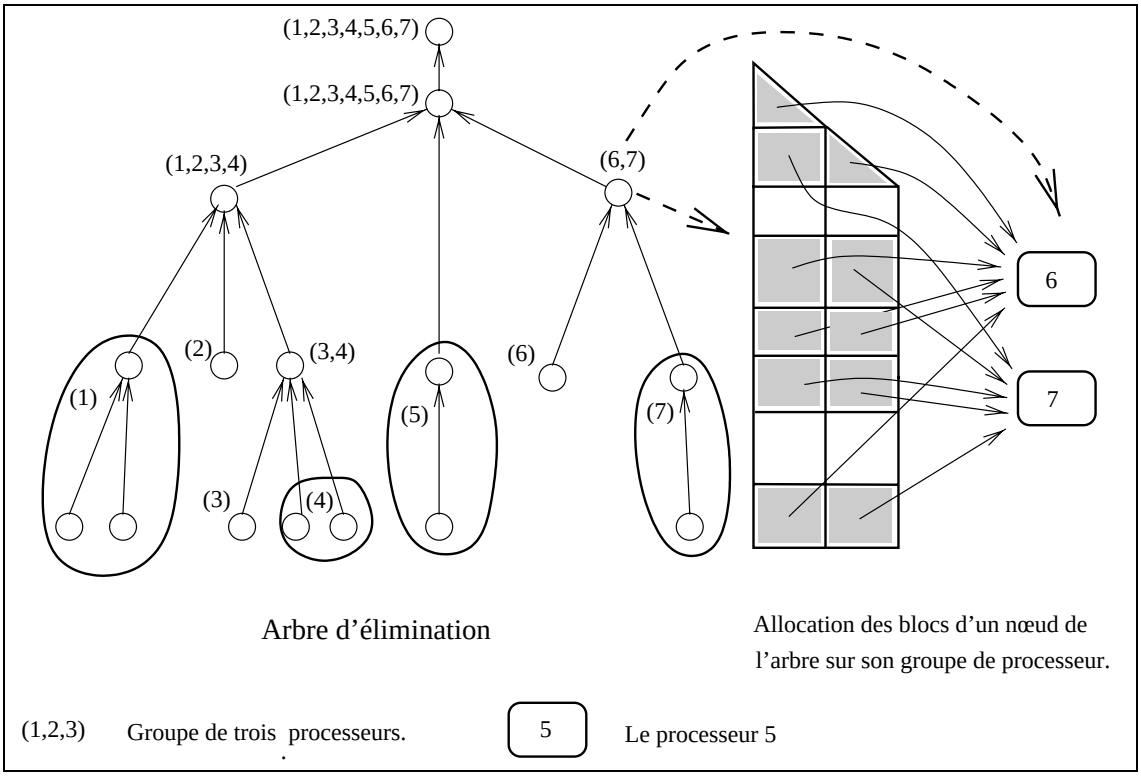

Figure 8.6 – *Placement proportionnel des blocs de la matrice sur 7 processeurs.*

Une de ces méthodes consiste à utiliser un placement cyclique bidimensionnel similaire à celui utilisé pour la factorisation de Cholesky dense vue dans la section 7.1

page 113. Cependant, ici, la taille des blocs, et donc le coût de chaque opération sur un bloc, est variable d'un bloc à l'autre. Le placement cyclique bidimensionnel doit donc être adapté à ce problème. La solution utilisée est de répartir équitablement les lignes et les colonnes de la sous matrice de blocs sur les lignes et les colonnes de la grille de processeurs.

L'autre solution est d'utiliser un algorithme glouton. Les blocs sont alors alloués sur chacun des processeurs, dans l'ordre où ils seront utilisés, de manière à répartir la charge de calcul sur chacun des processeurs. Par rapport à la première solution, cette méthode peut générer plus de communication ; par contre elle fournit un meilleur équilibrage de la charge de calcul sur les processeurs.

8.5.6 Expérimentations

Nous montrons dans cette section des expérimentations réalisées sur les différents algorithmes de factorisation numérique de Cholesky précédemment décrits.

Les expérimentations ont été réalisées sur deux types de machines parallèles. La première est un IMB SP1 à 32 processeurs précédemment décrit dans la section 7.5 page 120 du chapitre précédent. Nous rappelons ici les caractéristiques de cette machine ; puissance de référence d'un processeur de 100Mflops[3], bande passante de 30 méga octets par seconde et latence de 60 μs. La seconde machine est un Cray T3D offrant une puissance de référence par processeur de 110 Mflops [3] et une bande passante sur MPI de 35 méga octets par seconde.

Les programmes ont été testés sur plusieurs matrices creuses de la collection Harwell-Boeing [37]. La figure 8.1 montre les caractéristiques des matrices utilisées. Ces matrices ont été extraites des ensembles BCCSSTRUC2 et BCCSSTRUC3 de la collection, B15 désignant la matrice BCSSTK15, B16 la matrice BCSSTK16, etc. La dernière, G150 a été obtenue par une discrétisation à 5 points sur une grille à deux dimensions. Ces matrices ont été renumérotées avec METIS, mis à part G150 qui a été renumérotée par une dissection emboîté optimale.

Trois versions de la factorisation numérique sont évaluées ici. L'algorithme FO_GRID est l'algorithme fan-out avec placement des blocs sur une grille proposé par Rothberg. L'algorithme FI_GRID est l'algorithme fan-in présenté dans la section 8.5.4 page 142 avec un même placement des blocs sur une grille. L'algorithme FI_PROP_G est l'algorithme fan-in avec le placement proportionnel présenté dans la section 8.5.5.2 page 146. Ces différentes versions de la factorisation numérique ont été écrites en C. Les opérations sur les blocs sont réalisées par des appels aux BLAS de niveau 3 ainsi qu'à la bibliothèque LAPACK, mis à part pour l'opération d'addition de la fonction $bmod$ codée en C. Les communications sont réalisées avec la bibliothèque MPI.

3. Mesurée sur un produit de matrice par la fonction dgemm de la bibliothèque BLAS.

Nom	Taille	Nb. éléments de A	Nb. éléments L	Mflop pour L
B15_K	3,948	117,816	574,104	122.48
B16_K	4,884	290,378	754,734	150.53
B17_K	10,974	428,650	1,188,305	207.13
B18_K	11,948	149,090	673,070	111.58
B25_K	15,439	252,241	1,842,860	491.96
G150	22,500	111,900	721,862	62.51

Tableau 8.1 – *Différentes matrices de test. La troisième et quatrième colonne désignent le nombre d'éléments non nuls de la matrice initiale et du facteur obtenu après factorisation. La dernière colonne indique le nombre d'opération flottantes nécessaire pour factoriser la matrice.*

	p	B15_K	B16_K	B17_K	B18_K	B25_K	G150	B17_A	B25_A
SP1	16	0.7%	4.9%	0.4%	0.9%	0%	10.8%	0.7%	0%
	32	10.2%	9.9%	4.3%	15.5%	6.5%	18.0%	7.5%	7.1%
T3D	16	0.6%	3.6%	-2.6%	6.7%	2.0%	7.9%	0%	1.1%
	32	-0.2%	5.2%	1.8%	4.3%	3.5%	7.2%	-4.2%	-1.9%

Tableau 8.2 – *Amélioration des performances de l'algorithme FI_GRID par rapport à l'algorithme FO_GRID. Les rapports sont calculés par $\frac{FO-FI}{FO}$ où FO, respectivement FI, est le temps obtenu avec l'algorithme FO_GRID, respectivement FI_GRID.*

8.5.7 Comparaison expérimentale

Tout d'abord nous comparons, dans le tableau 8.2 les deux algorithmes fan-out et fan-in avec un même placement des blocs sur une grille de processeurs. Sur l'IBM SP1 avec 16 processeurs, l'algorithme FI_GRID a un faible avantage sur l'algorithme FO_GRID. Cet avantage se confirme sur 32 processeurs. Sur le Cray T3D, les performances des deux méthodes sont comparables. La figure 8.7 donne les résultats obtenus sur l'algorithme FI_GRID. Sur 32 processeurs du SP1, la factorisation de la matrice B25_K exploite plus de 11% de la puissance de la machine. Sur 32 processeurs du Cray T3D, la factorisation de la même matrice exploite plus de 13% de la machine.

La figure 8.8 page 151 montre les résultats obtenus sur l'algorithme fan-in avec placement proportionnel (FI_PROP_G). Pour un faible nombre de processeurs ($p \leq 8$), l'avantage du placement proportionnel sur le placement sur une grille n'est pas net. Ceci est dû au problème des parties entières intervenant dans l'algorithme de placement proportionnel. Cependant, pour $p = 16$ et $p = 32$ l'amélioration apportée par le placement proportionnel est significative. Sur 32 processeurs du SP1, la factorisation de la matrice B25_K exploite plus de 14% de la puissance de la machine. Sur 32 processeurs du Cray T3D, la factorisation de la même matrice exploite plus de 14% de la machine. Comparé

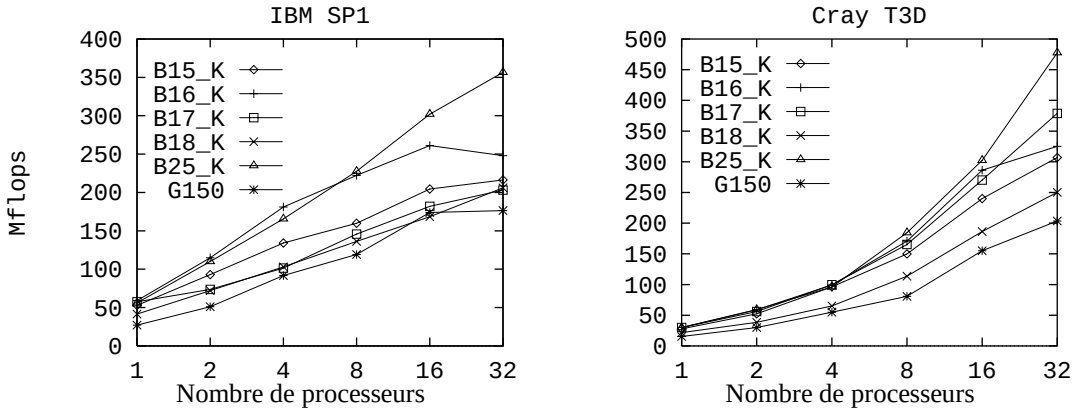

Figure 8.7 – *Performance (Mflops) pour l'algorithme fan-in avec un placement sur une grille (i.e. l'algorithme FI_GRID).*

		B15_K	B16_K	B17_K	B18_K	B25_K	G150
on SP1	$p = 16$	1.9%	21.8%	4.2%	14.4%	14.0%	16.3%
	$p = 32$	18.6%	43.1%	18.0%	27.5%	27.0%	40.2%
on T3D	$p = 16$	-6.4%	3.6%	-5.6%	8.1%	6.8%	6.3%
	$p = 32$	13.7%	32.7%	8.4%	28.4%	9.1%	13.8%

Tableau 8.3 – *Amélioration des performances de l'algorithme FI_PROP_G par rapport à l'algorithme FO_GRID.*

à l'algorithme FO_GRID, le gain apporté par l'algorithme FI_PROP_G va jusqu'à 40%. L'algorithme de placement proportionnel améliore donc d'une façon significative les performances obtenues par rapport à l'algorithme de placement sur une grille.

8.5.8 Influence de la taille des panneaux

Nous présentons ici l'influence de la taille maximum choisie pour la partition de la matrice (*i.e.* la taille des panneaux). Le choix de cette taille est en effet important et influence directement les performances des algorithmes. La taille des panneaux est directement reliée au degré de parallélisme généré. Plus cette taille est faible plus le degré de parallélisme est important, particulièrement en fin de factorisation. Cependant si cette taille est trop faible, la performance se dégrade du fait de l'augmentation du nombre de communications et de la mauvaise exploitation des processeurs lors des opérations BLAS 3 sur les blocs. Un compromis doit donc être cherché. La figure 8.9 page 151 montre les performances obtenues sur le SP1 et la matrice B25_K pour différentes tailles de pan-

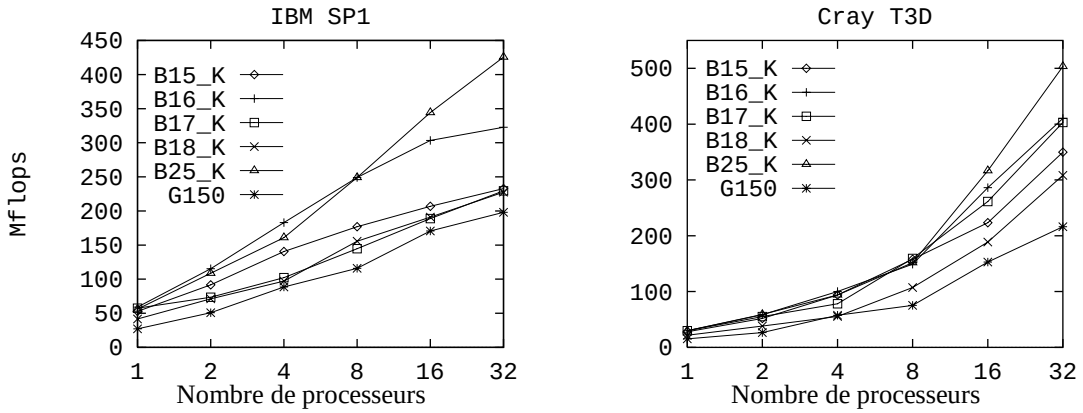

Figure 8.8 – *Performance (Mflops) pour l'algorithme fan-in avec un placement proportionnel glouton (i.e. l'algorithme FI_PROP_G).*

neaux. Pour cette matrice, une taille optimale d'environ 48 apparaît : elle est indépendante de la stratégie d'ordonnancement utilisée.

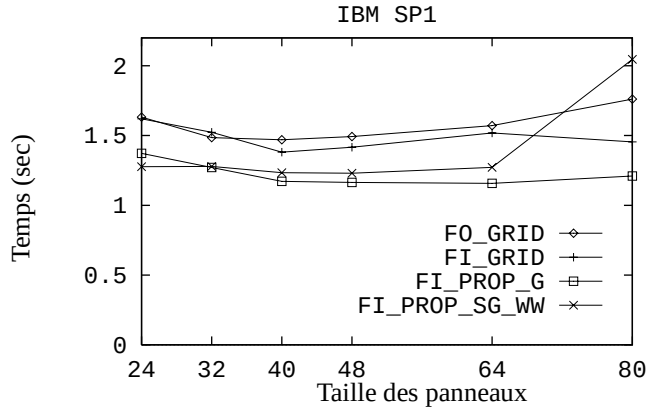

Figure 8.9 – *Temps d'exécution en secondes pour la factorisation de la matrice B25_K sur* 32 *processeurs du SP1 et pour différentes tailles de panneaux.*

8.6 Conclusion

Nous avons proposé dans ce chapitre un algorithme parallèle original pour la factorisation creuse de Cholesky, de type fan-in et qui utilise un partitionnement bidimensionnel de la matrice, gage d'une meilleure extensibilité de l'algorithme qu'un partitionnement mono-dimensionnel. Deux heuristiques de placement des données ont été proposées pour cet algorithme conduisant à deux versions FI_GRID (placement sur une grille) et

FI_PROP_G (placement proportionnel). Les expérimentations réalisées sur deux architectures distribuées montre premièrement que la version FI_GRID proposée est généralement meilleure que l'algorithme fan-out (FO_GRID) proposé par Rothberg (utilisant le même placement des blocs) et deuxièmement que la version FI_PROP_G proposée améliore encore les performances obtenues.

De plus, en complément de l'algorithme lui même, la stratégie d'ordonnancement choisie joue un rôle crucial pour les performances sur une machine donnée.

9

Factorisation numérique creuse de Cholesky en Athapascan-1

Ce chapitre est consacré à l'implantation de l'algorithme numérique de factorisation creuse de Cholesky, dans un modèle de programmation parallèle haut niveau (*i.e.* offrant une abstraction de la machine parallèle), et plus particulièrement en Athapascan-1. Nous supposerons dans ce chapitre que l'étape de renumérotation et de factorisation symbolique a été précédemment appliquée sur la matrice avec l'une des méthodes détaillées dans les sections 8.3 page 133 et 8.4 page 136 du chapitre précédent. Nous ne nous intéresserons donc qu'à l'étape de factorisation numérique présentée dans la section 8.5 page 138, qui est la plus coûteuse en temps de calcul.

Comme nous l'avons montré dans le chapitre précédent, la parallélisation de la factorisation numérique creuse de Cholesky dans le modèle de programmation par échange de messages permet d'obtenir de bonnes performances sur machine à mémoire distribuée. Cependant ces performances n'ont été obtenues qu'au prix d'un important travail sur le code par rapport au programme séquentiel équivalent. Il n'est qu'à voir l'algorithme présenté dans la section 8.5.4 page 142 pour se rendre compte de la difficulté de la parallélisation de l'application dans ce modèle. De plus, la factorisation de Cholesky creuse étant une application irrégulière, le programmeur doit mettre en œuvre des algorithmes d'ordonnancements souvent complexes (voir la section 8.5.5 page 143 pour les algorithmes d'ordonnancement spécifiques à la factorisation de Cholesky creuse et le chapitre 5 page 75 pour des algorithmes d'ordonnancement plus généraux). Un autre point concerne la portabilité de l'application écrite dans le modèle de programmation par échange de messages, ce modèle n'exploitant pas toujours au mieux les machines SMP ou les réseaux de machine SMP (voir par exemple les expérimentations sur la factorisation

de Cholesky dense de la section 7.5.2 et l'article [7]). Nous tenterons alors de montrer dans ce chapitre, l'apport d'un modèle de programmation parallèle haut niveau pour la parallélisation de ce type d'applications.

La structure du chapitre est la suivante. Tout d'abord, nous présentons quelques implantations dans des langages ou environnements de programmations parallèle de haut niveau de l'algorithme de factorisation numérique creuse de Cholesky. Ensuite nous présentons l'écriture de cet algorithme en Athapascan-1 et les performances obtenues.

9.1 Factorisation numérique creuse de Cholesky dans des langages de programmation parallèle haut niveau.

La factorisation de Cholesky creuse, de par son caractère irrégulier, est une application couramment présentée pour valider les langages ou interfaces de programmation parallèle de haut niveau. Nous présentons dans cette section les implantations de cette application sur trois des modèles de programmations haut niveaux présentés dans le chapitre 2 page 23.

9.1.1 Cilk

Une implantation en Cilk d'une factorisation creuse de Cholesky est proposée dans la thèse de Randall [104]. Cependant l'algorithme de factorisation numérique choisi est un algorithme récursif dans lequel la matrice creuse est représentée sous la forme d'un arbre à quatre fils (*quadtree*) [125].

Les expérimentations ont été réalisées sur une machine SMP à processeurs SUN UltraSPARC cadencée à 167 MHz. La matrice utilisée est la matrice BCSSTK32 renumérotée avec l'algorithme de degré minimum de MATLAB et nécessitant 1.1 Gflop pour calculer la factorisation numérique. Bien que les accélérations obtenues (accélérations de 6 sur 8 processeurs) pourraient être considérées comme satisfaisantes, le temps de factorisation numérique séquentiel pris comme référence pour le calcul de ces accélérations est de $1427s$. À titre de comparaison, le temps de factorisation numérique de cette matrice sur le même type de processeur cadencé à 295 MHz est de $8s$ sur notre code de factorisation numérique séquentiel. Ces très mauvaises performances sont certainement dû à une implantation trop simpliste de la factorisation creuse de Cholesky (pas d'utilisation des BLAS par exemple).

9.1.2 Jade

Une implantation en Jade de l'algorithme de factorisation numérique de Cholesky creux est également présentée dans [108, 107]. Contrairement à l'algorithme bidimensionnel du chapitre précédent, cet algorithme est de type monodimensionnel. Le parallélisme de l'algorithme est donc exprimé au niveau des opérations sur des groupes colonnes (*i.e.* sur des panneaux).

9.1.2.1 Écriture de l'algorithme

Jade est basé sur une sémantique séquentielle ; l'écriture de l'algorithme en Jade est donc très proche de l'écriture de l'algorithme séquentiel. La figure 9.1 page 156 montre l'écriture, en Jade, de l'algorithme de factorisation creuse de Cholesky par colonnes, proche de la version avec partitionnement monodimensionnel de la matrice.

Dans ce programme, la matrice à factoriser `L` est stockée sous le format compressé colonne. `L` est alors une structure composée de quatre champs : `num_column` contient le nombre de colonnes de la matrice, `column` est le tableau des éléments non nuls de la matrice (stockés par colonnes), `row_index` est le tableau des indices de ligne de ces éléments et `start_column` est le tableau des indices, dans les deux tableaux précédents, des premiers éléments de chaque colonne.

Les deux types de tâches sont créés aux lignes 5 et 15. À l'intérieur du bloc **withonly**, les données passées aux fonctions **rd_wr** indiquent les données qui seront accédées en lecture et écriture par la tâche. Les données passées aux fonctions **rd** indiquent les données qui seront accédées uniquement en lecture. Les tâches n'ont alors pas le droit d'accéder à d'autres données que celles ci.

9.1.2.2 Exécution de l'application

Une analyse des dépendances de données entre tâches lors de l'exécution de la fonction `factor` est réalisée pour déterminer les synchronisations nécessaires au respect de la sémantique séquentielle.

Une seule politique d'ordonnancement est implantée dans Jade qui par ailleurs n'offre aucun support pour l'écriture de nouvelles politiques. L'ordonnancement des tâches est toujours réalisé à la volée par une politique glouton de vol de tâches entre processeurs. Cependant une certaine prise en compte de la localité est considérée lors du vol, puisque seuls des groupes de tâches sont volés, un groupe de tâches correspondant à l'ensemble des tâches prenant comme premier paramètre le même objet partagé. Dans l'application de Cholesky creux, un groupe de tâches va alors correspondre à l'ensemble des tâches modifiant le même panneau (*i.e.* groupe de colonnes contiguës ayant la même structure), à savoir les tâches de mise à jour de ce panneau et la tâche de factorisation de ce panneau. Lorsqu'un processeur n'a plus de tâche prête à exécuter parmi les groupes de tâches qu'il

```
1:  factor( matrix* L ) {
2:     int i, j, first, last;
3:     for( j=0; j<L->num_column; j++) {
4:        /* tâche de factorisation de la colonne j */
5:        withonly { /* déclaration des données accédées par la tâche */
6:           rd_wr(L->column[j]);
7:           rd(L);
8:        } do(j) { /* paramètre et corps de la tâche */
9:           InternalUpdate(j);
10:       }
11:       first = L->start_column[j] + 1;
12:       last = L->start_column[j+1] - 1;
13:       for( i= first; i<=last; j++) {
14:          /* tâche de mise α jour de la colonne row_index[i] avec la colonne j */
15:          withonly { /* déclaration des données accédées par la tâche */
16:             rd_wr(L->column[L->row_index[i]]);
17:             rd(L->column[j]);
18:             rd(L);
19:          } do(i,j) { /* paramètres et corps de la tâche */
20:             XternalUpdate(j, L->row_index[i]);
21:          }
22:       }
23:    }
24: }
```

Figure 9.1 – *Écriture en Jade de l'algorithme numérique de factorisation de Cholesky creux par colonne.*

possède, il vole alors à l'un des autres processeurs un nouveau groupe de tâches contenant au moins une tâche prête.

9.1.2.3 Experimentations

Cette implantation a été validée expérimentalement dans [107] sur la matrice BCSSTK15 (présentée dans le tableau 8.1 page 149) de la collection Harwell-Boeing et sur deux machines parallèles d'architectures différentes.

La première est une machine DASH, utilisant une architecture NUMA (*Non uniform memories access*), comparable à l'architecture de l'Origine-2000 SGI, la machine étant composée de 32 processeurs répartis par groupes de quatre nœuds. La puissance de crête d'un processeur est de 10Mflops. Sur cette machine, la factorisation numérique séquentielle de la matrice prend $26.6s$ et la version parallèle écrite en Jade prend $7.8s, 5.9s$ et

$5.3s$ sur respectivement 8, 16 et 32 processeurs, ce qui donne une accélération de 5 sur 16 et 32 processeurs.

La seconde machine est un iPSC/860, utilisant une architecture distribuée. La puissance de crête d'un processeur est de 30Mflops, la latence du réseau est de $40\mu s$ et la bande passante de 2.8 Mbyte/s. Sur cette machine, la factorisation numérique séquentielle de la matrice prend $27.6s$ et le meilleur temps d'exécution de l'algorithme écrit en Jade est de $38.4s$. Aucune accélération n'est donc obtenue. La raison principale invoquée vient du volume trop important de communications générés lors de l'exécution et d'une granularité des tâches trop fine devant les coûts de création et de gestion de celles-ci.

9.1.2.4 Bilan

L'écriture de l'algorithme de factorisation de Cholesky en Jade est relativement facile du fait de la sémantique séquentielle du langage. Les résultats obtenus sur machine à mémoire partagée sont corrects. Cependant, de part l'algorithme d'ordonnancement à la volée implanté dans le support d'exécution de Jade, l'exécution sur machine à mémoire distribuée ne fournit pas d'accélération : le placement des tâches obtenu par l'algorithme d'ordonnancement à la volée engendre trop de communications. Jade n'est pas capable d'utiliser un autre type d'ordonnancement. Ce genre de problème ne se prête donc pas à une bonne parallélisation en Jade.

9.1.3 PYRROS/RAPID

PYRROS [56] et RAPID [126, 45] sont des environnements de programmation spécialisés pour les applications dont le graphe est connu avant l'exécution. Une exécution est alors composée d'une phase d'inspection du programme qui permet la construction du graphe de précédence de l'application (le graphe ne permet pas la représentation de diffusion de donnée) suivi d'une phase d'ordonnancement et d'une phase d'exécution. L'ordonnancement est composée d'un regroupement de tâches par l'algorithme DSC suivi d'un algorithme de placement de ces tâches sur les processeurs.

Une implantation, dans le système RAPID, d'une factorisation de Cholesky a été proposée dans [46]. Cette factorisation est de type bidimensionnelle.

9.1.3.1 Écriture de l'algorithme

RAPID est avant tout un système permettant l'exécution de programme représenté sous la forme d'un graphe de tâches. Cependant, seule une interface rudimentaire en C et FORTRAN permet de décrire explicitement le graphe de tâche. L'écriture de l'algorithme est donc relativement plus complexe que l'écriture obtenue avec d'autre langages de programmation haut niveau, tels que Cilk, Jade ou Athapascan-1.

9.1.3.2 Exécution de l'application

L'exécution de l'application représentée sous la forme d'un graphe de tâches est réalisée en deux étapes. Tout d'abord, un ordonnancement statique du graphe est réalisé. Cet ordonnancement est basé sur un regroupement des tâches par l'algorithme DSC puis par un placement de ces groupes de tâches sur les processeurs disponibles.

Une fois que ce placement est calculé, les tâches du graphe sont exécutées sur les processeurs en respectant les contraintes de précédence du graphe et le placement calculé par l'ordonnanceur. Pour cette exécution la particularité du système RAPID par rapport à PYROS est d'exploiter directement les mécanismes d'écriture en mémoire à distance proposés par les machines. Le gain annoncé sur le temps d'exécution par rapport au système PYROS [56] identique au système RAPID si ce n'est qu'il utilise des mécanismes de communication par échange de messages est de plus de 50% sur 16 processeurs. Cependant cette implantation rend RAPID peut portable d'une machine à une autre. Actuellement RAPID est porté sur la Meiko CS-2 et sur les Cray T3D et T3E qui offrent chacun un tel mécanisme d'écriture en mémoire à distance.

9.1.3.3 Expérimentations

Les expérimentations ont été réalisées sur une Meiko CS-2 et sur un Cray T3E qui sont des machines à mémoire distribuées supportant l'écriture à distance. Notons que les résultats présentés pour ces expérimentations ne prennent pas en compte les temps de création et d'ordonnancement du graphe.

Sur la Meiko CS-2, la puissance de crête d'un processeur est de 8 Mflops, le coût local d'une requête d'écriture à distance est de $9\mu s$ et la bande passante est de 39 Mbyte/s. Le temps séquentiel de factorisation de la matrice BCSSTK15 est de 26.7 secondes et les accélérations obtenues sur 16 et 32 processeurs sont respectivement de 6 et 10.

Sur le Cray T3E, la puissance de calcul obtenue sur le produit de matrice est de 388 Mflops, le coût local d'une requête d'écriture à distance est de $0.5 - 2\mu s$ et la bande passante est de 500 Mbyte/s. Les accélérations obtenues pour la factorisation de la matrice BCSSTK15 sur 16 et 32 processeurs sont respectivement de 5 et 9. Par comparaison, les résultats obtenus par notre implantation sur un Cray T3D, présentés dans la section 8.5.6 page 148 du chapitre précédant donnent, pour la factorisation de la même matrice, une accélération de 10 et 14 sur respectivement 16 et 32 processeurs.

9.1.3.4 Bilan

RAPID permet d'obtenir de bonnes performances sur l'application de factorisation numérique creuse de Cholesky. Cependant l'interface de programmation est rudimentaire. Par ailleurs, cet environnement de programmation est spécialisé pour le type d'application où le graphe est entièrement décrit en début d'exécution. Cet environnement est également

spécialisé pour les machines distribuées offrant des mécanismes d'écriture en mémoire à distance.

9.2 Factorisation numérique creuse en Athapascan-1

Nous présentons dans cette section l'écriture de la factorisation numérique creuse de Cholesky en Athapascan-1. L'algorithme utilisé est un algorithme avec partitionnement bidimensionnel de la matrice (*i.e.* l'algorithme 10 page 141), permettant de générer suffisamment de parallélisme en fin d'élimination. Le parallélisme sera donc exprimé au niveau des blocs d'un panneau.

9.2.1 Écriture de l'algorithme

La figure 9.2 page 160 montre la partie «parallèle» du programme de factorisation de Cholesky creux en Athapascan-1. Les types de tâches sont déclarés de la ligne 1 à la ligne 25 puis vient ensuite l'écriture de l'algorithme.

Pour représenter la matrice de blocs obtenue après la partition de la matrice initiale, deux types C++ sont utilisés. Le type `bloc` pour stocker un bloc de la matrice partitionnée et le type `matrix_bloc`<**Shared_rp_wp**<`bloc`> > pour stocker dans une matrice creuse des références contraintes sur des objets partagés de type `bloc`. Ce dernier type permet alors de représenter en mémoire partagée une matrice creuse partitionnée en blocs. Trois fonctions membres sont définies sur un objet `L` de ce type :

- `L.size()` retourne le nombre de lignes (et de colonnes) n de la matrice partitionnée en blocs (*i.e.* le nombre de partition qui a été appliqué sur la matrice creuse originale).

- `L(i,j)` retourne la référence contrainte du bloc d'indice (i, j) de la matrice creuse par blocs (implantée par une table de hachage).

- `L.next(i,j)` retourne l'indice de l'élément non nul situé immédiatement après l'élément d'indice (i, j) dans la colonne j, *i.e.*

$$k = \min\{i' \text{ tel que } i < i' \leq n \text{ et } L(i', j) \neq 0\}$$

La matrice partitionnée en bloc stockée dans un objet de ce type est alors passée en paramètre à la fonction `LLt_parallele` à la ligne 27.

```
1: struct potrf { // factorisation de Cholesky dense du bloc a
2:    void operator()(Shared_r_w<bloc> a) {
3:       potrf( a.access() );
4: }};
5: struct trsm {  // b ← b * (a^-1)^t
6:    void operator()(Shared_r<bloc> a, Shared_r_w<bloc> b) {
7:       trsm( a.read(), b.access() );
8: }};
9: struct add {    // b ← b + a
10:   void opérator()( bloc& b, const bloc& a) {
11:      b += a;
12: }};
13: struct syrk {  // b ← -a * a^t + b
14:   void operator()(Shared_r<bloc> a, Shared_cw<add,bloc> b) {
15:      bloc tmp;
16:      syrk( a.read(), tmp );  // tmp ← -a * a^t
17:      b.cumul( tmp );         // b ← b + tmp
18: }};
19: struct gemm { // c ← -a * b^t + c
20:   void operator()(Shared_r<bloc> a, Shared_r<bloc> b,
21:                   Shared_cw<add,bloc> c) {
22:      bloc tmp;
23:      syrk( a.read(), b.read(), tmp ); // tmp ← -a * b^t
24:      c.cumul( tmp );                  // c ← c + tmp
25: }};
26:
27: void LLt_parallele(matrix_bloc<Shared_rp_wp<bloc> >& L) {
28:   for( int k = 0; k<L.dim(); k++ ) {
29:      Fork<potrf>()( L(k,k) );
30:      for( int i = L.next(k,k); i<L.dim(); i = L.next(i,k) )
31:         Fork<trsm>()( L(k,k), L(i,k) );
32:      for( int j = L.next(k,k); j<L.dim(); i = L.next(j,k) ) {
33:         Fork<syrk>()( L(j,k), L(j,j) );
34:         for( int i = L.next(j,k); i<L.dim(); i = L.next(i,k) )
35:            Fork<gemm>()( L(i,k), L(j,k), L(i,j) );
36:      }
37:   }
38: }
```

Figure 9.2 – *Écriture de l'algorithme numérique de factorisation de Cholesky creux avec partitionnement bidimensionnel de la matrice.*

Comme Jade, Athapascan-1 est basé sur une sémantique séquentielle ; l'écriture de l'algorithme en Athapascan-1 est donc naturelle car très proche de l'écriture de l'algorithme séquentiel 10 page 141. Notons que l'itération sur les blocs non nuls d'une colonne de la matrice creuse (ligne 30, 32 et 34) est réalisée ici par l'appel de la fonction membre `L.next(i,j)` retournant le bloc non nul suivant de la colonne d'indice j.

Dans le programme réel, une estimation du coût des tâches est également fournie lors de leurs créations grâce à l'annotation de coût permis par Athapascan-1 de la manière suivante : **Fork**<[type_tâche]>([information de coût])([paramètres]).

9.2.2 Ordonnancement du graphe de tâches

Contrairement aux applications d'algèbre linéaire dense vues dans le chapitre 7, l'application de Cholesky creuse est une application irrégulière. Cette irrégularité s'exprime principalement au niveau de la granularité des tâches de l'application. Certaines tâches peuvent être de taille importante alors que d'autres peuvent être de taille très petite. Bien que la taille maximale des tâches puisse être bornée lors de la phase de factorisation symbolique (lors de la partition bidimensionnelle de la matrice), la taille des tâches dépend principalement des données en entrées ; c'est-à-dire de la structure creuse de la matrice à factoriser. Le calcul d'un bon ordonnancement du graphe est donc difficile et ne peut être fait qu'à partir d'informations sur les entrées.

Dans l'implantation MPI présentée dans le chapitre précédent, le placement des blocs sur les processeurs est calculé en trois étapes. L'algorithme de Geist et Ng [51] calcule les sous arbres de l'arbre d'élimination qui seront exécutés localement à un processeur. L'algorithme de placement proportionnel [102] est appliqué sur le restant de l'arbre d'élimination conduisant à associer un groupe de processeurs à chaque colonne de blocs. Finalement, un algorithme glouton est appliqué sur chaque colonne de blocs pour placer les blocs sur le groupe de processeurs associé à la colonne.

Nous avons utilisé pour la version Athapascan-1 un algorithme d'ordonnancement plus simple basé sur l'algorithme ETF déjà implanté dans la bibliothèque. L'algorithme ETF utilisé directement ne permet pas d'obtenir de bonnes performances sur cette application car il n'exploite pas la localité des calculs présents dans l'arbre d'élimination et conduit à un placement des tâches sur les processeurs générant beaucoup trop de communications.

Pour remédier à ce problème, nous avons appliqué l'algorithme de placement proportionnel sur l'ensemble de l'arbre d'élimination pour associer à chaque tâche un groupe de processeurs dans lequel celle-ci doit s'exécuter. À chaque tâche est ainsi associée une contrainte de placement. Une adaptation simple de l'algorithme ETF qui respecte les contraintes de placement ainsi calculées est ensuite utilisée pour obtenir un placement des tâches sur les processeurs.

9.2.3 Évaluation expérimentale

Nous avons mesuré expérimentalement les performances obtenues sur la factorisation creuse de Cholesky, en utilisant la politique d'ordonnancement proposée précédemment. Ces expérimentations ont été menées sur la machine SP1 présentée dans la section 7.5.

La figure 9.3 page 162 montre les performances obtenues sur les matrices creuses utilisées pour les expérimentations du chapitre précédent et dont les caractéristiques sont présentées dans le tableau 8.1 page 149. Notons que comme pour les mesures réalisées dans le chapitre précédent sur l'implantation directement en MPI du problème, le temps nécessaire au calcul de l'ordonnancement des tâches n'est pas pris en compte.

Ces résultats sont à comparer avec ceux de la figure 8.8 page 151 obtenus sur la même machine mais avec une implantation directe en MPI. On remarque alors que sur 16 processeurs, mis à part pour la matrice BCSSTK17 les résultats obtenus sont moins bons. Ceci s'explique par l'algorithme d'ordonnancement utilisé qui est plus simple (regroupement de tâches moins fin) que celui utilisé dans la version écrite en MPI. Un travail complémentaire resterait à faire pour implanter les mêmes techniques que celles mises en œuvre dans la version MPI.

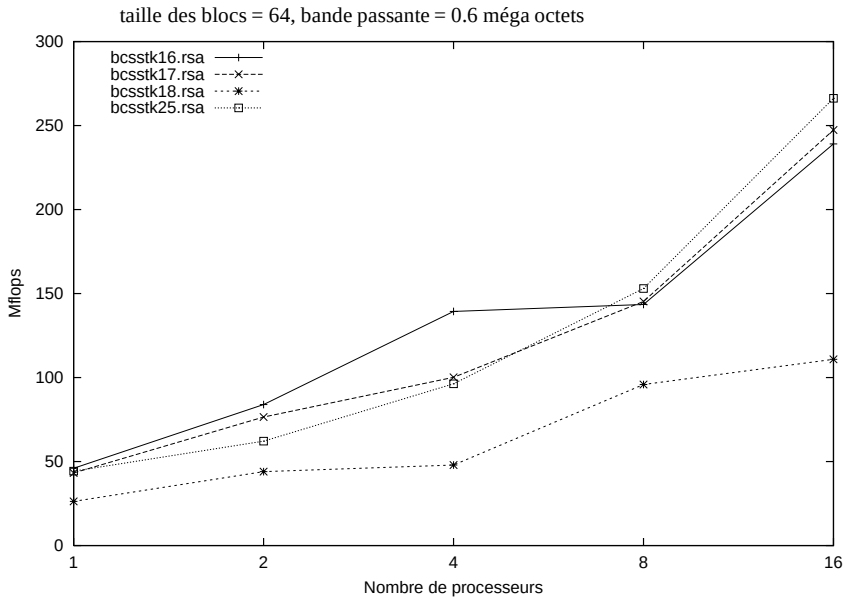

Figure 9.3 – *Performances obtenues pour différentes matrices creuses en fonction du nombre de processeurs utilisés.*

Une des difficultés rencontrées lors de ces expérimentations a été le choix des paramètres du modèle de machine nécessaire pour calculer l'ordonnancement avec prise en compte des communications. Nous avons constaté une instabilité des performances obte-

nues lorsque le rapport entre la puissance de calcul du processeur et la bande passante du réseau varie autour des valeurs réelles mesurées sur la machine. La figure 9.4 page 163 illustre bien ce problème puisqu'elle montre les performances obtenues en fonction de la valeur de la bande passante prise pour modéliser la machine, chaque courbe représentant les performances obtenues sur un nombre fixé de processeur.

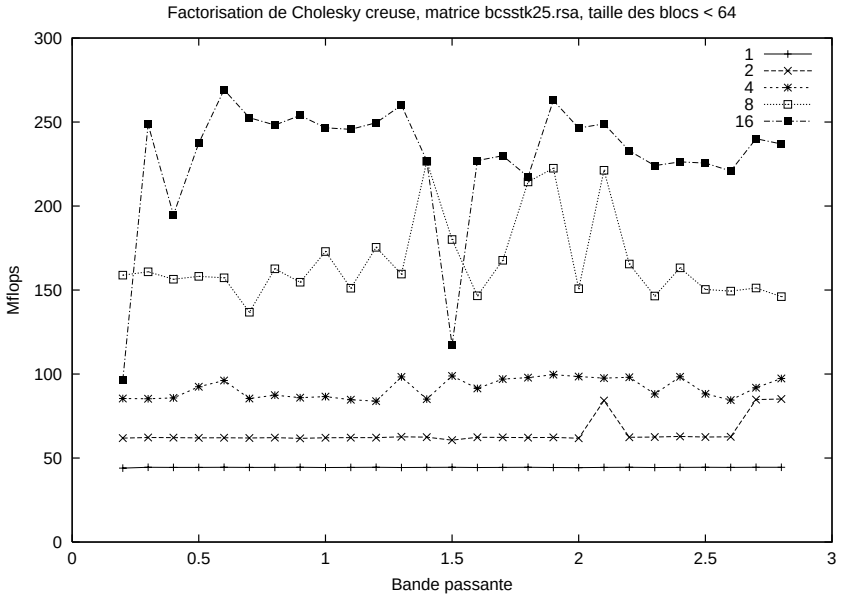

Figure 9.4 – *Performances obtenues en fonction de la valeur de la bande passante prise pour modéliser la machine. Chaque courbe correspond aux performances obtenues sur un même nombre de processeurs. Les expérimentations sont réalisées sur 1, 2, 4, 8 et 16 processeurs.*

9.3 Conclusion

Nous avons proposé dans ce chapitre une écriture de la factorisation de Cholesky creuse en Athapascan-1. De part les caractéristiques du modèle de programmation, cette écriture est très simple puisque le code dédié au parallélisme de l'application est réduit à une quarantaine de lignes ; de plus ce code consiste en une transposition directe de l'algorithme séquentiel. Comparé au modèle de programmation par échange de messages qui a été utilisé dans la section 8.5.4 page 142 pour implanter cette application, le modèle de programmation d'Athapascan-1 simplifie donc considérablement l'écriture de cette application, avec un gain en nombre de lignes de code très important (34 lignes seulement de code «parallèle»).

Par ailleurs Athapascan-1 permet également de supprimer ou de simplifier le problème de l'ordonnancement de l'application. Pour cette application de factorisation, les ordonnancements généraux (ETF, DSC) ne nous ont pas permis d'attendre les performances obtenues avec l'implantation réalisée en MPI. Cependant une légère modification d'un algorithme d'ordonnancement existant de la bibliothèque Athapascan-1 a alors conduit à des performances proches de celles obtenues en MPI pour certaines matrices.

De plus, par rapport aux autres implantations de cette application dans des langages ou environnements de haut niveau, les performances obtenues avec Athapascan-1 sont particulièrement bonnes. Seule l'implantation réalisée dans RAPID semble offrir des performances supérieures en terme de taux d'utilisation des processeurs (Mflops) [1] mais ceci n'est obtenu qu'au prix d'une réduction de la portabilité puisque RAPID s'appuie sur des mécanismes matériels d'écriture en mémoire à distance.

Cependant des problèmes subsistent principalement avec l'ordonnancement statique utilisé. Les erreurs dans la modélisation de la machine ainsi que dans l'évaluation des coûts d'exécutions font que l'ordonnancement obtenu peut conduire à une exécution non performante. Ces problèmes ne sont pas spécifiques à l'implantation en Athapascan-1 mais se posent pour les deux versions MPI et Athapascan-1 de la factorisation creuse de Cholesky.

Plusieurs solutions peuvent être envisagées pour résoudre ce problème. Tout d'abord, l'utilisation d'un modèle de machine comme le modèle LogP, plus proche des machines réelles que le modèle délai utilisé ici, devrait permettre d'améliorer la qualité de l'ordonnancement obtenu. Dans ce cadre, l'utilisation de l'algorithme ETFR [77] qui est une adaptation de l'algorithme ETF au modèle LogP pourrait être envisagée.

Par ailleurs il semble nécessaire de calibrer les coûts d'exécution des tâches sur la machine cible pour fournir des estimations plus réalistes. Cependant, cette calibration des coûts de tâches va à l'encontre de la portabilité de l'application puisque elle n'est valable que pour un type de machine parallèle. Une autre solution serait alors de mettre en œuvre des algorithmes d'ordonnancements mixant les approches statiques et dynamiques comme par exemple celui proposé dans la section 5.5 page 86, moins tributaire des estimations de coût d'exécution fournies par l'application.

1. La comparaison reste cependant difficile car les expérimentations ont eté réalisées sur des machines différentes.

10

Conclusions et perspectives

Nous avons présenté dans ce document l'environnement de programmation parallèle Athapascan-1 et nous l'avons validé sur des applications du calcul scientifique.

Ce modèle de programmation permet une écriture simple des applications parallèles régulières mais surtout irrégulières en fournissant plusieurs mécanismes permettant d'abstraire la machine parallèle et d'exprimer le parallélisme à un haut niveau. Ces mécanismes sont les suivants :

– Une mémoire partagée distribuée qui décharge le programmeur de la distribution et de la communication des données.

– Une virtualisation du nombre de processeurs par la décomposition de l'application sous la forme de tâches indépendamment du nombre de processeurs, l'allocation de ces tâches sur les ressources disponibles est prise en charge par le système chargé de l'exécution.

– Une analyse automatique des dépendances de données entre les tâches libérant le programmeur de la gestion des synchronisations entre les tâches.

Plusieurs applications ont été implantées sur cet environnement. Nous avons présenté dans cette thèse plusieurs de ces applications, en particulier des algorithmes de factorisation numérique pour les matrices denses et creuses. Ces applications ont permis de valider Athapascan-1 à la fois sur des problèmes réguliers avec les factorisations de matrice dense mais également sur les problèmes irréguliers avec la factorisation de Cholesky creuse.

Les résultats expérimentaux obtenus montrent que ce modèle de programmation, bien qu'offrant des mécanismes d'expression du parallélisme de haut niveau, peut être mise

en œuvre efficacement aussi bien sur des machines parallèles à mémoire partagée qu'à mémoire distribuée. Les performances obtenues sont alors proches, voir meilleures, que celles obtenues lorsque l'application est codée dans un modèle de programmation bas niveau comme MPI.

Ceci s'explique par les différents choix qui ont conduit à la définition du modèle de programmation et à sa mise en œuvre :

- Le parallélisme d'un programme Athapascan-1 est implicite mais la granularité de ce parallélisme est explicite (la tâche). Par ailleurs l'ensemble des effets de bord réalisés par les tâches sont connus lors de leur création. L'extraction du parallélisme de l'application peut alors être réalisée par l'analyse, lors de l'exécution, des dépendances de données entre les tâches, dépendances qui sont transcrites sous la forme d'un graphe de flot de données.

- La mémoire partagée distribuée de niveau objet, permet de simuler une mémoire physiquement partagée mais à une granularité fixée par le programmeur.

- La connaissance du flot de données autorise la mise en ouvre des techniques d'ordonnancements statiques lorsque l'application le permet (en particulier lorsque l'ensemble du graphe de flot de données est connu en début d'exécution et qu'une évaluation du coût des tâches et de la taille des données accédées est fournie). Cela est par exemple le cas pour les applications présentées dans cette thèse.

- Le niveau d'abstraction du modèle de programmation permet d'exécuter une application en exploitant au mieux les possibilités offertes par la machine parallèle. Par exemple sur les architectures SMP ou réseaux de SMP, la mise en œuvre proposée dans la bibliothèque Athapascan-1 utilise des processus légers et permet ainsi d'exploiter efficacement les diffèrentes ressources (processeurs, mémoire). Il serait également envisageable d'utiliser les mécanismes de lecture et d'écriture en mémoire à distance lors qu'ils sont présents sur la machine parallèle comme le fait par exemple RAPID [46].

Il ressort également de ces expérimentations que la difficulté majeure pour la mise en œuvre efficace du modèle de programmation Athapascan-1 sur machine à mémoire distribuée reste l'obtention d'un bon ordonnancement. Par ailleurs, aucun algorithme général d'ordonnancement parmi ceux utilisés ne s'est imposé sur les applications et les machines testées. Par exemple pour l'application régulière de factorisation dense de Cholesky, l'algorithme de placement cyclique bidimensionnel, privilégiant la réduction des communications, a donné les meilleurs résultats sur un réseau de stations alors que l'algorithme ETF, privilégiant la régulation de charge de calcul entre les processeurs a donné les meilleurs résultats sur une machine parallèle SP1. Pour l'application irrégulière de factorisation creuse de Cholesky, c'est un algorithme spécifique de regroupement de tâches à l'application qui a permis d'obtenir les meilleures performances.

Partant de ce constat, la bibliothèque Athapascan-1 a été construite de façon à séparer la politique d'ordonnancement utilisée des autres mécanismes nécessaires à l'exécution (analyse des dépendances de données entre les tâches en vue de la génération du graphe de flot de données, gestion de la mémoire partage distribuée, *etc*). Pour cela elle fournit une interface simple d'utilisation qui de plus permet d'implanter des algorithmes d'ordonnancement adaptés à la fois aux applications et aux machines.

Perspectives

Les perspectives ouvertes à la suite de ces travaux sont multiples et recouvrent plusieurs des domaines de recherche abordés dans cette thèse : le modèle de programmation parallèle Athapascan-1, la mise en œuvre de ce modèle dans la bibliothèque Athapascan-1, les algorithmes d'ordonnancement et le portage d'applications sur Athapascan-1.

Modèle de programmation d'Athapascan-1

Une des restrictions imposées dans le modèle de programmation d'Athapascan-1 interdit la synchronisation entre une tâche mère et les tâches filles créées par elle. Autoriser cette synchronisation peut conduire à des problèmes de régulation de charge si aucun mécanisme de migration de tâche sur les point de synchronisations n'est mis en œuvre. Cependant cette contrainte peut s'avérer gênante car elle n'est pas naturelle pour un modèle de programmation qui par ailleurs supporte une sémantique séquentielle. Nous envisageons donc de supprimer cette restriction, dans un première temps sans mécanisme de migration de tâche (à utiliser alors avec modération). Des mécanismes de migration de tâche à la manières de Cilk où de PM^2 pourraient dans un deuxième temps être envisagés si nécessaires.

Une autre amélioration du modèle de programmation concerne les objets partagés et le contrôle des effets de bord sur ces objets par les tâches. Actuellement, les références contraintes sur les objets partagés du modèle de programmation ne fixent des contraintes que sur le type d'accès réalisé sur l'objet partagé (lecture, modification, affectation, accumulation). Ces contraintes pourraient alors être étendus pour restreindre l'accès à une partie seulement de l'objet partagé. Il serait alors possible par exemple de définir une référence pour l'accès à un sous tableau seulement d'un tableau présent en mémoire partagée ou une sous matrice d'une matrice en mémoire partagée.

Modèle d'exécution Athapascan-1

Plusieurs améliorations ou optimisations de la bibliothèque Athapascan-1, notamment sur la gestion distribuée du graphe de flot de données sont envisageables. Ces optimisations peuvent s'inspirer de celles misent en œuvre dans la version spécialisée pour les

ordonnancements statiques (présentée dans la 4.4.2.2 page 71). Par ailleurs une amélioration de l'intégration d'Athapascan-1 sur Athapascan-0 pour les communications d'objets peut être envisagée. Une version C++ d'Athapascan-0, Athapascan-0++ offrant notamment des mécanismes de communications d'objets C++ est actuellement en cours de développement au sein du projet APACHE et devrait conduire à une meilleure efficacité dans les communications.

Ces optimisations, en réduisant le surcoût lié à la gestion de la mémoire partagée devraient alors permettre l'utilisation d'algorithmes d'ordonnancement à la volée pour les applications présentées dans ce document qui utilisent intensivement la mémoire partagée.

Ordonnancement

L'obtention de performances sur un modèle de programmation haut niveau comme Athapascan-1 passe par un bon ordonnancement des tâches de l'application. Plusieurs perspectives sont alors envisageables dans le but d'améliorer cet ordonnancement.

Tout d'abord, pour les algorithmes d'ordonnancement statiques, nous avons déjà discuté l'intérêt d'utiliser le modèle LogP dans le but de modéliser plus fidèlement les architectures distribuées.

Une autre approche, consiste à mixer les techniques statiques et dynamiques d'ordonnancement. Dans ce cas, l'ordonnancement peut aussi exploiter les informations connues de l'exécution future comme la structure du graphe de flot de données ou l'estimation du coût des tâches mais contrairement aux approches purement statique, l'ordonnanceur peut également utiliser des informations obtenues à l'exécution. Par exemple, le placement des tâches peut être calculé statiquement à partir des informations connues sur le graphe mais ce placement peut être remis en cause en cours d'exécution suivant le déroulement de celle-ci. L'algorithme proposé dans la section 5.5 page 86 est un exemple utilisant une approche mixte. L'intérêt de cette approche est de produire des résultats d'ordonnancement (temps d'exécution) moins sensibles aux informations fournies sur les tâches ou de la modélisation de la machine utilisée.

D'autre part, plusieurs des techniques utilisées dans le domaine de l'ordonnancement pourraient également être envisagées pour Athapascan-1 comme par exemple la duplication de tâches (déjà partiellement mis en œuvre dans la version spécialisée de la bibliothèque Athapascan-1) qui permet de réduire les communications.

Par ailleurs, la bibliothèque Athapascan-1 associée à l'ensemble des applications qui y sont portées constitue un bon support pour la validation expérimentale des algorithmes d'ordonnancement. Ceux-ci peuvent être facilement implantés grâce à l'interface fournie. Ils peuvent alors être évalués expérimentalement sur un ensemble d'applications réelles.

Annexe

Annexe A

Preuve de la borne sur l'algorithme d'ordonnancement ERT à la volée

Pour des raisons de simplicité, on considère que les processeurs sont synchronisés. Le temps d'exécution peut alors être discrétisé en intervalles correspondant à un cycle, donc à l'exécution d'une instruction.

On note $S(t)$ et $F(t)$ le temps respectivement de début de fin d'exécution de la tâche t.

Les intervalles de temps peuvent alors être partitionnés en deux ensembles disjoints A et I. L'ensemble A contient les intervalles de temps pendant lesquels tous les processeurs sont actifs, c'est-à-dire qu'ils exécutent une tâche. L'ensemble B contient les intervalles de temps pendant lesquels au moins un des processeurs est inactif.

Soit une tâche $t_{s(1)} \in G$ commençant son exécution à l'instant $S(t_{s(1)})$ sur le processeur $ALLOC(t_{s(1)})$. Deux cas sont alors possibles :

Cas 1 : $S(t_{s(1)}) - 1 \in I$.

De par la définition de I, il existe un processeur p_q inactif pendant le cycle $S(t_{s(1)}) - 1$.

Nous pouvons alors démontrer par l'absurde qu'il existe une tâche $t_{s(2)}$ prédécesseur immédiat de la tâche $t_{s(1)}$ vérifiant :

$$F(t_{s(2)}) + \max_{1 \leq i,j \leq m, t_l \in pred(t_{k+1})} C(t_l, t_{s(1)}, p_i, p_j) \geq S(t_{s(1)})$$

Supposons cette affirmation fausse. Alors pour toutes tâches $t_{s(2)} \in pred(t_{s(1)})$, il existe un $\epsilon > 0$ tel que :

$$F(t_{s(2)}) + \max_{1 \leq i,j \leq m, t_l \in pred(t_{k+1})} C(t_l, t_{s(1)}, p_i, p_j) = S(t_{s(1)}) - \epsilon$$

Prenons comme tâche $t_{s(2)}$ la tâche vérifiant :

$$F(t_{s(2)}) = \max_{t_i \in pred(t_{s(1)})} F(t_{s(1)})$$

$t_{s(2)}$ est la dernière tâche terminée de l'ensemble $pred(t_{s(1)})$.

$F(t_{s(2)})$ est la date à laquelle la tâche $t_{s(1)}$ est devenue prête. C'est aussi la date à laquelle la décision de placement de la tâche $t_{s(1)}$ a été prise par l'ordonnanceur. L'algorithme place cette tâche sur le processeur $ALLOC(t_{s(1)})$ vérifiant :

$$S(t_{s(1)}) = r(t_{s(1)}, ALLOC(t_{s(1)})) = \min_{1 \leq j \leq m} r(t_{s(1)}, p_j)$$

où

$$r(t_i, p_j) = max\{DISPO(p_j), F(t_{s(2)}) + \max_{t_k \in pred(t_i)} \{C(t_k, t_i, ALLOC(t_k), p_j)\}\}$$

On a donc pour $1 \leq j \leq m$:

$$r(t_{s(1)}, p_j) = max\{DISPO(p_j), S(t_{s(1)}) - \epsilon\}$$

Donc :

$$r(t_{s(1)}, p_q) = max\{DISPO(p_q), S(t_{s(1)}) - \epsilon\}$$

Comme $DISPO(p_q) < S(t_{s(1)})$ et $\epsilon > 0$ alors $r(t_{s(1)}, p_q) < S(t_{s(1)})$.

On obtient alors la contradiction :

$$r(t_{s(1)}, p_q) < S(t_{s(1)}) = r(t_{s(1)}, ALLOC(t_{s(1)})) = \min_{1 \leq j \leq m} r(t_{s(1)}, p_j)$$

Cas 2 : $S(t_{s(1)}) - 1 \in A$. Soit la date $u \in B$ vérifiant :

$$\forall x \text{ tel que } u < x < S(t_{s(1)}), \text{ alors } x \in B$$

Alors il existe une tâche $t_{h(1)}$ qui est, soit identique à la tâche $t_{s(1)}$, soit une tâche prédécesseur de $t_{s(1)}$, et telle que $S(t_{h(1)}) > u$ et il existe une tâche $t_{s(2)}$ prédécesseur immédiat de $t_{h(1)}$ vérifiant :

$$S(t_{s(2)}) \leq u$$

$$F(t_{s(2)}) + \max_{1 \leq i,j \leq m, t_l \in pred(t_{k+1})} C(t_l, t_{s(1)}, p_i, p_j) \geq S(t_{h(1)})$$

La démonstration est identique à celle du premier cas.

Nous pouvons alors construire une chaîne de tâches du graphe G, $(t_{c(1)}, \ldots, t_{c(s)})$, recouvrant l'ensemble I et tel que, pour tout $i, 1 \leq i \leq s$:

$$F(t_{c(i)}) + \max_{1 \leq i,j \leq m, t_l \in pred(t_{k+1})} C(t_l, t_{k+1}, p_i, p_j)) \geq S(t_{c(i+1)})$$

Soit Φ la somme des temps d'inactivités de chacun des processeurs pendant l'exécution du graphe G, on a alors :

$$\begin{aligned}
\Phi &\leq (m-1)\sum_{i=1}^{s}\mu(t_{c(i)}) + \\
&\quad m\sum_{i=1}^{s} \max_{1 \leq i,j \leq m, t_l \in pred(t_{k+1})} C(t_l, t_{k+1}, p_i, p_j) \\
&\leq (m-1)\sum_{i=1}^{s}\mu(t_{c(i)}) + \\
&\quad m\sum_{i=1}^{s} \max_{1 \leq i,j \leq m, t_l \in pred(t_{k+1})} \tau\eta(t_l, t_{k+1}) \\
&\leq (m-1)T_\infty + m\tau C_{max^*}
\end{aligned}$$

Le temps total d'exécution est alors borné par :

$$\frac{T_1}{m} + \left(1 - \frac{1}{m}\right) T_\infty + \tau C_{max^*}$$

Ce qui termine la preuve. □

Bibliographie

[1] Alpatov (Philip), Baker (Gregory), Edwards (H. Carter), Gunnels (John), Morrow (Greg), Overfelt (James) et van de Geijn (Robert). – PLAPACK: Parallel linear algebra libraries design overview. *In : SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, éd. par ACM. – California, USA, 1997.

[2] Amestoy (P.R.), Davis (T.A.) et Duff (I.S.). – An Approximate Minimum Degree Ordering Algorithm. *SIAM J.Matrix Anal.Appl.*, vol. 17, n4, octobre 1996, pp. 886–905.

[3] Amza (C.), Cox (A. L.), Dwarkadas (S.), Keleher (P.), Lu (H.), Rajamony (R.), Yu (W.) et Zwaenepoel (W.). – Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, vol. 29, n2, février 1996, pp. 18–28.

[4] Anderson (E.), Bai (Z.), Bischof (C.), Demmel (J.), Dongarra (J.), Du Croz (J.), Greenbaum (A.), Hammarling (S.), McKenney (A.), Ostrouchov (S.) et Sorensen (D.). – *LAPACK Users' Guide, Second Edition*. – Philadelphia, PA, SIAM, 1995.

[5] Ashcraft (C.), Eisenstat (S.C.) et Liu (J.W.H.). – A Fan-in Algorithm for Distributed Sparse Numerical Factorization. *SIAM J.Sci.Stat.Comput.*, vol. 11, n3, mai 1990, pp. 593–599.

[6] Ashcraft (C.) et Grimes (R.). – The Influence of Relaxed Supernode Partitions on the Multifrontal Method. *ACM Trans.Math.Soft.*, vol. 15, n4, décembre 1989, pp. 291–309.

[7] Bader (David A.) et JáJá (Joseph). – SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, vol. 58, 1999, pp. 92–108.

[8] Bal (H. E.), Kaashoek (M. F.) et Tanenbaum (A. S.). – Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, vol. 18, n3, mars 1992, pp. 190–205.

[9] Bal (Henri E.), Bhoedjang (Raoul), Hofman (Rutger), Jacobs (Ceriel), Langendoen (Koen), Rühl (Tim) et Kaashoek (M. Frans). – Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, vol. 16, n1, février 1998, pp. 1–40.

[10] Blumofe (R. D.), Joerg (C. F.), Kuszmaul (B. C.), Leiserson (C. E.), Randall (K. H.), Zhou (Y. C. E.), Randall (K. H.) et Zhou (Y.). – Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, vol. 30, n8, août 1995, pp. 207–216.

[11] Blumofe (Robert D.), Joerg (Christopher F.), Kuszmaul (Bradley C.), Leiserson (Charles E.), Randall (Keith H.) et Zhou (Yuli). – Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, vol. 37, n1, août 1996, pp. 55–69.

[12] Boeres (C.), Rebello (V.E.F.) et Nascimento (A.P.). – Scheduling arbitrary task graphs on logp machines. *In : 5th International Euro-Par Conference on Parallel Processing (EUROPAR'99)*. – Toulouse, France, septembre 1999.

[13] Böhm (A. P. W.), Cann (D. C.), Feo (J. T.) et Oldehoeft (R. R.). – *SISAL 2.0 Reference Manual*. – Report nCS-91-118, Fort Collins, CO, Computer Science Department, Colorado State University, 1991.

[14] Briat (Jacques), Gautier (Thierry) et Roch (Jean-Louis). – On-line scheduling. *In : Proc. of ESPPE'96, Parallel Programming Environments for High Performance Computing*, pp. 95–108.

[15] Carissimi (Alexandre). – *Le noyau executif Athapascan-0 et l'exploitation de la multiprogrammation legere sur les grappes de stations multiprocesseurs.* – France, Thèse de doctorat, Institut National Polytechnique de Grenoble, novembre 1999.

[16] Carissimi (Alexandre) et Pasin (Marcelo). – Athapascan: An experience on mixing MPI communications and threads. *Lecture Notes in Computer Science*, vol. 1497, 1998, pp. 137–144.

[17] Carriero (N.) et Gelernter (D.). – How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, vol. 21, n3, septembre 1989.

[18] Cary (John R.), Shasharina (Svetlana G.), Cummings (Julian C.), Reynders (John V. W.) et J. (Hinker Paul). – Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, novembre 1996.

[19] Cavalheiro (Gerson G. H.). – *Athapascan 1: Interface générique pour l'ordonnancement dans un environnement d'exécution parallèle.* – France, Thèse de doctorat, Institut National Polytechnique de Grenoble, novembre 1999.

[20] Cavalheiro (Gerson G. H.), Denneulin (Yves) et Roch (Jean-Louis.). – A general modular specification for distributed schedulers. *In : EuroPar'98*. – Southampton, England., septembre 1998.

[21] Cavalheiro (Gerson G. H.) et Roch (Jean-Louis). – Un schéma modulaire pour l'écriture des ordonnanceurs. *In : RenPar'98*. – Strasbourg, France., juin 1998.

[22] Choi (Jaeyoung), Demmel (J.), Dhillon (L.), Dongarra (J.), Ostrouchov (L. S.) et Petitet (A.). – Scalapack, a portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, vol. 97, 1996, pp. 1–15.

[23] Choi (Jaeyoung), 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, 1996, pp. 173–184.

[24] Chrétienne (P.), Coffman (E. G.), Lenstra (J. K.) et Liu (Z.). – *Scheduling Theory and its Applications*. – John Wiley and Sons, 1995.

[25] Colin (J.-Y.) et Chrétienne (P). – Cpm scheduling with small interprocessor communication delays. *Operations Research*, vol. 39, n3, 1991.

[26] Culler (David E.), Karp (Rachard M.), Patterson (David), Sahay (Abhijit), Santos (Eunice E.), Schauser (Klaus Erik), Subramonian (Ramesh) et von Eicken (Thorsten). – LogP: A practical model of parallel computation. *Communications of the ACM*, vol. 39, n11, novembre 1996, pp. 78–85.

[27] Culler (David E.), Karp (Richard M.), Patterson (David A.), Sahay (Abhijit), Schauser (Klaus E.), Santos (Eunice), Subramonian (Ramesh) et von Eicken (Thorsten). – LogP: towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, vol. 28, n7, juillet 1993, pp. 1–12.

[28] de Oliveira Stein (Benhur). – *Visualisation interactive et extensible de programmes parallèles à base de processus légers*. – Thèse de doctorat, Université Joseph Fourier, France, Octobre 1999.

[29] Deng (X.), Koutsoupias (E.) et MacKenzie (P.). – Competitive implementation of parallel programs. *Algorithmica*, vol. 23, 1999.

[30] Dijkstra (E. W.). – Solution of a problem in concurrent programming control. *Communications of the ACM*, vol. 8, n9, septembre 1965, p. 569.

[31] Dongarra (J. J.) et Whaley (R. C.). – *A Users' Guide to the BLACS v1.0*. – Rapport technique, University of Tennessee, 1995.

[32] Dongarra (Jack J.), Duff (Iain S.), Sorensen (Danny C.) et Henk (A. van der Vorst). – *Numerical Linear Algebra for High-Performance Computers*. – SIAM, 1998.

[33] Dongarra (J.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. 1–28.

[34] 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, février 1998, p. 17.

[35] Doreille (Mahtias), Galilée (François), Cavalheiro (Gerson-Geraldo-Homrich) et Roch (Jean-Louis). – Athapascan-1 : On-line building data flow graph in a parallel language. *In : Pact'98*. – Paris, France, octobre 1998.

[36] Dos Reis (Gabriel). – *Vers une nouvelle approche du calcul scientifique en C++*. – Rapport technique n3362, INRIA, février 1998.

[37] Duff (Iain), Grimes (Roger G.) et Lewis (John G.). – *Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)*. – Rapport technique nTR/PA/92/86, CERFACS, octobre 1992.

[38] Eswar (Kalluri), Sadayappan (P.), Huang (Chua-Huang) et Visvanathan (V.). – Supernodal sparse cholesky factorization on distributed-memory multiprocessors. *In : Proceedings of the 1993 International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, éd. par Hariri, Salim; Berra (P. Bruce). pp. 18–22. – Syracuse, NY, août 1993.

[39] Facq (L.) et Roman (J.). – Distribution par bloc pour une factorisation parallèle de Cholesky. *In : Parallèlisme et applications irrégulières*, éd. par Authié (G.) et al., pp. 135–147. – Paris, Hermès, 1995.

[40] Feautrier (Paul). – Compiling for massively parallel architectures: a perspective. *Microprogramming and Microprocessors*, vol. 41, 1995, pp. 425–439.

[41] Fortune (Steven) et Wyllie (James). – Parallelism in random access machines. *In : Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 114–118. – San Diego, California, 1–3 mai 1978.

[42] Foster (Ian), Kesselman (Carl) et Tuecke (Steven). – The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, vol. 37, n1, août 1996, pp. 70–82.

[43] Friedman (Roy), Goldin (Maxim), Itzkovitz (Ayal) et Schuster (Assaf). – MILLIPEDE: Easy parallel programming in available distributed environments. *Software—Practice and Experience*, vol. 27, n8, août 1997, pp. 929–965.

[44] Frigo (Matteo), Leiserson (Charles E.) et Randall (Keith H.). – The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, vol. 33, n5, mai 1998, pp. 212–223.

[45] Fu (Cong), Jiao (Xiangmin) et Yang (Tao). – Efficient sparse LU factorization with partial pivoting on distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, n2, février 1998, pp. 109–124.

[46] Fu (Cong) et Yang (Tao). – Run-time techniques for exploiting irregular task parallelism on distributed memory architectures. *Journal of Parallel and Distributed Computing*, vol. 42, n2, mai 1997, pp. 143–156.

[47] Gainzburg (Ilan). – *Athapascan-0b : Intégration efficace et protable de multiprogrammation légère et de communications.* – France, Thèse de doctorat, Institut National Polytechnique de Grenoble, septembre 1997.

[48] Galilée (François). – *Athapascan-1: Interprétation distribuée du flot de données d'un programme parallèle.* – France, Thèse de doctorat, Institut National Polytechnique de Grenoble, septembre 1999.

[49] Gallivan (K. A.), Plemmons (R. J.) et Sameh (A. H.). – Parallel algorithms for dense linear algebra computations. *SIAM Review*, vol. 32, n1, mars 1990, pp. 54–135.

[50] Gautier (Thierry), Roch (Jean-Louis) et Villard (Gilles). – Regular versus irregular problems and algorithms. *In : Proc. of IRREGULAR'95, Lyon, France*. – Springer-Verlag.

[51] Geist (G.A.) et Ng (E.). – Task Scheduling for Parallel Sparse Cholesky Factorization. *Internat. J. Parallel Programming*, vol. 18, 1989, pp. 291–314.

[52] George (A.). – Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, vol. 10, 1973, pp. 345–363.

[53] George (A.) et Liu (J.W.H.). – The Evolution of the Minimum Degree Ordering Algorithm. *SIAM Review*, vol. 31, n1, mars 1989, pp. 1–19.

[54] George (Alan) et Liu (Joseph W. H.). – *Computer solution of large sparse positive definite systems*. – Englewood Cliffs, NJ 07632, USA, Prentice-Hall, 1981, *Prentice-Hall series in computational mathematics*, xii + 324p.

[55] George (Alan) et Liu (Joseph W.H.). – An optimal algorithm for symbolic factorization of symmetric matrices. *SIAM Journal of Computing*, vol. 9, n3, août 1980, pp. 583–593.

[56] Gerasoulis (Apostolos) et Yang (Tao). – 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.

[57] Gibbons (Phillip B.), Matias (Yossi) et Ramachandran (Vijaya). – The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, vol. 28, n2, avril 1999, pp. 733–769.

[58] Graham (R. L.). – Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, vol. 17, n2, mars 1969, pp. 416–429.

[59] Grimshaw (Andrew S.). – Easy-to-use object-oriented parallel processing with Mentat. *Computer*, vol. 26, n5, mai 1993, pp. 39–51.

[60] Gupta (Anshul), Karypis (George) et Kumar (Vipin). – Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, n5, mai 1997, pp. 502–520.

[61] Halstead (R.H.). – Parallel computing using multilisp. *In : Parallel Computation and Computers for Artificial Intelligence*, éd. par Kowalik (J.S.), pp. 21–49. – Kluwer Academic Publishers, 1988.

[62] Heath (M.T.), Ng (E.) et Peyton (B.W.). – Parallel Algorithms for Sparse Linear Systems. *SIAM Review*, vol. 33, n3, septembre 1991, pp. 420–460.

[63] Hendrickson (B.) et Leland (R>). – *The CHACO User's Guide. Version 2.0*. – Rapport technique nSAND94-2692, Albuquerque, Sandia National Laboratories, octobre 1994.

[64] Hendrickson (Bruce) et Rothberg (Edward). – Improving the run time and quality of nested dissection ordering. *SIAM Journal on Scientific Computing*, vol. 20, n2, mars 1999, pp. 468–489.

[65] High Performance Fortran Forum. – *High Performance Fortran Language Specification, Version 2.0*, janvier 1997.

[66] Hochbaum (Dorit S.) et Shmoys (David B.). – A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, vol. 17, n3, juin 1988, pp. 539–551.

[67] Hu (Y. C.), Lu (H.), Cox (A.) et Zwaenepoel (W.). – OpenMP for networks of SMPs. *In : Proc. of the Second Merged Symp. IPPS/SPDP*. – San Juan, Puerto Rico, avril 1999.

[68] Hulbert (L.) et Zmijewski (E.). – Limiting Communication in Parallel Sparse Cholesky Factorization. *SIAM J.Sci.Stat.Comput.*, vol. 12, n 5, septembre 1991, pp. 1184–1197.

[69] Hwang (Jing Jang), Chow (Yuan-Chieh), Anger (Frank D.) et Lee (Chung-Yee). – Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, vol. 18, n2, avril 1989, pp. 244–257.

[70] INRIA. – *Stratagem'96*. – Sophia-Antipolis, France, juillet 1996.

[71] Junaidu (S.), Davie (A.) et Hammond (K.). – Naira: A parallel Haskell compiler. *Lecture Notes in Computer Science*, vol. 1467, 1998, pp. 214–230.

[72] Karp (R. K.), Luby (M.) et auf der Heide (F. Meyer). – Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, vol. 16, n4/5, octobre/novembre 1996, pp. 517–542.

[73] Karypis (G.) et Kumar (V.). – METIS – *Unstructured Graph Partitioning and Sparse Matrix Ordering System, version 2.0*. – Rapport technique, Department of Computer Science, University of Minnesota, Minneapolis, 1995.

[74] Karypis (George) et Kumar (Vipin). – A Fast and High Quality Mutilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, vol. 20, n 1, janvier 1999, pp. 359–392.

[75] Kleiman (Steve), Shah (Devang) et Smaalders (Bart). – *Programming With Threads*. – Mountainview, CA, USA, SunSoft Press, 1995, xxviii and 534p.

[76] 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, décembre 1997.

[77] Kort (Iskander). – *Ordonnacement et Modèle d'Execution : Cas des Graphes Spécifiques*. – France, Thèse de doctorat, Institut National Polytechnique de Grenoble, juillet 1998.

[78] Kruskal (C. P.), Rudolph (L.) et Snir (M.). – A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, vol. 71, n1, mars 1990, pp. 95–132.

[79] Kumar (V.), Grama (A.), Gupta (A.) et Karypis (G.). – *Introduction to Parallel Computing*. – Benjamin/Cummings, 1993.

[80] Lascaux (P.) et Théodor (R.). – *Analyse numérique matricielle appliquée à l'art de l'ingénieur*. – MASSON, 1993.

[81] Lee (C. Y.), Hwang (J. J.), Chow (Y. C.) et Anger (F. D.). – Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, vol. 7, n3, juin 1988, pp. 141–147.

[82] Lenstra (J. K.), Shmoys (D. B.) et Tardos (E.). – Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, vol. 46, 1990, pp. 259–271.

[83] Lipton (R.J.), Rose (D. J.) et E. (Tarjan R.). – Generalized nested dissection. *SIAM Journal on Numerical Analysis*, vol. 16, 1979, pp. 346–358.

[84] Liu (J.W.H.). – The Role of Elimination Trees in Sparse Factorization. *SIAM J.Matrix Anal. Appl.*, vol. 11, n1, January 1990, pp. 134–172.

[85] Liu (J.W.H.). – The Multifrontal Method for Sparse Matrix Solution: Theory and Practice. *SIAM Review*, vol. 34, n1, mars 1992, pp. 82–109.

[86] Liu (Zhen). – A note on Graham's bound. *Information Processing Letters*, vol. 36, n1, octobre 1990, pp. 1–5.

[87] Lu (H.), Hu (Y. C.) et Zwaenepoel (W.). – OpenMP on network of workstations. *In : Proc. of Supercomputing'98*. – Orlando, FL, octobre 1998.

[88] Maggs (B. M.), Matheson (L. R.) et Trajan (R. E.). – Models of parallel computation: A survey and synthesis. *In : Proceeding of the 28th Annual Hawaii International Conference on System Sciences*, éd. par El-Rewini (Hesham) et Shriver (Bruce D.). pp. 61–71. – Los Alamitos, CA, USA, janvier 1995.

[89] Mehaut (Jean-François) et Namyst (Raymond). – *Marcel : Une bibliothèque de processus légers*. – Rapport technique, Laboratoire d'Informatique Fondamentale de Lille, 1995.

[90] Middendorf (M.), Löwe (W.) et Zimmermann (W.). – Scheduling inverse trees under the communication model of the logp-machine. *Theoretical Computer Science*, vol. 125, 1999, pp. 137–168.

[91] Mohr (Eric), Kranz (David A.) et Halstead, Jr. (Robert H.). – Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, n3, juillet 1991, pp. 264–280.

[92] Musser (David R.) et Saini (Atul). – *STL tutorial and Reference Guide*. – Reading, Addison-Wesley, 1996.

[93] Namyst (R.) et Méhaut (J.-F.). – PM^2: Parallel multithreaded machine. A computing environment for distributed architectures. *In : Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, éd. par D'Hollander (E. H.), Joubert (G. R.), Peters (F. J.) et Trystram (D.). pp. 279–285. – Amsterdam, février 1996.

[94] Nikhil (R. S.). – Cid: A parallel, "shared-memory" C for distributed-memory machines. *Lecture Notes in Computer Science*, vol. 892, 1995, pp. 376–390.

[95] Nikhil (R. S.). – Parallel symbolic computing in Cid. *Lecture Notes in Computer Science*, vol. 1068, 1996, pp. 217–242.

[96] Oaks (Scott) et Wong (Henry). – *Java Threads*. – 981 Chestnut Street, Newton, MA 02164, USA, O'Reilly & Associates, Inc., 1999, second édition, 344p.

[97] Papadimitriou (Christos H.) et Yannakakis (Mihalis). – Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, vol. 19, n2, avril 1990, pp. 322–328.

[98] Pellegrini (F.) et Roman (J.). – SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *Lecture Notes in Computer Science*, vol. 1067, 1996, pp. 493–498.

[99] Pellegrini (F.) et Roman (J.). – Sparce matrix ordering with scotch. *In : Proceedings of HPCN'97*. pp. 370–378. – LNCS.

[100] Pellegrini (François), Roman (Jean) et Amestoy (Patrick). – Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *In : Proceeding of IRREGULAR'99*. pp. 986–995. – Puerto Rico, avril 1999.

[101] Plateau (Brigitte) et al. – *Présentation d'*APACHE. – Rapport APACHE n 1, Grenoble, IMAG, octobre 1993.

[102] Pothen (A.) et Sun (C.). – A Mapping Algorithm for Parallel Sparse Cholesky Factorization. *SIAM J.Sci.Comput.*, vol. 14, n5, septembre 1993, pp. 1253–1257.

[103] Prylli (L.) et Tourancheau (B.). – BIP: A new protocol designed for high performance networking on myrinet. *Lecture Notes in Computer Science*, vol. 1388, 1998, pp. 472–480.

[104] Randall (Keith H.). – *Cilk: Efficient Multithreaded Computing*. – Departement of Electrical Engineering and Computer Science, Thèse de doctorat, Massaschussetts Institute of Technology, juillet 1998.

[105] Rapine (Christophe). – *Algorithmes d'approximation garantie pour l'ordonnancement de tâches*. – Thèse, Institut National Polytechnique de Grenoble, janvier 1999.

[106] Rayward-Smith. – UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, no18, 1987, pp. 55–71.

[107] Rinard (Martin C.). – *The Design, Implementation and Evaluation of Jade: A Portable, Implicitly Parallel Programming Language*. – Thèse de doctorat, Stanford University, Computer Systems Laboratory, août 1994, 224p.

[108] Rinard (Martin C.) et Lam (Monica S.). – The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, vol. 20, n3, mai 1998, pp. 483–545.

[109] Robison (Arch D.) et Dubois (Paul F.). – C++ gets faster for scientific computing. *Computers in Physics*, vol. 10, 1996, pp. 458–462.

[110] Roch (J.-L.) et Villard (G.). – *Parallel Computer Algebra*. – Rapport technique, Lecture notes for a tutorial, 1997. ISSAC'97.

[111] Roch (Jean-Louis), Villard (Gilles) et Roucairol (Catherine). – Algorithmes irréguliers et ordonnancement. *In : Parallélisme et applications irrégulières*, éd. par Authié (Gérard) et al., chap. 4, pp. 71–84. – Hermès, 1995.

[112] Rothberg (E.) et Gupta (A.). – An Efficient Block-oriented Approach to Parallel Sparse Cholesky Factorization. *SIAM J.Sci.Comput.*, vol. 15, n6, novembre 1994, pp. 1413–1439.

[113] Rothberg (E.) et Schreiber (R.). – Improved Load Distribution in Parallel Sparse Cholesky Factorization. *In : Supercomputing '94*, pp. 783–792. – Washington, D.C., 1994.

[114] Rothberg (Edward) et Eisenstat (Stanley C.). – Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, vol. 19, n3, juillet 1998, pp. 682–695.

[115] Scales (Daniel J.) et Lam (Monica S.). – The design and evaluation of a shared object system for distributed memory machines. *In : Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI '94)*, pp. 101–114. – Monterey, California, novembre 1994.

[116] Schuster (A.) et Shalev (L.). – Using remote access histories for thread scheduling in distributed shared memory systems. *Lecture Notes in Computer Science*, vol. 1499, 1998, pp. 347–362.

[117] Shmoys (David B.), Wein (Joel) et Williamson (David P.). – Scheduling parallel machines on-line. *SIAM Journal on Computing*, vol. 24, n6, décembre 1995, pp. 1313–1331.

[118] Singh (Jaswinder Pal), Weber (Wolf-Dietrich) et Gupta (Anoop). – SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, vol. 20, n1, mars 1992, pp. 5–44.

[119] Snir (Marc), Otto (Steve W.), Hess-Lederman (S.), Walker (David) et Dongarra (Jack J.). – *MPI: The Complete Reference*. – Cambridge, Mass., MIT Press, 1996.

[120] Stroustrup (B.). – *The C++ Progamming Language*. – Addison-Wesley, sept 1997, 3rd édition.

[121] Trinder (P. W.), Hammond (K.), Mattson, Jr. (S. J.), Partridge (A. S.) et Jones (S. L. Peyton). – GUM : A portable parallel implementation of haskell. *In : Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*. pp. 79–88. – New York, mai 1996.

[122] Valiant (Leslie G.). – A bridging model for parallel computation. *Communications of the ACM*, vol. 33, n8, août 1990, pp. 103–111.

[123] Veldhnizen (Todd L.) et Jernigan (M. Ed). – Will c++ be faster than fortran? *In : Proceedings of the 1st International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'97)*. – California, USA, décembre 1997.

[124] Wilson (Gregory V.) et Lu (Paul) (édité par). – *Parallel Programming Using C++*. – Cambridge, Massachusetts, The MIT Press, 1996.

[125] Wise (David S.). – Representing matrices as quadtrees for parallel processors. *Information Processing Letters*, vol. 20, n4, mai 1985, pp. 195–199.

[126] Yang (Tao) et Fu (Cong). – Space/time-efficient scheduling and execution of parallel irregular computations. *ACM Transactions on Programming Languages and Systems*, vol. 20, n6, novembre 1998, pp. 1195–1222.

[127] Yang (Tao) et Gerasoulis (Apostolos). – DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, n9, septembre 1994, pp. 951–967.

[128] Yang (Tao) et Gerasoulis (Apostolos). – Scheduling of structured and unstructured computation. *In : DIMACS Book Series, 1994 DIMACS Workshop on Interconnections Networks and Mapping and Scheduling Parallel Computation*, éd. par D. Frank Hsu, Arnold Rosenberg (Dominique Sotteau). – American Math. Society.

[129] Yannakakis (M.). – Computing the minimum fill-in is np-complete. *SIAM Journal on Algebraic and Discrete Methods*, vol. 2, n1, mars 1981, pp. 77–79.

Résumé

Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique

Les ordinateurs parallèles offrent une alternative intéressante pour les applications de calcul scientifique, grandes consommatrices de ressources de calcul et de mémoire. Cependant, la programmation efficace de ces machines est souvent difficile et les implantations obtenues sont généralement peu portables. Nous proposons dans cette thèse un modèle de programmation parallèle permettant une programmation simple, portable et efficace des applications parallèles. Ce modèle est basé sur une décomposition explicite de l'application en tâches de calculs qui communiquent entre elles par l'intermédiaire d'objets en mémoire partagée. La sémantique des accès aux données partagées est quasi séquentielle et les précédences entre les tâches sont implicitement définies pour respecter cette sémantique.

Nous présentons dans une première partie la mise en œuvre de ce modèle de programmation dans l'interface applicative C++ Athapascan-1. Une analyse à l'exécution des dépendances de données entre tâches permet d'extraire le flot de données et donc les précédences entre les tâches à exécuter. Des algorithmes d'ordonnancement adaptables à l'application et à la machine cible sont également utilisés. Nous montrons comment, sur architecture distribuée, la connaissance du flot de données entre les tâches peut être utilisée par le système pour réduire les communications et gérer efficacement la mémoire partagée distribuée.

Ce modèle de programmation et sa mise en œuvre dans l'interface applicative Athapascan-1 sont ensuite validés expérimentalement sur différentes architectures et différentes applications d'algèbre linéaire, notamment la factorisation creuse de Cholesky avec partitionnement bidimensionnel. La facilité de programmation de ces applications grâce à cette interface et les résultats obtenus (amélioration des performances par rapport au code de factorisation dense de Cholesky de la bibliothèque ScaLapak sur une machine à 60 processeurs par exemple) confirment l'intérêt du modèle de programmation proposé.

Mots clés : Langages de programmation parallèle, ordonnancement, application irrégulière, factorisation parallèle creuse de Cholesky.

Abstract

Athapascan-1 : towards a parallel programming model adapted to the scientific computation

Parallel computers offer an interesting alternative for the applications of scientific computation, which need large resources of calculation and memory. However, the effective programming of these machines is often difficult and the obtained implementations are generally not easily portable. We propose in this thesis a parallel programming model allowing simple, portable and efficient programming of parallel applications. This model is based on an explicit decomposition of the application into tasks which communicate through objects in a shared memory. The semantics of the access to the shared data is quasi sequential and precedences between the tasks are implicitly defined to respect this semantic.

We present in a first part the implementation of this parallel programming model in the C++ Athapascan-1 interface. An analysis at run-time execution of the dependences of data among tasks makes it possible to extract the data flow and thus the precedences between the tasks to be run. Scheduling algorithms suitable to the application and to the target machine are then used. We also show how, on a distributed architecture, the knowledge of the data flow between the tasks can be used by the system to reduce the communications and to effectively manage the distributed shared memory.

This parallel programming model and its implementation in the Athapascan-1 interface are then validated on various architectures and various applications of linear algebra, in particular the two-dimensional Cholesky factorization. The simple programming of these applications in this interface, and the obtained results (for example, we outperform ScaLapack for dense Cholesky factorization on a machine with 60 processors) confirm the validity of our approach.

Keyword: Parallel programming languages, scheduling, irregular applications, parallel sparse Cholesky factorization.