Ordonnancement de graphe dynamique de tâches sur architecture de grande taille

Rémi Revire

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_/_/_/_/_/_/_/_/_/_/

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : "Informatique : Systèmes et Communications"

préparée au laboratoire Informatique et Distribution
dans le cadre de ***l'Ecole Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique"***

présentée et soutenue publiquement

par

Rémi Revire

le 10 septembre 2004

Titre :

Ordonnancement de graphe dynamique de tâches sur architecture de grande taille. Régulation par dégénération séquentielle et distribuée.

Directeur de thèse : Brigitte Plateau

JURY

M. Roger Mohr , Président
M. Denis Caromel , Rapporteur
M. Thierry Priol , Rapporteur
Mme Brigitte Plateau , Directeur de thèse
M. Jean-Louis Roch , Co-encadrant

Remerciements

Je tiens dans un premier temps à remercier les membres de mon jury qui ont accepté de s'intéresser à mon travail malgré leurs emplois du temps surchargés. Je remercie particulièrement Denis Caromel et Thierry Priol pour avoir pris le temps de lire mon manuscrit et d'avoir grandement contribué à son amélioration par leurs commentaires. Je remercie également Roger Mohr, pour avoir accepté de présider mon jury. Un très grand merci également à Brigitte pour avoir accepté de diriger ma thèse. Merci enfin à Jean-Louis pour tout le temps passé à m'expliquer les fondements théoriques d'Athapascan et pour avoir orienté mon travail tout au long de ma thèse.

Pendant ces quatre années, j'ai eu la chance de travailler avec de nombreuses personnes auxquelles je voudrais adresser des remerciements tout particuliers. Je pense en particulier à Thierry Gautier avec qui j'ai passé de longues soirées à coder et debugger. Sans son aide, je n'aurais jamais pu aller aussi loin dans mes expérimentations. Un grand merci aussi à Van-Dat Cung pour m'avoir fait participer activement au projet Doc-G et pour avoir donné une grande ampleur à mon travail en montant une expérience sur grille de calcul basée sur Athapascan. Merci évidemment à Cyrille et Florence avec qui j'ai pu collaborer, échanger des idées et partager les blues mais aussi les joies de la vie de thésard au laboratoire ID.

Je l'avoue bien volontiers, je n'ai pas fait que travailler pendant ces quatre années. J'ai aussi passé de longs moments à discuter et refaire le monde avec Corine qui a partagé son bureau avec moi pendant presque toute ma thèse. J'ai aussi passer la plupart des pauses déjeuner à me divertir en jouant à la coinche. Il est donc important pour moi de remercier tous mes partenaires et adversaires : Pierre, Pierre-François, Lionel, Philippe, Jean-Marc, Jacques, Bruno, Corine, Christian, Stéphane... Quoiqu'on en dise, la thèse a donc parfois des aspects très séduisants.

Je tenais également à remercier très chaleureusement mes amis qui ont été d'un inestimable soutiens tout au long de ma thèse et qui continuent à l'être pour la suite. Merci donc à Sandrine (pour son soutiens mes aussi pour ses corrections orthographiques de mon manuscrit), Sylvain, Ludine, Céline, Olivier, Elliot, Sonia, Philippe, Arthur, Raphaël, Adeline, François, Magali, Eric et Bastien.

Enfin, je remercie ma famille pour m'avoir permis d'arriver jusque là, pour leur soutiens moral, pour leurs encouragements. Merci donc à mes parents, Lysiane et Patrick mais aussi à ma soeur Cindy et mon frère Romain.

Table des matières

Table des figures

Introduction

Les récentes avancées dans le domaine des architectures parallèles et de leurs exploitations permettent aujourd'hui l'exécution efficace de programmes extrêmement coûteux en calcul et mémoire. De plus en plus, les acteurs de tous les domaines scientifiques utilisent cette puissance de calcul pour simuler des phénomènes qu'ils ne peuvent pas expérimenter réellement, pour résoudre de grands systèmes d'équations ou pour exécuter tout autre type de tâche nécessitant une grande puissance de calcul. Parmi ces applications, on compte de nombreux programmes de simulation de phénomènes physiques ou chimiques comme par exemple la simulation du comportement d'une molécule sous certaines conditions d'expérimentation[38, 16]. Plus récemment des applications plus ludiques comme par exemple la simulation de vêtements couplée à des systèmes de visualisation performants ont fait leur apparition [68, 70, 69, 54]. Enfin, différentes applications d'optimisation combinatoire basées sur des algorithmes de type *Branch and Bound* ou recuit simulé ont été portées sur des architectures de très grande taille [2].

Les architectures parallèles se sont elles aussi diversifiées. On peut aujourd'hui classer ces architectures en trois catégories :

- Architectures symétriques à mémoire partagée, communément appelées SMP (*Symetric Multi-Processors*). Ces machines sont constituées d'un ensemble de processeurs identiques pouvant communiquer par mémoire partagée.
- Architectures à mémoire distribuée, communément appelées "grappes". Une telle architecture est composée d'un ensemble de machines, généralement homogènes, interconnectées par un réseau local de communication comme par exemple Ethernet, Myrinet ou SCI.
- Grilles de calcul. Cette catégorie est la plus récente. Le principe est de fédérer des ressources géographiquement distantes afin d'obtenir une puissance de calcul très importante. Une grille est composée d'un ensemble de machines pouvant appartenir aux trois premières catégories. Ces machines peuvent être interconnectées en utilisant les liens à grande capacité de certains réseaux nationaux (Renater) ou par Internet.

La portabilité des programmes est le principal problème engendré par la diversité des architectures parallèles. En effet, les outils permettant d'implanter une application et d'obtenir de bonnes performances sont souvent différents d'une catégorie à l'autre. Pour la programmation des machines de type SMP, on compte de nombreux outils basés sur la création de processus légers comme par exemple Pthread [35], Java Thread [15], Lazy Threads [29] ou encore OpenMP [24, 33] et Cilk [11, 25]. Pour la programmation des machines à mémoire distribuée, on utilise souvent des

bibliothèques d'échange de messages comme MPI[23, 31] ou PVM [28] par exemple. Ces deux outils permettent la création explicite de processus communiquant par échange de messages. Cependant, d'autres environnements de plus haut niveau basés sur la création de tâches concurrentes sont aussi disponibles comme par exemple Athapascan [58, 27], Jade [56, 55], Orca [5] ou encore SMART [49]. Enfin, d'autres outils comme Charm++ [36, 37] ou ProActive [19, 4, 18], basés sur les concepts de la programmation objets permettent aussi l'exploitation de grappes et grilles de calcul.

Quelque soit l'outil utilisé, l'algorithme parallèle implanté dans l'application reste invariant. L'obtention de bonnes performances sur l'une ou l'autre des architectures dépend alors principalement de l'ordonnancement des différentes tâches de calcul du programme. De manière générale, l'ordonnancement consiste à placer les différents objets du programmes (tâches et données) sur les différentes ressources (processeurs et mémoires) disponibles. L'objectif des algorithmes d'ordonnancement existants est généralement d'optimiser le temps d'exécution parallèle des applications. Des contraintes supplémentaires peuvent cependant être prises en compte pour optimiser l'occupation mémoire du programme par exemple.

Indépendamment de la stratégie choisie, la réalisation de l'ordonnancement reste d'une importance cruciale quelque soit le grain de l'application. Considérons dans un premier temps une application à grain fin. Si pour chaque tâche, un temps important est consommé pour lui choisir un site d'exécution, il y a peu d'espoir d'obtenir une exécution parallèle efficace. Dans le cas d'une application à gros grain que l'on destine à être exécutée sur une architecture de grande taille de type grille de calcul, une implantation centralisée de l'ordonnancement limite le passage à l'échelle de l'application.

L'objectif de cette thèse est d'étudier les mécanismes permettant l'implantation efficace d'algorithmes d'ordonnancement. Ces mécanismes sont basés sur le couplage des mécanismes de dégénération séquentielle et distribuée [57] :

- **Dégénération séquentielle**, aussi appelé *Work First Principle*. Le principe de ce mécanisme est basé sur le fait qu'une application est généralement composée d'un nombre de tâches très largement supérieur au nombre de processeurs. De nombreux groupes de tâches sont alors exécutés séquentiellement sur un même processeur sans nécessiter la communication de données distantes. L'idée est donc d'optimiser le coût de création d'une tâche pour le cas où le parallélisme n'est pas nécessaire, et de tolérer un surcoût plus important lorsqu'on veut générer du parallélisme. La dégénération séquentielle consiste donc à générer, pour ces groupes de tâches une exécution aussi proche que possible de celle du programme équivalent écrit dans un langage séquentiel.
- **Dégénération distribuée**. Ce mécanisme consiste à générer de manière automatique une exécution parallèle aussi proche que possible de celle du programme équivalent écrit avec une bibliothèque d'échanges de messages (envois/réceptions) comme par exemple MPI. L'objectif est donc de minimiser la

quantité de messages échangés en générant un maximum de communications unidirectionnelles.

Notre travail est centré sur des algorithmes distribués permettant d'obtenir un meilleur passage à l'échelle que les algorithmes centralisés. Ces mécanismes sont implantés et testés dans l'environnement de programmation Athapascan [58, 27].

L'organisation de ce document est la suivante :

Dans un premier temps, nous étudierons les différentes possibilités offertes pour l'ordonnancement dans différents environnements de programmation parallèle. Nous étudierons des environnements proposant des stratégies d'ordonnancement statiques ou dynamiques et nous mettrons l'accent sur les environnements implantant des mécanismes de dégénération séquentielle et ceux proposant des algorithmes d'ordonnancement distribués passant à l'échelle.

Le deuxième chapitre présente en détail l'environnement de programmation parallèle Athapascan sur lequel ont été effectuées les expérimentations. Nous effectuerons un bref historique des différentes implantations réalisées en détaillant pour chacune d'elles les stratégies d'ordonnancement proposées. Enfin nous effectuerons un bilan récapitulatif de l'environnement Athapascan et des autres environnements étudiés dans le premier chapitre.

L'analyse des deux premiers chapitres montrant les difficultés pour implanter efficacement les techniques de dégénération séquentielle sur architecture à mémoire distribuée, nous proposons dans un troisième chapitre, une étude en termes de défauts de page de plusieurs protocoles de cohérence mémoire existants pour l'implantation d'une couche de mémoire partagée distribuée. En particulier, un protocole de cohérence plus restrictif basé sur la construction dynamique du flot de données de l'application sera détaillé. Par une analyse théorique de ce protocole, nous montrons que peu de défauts de pages interviennent lors d'une exécution parallèle si peu de tâches sont exportées (c'est-à-dire exécutées sur un processeur différent du processeur d'exécution de la tâche qui l'a créée).

Le quatrième chapitre est ensuite consacré à la présentation d'un mécanisme de pile pour l'implantation du protocole de cohérence flot de données. Ce protocole est basé sur le couplage des principes de dégénération séquentielle et distribuée. Le mécanisme de pile permet d'optimiser le coût de création d'une tâche (dégénération séquentielle) et la gestion distribuée du flot de données permet une implantation distribuée passant à l'échelle (dégénération distribuée).

Deux algorithmes d'ordonnancement permettant de tirer partie des mécanismes présentés dans le quatrième chapitre sont ensuite présentés. Le premier est un algorithme d'ordonnancement dynamique basé sur le vol de tâches sur inactivité. Nous montrerons que cet algorithme permet de borner le nombre de vols qui interviennent lors d'une exécution parallèle. Le deuxième est un algorithme statique dont l'objectif est de calculer un regroupement des données dans les différents modules de mémoire disponibles avant de placer les tâches en fonction des données auxquelles elles accèdent. Nous proposons d'effectuer ce regroupement en construisant un graphe de dépendance de données et d'utiliser une bibliothèque de partitionnement (Scotch[50]

ou Metis [39]) pour calculer le regroupement.

Le sixième chapitre sera consacré à l'implantation de ces mécanismes dans l'environnement Athapascan et à sa validation sur des tests simples. En particulier, nous détaillerons l'implantation des accès différés du langage, basée sur l'utilisation des I-structures [3]. Nous présenterons ensuite l'algorithme de détection de terminaison distribuée que nous avons implanté dans Athapascan. Cet algorithme est inspiré de l'algorithme de Rana décrit dans [63].

Dans un septième chapitre cette implantation sera validée pour des applications de taille plus importante. Nous étudierons, dans un premier temps, les résultats obtenus pour l'application Sappe [68, 70]. Cette application permet de simuler en temps réel le comportement de tissus comme par exemple un mouchoir sur un étendage ou les vêtements d'un personne. L'application Taktuk [54, 6] sera présentée dans un deuxième temps. Cette application est un code de dynamique moléculaire permettant de simuler le comportement d'un ensemble de molécules dans des conditions difficilement réalisables en laboratoire. Enfin nous présenterons les résultats obtenus pour une application d'optimisation combinatoire dont la parallélisation a été étudiée dans le cadre de l'ACI Doc-G (Défit en Optimisation Combinatoire sur Grilles). Nous décrirons comment cette application a été parallélisée en Athapascan ainsi que les performances obtenues sur grille.

En conclusions, nous présenterons les perspectives de ces travaux.

Première partie

Gestion de l'ordonnancement de tâches dans les environnements de programmation parallèle

CHAPITRE 1

Ordonnancement dans les environnements de programmation parallèle

La distribution des tâches de calcul sur les différents processeurs disponibles est un des points clés pour obtenir des exécutions parallèles efficaces. Ce chapitre est consacré à une étude des stratégies d'ordonnancement proposées par quelques environnements de programmation parallèle représentatifs. Ces stratégies sont dites *statiques* ou *dynamiques* en fonction du moment où la décision est prise et quelques environnements proposent des stratégies hybrides.

Pour appliquer une stratégie d'ordonnancement statique, on suppose qu'il est possible d'obtenir une description des calculs à effectuer avant le début de leur exécution. Cette description est souvent donnée sous la forme d'un graphe de tâches (graphe de précédence, graphe de dépendance ou graphe de flot de données). L'application d'une stratégie statique consiste alors à exécuter un algorithme qui désignera un processeur d'exécution pour chacune des tâches du graphe. Parmi les stratégies implantées dans des environnements de programmation parallèle, on peut noter les stratégies ETF (Earliest Task First) [44] et ERT (Earliest Ready First) [34] dont une variante a été implantée dans Athapascan [21], la stratégie DSC (Dominent Sequence Clustering) implantée dans Pyrros [66, 67] ou des stratégies de partitionnement récursif implantées dans Scotch [50] ou Metis [39].

Dans le cas où il n'est pas possible d'avoir une description complète des calculs à effectuer, des stratégies d'ordonnancement dynamique sont utilisées. Les décisions d'ordonnancement sont alors prises à des moments clés de l'exécution du programme comme par exemple la création, la terminaison d'une tâche ou l'inactivité d'un processeur. Ces stratégies sont implantées dans de nombreux environnements comme par exemple Cilk [11, 25], SilkRoad [51], Satin [65, 64], Jade [56, 55], OVM [14], Orca [5], Chime [61] ou Athapascan [27].

Quelle que soit la stratégie d'ordonnancement utilisée (statique ou dynamique), la réalisation de cet ordonnancement est d'une importance cruciale. En particulier, nous pouvons remarquer que l'écriture intuitive d'un algorithme parallèle génère souvent beaucoup plus de tâches et beaucoup plus de parallélisme que la machine cible ne peut en exploiter. Il est alors fréquent que des groupes de tâches de tailles importantes soient exécutés sur un unique processeur sans avoir besoin de données produites à distance. Dans ce cas, la création d'une tâche aurait pu être remplacée par un simple appel séquentiel de fonction. Il est cependant très difficile, voir impossible de décider a priori des tâches pouvant être dégénérées en appels séquentiels de fonction puisque cette décision dépend du contexte d'exécution et en particulier de la charge des

processeurs,...).

Dans ce chapitre, nous détaillerons dans un premier temps les différentes représentations des programmes parallèles avant de proposer une classification des modèles de synchronisation sur lesquels sont basés les environnements que nous présentons. Les stratégies statiques proposées dans les environnements Pyrros, Scotch et Metis seront ensuite étudiées, avant de présenter les stratégies dynamiques basées sur le principe de "dégénération séquentielle" à grain fin implantées dans les environnements Cilk et les processus légers paresseux (Lazy Threads), ou encore ProActive[19, 4, 18]. Nous nous intéresserons enfin aux stratégies d'ordonnancement dynamiques par dégénération distribuée. Nous verrons tout d'abord les stratégies centralisées utilisées dans Jade et OVM avant de détailler les stratégies hiérarchiques utilisées dans Satin et Charm++.

1.1 Représentation de l'exécution et classification des modèles de synchronisation

Qu'elles soient statiques ou dynamiques, les stratégies d'ordonnancement que nous allons présenter dans ce chapitre sont basées sur une représentation des programmes sous forme de graphe de tâches. Selon les spécificités de l'ordonnancement utilisé, ce graphe peut être réduit à un graphe de dépendances, un graphe de précédences ou un graphe de flot de données. Indépendamment de la représentation choisie, le graphe de tâche peut être contraint par l'environnement de programmation en fonction du modèle de synchronisation sur lequel il est basé. Nous détaillerons dans cette section les différentes représentations de l'exécution des programmes qui seront utilisées dans la suite avant de proposer une classification des modèles de synchronisation utilisés dans les environnements de programmation parallèle.

1.1.1 Représentation de l'exécution de programmes parallèles

Trois grandes familles de graphes de tâches sont fréquemment utilisées pour représenter l'exécution des programmes parallèles : graphe de dépendances, graphe de précédences et graphe de flot de données (figure 1.1). Un graphe de dépendances est un graphe non orienté dans lequel un noeud du graphe représente une tâche et une arête représente une dépendance entre deux tâches. Ce type de graphe est souvent utilisé pour le calcul d'un placement statique [50, 39]. Cependant pour les programmes parallèles dynamiques comme par exemple les programmes récursifs ou les programmes contenant des boucles parallèles dont on ne connaît pas les bornes avant l'exécution, il est indispensable de disposer d'une représentation dynamique de l'exécution des programmes.

Dans [47, 12, 9], les programmes parallèles sont représentés par des graphes de précédence orientés sans circuit, construits dynamiquement au cours de l'exécution. Ce type de représentation ne correspond donc plus à un programme particulier, mais à une exécution particulière dépendante des paramètres du programme. Dans cette

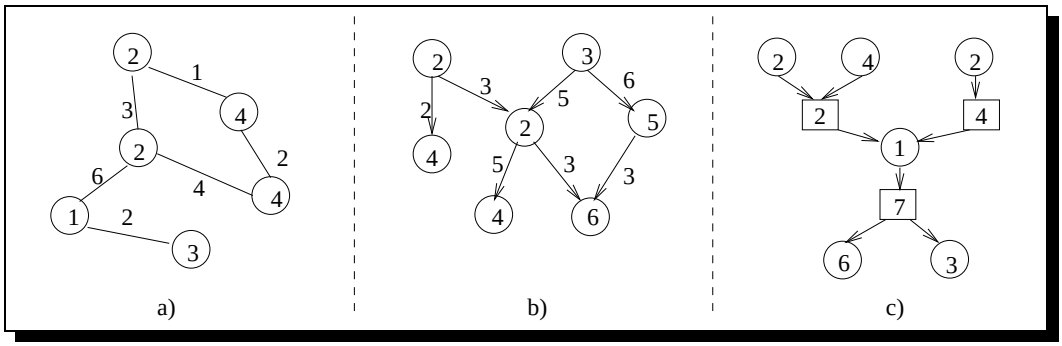


FIG. 1.1 – Représentation de l'exécution des programmes sous forme de graphe de dépendances (a), graphe de précédence (b) et graphe de flot de données (c).

représentation, les nœuds du graphe représentent les tâches du programme et les arêtes représentent une dépendance entre deux tâches. Lorsqu'une tâche t_m crée une tâche t_f, t_m est appelée tâche mère et t_f tâche fille. Dans la suite, cette dépendance "mère-fille" est représentée par une flèche en pointillés, les autres dépendances sont représentées par des flèches continues. A chaque nœud du graphe peut être associé un poids représentant le nombre d'instructions de la tâche, la quantité de mémoire allouée (ou désallouée) ou toute autre information utile à l'ordonnancement.

D'autre part, une représentation dynamique de l'exécution basée sur le flot de données de l'application a été proposée [27]. Dans cette représentation, un graphe bipartie orienté sans circuit construit au cours de l'exécution est utilisé. Les noeuds du graphe de flot de données peuvent représenter les tâches ou les données du programme. Une arête entre un noeud "tâche" et un noeud "donnée" représente la production de la donnée par la tâche et une arête entre un noeud "donnée" et un noeud "tâche" représente la consommation de la donnée par la tâche. A chaque noeud du graphe peuvent être associées des informations en fonction des besoins de l'ordonnancement.

1.1.2 Classification des modèles de programmes parallèles

De nombreux modèles pour la programmation parallèle ont été proposés et implantés dans les langages existants. Le modèle de programmation associé à un langage détermine la manière dont les tâches (ou processus légers) sont créées et quels types de synchronisation sont autorisés. Chacun de ces modèles impose un certain nombre de restrictions sur le type de synchronisation entre tâches. Dans [47] une classification en cinq catégories est proposée. Celle-ci est basée sur la création dynamique de tâches (contrairement à MPI pour lequel un processus est créé sur chaque noeud au démarrage de l'application). La figure 1.2 permet d'illustrer chaque classe avec un graphe de précédence caractéristique.

Modèle plat : Dans ce modèle (figure 1.2 a)), seul le processus principal (ra-

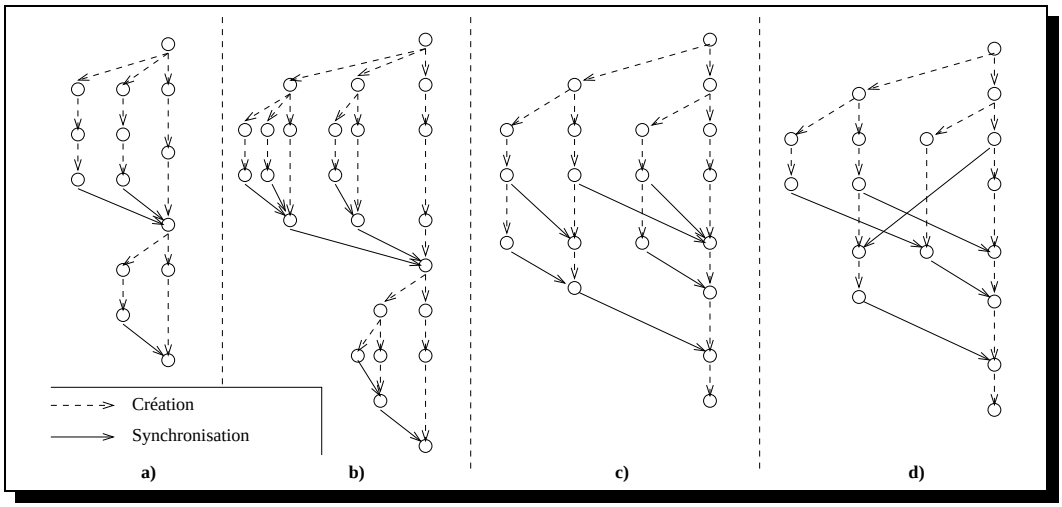


FIG. 1.2 – Exemple de graphes orientés sans cycle représentant les différents modèles de programmes. La partie a) illustre le modèle plat, la partie b) le modèle imbriqué, la partie c) le modèle strict et la partie d) le modèle déterministe non contraint.

cine) est autorisé à créer des processus. Chaque processus ne peut se synchroniser avec le processus principal qu'en un point unique du programme avant de terminer son exécution. Les processus créés par le processus principal ne peuvent pas communiquer entre eux. Après chaque point de synchronisation, le processus principal peut créer de nouveaux processus. Ce modèle est utilisé par exemple dans *High Performance Fortran* [32].

Modèle imbriqué : De la même manière que pour le modèle plat, chaque processus "fils" peut se synchroniser avec son processus "père" avant de terminer son exécution. Aucune autre synchronisation n'est permise et les processus "fils" ne peuvent pas non plus communiquer entre eux. Cependant, les processus "fils" peuvent créer de nouveau processus (figure 1.2 b)). Ce modèle est utilisé par exemple dans le langage NESL [8] ou dans Cilk [11].

Modèle strict : Ce modèle (figure 1.2 c)) défini dans [9] étend le modèle imbriqué en autorisant la définition de plusieurs points de synchronisation entre un processus et n'importe lequel de ses ancêtres. Le modèle strict réduit [9] est un sous-modèle du modèle strict pour lequel plusieurs points de synchronisation sont autorisés mais avec son processus père uniquement. Ce sous-modèle est équivalent au modèle imbriqué.

Modèle de synchronisation déterministe non contraint : Ce modèle de programmes (figure 1.2 d)) n'impose aucune restriction sur le schéma de synchronisation entre tâches. Chaque processus peut en effet se synchroniser avec n'importe quel autre processus de l'application. Ce type de synchronisation est souvent implanté en utilisant des "variables de synchronisation". Une variable

de synchronisation est une variable à affectation unique. Chaque tâche voulant lire cette variable est suspendue jusqu'à ce que celle-ci ait été écrite. Après avoir été écrite, une variable de synchronisation peut être lue par plusieurs tâches. Ce type de variable de synchronisation est utilisé dans les I-structures du langage Id [3], ou dans l'implantation des *streams* de Sisal [22].

Modèle non déterministe : Les autres modèles de synchronisation basés par exemple sur des verrous, des sémaphores ou des variables de condition ont la propriété d'engendrer des schémas de synchronisation non déterministes. Prenons l'exemple de deux processus exécutés en concurrence jusqu'à ce que chacun d'entre eux veuille entrer en section critique en utilisant une variable de type verrou. La dépendance entre les deux processus dépendra alors de l'ordre dans lequel les deux processus auront acquis le verrou. Dans ce cas, le graphe de dépendances ne dépend pas seulement du programme et de ses paramètres, mais aussi du contexte d'exécution. Des environnements de processus légers comme par exemple les Threads POSIX [35] proposent ce type de primitives de synchronisation.

1.2 Stratégies d'ordonnancement statique

Dans cette section, trois outils permettant de calculer statiquement le site d'exécution de chaque tâche d'un programme sont présentés. Le premier, Pyrros, utilise une heuristique basée sur l'analyse du chemin critique. Les deux autres Scotch et Metis sont basés sur des techniques de bipartitionnement de graphe.

1.2.1 Pyrros

Le projet Pyrros [66] a été conjointement conduit par l'université Rutgers (Piscataway, USA) et l'université de Californie à Santa Barbara (UCSB) entre les années 1990 et 1995. Son objectif était d'étudier les algorithmes d'ordonnancement et les systèmes de génération de code pour l'exécution de graphes de tâches sur des architectures pour lesquelles les communications s'effectuent par échange de messages.

Description du parallélisme

Dans l'environnement de programmation parallèle Pyrros, le parallélisme est décrit explicitement par l'utilisateur à l'aide d'une interface de description de graphe de tâches avec précédence. Les graphes ainsi construits sont définis par le tuple $G = (V, E, C, T)$ dans lequel V représente l'ensemble des tâches (sommets du graphe), E représente l'ensemble des dépendances de données entre les tâches (arêtes du graphe), C représente les coûts de communications associés à chaque arête et T représente les coûts de calculs associés à chaque sommet (figure **??**). Les programmes Pyrros appartiennent donc au modèle de synchronisation déterministe non contraint. Dans la suite on note, $\mid V \mid = v$ et $\mid E \mid = e$.

Algorithme d'ordonnancement utilisé

L'ordonnancement dans Pyrros est basé sur une stratégie en deux étapes.

1. Phase de regroupement :

   En supposant qu'une infinité de processeurs sont disponibles, calcul des groupes de tâches à exécuter sur un processeur.

2. Phase d'ordonnancement de chacun des groupes sur les p processeurs de la machine cible.

La phase de regroupement utilise l'algorithme DSC (Dominant Sequence Clustering) proposé par Gerasoulis et Yang [67]. Initialement chaque groupe n'est composé que d'une seule tâche. L'algorithme consiste ensuite à fusionner deux groupes communiquant ce qui correspond à les placer sur le même processeur. Une fusion permet d'annuler le surcoût de communication entre les deux groupes choisis et ainsi de réduire le temps d'exécution parallèle. L'idée importante de l'algorithme DSC est de fusionner à chaque étape deux groupes de la séquence dominante, c'est-à-dire du plus long chemin du graphe partiellement regroupé. Ce schéma est itéré jusqu'à ce qu'une nouvelle fusion ne puisse plus améliorer la durée totale du regroupement obtenu.

Pour calculer l'ordonnancement sur p processeurs, les groupes de tâches sont dans un premier temps triés dans l'ordre croissant de leur coût de calcul. Un algorithme d'équilibrage de charge est ensuite utilisé pour répartir les groupes de manière à ce que chaque processeur ait sensiblement la même charge.

Exécution selon l'ordonnancement

Afin d'exécuter le graphe de tâches sur une architecture à mémoire distribuée, un nouveau code est généré dans lequel des primitives d'envoi non bloquantes et de réception bloquantes ont été insérées. Chaque tâche ayant besoin d'une ou de plusieurs données produites à distance sera précédée des requêtes de réception associées. Chaque tâche produisant des données qui doivent être utilisées à distance sera suivie par les requêtes d'envoi associées. L'exécution et l'activation des tâches est ensuite dirigée par l'avancement du flot de données. Cette technique a été validée sur les algorithmes BLAS de niveau 3 ; les expériences ont montré que le code généré avait des performances très proches des codes directement écrits en MPI.

1.2.2 Scotch et Metis

Scotch [50] et Metis [39] sont deux bibliothèques de fonctions permettant de partitionner des graphes de tâches non orientés en utilisant des méthodes de partitionnement multi-niveaux. La bibliothèque Scotch a été développée au laboratoire Labri de l'université Bordeaux I et la dernière version a été produite en 2001. La bibliothèque Metis a été développée à l'université du Minnesota à Minneapolis et la dernière version a été produite en 2003.

Description du parallélisme

Dans chacune des deux bibliothèques, le parallélisme est décrit sous la forme d'un graphe de tâches non orienté (graphe de dépendances). Les arêtes entre deux tâches représentent les données échangées par celles-ci. Les tâches peuvent être annotées par leur coût de calcul et les arêtes par le volume de données échangées qu'elles représentent. La description du graphe est effectuée par l'utilisateur en construisant une structure de données composée d'entiers et de tableaux d'entiers permettant d'identifier chaque sommet, chaque arête ainsi que les coûts associés. Comme pour Pyrros, cette interface permet de décrire des programmes appartenant au modèle de synchronisation déterministe non contraint.

Algorithmes d'ordonnancement utilisés

L'ordonnancement calculé par la bibliothèque Metis considère que la machine cible est composée de processeurs homogènes et d'un réseau complétement maillé. Celui-ci est basé sur un algorithme de partitionnement multi-niveaux en trois étapes (figure 1.4).

1. Phase de regroupement : l'idée est ici de réduire la taille du graphe de tâches afin que l'étape de partitionnement puisse être effectuée de manière efficace. Le principe est de regrouper les tâches travaillant sur les mêmes ensembles de données c'est à dire les tâches reliées par des arêtes de poids important. Ce principe est appliqué de manière répétitive jusqu'à ce que le graphe ait atteint une taille pour laquelle un algorithme de partitionnement pourra être utilisé efficacement (*coarsing phase*).

2. Un partitionnement du graphe réduit est calculé (*partitionning phase*). Plusieurs algorithmes peuvent être utilisés pour cela comme par exemple celui de Fiduccia-Matteyses.

3. Le partitionnement est raffiné en projetant pas à pas le graphe partitionné vers le graphe initial (*uncoarsing phase*).

La bibliothèque Scotch permet de calculer un ordonnancement en considérant que la machine cible est composée de processeurs hétérogènes reliés par des liens de capacités différentes. La machine cible est modélisée par un graphe non orienté, annoté par la puissance de calcul des processeurs et la capacité des liens les reliant. Pour cela un algorithme de bipartitionnement récursif est utilisé. A chaque étape, des bipartitions du graphe de tâches et du graphe de processeurs sont calculées afin d'allouer un sous ensemble de tâches à un sous ensemble de processeurs de la machine cible. Cet algorithme est ensuite appliqué récursivement à chaque couple de partitions. Plusieurs stratégies sont disponibles pour calculer les bipartitions dont une variante de l'algorithme de partitionnement multi-niveau utilisé par Metis.

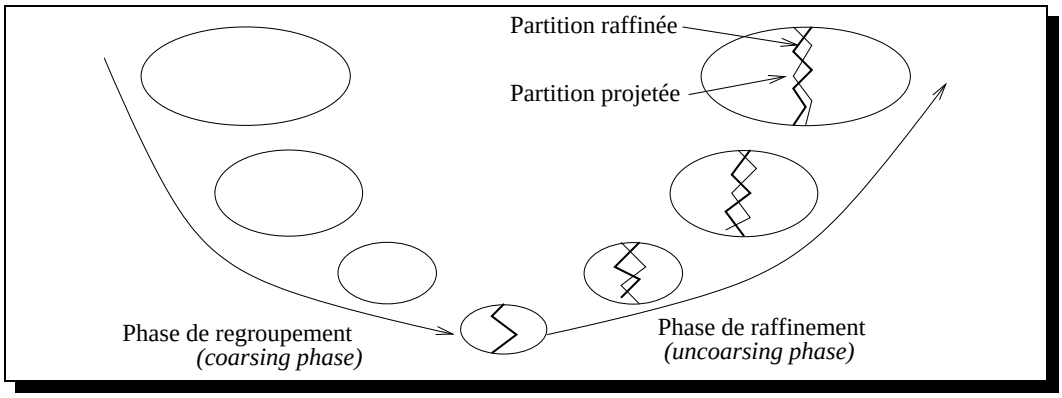


FIG. 1.4 – Fonctionnement de l'algorithme de partitionnement multi-niveaux.

Exécution selon l'ordonnancement

Aucune des deux bibliothèques ne permet d'exécuter une application selon l'ordonnancement qu'elles ont calculé. Cependant, leur utilisation simple et intuitive leur permet d'être intégrées à d'autres environnements de programmation parallèle offrant un support exécutif. Ces deux bibliothèques ont été validées sur une grande variété de graphes parmi lesquels des hypercubes ou des maillages en trois dimensions correspondant à des codes d'éléments finis.

1.3 Ordonnancement dynamique à grain fin, dégénération séquentielle

Pour les applications parallèles qui visent l'exploitation d'un nombre de noeuds potentiellement grand, le nombre de tâches pouvant s'exécuter en parallèle peut s'avérer, dans la pratique, très largement supérieur au nombre de processeurs effectivement disponibles. Dans ce cas, de nombreuses tâches sont placées sur un même noeud sur lequel elle sont exécutées séquentiellement. Peu d'entre elles dépendent d'autres tâches exécutées sur des processeurs différents. L'un des points clé pour une exécution parallèle efficace est donc de maîtriser la gestion du parallélisme en dégénérant séquentiellement l'exécution des blocs de tâches ordonnancées sur le même processeur.

Dans de nombreux environnements de programmation à base de processus légers (Lazy Threads [29], Cilk [25], Thread POSIX [35]) des mécanismes de "pile cactus" sont utilisés. Lorsqu'un seul processus léger s'exécute, le comportement de la pile est très proche de celui des langages de programmation standard (C, Java...). Lorsqu'une fonction est appelée, une structure de données appelée *bloc d'activation* permet de stocker les variables locales à la fonction. Cette structure est empilée lors de l'appel. La pile devient une pile cactus lorsqu'un nouveau processus léger est créé, chaque

processus empilant de nouveaux *blocs d'activation* sur sa propre branche de la pile.

D'autres environnements comme par exemple ProActive ont une approche différente de la dégénération séquentielle. Dans ces environnements, basés sur un modèle de programmation "objet", chaque appel de méthode peut être exécuté de manière standard ou être transformé en appel de méthode distant.

Dans Cilk ou Lazy Threads, les fonctions "a priori" parallèles sont donc dégénérées en appels séquentiels de fonctions, alors que dans ProActive, les appels "à priori" séquentiels de méthodes peuvent être transformés en appels distants s'ils sont effectués sur des objets présent sur des noeuds de calcul distants.

Dans cette section, nous détaillerons l'implantation basée sur une pile cactus du langage Cilk et des processus légers *Lazy Threads*. Ces deux implantations ont la particularité de fonctionner sur mémoire partagée. Certains projets comme par exemple SilkRoad [51] ou Chime [61] ont étudié la possibilié de porter ce type de mécanisme sur architecture à mémoire distribuée en utilisant une couche de mémoire virtuelle partagée distribuée. Dans les deux cas, ces implantations souffraient de faiblesses en terme de passage à l'échelle des couches de mémoire partagée utilisées. Nous étudierons ensuite l'approche "objet" de la dégénération séquentielle proposée dans ProActive.

1.3.1 Cilk

Cilk [11, 25] est un langage de programmation parallèle basé sur le langage C destiné aux machines à mémoire partagée. Il est développé au MIT (Massachusetts Institute of Technology) depuis 1993. Les dernières modifications ont été apportées en 2000.

Description du parallélisme.

Dans ce langage, le parallélisme et les synchronisations sont décrites par l'utilisateur à l'aide de trois mots clés : *cilk*, *spawn* et *sync*. Du parallélisme est généré lorsque l'invocation d'une procédure est précédée du mot clé *spawn*. La sémantique d'un *spawn* est différente de celle d'un appel de procédure C standard puisque la procédure mère peut continuer son exécution en parallèle avec la procédure fille. Pour qu'une procédure puisse être lancée en parallèle, sa déclaration doit être précédée du mot clé *cilk*.

Une procédure Cilk ne peut utiliser les valeurs retournées pas ses filles qu'après avoir exécuté la primitive *sync*. Cette primitive agit comme une barrière et permet d'attendre la fin de l'exécution des procédures filles lancées en parallèle.

Les programmes Cilk appartiennent au modèle de synchronisation strict réduit (ou imbriqué).

Stratégie d'ordonnancement utilisée.

La stratégie d'ordonnancement du langage Cilk est basée sur un mécanisme de vol de travail. Au cours de l'exécution d'un programme, lorsqu'un processeur devient

inactif, il passe en mode *voleur* et demande du travail à un processeur *victime* choisi aléatoirement.

Un modèle de coût est associé à l'exécution des programmes Cilk. Chaque programme est caractérisé par :

- T_1 qui représente le travail total de l'application, c'est-à-dire le nombre total d'instructions effectuées lors de l'exécution du programme. T_1 correspond donc au temps d'exécution séquentiel.
- T_∞ qui représente le travail sur le plus long chemin du graphe de précédence de l'application. T_∞ correspond au temps d'exécution sur une infinité de processeurs.
- S_1 qui représente la consommation mémoire nécessaire à une exécution séquentielle en profondeur d'abord.

L'implantation de l'algorithme de vol de travail utilisé garantit que le temps d'exécution d'un programme sur p processeurs est tel que : $T_p = \frac{T1}{p} + O(T_\infty)$ et la consommation mémoire sur p processeurs est telle que : $S_p \leq pS_1$.

Réalisation efficace de l'ordonnancement

Il est possible de transformer un programme Cilk en programme C standard en supprimant simplement les mots clés *cilk*, *spawn* et *sync* du code utilisateur. Le code résultat est syntaxiquement correct et a la même sémantique que le programme Cilk ; ce code est nommé la C-élision du programme initial.

L'implantation de l'algorithme de vol de travail est basé sur le principe "travail d'abord" *(work-first principle)* selon lequel le surcoût de l'ordonnancement lié au travail total T_1 doit être optimisé alors qu'un surcoût plus important peut être toléré sur le chemin critique T_∞. Ceci est justifié pour les applications cibles du langage Cilk (modèle de synchronisation strict réduit), pour lesquelles le nombre de vol est borné par T_∞ qui est très petit devant T_1.

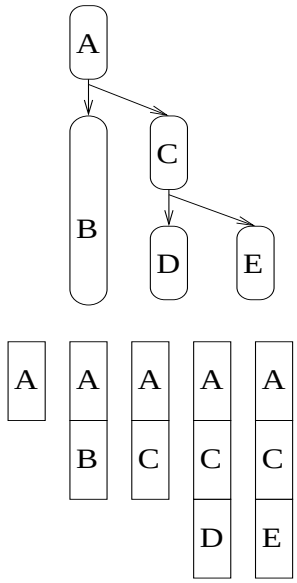
Le compilateur Cilk implante ce principe en générant deux versions pour chaque procédure cilk : une version "lente" et une version "rapide". La version rapide est très proche de la version C-élision de la procédure et n'engendre qu'un faible surcoût par rapport à une exécution séquentielle standard. Cette version est exécutée lorsque les processus sont actifs. Dans ce cas, la génération de plus de parallélisme n'est pas nécessaire. La version lente est exécutée lorsqu'il est nécessaire de générer du parallélisme, c'est à dire après une requête de travail.

Afin de permettre l'implantation de l'algorithme de vol de travail, un bloc d'activation *(frame)* est associé à chaque appel de procédure Cilk (quelque soit la version utilisée). Les blocs d'activation permettent de stocker les variables locales à la procédure ainsi que son état. L'ensemble des blocs d'activation est géré à l'aide d'une pile dont le fonctionnement est très proche de pile C standard lorsque les procédures sont exécutées séquentiellement. Lorsqu'un processeur est victime d'une requête de vol, si sa pile n'est pas vide, le bloc d'activation placé à son sommet (donc le plus ancien) est copié vers la pile du processeur voleur. La pile devient alors une "pile cactus".

La pile cactus de Cilk est implantée en chaînant entre eux les blocs d'activation alloués dans le tas (dans le langage C) (figure 1.5).

```
1  cilk void A() {
2    spawn B();
3    spawn C();
4  }
5
6  cilk void B() {...}
7
8  cilk void C() {
9    spawn D();
10   spawn E();
11 }
12
13 cilk void D() {...}
14 cilk void E() {...}
```

a()



b()

FIG. 1.5 – Un exemple de programme Cilk (a) et la pile associée (b)

Implantations distribuées

Plusieurs tentatives de portage du langage Cilk sur architecture à mémoire distribuée ont été effectuées. La première "Cilk-NOW" [13] implante la version 2 du langage de manière distribuée et tolérante aux fautes. Une implantation plus récente appelée "Distributed Cilk" [53] porte la version 5.1 du langage sur ce type d'architecture. Pour cela, une couche de mémoire partagée, distribuée, paginée et basée sur un protocole de cohérence adapté aux spécificités du langage est utilisée. Enfin le projet "SilkRoad" [51] a comparé plusieurs couches de mémoire partagées en utilisant différents protocoles de cohérence afin d'améliorer les performances de "Distributed Cilk". Pour ces trois tentatives, seules des grappes ayant au maximum dix processeurs ont été exploitées de manière efficace [13, 53, 51].

1.3.2 Les processus légers paresseux : *Lazy Threads*

Dans [29], les auteurs présentent une implantation paresseuse de processus légers. Dans cette implantation la création d'un nouveau processus léger est en fait une création "potentielle" de processus léger (de même que le `spawn` du langage Cilk). Chaque création potentielle peut être transformée en création appel séquentiel de fonction ou en création de tâche parallèle. Ce choix est effectué en fonction d'évènements comme

l'inactivité d'un processeur. Ces travaux ont été menés à l'université de Berkeley en Californie entre 1995 et 1997.

Description du parallélisme

Les travaux présentés dans [29] sont destinés à l'implantation de processus légers paresseux sans restriction par rapport aux autres bibliothèques de processus légers. Le parallélisme est donc généré en créant un nouveau processus léger (un nouveau flot d'exécution) afin d'exécuter une procédure de manière asynchrone. Les communications entre processus légers sont effectuées grâce à l'utilisation d'une mémoire partagée. Des instructions (mutex, conditions...) permettent de gérer les accès concurrents à la mémoire partagée. Un processus peut donc être dans les états suivants :

- En cours d'exécution
- Suspendu
- Prêt à être exécuté
- Terminé

Le modèle de synchronisation des processus légers paresseux fait partie du modèle non déterministe proposée dans la section 1.1.

Stratégie d'ordonnancement

Comme dans le langage Cilk, l'algorithme d'ordonnancement utilisé est basé sur l'envoi de requêtes de vol sur inactivité. S'il n'est pas nécessaire de créer plus de parallélisme (cas le plus fréquent), lors de la création d'un nouveau processus, le flot d'exécution est directement transmis du processus père au processus fils. A la fin de l'exécution du processus fils, si la continuation du processus père n'a pas été volée, le flot d'exécution ainsi que les valeurs retournées lui sont retransmis de la même manière que pour un appel séquentiel de fonction.

Lorsqu'une requête de vol intervient, le processeur voleur peut :

- voler un processus préalablement suspendu, mais dont l'exécution peut reprendre,
- utiliser la continuation d'un processus père dont l'un des fils est en cours d'exécution pour générer du parallélisme.

A la différence de Cilk, la continuation d'un processus n'est pas entièrement volée afin d'éviter la recopie de tout l'environnement du processus père, ce qui peut s'avérer coûteux en particulier dans la perspective d'une implantation distribuée. Dans ce cas, le processus fils en cours d'exécution est suspendu et le flot d'exécution est rendu au processus père jusqu'à ce qu'un nouveau processus fils soit créé. C'est ce nouveau processus qui sera volé par le processeur inactif.

Réalisation efficace de l'ordonnancement

L'algorithme d'ordonnancement présenté ci-dessus est implanté en gérant deux files d'attente : la première pour les processus prêts à être exécutés et la seconde pour

les continuations des processus parents dont l'un des fils est en cours d'exécution.

A chaque processus créé (dégénéré séquentiellement ou non) est associé un bloc d'activation permettant de stocker les variables locales du processus. Comme dans le langage Cilk, l'ensemble des blocs d'activation est géré à l'aide d'une pile dont le fonctionnement est très proche d'une pile C standard lorsque les processus sont exécutés séquentiellement, et qui devient une pile cactus lorsqu'un processus est volé. Afin d'optimiser son implantation, la pile cactus n'est pas gérée en chaînant entre eux les blocs d'activation alloués dans le tas, mais en allouant et en chaînant de plus gros blocs de mémoire permettant de stocker plusieurs blocs d'activation. L'allocation d'un nouveau bloc d'activation dans la pile cactus revient simplement à incrémenter un pointeur dans le bloc mémoire courant, sauf si le bloc mémoire n'est plus assez grand. Dans ce cas un nouveau bloc mémoire est alloué dans le tas.

Implantations distribuées

Même si l'environnement Lazy Thread a été réalisé dans la perspective d'une implantation distribuée, aucune version adaptée à ce type d'architecture n'a encore été proposée.

1.3.3 ProActive

ProActive [19, 4, 18] est un environnement de programmation distribuée "objet" développé au sein du projet *Oasis* à l'université de Nice Sophia Antipolis.

Dans cet environnement, une application distribuée est composée d'un ensemble "d'objets actifs" (*active objects*) répartis sur l'ensemble des noeuds de calcul disponibles. Les communications entre objets actifs sont effectuées par l'intermédiaire des appels aux méthodes des objets. Chaque objet actif dispose de son propre flot d'exécution et peut disposer de sa propre politique pour choisir l'ordre de traitement des appels de méthodes qui lui sont transmises. Chaque appel de méthode d'un objet actif est asynchrone et retourne de manière transparente un objet appelé "futur". Cet objet représente le futur résultat calculé par la méthode appelée. L'appelant peut donc continuer son exécution jusqu'à ce qu'une méthode ayant besoin du résultat soit appelée. Dans ce cas, l'appelant est bloqué jusqu'à ce que la donnée dont il a besoin ait été produite. ProActibe appartient donc au modèle de programmation déterministe non contraint.

Contrairement à Cilk ou Lazy Threads, la dégénération séquentielle dans ProActive est explicite. L'utilisateur fait appel aux méthodes d'un objet actif de la même manière que pour les appels aux objets standards. Cet appel peut, par liaison dynamique être un appel de méthode normal ou un appel distant parallèle et asynchrone. La figure 1.6 illustre ces deux possibilités.

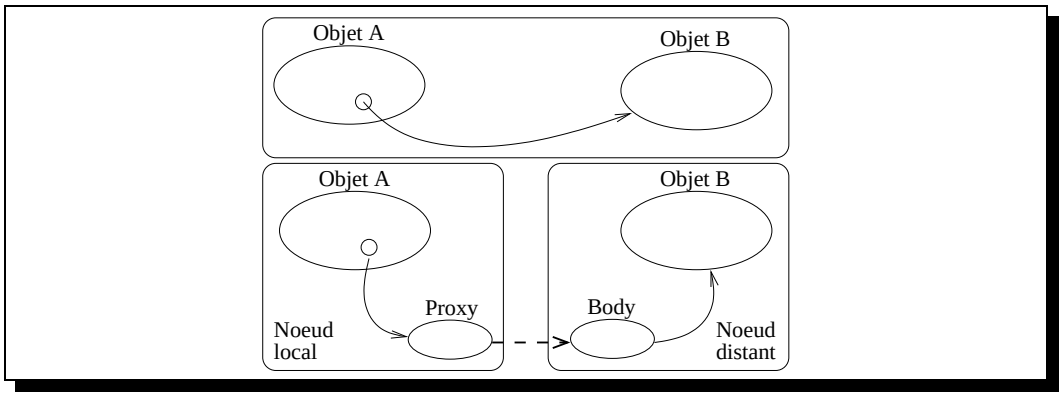


FIG. 1.6 – Appel de méthode standard ou distant dans ProActive.

1.4 Ordonnancement distribué centralisé

Dans cette section, nous proposons d'étudier les stratégies d'ordonnancement dynamique implantées dans quelques environnements de programmation parallèle. Nous présenterons dans un premier temps, les implantations centralisées des stratégies d'ordonnancement utilisées par Jade et OVM. Bien que ce type d'implantation ne permette pas de passer à l'échelle, une implantation centralisée a l'avantage d'être plus simple à mettre en œuvre qu'une implantation distribuée. De plus, de bonnes performances peuvent être obtenues sur des grappes de PCs de taille moyenne. Dans un deuxième temps, nous présenterons deux types de stratégies permettant d'obtenir des exécutions performantes sur des architectures de grande taille. La stratégie utilisée par Satin, basée sur du vol de travail hiérarchique ainsi que celle utilisée dans Charm++ seront détaillées.

1.4.1 Jade

Le langage Jade [56, 55] a été développé à l'université de Stanford entre les années 1990 et 1994. Son interface, basée sur le langage C, permet d'exprimer de manière implicite et portable le parallélisme entre les tâches générées par une application.

Description du parallélisme

La description du parallélisme dans Jade est basée sur la décomposition explicite du programme en tâches de calcul. Le parallélisme et les synchronisations entre tâches sont automatiquement détectés en spécifiant les accès effectués par les tâches sur les données allouées en mémoire partagée. Les types d'accès qu'une tâche peut déclarer sont les suivants : la lecture, l'écriture, l'accès commutatif et la destruction. Par exemple, lorsqu'une tâche déclarant un accès en écriture est créée juste avant une tâche déclarant un accès en lecture sur la même donnée partagée, une synchronisation est automatiquement insérée entre ces tâches. Si par contre deux tâches

n'ont aucune donnée en commun, celles-ci ne sont pas dépendantes l'une de l'autre et peuvent être exécutées dans n'importe quel ordre l'une par rapport à l'autre. Le noyau exécutif de Jade peut dans ce cas décider d'exécuter les deux tâches en parallèle. Cette interface permet de décrire des programmes appartenant au modèle de synchronisation déterministe non contraint.

Stratégie d'ordonnancement utilisée

L'implantation distribuée du langage Jade est basée sur un algorithme d'ordonnancement centralisé. L'objectif de cet algorithme est d'équilibrer dynamiquement la charge entre les différents processeurs tout en maximisant la localité des données (ordonnancement par affinité).

Cette implantation est optimisée pour le cas où le processus principal crée toutes les tâches du programme. Le principe est d'associer chaque tâche à un objet partagé (le premier objet partagé que la tâche déclare accéder) afin d'exécuter en priorité la tâche sur le site propriétaire de la donnée. Ce site est appelé le site préférentiel de la tâche. Les choix liés à l'ordonnancement sont centralisés sur le site d'exécution du processus principal. L'ordonnanceur a pour objectif de maintenir un niveau de charge minimum sur tous les sites en distribuant les tâches de manière à ce que le nombre de tâches prêtes à s'exécuter sur chacun des sites ne descende pas en dessous d'un certain seuil. Lorsqu'une tâche devient prête sur le site principal, l'ordonnanceur cherche dans un premier temps un site dont le nombre de tâches prêtes est en dessous du seuil, plusieurs cas sont alors possibles :

- Le site préférentiel est en sous-charge, la tâche lui est allouée.
- Le site préférentiel n'est pas en sous-charge alors que d'autres sites le sont, la tâche est arbitrairement allouée à l'un des sites en sous-charge.
- Aucun des sites n'est en sous-charge, la tâche est stockée dans une file d'attente locale.

Lorsqu'une tâche termine son exécution sur un site distant, le site principal en est informé. Celui-ci recherche alors une tâche dans sa file d'attente locale en donnant la priorité aux tâches dont le site préférentiel est le site distant.

Cette implantation a deux inconvénients principaux :

1. La non prise en compte des tâches créées par les tâches exécutées sur un site distant, ce qui exclut d'exécuter efficacement un grand nombre d'applications.
2. La gestion centralisée de l'algorithme limitant les capacités de passage à l'échelle.

Des applications d'algèbre linéaire et de simulation de l'influence de certains courants sur les mouvements à grande échelle des océans ont permis de valider l'implantation sur machine à mémoire partagée comme sur machine à mémoire distribuée en utilisant jusqu'à 32 processeurs de manière efficace.

1.4.2 OVM

OVM (Out-of-order execution parallel Virtual Machine) [14] est un environnement de programmation parallèle basé sur une architecture client-serveur. Il est dé-

veloppé au laboratoire LRI de l'université Paris-Sud.

Description du parallélisme.

Dans l'environnement OVM, le parallélisme est exprimé à l'aide d'appels à distance de procédure qui sont insérés dans le programme utilisateur (*client*). Chaque appel de procédure distant étant asynchrone, le client peut en exécuter plusieurs consécutivement permettant ainsi une exécution parallèle sur les différents *serveurs* d'exécution. Chaque appel est directement transmis à un négociateur centralisé (*broker*) qui est chargé de la gestion des dépendances de données entre les procédures et de la répartition de la charge sur les différents serveurs d'exécution. Les programmes OVM appartiennent au modèle de synchronisation déterministe non contraint.

Stratégie d'ordonnancement utilisée

L'ordonnanceur de l'environnement OVM est basé sur deux stratégies que l'utilisateur peut choisir en fonction de la complexité des tâches de son programme.

Si la granularité des tâches et des données accédées est importante, OVM propose comme dans Jade de distribuer statiquement l'ensemble des données sur les serveurs d'exécution et d'ordonnancer les tâches en fonction des données auxquelles elles accèdent. L'utilisateur est chargé de partitionner ses grands ensembles de données et de demander leur allocation persistante côté serveur. Les partitions seront ensuite distribuées de manière cyclique sur les serveurs disponibles. Pour chaque appel de procédure à distance, l'utilisateur spécifie sur quelle partition s'applique la procédure appelée. Le négociateur choisit alors automatiquement le serveur possédant la partition concernée pour exécuter la tâche.

Si la granularité des tâches est fine et que les données auxquelles elles accèdent sont de petite taille, OVM propose de laisser le négociateur choisir lui même les serveurs d'exécution des tâches en fonction de leur charge. Cette stratégie est basée sur le principe *premier arrivé, premier servi* et ne tient pas compte de la localité des données.

Cette implantation a été validée sur les applications tests NAS, ainsi que sur une application de simulation de l'influence de l'impact des rayons cosmiques sur l'atmosphère terrestre. De bonnes performances (accélération proche de l'optimale) ont été obtenues sur une grappe de PC ayant 64 processeurs.

1.5 Ordonnancement distribué hierarchique

1.5.1 Satin

Satin [65, 64] est un environnement de programmation parallèle développé au *Department of Mathematics and Computer Science* de l'université Vrije d'Amsterdam. Son interface étend le langage Java avec trois primitives inspirées du langage Cilk afin de permettre l'implantation simple et efficace des applications de type *diviser pour régner* sur des architectures de grande taille.

Description du parallélisme

De la même manière que dans le langage Cilk, le parallélisme est exprimé dans Satin grâce à la création de processus légers. Lorsqu'un appel de fonction est précédé du mot clé `spawn`, un nouveau processus léger est lancé. Celui-ci peut s'exécuter en concurrence avec le processus qui l'a créé. Le mot clé `sync` permet au processus qui l'appelle d'attendre la fin de l'exécution de tous les processus lancés précédemment. Les procédures destinées à être lancées à l'aide du mot clé `spawn` doivent être précédées du mot clé `satin`.

A la différence de Cilk, Satin n'est pas uniquement destiné aux machines à mémoire partagée. Chaque fonction lancée à l'aide du mot clé `spawn` doit pouvoir s'exécuter sur n'importe quel site d'exécution en n'utilisant que les données présentes en mémoire locale. L'environnement ne fournit pas de mémoire partagée, mais utilise les mécanismes de sérialisation du langage Java pour communiquer les paramètres et les résultats des fonctions exécutées à distance. Aucun autre moyen de communiquer avec ce type de fonction (par exemple par *effets de bords*) n'est autorisé. Comme pour le langage Cilk, les programmes Satin appartiennent au modèle de synchronisation strict réduit (ou imbriqué).

Stratégie d'ordonnancement utilisée

L'un des objectifs de l'environnement Satin est de permettre l'exécution efficace des applications parallèles de type *diviser pour régner* sur des architectures de type grille. Ces architectures sont souvent composées de plusieurs grappes de PCs, chacune utilisant un réseau rapide pour les communications internes alors que les liens inter-grappes peuvent avoir des débits et latences beaucoup plus faibles. La stratégie d'ordonnancement de Satin, appelée CRWS (*Cluster-aware Random Work Stealing*), permet de prendre en compte les caractéristiques de ces machines.

Cette stratégie a l'avantage d'être très intuitive. Lorsqu'un site d'exécution devient inactif, il tire aléatoirement un site parmi tous les sites disponibles afin de lui voler du travail. Si le site choisi est localisé sur la même grappe que le site inactif, une requête de vol synchrone est générée. Si le site choisi est localisé sur une grappe distante, on suppose que le temps d'attente de la réponse peut être long. Une requête de vol asynchrone est alors générée. Afin de ne pas attendre inutilement, le site voleur va tirer aléatoirement un site parmi les sites de la grappe locale et générer une requête de vol synchrone vers ce site. A tout moment, chaque noeud ne doit pas avoir plus d'une requête de vol en cours à destination d'une grappe distante. Si après avoir reçu du travail d'un site de la grappe locale, le site redevient inactif et que la réponse de la grappe distante n'est toujours pas arrivée, un site de la grappe locale sera de nouveau choisi aléatoirement.

Cet algorithme a l'avantage de prendre en compte les caractéristiques des liens inter-grappes sans nécessiter de réglage particulier ni d'informations de charge des processeurs.

Cet algorithme a été testé sur des applications de type produit de matrices, calcul

du n^{ieme} terme de la suite de Fibonacci ou encore résolution du problème du voyageur de commerce. Les résultat ont montré l'efficacité de cette stratégie en comparaison avec des algorithmes de type vol de travail basés sur une recherche aléatoire du site victime. Ces expériences ont été menées sur une grappe de PCs sur laquelle certains liens ont été ralentis pour simuler le comportement d'une grille.

1.5.2 Charm++

Charm++ [36] est un langage de programmation parallèle par objet basé sur le langage C++. Il est développé au département d'informatique de l'université de l'Illinois depuis 1993. L'une des principales applications développée avec Charm++ est une application de dynamique moléculaire NAMD [38]. Lors d'une récente expérience cette application a utilisé efficacement 3000 processeurs sur une grosse instance du problème.

Description du parallélisme

Le parallélisme dans Charm++ est basé sur la création d'objets concurrents appelés *chares*. Un *chare* est un objet C++ standard dont les méthodes peuvent être appelées à distance par n'importe quel autre *chare* disposant d'une référence sur cet objet. Un appel de méthode est effectué en créant un *message Charm++* contenant les paramètres de la méthode à appeler. Chaque *chare* n'est autorisé à accéder qu'à ses propres attributs. L'accès aux attributs d'autres *chares* n'est autorisé qu'en passant par leurs méthodes, contrairement aux objets C++ standards pour lesquels les attributs publiques peuvent être accédés directement par d'autres objets. Chaque *chare* étant autorisé à appeler les méthodes de tout autre *chare*, les programmes Charm++ appartiennent au modèle de synchronisation déterministe non contraint.

Stratégie d'ordonnancement utilisée

L'ordonnancement dans Charm++ permet de distribuer statiquement un ensemble de *chares* sur les processeurs disponibles, de choisir dynamiquement un site lorsqu'un nouveau *chare* est créé et de migrer les *chares* en fonction des déséquilibres de charge. Un certain nombre de stratégies d'ordonnancement sont disponibles,et la bibliothèque permet d'intégrer facilement de nouvelles stratégies statiques ou dynamiques en fonction des caractéristiques de l'application et de la machine cible. L'une des stratégies utilisable par défaut dans la bibliothèque est la stratégie ACWN *adaptive contracting-within-neighborhood*. Lorsqu'un message de création de nouveau *chare* est envoyé, cette stratégie choisit le moins chargé des sites de son voisinage comme destinataire. Le nouveau *chare* sera alors alloué sur le site choisi. Les informations de charge sont échangées soit en les concaténant aux messages Charm++ de l'application, soit en créant périodiquement des messages spécifiques dédiés à cette tâche.

Dans le cadre du développement de l'application NAMD, de nouvelles stratégies d'ordonnancement ont été intégrées à Charm++ afin de prendre en compte ses carac-

téristiques. NAMD est une application de simulation du comportement des atomes d'une molécule au cours du temps. Elle est implantée de manière à calculer itérativement les nouvelles coordonnées spatiales de chaque atome en fonction des forces auxquelles il est soumis.

Pour distribuer les *chares* de cette application, deux types de stratégie d'ordonnancement sont utilisés : un pré-placement initial est tout d'abord calculé statiquement, puis ce pré-placement est affiné dynamiquement. Au cours de la phase de pré-placement, l'espace tridimensionnel occupé par la molécule est partitionné en régions de forme cubique. Les *chares* permettant de simuler le comportement des atomes contenus dans chaque région sont ensuite créés. L'ensemble des *chares* d'une même région est appelé *patche*. Les *patches* sont distribués sur les processeurs en utilisant un algorithme de bi-partitionnement récursif. Une fois la simulation lancée, les *chares* ne migrent pas, sauf lors des phases de réajustement de la charge. Ces phases de réajustement n'interviennent qu'après avoir exécuté un certain nombre d'itérations au cours desquelles des informations de charge ont été récoltées. Si un processeur est surchargé, certains *chares* seront migrés en choisissant en priorité les processeurs chargés de la simulation des régions voisines.

1.6 Conclusions

Dans ce chapitre, nous avons présenté les stratégies d'ordonnancement proposées dans des environnements de programmation parallèles représentatifs. Nous avons vu dans un premier temps que l'utilisation d'algorithmes d'ordonnancement statique pouvait conduire à la génération de programmes distribués dont l'efficacité est très proche des programmes MPI optimisés. Cette technique est utilisée dans Pyrros avec l'algorithme d'ordonnancement DSC, mais peut être étendue à tout autre algorithme statique comme par exemple les algorithmes de partitionnement de Scotch et Metis.

Dans l'étude des environnements basés sur des algorithmes d'ordonnancement dynamiques, nous avons vu que de bonnes performances pouvaient être obtenues à grain fin en optimisant l'exécution séquentielle des tâches même si le vol de travail est rendu plus coûteux. Cette technique permet d'obtenir d'excellentes performances sur machine à mémoire partagée, mais les tentatives de portage sur machine à mémoire distribuée se sont avérées inefficaces.

Pour les programmes à granularité moyenne, les implantations de Jade et OVM ont montré que des algorithmes d'ordonnancement centralisés étaient suffisant pour exploiter des grappes de PC. Afin de passer à l'échelle et tirer partie des grilles de grappes, l'utilisation d'algorithmes d'ordonnancement distribués comme ceux proposés par Satin ou Charm++ s'avèrent indispensables.

L'objectif de cette thèse est de coupler dans un même environnement les techniques de dégénération séquentielles proposées par Cilk et Lazy Threads avec les techniques d'ordonnancement distribués statiques ou dynamiques (dégénération distribuée) que nous avons vues dans ce chapitre. Ce couplage est réalisé au sein de l'environnement Athapascan présenté dans le chapitre suivant.

CHAPITRE 2

Athapascan (version 1) : interface de programmation et spécialisations

Athapascan est un environnement de programmation parallèle de haut niveau basé sur l'interprétation distribuée du flot de données de l'application. Son développement a débuté en 1993 et se poursuit actuellement dans l'équipe MOAIS du laboratoire ID-IMAG. Il est constitué d'une bibliothèque C++ permettant de décrire le parallélisme des applications indépendamment de la machine cible. L'implantation de cette bibliothèque permet de générer des exécutables performants pour des architectures de type multiprocesseur à mémoire partagée ou grappes de PC. Dans ce chapitre, nous présentons l'état d'avancement du développement d'Athapascan (version 1) au début de cette thèse (2000). Cet environnement, dont l'architecture est détaillée dans la figure 2.1, était alors divisé en deux composants :

- Athapascan-0 [17] : fournit un noyau exécutif portable en couplant une bibliothèque de processus légers (POSIX Threads [35]) et une bibliothèque de communication (MPI, Message Passing Interface [23, 31]).
- Athapascan-1 [58, 26, 21, 20, 27] : fournit une interface de programmation parallèle facile à utiliser et efficace.

Dans la suite du chapitre, nous présenterons dans un premier temps l'interface applicative de cet environnement. Les stratégies d'ordonnancement et les modèles de coût associés seront présentés dans une deuxième section. Nous détaillerons ensuite son modèle exécutif avant de présenter les différentes implantations de ce modèle disponibles au début de la thèse. En conclusion, nous étudierons les avantages et les inconvénients de ces implantations qui ont motivés les travaux présentés dans cette thèse.

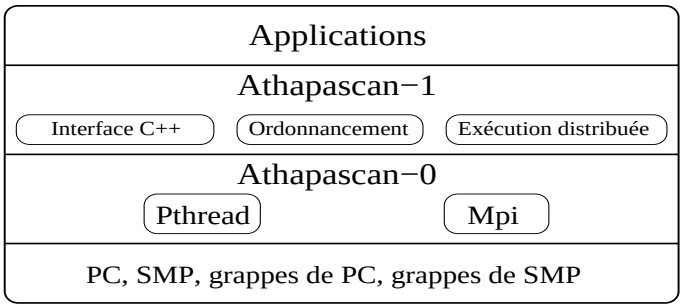


FIG. 2.1 – Architecture de l'environnement Athapascan.

```
1 struct Typetache {
2   void operator() (int i, int j) {
3     ...
4   }
5 };
6 Fork <TypeTache> (2,5);
```

FIG. 2.2 – Déclaration d'un type de tâche.

2.1 Modèle de programmation

Le modèle de programmation d'Athapascan [58] est basé sur le modèle procédural. L'utilisateur décrit de manière explicite le parallélisme de son application en créant des "tâches" (procédures appelables à distance) de manière asynchrone. Les synchronisations entre tâches sont implicitement détectées en fonction des accès qu'elles effectuent sur les données d'une mémoire partagée.

2.1.1 Tâches et données partagées.

D'un point de vue syntaxique, deux mots clés sont principalement rajoutés au langage C++ : le mot clé `Fork`, qui permet de créer une nouvelle tâche, et le mot clé `Shared`, qui permet de créer une nouvelle donnée partagée. L'utilisation du mot clé `Fork` est la suivante :

```
Fork<TypeTache> () (ParametresTache);
```

Cette instruction permet de créer une nouvelle tâche de type "TypeTache" avec les paramètres "ParametresTache" qui sera prise en charge par le système pour être exécutée par le même processeur ou par le processeur d'une machine distante.

La figure 2.2 illustre la manière de déclarer un type de tâche. Une telle déclaration revient à écrire une classe ou une structure C++ avec la définition d'un opérateur `operator()`. Dans l'exemple de la figure 2.2, l'opérateur `operator()` spécifie deux paramètres de type entier. La création syntaxiquement correcte d'une tâche de type "LaTache" peut être par exemple : `Fork<LaTache> () (2,4);`

La création d'une tâche est asynchrone et le noyau exécutif de la bibliothèque peut décider de l'exécuter sur n'importe quel processeur disponible de la machine cible. Toutes les données accédées par la tâche doivent être passées en paramètre et aucun "effet de bord" n'est autorisé pour les tâches. Les paramètres peuvent être passés par valeur, comme dans l'exemple de la figure 2.2, ou par référence en utilisant une mémoire partagée. L'utilisateur peut créer un nouvel objet dans cette mémoire à l'aide du mot clé `Shared` dont l'utilisation est la suivante :

```
Shared<TypeObjet> NomObjet;
```

Il est possible de créer des objets de n'importe quel type structuré en mémoire partagée. L'utilisateur peut donc gérer la granularité des données communiquées au cours de l'exécution de son programme. Par exemple la déclaration d'une matrice découpée en blocs pour un traitement parallèle peut être réalisé en C++ de la manière suivante :

```
vector< Shared< vector<double > > > matrice(nombre_blocs);
```

Ainsi, la granularité des communications est contrôlée par l'utilisateur en allouant chaque bloc à la taille voulue. Dans un environnement distribué, il est nécessaire d'être capable de communiquer chaque objet passé en paramètre des tâches, qu'ils soient passés par valeur ou par référence en utilisant la mémoire partagée. Pour cela, la bibliothèque Athapascan-1 implante les primitives permettant de sérialiser les objets dont le type est un type de base du langage C++ (`int`, `double`, `float`...). Pour chaque type structuré, l'utilisateur doit cependant fournir des fonctions d'emballage et de déballage permettant de sérialiser les objets passés en paramètre.

2.1.2 Détection des synchronisations : typage des accès aux données partagées

Comme nous l'avons dit précédemment, les synchronisations et le parallélisme entre les différentes tâches des programmes sont détectées automatiquement par la bibliothèque. Cette détection est rendue possible par le typage explicite des accès effectués par chacune des tâches sur les données allouées en mémoire partagée. L'utilisateur doit spécifier pour chaque tâche si les données partagées prises en paramètre seront accédées en *lecture*, *écriture*, *modification* ou *accumulation*. Les trois premiers modes d'accès sont associés aux mots clés `Shared_r`, `Shared_w` et `Shared_r_w`. Le dernier mode permet à plusieurs tâches de participer de manière concurrente à la production d'une même donnée partagée. Chaque contribution est accumulée à l'aide d'une fonction définie par l'utilisateur qui est supposée associative et commutative. Le mot clé `Shared_cw` est associé à ce mode d'accès.

En fonction des modes d'accès spécifiés, des méthodes spécifiques à chaque mode doivent être utilisées pour accéder aux données partagées. Lorsqu'un accès en lecture est spécifié, la donnée peut être accédée dans le corps de la tâche avec la méthode `read`. Si la donnée est accédée en écriture, la méthode `write` doit être utilisée. Si la donnée est accédée en modification, la méthode `access` est disponible. Enfin, si la donnée est accédée en accumulation, la méthode `cumul` doit être utilisée. En plus du type d'accès effectué par la tâche, l'utilisateur peut préciser si la donnée sera effectivement accédée par la tâche ou uniquement par son éventuelle descendance. Dans ce cas l'accès est dit différé (*postponed*). La tâche déclarant un tel accès ne peut que transmettre la référence à sa descendance et ne peut en aucun cas accéder elle-même à la donnée.

La sémantique des accès aux données partagées est intuitive car directement liée à l'ordre lexicographique (profondeur d'abord) des accès aux données. Cet ordre correspond à celui d'une exécution obtenue en remplaçant chaque création de tâche par l'exécution de sa procédure associée. Lors d'une exécution parallèle, la sémantique

```
1  struct tache_écriture {
2    void operator() (Shared_w<int> i, Shared_w<int> j) { ... }
3  };
4  struct tache_lecture {
5    void operator() (Shared_r<int> i) { ... }
6  };
7  struct tache_modification {
8    void operator() (Shared_r_w<int> i) { ... }
9  };
10
11 void main() {
12   Shared<int> i, j;
13   Fork<tache_écriture> () (i, j);   // création de t0
14   Fork<tache_lecture> () (i);       // création de t1
15   Fork<tache_modification> () (j);  // création de t2
16   Fork<tache_lecture> () (j);       // création de t3
17   Fork<tache_lecture> () (j);       // création de t4
18 };
```

FIG. 2.3 – Un exemple simple de programme Athapascan.

des accès aux données partagées est telle que tout accès en lecture voit la dernière écriture relativement à l'ordre séquentiel.

L'exécution des tâches est non préemptive, c'est-à-dire que lorsqu'une tâche commence son exécution, le noyau exécutif d'Athapascan-1 garantit qu'elle ne sera pas suspendue tant que son exécution n'est pas terminée. En particulier, lorsqu'une tâche fait appel à une instruction `Fork`, le flot d'exécution n'est pas transmis à la tâche nouvellement créée, mais il est conservé par la tâche mère jusqu'à la fin de son exécution. Cette particularité implique qu'une tâche ne peut pas utiliser les résultats produits pas ses filles. Il n'y a donc pas de synchronisation entre une tâche et celles qu'elle a créées. Un ordre séquentiel d'exécution des tâches, appelé *ordre de référence* respectant la sémantique des programmes ainsi que la propriété de non préemptivité des tâches a été introduit. Appelons i une séquence d'instructions ne contenant pas de création de tâches et T l'instruction permettant de créer une tâche. La trace correspondant à une exécution séquentielle standard en profondeur d'abord peut être la suivante $f = i_1 T_1 i_2 T_2 ... i_{n-1} T_{n-1} i_n$. La propriété de non-préemptivité énoncée ci-dessus implique donc que la trace f est sémantiquement équivalente à la trace $f' = i_1 i_2 .. i_{n-1} i_n T_1 T_2 ... T_{n-1}$.

Définition 1 *L'ordre total appelé $<_r$, dit "ordre de référence" est défini de la manière suivante (t, t', t'' et t_i représentent des tâches) :*

1. *Si t crée t', alors $t <_r t'$*

2. *Si l'exécution de t crée t_1, t_2...,t_n dans cet ordre, alors $t_1 <_r t_2 <_r ... <_r t_n$*
3. *Soient t_i et t_{i+1} deux tâches soeurs ($t_i <_r t_{i+1}$), alors pour toute tâche t' créée lors de l'exécution de la descendance de t_i, $t' <_r t_{i+1}$.*
4. *$\forall t : t_{principale} <_r t$.*

2.2 Ordonnancement et modèle de coût

L'exécution des programmes Athapascan est modélisée par un graphe de flot de données (chapitre 1 section 1.1) évoluant dynamiquement au cours de l'exécution. Les algorithmes d'ordonnancement utilisés sont donc nécessairement dynamiques. Cependant dans certain cas, des algorithmes d'ordonnancement statiques peuvent être utilisés. Imaginons par exemple qu'un programme crée de manière itérative l'ensemble des tâches du programme, ce qui peut être le cas par exemple pour un produit de matrice ou une factorisation de Choleski. Il est dans ce cas possible d'attendre la fin de l'exécution de la tâche principale et donc de connaître le graphe dans sa totalité pour appliquer un algorithme d'ordonnancement statique adapté au problème considéré.

Un modèle de coût est associé au modèle de programmation présenté dans les sections précédentes. Ce modèle permet de prédire théoriquement les performances en temps et en occupation mémoire d'un programme Athapascan. Pour cela, les grandeurs suivantes sont associées à tout graphe de flot de données :

- T_1, représente la durée d'exécution du programme parallèle sur un seul processeur. Cette grandeur tient donc compte du surcoût lié à la description du parallélisme. Elle est aussi appelée "travail de l'application".
- S_1, représente la consommation mémoire lors d'une exécution séquentielle selon l'ordre de référence.
- T_∞, représente le temps d'exécution sur une infinité de processeurs, c'est-à-dire le temps d'exécution d'un plus long chemin critique du graphe de flot de données.

2.2.1 Ordonnancement dynamique

L'un des algorithmes d'ordonnancement dynamique le plus utilisé consiste à allouer une tâche prête (s'il en existe) à un processeur lorsque celui-ci devient inactif. On appelle fréquemment ce type d'algorithme "algorithme glouton" ou "algorithme de liste".

Considérons dans un premier temps qu'on utilise une machine PRAM (*Parallel Random Access Machine*). Ce type de machine est composée de p processeurs et d'une mémoire partagée.

Théorème 1 *Si les communications ne sont pas prises en compte, tout algorithme d'ordonnancement de type liste a un ratio de performance borné par $(2 - \frac{1}{m})$ sur une machine composée de m processeurs identiques.*

La preuve de ce résultat est apportée dans [30].

Si on considère le modèle délai afin de prendre en compte le coût des communications entre processeurs, le théorème suivant permet de majorer le temps d'exécution parallèle.

Théorème 2 *Soit un programme parallèle dont l'exécution génère n tâches et n_d dépendances (noeuds "données" du graphe de flot de données). Soit T_1 le temps d'exécution séquentielle et t_∞ le temps d'exécution minimal sur une infinité de processeurs. Alors, dans le cadre d'un modèle délai de communication, le graphe de flot de données G peut être ordonnancé sur m processeurs identiques en un temps T_m borné par [43] :*

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

La consommation mémoire de l'exécution d'un programme sur p processeurs est majorée par $S_p \leq pS_1$ [26].

2.2.2 Ordonnancement statique

Dans le cas particulier où le graphe de flot de données peut être connu statiquement, il est possible d'utiliser des algorithmes d'ordonnancement statiques comme par exemple ETF (*Earliest Task First)* ou ERT (*Earliest Ready First)* [34]. Ces deux algorithmes d'ordonnancement sont de type liste. Ils consistent tous les deux à ordonnancer la tâche qui pourra commencer son exécution au plus tôt (en prenant en compte les communications) sur l'un des processeurs inactifs. Le théorème suivant donne une évaluation des performances de ces algorithmes sur un modèle délai. Le temps de communication de n mots entre deux processeurs est supposé prendre τn unités de temps. On note C_{max} le maximum, sur tous les chemins du graphe, du volume de communication cumulé sur un chemin.

Théorème 3 *Si on néglige le coût de calcul et de réalisation de l'ordonnancement, tout ordonnancement de type liste assignant une tâche prête sur le processeur qui démarrera son exécution au plus tôt conduit à un temps d'exécution T_p majoré par :*

$$T_p \leq \frac{T_1}{p} + (1 - \frac{1}{p})T_\infty + \tau C_{max}$$

2.3 Modèle d'exécution et implantation

Le noyau d'exécution d'Athapascan a pour objectif de détecter automatiquement les synchronisations et le parallélisme de l'application et de générer une exécution distribuée efficace. Ce noyau utilise pour cela un modèle basé sur la gestion distribuée du flot de données. Ce modèle, décrit par la figure 2.4, peut être décomposé en trois étapes se répétant de manière cyclique tout au long de l'exécution du programme :

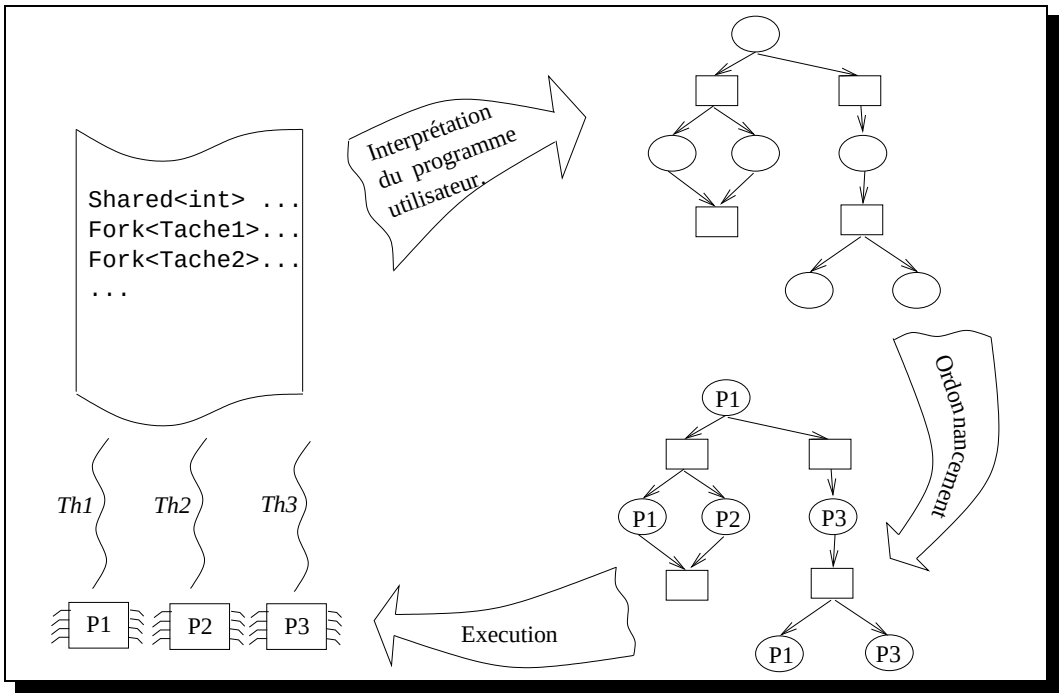


FIG. 2.4 – Modèle d'exécution d'un programme Athapascan-1

- Exécution d'une tâche sur l'un des processeurs disponibles. La tâche choisie est dite "prête", c'est à dire que tous ses paramètres en entrée ont été produits. L'exécution du programme commence par la tâche principale (fonction `main` du programme).
- Au cours de l'exécution de la tâche, les instructions `Fork` et `Shared` sont interprétées et un graphe représentant le flot de données de l'application est maintenu.
- Les tâches du graphe de flot de données sont ensuite placées sur les différents processeurs disponibles.

2.3.1 Gestion du flot de données

La détection automatique du parallélisme et des synchronisations entre tâches est basée sur la construction "à la volée" du *graphe de flot de données* représentant le futur de l'exécution du programme. Le modèle d'exécution d'Athapascan s'articule donc autour de la gestion de ce graphe.

Construction du graphe de flot de données

Au début de l'exécution d'un programme, la seule tâche connue est la tâche principale (le `main`). Le graphe de flot de données est donc nécessairement construit

dynamiquement au cours de l'exécution du programme. L'exécution de chaque instruction Athapascan-1 est associée à une opération sur le graphe de flot de données. Lorsqu'une instruction `Shared` est exécutée, un nouveau noeud "donnée" est initialisé. Lorsqu'une instruction `Fork` est exécutée, un nouveau noeud "tâche" est inséré dans le graphe. Des arcs et éventuellement de nouveaux noeuds "donnée" sont ensuite insérés dans le graphe en fonction des données partagées accédées par la nouvelle tâche. Lorsqu'une tâche fille t_f est créée par la tâche mère t_m, les opérations suivantes sont réalisées :

- La tâche t_f est insérée dans le graphe.
- Pour chaque objet partagé passé en paramètre de la tâche :
  - Si l'objet partagé est accédé en lecture : une arête est ajoutée entre le noeud "donnée" associé à la version à lire et le noeud "tâche".
  - Si l'objet partagé est accédé en écriture : un nouveau noeud "donnée" est créé et une arête est ajoutée entre le noeud "tâche" et le noeud "donnée".
  - Si l'objet partagé est accédé en modification : une arête est ajoutée entre le noeud "donnée" associé à la version à lire et le nœud "tâche". Un nouveau nœud "donnée" est créé correspondant à la version modifiée par la tâche. Une arête est ajoutée entre le noeud "tâche" et le nouveau noeud "donnée".
  - Si l'objet partagé est accédé en accumulation, deux cas sont possibles : soit la tâche t_f est la première à déclarer l'accès en accumulation, soit d'autres tâches accumulant sur cette donnée ont déjà été déclarées. Dans le premier cas, un nouveau noeud "donnée" est créé et une arête est ajoutée entre le noeud "tâche" et le noeud "donnée". Dans le second cas, une arête est ajoutée entre le noeud "tâche" et le noeud "donnée" correspondant à la donnée à accumuler.

A la fin de son exécution, la tâche t_m est supprimée du graphe. La figure 2.5 représente l'état du graphe de flot de données à différentes étapes de l'exécution du programme de la figure 2.3. La partie a) de la figure représente l'état du graphe après l'exécution de la ligne 14 de ce programme. A cette étape, deux objets partagés i et j ont été déclarés. Les parties b) c) d) e) et f) montrent l'état du graphe après l'exécution des lignes 15, 16, 17, 18 et 19 du programme.

Exécution dans le respect des contraintes de flot de données

Pour pouvoir exécuter les tâches de l'application en respectant les contraintes de précédence décrites par le graphe de flot de données, il faut être capable de déterminer à tout moment l'ensemble des tâches "prêtes" à être exécutées. Pour cela, un état est associé à chaque noeud du graphe. Les états possibles pour les noeuds "tâches" sont les suivants :

- Attente : au moins un des paramètres de la tâche n'a pas encore été produit.
- Prêt : tous les paramètres de la tâche ont été produits, elle est prête à être exécutée.
- Exécution : la tâche est en cours d'exécution.
- Terminé : l'exécution de la tâche est terminée, elle peut donc être supprimée

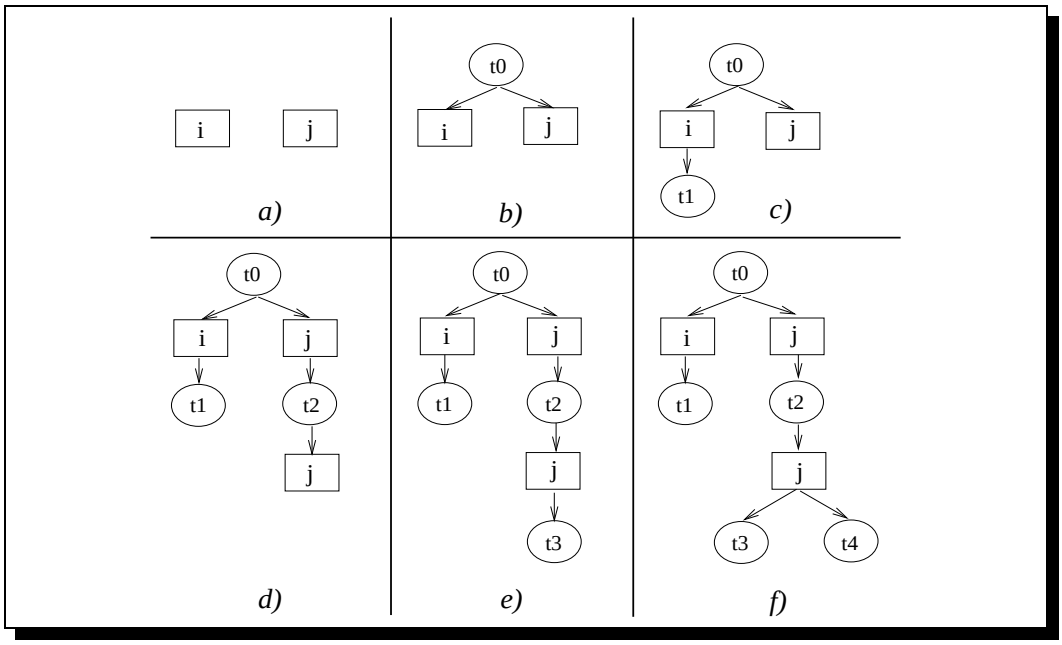


FIG. 2.5 – Construction dynamique du graphe de flot de données.

du graphe.

Les états possibles pour les noeuds "données" sont les suivants :

- Attente : l'un des noeuds "tâches" prédécesseurs du noeud "donnée" n'a pas terminé son exécution.
- Prêt : tous les noeuds "tâches" prédécesseurs du noeud "donnée" ont terminé leur exécution.
- Terminé : toutes les tâches reliées au noeud "donnée" ont terminé leur exécution. La donnée peut donc être supprimée du graphe.

Les états des noeuds "tâches" et des noeuds "données" sont calculés et mis à jour à chaque fois qu'une tâche termine son exécution. Dans un premier temps, l'état de chaque noeud "donnée" succédant le noeud "tâche" est passé à l'état prêt (sauf dans le cas des écritures concurrentes où l'état du noeud "donnée" est passé à prêt seulement au moment de la dernière accumulation). Dans un deuxième temps, les noeuds "tâche" successeurs de ces noeuds "donnée" sont analysés et éventuellement passés à l'état prêt. La figure 2.6 illustre ce mécanisme en décrivant quelques étapes d'une exécution possible du programme de la figure 2.3. a) Initialement, seule la tâche t_0 est prête à s'exécuter. c) Après son exécution, les états des noeuds "donnée" i et j sont mis à jour ainsi que l'état des noeuds "tâche" t_1 et t_2. d) Comme ces deux tâches deviennent prêtes, elles peuvent s'exécuter en parallèle. e) et f) Après l'exécution de t_2, la deuxième version de la donnée j devient prête et les tâches t_3 et t_4 peuvent elles aussi s'exécuter en parallèle.

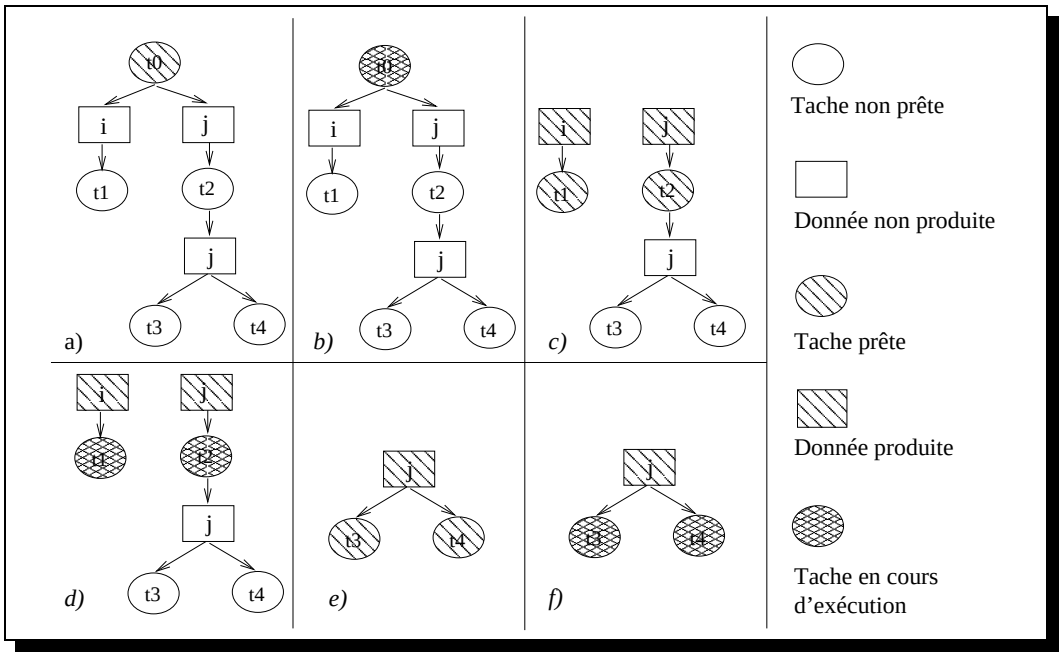


FIG. 2.6 – Exécution selon les contraintes de flot.

2.3.2 Les différentes implantations de ce modèle

Au début des travaux de thèse présentés dans ce manuscrit, plusieurs implantations du modèle présenté ci-dessus ont été proposées. Chacune de ces implantations était spécialisée pour un type de problème donné ou un type d'architecture cible.

Implantation spécialisée SMP (a1-smp)

L'une des implantations proposées était destinée aux architectures multiprocesseurs à mémoire partagée. Dans cette implantation, à chaque fois qu'une nouvelle tâche passait dans l'état "prête", elle était insérée dans une liste. Cette liste était accédée de manière concurrente par plusieurs processus légers dont le rôle était d'exécuter les tâches de la liste. Le principale inconvénient de cette implantation provient du fait que chacune des opérations sur cette liste devait être synchronisée à l'aide d'un verrou associé à la liste. Pour cette raison, les exécutions sur un grand nombre de processeurs ou sur des applications ayant un grain trop fin s'avéraient inefficaces.

Implantation SPMD distribuée optimisée pour les graphes statiques (a1-mpi)

Une deuxième implantation spécialisée pour les applications pour lesquelles un algorithme d'ordonnancement statique peut être utilisé a aussi été proposée. L'objectif de cette implantation est d'obtenir des exécutions sur des machines à mémoire

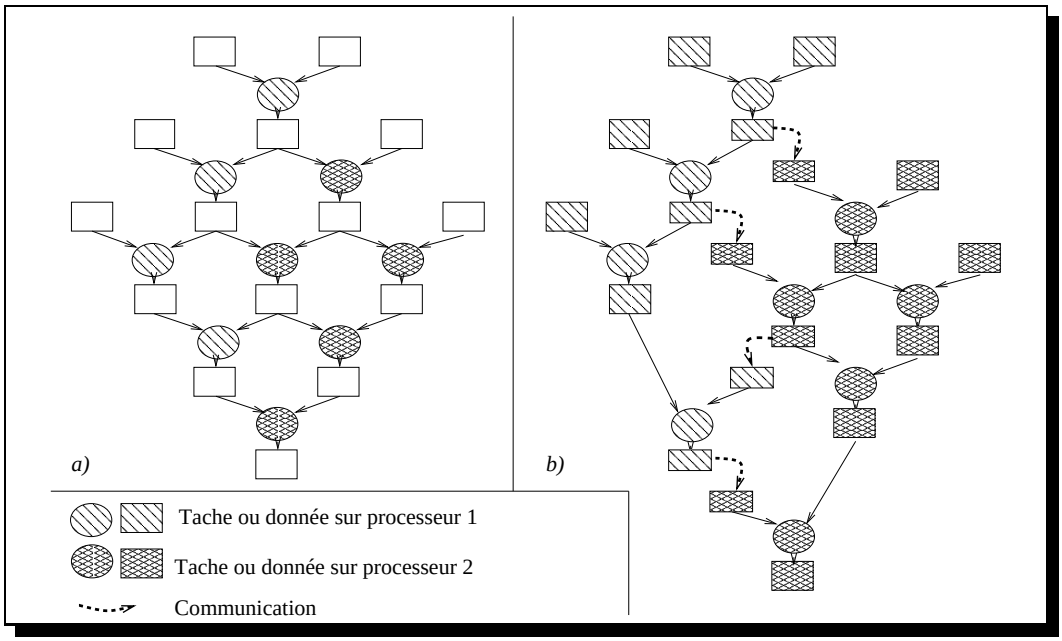


FIG. 2.7 – Génération de communications unidirectionnelles dans le cas où le graphe de flot de données est ordonnancé statiquement.

distribuée aussi efficace qu'un programme MPI équivalent. On considère pour cela que la totalité du graphe de flot de données a été construit par la tâche principale du programme. Celle-ci était dupliquée sur chacun des sites participant au calcul. A la fin de son exécution, chaque noeud dispose alors en mémoire locale de la totalité du graphe de l'application. Un algorithme d'ordonnancement statique déterministe est appliqué sur chaque site avant de lancer l'exécution. Les algorithmes d'ordonnancement comme ceux proposés par les bibliothèques Scotch et Metis utilisant des primitives de tirage aléatoire de nombres ne peuvent donc pas être utilisés.

Le principal avantage de cette implantation est la possibilité d'optimiser très finement la communication des objets partagés entre les sites d'exécutions des tâches. En effet, puisque chacun des noeuds connaît le graphe dans sa totalité ainsi que le site d'exécution de chaque tâche, il est facile de calculer l'ensemble des noeuds devant consommer une donnée partagée venant d'être produite. Une simple primitive d'envoi de message peut donc être utilisée pour envoyer la donnée sur les sites concernés. De la même manière, lorsqu'une tâche n'est pas prête, il est facile de connaître le site de production des données partagées dont elle a besoin. Dans ce cas, une primitive de réception asynchrone de message peut être utilisée. Les accès distants sont donc dégénérés en communications unidirectionnelles.

Outre les restrictions sur les algorithmes d'ordonnancement utilisables, cette implantation ne permet pas d'utiliser toutes les possibilités du langage. En particulier, les accès différés ne sont pas disponibles. La figure 2.7 illustre la manière dont les

communications sont générées automatiquement par cette implantation. La partie *a)* de la figure montre un graphe dont les tâches ont été placées sur une machine à deux processeurs. La partie *b)* de la figure montre les communications qui ont été générées.

L'exécution des tâches est ensuite gérée de la même manière que dans la version SMP. Sur chacun des sites de calcul, un ensemble de processus légers partagent une liste de tâches prêtes.

Implantation distribuée sans restriction (a1-dist)

Enfin, une implantation distribuée sans restriction sur le modèle de programmation a été développée. Cette implantation est considérée comme l'implantation de référence. L'une de ses principales caractéristiques est l'utilisation d'un algorithme de terminaison distribuée pour détecter pour chaque donnée partagée la fin de la phase de production, ainsi que celle de la phase de consommation. Dans la pratique, l'utilisation d'un tel algorithme s'avère coûteux et affecte les performances globales de la bibliothèque. Cette implantation est donc destinée aux applications à "gros" grain pour lesquelles ce surcoût peut être masqué par les calculs.

Dans cette implantation, l'ordonnancement est une extention dynamique ("plug-in") que l'utilisateur peut spécialiser. De plus, la notion de groupe d'ordonnancement a été introduite. Chaque tâche est ordonnancée en fonction de la politique du groupe d'ordonnancement auquel elle appartient. Par défaut, le groupe choisi est celui de la tâche mère, mais l'utilisateur peut spécifier pour chaque tâche son groupe d'appartenance.

Le principal inconvénient de cette implantation provient du surcoût d'utilisation lié en particulier à la version du noyau exécutif Athapascan-0 utilisé. Dans cette version, un démon dont l'objectif était de traiter les communications entrantes de manière réactive utilisait une partie non négligeable de la puissance de calcul.

2.4 Applications et performances

Nous présentons dans cette section quelques applications de la bibliothèque Athapascan-1 qui ont permis de valider les implantations décrites dans les sections précédentes. Ces applications vont du calcul du placement des n-reines qui génère récursivement beaucoup de parallélisme à des applications d'algèbre linéaire dense et creuses en passant par une application de compression de fichiers en parallèle.

2.4.1 Placement des n-reines

Cette application consiste à placer n reines sur un échiquier de taille $n * n$ de manière à ce qu'aucune d'entre elles ne soit en prise. L'application calcule le nombre de positions possibles respectant cette propriété. Les tâches de cette application sont créées de manière récursive afin de parcourir la totalité de l'arbre de recherche.

Le tableau 2.1 présente les temps d'exécution de cette application sur une IBM-SP1 sous AIX-4.2 avec 32 processeurs RS6000 cadencés à 120MHz avec 64 Mo de mémoire. On peut remarquer que l'accélération est de 15.83 sur 16 processeurs pour une taille d'échiquier de $15 * 15$. Elle est donc proche de l'accélération optimale.

taille	# tâches	T_1	T_2	T_4	T_8	T_{12}	T_{16}
13	1178	38.04	20.03	9.28	5.06	2.92	2.40
14	1537	225.34	113.95	57.01	28.44	18.65	14.65
15	1964	1450.74	779.39	364.84	188.41	130.80	91.60

TAB. 2.1 – Temps d'exécution en secondes de l'application de placement des n-reines en utilisant de 1 à 16 processeurs d'une IBM-SP1 [26].

2.4.2 Compression parallèle, *gzip*

La parallélisation de l'application *gzip* permettant de compresser des fichiers est basée sur le fait que l'application complémentaire *gunzip* est capable de décompresser des fichiers découpés en blocs compressés. Une stratégie "diviser pour régner" a donc été utilisée pour sa parallélisation. Au cours de l'exécution parallèle de ce programme, le fichier cible est dans un premier temps découpé en blocs qui seront ensuite compressés indépendamment par des tâches Athapascan-1 qui encapsulent la version séquentielle de *gzip*. Les blocs sont ensuite réordonnés pour créer le fichier compressé. La figure 2.8 montre les temps d'exécution de cette application en fonction de la taille du fichier à compresser sur une SMP ayant 4 processeurs cadencés à 200 MHz avec 256Mo de mémoire sous SunOs 5.6. L'implantation SMP de la bibliothèque Athapascan-1 a été utilisée pour ces expérimentations.

2.4.3 Algèbre linéaire dense, factorisation de Choleski

Dans cette sous-section, nous présentons les résultats obtenus sur une version parallèle d'une factorisation LL^t d'une matrice dense A symétrique, tel que $A = LL^t$ avec L triangulaire inférieure. Cette factorisation est réalisée par une élimination de Choleski.

La parallélisation de cette application a été effectuée en restructurant l'algorithme de manière à travailler au niveau d'opérations sur des sous-matrices. Cette décomposition par bloc permet l'utilisation des BLAS de niveau 3.

Pour ces expérimentations, la version d'Athapascan-1 optimisée pour les graphes de flot de données statiques à été utilisée avec un placement cyclique bi-dimensionel des tâches. Ce placement est le même que celui utilisé par le bibliothèque ScaLapack avec laquelle la comparaison est effectuée. La figure 2.9 montre les résultats obtenus en comparant Athapascan-1 et Scalapack sur cette application. Cette expérience a été effectuée sur un réseau de 16 stations SUN sous Solaris 7. Chacune des stations est composée de 4 processeurs Ultra Sparc II cadencés à 295 MHz avec 512Mo de

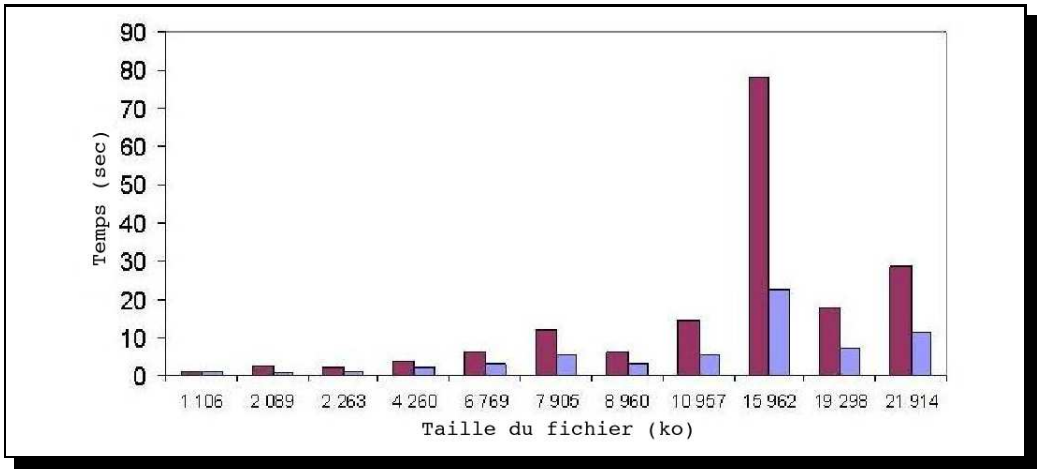


FIG. 2.8 – Performance de la version Athapascan-1 du programme Gzip. Cet histogramme présente les temps d'exécution en seconde pour différentes tailles de fichiers sur une SMP à 4 processeurs.

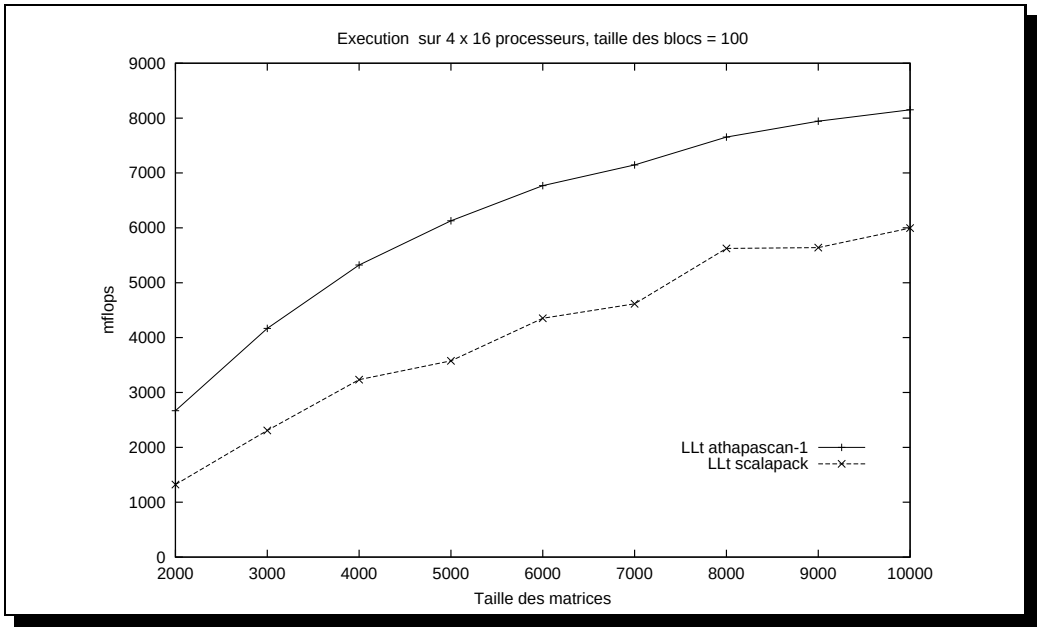


FIG. 2.9 – Factorisation de Choleski, comparaison Athapascan-1/Scalapack sur un réseau de 16 stations SUN [21].

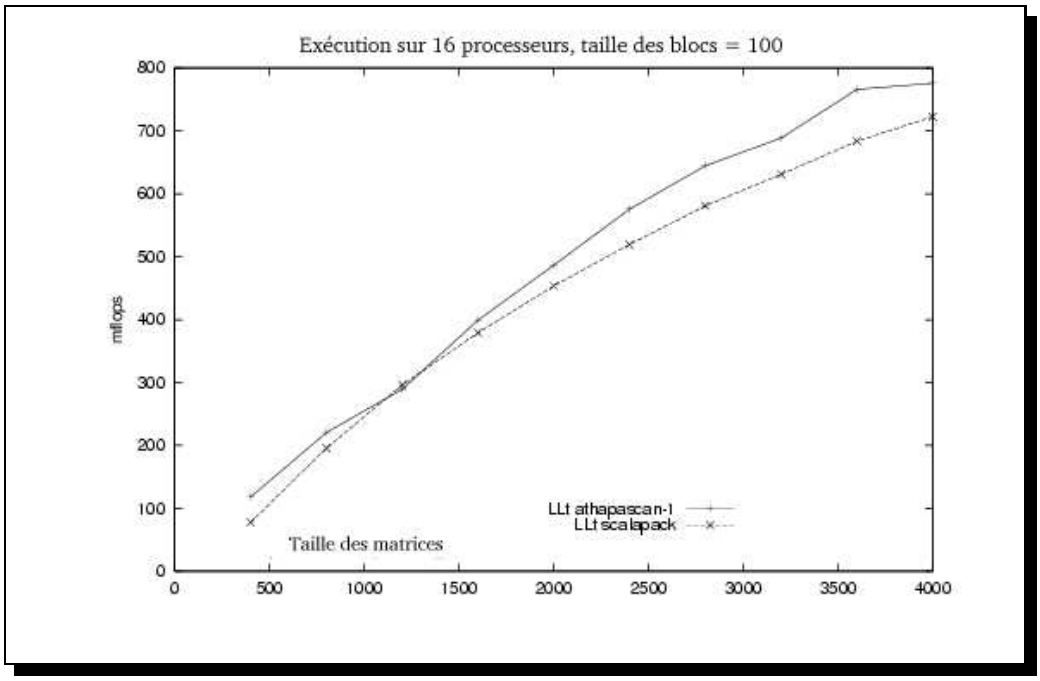


FIG. 2.10 – Factorisation de Choleski, comparaison Athapascan-1/Scalapack sur une IBM-SP1 avec 16 processeurs [21].

mémoire. Le réseau d'interconnection utilisé est un réseau Myrinet. La puissance crête de cette machine est de 390 Mflops (million d'opérations par seconde).

La figure 2.10 montre les résultats obtenus en effectuant la même expérience mais sur 16 processeurs de l'IBM-Sp1 dont les caractéristiques ont été présentées dans la sous-section précédente. Sur le réseau de stations SUN, on peut remarquer qu'Athapascan-1 se comporte nettement mieux que la bibliothèque Scalapack. Cet écart de performance est lié à l'utilisation de processus légers communiquant en utilisant la mémoire partagée pour exploiter le parallélisme intra-noeud dans Athapascan-1 alors que la version de Scalapack disponible sur cette architecture utilisait des processus lourds communiquant par échange de messages même entre deux processus placés sur la même machine.

Sur IBM-SP1, pour Athapascan comme pour Scalapack, chaque processus communique avec les autres par échange de messages, l'écart entre les deux bibliothèques est donc beaucoup moins important. On peut toutefois remarquer que Athapascan-1 est légèrement meilleur pour les matrices de grande taille, ce qui peut s'expliquer par l'utilisation de communications asynchrones dans Athapascan-1, ce qui n'est pas le cas dans ScaLapack.

	Avantages	Inconvénients
Version SMP	- Bonnes performances si peu de processeurs.	- Synchronisation globale sur la liste des tâches prêtes.
Version pour graphes statiques	- Très bonnes performances.	- Restrictions sur le langage : seule les tâches créées par la tâche principale sont pris en compte par l'ordonnancement. - Seuls les algorithmes déterministes d'ordonnancement statique peuvent être utilisés.
Version générale	- Pas de restrictions sur le langage.	- Performances à gros grain seulement.

TAB. 2.2 – Récapitulation des avantages et inconvénients des différentes implantations d'Athapascan.

2.5 Bilan

Dans ce chapitre nous avons présenté l'environnement de programmation parallèle Athapascan-1 tel qu'il était au début de la thèse. Nous avons détaillé l'interface de programmation ainsi que les différentes implantations qui ont été réalisées. Historiquement, la première implantation à être réalisée fut la version générale n'imposant pas de restriction sur le modèle de programmation ni sur le type d'architecture cible. Cependant, cette version n'était efficace que pour les applications ayant une grosse granularité. Les deux autres implantations ont la particularité d'être spécialisées pour un type d'architecture (implantation SMP) ou pour un type d'application (implantation optimisée pour les graphes statiques). Elles ont été développées afin de montrer que le modèle de programmation pouvait être implanté de manière à rendre possible l'exécution de programmes parallèles à grain fin. Malgré certaines restrictions, l'implantation optimisée pour graphes statiques a permis d'atteindre cet objectif sur architecture distribuée. Le tableau 2.2 récapitule les avantages et inconvénients de chacune des ces implantations.

Pour terminer cette partie, nous proposons une synthèse des environnements présentés dans les deux chapitres précédents. Cette synthèse est effectuée en comparant les environnements étudiés sur quelques points critiques qui les caractérisent.

- Modèle de programmes : ce premier critère consiste à placer le modèle de programme parallèle de chaque environnement dans la classification proposée dans le chapitre 1.
- Modèle de coût : l'existence d'un modèle de coût permettant de prédire les performances en temps ou en mémoire associé à l'exécution des programmes.
- Ordonnancement : le type d'algorithme d'ordonnancement utilisé.

- Application cible : les applications sur lesquelles ont été validés ces environnements.
- Architecture cible : les types d'architecture (SMP, grappe, grille de grappes) sur lesquels ces environnements peuvent être utilisés.

Environnement	Modèle de programmation	Modèle de coût	Ordonnancement	Applications	Architecture cibles
Pyrros	Déterministe non contraint	Non	Statique (DSC)	BLAS,...	Grappes
Scotch/Metis	Déterministe non contraint	Non	Statique (bipartitonnement récursif)	Algèbre linéaire, éléments finis...	Grappes
Cilk	Strict réduit	Temps et mémoire	Dynamique (work stealing)	Joueur d'echec parallèle,...	SMP
Lazy Threads	Autre	Non	Dynamique (work stealing)	Applications tests,...	SMP
Jade	Déterministe non contraint	Non	Dynamique centralisé	Simulations...	SMP et grappes
OVM	Déterministe non contraint	Non	Dynamique centralisé	NAS, simulations,...	Grappes
Satin	Strict réduit	Non	Dynamique distribué hiérarchique	Applications tests,...	Grappes et grilles
Charm++	Déterministe non contraint	Non	Plug-in	Simulations,...	grappes et grilles
Athapascan	Déterministe non contraint	Temps et mémoire	Plug-in	Algèbre linéaire, simulations...	SMP, grappes et grilles

TAB. 2.3 – Synthèse des caractèristiques des environnements présentés dans cette partie.

Deuxième partie

Une pile distribuée permettant le couplage des dégénérations séquentielle et distribuée

CHAPITRE

3

Vers un couplage des dégénérations séquentielle et distribuée

Dans la première partie de cette thèse, nous avons montré la variété des algorithmes d'ordonnancement utilisés dans les environnements de programmation parallèle. La plupart de ces environnements permettent d'obtenir de bonnes performances pour des applications à grain moyen sur des architectures de type grappe ou grille de grappes. Pour obtenir des performances à grain fin, certains environnements comme Cilk ou *Lazy Threads* proposent de dégénérer séquentiellement les créations de tâches (ou processus légers) lorsque la génération de plus de parallélisme n'est pas nécessaire. Cependant, ces méthodes de dégénération séquentielle se sont avérées difficiles à porter sur des architectures à mémoire distribuée [51, 10, 53].

Dans le but de permettre un meilleurs passage à l'échelle des mécanismes de dégénération séquentielle, nous proposons d'étudier plusieurs protocoles de cohérence mémoire existants pour l'implantation d'une mémoire partagée distribuée. Pour cette étude, nous proposons d'évaluer le nombre de défauts de page traités lors de l'exécution parallèle d'un programme sur une couche de mémoire partagée distribuée. Dans un environnement distribué, le nombre de défauts de page est étroitement lié au nombre de communications. Trois protocoles seront étudiés. Dans un premier temps, nous évaluerons les protocoles de cohérence paresseuse à la libération [40] et de cohérence DAG [10]. Ces deux protocoles ont été utilisés pour les implantations distribuées SilkRoad [51] et Distributed Cilk [10, 53] de l'interface de programmation du langage Cilk. Bien que ces protocoles de cohérence soient bon théoriquement (peu de défauts de page engendrés par une exécution parallèle), les implantations existantes s'avèrent peu performantes au delà d'une dizaine de processeurs. Nous introduisons donc le protocole de cohérence "flot de données", utilisé par le langage Athapascan [27, 26], qui permet de borner plus finement le nombre de défauts de page que les deux protocoles précédents, mais surtout dont les implantations distribuées existantes permettent d'obtenir un meilleurs passage à l'échelle.

Les bornes qui seront démontrées dépendent étroitement du nombre de tâches exportées lors de l'exécution parallèle d'un programme (une tâche est exportée si elle ne s'exécute pas sur le processeur qui l'a créé). Le résultat principal de ce chapitre est le suivant :

Lors d'une exécution parallèle, si le nombre de tâches exportées est faible, alors le nombre de défauts de page de cette exécution est proche du nombre de défauts de page d'une exécution séquentielle.

Enfin, nous proposerons une machine abstraite dont les instructions permettent

de construire dynamiquement le graphe de flot de données nécessaire à l'implantation du protocole de cohérence flot de données.

3.1 Dégénération distribuée - analyse du nombre de défauts de page

Nous proposons dans cette section de comparer trois implantations possibles de mémoire partagée distribuée associées à trois protocoles de cohérence différents :

- Cohérence paresseuse à la libération (*Lazy Release Consistency*). Les couches de mémoire partagée TreadMarks [1, 40] et SCIOS [41] sont basées sur ce protocole. Bien que cette cohérence soit associée au modèle de synchronisation non déterministe, TreadMarks a été utilisé pour l'implantation distribuée du langage Cilk du projet SilkRoad [51].
- Cohérence graphe de précédence (*Dag-consistency*). Cette cohérence est associée au modèle de synchronisation strict réduit. Elle est employée dans Distributed Cilk [10, 53]. Nous verrons que cette cohérence peut aussi être utilisée dans le cadre du modèle de synchronisation déterministe non contraint.
- Cohérence flot de données. Cette cohérence est associée au modèle de synchronisation déterministe non contraint. Elle est utilisée dans le langage Athapascan [27, 26].

Pour effectuer cette comparaison, nous proposons d'analyser le nombre de défauts de page que peut engendrer l'exécution de programmes parallèles qui utilisent ces mémoires partagées. Si une page n'est pas dans la mémoire locale du processeur qui veut l'utiliser, la page devra lui être communiquée. Nous considérons dans cette étude que chacune des couches de mémoire partagée fonctionne sur une architecture distribuée composée d'un ensemble d'unités de calcul P_i chacune associée à un module de mémoire M_i (figure 3.1). Pour que P_i puisse utiliser une donnée, celle-ci doit se trouver dans M_i. Chacune des mémoires M_i est virtuellement divisée en deux zones mémoires : la zone de mémoire locale et la zone de mémoire partagée. Des allocateurs spécifiques permettent d'allouer des données dans l'une ou l'autre des zones mémoires.

Nous considérons qu'un programme parallèle est composé de tâches pour lesquelles le seul moyen de communiquer est d'utiliser la mémoire partagée. Toute tâche peut créer d'autres tâches et se synchronise avec les autres en fonction du modèle de programme considéré (verrous pour TreadMarks et SCIOS, graphe de précédence pour Cilk,...).

3.1.1 Cohérence paresseuse à la libération

Présentation du protocole

Cette cohérence (*Lazy Release Consistency*) est utilisée dans les environnements TreadMarks [1] et SCIOS [41]. On considère dans ce protocole que tous les accès à la mémoire partagée doivent être synchronisés en utilisant les verrous proposés par la

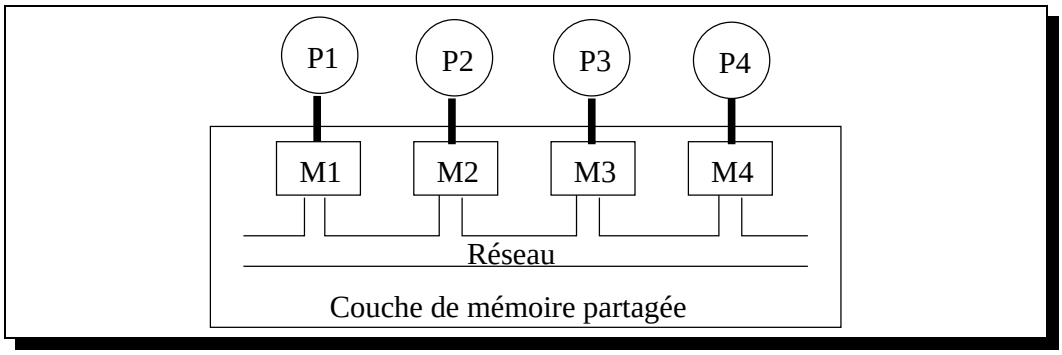


FIG. 3.1 – Modèle d'architecture considéré

bibliothèque utilisée. Toute tâche du programme doit donc prendre un verrou avant d'effectuer tout accès à la mémoire partagée et relâcher le verrou en fin d'utilisation. On dit que le programme est correctement synchronisé si tous les accès sont effectués entre une prise et un relâchement de verrou. L'exécution doit être garantie sans inter-blocage. Ce protocole de cohérence est donc associé au modèle de synchronisation non déterministe. Lorsqu'un processeur accède localement à une ou plusieurs pages, les autres processeurs ne pourront pas accéder aux modifications tant que le verrou n'aura pas été relâché. Le principe est donc de déléguer les opérations de mise en cohérence de la mémoire partagée aux primitives de synchronisations : `verrou.prendre()` et `verrou.relacher()`.

Dans l'implantation proposée par Treadmarks, à chaque page de la mémoire distribuée est associé un site "maître" qui conserve la copie "à jour" de cette page. Si un processeur veut accéder à une page qui ne se trouve pas dans sa mémoire locale, la page "maîtresse" peut être recopiée et envoyée à ce processeur ou être migrée vers ce processeur. Si la page est migrée, le processeur recevant la page devient le processeur maître de celle-ci.

Chaque page de la mémoire partagée peut être dans trois états possibles :

- **Modifiée** La page a été modifiée par le processeur local.
- **Non modifiée** La page n'a pas été modifiée par le processeur local depuis son installation dans la mémoire locale.
- **Invalide** La page ne peut pas être accédée. Lors du prochain accès effectué par le processeur local, un défaut de page interviendra et la page devra être récupérée chez le processeur maître.

Pour assurer la cohérence, chaque processeur dispose de quatre opérations sur les pages qu'il possède en mémoire :

- **Mettre-à-jour** : Les modifications apportées à la copie locale sont mises à jour sur le site maître de la page.
- **Ramener** : cette opération est utilisée lorsqu'un processeur veut accéder à une page qu'il ne possède pas en mémoire. La page "maîtresse" est soit recopiée soit migrée vers ce processeur. Sauf si la mémoire locale n'est pas pleine, la nouvelle

page devra être allouée à la place d'une autre page. Le protocole LRU *Least Recently Used* est utilisé pour choisir la page à supprimer. Si la page choisie est dans l'état *Modifiée*, l'opération *Mettre-à-jour* est préalablement utilisée.
- **Invalider** : cette opération prend en paramètre une page dont le processeur local est le maître. Toutes les copies distantes de cette page sont invalidées.

Lors de l'exécution d'une primitive `verrou.relacher()` l'opération *Mettre-à-jour* est utilisée pour toutes les pages dans l'état *Modifiée*. Lorsqu'un site maître reçoit la mise à jour d'une page, des messages d'invalidation sont envoyés à chaque site possédant une copie de la page. Ces invalidations ne sont pas prises en compte immédiatement par les processeurs qui les reçoivent mais sont différées jusqu'à l'exécution du prochain `verrou.prendre()`.

L'un des principaux problèmes lié à l'utilisation d'une mémoire partagée paginée distribuée est le problème du faux partage. On considère qu'il y a faux partage si deux processeurs accèdent simultanément en écriture à deux données partagées distinctes appartenant à la même page. Ce problème est résolu dans TreadMarks et SCIOS en autorisant les deux processeurs à travailler sur des copies distinctes (technique des pages "jumelles"). Lorsque les processeurs arrivent au point de synchronisation suivant, les modifications apportées à chacune des copies sont envoyées au processeur maître de la page pour y être prise en compte.

Evaluation

Afin d'évaluer les performances en terme de défauts de page de chacune des implantations étudiées, nous proposons de s'inspirer de la méthode proposée dans [10] dans laquelle le nombre de défauts de page lors de l'exécution parallèle est calculé en fonction du nombre de défauts de page lors d'une exécution séquentielle du programme. Nous considérons pour cela que l'exécution séquentielle des tâches est effectuée en profondeur d'abord. Lors de la création d'une tâche, le flot d'exécution est donc transféré de la tâche mère à la tâche nouvellement créée. Dans la suite, on note Q le nombre de pages que peut contenir un module de mémoire et F_{seq} le nombre de défauts de page intervenus lors de l'exécution séquentielle d'un programme. On considère, pour cette exécution de référence, que la mémoire est initialisée de manière à contenir les Q premières pages qui seront accédées. Les Q premiers accès à des pages distinctes n'entraînent donc pas de défauts de page comptabilisés dans F_{seq}.

Contrairement au résultat proposé dans [10], l'analyse suivante ne repose pas sur un algorithme d'ordonnancement particulier. L'encadrement du nombre de défauts de page du théorème 4 est basé sur la comparaison entre une exécution parallèle et l'exécution séquentielle de référence. Il dépend uniquement du nombre d'instructions `verrou.prendre()` du programme. On considère dans cette évaluation que si une tâche prend un verrou, elle doit le relâcher avant la fin de son exécution.

Lemme 1 *On considère deux exécutions séquentielles $E1$ et $E2$ d'un même programme sur deux processeurs identiques ($E1$ sur $P1$ et $E2$ sur $P2$) dont les mémoires locales sont initialement dans des états différents. Chaque mémoire peut contenir ou*

non des pages nécessaires au calcul. Soit $F1$ (resp. $F2$) le nombre de défauts de page effectués lors de l'exécution $E1$ (resp. $E2$) et Q le nombre de pages que peut contenir la mémoire d'un processeur. Si l'algorithme de remplacement de page LRU est utilisé, alors $|F_1 - F_2| \leq Q$.

Preuve. Soit p le nombre de pages différentes accédées par le programme. Si $p < Q$ l'inégalité est évidemment vérifiée. Si $p \geq Q$, on peut remarquer que dès que Q pages différentes ont été accédées, du fait de l'utilisation du protocole LRU, la mémoire est entièrement occupée par ces Q pages, indépendamment de l'état initial de la mémoire. A partir de cet instant, autant de défauts de page seront traités pendant les deux exécutions. On a donc bien l'inégalité $|F_1 - F_2| \leq Q$. □

Théorème 4 *Soit F_{par} le nombre de défauts de page lors d'une exécution parallèle pour laquelle une couche de mémoire partagée basée sur le protocole de cohérence paresseuse à la libération est utilisée. Soit V le nombre d'instructions* `verrou.prendre()` *du programme. Si les tâches du programme ne peuvent communiquer qu'en utilisant la mémoire partagée et que les accès à la mémoire partagée sont correctement synchronisés, alors on a : $F_{seq} - (V-1)Q \leq F_{par} \leq F_{seq} + VQ$*

Preuve. Considérons qu'un programme parallèle est composé de V blocs d'instructions $b_1, b_2, ..., b_V$ exécutés dans cet ordre lors d'une exécution séquentielle. Le découpage est effectué de manière à ce que la première instruction de chaque bloc soit l'acquisition d'un verrou et qu'aucune autre acquisition ne soit effectuée jusqu'à la fin du bloc. Les créations de tâches sont remplacées par les instructions du corps de la tâche. On note $f_1, f_2, ..., f_V$ le nombre de défauts de page intervenus au cours de l'exécution de chacun des blocs lorsque le programme est exécuté en séquentiel. Considérons le nombre de défauts de page engendrés par un bloc b_i lors d'une exécution parallèle. Selon le protocole de cohérence utilisé, l'acquisition du verrou a pour effet d'invalider les pages locales qui ont été modifiées à distance. La mémoire locale pouvant contenir Q pages, au plus Q pages peuvent être invalidées. Selon le lemme 1, on a $f_i^{par} \leq f_i + Q$ où f_i^{par} représente le nombre de défauts de page intervenus lors de l'exécution de b_i lorsque le programme est exécuté en parallèle. Si on fait la somme du nombre de défauts de page de chaque bloc du programme, on a $F_{par} \leq \sum_{i=1}^{v} f_i^{par} + VQ$ et donc $F_{par} \leq F_{seq} + VQ$.

Le même type de raisonnement peut être effectué pour la borne inférieure du théorème puisque le lemme 1 permet aussi de déduire l'inégalité $f_i - Q \leq f_i^{par}$. Cette inégalité est vérifiée pour les blocs $b_2, ..., b_V$. Pour le bloc b_1, on peut remarquer qu'il n'est pas possible de faire moins de défauts de pages en parallèle qu'en séquentiel puisque l'exécution séquentielle débute avec les Q premières pages disponibles en mémoire. Pour ce bloc, on a donc au mieux $f_1^{par} = f_1$. Si on fait la somme des résultats de chaque bloc du programme on obtient l'inégalité $\sum_{i=1}^{v} f_i^{par} - (V-1)Q \leq F_{par}$ et donc $F_{seq} - (V-1)Q \leq F_{par}$. □

La figure 3.2 donne deux exemples de programmes permettant d'atteindre les bornes de l'encadrement du théorème 4. L'exécution parallèle du premier programme permet d'atteindre la borne supérieure du théorème. On considère que les tâches sont

```
1  main() {
2    pour ( v de 1 à V/2 ) {
3      Tache Phase0();
4      Tache Phase1();
5    }
6  }
7
8  Phase0() {
9    prendre_un_verrou();
10   Accès P1, ... PQ+1;
11   relacher_un_verrou();
12 }
13
14 Phase1() {
15   prendre_un_verrou();
16   Accès PQ+1, ... P1;
17   relacher_un_verrou();
18 }
```

a)

```
1  main() {
2    pour ( v de 1 à V/(#proc*2) ) {
3      pour ( p de 1 à #proc )
4        Phase0(p);
5      pour ( p de 1 à #proc )
6        Phase1(p);
7    }
8  }
9
10 Phase0(int p) {
11   prendre_un_verrou();
12   Accès P1(p), ... PQ(p);
13   relacher_un_verrou();
14 }
15
16 Phase1(int p) {
17   prendre_un_verrou();
18   Accès PQ(p), ... , P1(p);
19   relacher_un_verrou();
20 }
```

b)

FIG. 3.2 – Exemples de programmes permettant d'atteindre la borne supérieure a) et inférieure b) sur le nombre de défauts de page lorsque le protocole de cohérence paresseuse à la libération est utilisé.

allouées de manière cyclique sur un nombre pair de processeurs. Dans ce programme, V tâches sont créées, alternativement de type `Phase0` et `Phase1`. La première instruction de chaque tâche correspond à la prise d'un verrou et la dernière au relâchement d'un verrou. On considère dans cet exemple que les données du problème occupent $Q+1$ pages soit 1 page de plus que peut en contenir la mémoire. Ces pages sont numérotées de P_1 à P_{Q+1}. La figure 3.3 donne la trace des exécutions séquentielle et parallèle pour lesquelles la borne supérieure du nombre de défauts de page est atteinte. Dans l'exécution séquentielle (partie a) de la figure), les pages présentes en mémoire sont accédées plusieurs fois avant d'être remplacées. La page $P2$ reste même en mémoire tout au long de l'exécution. Dans l'exécution parallèle (partie b) de la figure), le processeur 0 exécute les tâches `Phase0` et le processeur 1 exécute les tâches `Phase1`. Afin de se placer dans le pire des cas, on considère qu'aucune des données nécessaires aux calculs ne sont initialement installées dans la mémoire des processeurs. Une telle exécution parallèle est possible si les tâches `Phase0` et `Phase1` accèdent les pages P1,P2 et P3 en lecture seule. Au cours d'une telle exécution parallèle, chaque instruction `verrou.prendre()` implique l'invalidation de toutes les pages de la mémoire. Aucune page ne peut donc être réutilisée et on observe un défaut de page par accès. Dans ce cas on atteint bien la borne $F_{par} = F_{seq} + QV$.

La partie b) de la figure 3.2 montre un programme pour lequel la borne inférieure de l'encadrement du théorème 4 est atteinte. Pour l'exécution parallèle de ce

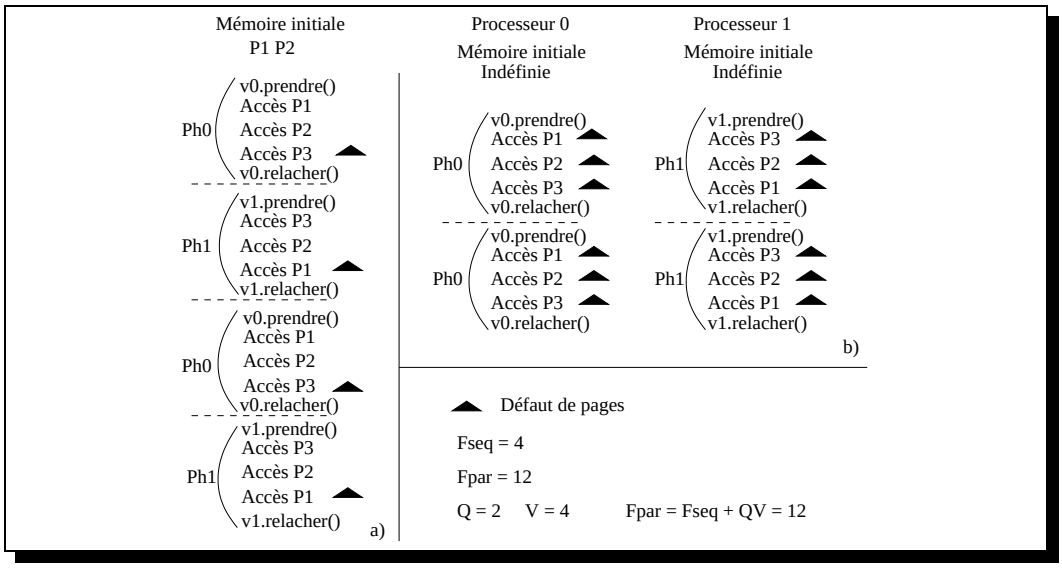


FIG. 3.3 – Exemple de trace d'exécution du programme de la figure 3.2 a) permettant d'atteindre la borne supérieure sur le nombre de défauts de page. La partie a) montre la trace d'une exécution séquentielle et la partie b) celle d'une exécution parallèle sur deux processeurs.

programme, on considère que les tâches sont allouées de manière cyclique sur $\#proc$ processeurs. Les données du programme sont allouées dans $p \times Q$ pages. La mémoire de chaque processeur p est initialisée avec les pages numérotées de $P_1(p), ...P_Q(p)$ sauf lors de l'exécution séquentielle où la mémoire du processeur est initialisée avec les pages $P_1(0), ..., P_Q(0)$ qui sont les Q premières pages accédées par le programme. La trace de l'exécution séquentielle du programme (figure 3.4 a)) montre qu'après les Q premiers accès, chaque nouvel accès entraîne un défaut de page. Lors de l'exécution parallèle (figure 3.4 b)), les données nécessaires à l'exécution des tâches sont toujours en mémoire. Aucun défaut de page n'intervient donc au cours de cette exécution. Pour cette trace, on a bien l'égalité $F_{par} = F_{seq} - Q(V-1)$.

3.1.2 Cohérence graphe de précédence (*DAG-consistency*)

Présentation du protocole

Comme celles de TreadMarks et SCIOS, la couche de mémoire partagée utilisée dans Distributed Cilk est implantée avec un mécanisme distribué de pagination. Cependant, le protocole de cohérence utilisé n'est pas basé sur des synchronisations à base de verrous, mais sur les spécificités du modèle strict réduit et de la représentation des programmes sous forme de graphe de précédence. Le protocole de cohérence utilisé est appelé cohérence DAG (*Direct Acyclic Graph*) [10] ou cohérence graphe de précédence. L'idée intuitive de ce protocole est qu'une lecture ne voit une écriture

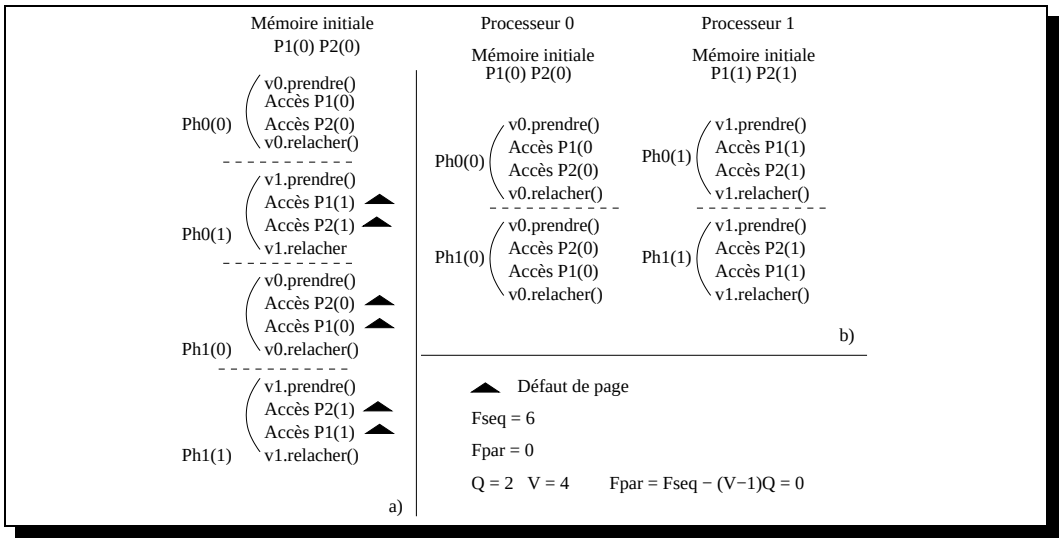


FIG. 3.4 – Exemple de trace d'exécution du programme de la figure 3.2 b) permettant d'atteindre la borne inférieure sur le nombre de défauts de page. La partie a) montre la trace d'une exécution séquentielle et la partie b) celle d'une exécution parallèle sur deux processeurs.

que s'il existe un chemin dans le graphe de précédence tel que l'écriture précède la lecture.

Afin de donner une définition plus formelle de ce protocole, les notations suivantes seront utilisées. Soit $G = (V, E)$ le graphe de précédence associé à un calcul. Soit deux tâches $i, j \in V$; s'il existe un chemin de longueur non nulle de i à j dans G, on dit que i précède j et on note $i \prec j$. Si par contre on a $i \not\prec j$, $j \not\prec i$ et $i \neq j$ alors on dit que i et j sont incomparables. Afin de tracer quelle tâche est responsable de la valeur d'un objet, on imagine que chaque objet partagé est marqué avec l'identificateur de la tâche ayant effectué la dernière écriture.

Définition 2 *Pour un programme parallèle représenté par le graphe $G = (V, E)$, la mémoire partagée M respecte la cohérence DAG si les deux conditions suivantes sont vérifiées :*

1. *Lorsqu'une tâche $i \in V$ lit un objet $m \in M$, elle reçoit une valeur v marquée par une tâche $j \in V$ telle que j a donné la valeur v à m et $i \not\prec j$*
2. *Soient trois tâches $i, j, k \in V$ telles que $i \prec j \prec k$; si j écrit sur un objet $m \in M$ et que k lit m, alors la valeur lue par k n'est pas marquée par i.*

Pour implanter ce protocole, on considère que les pages peuvent être dans l'état *modifiée* ou *non modifiée*. A chaque page est associée un processeur maître et les opérations autorisées pour maintenir la cohérence sont :

- **Mettre-à-jour** Les modifications apportées à la copie locale sont mises à jour sur le site maître de la page.
- **Ramener** La copie "à jour" d'une page est installée dans la mémoire locale. Pour cela, un emplacement mémoire doit être disponible. L'opération **Supprimer** doit préalablement être utilisée.
- **Supprimer** Une page de la mémoire locale est supprimée. Le protocole LRU est utilisé pour choisir la page à supprimer.

L'algorithme de BACKER [10] est utilisé pour implanter ce protocole de cohérence. Pour chaque arrête $i \rightarrow j$ du graphe de précédence, si les tâches i et j sont ordonnancées respectivement sur les processeurs p et q avec $p \neq q$, toutes les pages de p sont mises à jour après l'exécution de i et toutes les pages de q sont mises à jour puis supprimées de la mémoire de q. Il est à noter que les mises à jour du processeur p sont effectuées avant l'activation de la tâche j.

Evaluation

Une évaluation du nombre de défauts de page lors d'une exécution d'un programme Cilk en utilisant le protocole de cohérence DAG est proposée dans [10]. Cette borne est donc associée au modèle de programmation strict réduit et l'algorithme de vol de travail du langage Cilk est utilisé. Le théorème suivant rappelle ce résultat.

Théorème 5 *Soit F_{par} le nombre de défauts de pages d'un programme Cilk s'exécutant sur P processeurs ayant chacun une mémoire pouvant contenir Q pages. Soit F_{seq} le nombre de défauts de pages du même programme lors d'une exécution séquentielle. Si le protocole de cohérence DAG est utilisé, on a $F_{par} \leq F_{seq} + 2Qs$ avec s qui représente le nombre de vol de tâches intervenus au cours de l'exécution parallèle du programme.*

Cette borne est donc très liée au nombre de vols effectués lors de l'exécution parallèle du programme. Dans [9], l'auteur montre que le nombre de vols lors de l'exécution d'un programme Cilk est de l'ordre de $O(pT_\infty)$ où p représente le nombre de processeurs et T_∞ représente le temps d'exécution du programme si une infinité de processeurs sont disponibles.

Nous proposons d'étendre ce résultat au modèle de programmation déterministe non contraint et de proposer un encadrement du nombre de défauts de page des programmes parallèles. On considère pour cela que toute nouvelle tâche s'exécute par défaut sur le processeur qui l'a créée. La tâche principale (`main` du programme) est exécutée sur le processeur 0. L'algorithme d'ordonnancement utilisé peut décider à tout moment de remettre en cause le site par défaut d'une tâche non encore exécutée en l'exportant sur un autre processeur. Une tâche ne peut cependant pas être migrée si elle est en cours d'exécution. Si aucune tâche n'est exportée (exécution séquentielle), les tâches sont exécutées en profondeur d'abord. Cet ordre d'exécution est

noté o_{ref} dans la suite. On considère lors de cette exécution de référence que les Q premières pages accédées par le programme sont initialement installées en mémoire.

Le lemme suivant, corollaire du lemme 1, permet de borner le nombre de défauts de page liés à l'exportation d'une tâche.

Lemme 2 *Considérons 3 tâches T_1, T_2 et T_3. Soit F_1 le nombre de défauts de page engendrés lorsque ces trois tâches s'exécutent dans cet ordre sur P_1. Soit F_2 le nombre de défauts de page engendrés lorsque T_1 et T_3 s'exécutent sur P_1 et que T_2 est exportée sur P_2. Alors on a $F_1 - 2Q \leq F_2 \leq F_1 + 2Q$.*

Preuve. Le corollaire est une conséquence immédiate du lemme 1, puisque l'exécution de T_2 sur P_2 peut faire varier le nombre de défauts de page de plus ou moins Q, de même que l'exécution de T_3 si T_2 n'est pas exécutée juste avant elle. □

Le théorème 6 donne un encadrement du nombre de défauts de page d'un programme parallèle lorsque le protocole de cohérence DAG est utilisé. La preuve de ce théorème est basé sur le lemme 2.

Théorème 6 *Soit F_1 le nombre de défauts de page dans l'exécution séquentielle respectant l'ordre de référence o_{ref}. Soit F_p, le nombre de défauts de page dans une exécution du programme sur p processeurs. Si l'ordonnancement O_p utilisé respecte localement l'ordre de référence o_{ref}, et que le protocole de cohérence DAG est utilisé, alors on a*

$$F_1 - 2Qe \leq F_p \leq F_1 + 2Qe \qquad (1)$$

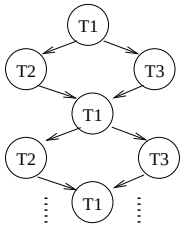
où e désigne le nombre de tâches exportées dans l'ordonnancement O_p.

Preuve. La preuve s'effectue par récurrence sur e.

• Si $e = 0$, le programme démarrant sur un unique processeur, la totalité du programme sera exécutée sur ce processeur. On a dans ce cas $F_p = F_1$ et l'inégalité (1) est vérifiée.

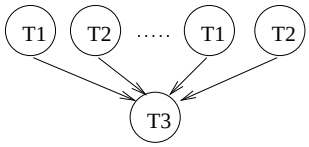
• Si $e \neq 0$, soit e_i la tâche exportée dont la date de création est maximale et P_i son processeur d'exécution. Notons E_i l'ensemble de tâches contenant e_i et les tâches qu'elle a engendrées. Comme e_i est la tâche exportée dont la date de création est maximale et que O_p respecte localement l'ordre de référence, on en déduit que toutes les tâches engendrées par e_i sont exécutées par P_i.

Construisons à partir de O_p un autre ordonnancement O'_p qui respecte aussi sur chaque processeur l'ordre de référence ; O'_p est identique à O_p, si ce n'est que toutes les tâches de E_i sont exécutées dans O'_p sur le processeur P_0 qui a créé e_i. Comme l'ordre de création est respecté sur P_0, toutes les tâches de E_i sont exécutées dans O'_p successivement et sans préemption sur P_0, selon le même ordre que dans l'ordonnancement O_p sur P_i. Notons qu'il est possible d'avoir des tâches qui s'exécutent sur P_0 après les tâches de E_i dans l'ordonnancement O'_p. D'après le lemme 2, le changement de processeur des tâches E_i peut faire varier le nombre de défauts de page de l'exécution du programme suivant O_p de plus ou moins $2Q$. On a donc l'inégalité :




```
1  main(){
2    faire t fois {
3      T1(d1, d2, d3, d4);
4      T2(d1, d3);
5      T3(d2, d4);
6    }
7    T1(d1, d2, d3, d4);
8  }
9
10 T1(d1, d2, d3, d4) {
11   Accès d1/P1; Accès d2/P1;
12   Accès d3/P2; Accès d4/P2;
13 }
14
15 T2(d1, d3) {
16   Accès d1/P1; Accès d3/P2;
17 }
18
19 T3(d2, d4) {
20   Accès d2/P1; Accès d4/P2;
21 }
```

a)




```
1  main() {
2    faire t fois {
3       T1(d1,d2);
4       T2(d3,d4);
5    }
6    T3(d1,d2,d3,d4);
7  }
8
9  T3(d1,d2,d3,d4) {
10   Accès d1/P1; Accès d2/P2;
11   Accès d3/P3; Accès d4/P4;
12 }
13
14 T1(d1,d2) {
15   Accès d1/P1; Accès d2/P2;
16 }
17
18 T2(d3,d4) {
19   Accès d3/P3; Accès d4/P4;
20 }
```

b)

FIG. 3.5 – Exemples de programmes permettant d'atteindre la borne supérieure a) et inférieure b) sur le nombre de défauts de page lorsque le protocole de cohérence DAG est utilisé.

$$F_{O'_p} - 2Q \leq F_p \leq F_{O'_p} + 2Q$$

Or, par construction, l'ordonnancement O'_p contient $e-1$ tâches migrantes. D'après l'hypothèse de récursion, on a donc pour cet ordonnancement :

$$F_1 - 2Q(e-1) \leq F_p \leq F_1 + 2Q(e-1)$$

D'où l'encadrement : $F_1 - 2Qe \leq F_p \leq F_1 + 2Qe$. □

La figure 3.5 montre que les bornes du théorème 6 sont atteignables. La partie a) montre un programme permettant d'atteindre la borne supérieure ainsi que le graphe de précédence associé. Pour chaque tâche du programme, la donnée d_i accédée ainsi que la page P_i à laquelle elle appartient sont spécifiées. La figure 3.6 donne les traces

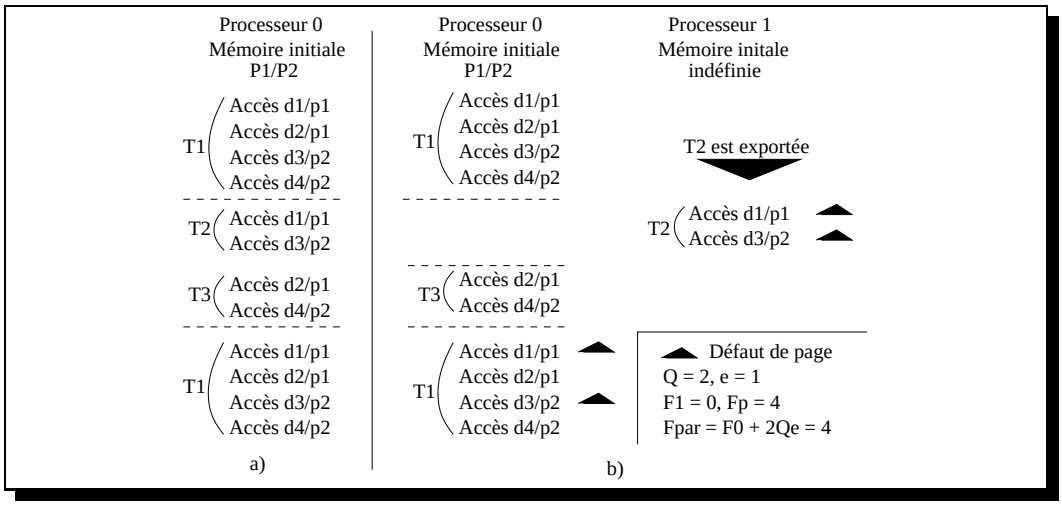


FIG. 3.6 – Exemple de trace d'exécution du programme de la figure 3.5 a). La partie a) donne la trace d'une exécution séquentielle et la partie b) celle d'une exécution parallèle sur deux processeurs.

des exécutions séquentielle et parallèle sur deux processeurs de ce programme. Ces traces permettent de vérifier que la borne $F_p = F_1 + 2Qe$ est atteignable.

La partie b) de la figure 3.5 montre un programme permettant d'atteindre la borne inférieure sur le nombre de défauts de page. La figure 3.7 donne les traces d'exécutions de ce programme pour lesquelles la borne est atteinte. Il est à noter que pour atteindre cette borne, la mémoire du processeur 0 est initialisée avec les pages $p1$ et $p2$ et que celle du processeur 1 est initialisée avec les pages $p3$ et $p4$. Les tâches de type $T2$ qui accèdent $p3$ et $p4$ sont exportées sur le processeur 1 pour obtenir une bonne localité. Dans ce cas, la borne $F_p = F_1 - 2Qe$ est aussi atteignable.

3.1.3 Cohérence flot de données

Présentation du protocole

A la différence de la cohérence DAG, la cohérence flot de données est basée sur la représentation des programmes sous forme de graphe de flot de données (chapitre 1 section 1.1). Dans ce protocole, l'écriture d'une donnée n'a pas besoin d'être visible par tous les successeurs dans le graphe, mais seulement par ceux qui lisent cette donnée.

Afin de définir plus formellement le protocole, les notations suivantes sont introduites. Soit $G = (D, T, A)$, le graphe de flot de donnée de l'application considérée. D représente l'ensemble des noeuds "donnée" du graphe, T représente l'ensemble des noeuds "tâches" et A l'ensemble des arêtes. Soit $n \in D \cup T$ et $n' \in D \cup T$. S'il existe un chemin dans G reliant n à n', on dit que n précède n' et on note $n \prec n'$. Pour

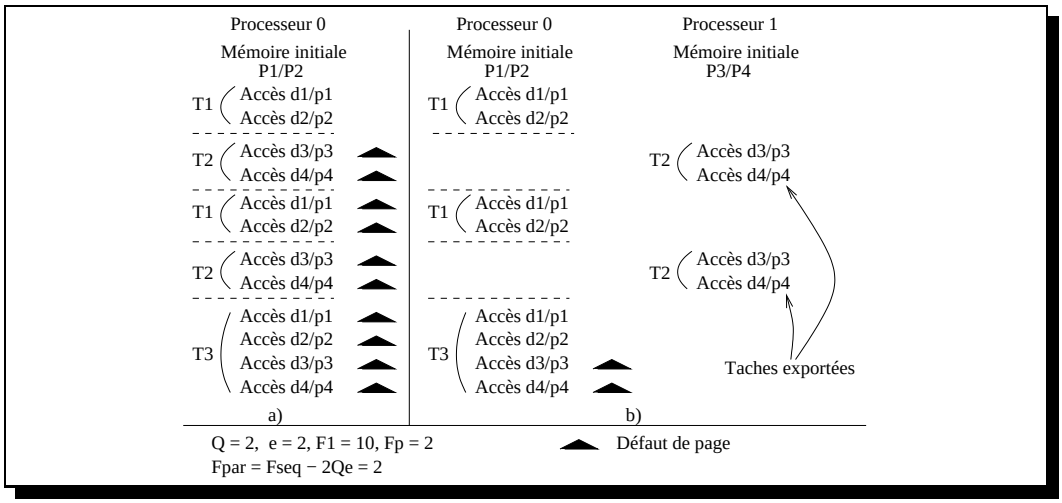


FIG. 3.7 – Exemple de trace d'exécution du programme de la figure 3.5 b). La partie a) donne la trace d'une exécution séquentielle et la partie b) celle d'une exécution parallèle sur deux processeurs.

qu'une tâche $t \in T$ puisse lire une donnée $d \in D$, il faut que $d \prec t$ et que $\nexists n \in D \cup T$ tel que $d \prec n \prec t$.

La figure 3.8 illustre la différence entre la cohérence DAG et la cohérence flot de données. Dans cet exemple, la donnée D1 est allouée dans la page P1 et les données D2 et D3 dans la page P2. Dans le protocole de cohérence flot de données, seule la page P1 est invalidée à la fin de l'exécution de T1 puisque T2 ne lit que D1. Dans le protocole de cohérence DAG, T2 voit toutes les écritures de T1 donc P1 et P2 sont invalidées.

Ce protocole peut être implanté de deux manières. La première consiste à utiliser un mécanisme de pagination similaire à *Distributed Cilk*. On considère dans ce cas que les pages peuvent être dans l'état *Modifiée* ou *Non modifiée* et que les opérations *Mettre-à-jour*, *Ramener* et *Supprimer* sont disponibles pour maintenir la cohérence. La sémantique de ces opérations est la même que pour le protocole de cohérence DAG. Soit $t \in T$ et $\{d_1, ..., d_k\} \in D$ les k données produites par t ($\{d_1, ..., d_k\}$ sont donc des successeurs immédiats de t dans G). Soit p le processeur d'exécution de t. Lorsque t termine son exécution, l'opération *Mettre-à-jour* est utilisée pour chacune des pages associées aux données $\{d_1, ..., d_k\}$. Soit $\{t_1, ..., t_l\} \in T$ l'ensemble des successeurs immédiats de $\{d_1, ..., d_k\}$. Soit $\{p_1, ..., p_m\}$ l'ensemble des processeurs exécutant au moins une tâche de l'ensemble $\{t_1, ..., t_l\}$. Sur chaque processeur, avant l'exécution de la première tâche appartenant à $\{t_1, ..., t_l\}$, l'opération *Mettre-à-jour* est utilisée sur chacune des pages associées aux données $\{d_1, ..., d_k\}$. Ces pages sont ensuite supprimées de la mémoire locale.

La connaissance du graphe de flot de données permet d'envisager une autre implantation qui n'utilise pas de mécanisme de pagination. Lorsqu'une tâche termine

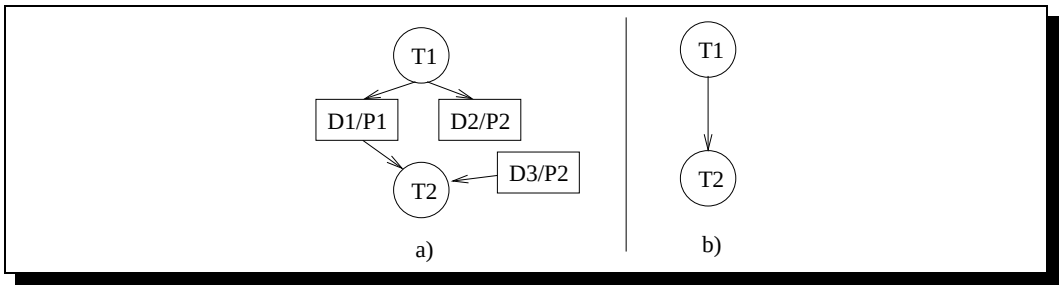


FIG. 3.8 – Comparaison cohérence DAG, cohérence flot de données. L'écriture de D2 est vue par T2 dans la cohérence DAG b) et pas dans la cohérence flot de données a).

son exécution, on sait exactement quelles données doivent être envoyées aux successeurs. On peut dans ce cas utiliser des primitives d'envoi et réception de messages pour envoyer les objets concernés.

L'évaluation suivante est valable pour les deux implantations. Pour cela, on peut considérer que la deuxième implantation revient à implanter une mémoire partagée sans faux partage. Plusieurs pages peuvent être utilisées pour allouer une donnée, mais une page contient au plus une donnée.

Evaluation

La connaissance des données partagées accédées par les tâches permet de borner un peu plus finement le nombre de défauts de page. Nous proposons pour cela de considérer le même ordonnancement que dans la preuve précédente et de se baser sur le même ordre séquentiel de référence o_{ref}. Chaque tâche s'exécute par défaut sur le même processeur que la tâche qui l'a créée. L'ordonnancement peut à tout moment remettre en cause cet ordonnancement en exportant une ou plusieurs tâches du calcul. La tâche principale est exécutée sur le processeur 0.

Dans la suite, on note k_{max} le nombre maximal de pages accédées par une tâche d'un graphe $G = (T, D, A)$. Cette grandeur est aisément calculable à partir du graphe de flot de données. Soit d_{in} l'ensemble des noeuds "donnée" qui précèdent immédiatement une tâche t. Soit d_{out} l'ensemble des noeuds "donnée" qui succèdent immédiatement la même tâche t. Le nombre de pages accédées par t correspond à la grandeur :

$$k_t = \sum_{d \in d_{in}} \frac{\text{taille(d)}}{\text{TaillePage}} + \sum_{d' \in d_{out}} \frac{\text{taille(d')}}{\text{TaillePage}}$$

La grandeur k_{max} correspond à la valeur maximale que peut prendre k_t pour $t \in T$. Les lemmes suivants reprennent les lemmes 1 et 2 en tenant compte des données accédées par la tâche exportée.

Lemme 3 *On considère deux exécutions séquentielles $E1$ et $E2$ d'une même tâche sur deux processeurs identiques ($E1$ sur $P1$ et $E2$ sur $P2$) dont les mémoires locales sont initialement dans des états différents. Chaque mémoire peut contenir ou non des pages nécessaires au calcul. Soit $F1$ (resp. $F2$) le nombre de défauts de page effectués lors de l'exécution $E1$ (resp. $E2$) et Q le nombre de pages que peut contenir la mémoire d'un processeur. Si le protocole de cohérence flot de données et l'algorithme de remplacement de page LRU sont utilisés, alors $|F_1 - F_2| \leq min(Q, k_{max})$.*

Preuve. Si $Q < k_{max}$, on se trouve dans le même cas que pour le lemme 1. Si $k_{max} < Q$, les k_{max} pages tiennent en mémoire, il ne peut donc pas y avoir plus de k_{max} défauts de page lors de l'exécution de la tâche.□

Lemme 4 *Considérons le graphe $G = (T, D, A)$ et trois tâches T_1, T_2 et T_3 dans T avec $T_1 \prec T_2 \prec T_3$. Soit F_1 le nombre de défauts de page engendrés lorsque ces trois tâches s'exécutent dans cet ordre sur P_1. Soit F_2 le nombre de défauts de page engendrés lorsque T_1 et T_3 s'exécutent sur P_1 et que T_2 est exportée sur P_2.*

Alors on a $F_1 - 2\min(Q, k_{max}) \leq F_2 \leq F_1 + 2\min(Q, k_{max})$.

Preuve. Le corollaire est une conséquence immédiate du lemme 3, puisque l'exécution de T_2 sur P_2 peut faire varier le nombre de défauts de page de plus ou moins $\min(Q, k_{max})$, de même que l'exécution de T_3 si T_2 n'est pas exécutée juste avant elle. □

Théorème 7 *Soit F_1 le nombre de défauts de page dans l'exécution séquentielle respectant l'ordre de référence o_{ref}. Soit F_p, le nombre de défauts de page dans une exécution du programme sur p processeurs. Si l'ordonnancement O_p utilisé respecte localement l'ordre de référence o_{ref} et que le protocole de cohérence flot de données est utilisé, alors on a :*

$$F_1 - 2\min(Q, k_{max})e \leq F_p \leq F_1 + 2\min(Q, k_{max})e \qquad (1)$$

où e désigne le nombre de tâches exportées dans l'ordonnancement O_p.

Preuve. Comme pour le théorème 6, la preuve s'effectue par récurrence sur e.

• Si $e = 0$, le programme démarrant sur un unique processeur, la totalité du programme sera exécutée sur ce processeur. On a dans ce cas $F_p = F_1$ et l'inégalité (1) est vérifiée.

• Si $e \neq 0$, soit e_i la tâche exportée dont la date de création est maximale et P_i son processeur d'exécution. Notons E_i l'ensemble de tâches contenant e_i et les tâches qu'elle a engendrées. Comme e_i est la tâche exportée dont la date de création est maximale et que O_p respecte localement l'ordre de référence, on en déduit que toutes les tâches engendrées par e_i sont exécutées par P_i.

Construisons à partir de O_p un autre ordonnancement O'_p qui respecte aussi sur chaque processeur l'ordre de référence ; O'_p est identique à O_p, si ce n'est que toutes les tâches de E_i sont exécutées dans O'_p sur le processeur P_0 qui a crée e_i. Comme l'ordre de création est respecté sur P_0, toutes les tâches de E_i sont exécutées dans

O'_p successivement et sans préemption sur P_0, selon le même ordre que dans l'ordonnancement O_p sur P_i. Notons qu'il est possible d'avoir des tâches qui s'exécutent sur P_0 après les tâches de E_i dans l'ordonnancement O'_p. D'après le lemme 4, le changement de processeur des tâches E_i peut faire varier le nombre de défauts de page de l'exécution du programme suivant O_p de plus ou moins $2\min(Q, k_{max})$. On a donc l'inégalité :

$$F_{O'_p} - 2\min(Q, k_{max}) \leq F_p \leq F_{O'_p} + 2\min(Q, k_{max})$$

Or, par construction l'ordonnancement O'_p contient $e-1$ tâches migrantes. D'après l'hypothèse de récursion, on a donc pour cet ordonnancement :

$$F_1 - 2\min(Q, k_{max})(e-1) \leq F_p \leq F_1 + 2\min(Q, k_{max})(e-1)$$

D'où : $F_1 - 2\min(Q, k_{max})e \leq F_p \leq F_1 + 2\min(Q, k_{max})e$. □

Lorsqu'un algorithme de vol de travail est utilisé, le résultat du théorème 7 dépend étroitement du nombre de vols intervenus lors d'une exécution parallèle. Un algorithme d'ordonnancement permettant de borner le nombre de vols est présenté dans la section 5.1.

D'autre part, on peut remarquer que de nombreuses interfaces de programmation (Cilk, TreadMarks,...) ne permettent pas de prédire quelles données seront accédées par chaque tâche. L'implantation du protocole flot de données nécessite donc l'utilisation d'une interface de programmation permettant de déduire les échanges de données entre les tâches d'un programme.

3.2 Machine abstraite pour la construction du flot de données

Nous proposons dans cette section de définir une machine abstraite composée d'un ensemble d'opérateurs permettant la description du flot de données. Bien qu'inspirée de langages de programmation "flot de données" comme Athapascan ou Jade, cette interface se veut indépendante de tout langage. Les mécanismes présentés dans le chapitre suivant pouvant être implantés dans tout langage permettant de décrire le flot de données, cette machine abstraite sera utilisée pour leur description.

De la même manière que dans Jade [56] et Athapascan [27, 58], la description des programmes basée sur le modèle présenté précédemment repose sur trois concepts essentiels : tâches, objets partagés et spécification d'accès. Une tâche représente un ensemble d'instructions consécutives pouvant être exécutées sans interruption. Chaque tâche peut donc être exécutée de manière non préemptive sur un seul processeur. L'exécution d'une tâche revient à exécuter une fonction pour laquelle aucun effet de bord n'est autorisé. Chaque objet accédé par la tâche doit donc être explicitement passé en paramètre de celle-ci. Au cours de l'exécution, on considère que les données produites par une tâche ne sont disponibles qu'à la fin de son exécution. Par ailleurs toute tâche est autorisée à créer d'autres tâches récursivement.

```
1   ...
2   int premier_tableau[10];
3   int deuxième_tableau[10];
4   ...
5   Tache t;
6   t = AllocationCloture( nom_fonction ) ;
7   t.empiler Lecture ( premier_tableau ) ;
8   t.empiler Ecriture-différée( deuxième_tableau[0..4] ) ;
9   t.FermerContexte() ;
10  ...
```

FIG. 3.9 – Exemple d'utilisation de la machine abstraite

Les synchronisations entre tâches sont détectées automatiquement à l'aide d'objets partagés à assignation unique. Chaque tâche déclare les objets partagés auxquelles elle accède et spécifie l'accès (lecture ou écriture) qu'elle et ses descendantes effectueront sur cette donnée au cours de leurs exécutions. Une tâche peut aussi déclarer des accès différés (lecture différée, écriture différée) sur des objets partagés. Ce type de spécification est utile lorsqu'une tâche n'a pas besoin d'accéder elle-même à une donnée partagée mais doit créer des tâches qui y accèdent. Dans ce cas, la tâche mère n'a pas besoin de se synchroniser sur la donnée partagée. Cette synchronisation est "transférée" aux tâche filles qui effectueront réellement l'accès.

La création d'une tâche est opérée en allouant un bloc de mémoire appelée "clôture" de la tâche permettant de fermer le contexte d'exécution de la tâche. Une "clôture" contient donc l'ensemble des instructions qu'elle doit exécuter ainsi que les paramètres dont elle a besoin. L'allocation d'une "clôture" est effectuée à l'aide de l'instruction `AllocationCloture`. Chaque paramètre de la tâche est ensuite inséré dans la clôture à l'aide de l'instruction `empiler`. Pour chacune de ces insertions, le type d'accès est spécifié : `Lecture[différée]` ou `Écriture[différée]`. Les données empilées de cette manière dans une clôture peuvent être de type scalaire ou de type tableau. La déclaration d'un tableau est effectuée à l'aide de l'instruction `type nom_du_tableau[nombre_d'elements]` et un élément du tableau peut être accédé à l'aide de l'instruction `nom_du_tableau[indice]`. Par ailleurs, il est possible d'utiliser un sous-tableau d'éléments contigus d'un tableau. Pour cela, la notation `[indice_debut..indice_fin]` est utilisée. Une fois tous les paramètres empilés dans la clôture, l'instruction `FermerContexte` permet de fermer le contexte de celle-ci. La figure 3.9 illustre brièvement l'utilisation de l'ensemble des instructions présentées ci-dessus. Dans ce programme, deux tableaux contenant dix entiers sont tout d'abord déclarés (lignes 2 et 3). Une tâche nommée t est ensuite déclarée et une clôture associée à cette tâche est allouée (lignes 5 et 6). Deux paramètres sont ensuite insérés dans la clôture (lignes 7 et 8). Le premier est inséré en mode lecture et le deuxième en écriture différée. Il est à noter que la tâche t et ses descendantes ne produiront

que les 5 premiers éléments du tableau `deuxième_tableau`.

Les synchronisations entre tâches sont déduites de la spécification des accès effectués par les tâches sur les objets de la mémoire partagée. La sémantique est lexicographiquement ordonnée par ';', c'est-à-dire que toute lecture voit la dernière écriture dans l'ordre lexicographique. La figure 3.10 illustre cette sémantique avec l'exemple du calcul du n^{ieme} terme de la suite de Fibonacci. Dans cet exemple, la tâche *som* voit la dernière écriture dans l'ordre lexicographique de la donnée $r1$ (respectivement $r2$), c'est-à-dire la valeur produite par la tâche $fibo(n-1)$ (respectivement $fibo(n-2)$).

Même si la sémantique associée à cette machine abstraite est lexicographique, une exécution non-préemptive des tâches est autorisée. Lorsqu'une tâche commence son exécution, elle peut donc s'exécuter totalement sans nécessiter de synchronisations. Cette exécution est assez proche de l'ordre d'exécution séquentiel habituel dans lequel le flot d'exécution est transmis de la procédure mère à la procédure fille lors d'un appel de procédure. Dans cet ordre, appelé *ordre de référence*, lorsqu'une tâche est créée, la tâche mère conserve le flot d'exécution jusqu'à sa dernière instruction. Celui-ci est ensuite transmis à la première tâche fille créée.

L'avantage d'un tel ordre d'exécution est de permettre une description du parallèlisme avant son exécution. En effet, cet ordre permet de disposer à la fin de l'exécution d'une tâche, de la description de l'ensemble des tâches générées et donc du parallélisme potentiel entre ces tâches. S'il est nécessaire d'exporter un tâche pour palier à l'inactivité d'un processeur, cette description pourra être utilisés pour choisir la tâche à exporter.

Cet ordre peut être défini formellement de la manière suivante :
Soit s, une séquence d'instructions éventuellement vide ne contenant aucune création de tâches et T, la séquence d'instructions permettant la création d'une tâche. Une trace d'exécution d'un ensemble de tâches en profondeur d'abord peut être représentée par $f = s_1T_1s_2T_2...s_{n-1}F_{n-1}s_n$. L'indépendance entre la tâche créatrice et les tâches créées implique que cette trace est sémantiquement équivalente à la trace $f' = s_1s_2...s_{n-1}s_nT_1T_2...T_{n-1}$. Cet ordre d'exécution est noté R dans la suite.

3.3 Conclusions

Dans ce chapitre, nous avons évalué trois protocoles de cohérence mémoire pour l'implantation distribuée d'une mémoire partagée. Nous avons vu que le protocole de cohérence à la libération paresseuse était dépendant du nombre de prises de verrous du programme. Si on considère que chaque tâche effectue au moins un accès, le nombre de prises de verrous est supérieur ou égal au nombre de tâches du programme. Le nombre de défauts de page si ce type de cohérence est utilisé est donc potentiellement grand. Dans le but d'implanter une couche de mémoire partagée plus efficace, Distributed Cilk a proposé le protocole de cohérence DAG. Dans [10], il est montré que le nombre de défauts de page d'un programme Cilk dépend du nombre de

```
1  procedure Somme(Donnee a, Donnee b, Donnee r) {
2     r = a + b;
3  }
4
5  procedure Fibo(int n, Donnee r) {
6     if (n <2) r = n ;
7     else {
8        int r1, r2;
9        Tache f1 = AllocationCloture( Fibo );
10         f1.empiler::Lecture (n-1);
11         f1.empiler::WriteAccess (r1);
12         f1.FermerContexte();
13       Tache f2 = AllocationCloture ( Fibo );
14         f2.empiler::Lecture (n-2);
15         f2.empiler::WriteAccess (r2);
16         f2.FermerContexte();
17       Tache som = AllocationCloture ( Somme );
18         som.empiler::WriteAccess (r);
19         som.empiler::Lecture  (r1);
20         som.empiler::Lecture  (r2);
21         som.FermerContexte();
22     }
23 }
```

FIG. 3.10 – Exemple de calcul du n^{ieme} terme de la suite de fibonacci.

vols de tâche effectués par l'ordonnancement, lorsque ce protocole est utilisé. Nous avons montré dans ce chapitre que ce résultat pouvait être étendu aux graphes de précédence associés au modèle déterminisite non contraint. Enfin, nous avons montré que le protocole de cohérence "flot de données" pouvait être implanté efficacement et qu'une borne sur le nombre de défauts de page similaire à la cohérence DAG pouvait être calculée. Cette borne est dépendante du nombre de tâches exportées et de la construction dynamique du graphe de flot de données de l'application. Parmi les ordonnancements considérés dans le chapitre 5 nous proposerons un algorithme permettant de majorer le nombre de tâches exportées. Cet ordonnancement tire donc partie du protocole de cohérence "flot de données" détaillé dans ce chapitre.

Dans le chapitre suivant, nous décrivons une implantation possible d'une mémoire partagée basée sur le protocole de cohérence "flot de données". Pour cela, on considère que les programmes sont écrits en utilisant les instructions de la machine abstraite que nous avons présentées dans ce chapitre.

CHAPITRE 4

Chaînage des accès dans une pile distribuée

Dans ce chapitre, nous décrivons un mécanisme de pile cactus étendue dont l'objectif est de permettre l'implantation efficace d'ordonnancements distribués. Ce mécanisme est basé sur le couplage d'un mécanisme d'allocation à l'aide d'une pile et d'une gestion distribuée du flot de données. L'utilisation d'une pile permet d'optimiser les allocations mémoire en réduisant leur coût à celui d'une incrémentation de pointeur. La gestion du flot de données permet quant à elle d'implanter le protocole mémoire flot de données. Cette implantation est basée sur la machine abstraite du chapitre précédent. Nous présentons dans un premier temps le fonctionnement de la pile en détaillant l'allocation des tâches et le chaînage permettant la gestion du flot de données. Nous détaillons ensuite le fonctionnement de la pile lorsqu'un algorithme d'ordonnancement basé sur l'exportation de tâches est utilisé. Bien que ce mécanisme soit optimisé pour cet ordonnancement, nous montrerons comment il peut aussi être utilisé lorsqu'un algorithme d'ordonnancement statique est utilisé. Enfin, nous discuterons du couplage de stratégies d'ordonnancement statiques et dynamiques dans la dernière section de ce chapitre.

4.1 Construction du graphe de flot de données

Dans cette section, nous présentons les mécanismes d'allocation et de construction dynamique du graphe de flot de données basés sur la machine abstraite présentée dans le chapitre précédent. Ce mécanisme a pour objectif de permettre l'exécution des tâches avec un surcoût aussi faible que possible lorsque celles-ci sont exécutées de manière séquentielle, tout en permettant l'exécution distribuée à grande échelle des programmes. Comme dans [21], nous proposons d'utiliser une structure de données appelée "clôture" pour stocker le graphe de flot de données. Une clôture représente l'exécution future d'une tâche. Elle est constituée de la fonction à exécuter par la tâche et de l'ensemble de ses paramètres. Chaque paramètre est "représenté" dans la clôture à l'aide d'une structure de donnée appelée "accès". Le graphe est construit en chaînant les "accès" à une même donnée partagée dans l'ordre de référence.

Dans cette section, nous détaillons dans un premier temps le mécanisme de pile permettant l'allocation des "clôtures" du graphe, avant de détailler le chaînage des "accès" permettant de calculer le flot de données d'une application.

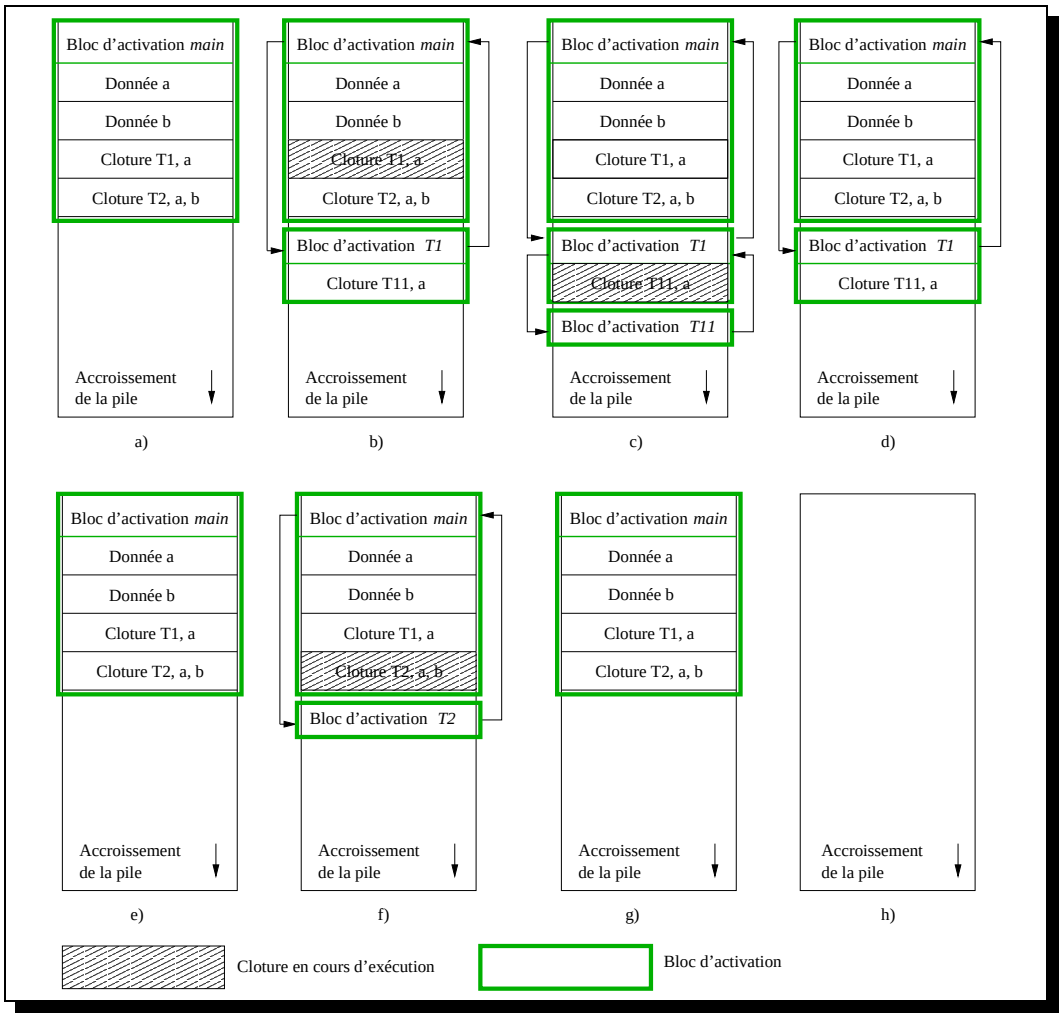


FIG. 4.1 – Evolution de la pile lors de l'exécution du programme de la figure 4.3

4.1.1 Allocation des clôtures dans une pile

Plusieurs solutions peuvent être envisagées pour l'allocation de clôtures du graphe. La première consiste à utiliser l'allocateur standard du langage de programmation utilisé (les opérateurs `new` et `delete` du langage C++ dans le cas d'Athapascan). La "réentrance" de ces opérateurs est généralement garantie par les langages. Cependant, lorsqu'une machine mutliprocesseurs à mémoire partagée est utilisée, une contention non négligeable sur les sections critiques liées à ces opérateurs peut intervenir principalement à grain fin. De plus, ces contentions peuvent intervenir même si tous les processeurs de la machine travaillent sur des parties disjointes du graphe.

Afin d'éviter ces contentions, nous proposons d'allouer les clôtures du graphe de flot de données dans un ensemble de piles dont le fonctionnement est proche de

```
1  Debut de l'exécution d'une tâche:
2     Empilement d'un nouveau bloc d'activation.
3
4  Création d'une tâche:
5     Allocation d'une clôture dans le bloc d'activation courant.
6
7  Fin de l'exécution d'une tâche:
8     Si le bloc d'activation contient des clôtures non exécutées:
9       Exécution de la tâche associée à la prochaine clôture
10      du bloc d'activation courant.
11    Sinon
12      Dépilement du bloc d'activation.
```

FIG. 4.2 – Exécution des tâches dans une pile suivant l'ordre de référence

la pile standard de nombreux langages de programmation (C++, Java,...). Comme dans Lazy Threads [29], une pile est composée d'un ensemble de blocs mémoire de taille fixe pouvant contenir un grand nombre de clôtures. Ces blocs sont alloués avec l'allocateur standard du langage utilisé. Initialement, une pile est composée d'un seul bloc. L'allocation (respectivement la désallocation) d'une nouvelle clôture dans la pile revient à incrémenter (respectivement décrémenter) le pointeur de sommet de pile. Lorsque le pointeur de sommet de pile atteint la fin du bloc courant, un nouveau bloc est alloué et chaîné avec le bloc précédent.

L'ordre séquentiel d'exécution des tâches est la principale différence entre une pile standard et la pile que nous proposons. Dans une pile standard, lors d'un appel de fonction, le flot d'exécution est transmis de la fonction mère à la fonction fille. Dans notre cas, l'ordre d'exécution séquentiel est l'ordre de référence défini dans la description de la machine abstraite du chapitre précédent. Lors de la création d'une tâche, la tâche mère conserve la main jusqu'à la fin de son exécution. Le flot d'exécution est ensuite transmis à la première tâche qu'elle a créée. Dans les deux cas, un nouveau "bloc d'activation" est alloué dans la pile au début de l'exécution de chaque tâche ou fonction. Dans le cas d'une pile standard, le bloc d'activation permet de stocker les variables locales à la fonction. Dans notre cas, le bloc d'activation permet de stocker les nouvelles clôtures qui seront allouées par la tâche. L'algorithme décrit dans la figure 4.1.1 suivant décrit le fonctionnement de cette pile.

La figure 4.1 illustre les différentes étapes de l'évolution de la pile associée au programme de la figure 4.3. La partie a) de cette figure montre l'état de la pile après l'exécution de la fonction `main`. La pile contient donc les deux données `a` et `b` ainsi que les clôtures `t1` et `t2`. Dans la partie b), la fonction `main` a terminé son exécution et a passé la main à la première tâche qu'elle a créée. Un nouveau bloc d'activation associé à l'exécution de `t1` a donc été empilé. La tâche `t1` crée quant à elle la tâche `t11` ce qui produit la pile présentée dans la partie c). Cette nouvelle tâche n'a pas de

```
1  procedure main() {
2    Donnée a, b;
3    Tache t1 =
4      AllocationCloture( T1 ) ;
5    t1.insérer Ecriture (a) ;
6    t1.FermerContexte() ;
7    Tache t2 =
8      AllocationCloture( T2 );
9    t2.insérer Lecture (a) ;
10   t2.insérer Ecriture (b) ;
11   t2.FermerContexte() ;
12 }
13 procedure T1(Donnée a) {
14   Tache t11 =
15     AllocationCloture( T11 ) ;
16   t11.insérer Ecriture (a);
17   t11.FermerContexte() ;
18 }
19
20 procedure T2(Donnée a, Donnée b)
21 { ... }
22
23 procedure T11(Donnée a)
24 { ... }
```

FIG. 4.3 – Programme associé à la pile présentée à la figure 4.1.

fille, après son exécution, le bloc d'activation qui lui est associé est supprimé de la pile (partie d) de la figure). De la même manière, la tâche `t1` n'a pas d'autres filles à exécuter, son bloc d'activation est donc à son tour dépilé. La main passe ensuite à la deuxième tâche fille de la fonction `main` (partie f) de la figure) avant que les derniers blocs d'activations ne soient dépilés et que l'exécution se termine (parties g) et h) de la figure).

4.1.2 Couplage du mécanisme de pile et de la gestion du flot de données

La gestion du flot de données de l'application est effectuée en chaînant les accès aux données partagées. Un nouvel accès est inséré dans la clôture au moment de l'exécution de l'instruction `tache.inserer mode (donnée)` de la machine abstraite. Cette structure de donnée contient le mode d'accès et une référence vers la donnée partagée considérée, ainsi que les pointeurs *prédécesseur* et *successeur* permettant de doublement chaîner chaque accès à une donnée partagée, suivant l'ordre de référence. Les règles suivantes permettent de construire le chaînage lorsqu'une nouvelle clôture c accédant une donnée partagé d est allouée. Dans ce cas, un nouvel accès a est créé dans la clôture.

1. La clôture c déclare le premier accès à la donnée d : la chaîne est créée et ne contient que a.
2. Le prédécesseur immédiat c' ayant déclaré un accès a' à la donnée d appartient au même bloc d'activation que la clôture c. L'accès a est inséré dans la chaîne associée à d, immédiatement après a'.
3. La clôture c est allouée dans un bloc nouvellement créé, ou n'a pas de prédécesseur accédant d dans le bloc d'activation courant. Dans ce cas, la référence vers la donnée d à été transférée par la tâche mère de c dont la clôture est allouée dans le bloc d'activation qui précède le bloc d'activation courant dans

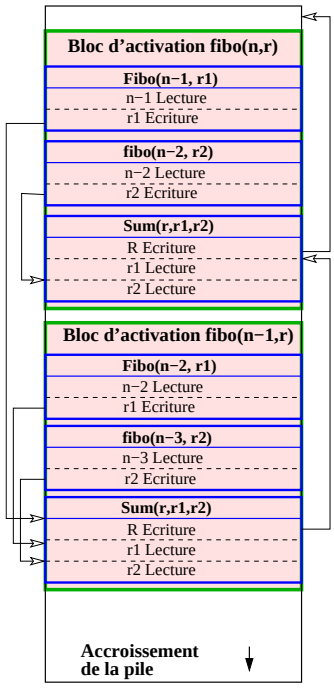


FIG. 4.4 – Chainage des accès.

la pile. L'accès a est dans ce cas inséré dans la chaîne immédiatement après l'accès déclaré par la tâche mère.

La figure 4.4 illustre ce chaînage en décrivant la pile à un instant donné de l'exécution du programme de la figure 3.10. La tâche $fibo(n)$ a terminé son exécution et a donné la main à la première tâche qu'elle a créée $fibo(n-1)$ qui a son tour vient de terminer son exécution. Le chaînage des successeurs est représenté par les flèches. Le chaînage des prédécesseurs est l'opposé du chaînage des successeurs et n'est pas représenté sur la figure pour des raisons de lisibilité.

4.2 Parcours du graphe en $O(1)$ pour l'ordonnancement

L'avantage de la construction d'un tel chaînage est double. Il permet d'abord de déduire un graphe de précédence des tâches allouées dans la pile à un instant donné, mais aussi de calculer le détail du flot de données entre les tâches entraînant ces dépendances. Ainsi, lorsqu'une tâche t produit une donnée d (accès déclaré en écriture), l'ensemble des tâches successeurs de t suivant d peut être simplement calculé en parcourant la chaîne des accès associée à d. De la même manière, l'ensemble des prédécesseurs d'une tâche peut aisément être calculé en utilisant ce chaînage. Il est donc possible de proposer une interface de parcours du graphe de flot de données dont chaque opération est effectuée à coût constant $O(1)$. L'interface que nous proposons

```
1  G = le_graphe_de_l_application();
2  IterToutesLesTaches it_taches = G.DebutToutesLesTaches();
3  tant que (it_taches != G.FinToutesLesTaches() ) {
4    Tache t = *it_taches;
5    IterSuccTache it_succ = t.DebutSuccesseurs();
6    tant que (it_succ != t.FinSuccesseurs() ) {
7      Donnee d = *it_succ;
8      ...
9      it_succ++;
10   }
11   it_taches++;
12 }
```

FIG. 4.5 – Exemple d'utilisation des iterateurs de l'interface pour l'ordonnancement associée à la machine abstraite.

est basée sur la notion d'itérateurs [62]. Un itérateur est un objet permettant de parcourir les éléments d'un ensemble d'objets de même type. Chaque itérateur que nous proposons permet de parcourir un ensemble de noeuds du graphe, par exemple les successeurs ou les prédécesseurs d'une tâche, en respectant dans tous les cas l'ordre de référence. A chaque itérateur sont associées les méthodes suivantes permettant de parcourir et d'accéder aux noeuds du graphe :

`++` : permet de passer à l'élément suivant dans l'ordre de référence.

`--` : permet de passer à l'élément précédent dans l'ordre de référence.

`*` : retourne l'élément pointé par l'itérateur en utilisant la même syntaxe que pour les pointeurs du langage C (par exemple *it).

L'interface que nous proposons est composée de plusieurs type d'itérateurs :

- Itérateur sur toutes les tâches du graphe : `IterToutesLesTaches`.
- Itérateur sur les tâches prêtes : `IterTachesPretes`.
- Itérateur sur tous les noeuds "donnée" du graphe : `IterToutesLesDonnées`.
- Itérateur sur les successeurs d'une tâche : `IterSuccTache`.
- Itérateur sur les prédécesseurs d'une tâche : `IterPredTache`.
- Itérateur sur les successeurs d'un noeud "donnée" : `IterSuccDonnée`.
- Itérateur sur les prédécesseurs d'un noeud "donnée" : `IterPredDonnée`.

Un itérateur peut être dans l'état "non initialisé", peut pointer sur un élément du graphe (en fonction de son type) ou peut prendre la valeur d'un marqueur de fin. Cette valeur est prise lorsque la méthode `++` (respectivement `--`) est appelée alors que l'itérateur pointe sur le dernier (respectivement premier) élément.

L'initialisation des itérateurs est effectuée à l'aide des méthodes suivantes :

DebutToutesLesTaches() : retourne un itérateur de type `IterToutesLesTaches` initialisé sur la première tâche créée dans l'ordre de référence.

FinToutesLesTaches() : retourne un itérateur de type `IterToutesLesTaches`

initialisé sur un marqueur de fin.

DebutTachesPretes() : retourne un itérateur de type `IterTachesPretes` initialisé sur la première tâche prête dans l'ordre de référence.

FinTachesPretes() : retourne un itérateur de type `IterTachesPretes` initialisé sur un marqueur de fin.

DebutToutesLesDonnées() : retourne un itérateur de type `IterToutesLesDonnées` initialisé sur le premier noeud "donnée" du graphe.

FinToutesLesDonnées() : retourne un itérateur de type `IterToutesLesDonnées` initialisé sur un marqueur de fin.

Les méthodes disponibles sur un noeud tâche sont :

DebutSuccesseurs() : retourne un itérateur de type `IterSuccTache` initialisé sur le premier noeud "donnée" successeur de la tâche.

FinSuccesseurs() : retourne un itérateur de type `IterSuccTache` initialisé sur un marqueur de fin.

DebutPredecesseurs() : retourne un itérateur de type `IterPredTache` initialisé sur le premier noeud "donnée" prédécesseur de la tâche.

FinPredecesseurs() : retourne un itérateur de type `IterPredTache` initialisé sur un marqueur de fin.

Enfin, les méthodes disponibles sur un noeud "donnée" sont :

DebutSuccesseurs() : retourne un itérateur de type `IterSuccDonnee` initialisé sur la première tâche successeur du noeud "donnée".

FinSuccesseurs() : retourne un itérateur de type `IterSuccDonnee` initialisé sur un marqueur de fin.

DebutPredecesseurs() : retourne un itérateur de type `IterPredDonnee` initialisé sur la première tâche prédécesseur du noeud "donnée".

FinPredecesseurs() : retourne un itérateur de type `IterPredDonnee` initialisé sur un marqueur de fin.

La figure 4.5 donne un exemple d'utilisation de ces itérateurs. La première ligne permet de récupérer une référence sur le graphe de l'application. A la ligne 2, un itérateur sur toutes les tâches est déclaré et initialisé. Les lignes 3 à 12 permettent de parcourir toutes les tâches du graphe. Pour chacune de ces tâches, les noeuds "donnée" successeurs sont parcourus (lignes 6 à 10).

4.3 Exportation de tâches

Dans cette section, nous présentons le fonctionnement de la pile lorsqu'un algorithme d'ordonnancement dynamique basé sur l'exportation de tâches est utilisé. La décision d'exporter une tâche dépend de l'algorithme d'ordonnancement utilisé. Un algorithme basé sur du vol de travail permettant de borner le nombre de tâches exportées est présenté dans le chapitre suivant.

De la même manière que dans le langage *Cilk* [11, 25] ou dans *Lazy Threads* [29], la pile présentée précédemment devient une pile "cactus" au moment où une tâche est exportée sur un autre processeur. Une deuxième branche de la pile est alors

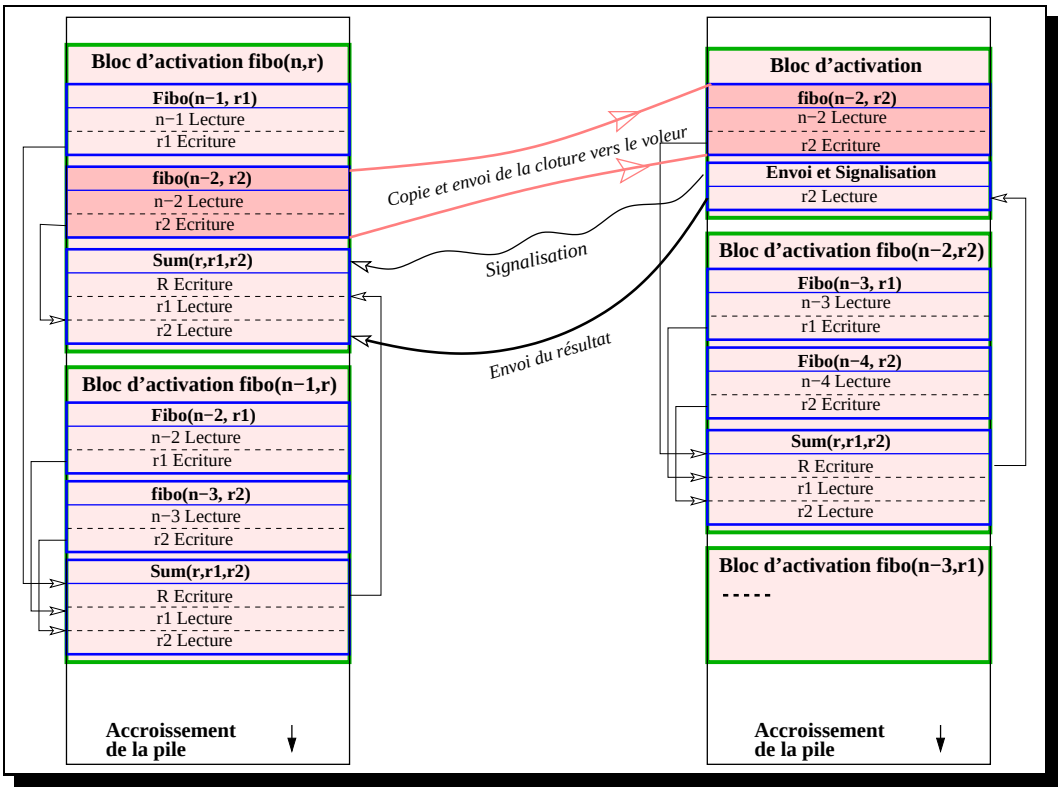


FIG. 4.6 – Exportation d'une clôture : la pile devient une pile cactus.

créée à partir de celle du processeur initialement propriétaire de la tâche. Chaque processeur est associé à une branche de la pile.

Si le processeur sur lequel la tâche choisie est exportée est un processeur distant, les données en entrée de la tâche exportée sont envoyées avec la clôture. Une nouvelle pile est alors créée sur le processeur cible pour stocker la clôture exportée. La figure 4.7 illustre ce mécanisme. La partie gauche de cette figure représente la pile du processeur initial de la clôture et la partie droite celle du processeur cible. La clôture `fibo(n-2, r2)` est choisie pour être exportée, elle est alors entièrement recopiée vers la branche du processeur cible.

Quelle que soit la stratégie choisie, l'exportation d'une ou plusieurs clôtures implique l'introduction de nouvelles synchronisations et communications de données. Si une tâche non prête est exportée, celle-ci ne peut pas commencer son exécution tant que ses paramètres n'ont pas été produits sur le processeur initial et envoyés sur le processeur cible. De la même manière, les tâches successeurs d'une tâche exportée qui sont restées sur le processeur initial, ne peuvent pas commencer leur exécution tant que le processeur cible n'a pas envoyé les résultats produits par les tâches exportées. Nous proposons de traiter ces cas de figure en insérant, lorsque cela est nécessaire des clôtures "d'envoi et de signalisation" aux endroits nécessaires dans les piles des processeurs voleurs et volés. Dans la figure 4.7, la clôture "Envoi et

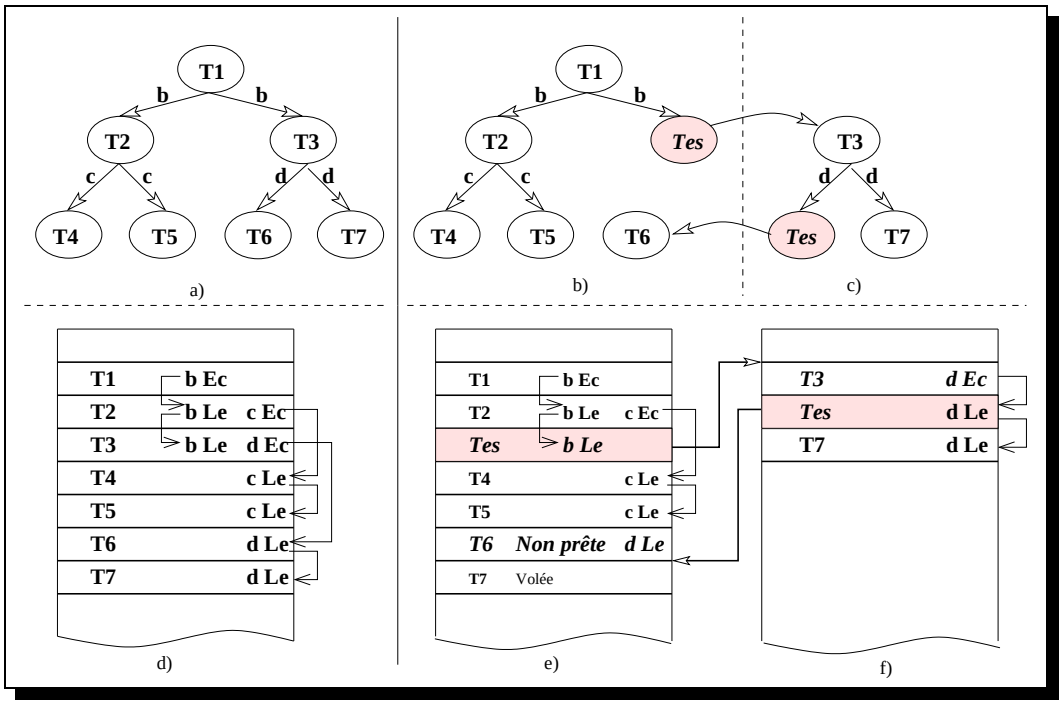


FIG. 4.7 – Génération des tâches d'envoi et de signalisation.

Signalisation" a pour objectif d'envoyer sur le processeur volé la donnée $r2$ dès que celle-ci aura été produite et de signaler sa production aux clôtures qui devront la consommer. Côté processeur initial, les clôtures "d'envoi et de signalisation" des paramètres en entrée des tâches exportées peuvent être insérées dans la pile à la place des tâches exportées. De cette manière, on aura l'assurance que ces tâches ne seront exécutées que lorsque les données qu'elles doivent envoyer auront été produites sans changer l'ordre d'exécution local des tâches ni le fonctionnement standard de la pile. Les tâches dépendantes des tâches exportées sont quant à elles marquées non prêtes. Côté voleur, il suffit d'empiler les clôtures d'envoi et de signalisation après la dernière tâche accédant la donnée à envoyer. Suite à l'exportation d'une clôtures, d'autres clôtures de la pile initiale sont éventuellement devenues non prêtes. Le traitement associé à ce cas de figure dépend de l'algorithme d'ordonnancement choisi. L'algorithme d'ordonnancement dynamique présenté dans le chapitre suivant inclut un tel traitement.

4.4 Execution parallèle par placement statique des tâches

Dans le cas particulier où le graphe de flot de données est entièrement construit dans une seule tâche du programme (la tâche principale par exemple), il est souvent préférable d'utiliser un algorithme d'ordonnancement statique pour placer efficacement les tâches du graphe de flot de données. Il est alors possible d'appliquer les

optimisations présentées dans [21] dans le cadre du développement d'une version d'Athapascan dédiée à ce type d'applications. En particulier, la connaissance du graphe de flot de données de l'application sur chaque noeud de la machine permet la génération d'un schéma de communication unidirectionnel optimal. Dans cette section, nous présentons les adaptations de ces optimisations au mécanisme de pile présenté dans ce chapitre.

Le modèle d'exécution associé à ce type d'ordonnancement proposé dans [21] a donc été mis à jour pour prendre en compte les spécificités du mécanisme de pile, mais aussi pour lever certaines contraintes. En particulier, aucune hypothèse n'est faite sur le site d'exécution de la tâche générant le graphe. En effet, dans [21], le modèle d'exécution était bâti sur l'hypothèse forte que la tâche générant le graphe était exécutée sur chaque site participant au calcul et qu'un algorithme d'ordonnancement déterministe était appliqué sur le graphe construit. Cette approche avait pour avantage d'éviter de diffuser la totalité du graphe et son ordonnancement. Cependant, elle imposait que le graphe soit construit uniquement par la tâche principale ou après une barrière globale sur tous les processeurs. Dans un cadre plus général, il peut être intéressant d'appliquer ce type de stratégie sur le graphe construit par n'importe quelle tâche de l'application mais aussi d'autoriser l'utilisation d'ordonnancement non déterministe. Pour cette raison, nous proposons le modèle d'exécution suivant en six étapes :

1. Exécution de la tâche principale et construction de la pile contenant le graphe de flot de donnée.
2. Ordonnancement puis diffusion de la pile à tous les noeuds participant au calcul.
3. Sur chaque noeud, création d'une nouvelle pile vide.
4. Sur chaque noeud, parcours de la pile contenant le graphe et calcul pour chaque noeud "donnée" d'un identifiant de communication et de l'ensemble des sites de diffusion. Ces informations sont empilées dans la nouvelle pile locale.
5. Sur chaque noeud, deuxième parcours de la pile contenant le graphe, empilement des tâches locales dans la nouvelle pile ainsi que des tâches d'envoi et de signalisation. A chaque tâche est associé un compteur du nombre de paramètres en entrée produit à distance. Lorsqu'une donnée produite à distance est réceptionnée, le compteur des tâches locales devant consommer cette donnée est décrémenté. Lorsque ce compteur a la valeur zéro, la tâche est considérée comme prête à s'exécuter.
6. Sur chaque noeud, lancement de l'exécution.

La phase de génération des communications est une phase prépondérante dans ce modèle d'exécution puisque la connaissance sur chaque noeud de la totalité du graphe et de son placement permet de générer des communications unidirectionnelles entre les tâches du graphe. De plus, dans le cas où les lecteurs d'une même donnée partagée sont distribués sur plusieurs noeuds de la machine, il est possible d'utiliser des primitives de diffusion efficaces.

```
1  G = le_graphe_de_l_application();
2  IterToutesLesDonnees it_donnees = G.DebutToutesLesDonnees();
3  Tant que(it_donnees != G.FinToutesLesDonnees() ) {
4     Tableau sitediffusion[ ];
5
6     Si local((*it_donnees).SiteProducteur()) == faux {
7        (*it_donnees).marquer_non_pret();
8     }
9
10    Si local((*it_donnees).SiteProducteur()) == vrai {
11       TacheEnvoiSignalisation
12          tes((*it_donnees).SitesConsommateurs());
13       AjouterConsommateur(*t_pred_donnee, tes);
14    }
15    it_donnees++;
16 }
```

FIG. 4.8 – Algorithme de génération des tâches d'envoi et de signalisation.

Du point de vue synchronisation, la phase de génération des communications revient à modifier le graphe de flot de données en y insérant des tâches chargées de communiquer les données entre processeurs. L'algorithme de génération de ces tâches dans le graphe de flot de données est présenté dans la figure 4.8. Dans cet algorithme, la fonction `local(t)` retourne un booléen dont la valeur est `vrai` lorsque la tâche t a été ordonnancée sur le site local. La fonction `site(t)` retourne un entier dont la valeur représente le site d'exécution de la tâche t. L'accès au site producteur est effectué à l'aide de la méthode `d.SiteProducteur()` et l'accès à l'ensemble des consommateurs est effectué par la méthode `d.SitesConsommateurs()`. Enfin, la fonction `AjouterConsommateur(d, t)` est utilisée pour rajouter la tâche t dans le graphe comme consommatrice de la donnée d. La tâche t étant une tâche d'envoi et de signalisation, la donnée d est toujours le seul paramètre de la tâche t.

Cet algorithme est basé sur le parcours des noeuds "donnée" du graphe de flot de données (lignes 2,3 et 15). Deux cas sont alors à traiter :

1. Le producteur de la donnée est distant et au moins un des consommateurs est local.
2. Le producteur de la donnée est local et au moins un des consommateurs est distant.

Le premier cas est traité par les lignes 6, 7 et 8. Le noeud "donnée" courant est simplement marqué non prêt. Une tâche d'envoi et de signalisation exécutée sur le site producteur aura la charge d'envoyer la valeur et de passer le noeud "donnée" dans l'état prêt. L'exécution de la partie du graphe qui succède ce noeud "donnée"

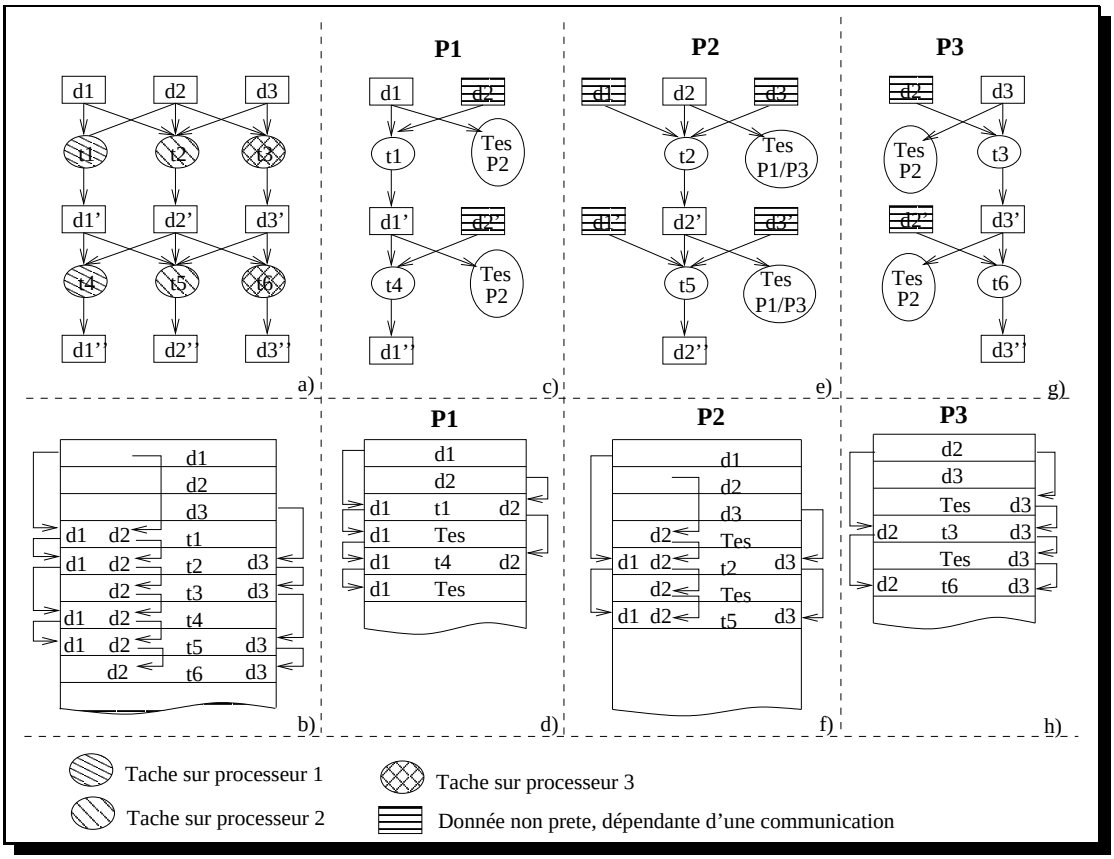


FIG. 4.9 – Génération statique des tâches d'envoi et de signalisation (notées *Tes*).

pourra alors éventuellement commencer. Le deuxième cas est traité de la ligne 10 à la ligne 14. Une tâche d'envoi et de signalisation est initialisée avec l'ensemble des noeuds consommateurs de la donnée courante (ligne 11 et 12). Cette tâche a pour objectif de diffuser la donnée aux sites consommateurs excepté le site courant (dans le cas où le site courant est aussi consommateur). Cette tâche étant successeur du noeud donnée (ligne 13), son exécution ne pourra commencer qu'après sa production, garantissant ainsi la diffusion de la bonne valeur.

La figure 4.9 donne un exemple de génération des tâches d'envoi et de signalisation. Les parties a) et b) montrent un exemple de graphe de flot de données ainsi que la pile associée après l'exécution de la tâche principale et de l'ordonnancement du graphe. Les parties c), e) et g) montrent l'état du graphe du flot de données sur les trois processeurs participant à l'exécution après la phase de génération des communications. Les parties d), f) et h) montrent la pile générée sur chaque processeur.

Le mécanisme présenté dans cette section est totalement indépendant de l'algorithme d'ordonnancement choisi. L'algorithme présenté dans la section 5.1 sera utilisé pour valider cette approche.

4.5 Couplage ordonnancement statique, ordonnancement dynamique

Il existe de nombreux cas pour lesquels il peut être intéressant de remettre en cause l'ordonnancement statique effectué initialement. Une ou plusieurs machines peuvent par exemple subir une hausse de charge inattendue au cours de l'exécution. Il peut aussi s'avérer que l'algorithme d'ordonnancement statique utilisé ne soit pas suffisamment adapté à l'application et à la machine cible.

Cette remise en cause peut être effectuée en couplant un ordonnancement statique, qui sera utilisé pour effectuer un préplacement des tâches du graphe de flot de données et un algorithme d'ordonnancement dynamique utilisé pour équilibrer la charge au cours de l'exécution.

Du point de vue du mécanisme de pile présenté dans les sections précédentes, les piles construites statiquement sur chaque site en utilisant le modèle d'exécution de la section 4.4 peuvent parfaitement être remises en cause en utilisant les mécanismes décrits dans la section 4.3. Cependant, deux problèmes importants restent à résoudre :

1. Quand déclencher une remise en cause ?
2. Quelles tâches voler pour que la remise en cause soit bénéfique ?

Le déclenchement d'une remise en cause sur inactivité semble toujours être une bonne solution. Il est toutefois difficile d'évaluer le temps pendant lequel le processeur va rester inactif et donc d'évaluer s'il peut être bénéfique de remettre en cause l'ordonnancement initial ou non.

Pour ce qui est des tâches à voler, il est impératif de choisir une tâche qui peut permettre de déclencher l'exécution des tâches non prêtes chez le voleur. Les algorithmes basés sur la connaissance du graphe de flot de données présentés dans la section 4.3 peuvent dans ce cas être utilisés.

4.6 Conclusion

Dans ce chapitre, nous avons présenté le couplage d'un mécanisme d'allocation à base de pile cactus avec un mécanisme de gestion distribuée du flot de données. Quel que soit le type de stratégie d'ordonnancement utilisée (statique où dynamique), nous avons montré que les mécanismes de dégénération distribuée permettent de générer automatiquement des communications de type "unidirectionnelles" pour les échanges de données du programme. Les mécanismes de dégénération séquentielle basés sur un allocateur de type pile "cactus" permettent, quant à elle, de limiter le surcoût lié à la création des tâches. Dans le chapitre suivant, deux algorithmes d'ordonnancement, l'un dynamique l'autre statique, permettant de tirer partie des particularités de ces mécanismes sont présentés.

CHAPITRE 5

Intégration de stratégies d'ordonnancement

Nous proposons dans ce chapitre d'étudier l'intégration de deux stratégies d'ordonnancement permettant de tirer partie des mécanismes présentés dans les chapitres précédents. Nous présenterons dans un premier temps une stratégie d'ordonnancement dynamique basée sur du vol de travail. Nous montrerons que cet algorithme permet de majorer le nombre de vols réussis et donc le nombre de tâches exportées intervenant au cours de l'exécution d'un programme. Ce résultat permet d'affiner la borne du nombre de défauts de page présentée dans le chapitre 3 lorsque le protocole de cohérence "flot de données" est utilisé.

Nous étudierons ensuite un algorithme d'ordonnancement statique basé sur le regroupement des données partagées du programme dans les différents modules de mémoire disponibles. Nous proposons d'effectuer ce regroupement en construisant un graphe de dépendance de données et en utilisant une bibliothèque de partitionnement comme Scotch [50] ou Metis [39] pour le calculer. Les tâches du programme sont ensuite distribuées en fonction des données auxquelles elles accèdent en suivant la règle OCR (*Owner Compute Rule*).

5.1 Ordonnancement par vol de travail

Nous proposons ci-dessous un algorithme d'ordonnancement distribué de type vol de travail pour un programme basé sur la machine abstraite présentée dans le chapitre 3. Les performances théoriques de cet algorithme d'ordonnancement seront analysées en prouvant l'existance d'une majoration du nombre de vols réussis. Dans ce type d'algorithmes, lorsqu'un processeur devient inactif, il passe à l'état "voleur" et cherche à récupérer du travail sur les autres processeurs. Cet état reste inchangé tant qu'une tâche prête n'a pas été importée ou qu'un signal de terminaison n'a pas été reçu. L'exportation d'une tâche correspond donc à l'exécution de celle-ci sur un site différent de celui où elle a été créée ; un vol réussi correspond donc à l'exportation d'une tâche (voir le chapitre 3).

5.1.1 Ordonnancements par vol de travail existants

Pour les programmes de la classe déterministe non contraint, les algorithmes par vol de travail (*work-stealing*) peuvent être classés en deux catégories :

- LIFO-local. Chaque processeur gère localement ses tâches dans une pile LIFO ; lorsqu'une nouvelle tâche est créée, la tâche mère est empilée (i.e. la continua-

tion) et la tâche fille commence son exécution. Lorsqu'une tâche se termine ou qu'une opération de synchronisation bloquante est exécutée, trois cas sont distingués :
- si la pile n'est pas vide, la dernière tâche prête empilée est exécutée (i.e. exécution locale en profondeur d'abord) ;
- si la pile est vide, un autre processeur dont la pile est non vide est choisi ; la tâche la plus ancienne dans la pile de ce processeur est volée (FIFO).
- la terminaison d'une tâche F peut rendre prête la tâche P qui l'avait créée : P a pu être volée par un autre processeur pendant l'exécution locale de F et P était bloquée sur une opération de synchronisation sur la fin de F. Dans ce cas, le processeur qui vient de terminer F reprend l'exécution de P.

Plusieurs systèmes incluent un tel ordonnancement (Cilk, LazyThreads, Satin,...). Dans le cas des programmes du modèle strict, cet ordonnancement permet une bonne localité : le nombre de vols réussis est majoré par $O(p.T\infty)$ avec une bonne probabilité, au prix d'un espace mémoire éventuellement important mais majoré par $O(p.S_1)$.

- DF-global. Toutes les tâches sont ordonnées selon un ordre séquentiel total ; lorsqu'un processeur devient inactif, il vole le thread prêt le plus prioritaire selon cet ordre. Dans le cas d'un programme parallèle récursif, l'ordre total est l'ordre "profondeur d'abord". L'exécution distribuée respecte donc cet ordre au plus près ; à chaque top, au plus $p-1$ threads sont exécutés de manière prématurée par rapport à l'exécution séquentielle "profondeur d'abord", ce qui permet de limiter l'espace mémoire requis à $S_1+O(p.T_\infty)$. Par contre, le nombre d'exportations est important, majoré par $O(T_1/p+T_\infty)$, ce qui entraîne une perte de localité.

L'implémentation DF-global proposée dans [48] est un compromis entre les deux précédents ordonnancements LIFO-local et DF-global. Ici, le nombre de piles peut être supérieur au nombre de processeurs. A un instant donné, un processeur actif ne possède qu'une seule de ces piles. Comme dans LIFO-local, chaque pile est exécutée localement par le processeur actif qui la possède selon l'ordre séquentiel "profondeur d'abord" ; de plus, les différentes piles sont (doublement) chaînées entre elles dans une liste R selon l'ordre séquentiel profondeur d'abord, y compris celles qui ne sont pas associées à un processeur. Lorsqu'un processeur se bloque ou termine l'exécution de toutes les tâches d'une pile, il relâche la propriété de sa pile et vole une tâche prête en sommet de l'une des p premières piles de R. La tâche volée est la tâche prête la plus prioritaire selon l'ordre séquentiel global, à la différence de la stratégie LIFO-local qui peut voler dans n'importe quelle pile.

Lorsqu'une tâche termine son exécution, deux cas sont distingués :
- Si sa tâche mère est dans la pile locale : elle est dépilée selon l'ordre normal LIFO et son exécution reprend ;
- Si sa tâche mère a été volée et figure dans une autre pile P'. Dans le cas où la tâche mère est bloquée sur une synchronisation et que la terminaison de la tâche courante rend sa tâche mère prête, le processeur relâche la propriété de sa pile courante et prend la propriété de la pile P' dont il redémarre l'exécution.

Enfin, lorsqu'un processeur est inactif (ne possède pas de pile en cours d'exécution), il cherche à voler une des p premières piles (qui peuvent être éventuellement en cours d'exécution sur l'un des processeurs) jusqu'à trouver la plus ancienne tâche prête[1]. Cette tâche est alors volée (comme dans LIFO-local) et une nouvelle pile est créée pour son exécution ; cette nouvelle pile est insérée dans R juste après la pile qui possédait la tâche avant le vol.

L'inconvénient de cette structure est la gestion dynamique de la liste R, qui peut posséder un nombre de piles très supérieur au nombre de processeurs. Ce nombre est majoré par $O(K.p.T_\infty)$ [48] ; K est un paramètre entier qui spécifie le volume mémoire maximal qu'un processeur peut allouer entre deux vols consécutifs. K peut être ajusté dynamiquement par l'utilisateur. Si K est fixé à une constante petite, l'algorithme DF-Deques [48] a le même comportement que DF-Global : la taille de R est faible – $O(p.T_\infty)$ – mais le nombre d'exportations, majoré par $O(T_1/p + T_\infty)$, est important ce qui entraîne une perte de localité.

Dans la suite, nous proposons un algorithme d'ordonnancement de vol de type LIFO-local basé sur la machine abstraite du chapitre 3 et sur le mécanisme de pile présenté dans la section précédente. A la différence des algorithmes précédents, basés sur une exécution "profondeur d'abord" (DF pour *Depth first*), le programme est exécuté selon l'ordre de référence (RO pour *Reference Order*). A la différence de DF-Deques, le nombre de piles est limité à $\#L$ pour limiter le nombre d'exportations.

5.1.2 Ordonnancement RO-local/FIFO-distribué

Dans cette section nous proposons un algorithme d'ordonnancement original appelé RO-local/FIFO-distribué. Dans cet ordonnancement, l'exécution séquentielle est effectuée selon l'ordre de référence (RO-local) présenté dans la section 3.2.

Dans cet algorithme, on considère qu'une tâche t est prête si tous les paramètres auxquels elle accède en lecture ont été produits et que les prédécesseurs dans l'arbre de création des tâches de t sont démarrés.

L'ordonnancement favorise l'exécution locale "profondeur d'abord" ; mais en cas de vol sur un autre processeur, afin d'éviter qu'une tâche prête reste longtemps sans être volée, l'ordonnancement choisit toujours prioritairement une tâche prête parmi les plus anciennes. A tout processeur i est associée une liste L_i de piles dont il contrôle l'exécution. L'une de ces piles est en cours d'exécution sur le processeur ; elle est pointée par le pointeur de pile courant P_i du processeur i. La tâche en cours d'exécution est en bas de la pile P_i. Lorsqu'une nouvelle tâche est créée, elle est empilée dans la pile en cours d'exécution.

Initialement, toutes les listes sont vides sauf celle d'un processeur qui contient

[1] Dans [48], la pile victime est tirée au hasard parmi les p premières piles de R

une unique pile dans laquelle est exécutée le point d'entrée *main* du programme.

La taille de chacune des listes est limitée à un nombre de piles $\#L \geq 1$ fixé par l'utilisateur.

Les opérations d'ordonnancement effectuées localement par chaque processeur sont les suivantes.

Exécution-tâche (RO-local) : L'exécution d'une tâche dans l'ordre de référence est gérée par la pile dont le fonctionnement est décrit dans la section 4.1. Lorsqu'une tâche commence son exécution, un bloc d'activation permettant de stocker les données allouées lors de l'exécution de la tâche est empilé. Lorsqu'une nouvelle tâche est créée, une structure de données appelée clôture est allouée dans le bloc d'activation de la tâche en cours d'exécution. Cette structure de données est utilisée comme "graine" pour l'exécution future de la tâche. Lorsqu'une tâche T termine son exécution, le flot d'exécution du programme est transmis à la première tâche créée par T. Si une telle tâche n'existe pas, le bloc d'activation courant est dépilé et le flot d'exécution est transmis à la tâche suivant T dans le bloc d'activation de la tâche mère de T.

Lorsque la dernière tâche d'une pile termine son exécution, la pile est supprimée de la liste L_i.

Recherche-locale : Cette opération est effectuée lorsqu'une pile devient vide ou que la prochaine tâche à exécuter suivant l'ordre de référence n'est pas prête. Dans ce cas, le processeur cherche prioritairement dans ses piles locales une nouvelle tâche à exécuter. Cette recherche est effectuée de la manière suivante :

1. Le processeur i parcourt alors la liste L_i pour trouver une nouvelle pile locale dont la dernière tâche (la dernière empilée dans la pile non démarrée) est prête. Si une telle pile existe, elle devient la pile courante du processeur qui peut alors démarer l'exécution de la dernière tâche.
2. Si L_i contient déjà $\#L$ piles et qu'aucune pile de L_i n'est prête, le processeur i ne peut plus créer une nouvelle pile ; il se bloque alors jusqu'à ce que la liste L_i contienne moins de $\#L$ piles ou que la liste L_i contienne une pile dont la dernière tâche non démarrée est prête.
3. Sinon, il n'existe pas de pile prête localement mais la liste L_i contient moins de $\#L$ piles. Le processeur recherche une nouvelle tâche prête pour l'exécuter dans une nouvelle pile.

   Il parcourt alors dans l'ordre les piles de la liste L_i jusqu'à trouver une tâche t prête. Chaque pile est parcourue de haut en bas (ordre FIFO) ; la tâche prête t choisie est donc la tâche la plus ancienne dans la pile qui a été insérée dans L_i depuis le plus longtemps (parmi les piles qui contiennent des tâches prêtes). Si le processeur trouve une telle tâche t, il crée une nouvelle pile Q pour l'exécution de cette tâche t ; la pile Q est placée en queue de la liste L_i et la pile P_i courante devient la pile Q. Le processeur exécute alors la tâche t localement selon l'opération *Exécution-tâche*.

Si aucune tâche prête n'est trouvée dans la liste L_i locale, le processeur effectue alors une opération de Vol.

Vol (FIFO-distribué) : Le processeur i examine cycliquement et dans l'ordre les processeurs $s+1, \ldots, p, 1, \ldots, s-1, s, \ldots$ où s est l'indice du dernier processeur que i vient de voler avec succès (initialement un indice arbitraire) ; il s'arrête dès qu'il trouve un processeur j qui contient une tâche prête. Pour éviter les problèmes de concurrence, chaque pile est associée à un verrou qui est pris par le processeur qui cherche à voler dans la pile.

Le processeur i choisit chez le processeur j la tâche qu'il va voler de la manière suivante. La liste L_j contient les piles $[Q_1, Q_2, \ldots, Q_k]$ dans cet ordre. Soit v l'indice de la pile qui suit dans L_j la dernière pile dont une tâche prête a été extraite (soit par recherche locale soit par vol) ; initialement, $v = 1$. Le processeur parcourt alors dans l'ordre les piles $Q_v, Q_{v+1}, \ldots, Q_k, Q_1, Q_{v-1}$ de la liste L_j jusqu'à trouver une tâche t prête. Chaque pile est parcourue de haut en bas (ordre FIFO) ; la tâche prête t choisie est donc la tâche qui a été insérée dans L_j depuis le plus longtemps (parmi les piles qui contiennent des tâches prêtes). Si L_j ne contient aucune tâche prête, le processeur parcourt alors la pile P_j du processeur j, de haut en bas (ordre FIFO) pour trouver une tâche prête.

Le processeur i crée alors localement une nouvelle pile Q dans laquelle la tâche t est empilée. La pile Q est placée en queue de L_i et la pile P_i courante devient cette nouvelle pile Q. Le processeur exécute alors localement la tâche t dans P_i selon *Exécution-tâche.*

Proposition 1 *Dans l'ordonnancement RO-local/FIFO-distribué, il y a à tout instant au moins un processeur actif. L'ordonnancement termine donc l'exécution en temps fini.*

Preuve: Il existe toujours une pile dans laquelle la tâche la plus prioritaire selon l'ordre séquentiel total de référence est la dernière tâche de cette pile. Cette pile est prête : elle est soit en cours d'exécution, soit dans la liste L_i d'un processeur. Si ce processeur est inactif, il possède donc une pile locale prête dont il peut directement démarrer l'exécution. Par suite, il existe toujours une pile prête : ainsi, il n'y a pas d'inter-blocage global lié à l'attente d'une tâche prête lorsque les listes L_i des processeurs ont atteint la taille maximale $\#L$. □

Remarques Dans cet ordonnancement, un processeur choisit prioritairement :

1. la tâche locale la plus proche selon l'ordre séquentiel profondeur d'abord si elle est prête ;
2. une autre pile locale dont la tâche en sommet de pile est prête ;
3. une autre tâche prête dans l'une des piles locales ;
4. une tâche prête sur un autre processeur.

En l'absence de vol, lorsqu'une tâche est prête et que toutes les données qu'elle peut accéder en lecture sont prêtes, elle et ses descendantes selon l'arbre de création peuvent être exécutées séquentiellement sans blocage selon l'ordre de référence. Cependant, en cas de vol, il se peut qu'une de ses filles t créées localement ne soit pas prête si l'un des prédécesseurs de t a été volé par un autre processeur.

5.1.3 Analyse de la performance de RO-local/FIFO-distribué

Cette section analyse le nombre de vols et la performance de l'algorithme précédent. Comme dans le modèle "accès atomique" introduit dans [9], nous considérons que les opérations de modification d'une pile sont effectuées en exclusion mutuelle.

Nous montrons tout d'abord (§5.1.3) que le nombre V de vols est majoré par

$$V \leq p.(T_\infty.\Phi_v + N_\infty.\#L)$$

où :

- p est le nombre de processeurs identiques ;
- T_∞ est le temps d'exécution minimal sur une infinité de processeurs identiques (chemin critique) ; T_∞ est évalué en unité de temps.
- Φ_v est la fréquence entre deux vols réussis successifs réalisés par un même processeur ; nous supposons donc qu'un processeur n'effectue pas plus de Φ_v vols réussis par unité de temps. Par exemple, si l'unité de temps est l'opération élémentaire et qu'un vol réussi nécessite x opérations élémentaires, on aura $\Phi_v = \frac{1}{x}$. Généralement, Φ_v est très inférieur à 1.
- N_∞ est le nombre maximal de tâches sur un chemin dans le graphe de précédences.
- $\#L$ est le paramètre nombre maximal de piles locales à un processeur ; $\#L \geq 1$ est fixé par l'utilisateur.

Par souci de simplification, comme l'analyse est centrée sur le nombre de vols, nous négligeons, les délais de contention liées aux verrous (en $O(p)$ pour chaque accès) : le nombre de vols étant montré faible dans le cadre de programmes avec un grand degré de parallélisme (T_∞ très petit devant le temps T_1 d'une exécution séquentielle), le surcôut lié à ces contentions est effectivement négligeable.

Nous déduisons ensuite le temps d'exécution (§5.1.3) et l'espace mémoire (§5.1.3) requis par l'ordonnancement.

Majoration du nombre de vols

Pour la preuve du résultat ci-dessus, nous considérons un programme quelconque avec une entrée arbitraire ; ce programme est constitué d'une unique tâche principale appelée *main*. La preuve est basée sur l'analyse d'une exécution quelconque de ce programme de durée d. À partir d'une séquence de tâches deux à deux en précédence, de l'ensemble $\{1, \ldots, d\}$ des tops est partitionné en deux ensembles : l'un de durée faible dans lequel le nombre de vols est majoré à partir de T_∞ ; l'autre dans lequel le nombre de vols est majoré par le nombre maximal N_∞ de tâches en précédence.

Pour définir N_∞, nous introduisons tout d'abord le graphe de création et le graphe de précédence qui sont indépendants de l'ordonnancement de l'exécution de la tâche *main*.

Définition 3 *Soit une tâche u dont le corps est la séquence d'instructions [I_1, création t_1, I_2, création t_2, ..., création t_k, $I_k + 1$] où I_j est un bloc d'instructions ne comportant aucune création de tâches. Le graphe de création A_u associé à u est définit récursivement comme suit :*

- *A_u possède $k+1$ nœuds c_i $(1 \leq i \leq k+1)$: pour $1 \leq i \leq k$, chaque nœud c_i modélise la séquence (I_i ; création t_i), i.e. l'exécution de I_i et de la création de t_i ;*
- *A_u possède k nœuds d_i $(1 \leq i \leq k)$: d_i modélise l'instruction élémentaire de démarrage de la tâche t_i ;*
- *la racine de A_u est le nœud c_1 ;*
- *pour $1 \leq i \leq k$, il y a une arête de c_i à c_{i+1} qui modélise la précédence entre (I_i ; création t_i) et I_{i+1}.*
- *pour $1 \leq i \leq k$, il y a une arête de c_i à d_i qui modélise la précédence entre la création de t_i et son démarrage ;*
- *pour $1 \leq i \leq k$, il y a une arête de c_k à d_i qui modélise qu'aucune des tâches filles de u ne sera démarrée avant que toutes les tâches filles de u n'aient été créées*[2].
- *pour $1 \leq i \leq k$, il y a une arête entre d_i et la racine du graphe A_{t_i} associé à t_i, qui modélise que A_{t_i} ne sera pas démarrée avant que t_i ne soit démarrée.*

Le graphe de création A de l'exécution est le graphe A_{main}. Soit F_∞ le nombre de tâches maximal sur un chemin dans A ; F_∞ est la taille de la pile (en nombre de tâches) lorsque le programme est exécuté séquentiellement selon l'ordre de référence.

Le graphe (dynamique) de précédence G est construit à partir du graphe de création A en le complétant par les arcs de précédence (dépendances de donnée écriture-lecture) entre les tâches créées (une tâche correspond à un sous-arbre). Dans la suite, nous notons N_∞ le nombre maximal de tâches sur un chemin dans G et T_∞ la durée d'exécution de G avec un nombre de processeurs infini.

Lemme 5 *Dans l'ordonnancement RO-Global/FIFO-distribué, le nombre de tâches dans une pile est toujours borné par F_∞.*

Preuve: Dans l'algorithme d'ordonnancement proposé, si un processeur est inactif, sa pile est vide ; si il est actif, il a toujours au plus une tâche en cours d'exécution et sa pile possède alors uniquement des prédécesseurs dans A de cette tâche. □

[2]Cette précédence artificielle est ajoutée pour que le nombre de tâches au dessus d'une tâche u dans une pile soit majoré par la profondeur de u dans le graphe de création. Cette précédence ne change pas les caractéristiques T_1 et T_∞ du programme. En effet, on peut toujours transformer un programme quelconque de la machine abstraite en un programme équivalent (même T_1 et T_∞) dans lequel une tâche crée toujours au plus deux filles et dans laquelle la précédence artificielle devient implicite.

Définition 4 *Une tâche est dite* volable *ssi elle est créée et prête mais que son exécution n'a pas démarrée.*

Une tâche volable est dans une pile Q qui figure dans une des listes L_j des piles en attente. Cette pile est éventuellement en cours d'exécution si la pile P_j courante sur le processeur j est la pile Q.

Un processeur i ne vole une tâche sur un autre processeur que lorsque qu'aucune des piles de sa liste locale L_i ne contient de tâches prêtes. Il parcourt alors cycliquement les autres processeurs pour en trouver un qui possède une tâche volable t.

Lemme 6 *Soit u une tâche volable pendant un intervalle de temps $[\tau_O, \tau_1]$. Soit j l'indice du processeur qui possède u (il reste inchangé tant que u n'est pas volée). Soit $h_{[\tau_O,\tau_1]}(u)$ la position maximale de u dans sa pile (i.e. le nombre maximal de prédécesseurs de u qui figurent dans la même pile que u) à un instant $t \in [\tau_O, \tau_1]$. Le nombre de vols réussis pendant que u est volable est majoré par*

$$(p-1).\#L.h(u).$$

Preuve: A tout instant, il y a au moins un processeur actif, donc au plus $p-1$ processeurs inactifs. Soit j le processeur qui possède la tâche volable u pendant $[\tau_O, \tau_1]$.

La tâche u est volable et dans l'une des piles de L_j à une hauteur $h(u)$. Sur un même processeur, chacune des au plus $\#L$ piles est examinée cycliquement après chaque vol réussi. Donc, après au plus $h(u).\#L$ vols réussis effectués sur le processeur j durant $[\tau_0, \tau_1]$, la tâche u aura été soit démarrée sur un autre processeur (en retour d'un vol réussi), soit exécutée localement.

De plus, après chaque vol réussi, un processeur cherche à voler le processeur suivant, cycliquement. Ainsi, en pire cas, après $(p-1).\#L.h(u)$ vols réussis par n'importe quel processeur durant $[\tau_0, \tau_1]$, la tâche u aura nécessairement été soit volée soit exécutée localement. On en déduit que le nombre total de vols réussis pendant que t est volable est majoré par $(p-1).h(u).\#L$. □

Proposition 2 *Le nombre V de vols réalisés par l'ordonnancement RO-local/FIFO-distribué est majoré par*

$$V \leq (p-1).\left(T_\infty.\Phi_v + N_\infty.\#L\right).$$

Preuve: Il existe dans G une séquence S de tâches volables, situées toutes sur un chemin de la première tâche du programme à une de celles qui terminent en dernier ; S est telle qu'à tout top de l'exécution une tâche de S est soit volable soit en exécution. Cette séquence est construit comme suit. Soit t_1 une tâche qui s'est terminée au dernier top de l'exécution. Soit t_2 un prédécesseur de t_1 dans G (non nécessairement immédiat) qui s'est terminé le plus tard ; entre la date de fin de t_2 et la date de début de l'exécution de t_1, t_1 est volable et t_2 précède t_1 dans G. On construit ainsi

en remontant dans l'exécution une séquence de tâches $S = (t_i)_{i=1..m}$ prédécesseurs (non nécessairement immédiats) dans G et telle qu'à tout top de l'exécution il existe une de ces tâches qui est soit volable soit en cours d'exécution. Le nombre de tâches dans cette séquence S est majoré par le nombre de tâches sur un chemin critique dans G, donc par N_∞.

Finalement, on partitionne les tops $\{1, ..., d\}$ en d'une part les tops A où une tâche de $S = (s_1, \ldots, s_m)$ (i.e. $s_i = t_{m+1-i}$) est en cours d'exécution ; d'autre part les tops B où une tâche de S est volable.

- Lors d'un top de A, le nombre de vols réussis possibles est borné par $p - 1$. Or, comme les tâches de S sont en précédence dans G, le nombre de tops dans A est majoré par T_∞. De plus, un processeur n'effectue pas plus de Φ_v vols réussis par unité de temps. Comme au plus $p-1$ processeurs effectuent un vol réussi à chaque top, le nombre de vols réussis dans A est majoré par $T_\infty.\Phi_v$.
- B est constitué des tops où s_1 puis s_2, ... puis s_m sont volables.
  Soit $h(s_i)$ la hauteur de s_i dans la pile dans laquelle elle a été créée. Lorsque s_i est volable, d'après le lemme 6, le nombre de vols est majoré par $(p - 1).\#L.h(s_i)$.
  Par construction des piles et définition du graphe de création A, au dessus de s_i et dans sa pile ne figurent que des prédéccesseurs de s_i. Comme les tâches dans S sont en précédence, on a donc $\sum_{u \in S} h(u) \leq N_\infty$.
  Le nombre de vols réussis dans B est donc majoré par $(p-1).N_\infty.\#L$.

Finalement, on en déduit que le nombre V de vols est majoré par :

$$(p-1).\left(T_\infty.\Phi_v + \#L.N_\infty\right).$$

□

Remarque sur la majoration de $\#L$. Le nombre de piles en attente sur chaque processeur influence directement la majoration sur le nombre de vols. Pour certains programmes, cette taille est faible. Par exemple, pour le cas d'un programme série-parallèle, il suffit d'une pile en attente sur un processeur.

Analyse de la durée d'exécution

Si, lorsqu'un processeur devient inactif, la taille de sa liste de pile L_i reste toujours inférieure à $\#L$, l'ordonnancement effectué est de type glouton : il n'existe pas de top où un processeur inactif n'est pas en train de voler une tâche alors que des tâches sont prêtes. Cette hypothèse est vérifiée en particulier lors de l'ordonnancement de programmes série-parallèle ou stricts avec $\#L = 0(1)$.

Dans la suite, nous supposons que l'ordonnancement réalisé vérifie cette propriété (ordonnancement glouton). Si l'on ne prend pas en compte le surcoût d'ordonnancement, la borne de Graham [30] montre que le temps d'exécution T_p sur p processeurs est alors majoré par $T_p \leq \frac{T_1}{p} + T_\infty$. Le surcoût d'ordonnancement est lié à la gestion locale des piles LIFO (RO-local) et aux vols (FIFO-distribué). On suppose que les

coûts d'exécution T_1 et T_∞ prennent en compte le coût de gestion d'une pile locale (exécution sur un processeur selon l'ordre de référence en gérant les dépendances dans la pile respectivement sur un et une infinité de processeurs).

Il reste alors à majorer le surcoût lié aux vols. En l'absence de contention, l'opération de vol sur un processeur peut être réalisé en temps constant ; il suffit, dans chaque pile, de châiner (par un chaînage double) les tâches prêtes, et sur chaque processeur de chaîner similairement les piles qui possèdent des tâches prêtes. Ainsi, le vol d'une tâche sur un processeur prend un temps $O(1)$, et la recherche d'une tâche prête si il en existe sur l'un des processeurs prend un temps $O(p)$.

Pour prendre en compte la contention, on majore le coût comme dans [42] par le cas où tous les vols sur un même processeur sont effectués en exclusion mutuelle ; lié au parcours cyclique des processeurs, le délai pour obtenir une tâche prête lorsqu'il en existe est alors en coût amorti $O(p)$ en pire cas où les tâches prêtes sont sur un même processeur. En choisissant les processeurs à voler de manière aléatoire, l'analyse selon le modèle atomique réalisée dans [9] montre que le nombre de vols est alors majoré par $O(V + p.\log\frac{1}{\epsilon})$ avec une probabilité supérieure à $(1-\epsilon)$ pour tout $\epsilon > 0$; le coût moyen d'un vol est constant en moyenne. Le temps de l'exécution est alors finalement majoré par

$$\frac{T_1}{p} + T_\infty + O(p\,(T_\infty.\Phi_v + \#L.N_\infty)\,.$$

Cette performance est asymptotiquement optimale pour des programmes possédant un très grand degré de parallélisme.

Cependant, si des processeurs sont bloqués suite à un nombre de piles locales excédant $\#L$, cette performance n'est plus vérifiée même si l'ordonnancement se termine en temps fini. En pratique, sauf limitation mémoire, $\#L$ peut être choisi arbitrairement grand pour garantir qu'à chaque top où un processeur est inactif et n'a pas de piles prêtes, il soit autorisé à voler une tâche.

Analyse de l'espace mémoire requis

Soit S_1 la place mémoire requise pour l'exécution séquentielle selon l'ordre de référence. Si l'on suppose que les tâches peuvent allouer dynamiquement de la mémoire en dehors des opérations de création de tâches, l'espace mémoire n'est pas garanti par rapport à l'ordre de référence.

Mais si la place mémoire pour les données est pré-allouée et les calculs effectués en place, le surcoût mémoire nécessaire pour l'ordonnancement est alors la place requise pour les piles. A tout instant de l'exécution le nombre de tâches dans chaque pile est majoré par F_∞. Soit s_t un majorant de la place mémoire nécessaire pour stocker la description d'une tâche est majorée ; la place mémoire requise pour une pile est majorée par $\min(F_\infty.s_t, S_1)$. Lorsque les calculs sont effectués en place, la place mémoire globale requise pour l'ordonnancement est alors majorée par

$$S_1 + p.\#L.\min(F_\infty.s_t, S_1).$$

Cette majoration peut être illustrée sur deux cas pratiques d'utilisation :
- pour les applications numériques qui effectuent des calculs en place (par exemple décomposition de domaines), la place mémoire est majorée par $S_1 + p.\#L.F_\infty.s_t = S_1 + O(p)$.
- Pour les applications de type recherche récursive (par exemple optimisation combinatoire par *Branch&Bound*), la place mémoire peut être majorée par $p.\#L.S_1 = O(p.S_1)$ car, ces applications étant de type série-parallèle, le nombre de piles L peut être limité à une constante petite ($\#L = 2$) sur chaque processeur.

5.2 Ordonnancement statique basé sur un préplacement des données

De nombreux environnements de programmation parallèle comme HPF [32] ou Orca [5] utilisent un regroupement des données du programme dans les différents modules de mémoire disponibles. Pour cela, on considère que chaque donnée est initialement placée dans la mémoire d'un processeur virtuel distinct. Le calcul d'un regroupement consiste à placer chacune de ces données sur les processeurs réels disponibles. Les tâches de calcul sont ensuite placées en fonction des données auxquelles elles accèdent. On dit alors que les tâches sont placées en suivant la règle *Owner Compute Rule* (OCR). Ce type de stratégie est généralement utilisé lorsque le programme considéré manipule des données de grande taille qu'il est coûteux de déplacer. On préfère dans ce cas donner un caractère définitif au placement des données et d'adapter le site d'exécution des tâches plutôt que de déplacer des données. Dans Hpf et Orca le regroupement des données est effectué en découpant les structures de données de grande taille de type tableau et en affectant chaque sous-tableau à un module de mémoire. Dans ces environnements, ce regroupement peut être effectué de plusieurs manières :
- Par bloc : si un tableau de 10000 éléments doit être partitionné pour un ensemble de 50 processeurs virtuels, le tableau sera partitionné en 50 blocs de taille 200.
- Cyclique : si un tableau de 8 éléments doit être partitionné pour un ensemble de 4 processeurs virtuels, le processeur 0 disposera des éléments $0, 4$, le processeur 1 des éléments $1, 5$, le processeur 2 des éléments $2, 6$ et le processeur 3 des éléments $3, 7$.

Ces deux types de regroupement peuvent être combinés pour effectuer un regroupement bloc-cyclique des données. Pour les tableaux à plusieurs dimensions, un type de regroupement peut être spécifié pour chaque dimension.

Nous proposons dans cette section un algorithme d'ordonnancement basé sur le préplacement des données du programme décrit en utilisant la machine abstraite du chapitre 3. Contrairement aux environnements Hpf et Orca, le préplacement des données n'est pas basé sur le partitionnement de tableaux, mais sur le calcul d'un regroupement sur l'ensemble des données partagées accédées par les tâches. Cet al-

gorithme est donc adapté aux programmes de type décomposition de domaines ou simulation de systèmes de particules. Son objectif est d'équilibrer la charge de calcul en calculant un partitionnement en sous-domaines équilibrés tout en réduisant les volumes de données échangées. Dans la suite, cet algorithme est appelé DP-OCR (*Data Partitionning and Owner Compute Rule*).

5.2.1 Calcul d'un graphe de dépendance de données

Le principe de cet algorithme est de construire un graphe de dépendance et d'utiliser une bibliothèque de partitionnement (Scotch [50] ou Metis [39] par exemple) pour calculer un regroupement des sommets du graphe.

Ce type de bibliothèque permet de partitionner des graphes de dépendances de tâches dans lesquels un sommet représente une tâche et une arête représente les communications entre deux tâches (c.f. section 1.1). Notre objectif étant de calculer un regroupement des données de l'application, nous proposons d'utiliser ces bibliothèques pour partitionner un graphe de dépendance de données (et non pas un graphe de dépendance de tâches). La définition de ce type de graphe est la suivante :

Définition 5 *Dans un* ***graphe de dépendance de données*** *chaque sommet représente une donnée partagée déclarée par l'utilisateur. Une arête entre deux sommets représente une dépendance entre deux données. Une arête est construite entre deux sommets si il existe une tâche dans le graphe de flot de données qui prend les deux données partagées en paramètre.*

Une tâche ne pouvant s'exécuter que sur un seul processeur, on exprime par ces dépendances le fait qu'une communication sera nécessaire pour l'exécution de chaque tâche prenant les deux données en paramètre si celles-ci ne sont pas allouées dans la mémoire du même processeur.

La construction du graphe de dépendance de données est effectuée à partir du graphe de flot de données de l'application ainsi que de l'ensemble des données partagées déclarées par l'utilisateur. Cet ensemble peut être parcouru à l'aide d'un itérateur de type `IterToutesLesDonnees`. Cet algorithme peut donc être appliqué de manière statique en utilisant les mécanismes présentés dans la section 4.4.

La figure 5.1 détaille l'algorithme de génération du graphe de dépendance de données à partir du graphe de flot de données construit à l'aide de la machine abstraite de la section 3.2. Les quatre premières lignes de cet algorithme permettent de définir le type `GrapheDepDonnees` représentant le type de structure de données associé au graphe de dépendances de données qu'il faut construire. Cette structure est composée d'un ensemble de sommets et d'un ensemble d'arêtes. L'ajout d'un sommet s'effectue à l'aide de la fonction `ajouter_sommet(NoeudDonnee)` et l'ajout d'une arête s'effectue à l'aide de la fonction `ajouter_arête(NoeudDonnee,NoeudDonnee)`. Les lignes 9 à 14 permettent de construire l'ensemble des sommets du graphe en utilisant l'itérateur sur les noeuds données du graphe de flot de données. Les lignes 16 à 32 permettent de construire l'ensemble des arêtes du graphe. Pour cela, toutes les tâches du graphe de flot de données sont parcourues. Pour chaque tâche, une

arête est ajoutée dans le graphe de dépendance de données pour chaque couple de paramètre de la tâche.

5.2.2 Pondérations du graphe de dépendance de données

A chaque sommet et chaque arête du graphe de dépendance de données, peut être associé un coût qui sera utilisé lors du partitionnement. Dans notre modèle, le coût associé à un sommet représente la quantité de travail associé à la donnée considérée. Ce coût est donc dépendant des tâches du graphe de flot de données qui la prennent en paramètre. Dans notre étude, nous avons choisi d'associer à chaque sommet du graphe de dépendance de données la moyenne des coûts des tâches qui prennent la donnée en paramètre.

Le poids d'une arête correspond au coût qu'il faudra payer si les deux données qu'elle relie ne sont pas placées sur le même site d'exécution. Ce poids dépend donc de la taille des deux données. Dans la suite, nous pondérerons les arêtes en faisant la moyenne des tailles des deux données considérées.

5.2.3 Partitionnement du graphe de dépendance

Une fois le graphe de dépendance de données construit, une bibliothèque de partitionnement est utilisée pour calculer un regroupement des données. Chaque noeud du graphe de dépendance de données est alors associé à un noeud du graphe représentant l'architecture cible en fonction de la stratégie de partitionnement utilisée. Plusieurs bibliothèques permettent d'effectuer cette opération comme par exemple Scotch ou Metis. Dans les expériences qui suivent, nous avons utilisé la bibliothèque Scotch.

5.2.4 Placement des tâches du graphe de flot de données

Les tâches du graphe de flot de données sont ensuite distribuées en fonction des données auxquelles elles accèdent. Chaque tâche prenant plusieurs données en paramètres, le site d'un des paramètre doit être choisie pour y placer la tâche. Ce choix peut être effectué de manière automatique en choisissant le paramètre dont la taille est la plus importante, ou en suivant les directives de l'utilisateur. Pour cela, nous proposons d'ajouter une fonction dans la machine abstraite de la section 3.2 permettant pour chaque clôture allouée de spécifier le paramètre significatif sur lequel sera appliqué la règle *Owner Compute Rule* : `tache.ocr(donnee_parametre_de_la_tache)`. Cette fonction est considérée comme directive ; si elle est utilisée, la tâche sera forcement exécutée sur le site de la donnée spécifiée. La figure 5.2 donne un exemple d'utilisation de cette fonction. Dans cet exemple, la ligne 9 permet de spécifier que la tâche doit de préférence être ordonnancée sur le site qui possède la donnée $d1$.

```
1  structure GrapheDepDonnees{
2    EnsembleSommet sommets;
3    EnsembleAretes aretes;
4  };
5
6  GrapheDepDonnees GDD;
7  GrapheFlotDonnees GFD;
8
9  GFD = le_graphe_de_l_application();
10 IterToutesLesDonnees it_donnees = GFD.DebutToutesLesDonnees();
11 tant que (it_donnees != GFD.FinToutesLesDonnees()) {
12   GDD.sommets.ajouter_sommet(*it_donnees);
13   it_donnees++;
14 }
15
16 IterToutesLesTaches it_taches = GFD.DebutToutesLesTaches();
17 tant que (it_taches != GFD.FinToutesLesTaches() ) {
18   IterTouslesParametres it_param_i =
19       *it_taches.DebutTousLesParamètres();
20
21   tant que (it_param_i != *it_taches.FinTousLesParamètres()) {
22     IterTouslesParametres it_param_j =
23          *it_taches.DebutTousLesParamètres();
24
25     tant que (it_param_j != it_param_i) {
26       GFD.ajouter_arete(*it_param_i, *it_param_j);
27       it_param_j++;
28     }
29     it_param_i++;
30   }
31   it_taches++;
32 }
```

FIG. 5.1 – Construction du graphe de dépendance de données

```
1   ...
2   int d1[10];
3   int d2[10];
4   ...
5   Tache t;
6   t = AllocationCloture( nom_fonction ) ;
7   t.empiler Lecture ( d1 ) ;
8   t.empiler Lecture ( d2 ) ;
9   t.ocr(d1);
10  t.FermerContexte() ;
11  ...
```

FIG. 5.2 – Exemple d'utilisation de la fonction OCR

5.2.5 Exemples

Nous proposons maintenant d'évaluer l'algorithme d'ordonnancement décrit ci-dessus en considérant deux exemples de graphe de flot de données. Le premier exemple est le graphe de l'application Jacobi. Le deuxième correspond au graphe d'un grand nombre d'applications de simulation et en particulier des applications étudiées dans le chapitre 7. Dans le cas de l'application Jacobi, nous évaluerons l'influence de la pondération des noeuds du graphe de flot de données. Pour chacun des ces graphes, nous calculerons la charge de calcul par processeur et nous évaluerons le volume de communications engendrées par l'ordonnancement effectué.

Graphe de Jacobi

Nous proposons dans un premier temps d'étudier l'exemple de l'application Jacobi. Cette application consiste à mettre à jour itérativement chaque sous-domaine en fonction de sa valeur précédente ainsi que de la valeur des sous-domaines voisins. Nous considérons dans ces expériences, la version deux dimensions du programme pour laquelle l'ensemble des sous-domaines est organisé en grille. La représentation du graphe de flot de données de cette version est difficile. Trois dimensions sont en effet nécessaires : deux dimensions liées aux données et une dimension liée au caractère itératif de l'application. Pour des raisons de lisibilité, nous avons choisi de représenter le graphe de flot de données de la version à une dimension du programme dans la figure 5.3. La projection du graphe de flot de données vers un graphe de dépendance de données est illustrée par les figure 5.3 b) et 5.4, respectivement pour la version une dimension et deux dimensions du programme.

Afin d'évaluer les performances de l'ordonnancement DP-OCR sur cette application, nous proposons d'effectuer les expériences suivantes.

Dans un premier temps, nous comparerons les résultats obtenus avec un ordonnancement par bloc sur une grille de taille 64×64. Les coûts des tâches et des

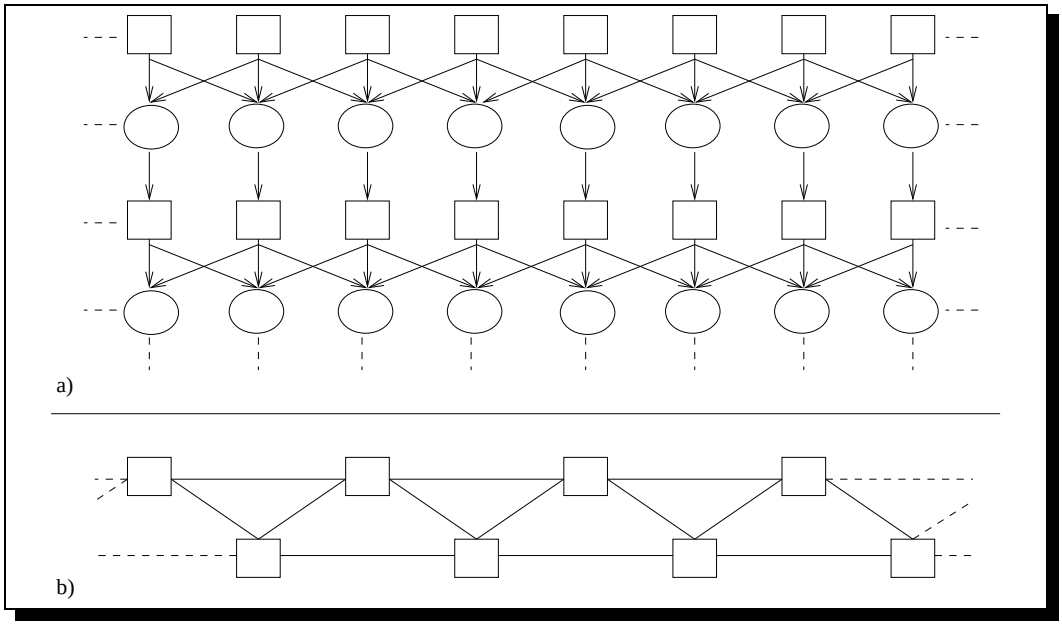


FIG. 5.3 – Graphe de flot de données (a) et graphe de dépendances de données (b) de l'application jacobi une dimension

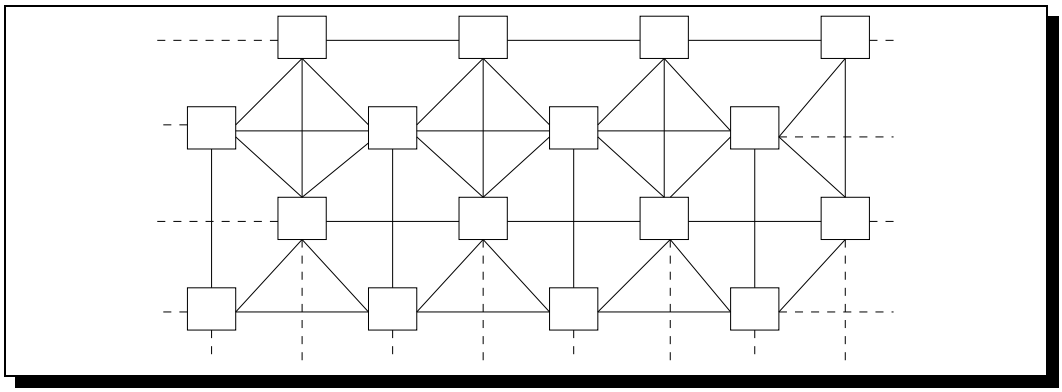


FIG. 5.4 – Graphe de dépendance de données de l'application Jacobi deux dimensions.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Charge	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Données envoyées	1248	1040	1040	1664	1456	624	1456	1456
Données reçues	624	832	832	1456	1664	1248	1664	1664

a) Résultats pour l'ordonnancement DP-OCR

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Charge	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Données envoyées	832	2496	2496	2496	2496	2496	2496	1664
Données reçues	1664	2496	2496	2496	2496	2496	2496	832

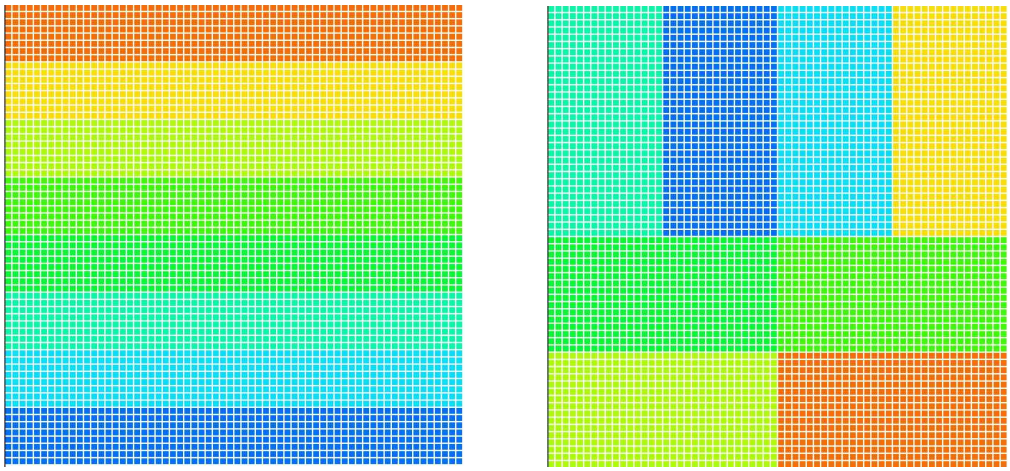
b) Résultats pour l'ordonnancemment bloc cyclique

TAB. 5.1 – Comparaison des résultats en terme de charge de calcul et de volumes de commumnications (en octets) échangées pour les ordonnancement DP-OCR et bloc cyclique.

données sont unitaires pour cette expérience. La figure 5.5 montre le résultat obtenu dans les deux cas. Le tableau 5.1 permet de comparer les pourcentages de charge de calcul alloués à chaque processeur ainsi que les volumes de communications engendrés. Dans les deux cas, l'ordonnancement calculé permet d'équilibrer parfaitement la charge. L'ordonnancement DP-OCR permet toutefois de limiter les échanges de données entre processeurs. Il est à noter que l'ordonnancement effectué privilégie l'équilibrage de charge par rapport à la minimisation des volumes de données échangées. Si un léger déséquilibre de charge est toléré, il est possible de calculer des ordonnancements pour lesquels les volumes de données échangées sont plus faibles.

Nous proposons dans un deuxième temps d'exécuter le même programme en faisant varier les coûts associés aux nœuds du graphe de flot de données. Nous proposons pour cela d'utiliser une fonction de coût qui pourra être appliquée sur les "nœuds" tâches ou les nœuds "données" du graphe de flot de données en fonction de leur placement dans la grilles des sous-domaines. Cette fonction : $\frac{1000}{e^{(x^2+y^2)/100}}$ est représentée par la figure 5.6 Cette expérience permet de valider pour cette application la pondération du graphe de dépendance de données automatique présentée précédemment. Dans un premier temps, cette fonction est appliquée au coût des nœuds "données" du graphe 5.2 a), avant d'être appliquée aux nœuds "tâches" 5.2 b). Le tableau 5.2 montre la répartition de la charge obtenue dans les deux cas. Dans le premier cas, l'équilibrage de charge obtenu est presque optimal. La figure 5.2 a) montre que la fonction de coût a été prise en compte de manière à limiter les volumes de données échangées. La même remarque que pour l'expérience précédent peut toutefois être effectuée : les volumes de données échangées peuvent être réduit si on tolère un déséquilibre de charge plus important.

Lorsque la fonction de coût est appliquée aux tâches, la taille de la grille n'est pas assez importante pour équilibrer parfaitement la charge. 18.58% de la charge totale



a) b)

FIG. 5.5 – Comparaison du préplacement proposé avec un préplacement cyclique et un préplacement par bloc.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Charge	12.55%	12.72%	12.55%	12.55%	12.33%	12.22%	12.55%	12.52%

a) Fonction de coût appliquée aux données

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Charge	11.31%	11.80%	11.31%	11.31%	11.39%	12.51%	18.58%	11.79%

b) Fonction de coût appliquée aux tâches

TAB. 5.2 – Résultats en terme de charge de calcul pour l'ordonnancement DP-OCR lorsque la fonction de coût est appliquée aux données a) et aux tâches b).

de calcul a donc été attribuée à l'un des processeurs alors que la charge attribuée aux autres processeurs varie entre 11.31% et 12.51%.

Ces deux expériences montrent que les coûts attribués aux nœuds du graphe de flot de données sont pris en compte par l'ordonnancement pour équilibrer la charge de calcul, mais aussi pour réduire les volumes de données échangées.

Graphe des applications de simulation

Nous proposons maintenant d'étudier un graphe de flot de données typique des applications de simulations. Les applications de simulation de tombé de tissus et de dynamique moléculaire présentées dans le chapitre 7 sont basées sur un tel graphe. Elle consistent toutes les deux à simuler le comportement d'un ensemble de particules, pour un intervalle de temps donné.

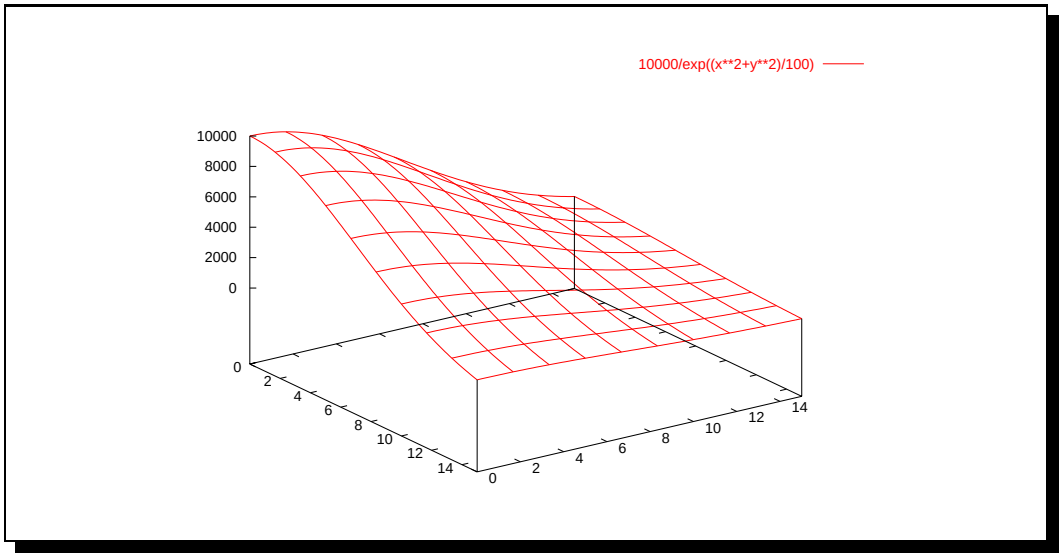
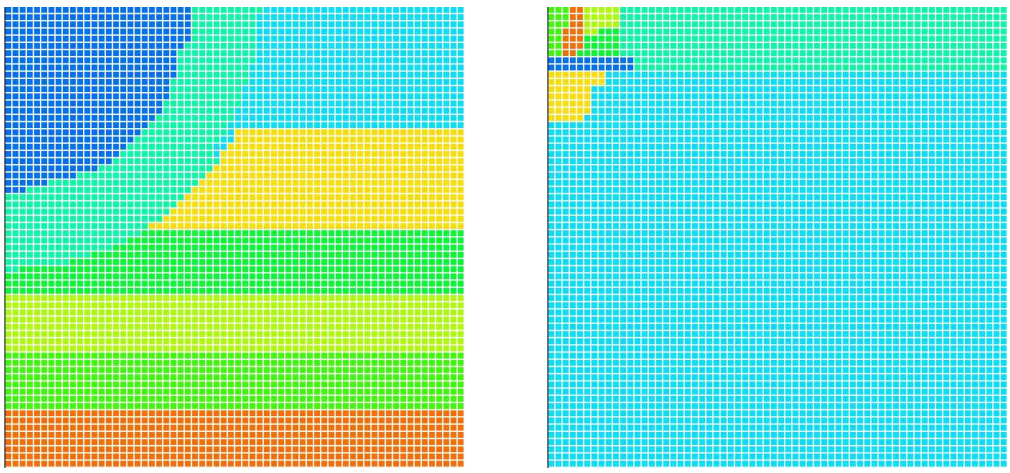


FIG. 5.6 – Fonction de coût appliquée sur les nœuds du graphe de flot de données.



a) b)

FIG. 5.7 – Influence de la pondération du graphe de flot de données sur l'ordonnancement. a) fonction de coût appliquée sur les nœuds "données" du graphe de flot de données, b) fonction de coût appliquée sur les nœuds "tâches" du graphe de flot de données.

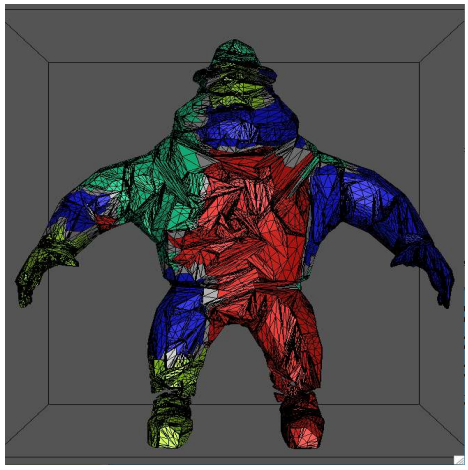


FIG. 5.8 – Application de l'ordonnancement sur 8 processeurs pour le maillage *bigman*.

A chaque particule est associé un état composé de sa position, de la somme de forces auxquelles elle est soumise, de sa vitesse ainsi que de son accélération. A chaque pas de temps, un ensemble de tâches permet de calculer le nouvel état de chaque particule. Ces tâches peuvent prendre en paramètre la particule courante ainsi que les particules voisines avec lesquelles elle interagit.

Nous proposons dans cette expérience, d'appliquer l'ordonnancement DP-OCR sur un système de particule appelé *bigman* composé de 42536 particules. La figure 5.8 montre le système de particule ainsi que l'ordonnancement calculé. Le système est représenté par un maillage pour lequel un sommet représente une particule et une arête représente une interaction entre deux particules. Chaque arête du maillage implique donc la création d'une tâche permettant de calculer les forces induites par cette interaction. Cette tâche prend nécessairement les deux particules concernées ce qui implique la création d'une dépendance de données. Le graphe de dépendance de données est donc identique au maillage représenté.

Le tableau 5.3 montre que la charge attribuée à chaque processeur varie entre 10.88% et 12.82%. L'algorithme DP-OCR permet donc de calculer un ordonnancement correct des tâches du graphe de flot de données pour des systèmes de particules complexes.

L'utilisation de cet algorithme souffre cependant d'un défaut majeur. En effet, le temps nécessaire à la bibliothèque Scotch pour calculer son partitionnement varie de quelques dixièmes de secondes (graphe de jacobi) à plusieurs dizaines de minutes lorsque le graphe de dépendance de donnée est plus complexe. En particulier, 30 minutes ont été nécessaires pour calculer le partitionnement du système de particules *bigman*. Cet algorithme ne peut donc être utilisé que pour des systèmes de particules réduits.

Dans le cadre des applications de simulation, des méthodes permettant de calcu-

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Charge	10.88%	11.46%	11.95%	12.82%	12.52%	11.80%	11.79%	16.78%

TAB. 5.3 – Charge de chaque processeur pour l'ordonnancement présenté dans la figure 5.8

ler, avant la phase de partitionnement, un pré-regroupement en fonction des positions dans l'espace des particules sont à l'étude. L'algorithme DP-OCR ne placerait plus chaque particule mais chaque groupe de particules. Ce pré-regroupement permettrait à la bibliothèque Scotch de considérer des graphes de tailles beaucoup plus réduites et donc de calculer le partitionnement sans pénaliser lourdement le temps total d'exécution de l'application.

5.3 Conclusions

Nous avons présenté dans ce chapitre deux algorithmes d'ordonnancement. Le premier est un algorithme dynamique basé sur des mécanismes de vol de travail sur inactivité. Cet algorithme a la particularité de permettre de borner le nombre de vol et donc d'exportations de tâches effectués lors de l'exécution de tout programme appartenant au modèle déterministe non contraint. Cet algorithme est adapté en particulier aux applications d'optimisation combinatoire comme celle qui sera étudiée dans le chapitre 7.

Le deuxième algorithme d'ordonnancement proposé est un algorithme statique qui permet donc d'utiliser les mécanismes présentés dans la section 4.4. Cet algorithme est basé sur un préplacement des données en utilisant une bibliothèque de partitionnement. Les tâches de l'application sont ensuite distribuées en fonction des données qu'elles prennent en paramètre. Cet algorithme est adapté aux applications manipulant des données de grande taille qu'il est coûteux de déplacer comme par exemple les applications de simulation de tombé de tissus et de dynamique moléculaire présentées dans le chapitre 7.

Troisième partie

Implantation pour Athapascan et expérimentations

CHAPITRE 6

Implantation distribuée pour Athapascan et validation

Ce chapitre est dédié à la description de l'implantation des mécanismes présentés dans le chapitre 4 dans une nouvelle implantation du langage Athapascan. Cette implantation est basée sur le noyau exécutif Inuktitut [46] qui propose une interface pour l'utilisation de processus légers ainsi que d'une interface de communication basée sur des messages actifs. Les particularités de ces outils seront détaillées dans la première section.

De nombreux algorithmes centralisés ou distribués cohabitent dans cette nouvelle implantation. Il est donc impossible dans le cadre de cette thèse de proposer une description complète de tous ces algorithmes. Pour cette raison, nous nous contenterons de détailler dans un premier temps l'implantation des accès différés et de décrire, dans un deuxième temps, l'algorithme de terminaison distribué utilisé.

Dans la deuxième section, nous proposons de valider cette implantation sur des applications simples. Dans un premier temps, nous étudierons deux applications récursives *Knary* et *Diamant* qui permettront d'appliquer l'ordonnancement par vol de travail décrit dans la section 5.1. Nous évaluerons ensuite deux applications itératives *Jacobi* et *l'élimination de Gauss* qui permettront d'appliquer l'ordonnancement décrit dans la section 5.2.

6.1 Implantation

6.1.1 Processus légers et communications : Inuktitut

Le noyau exécutif Inuktitut[46] est une bibliothèque C++ destinée à l'exploitation de grappes de SMP. Elle est composée d'une interface de processus légers permettant d'exploiter le parallélisme intra-noeud et d'une interface pour la gestion de communications unidirectionnelles par messages actifs permettant d'exploiter le parallélisme inter-noeuds.

L'interface de processus légers est inspirée à la fois des processus légers posix et des processus légers windows et permet la création et la synchronisation des processus légers à l'aide de verrous et de variables de condition.

L'interface de communication est basée sur la notion de message actif. Ce type de message a pour principal avantage d'éviter la mise en place de rendez-vous (association d'un envoi à une réception). Le comportement d'un message actif est proche de celui d'un appel de procédure à distance (RPC). Il est composé de l'identificateur d'une procédure à exécuter et d'un ensemble de données qui peuvent éventuellement

être prises en paramètre par la procédure. Cependant aucun message de retour n'est envoyé après réception d'un message actif. Ces messages sont donc unidirectionnels et asynchrones.

Trois types de messages actifs sont proposés dans l'interface :

- Message actif standard : les données communiquées sont allouées dans la pile du processeur cible. La mémoire associée aux paramètres est donc systématiquement libérée après l'exécution de la procédure.
- Écriture et signalisation : les données communiquées sont allouées dans une zone mémoire définie par l'utilisateur. Elles restent donc disponibles après l'exécution de la procédure. L'utilisateur a la charge de désallouer correctement ces données.
- Allocation, écriture et signalisation : les données communiquées sont allouées dans le tas par Inuktitut. L'utilisateur a la charge de désallouer correctement ces données.

En plus des message actifs unidirectionnels, la bibliothèque Inuktitut permet l'utilisation de primitives de communications collectives basées sur des algorithmes passant à l'échelle (*chaînes*, *arbres* α, *arbres plats*).

Plusieurs implantations de l'interface de communication d'Inuktitut sont actuellement disponibles :

- Implantation TCP/IP basée sur l'utilisation de sockets.
- Implantation Corba.

Les implantations TCP/IP et Corba ont été testées et validées dans le cadre du développement d'Athapascan. La version Myrinet est en cours de finalisation. Toutes les expériences effectuées dans la suite ont été réalisées avec la version TCP/IP de la bibliothèque.

6.1.2 Implantation des accès différés

Nous proposons maintenant de détailler l'implantation des accès différés dans cette nouvelle version de la bibliothèque Athapascan. Les accès non différés du langage sont automatiquement gérés par les mécanismes de pile distribuée présentés dans le chapitre 4. Dans ces mécanismes, le chaînage des accès aux données partagées permet d'utiliser des primitives de communication unidirectionnelles pour les échanges de données entre tâches "soeurs".

Les accès différés permettent d'exprimer des dépendances de données entre tâches "cousines". Les accès différés étant des accès concurrents, la génération des producteurs et des consommateurs d'une même version d'une donnée partagée peut éventuellement être effectuée en parallèle. Il est donc impossible de générer des communications unidirectionnelles puisque les producteurs ne connaissent pas les sites d'exécutions des consommateurs.

La figure 6.1 donne un exemple d'utilisation des accès différés. Dans cet exemple, la production et la consommation de la donnée i sont différés. Après l'exécution de la fonction `main` les tâches `tache_ecriture_differee` et `tache_lecture_differee`

```
1  struct tache_ecriture {
2    void operator() (Shared_w<int> i) { ... }
3  };
4  struct tache_ecriture_differee {
5    void operator() (Shared_wp<int> i)
6       { Fork<tache_ecriture> () (i); }
7  };
8
9
10 struct tache_lecture {
11   void operator() (Shared_r<int> j) { ... }
12 };
13 struct tache_lecture_differee {
14   void operator() (Shared_rp<int> j)
15      { Fork<tache_lecture> () (j); }
16 };
17
18
19 void main() {
20   Shared<int> i;
21   Fork<tache_ecriture_differee> () (i);
22   Fork<tache_lecture_differee> () (i);
23 };
```

FIG. 6.1 – Un exemple d'utilisation des accès différés.

sont donc toutes les deux prêtes et peuvent s'exécuter en parallèle. La synchronisation entre les tâches `tache_ecriture` et `tache_lecture` doit cependant être gérée.

Utilisation des I-structures

Afin de résoudre ce problème, nous avons choisi d'implémenter les accès différés à l'aide de structures de données proches des *I-structures* présentées dans [3]. Une I-stucture est une structure de données permettant la synchronisation d'une écriture unique avec un ensemble de lecteurs. Dans notre implantation, elle est composée de :

- Un site de référence(`site_ref`).
- Un identificateur valide sur le site de référence permettant de retrouver la I-structure concernée lors de la réception d'un message.
- Une adresse locale pointant sur une zone de mémoire permettant de stocker la donnée partagée(`addr`).
- Un booléen dont la valeur devient VRAI lorsque la donnée peut être consommée (`bool_prod`).

- Un ensemble de site *lecteurs* en attente de la production de la donnée (`liste_lecteurs`).

Chaque site participant à la production ou à la consommation d'une donnée partagée *à accès différé* dispose d'une copie de la même I-structure. Cependant, les attributs `bool_prod` et `liste_lecteurs` ne sont valides que sur le site de référence et reste inutilisés sur les autres sites.

Outre les attributs décrits ci-dessus, chaque I-structure dispose de deux méthodes :

- `Produire()` : la valeur est envoyée sur le site de référence. Sur ce site, la valeur VRAI est donnée au booléen `bool_prod` et la donnée est envoyée à tous les lecteurs de la liste `liste_lecteurs` qui peuvent alors continuer leur exécution.
- `Consommer()` : une requête de lecture est envoyée au site de référence. Si la donnée est déjà produite, le site de référence répond en envoyant la donnée. Si la donnée n'est pas produite, l'identificateur du site lecteur est ajouté à la liste des lecteurs en attente du site de référence.

Après réception d'une donnée partagée envoyée à travers l'une des deux méthodes des I-structures, l'attribut `addr` est positionné de manière à pointer sur la donnée reçue.

Lorsque deux tâches cousines doivent se synchroniser sur une donnée partagée, une nouvelle I-structure est créée. Cette I-structure doit être associée aux tâches productrices comme aux tâches consommatrices de cette donnée partagée. La structure de donnée "accès" interne à une clôture 4 contient donc un pointeur qui prend la valeur de l'adresse de la I-structure associée. Dans le cas des accès en modification (rw) une synchronisation à l'aide d'une I-structure peut être nécessaire pour la partie "lecture" comme pour la partie "écriture" de cet accès. Deux I-structures peuvent dans ce cas être associées à l'accès en question.

Lorsqu'une tâche consommatrice d'une donnée partagée (accès non différé) à laquelle est associée une I-structure vient d'être créée, la méthode `Consommer()` de la I-structure est appelée. La tâche ne sera considérée comme "prête" que lorsque la donnée aura été produite et diffusée par le site de référence de la I-structure. Lorsqu'une tâche productrice d'une donnée partagée à laquelle est associée une I-strucutre termine son exécution, la méthode `Produire()` de la I-structure est appelée.

Lorsqu'une tâche est volée, les I-structures associées à chaque paramètre sont recopiées et envoyée sur le site "voleur" en même temps que la tâche volée. Le site voleur peut ainsi faire appel aux méthodes des I-structures de manière transparente.

Création des I-structures

La création des I-structures est effectuée au moment de la création des tâches. Pour chaque "accès" à une donnée partagée déclaré par la tâche, on cherche à déterminer si une synchronisation doit avoir lieu en fonction de l'accès précédent dans l'ordre de référence.

L'exemple de programme de la figure 6.2 ainsi que sa trace d'exécution (figure 6.3) permettent d'illustrer la génération automatique des I-structures permettant

```
1  struct tache_modification {
2    void operator() (Shared_rw<int> i) { ... }
3  };
4  struct tache_modification_diff {
5    void operator() (Shared_rwp<int> i, int seuil, int prof) {
6       if (profondeur < seuil)
7          Fork<tache_modification_diff> ()(i, seuil, prof+1);
8       else
9          Fork<tache_modification> () (i);
10   }
11 };
12
13 void main() {
14   Shared<int> i;
15   int seuil = 2;
16   Fork<tache_modification_diff> () (i, seuil, 0);
17   Fork<tache_modification_diff> () (i, seuil, 0);
18 };
```

FIG. 6.2 – Un exemple d'utilisation des accès différés.

de synchroniser les écritures et les lectures d'une donnée lorsque des accès différés sont utilisés. Au premier niveau de récursivité deux tâches sont créées ainsi qu'une I-structure qui permettra de synchroniser les tâches créées par la première avec celles créées par la deuxième. Lors de leur exécution, ces deux tâches créent à leur tour deux autres tâches accédant la donnée partagée en modification différée (deuxième niveau de récursivité). De nouvelles I-structures sont alors créées de la même manière qu'au premier niveau de récursivité.

Au troisième niveau de récursivité, les tâches effectuant les accès sont exécutées. Quel que soit leur site d'exécution, ces tâches seront synchronisées grâce aux méthodes `Lecture()` et `Ecriture()` des I-structures.

6.1.3 Algorithme de terminaison distribué

La détection distribuée de la terminaison d'un programme est un point crucial pour l'implantation de la plupart des programmes parallèles. L'objectif est d'implanter un algorithme réactif et le moins intrusif possible. La définition 6 permet de définir de manière formelle le concept de terminaison distribuée d'un programme Athapascan.

Définition 6 *On considère que le programme a terminé son exécution si plus aucun processeur n'exécute de tâche et qu'aucun message n'est en transit sur le réseau.*

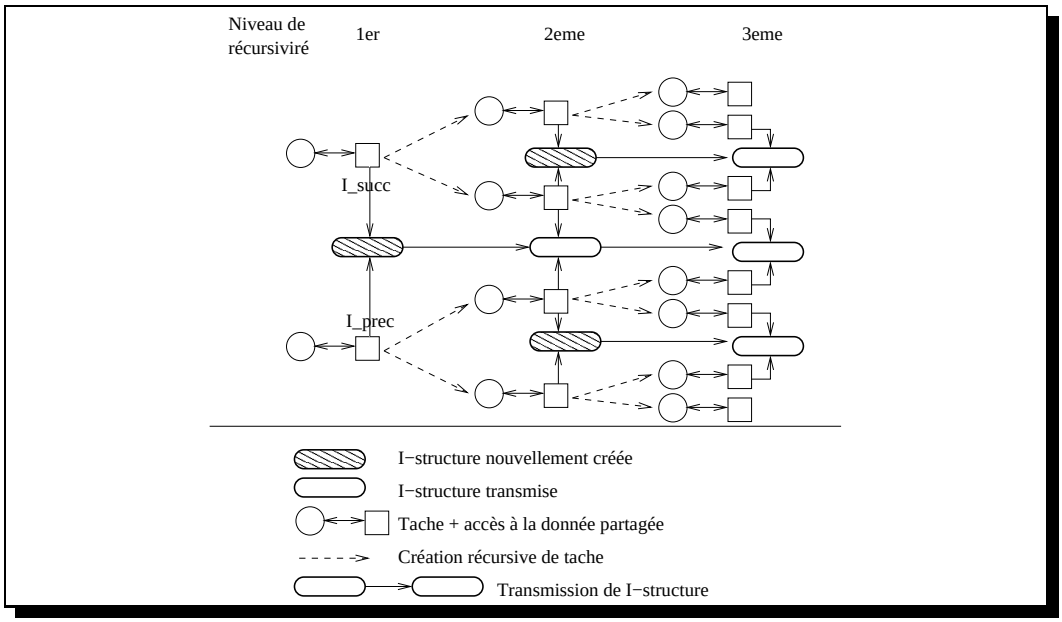


FIG. 6.3 – Représentation de l'exécution du programme de la figure 6.2.

Dans le cas du fonctionnement d'Athapascan présenté dans la section 4.4 lorsque des algorithmes d'ordonnancement statiques sont utilisés, la détection de la terminaison est simplifiée. En effet, si un message est en transit, il existe forcement une tâche en attente de cette donnée sur le site destinataire du message. La terminaison peut donc être simplement détectée si tous les processeurs ont exécuté toutes les tâches qui leur ont été attribuées.

Dans le cas des ordonnancements dynamiques, la détection est un peu plus difficile. Même si tous les noeuds sont inactifs à un moment donné, une tâche peut être en cours de migration suite à une requête de vol et donc relancer le calcul.

L'algorithme que nous proposons est basé sur l'algorithme de Rana présenté dans [63]. Le principe de cet algorithme est d'acquitter chaque migration de tâche. Chaque noeud peut être dans l'état *actif*, *inactif* ou *terminaison locale*. Si un noeud est actif, une tâche est en cours d'exécution sur ce noeud. Si il est inactif, aucune tâche n'est en cours d'exécution mais des tâches sont en attente de données ou une migration de tâche n'a pas été acquittée. Si un noeud est dans l'état *terminaison locale*, aucune tâche n'est en cours d'exécution sur ce noeud, aucune tâche n'est en attente et toutes les migrations ont été acquittées.

Les noeuds sont virtuellement organisée en anneau monodirectionnel, lorsqu'un noeud rentre dans l'état *terminaison locale*, il interroge son successeur dans l'anneau. Si celui-ci est aussi dans l'état *terminaison locale* il propage à son tour le message. Si le message revient à son initiateur, tous les noeuds sont alors dans l'état *terminaison locale* et la terminaison est détectée.

Cet algorithme est décrit dans la figure 6.1.3. On considère qu'il est exécuté de la même manière sur chaque noeud participant au calcul. A chaque noeud est associé un identificateur. L'identificateur du noeud local est obtenu en lisant la variable `id_noeud_local` Le successeur dans l'anneau est obtenu à l'aide de la fonction `Suivant()`. Trois type de message interviennent dans l'algorithme :

- Migration de tâche : identifié par le tag *Migr*
- Acquittement de migration : identifié par le tag *Acq*
- Détection de terminaison : identifié par le tag *Term*

Cet algorithme à l'avantage d'être réactif. En effet, il faut un tour d'anneau pour détecter la terminaison soit $\#processeurs + 1$ messages. Il est aussi peu intrusif. Lorsqu'un noeud lance prématurément une détection, celle-ci sera stoppée lorsqu'elle rencontrera le premier noeud actif sur l'anneau.

6.2 Validation : applications récursives

Afin de valider cette nouvelle implantation d'Athapascan, nous proposons dans un premier temps d'étudier deux applications récursives. Ces applications permettent en particulier de tester l'algorithme d'ordonnancement dynamique basé sur le vol de tâches. Nous étudierons dans un premier temps l'application Knary avant d'évaluer la bibliothèque avec l'application Diamant.

6.2.1 Knary

L'application Knary permet de calculer récursivement les éléments de la suite :

$$F(n) = F(n-1) + F(n-2) + ... + F(n-k)$$
$$F(0) = 0,\ F(1) = 1$$

Cette application sera validée en deux étapes. Nous étudierons tout d'abord son comportement sur une architecture à mémoire partagée de grande taille. Cette première expérience nous permettra de comparer les performances de la bibliothèque Athapascan avec la bibliothèque Cilk. Nous étudierons ensuite son comportement sur architecture à mémoire distribuée ce qui nous permettra de comparer les résultats obtenus avec ceux de la version 1 d'Athapascan décrite dans la section 2.3.

Comparaison Athapascan/Cilk

Cette expérience 6.5 est effectuée sur une Origin3800 ayant 256 processors utilisant la version 6.4 du système d'exploitation Irix. Les paramètres de l'application utilisés sont $n = 28$ et $k = 10$. Afin d'adapter la granularité, les expériences sont effectuées avec un seuil d'arrêt de la découpe récursive égal à 10 ou à 12, ce qui correspond à une granularité de respectivement 20 et 60 secondes. Cette expérience permet de comparer deux environnements qui implantent des mécanismes de dégénération

Variables locales :

```
Etat{actif, inactif, terminaison_locale}
int Date = 0;              // Horloge logique
int Nb_non_acquittes = 0; // Envoi de tâches non acquittées
int Date_inactif = 0;      // Date du dernier passage à l'état de terminaison lo
```

Decision de migration d'une tâche vers le processeur p :

```
Date = Date + 1;
Envoi(Migr, t, id_noeud_local, Date) à p;
Nb_non_acquittes = Nb_non_acquittes + 1;
```

Reception d'une tâche t en provenance de p, Date du message = d :

```
Date = max (Date, d) + 1; //recalage de l'horloge logique
Envoi(Acq, Date) à p;
Etat = actif;
```

Fin de l'exécution de la dernière tâche présente sur le noeud :

```
Date = Date + 1;
Etat = inactif;
si (Nb_non_acquittes == 0) { // passage à l'état de terminaison locale
  Etat = terminaison_locale
  Date_inactif = Date;
  Envoi (Term, Date, Date_inactif, id_noeud_local) à Suivant();
}
```

Reception d'un message (Acq, d) :

```
Date = max (Date, d) + 1; //recalage de l'horloge logique
Nb_non_acquittes = Nb_non_acquittes - 1;
si ( Nb_non_acquittes == 0 && Etat == inactif) {
  Date_inactif = Date;
  Envoi (Term, Date, Date_inactif, id_noeud_local) à Suivant();
}
```

Réception d'un message <Term,d,d_inactif,p> :

```
Date = max (Date, d) + 1; //recalage de l'horloge logique
si (etat == terminaison_locale) {
  si (p = id_noeud_local) TERMINAISON DISTRIBUEE DETECTEE
  sinon si (d >= Date_inactif)
    Envoi (Term, d, d_inactif, p) à Next();
}                        122
```

FIG. 6.4 – Algorithme de terminaison distribuée implanté dans Athapascan.

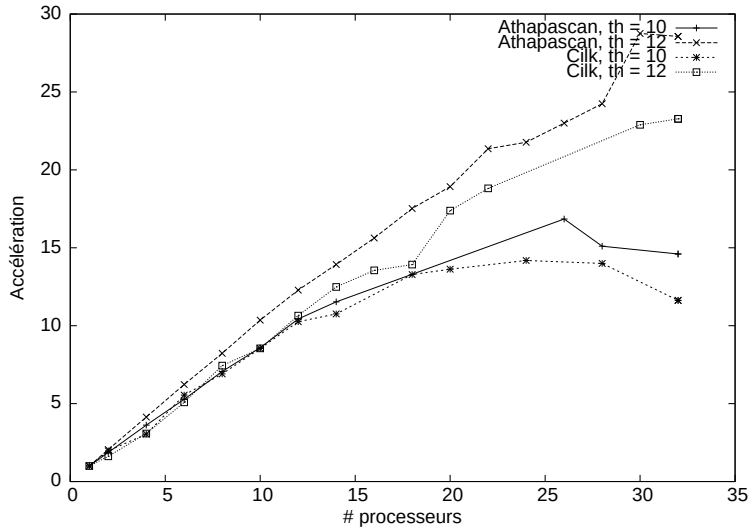


FIG. 6.5 – Comparaison Athapascan/Cilk

séquentielle. On peut constater qu'Athapascan a des performances comparables voire meilleures que Cilk sur ce type d'architecture. Lorsqu'on augmente la granularité, la gestion du parallélisme est dans les deux cas négligeable devant les coûts de calcul de l'application. Les performances sont dans ce cas identiques.

Comparaison Athapascan (version 1)/ Athapascan (version 2)

Afin d'évaluer le gain de l'utilisation des mécanismes de dégénération séquentielle et distribuée implantés dans la version 2 d'Athapascan, nous avons comparé les performances obtenues avec les deux versions de la bibliothèque. Cette expérience est effectuée sur un cluster de 100 machines monoprocesseur de type Pentium III, cadencées à 733Mhz, disposant de 256Mo de mémoire et connectées par une réseau fast Ethernet 100Mo/s. De la même manière que pour l'expérience précédente, des seuils d'arrêt de la découpe récursive de 10 ou 15 sont utilisés afin d'adapter la granularité. On peut constater qu'à grain fin (10), la version 2 de la bibliothèque continue à avoir des performances proches de l'optimal alors que celles de la version 1 s'écroulent. Un gain significatif est donc apporté par cette nouvelle implantation.

6.2.2 Diamant

Nous proposons maintenant d'étudier une application basée sur un graphe de flot de données en forme de diamant (6.7 partie gauche). Plusieurs écritures de cette application sont possibles. Quelle que soit l'écriture choisie, deux types de synchronisations interviennent lors de l'exécution

- Les synchronisations liées à la description du graphe.
- Les synchronisations liées aux dépendances de données du programme.

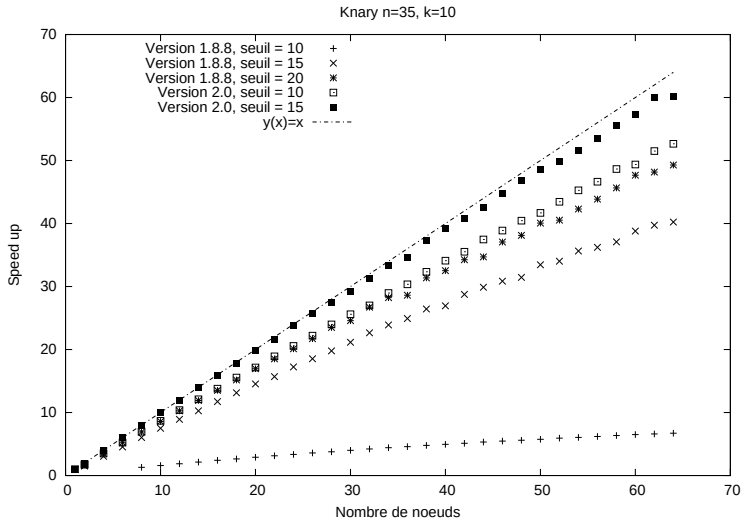


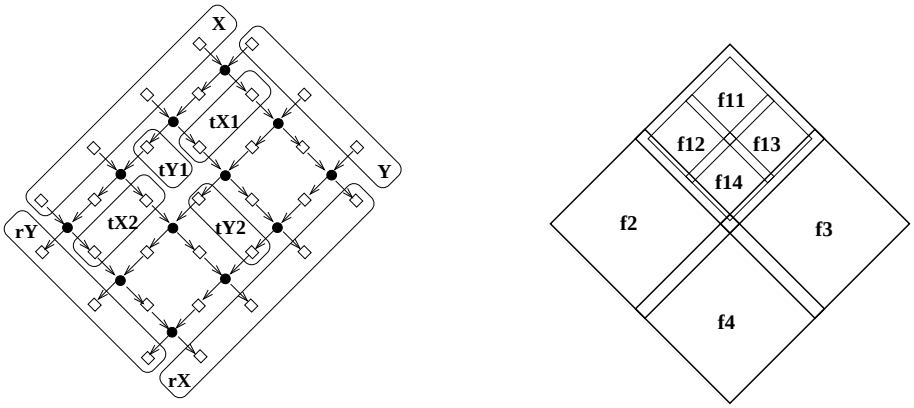
FIG. 6.6 – Comparaison Athapascan (version 1)/Athapascan (version 2)

En particulier l'utilisation d'un langage de programmation du modèle déterministe non contraint (Athapascan) permet d'exprimer plus de parallélisme et donc moins de synchronisations qu'un langage de programmation du modèle strict (Cilk). La figure 6.8 propose une écriture basée sur les accès différés d'Athapascan. Ce programme est basé sur le principe "diviser pour réigner". Deux types de tâches sont implantés :

- `Diamant-seq` qui permet d'effectuer le calcul sur un bloc séquentiel.
- `Diamant-par` qui permet de découper récursivement le calcul en quatres nouvelles tâches `Diamant-par` travaillant sur des parties plus petites du graphe diamant.

La figure 6.7 illustre la découpe récursive effectuée et permet de définir les notations utilisées par le programme de la figure 6.8. Chaque tâche prend quatre données partagées en paramètre de type tableau : deux en lecture et deux en écriture. Lors de la découpe récursive, les quatre tableaux (X, Y, rX, rY) sont coupés en deux parties et quatres tableaux intermédiaires (tX1, tX2, tY1, tY2) sont créés permettant de créer les quatre nouvelles tâches travaillant sur des domaines plus petits.

La figure 6.2.2 montre les résultats obtenus pour cette application sur la grappe (partie a) et sur l'Origin 3800(partie b). Cette dernière courbe permet de comparer les résultats obtenus avec le même programme écrit dans le langage Cilk. Ces résultats mettent en évidence l'intérêt d'utiliser un langage de programmation basé sur le modèle déterministe non contraint. Le programme écrit en Cilk n'exprime pas suffisamment de parallélisme pour exploiter efficacement plus de 6 processeurs.



a) Découpage des données et notations. b) Découpe récursive.

FIG. 6.7 – Découpages et notations utilisées dans le programme de la figure 6.8

6.3 Validation : applications itératives

Nous proposons maintenant d'étudier deux applications itératives permettant de valider les mécanismes présentés dans la section 4.4 ainsi que l'ordonnancement présenté dans la section 5.2. Pour cela nous évaluerons dans un premier temps l'application Jacobi pour un domaine à 2 dimensions avant de proposer les performances obtenues pour une élimination de Gauss. Cette application nous permettra de comparer les résultats d'Athapascan avec la bibliothèque Linpack.

6.3.1 Jacobi

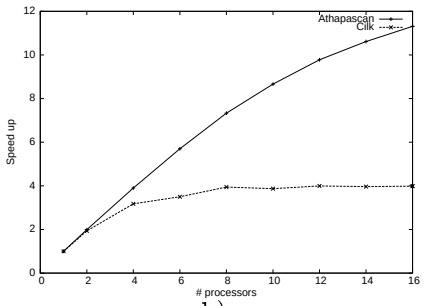
Cette première expérience a pour but d'évaluer le surcoût de la bibliothèque Athapascan lorsqu'un ordonnancement statique est utilisé. Pour cela, nous proposons de comparer les accélérations obtenues avec l'application Jacobi pour plusieurs granularité des tâches possibles. Cette expérience a été effectuée sur un domaine deux dimensions de taille 64×64. Le temps séquentiel de référence est celui d'une exécution du même algorithme écrit directement en C++ (sans Athapascan). L'ordonnancement des tâches est effectué par l'algorithme DP-OCR. Les temps parallèles n'incluent pas le temps de calcul de l'ordonnancement. La machine utilisée est composée de 16 bi-processeurs Pentium IV Xéon 2,7Ghz disposant chacun de 1,5Go de mémoire et interconnectés par un réseau Ethernet Gigabit. La figure 6.10 montre les accélérations obtenus pour cette application pour trois granularités de tâches différentes : 0.01, 0.005 et 0.001 secondes. Ces résultats montrent que le surcoût du noyau exécutif d'Athapascan est imperceptible pour les granularités supérieures à 0.005 secondes. Pour une granularité de 0.001 secondes, les surcoûts liés aux communications engendrées commencent à pénaliser le passage à l'échelle.

```
1  procedure Diamant-seq( int x, y;
2                int rx, ry )
3  {  rx = ry = calcul-local ( x, y ) ; }
4
5  procedure Diamant-par( int nX, nY ;
6           int X[0..nX-1], Y[0..nY-1] ;
7           int rX[0..nX-1], rY[0..nY-1] )
8  {
9    if (nX == 1) && (nY == 1) {
10     Tache f0 = AllocationCloture( compute ) ;
11     f0.empiler::Lecture( X[0], Y[0] ) ;
12     f0.empiler::Ecriture( rX[0], rY[0] ) ;
13     f0.FermerContexte();
14   }
15   else {
16     int tX1[0..nX/2-1], int tX2[0..nX-nX/2] ;
17     int tY1[0..nY/2-1], int tY2[0..nY-nY/2] ;
18
19     Tache f1 = AllocationCloture( Diamant-par );
20       f1.empiler::Valeur( nX/2, nY/2 ) ;
21       f1.empiler::Lecture( X[0..nX/2-1], Y[0..nY/2-1] ) ;
22       f1.empiler::Ecriture( tX1[0..nX/2-1], tY1[0..nY/2-1] ) ;
23       f1.FermerContexte();
24     Tache f2 = AllocationCloture( Diamant-par );
25       f2.empiler::Valeur( nX - nX/2, nY/2 ) ;
26       f2.empiler::Lecture( X[nX/2..nX-1], tY1[0..nY/2-1]);
27       f2.empiler::Ecriture( tX2[0..nX-nX/2-1], rY[0..nY/2-1]);
28       f2.FermerContexte();
29     Tache f3 = AllocationCloture( Diamant-par );
30       f3.empiler::Valeur( nX/2, nY - nY/2 ) ;
31       f3.empiler::Lecture( tX1[0..nX/2-1], Y[nY/2..nY-1]);
32       f3.empiler::Ecriture( rX[0..nX/2-1], tY2[0..nY-nY/2-1]);
33       f3.FermerContexte();
34     Tache f4 = AllocationCloture( Diamant-par );
35       f4.empiler::Valeur( nX - nX/2, nY - nY/2 ) ;
36       f4.empiler::Lecture( tX2[0..nX-nX/2-1], tY2[0..nY-nY/2-1]);
37       f4.empiler::Ecriture( rX[nX/2..nX-1], rY[nY/2..nY-1]);
38       f4.FermerContexte();
39 } }
```

FIG. 6.8 – Programation récursive d'un graphe diamant de taille nX× nY.

Inputs $T_1 = 252.42$ $T_S = 248.64$ 128×128 blocs $sz\_bloc = 400$	T_p	T_1/T_p	T_S/T_p
diamond, p=4	98.80	2.55	2.52
diamond, p=8	61.12	4.13	4.06
diamond, p=16	53.64	4.71	4.64
diamond, p=32	37.13	6.80	6.70

a)



b)

FIG. 6.9 – Résultats obtenus pour l'application Diamant sur le cluster a) et sur l'origin 3800 b)

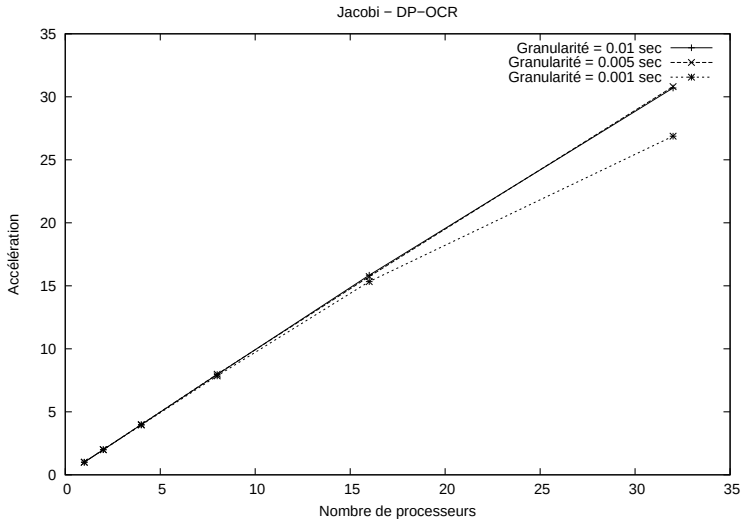


FIG. 6.10 – Accélération pour l'application Jacobi pour plusieurs granularités.

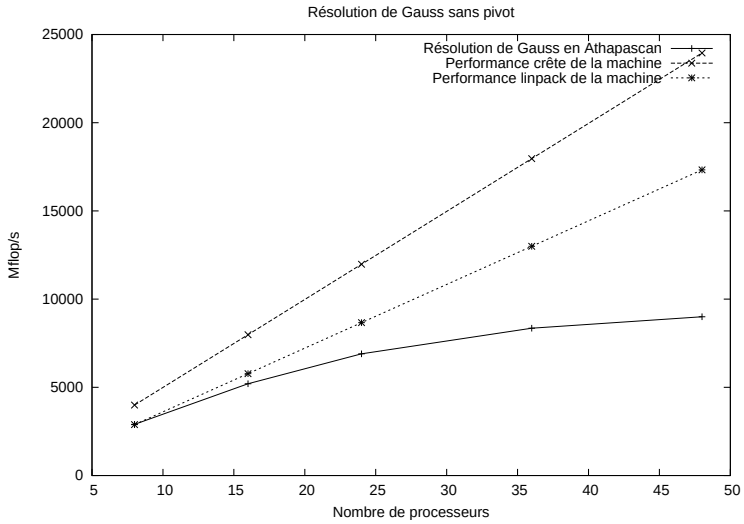


FIG. 6.11 – Résolution de Gauss sur un cluster de 50 machines.

6.3.2 Élimination de Gauss

Enfin, nous proposons de comparer les performances obtenues par Athapascan sur une élimination de Gauss avec la bibliothèque Linpack. Cette expérience est effectuée sur un cluster de 50 mono-processeurs Pentium III cadencés à 733Mhz, disposant de 256 Mo de mémoire et interconnectés par un réseau Fast Ethernet 100Mo/s. Le calcul est effectué sur une matrice de taille 10240 découpée en blocs de taille 256. Les résultats sont comparés avec la performance crête de la machine (499 Mflop/s par processeur) ainsi qu'a la performance obtenu avec l'application *linpack* (361 MFlop/s par processeur) [45]. Les performances obtenues sont correctes mais peuvent être améliorées. En effet, cette application devrait permettre d'exploiter pleinement la puissance de calcul des processeurs de manière comparable à l'application linpack (même si les deux algorithmes sont différents). Des améliorations peuvent être apportées en utilisant des primitives *blas* optimisées pour le processeur cible. Dans le cadre de cette expérience, les *blas* ATLAS ont été utilisées. De nouvelles mesures basées sur l'utilisation des primitives *blas* MKL (Math Kernel Libray) développées par intel devrait permettre d'obtenir de meilleurs performances sur les processeurs dont nous disposons.

6.4 Conclusions

Dans ce chapitre, nous avons présenté quelques spécificités de l'implantation de la nouvelle implantation d'Athapascan. Nous avons décrit dans un premier temps l'implantation des accès différés aux objets partagés avant de décrire l'algorithme de terminaison distribué utilisé. Dans une deuxième section, nous avons décrit une série

d'expériences permettant de valider cette nouvelle implantation. Ces expériences sont divisées en deux catégories d'applications :

- Les applications récursives permettant de valider entre autre l'ordonnancement dynamique de la section 5.1.
- Les applications itératives permettant de valider entre autre l'ordonnancement de la section 5.2

Ces applications sont des applications simples dont les codes ne dépassent pas quelques centaines de lignes. Le chapitre suivant est consacré à des expérimentations sur des applications beaucoup plus importantes permettant de valider la bibliothèque dans un contexte beaucoup plus réaliste.

CHAPITRE 7

Applications

Les expérimentations du chapitre précédent ont permis de valider l'implantation d'Athapascan sur des exemples simples. Nous proposons dans ce chapitre d'étudier le comportement de la bibliothèque dans le cas d'applications de plus grande envergure. Parmi les applications les plus coûteuses en puissance de calcul, les applications de simulation tiennent une part très importante. Des architectures parallèles de grande taille sont donc souvent nécessaires pour simuler des phénomènes de plus en plus complexes. Afin de valider la bibliothèque Athapascan pour ce type d'application, nous avons implanté deux d'entre elles afin d'en évaluer le comportement. La première permet de simuler le comportement de tissus en mouvements comme par exemple la chute d'un mouchoir ou les déplacements des vêtements portés par une personne. Cette application a fait l'objet d'une thèse [68] et d'une collaboration entre le laboratoire GRAVIR et le laboratoire ID-imag.

La deuxième est une application de dynamique moléculaire permettant de simuler le comportement de molécules sur de très courtes périodes et dans des conditions qu'il est difficile d'observer expérimentalement. Cette application a aussi fait l'objet de précédents travaux [7, 6] qui ont été repris pour effectuer les expériences présentées dans ce chapitre.

Dans un deuxième temps, nous nous intéresserons à l'application QAP (Quadratic Assignment Problem). Cette application est une application d'optimisation combinatoire implantant un algorithme de type *Branch-and-Bound*. Ce type d'algorithme a l'avantage d'être très parallèle, ce qui nous a permis d'effectuer les expérimentations sur une architecture de type grille de calcul.

7.1 Applications de simulation

7.1.1 Simulation de tombé de tissus : Sappe

La première application que nous allons étudier est une application de simulation de tombé de tissus appelée Sappe [70, 69, 68, 54]. L'objectif principal de ce projet consiste à paralléliser les méthodes de simulation existantes afin de permettre la simulation de systèmes de grandes tailles en temps réel.

L'application Sappe est basée sur la modélisation du tissus simulé par un maillage de particules interconnectées par des ressorts. Ce maillage permet pour chaque particule de calculer les forces auxquelles elle est soumise à partir de sa position, de sa vitesse ainsi que de celles des particules voisines. L'équation du mouvement de Newton permet ensuite de calculer la trajectoire résultante. La figure 7.1 montre le

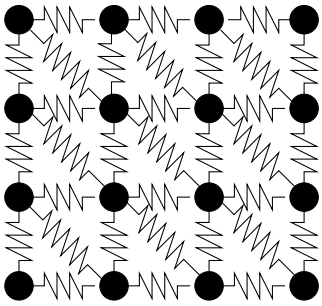


FIG. 7.1 – Maillage associé à un mouchoir de 4×4 particules.

maillage d'un mouchoir composé de 4×4 particules.

La simulation consiste pour chaque pas de temps à calculer le nouvel état de chaque particule. L'état d'une particule est composé de sa position, de sa vitesse et de son accélération. La loi fondamentale de la dynamique énoncée par Newton permet de le calculer. Chaque pas de temps peut donc être décomposé en deux étapes :

1. Calcul des forces appliquées à chaque particule.
2. Calcul des positions vitesses et accélérations en intégrant les équations de la dynamique associées au système.

La méthode de parallélisation utilisée consiste à décomposer l'ensemble des particules en blocs de taille équivalente et de distribuer ces blocs sur les différents processeurs disponibles. Les tâches permettant de calculer les forces, positions, vitesses et accélérations sont ensuite créées. La figure 7.2 montre une partie du graphe de flot de données construit à chaque itération. On considère dans cette illustration que la particule i a comme voisine la particule j et seules les tâches permettant de calculer le nouvel état de la particule i sont représentées. Même avec peu de particules la totalité du graphe de flot de données pour une itération est difficilement représentable.

Le placement des tâches de calcul est donc dirigé par la distribution initiale des blocs de particules. Cependant, le calcul des forces soumises aux particules situées aux frontières d'un bloc dépend de l'état des particules des blocs voisins. Les tâches "frontières" de deux blocs sont alternativement distribuées sur les deux processeurs associés afin d'équilibrer au mieux la charge.

Les expérimentations ont été effectuées pour deux distributions initiales. La première basée sur une allocation cyclique des blocs et la seconde sur une allocation basée sur l'ordonnancement DP-OCR présenté dans la section 5.2. Deux machines ont pour cela été utilisées :

- Une grappe appelée **e-cluster** constituée de 100 monoprocesseurs Pentium III 733Mhz disposant de 256 Mo de mémoire et interconnectés par un réseau Fast Ethernet 100Mo/s.

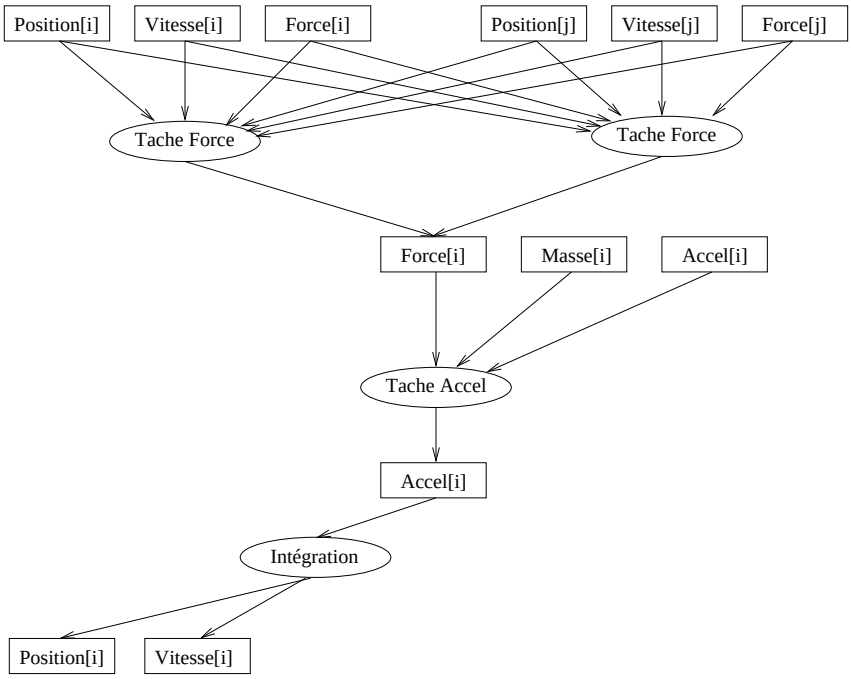


FIG. 7.2 – Graphe de flot de données associé au calcul du nouvel état de la particule i ayant la particule j comme voisine pour une itération de la simulation.

– Une grappe appelée **ARV** (Atelier de Réalité Virtuelle) composée de 10 bi-processeurs Pentium III 866Mhz disposant de 512 Mo de mémoire et aussi interconnectés par un réseau Fast Ethernet 100Mo/s.

La figure 7.3 montre les résultats obtenus sur ces deux machines. La partie a) permet de comparer les résultats obtenus sur la grappe ARV pour un préplacement cyclique des données en utilisant la règle OCR (Owner Compute Rule) pour les tâches avec l'ordonnancement DP-OCR. Lors de cette expérience, un mouchoir composé de 1000000 de particules est simulé. Cette courbe permet de remarquer que pour cette application, l'ordonnancement DP-OCR a un comportement moins bon que le préplacement cyclique. Ces résultats sont liés à la compléxité du graphe de flot de données de l'application qui demande un temps non négligeable pour être partitionné. Par ailleurs, le placement cyclique des données est un placement difficile à améliorer compte tenu des caractéristiques de l'application.

La partie b) de la figure montre les résultats obtenus sur la grappe e-cluster pour un mouchoir composé de 490000 particules. Cette courbe montre qu'à partir de 15 noeuds, l'ajout de processeur n'améliore pas sensiblement le temps de calcul. Cette limitation est liée à une gestion mémoire de l'application Sappe pouvant être améliorée. Ce travail est actuellement en cours de réalisation.

Ces résultats sont difficilement comparables avec les différents résultats publiés pour le même type d'application. En effet, à la différence des autres travaux, les

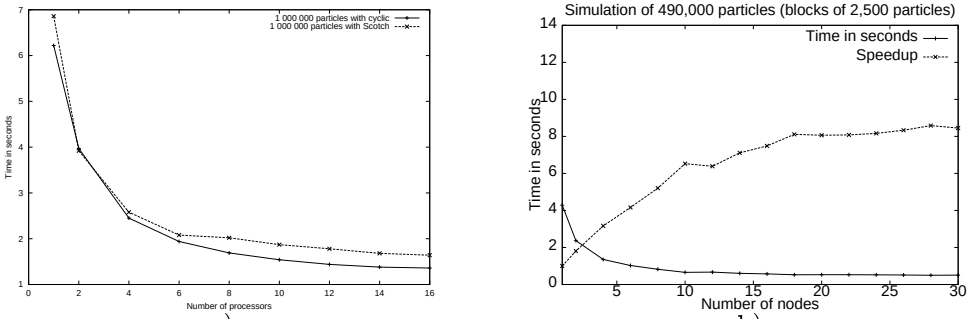


FIG. 7.3 – Résultats obtenus pour l'application Sappe sur la grappe ARV a) et sur le e-cluster b)

simulations de l'application Sappe sont effectuées sans détection des collisions entre particules. On peut cependant noter que dans [59] les résultats présentés portaient sur un tissus composé de 3520 particules sur 8 processeurs.

7.1.2 Dynamique moléculaire : Tuktut

L'application Tuktut est une application permettant de simuler le comportement de certaines molécules dans des conditions particulières, difficilement réalisable en laboratoire. Cette application est basée sur les travaux de Pierre Eric Bernard au cours de son doctorat [7, 6].

Les caractéristiques de cette application sont très proches de l'application Sappe. Comme pour Sappe, chaque itération peut être décomposée en une phase de calcul des forces suivie par une phase de calcul des positions, vitesses et accélérations des particules. Les forces auxquelles sont soumises les particules sont pour certaines de natures très différentes. Ces forces sont de deux types :

- Les forces intra-moléculaires géométriques qui sont représentées comme dans Sappe par des ressorts.
- Les forces "non liées" engendrées par le champ électrostatique induit par la charge des particules.

Contrairement à Sappe, la parallélisation effectuée n'est pas basée sur une décomposition géométrique de l'ensemble des particules, mais sur une décomposition spaciale dépendante de la position dans l'espace des atomes de la molécule simulée. L'espace de simulation est donc divisé en blocs qui sont ensuite affectés à un processeur. Les calculs liés aux particules d'une boîte sont donc effectués sur le processeur associé.

Dans le cadre des applications de dynamique moléculaire, l'ordonnancement généralement effectué est basé sur une bipartition récursive de l'espace de simulation. Dans les expérimentations décrites ci-dessous, cet algorithme d'ordonnancement est comparé à trois autres algorithmes d'ordonnancement :

- Préplacement cyclique des blocs et utilisation de la règle OCR pour les tâches.

- Ordonnancement DP-OCR.
- LPTF (Largest Process Time First). Cet algorithme suppose que toutes les tâches sont indépendantes et qu'une annotation de coût leur est associée. Les tâches sont alors allouées de manière cyclique en commencant par les tâches dont le coût est le plus grand. Les communications de données nécessaires sont donc générées en fonction du placement des tâches.

Les expériences sont effectuées sur deux molécules. La première, appelée GPIK, est composée de 11615 atomes. La seconde appelée BGLA (hydrated β-galactosidase) est composée de 413039 atomes. Le temps séquentiel d'exécution est de 4.6 secondes pour GPIK et de 324.0 secondes pour BGLA sur un processeur de la grappe ARV décrite précédemment.

#nodes	CYCLIC	LPTF	ORB	DP-OCR
2	1,6	1,6	1,2	1,7
4	1,3	1,4	1,2	1,3
8	1,0	1,0	1,1	0,9

#noeuds	CYCLIC	LPTF	ORB	DP-OCR
2	87,2	87,3	104,7	111,0
4	48,4	46,8	51,0	52,9
8	27,6	25,5	40,0	36,3

TAB. 7.1 – GPIK (gauche) et BGLA (droite) temps d'exécution (sec) sur la grappe ARV

#noeuds	CYCLIC	LPTF	ORB	DP-OCR
2	2.87	2.87	3.83	2.7
4	3.53	3.28	3.83	3.53
8	4.6	4.6	4.18	5.11

#nodes	CYCLIC	LPTF	ORB	DP-OCR
2	3.71	3.71	3.09	2.91
4	6.69	6.92	6.35	6.12
8	11.7	12.7	8.1	8.92

TAB. 7.2 – GPIK (gauche) et BGLA (droite) accélération sur la grappe ARV

7.2 Optimisation combinatoire (ACI Doc-G)

Cette section est consacrée à la description des travaux menés dans le cadre de l'ACI Doc-G (Défi en Optimisation Combinatoire sur Grille). Ces travaux ont pour objectif d'exploiter des architectures de type grille de calcul pour la résolution de problèmes d'optimisation combinatoire de type *Branch-and-X* (*Branch-and-Bound*, *Branch-and-Cut* ou *Branch-and-Price*). Nous proposons donc, dans le cadre de ces travaux, une implantation de l'application QAP (Quadratic Assignment Problem) permettant d'exploiter des architectures de type grille de calcul. Pour cela, nous avons utilisé la bibliothèque Bob++ développée au laboratoire PRISM à Versailles dont l'objectif est de faciliter le développement de telles applications.

De nombreux travaux ont montré que les parcours arborescents de type *Branch-and-X* permettent d'obtenir de très bons résultats sur des architectures de grande taille. Dans [2] la résolution du problème du Qap (*quadratic assignment problem*) est reportée pour la taille 30 du problème *Nugent* (30 usines pour 30 sites). Cette résolution considérée comme très difficile a été menée sur une grille composée de 2510 machines et a nécessité un temps de calcul d'environ 7 jours et le développement d'une arborescence de plus de 12 milliards de sommets.

Nous présenterons dans cette section l'implantation parallèle effectuée pour cette application ainsi que les expériences effectuées sur une grille de calcul.

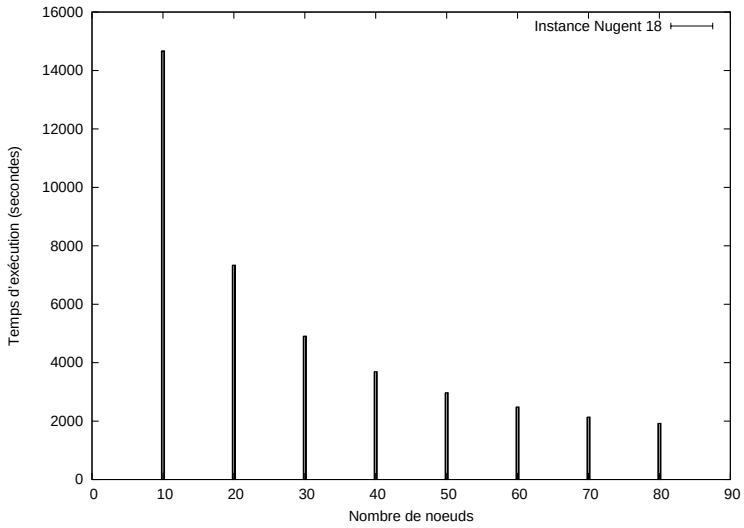


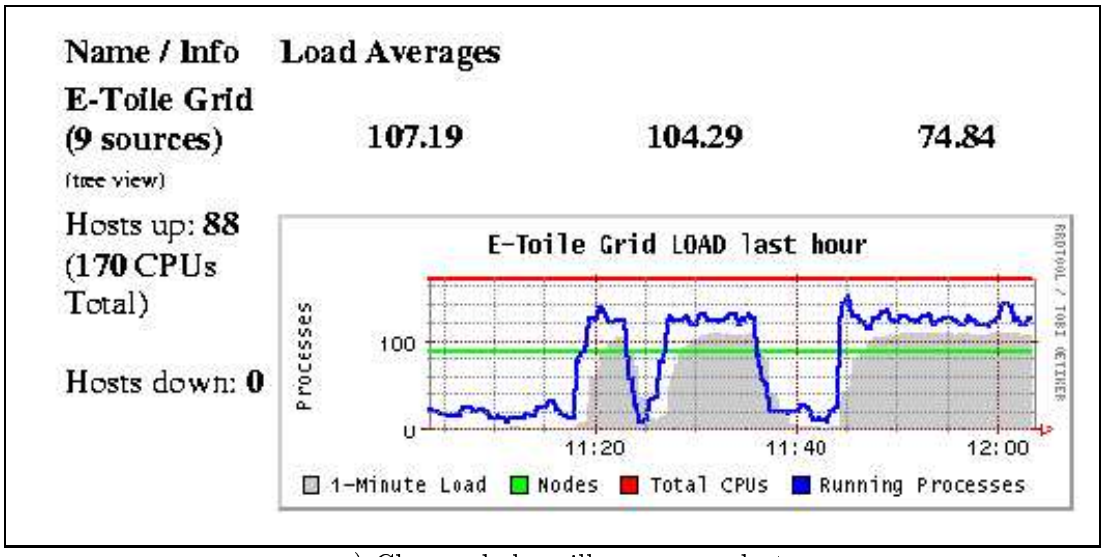
FIG. 7.4 – Résultats obtenus pour l'instance Nugent 18 du problème sur une grappe de PC

7.2.1 Présentation du problème du Qap et des développements effectués

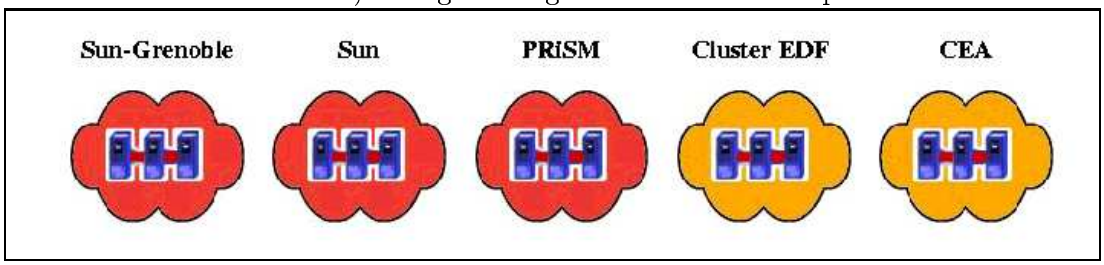
Le problème du QAP consiste à affecter n usines à n emplacements disponibles de manière à réduire le coût d'exploitation de l'ensemble des usines. A chaque paire d'usine est associé un volume d'échange de matériaux et à chaque paire d'emplacement est associée une distance. L'évaluation d'une solution consiste donc à multiplier pour chaque paire d'usine, le volume de matériaux échangés par la distance qui les sépare.

Un code de cette application écrit en Bob++ était disponible au début du projet. Ce code utilise un algorithme de type Branch and Bound proposé par Bob++. Les principaux travaux de développement ont consisté à paralléliser la bilbiothèque Bob++ en utilisant la bibliothèque Athapascan. Le code de l'application QAP est donc identique au code initial et ne contient aucun appel aux primitives Athapascan.

La parallélisation proposée consiste à décomposer l'arborescence parcourue lors de la recherche dans l'espace de solutions. Cette décomposition consiste à affecter la recherche d'une branche (sous arborescence) à un processeur disponible. Cette technique impose la recopie du contexte de calcul sur tous les noeuds participant à la recherche et implique donc une consommation mémoire plus importante qu'en séquentiel mais bornée par $O(pS_1)$ où p correspond au nombre de processeur et S_1 à la quantité de mémoire nécessaire pour une exécution séquentielle. Cependant cette technique a l'avantage de permettre d'adapter la granularité du calcul en ajustant la taille des sous-arbres affectés aux processeurs.



a) Charge de la grille au cours du temps



b) Charge de la grille à un instant donné de l'exécution

FIG. 7.5 – Observation de l'exécution du programme sur la grille avec Ganglia.

7.2.2 Expériences sur grille de calcul

La première expérience effectuée consiste à lancer le programme du QAP sur une instance du problème de taille moyenne afin de valider son comportement sur une architecture de type grappe.

La figure 7.4 montre les résultats obtenus pour l'instance Nugent 18 du problème (18 usines, 18 emplacements) sur une grappe de PC. Cette grappe est composée de 100 monoprocesseurs Pentium III 733 disposant de 256 Mo de mémoire et interconnectés par un réseau Fast Ethernet 100Mo/s. Le temps séquentiel pour cette instance est de 123650 secondes (non représenté sur la figure pour des raisons de lisibilité). On obtient donc une accélération de 64 lorsque 80 processeurs sont utilisés.

Les résultats obtenus pour une instance de taille moyenne sur un grappe de PC homogènes nous ont conduits à mener un expérience de plus grande envergure pour une instance de taille supérieure (Nugent 20, 20 usines, 20 emplacements) sur une architecture de type grille. Pour cette expérience, nous disposons de quatre grappes géographiquement distantes :

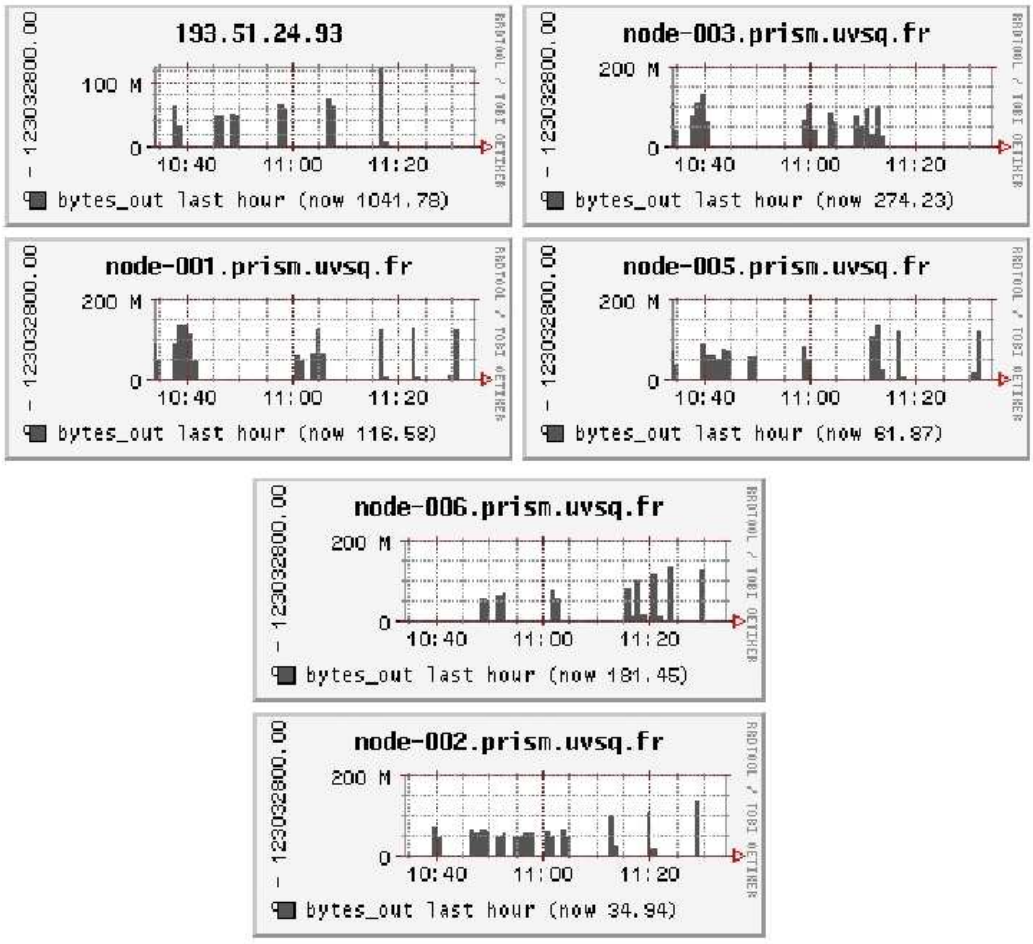


FIG. 7.6 – Observation des communications sur les noeuds de la grappe du PRiSM.

- CEA Saclay 10 bi-AMD MP 1800+
- SUN ENS-Lyon 15 bi-Pentium III 1.4Ghz
- PRiSM Versailles 7 bi-Xeon 2.4Ghz
- EDF Clamart 10 bi-Xeon 2.2 Ghz

La grille dont nous disposons est donc composée de 84 processeurs fortement hétérogènes. Nous avons dans un premier temps lancé le programme sur les 20 processeurs de la grappe du CEA. Nous avons alors obtenu le résultat en $T_{20} = 1648$ secondes. La même instance a ensuite été résolue sur les 84 processeurs de la grille nous permettant d'obtenir le résultat en $T_{84} = 499$ secondes. Nous obtenons donc une accélération de 3.1 lorsque les 84 processeurs sont utilisés par rapport à l'exécution sur la grappe CEA Saclay.

L'évaluation de ces résultats est confrontée à deux problèmes. Le premier est lié à la taille du problème. Il est en effet difficile d'exécuter des instances du problème

de taille importante sur un seul processeur pour obtenir une exécution de référence. Pour certaines tailles du problème, les temps d'exécution séquentielle peuvent atteindre plusieurs jours voir même plusieurs mois. Le deuxième problème est lié à l'hétérogénéité des processeurs. En effet, même dans le cas où un temps d'exécution séquentiel de référence est disponible, une accélération sur plusieurs processeurs hétérogènes reste difficile à analyser.

Une solution grossière à ces problèmes consiste à se baser sur les performances d'un processeur virtuel et d'utiliser le critère des fréquences pour ramener la puissance des autres processeurs à celle du processeur virtuel. Ainsi, la somme des fréquences de l'ensemble des processeurs est de 155.6Ghz ce qui correspond à 86.44 processeurs à 1.8Ghz (fréquence des processeurs de la grappe CEA Saclay). De cette manière, nous pouvons évaluer la performance globale de la grille dont nous disposons à 4.32 fois la performance de la grappe CEA Saclay pour une accélération observée de 3.1. Cette perte d'efficacité est principalement dûe à l'utilisation d'un réseau métropolitain qui ajoute un coût non négligeable aux communications. Cette évaluation reste une évaluation très grossière, elle ne prend pas en compte les différences d'architecture des processeurs (caches,...) ni les caractéristiques des réseaux intra et inter grappes.

L'algorithme d'ordonnancement utilisé tire aléatoirement un processeur victime pour lui voler du travail. Ce tirage est effectué de manière uniforme sur l'ensemble des processeurs. Une optimisation consisterait à favoriser le tirage des processeurs proches de manière à limiter le surcoût d'utilisation du réseau métropolitain.

L'exécution sur grille peut être observée grâce à l'outil GANGLIA [60]. La figure 7.5 montre deux types d'observations fournis par cet outil. La partie a) de la figure montre l'évolution de la charge de la grille au cours du temps. Les paliers sur la courbe permettent de voir les périodes de temps au cours desquelles une instance du QAP était en cours de résolution. La partie b) montre la charge de la grille à un instant donné de l'exécution. Lorsqu'une grappe est en rouge (comme les trois grappes situées à gauche), sa charge est à 100% et lorsque la grappe est en orange (comme les deux grappes situées à droite), sa charge se situe entre 75 est 100%. A cet instant de l'exécution, la grille était donc proche du maximum possible de son exploitation.

La figure 7.6 montre l'évolution des communications sortantes pour chaque nœud de la grappe PRiSM. Les pics permettent de visualiser les requêtes de vols traitées par le noeud visualisé. On peut donc remarquer que les communications ne représentent qu'une petite partie du temps d'exécution pour cette application.

Les résultats de l'application QAP sont donc très encourageants mais de nombreuses expériences supplémentaires sont à effectuer. En particulier, des problèmes de taille supérieure peuvent être résolus avec la grille à notre disposition.

7.3 Conclusions

Dans ce chapitre, nous avons pu montrer que l'implantation d'Athapascan basée sur les mécanismes de dégénération séquentielle et distribuée présentés dans les

chapitres précédents se comportait correctement pour de réelles applications. Nous avons pour cela présenté les résultats obtenus pour deux applications de simulation Sappe et Tuktut. Bien que les résultats pour ces deux applications soient acceptables, de nouvelles expérimentations permettant d'exploiter efficacement un plus grand nombre de machines sont en cours de réalisation. Dans une deuxième partie, nous avons présenté les résultats obtenus pour la résolution du QAP (Quadratic Assignment Problem). De bons résultats ont tout d'abord été obtenus sur grappe de PC avant d'expérimenter cette application sur une grille de 84 processeurs. Les résultats sont encourageants et permettent d'envisager la résolution d'instances de plus grande taille et l'exploitation d'un plus grand nombre de processeurs.

CHAPITRE 8

Conclusions et perspectives.

Lors d'une exécution parallèle, le placement des différentes tâches de calcul sur les processeurs disponibles est d'une importance cruciale. Cependant, une mauvaise réalisation de ce placement peut conduire à une dégradation importante des performances, y compris dans le cas où un placement optimal des tâches peut être calculé. Dans la pratique, le coût de création d'une tâche parallèle est toujours très supérieur au coût d'un appel séquentiel de fonction. Ce surcoût est principalement lié à la gestion des communications et des synchronisations inhérents à toute exécution parallèle.

Dans cette thèse, nous avons proposé des mécanismes de dégénération séquentielle et distribuée permettant d'optimiser les surcoûts de gestion du parallélisme. La dégénération séquentielle (ou *work first principle*) consiste à optimiser l'exécution séquentielle d'une tâche quitte à payer un surcôut plus important lorsque la génération de plus de parallélisme est nécessaire. Ce principe utilise le fait que lors d'une exécution parallèle, de nombreuses tâches sont exécutées de manière séquentielle sans nécessiter la communication de données distantes. Dans ce cas, la création d'une tâche aurait pu être remplacée par un simple appel séquentiel de fonction. La dégénération distribuée consiste à affecter chaque tâche de calcul à un processeur et à générer de manière automatique les communications nécessaires à l'exécution distribuée du programme selon cet ordonnancement.

Les mécanismes que nous avons proposés s'appliquent aux environnements de programmation parallèle de haut niveau basés sur la création de tâches qui communiquent via une couche de mémoire partagée. Nous avons donc étudié et comparé théoriquement les performances en terme de défauts de page de plusieurs protocoles de cohérence mémoire communément utilisés. Les différentes implantations de ces protocoles ont cependant montré qu'il était difficile d'obtenir des exécutions parallèles performantes au-delà d'une dizaine de processeurs. Nous avons pour cela proposé un protocole de cohérence plus restrictif basé sur le typage explicite des accès aux données partagées effectués par les tâches ainsi que sur l'analyse dynamique du flot de données de l'application. L'analyse de ce protocole montre que peu de défauts de page interviennent lors d'une exécution parallèle si peu de tâches sont exportées (c'est-à-dire exécutée sur un processeur différent du processeur où elle a été créée). Cette analyse est théorique, il serait intéressant d'observer l'impact du protocole de cohérence en terme de défauts de page sur de réelles exécutions parallèles.

Un mécanisme de pile distribué basé sur la construction distribuée du flot de données de l'application a ensuite été proposé. Nous avons vu que les avantages de ce mécanisme sont multiples :

- Implantation du protocole de cohérence "flot de données".
- Exécution des tâches en série optimisée par la gestion de la pile.
- Echanges des données par communications unidirectionnelles lors d'une exécution distribuée.

Deux algorithmes d'ordonnancement tirant partie des spécificités de ce mécanisme ont été proposés dans le chapitre 5. Le premier est un algorithme dynamique basé sur le vol de tâches déclenché lorsqu'un processeur devient inactif. Nous avons montré que cet algorithme permettait de majorer le nombre de tâches volées lors de l'exécution du programme. L'utilisation du protocole de cohérence "flot de données" avec cet algorithme permet donc d'affiner la borne sur le nombre de défauts de page présentée dans le chapitre 3. Le deuxième est un algorithme d'ordonnancement statique basé sur le regroupement des données partagées dans les différents modules de mémoires disponibles. Nous avons vu que cet algorithme permettait d'obtenir un gain significatif en terme de volume de données échangés particulièrement pour le graphe *"simulation"* étudié.

Le mécanisme de pile distribuée ainsi que les algorithmes d'ordonnancement proposés ont été implantés pour le langage de programmation parallèle Athapascan. Cette nouvelle implantation du langage ainsi que sa validation sur des tests simples ont donc été présentés dans le chapitre 6.

Les résultats obtenus sur l'application d'optimisation combinatoire *QAP* montre que l'implantation distribuée de la pile présentée dans la chapitre 4 permet d'obtenir des exécutions parallèles performantes sur des architectures de type grille de calcul. Nous avons observé un bon comportement de cette application jusqu'à 80 processeurs et des programmes tests comme *Knary* ont pu tirer partie d'une centaine de processeurs. Certains projets comme *Grid 5000* ont pour objectif l'exploitation de plusieurs milliers de processeurs ; l'exploitation de telles architectures en utilisant un langage de programmation parallèle de haut niveau comme Athapascan peut donc être considéré comme l'une des perspectives les plus importantes de cette thèse.

Certaines application de simulation permettent elles aussi d'exploiter un grand nombre de processeurs [52]. Cependant, les premiers résultats obtenus avec Athapascan ne permettent pas d'envisager d'expériences sur de telles architectures dans l'immédiat.

Pour ces applications, atteindre la portabilité et le passage à l'echelle nécessite la recherche de schémas algorithmiques adaptés permettant un ordonnancement automatique et portable. Ce travail est un des objectifs du projet MOAIS.

Bibliographie

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks : Shared memory computing on networks of workstations. *IEEE Computer*, 29(2) :18–28, 1996.

[2] K.M. Anstreicher, Brixius N.W., J.P. Goux, and Linderoth J. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3) :563–588, 2002.

[3] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures : Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4) :598–632, 1989.

[4] L. Baduel, F. Baude, D. Caromel, C. Delbe, N. Gama, S. E. Kasmi, and S. Lanteri. A parallel object-oriented application for 3d electromagnetism. In IEEE ACM, editor, *IPDPS'04*, avril 2004.

[5] S. Ben Hassen and H.E. Bal. Integrating task and data parallelism using shared objects. In *International Conference on Supercomputing*, pages 317–324, 1996.

[6] P.-E. Bernard, T. Gautier, and D. Trystram. Large scale simulation of parallel molecular dynamics. In *Proceedings of Second Merged Symposium IPPS/SPDP*, San Juan, Puerto Rico, April 1999.

[7] P.-E. Bernard, B. Plateau, and D. Trystram. Using threads for developing parallel applications : Molecular dynamics as a case study. In Trobec, editor, *Parallel Numerics*, pages 3–16, Gozd Martuljek, Slovenia, September 1996.

[8] G.E. Blelloch, J.C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1) :4–14, 1994.

[9] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.

[10] R.D. Blumofe, M. Frigo, C.F. Joerg, C.E. Leiserson, and K.H. Randall. Dag-consistent distributed shared memory. In *IPPS'96*, pages 132–141, 1996.

[11] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk : An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1) :55–69, 1996.

[12] R.D. Blumofe and C.E. Leiserson. Space-efficient scheduling of multithreaded computations. In *STOC'93 ACM Symposium on the Theory of Computing*, pages 362–371, 1993.

[13] R.D. Blumofe and P.A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX'97*, pages 133–147, 1997.

[14] G. Bosilca, G. Fedak, and F. Capello. Ovm : Out-of-order execution parallel virtual machine. In IEEE/ACM, editor, *Proceedings of the first International Symposium on Cluster Computing and the grid*, 2001.

[15] S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma, and Boyer F. Experiences implementing efficient java thread, serialization, mobility and persistence. Technical Report RR-4662, INRIA Rhône-Alpes, projet APACHE, Dec 2002.

[16] R.K. Brunner, J.C. Phillips, and Kale L.V. Scalable Molecular Dynamics for Large Biomolecular Systems. In *Proceedings of Supercomputing (SC) 2000, Dallas, TX*, November 2000.

[17] A. Carissimi. *Le noyau exécutif Athapascan-0 et l'exploitation de la multiprogrammation légère sur les grappes de stations multiprocesseurs.* Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, November 1999.

[18] D. Caromel, L. Henrio, and Serpette. B. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pages 123–134. ACM Press, 2004.

[19] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in java. In Geoffrey C. Fox, editor, *Concurrency Practice and Experience*, pages 1043–1061, sept-nov 1998.

[20] G. Cavalheiro. *Athapascan 1 : Interface genérique pour l'ordonnancement dans un environnement d'exécution parallèle.* Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, November 1999.

[21] M. Doreille. *Athapascan 1 : vers un modèle de programmation parallèle adapté au calcul scientifique.* Thèse de doctorat en mathématiques appliquées, Institut National Polytechnique de Grenoble, France, December 1999.

[22] J.T. Feo, D.C. Cann, and R.R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10(4) :349–366, 1990.

[23] Message Passing Interface Forum. MPI : A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[24] The OpenMP Forum. Openmp c and c++ application program interface, version 1.0, Octobre 1998. http ://www.openmp.org.

[25] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[26] F. Galilée. *Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle.* Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, September 1999.

[27] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1 : On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, pages 88–95, Paris, France, October 1998.

[28] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, (A) User's Guide nd Tutorial for Networked Parallel Computing*. MIT Ress, Cambridge, Mass., 1994.

[29] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads : Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1) :5–20, 1996.

[30] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Tech J.*, 45 :1563–1581, 1966.

[31] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6) :789–828, 1996.

[32] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, Dept. Computer Science, Houston, Tex., 1993.

[33] Y.C. Hu, A.L. Lu, H. Cox, and Zwaenepoel W. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12) :1512–1530, 2000.

[34] J.J. Hwang, Y-C Chow, F.D. Anger, and C-Y Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2) :244–257, 1989.

[35] IEEE. Ieee 1003.1c-1994 : Standard for information technology - portable operating system interfaces (posix) - part 1 : System application programm interface (api) - amendment 2 : Thread extension [c language]. *IEEE Computer Society Press*, MD 20910, 1994.

[36] L. V. Kale and S. Krishnan. *Parallel Programming using C++*, chapter Charm++ : Parallel Programming with Message-Driven Objects, pages 175–213. MIT Press, 1996.

[37] L.V. Kale and S. Krishnan. Charm++ : a portable concurrent object-oriented system based on c++. In *OOPSLA'93*, pages 91–108. ACM Press, Septembre 1993.

[38] L.V. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. Namd2 : Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, (151) :283–312, 1999.

[39] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning : Applications in VLSI domain. Technical report, 1997.

[40] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks : Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[41] P. T. Koch, J. S. Hansen, E. Cecchet, and X. Ronsset de Pina. SciOS : An SCI-based software distributed shared memory. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, 1999.

[42] J.-C. Konig and J.-L. Roch. Machines virtuelles et techniques d'ordonnancement. In *Proceedings école d'hiver de PRS ICARE'97*, Auusois, France, dec 1997.

[43] J.-C. Konig and J.-L. Roch. Machines virtuelles et techniques d'ordonnancement. In D. Barth, J. Chassin de Kergommeaux, J.-L. Roch, and J. Roman, editors, *ICaRE'97 : conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*. CNRS, December 1997.

[44] C-Y Lee, J.J. Hwang, Y-C Chow, and F.D. Anger. Multiprocessor scheduling with interprocessor communication delay. *Operations Research Letters*, 7(3) :141–147, 1988.

[45] N. Maillard. *Calcul Haute-Performance et Mécanique Quantique : analyse des ordonnancements en temps et en mémoire*. Thèse de doctorat en mathématiques appliquées, parallélisme, Institut National Polytechnique de Grenoble, France, November 2001.

[46] C. Martin. *Déploiement et contrôle d'applications parallèles sur grappes de grandes tailles*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, dec 2003.

[47] Girija J. Narlikar. *Space-Efficient Scheduling for Parallel, Multithreaded Computations*. PhD thesis, Carnegie Mellon University, Mai 1999. Available as CMU-CS-99-119.

[48] G.J. Narlikar. Scheduling threads for low space requirement and good locality. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 83–95, 1999.

[49] J. Nieh and Lam M. The Design, Implementation and Evaluation of SMART : A scheduler for Multimedia Applications. In *ACM Symposium on Operating Systems Principles*, number 3, pages 184–197, 1997.

[50] F. Pellegrini and J. Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical Report 1038-96, 1996.

[51] L. Peng, W.F. Wong, M.D. Feng, and C.K. Yuen. SilkRoad : A multithreaded runtime system with software distributed shared memory for SMP clusters. In IEEE, editor, *Cluster'2000*, pages 243–249, November 2000.

[52] J.C. Phillips, G. Zheng, S. Kumar, and L.V. Kalé. Namd : Biomelecular simulation on thousand of processors. In *IEEE/ACM SC2002*.

[53] K. Randall. Cilk : Efficient multithreaded computing. Technical Report MIT/LCS/TR-749, 1998.

[54] R. Revire, F. Zara, and T. Gautier. Efficient and easy parallel implementation of large numerical simulation. In Springer, editor, *Proceedings of ParSim03 of EuroPVM/MPI03*, pages 663–666, Venice, Italy, 2003.

[55] M.C. Rinard. *The design, implementation and evaluation of Jade : a ortable, implicitly parallel programming language*. PhD thesis, Stanford University, Juin 1994.

[56] M.C. Rinard and M.S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3) :483–545, 1998.

[57] J.-L. Roch. Ordonnancement de programmes parallèles sur grappes : théorie versus pratique. In *Actes du congrès international ALA 2001, Université Mohammed V*, pages 133–144, Rabat, Maroc, mai 2001.

[58] J.-L. Roch and *et al.* Athapascan : Api for asynchronous parallel programming. Technical Report RR-0276, INRIA Rhône-Alpes, projet APACHE, February 2003.

[59] S. Romero, L.F. Romero, and E.L. Zapata. Fast cloth simulation with parallel computers. In *Euro-Par 2000*, pages 491–499, Munich, August 2000.

[60] F.D. Sacerdoti, Katz M.J., Massie M.L., and Culler D.E. Wide area cluster monitoring with ganglia. In *IEEE Cluster*, Hong Kong, 2003. http ://ganglia.sourceforge.net.

[61] S. Sardesai, D. McLaughlin, and P. Dasgupta. Distributed cactus stacks : Runtime stack-sharing support for distributed parallel programs. In *PDPTA'98*, volume I, pages 57–65, 1998.

[62] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[63] G. Tel. *Introduction to distributed algorithms*. Cambridge university press edition, 1994. chapitre 8.

[64] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Satin : Efficient parallel divide-and-conquer in java. In *Euro-Par 2000 Parallel Processing*, Lecture Notes in Computer Science, pages 690–699, Munich, Germany, aout 2000. Springer.

[65] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proc. Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 34–43, June 2001.

[66] T. Yang and A. Gerasoulis. PYRROS : Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, 1992.

[67] T. Yang and A. Gerasoulis. Dsc : Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9) :951–967, 1994.

[68] F. Zara. *Algorithmes parallèles de simulation physique pour la synthèse d'images : application à l'animation de textiles*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, Septembre 2003.

[69] F. Zara, F. Faure, and J-M. Vincent. Physical cloth simulation on a pc cluster. In X. Pueyo D. Bartz and E. Reinhard, editors, *Fourth Eurographics Workshop on Parallel Graphics and Visualization 2002*, Blaubeuren, Germany, September 2002.

[70] F. Zara, F. Faure, and J-M. Vincent. Parallel Simulation of Large Dynamic System on a PCs Cluster : Application to Cloth Simulation. *International Journal of Computers and Applications*, march 2004.