Ordonnancement efficace d'applications parallèles : les tâches malléables monotones

Grégory Mounié

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE

présentée et soutenue publiquement par

Grégory MOUNIÉ

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Spécialité : **Informatique : Systèmes et Communications**

Thèse soutenue au laboratoire Informatique et Distribution
et préparée au sein du Laboratoire de Modélisation et Calcul
(Institut d'Informatique et de Mathématiques Appliquées de Grenoble)
dans le cadre de
l'École Doctorale Mathématiques, Sciences et Technologie de l'Information.

Ordonnancement efficace d'applications parallèles : les tâches malléables monotones

---

Date de soutenance : 26 juin 2000

Composition du Jury

Président :	Yves	ROBERT
Rapporteurs :	Claire	KÉNYON
	Sanjay	RAJOPADHYE
Examinateurs :	Eric	BLAYO
	Brigitte	PLATEAU
	Denis	TRYSTRAM (directeur de thèse)

Remerciements

La thèse est un long chemin que l'on ne parcourt pas seul.

Je tiens à remercier mes rapporteurs et les membres de mon jury, et tout d'abord, Claire Kenyon, Professeur à l'université de Paris XI pour sa lecture attentive et ses corrections des preuves incluses dans cette thèse ainsi que pour la pertinence de ces remarques qui ont amélioré la qualité du document. Je remercie aussi Sanjay Rajopadhye, directeur de recherche CNRS à l'IRISA, pour ses améliorations sur la structure du document et ses remarques sur le domaine de la parallélisation automatique.

Je remercie Yves Robert de m'avoir fait l'honneur de présider mon jury et Brigitte Plateau qui m'a accueilli au sein de l'équipe APACHE puis au sein de son laboratoire.

Je remercie Éric Blayo, avec qui travailler a été très enrichissant scientifiquement et qui a accepté d'évaluer mon travail.

Enfin je remercie mon directeur Denis T. pour avoir encadré mon travail pendant toute cette thèse, et m'avoir fait découvrir la recherche en Informatique.

J'ai eu le plaisir de travailler, ou de ne pas travailler, avec λ Christophe$/2$, Alfredo, papaisino, et l'autre spécialiste français des protées, Renaud.

Il faut que je remercie mes collègues de bureau qui m'ont supporté pendant ces années, notamment Paulo, mon premier professeur de portugais, Roberta ma première victime de ce portugais, et Olivier, artiste quasi breton méconnu.

Les membres du laboratoire forment une équipe à l'ambiance très agréable : Jean-Marc, Jacques et Jacques, Jean-Louis, Joëlle, Philippe et Philippe, Yves. Les nombreux Gauchos que j'ai côtoyés, Alexandre, Benhur, Andrea, Denise, Gersonc et beaucoup d'autres, m'ont ouvert de nouveaux horizons.

Je remercie les fournisseurs officiels de gâteaux, pâtissiers hors pairs, que sont Cécile, Florence et Manu.

Je tiens à remercier celle qui a lu ma thèse deux fois en entier, Fredérique, et qui m'a soutenue tout au long de ce long chemin.

Finalement, je remercie les membres de ma famille grâce à qui je suis arrivé ici.

Juste un dernier mot, pour remercier le premier conseiller du ministre de la recherche et les aiguilleurs du ciel pour leur participation involontaire, mais active, à la soutenance de cette thèse.

Table des matières

Avant-Propos

Après des débuts très prometteurs (il y a plus de 20 ans), le domaine du parallélisme a connu quelques difficultés, autour des années 1990, principalement à cause de problèmes commerciaux des constructeurs de machines. Aujourd'hui, la technologie est maîtrisée, et l'apparition de systèmes parallèles et distribués "domestiques" a relancé le domaine. Par exemple, le CPlant du Sandia National Laboratories est classé 44-ième machine du TOP 500 des superordinateurs en novembre 1999. Il est juste composé de 592 stations Alpha sous Linux, relié par un réseau Myrinet. Le besoin en puissance de calcul reste important dans des domaines tels que la météorologie, la biologie ou la simulation numérique. De nouveaux domaines devraient émerger, tels que les bases de données ou le décodage du génome.

Malheureusement, la parallélisation d'une application reste un problème ardu. Plusieurs voies sont envisageables, mais une approche universelle efficace n'émerge pas encore. Un des problèmes réside dans la multiplicité des niveaux dans lesquels le problème peut être considéré : architecture, système, environnement, langage…

Nous avons étudié, en collaboration avec une équipe de numériciens, la parallélisation d'une application, en océanographie, en nous focalisant sur la couche intermédiaire (gestion et optimisation des ressources de la machine). Notre contribution a consisté à proposer une nouvelle façon de gérer les ressources, d'en étudier les fondements et de la valider sur cet exemple.

Nous pensons qu'il est encore illusoire à l'heure actuelle de vouloir paralléliser une application sans l'aide de l'utilisateur qui, lui, connaît son application. En particulier, il peut aisément identifier les points de calculs critiques. Cette position est confirmée par l'état des recherches dans le domaine de la parallélisation automatique, prometteur mais encore restreint à des cas idéalisés. La difficulté principale est dans la recherche d'algorithmes d'ordonnancement efficaces qui réalisent un bon compromis entre l'utilisation des ressources et des communications pas trop coûteuses. Il est connu, même sous le modèle le plus simple sans communica-

tion, qu'ordonnancer un graphe quelconque pour qu'il s'exécute en un minimum de temps, est NP-difficile. Il n'existe pas, de par le théorème de l'impossibilité [22], d'algorithmes compétitifs à moins de $5/4$ pour les problèmes à *gros grain* où le coût d'une communication est plus petit qu'un calcul [70], si P est différent de NP. Pour le cas des grands temps de communication, les algorithmes ont des garanties encore plus mauvaises (pas de compétitivité constante connue à ce jour).

Notre contribution principale a consisté à proposer un modèle, le modèle des tâches malléables, pour tenir compte des communications de l'application de manière implicite, dans le problème d'ordonnancement. Cette prise en compte s'appuie sur un regroupement des activités parallèles défini par le programmeur, lui-même. Notre apport se situe tant au niveau théorique qu'au niveau pratique. Nous proposons plusieurs algorithmes d'ordonnancement pour ce modèle et nous analysons leurs performances. Pour évaluer l'utilité de ce modèle en pratique, nous avons également implanté une application de simulation océanique adaptative. Ce travail pratique a requis beaucoup de temps (plus de 6 mois). Il a consisté à écrire une maquette à partir d'un code existant écrit en Fortran 77/90 et PVM. Ce document ne reflète que partiellement ce travail.

Chapitre 1

Introduction

Contents

La répartition des calculs et des données est l'un des problèmes majeurs à résoudre pour réaliser d'une application parallèle efficace. Il faut décider de la date et du lieu d'exécution des calculs du programme parallèle sur l'ensemble des ressources, processeurs et mémoire, de la machine. L'efficacité de l'exécution va dépendre de ces décisions. Nous nous attachons à résoudre ce problème, nommé dans ce mémoire problème d'ordonnancement. *Nous allons montrer l'utilisation pratique d'un "modèle" proposé récemment pour l'ordonnancement d'applications parallèles : les Tâches Malléables [110].*
Nous introduisons dans ce chapitre les différentes architectures de machines parallèles. Nous présentons ensuite les principales méthodes existantes de programmation parallèle sur ces machines. Finalement, nous montrons quelques modélisations classiques de l'exécution d'une application parallèle, en mettant l'accent sur ce qui constitue, de notre point de vue, leurs principaux défauts.

1.1 La programmation parallèle

Bien que la puissance et la capacité mémoire des ordinateurs ne cessent de croître, certains utilisateurs désirent toujours obtenir leurs résultats plus rapidement. D'autres souhaitent aussi faire tourner des simulations de plus grande précision en conservant un temps de calcul raisonnable.

Le coût de fabrication d'un processeur toujours plus puissant croît exponentiellement avec la vitesse avec laquelle il peut faire les calculs. Par contre obtenir N processeurs ne représente qu'un coût N fois plus grand que celui d'un seul processeur. Le coût total de la machine parallèle (incluant le réseau d'interconnexion...) reste donc du même ordre.

Le parallélisme consiste à utiliser plusieurs ressources disponibles, processeurs, mémoires, disques, etc., pour qu'elles participent ensemble au calcul d'une application. En multipliant les ressources par N, un utilisateur peut espérer :

- calculer N fois plus vite,
- calculer des problèmes occupant N fois plus d'espace mémoire.

Dans ce mémoire nous traitons le problème de la minimisation du temps d'exécution d'une application parallèle.

Pour une introduction aux problèmes du parallélisme, le lecteur pourra se référer à [83] et [27], ou plus spécifiquement dans le cadre de l'algèbre linéaire à [32].

1.1.1 Les différents types de machines parallèles

Un grand nombre de constructeurs et différents types de machines parallèles sont apparus au fil du temps. Puis, au fur et à mesure de l'augmentation des coûts de production, ils ont disparu peu à peu. Le paysage des machines parallèles actuelles est maintenant relativement homogène, à l'exception de quelques réalisations académiques, ou des machines comme la TERRA [113], qui utilisent des techniques originales, que nous allons évoquer, pour maintenir l'activité des unités de calculs du processeur.

Si l'on considère le paysage des machines parallèles du point de vue des classifications standard qui distiguent SIMD (machine effectuant une même opération sur plusieurs données) et MIMD (machine effectuant des opérations différentes sur plusieurs données), on peut noter que les machines SIMD à processeurs spécialisés (Cray J90, Fujitsu VPP) sont cantonnées au marché du très haut de gamme sur des applications spécialisées de calcul vectoriel. Parallèlement, les processeurs standard contiennent de plus en plus d'instructions SIMD pour accélérer les calculs répétitifs que l'on trouve dans les algorithmes multimédias (compression/décompression d'images ou de sons, calcul d'images en trois dimensions, etc.). Les machines MIMD, qui incluent ces processeurs aux capacités de calculs sans cesse croissantes, deviennent, elles, de plus en plus présentes. Ces machines sont maintenant presque toujours mises en réseaux. Il est donc facile de trouver des réseaux de plus en plus rapides reliant des ordinateurs toujours plus puissants.

En fait, la plupart des grandes machines parallèles MIMD actuelles ne sont plus qu'un assemblage de matériel standard que l'on trouve dans les stations de travail (processeurs, disques, etc.) mais en les reliant avec un réseau très rapide. Les plus grandes machines parallèles contiennent jusqu'à 10000 processeurs (cf les différents projets ASCI aux États-Unis) et dépassent maintenant le teraflops (10^{12} opérations flottantes par seconde) [31].

Les stations de travail deviennent des machines multiprocesseurs à mémoire physiquement partagée (SMP, symmetric multi-processing). Tous les processeurs accèdent à toutes les zones de la mémoire commune avec la même vitesse. Ces machines avec un temps d'accès uniforme à la mémoire sont appelées *machines UMA* (en anglais, Uniform Memory Acces). Cette mémoire commune facilite grandement la programmation parallèle puisqu'il n'est plus nécessaire de mettre en oeuvre des communications explicites, il suffit juste de lire ou d'écrire des zones de mémoire partagées. Les SMP de plus grande taille actuellement sont les Entreprise 10000 de Sun à 64 processeurs.

Suivant l'évolution du matériel standard les machines parallèles sont devenues des réseaux de multiprocesseurs (Cray T3D/E, SGI Origin).

La dernière évolution concerne maintenant la création de machines parallèles avec une mémoire virtuellement partagée. Pour être performant, ce partage de

mémoire nécessite une gestion de la mémoire par le matériel gérant les communications. On obtient alors des machines parallèles avec plusieurs processeurs sur la carte mère. Ils partagent la même mémoire locale avec une interface réseau permettant les lectures/écritures de la mémoire à distance (SGI Origin 2000, réseau SCI de Dolphin pour des PC classiques). Contrairement à un SMP, l'accès à la mémoire commune dépend de la localisation physique du processeur et de la mémoire dans la machine parallèle. On parle de machines NUMA (Non Uniform Memory Access). Mais, pour ne pas compliquer le travail du programmeur, il faut aussi que les différents niveaux de mémoire cache des processeurs soient synchronisés avec les adresses mémoire qu'ils représentent. On parle alors de machines CC-NUMA (Cache Coherency NUMA). Si cette mémoire virtuellement partagée facilite la programmation, les techniques algorithmiques employées pour obtenir des applications performantes restent celles employées sur les machines à mémoire distribuée car les contraintes physiques sur le temps d'accès aux données restent pratiquement les mêmes.

Il existe une exception récente et notable à ce paysage homogène, la machine TERRA [113]. Elle utilise des processeurs spécialisés "multiflots", qui sont capables de calculer un très grand nombre de flots d'instructions en parallèle. Les processeurs utilisent ce très grand nombre de tâches concurrentes à exécuter pour recouvrir les temps d'accès à la mémoire. Pour l'instant, le marché visé semble être celui du très haut de gamme et des applications spécialisées. Cette architecture innovante est encore en phase de validation expérimentale afin de prouver son efficacité (que ce soit pour la performance pure ou la programmation).

En résumé, les machines parallèles et les réseaux d'ordinateurs sont de plus en plus semblables. Leur architecture semble converger vers celui d'un réseau de SMP. Coté programmation, de nouvelles facilités fournies par le matériel apparaissent. La possibilité qu'un grand nombre de noeuds du réseau partagent le même espace d'adressage est la plus importante pour la facilité de programmation.

1.1.2 Les différents modèles de programmation

La programmation parallèle la plus simple consiste à avoir plusieurs flots d'instructions (en anglais, *Threads*) qui s'exécutent en manipulant des données stockées dans une mémoire commune [85, 84, 81, 18]. Elle est donc naturellement le type de programmation le plus employé sur les machines SMP. La programmation peut se faire avec des outils spécialisés comme les bibliothèques implantant la norme POSIX [74]. Elle peut aussi utiliser des langages qui intègrent les *threads* dans leur sémantique, comme Java [92]. La difficulté de cette programmation est de garantir la sémantique correcte du programme en synchronisant les accès à la mémoire entre calculs se déroulant de manière concurrente. Pour obtenir des per-

formances, il suffit d'avoir toujours suffisamment de calculs concurrents à faire pour occuper les différents processeurs.

La programmation d'une machine à mémoire distribuée impose plusieurs contraintes. Il faut exprimer explicitement quelles vont être les données transmises et à quels moments les transmissions doivent se faire. La façon standard de programmer est d'utiliser une bibliothèque d'échanges de messages comme MPI [45, 46] ou PVM [52], qui permet de s'abstraire de la réalité physique de la machine ou du réseau, et de se concentrer sur les échanges de données. Ces outils ont permis la mise au point d'applications parallèles efficaces sur des architectures extensibles avec des rapports prix/performance très avantageux (actuellement jusqu'à 10000 processeurs). Ils ont sûrement beaucoup contribué à l'homogénéisation des architectures de machine. En contrepartie de la grande taille des machines, le temps de transmission de l'information est très grand par rapport à la puissance de calcul des processeurs. Cela induit que la programmation parallèle devient plus complexe : il faut partager les données et les activités pour maintenir les processeurs occupés, et, dans le même temps, minimiser les communications, coûteuses en temps.

Les architectures deviennent maintenant des hybrides de ces deux modèles. Des outils comme la bibliothèque athapascan-0 [21], ou Nexus [47] combinent les fonctionnalités d'une bibliothèque d'échanges de message comme MPI et d'une bibliothèque de contrôle de flots multiples comme POSIX Threads. Ce style de programmation permet de coder efficacement des applications hautement irrégulières comme les problèmes à N-Corps que l'on trouve en dynamique moléculaire [9]. Il utilise l'asynchronisme des flots de calculs pour recouvrir les temps d'attentes des communications et gagner ainsi en efficacité.

Les machines à mémoire virtuellement partagée CC-NUMA permettent une programmation aisée, comme sur une machine SMP, et sont extensibles, comme les machines à mémoire distribuée. Mais, pour obtenir une application efficace, il faut résoudre les mêmes problèmes de distributions des calculs et des données sur la machine physique que pour les machines à mémoire distribuée car les différences d'ordre de grandeur entre puissance de calcul et temps de transmission des données n'ont pas changé.

Certains outils tentent de paralléliser automatiquement des applications. On trouve par exemple le langage HPF. Les compilateurs doivent détecter le parallélisme disponible dans l'application et donc analyser tous les accès mémoire. Ensuite il faut réussir à générer un code parallèle efficace. La parallélisation automatique se heurte à plusieurs difficultés :

– la détection du parallélisme. Elle est souvent limitée à la détection de schémas d'accès aux données simples (tableaux, avec une fonction d'accès affine);

– l'exécution efficace par rapport aux meilleurs compilateurs séquentiels qui peuvent utiliser de très nombreuses techniques d'optimisation sur ces codes simples. Ces techniques ne seront pas employées à la même échelle en parallèle, d'où une perte d'efficacité dans la rapidité des codes.

La parallélisation automatique est utilisée sur des codes simples pour lesquels on ne souhaite qu'une petite accélération.

Pour faciliter la parallélisation, il est possible d'indiquer au compilateur des directives qui vont l'aider à prendre ses décisions. OpenMP est le nouveau standard qui définit ces directives. Elles consistent simplement à indiquer où placer les données et comment effectuer les calculs.

De nouveaux outils [17, 49] apparaissent, se plaçant à un niveau intermédiaire entre la parallélisation explicite et la parallélisation automatique. Ces outils implantent des langages qui permettent de séparer l'expression du parallélisme du placement réel des données et des calculs sur la machine. Le programmeur fixe les différents niveaux de découpes possibles et indique les contraintes de communication et de synchronisation nécessaires à une exécution correcte de l'application. Au lieu de définir les communications de données, le programmeur définit les données elles-mêmes. Ainsi l'environnement d'exécution peut garantir leur valeur en fonction des contraintes imposées par le programmeur. Par exemple dans Cilk [17], le programmeur pose des barrières de synchronisation. Le système va alors garantir que les données partagées seront à jour après la barrière. En Athapascan-1 [49], le programmeur spécifie les accès en lecture et en écriture de chaque donnée partagée. Le système peut alors garantir l'état d'une variable avant l'appel d'une fonction qui doit la lire. Les synchronisations peuvent ainsi être gérées beaucoup plus finement.

Une autre différence entre Athapascan-1 et Cilk concerne le fait que Cilk est un véritable langage avec son propre compilateur tandis qu'Athapascan-1 est une librairie C++ générique. Cilk souffre donc du même problème que les compilateurs HPF, le code généré n'est pas forcément le plus efficace. À l'inverse Athapascan-1 peut utiliser le meilleur compilateur C++ disponible pour générer son code. Cet effet est encore accentué par le fait que, dans ces deux langages, des portions de l'exécution peuvent s'exécuter séquentiellement afin d'éviter le surcoût de la parallélisation. Mais, pour obtenir une application performante, ces outils laissent au programmeur la décision du placement des données et des calculs.

Avec ces environnements de haut niveau, il est plus facile de montrer que le placement des données et l'ordonnancement des calculs sont cruciaux pour les performances. Par exemple Doreille [34] a montré que l'on pouvait dépasser les performances des environnements classiques d'algèbre linéaire à échange de message comme Scalapack, sur une grappe de SMP, pour des applications non triviales

comme une factorisation de Cholesky. Pour cela, il a utilisé un ordonnancement approprié et une exécution asynchrone à flots multiples. Il a aussi montré que même sur des machines d'architectures similaires (processeurs standard et réseau rapide), le meilleur algorithme d'ordonnancement n'est pas toujours le même.

Pour toutes ces méthodes de programmation parallèle, l'ordonnancement des calculs et le placement des données sont deux facteurs importants pour concevoir une application parallèle efficace.

1.2 Les modèles classiques d'ordonnancement

Le comportement réel d'une application est relativement facile à prévoir en séquentiel. On peut aisément approcher le temps d'exécution d'un programme, en étudiant par exemple sa complexité.

En parallèle, d'infimes variations dans l'environnement d'exécution, que ce soit sur un noeud de calcul ou sur la rapidité du réseau, peuvent changer complètement le temps d'exécution d'un algorithme non déterministe sensible aux conditions de course (race condition). Quelques outils de réexécution déterministes [39] existent et peuvent faciliter le débogage des applications.

Le comportement réel d'une application est donc très souvent difficile à prévoir, et plus encore à optimiser. Dans une exécution séquentielle, les portions de code où le programme passe le plus de temps, sont facilement identifiables. En parallèle, l'optimisation du code n'est pas suffisante, il faut aussi que l'ordre des opérations et leurs lieux d'exécution soient judicieusement choisis. Dans le cas contraire, même si chaque opération se déroule rapidement, les processeurs de la machine peuvent rester inactifs en attendant des données calculées sur d'autres processeurs. Le problème de ce choix est nommé dans ce mémoire, problème *d'ordonnancement*. Nous le définissons plus formellement dans la section 1.2.1.

Le temps d'exécution va donc dépendre de toutes les charges de calculs de tous les processeurs et de la charge du réseau. En pratique, pour pouvoir concevoir et évaluer des algorithmes, les machines parallèles sont modélisées plus ou moins finement. Les différences entre les modèles concernent principalement la modélisation des communications.

1.2.1 Généralités sur les modèles

Pour modéliser l'exécution d'une application parallèle, il faut décrire l'application et modéliser la machine qui l'exécute [27].

Une application est représentée par un graphe orienté $G(V, E)$ [25]. Un sommet du graphe représente un calcul local à un processeur. Ces calculs locaux sont nommés *tâches*.

Les arcs du graphe représentent les contraintes de précédence entre calculs. Par exemple un arc peut modéliser le fait qu'une tâche attend un résultat produit par une autre tâche.

Le graphe peut être pondéré. La pondération d'un noeud représente le coût (nombre d'instructions, temps, etc.) du calcul associé à ce noeud. La pondération d'un arc représente le volume de données à transmettre d'un noeud à un de ses successeurs.

Une tâche est *prête* si tous ses prédécesseurs ont déjà été exécutés et que les données utiles, calculées par les prédécesseurs de la tâche, ont été acheminées dans la mémoire locale du processeur où vont avoir lieu les calculs.

Si les données sont déjà présentes localement, elles n'ont pas besoin d'être communiquées. Le coût de transfert de données entre deux tâches exécutées par le même processeur est donc considéré comme nul.

Si les données ne sont pas présentes localement, il va falloir les communiquer. Une tâche ne devient prête que lorsque toutes les données en provenance de tous ses prédécesseurs sont finalement arrivées. La date à laquelle elle devient prête dépend donc du nombre de prédécesseurs, du volume des données à transférer et du temps que va mettre le réseau pour effectuer chacun de ces transferts.

Les différences entre les modèles d'exécution correspondent à des différences dans l'évaluation du coût de telles communications. Ces différences ont un impact sur les stratégies d'ordonnancement [55].

Formellement nous définissons donc le problème du calcul d'un ordonnancement σ sur m processeurs identiques comme suit :

Définition 1 *Un ordonnancement σ d'un graphe $G(V, E)$ est une paire de fonctions $(Date, Proc)$, avec $Date : V \rightarrow R^+$ et $Proc : V \rightarrow 1, 2, ..., m$ qui assignent à chaque tâche, respectivement, sa date de début d'exécution et son processeur d'exécution.*

Nous allons aussi utiliser les notations suivantes :

Définition 2 *Le temps d'exécution de la tâche i est t_i, sa date de terminaison $Term(i) = Date(i) + t_i$. $succ(i)$ est l'ensemble des successeurs de la tâche i.*

Dans les modèles suivants, nous nous limitons au cas où chaque processeur ne peut exécuter qu'une seule tâche à la fois, sans pouvoir la préempter. Un ordonnancement σ, valide, respecte la propriété suivante :

$$\begin{aligned} \forall i, j \in V, \quad & \text{tel que } Proc(i) = Proc(j) \\ & Term(i) \leq Date(j) \\ \text{ou} \quad & Term(j) \leq Date(i) \end{aligned}$$

De plus, pour qu'un ordonnancement soit valide, il faut qu'il respecte les

contraintes de précédences entre les tâches. Ces contraintes dépendent du modèle de coûts des communications.

1.2.2 Modèles à coût de communications nul

Beaucoup de travaux ont été menés en négligeant l'influence des communications. Cette hypothèse est plus ou moins réaliste. Elle est justifiée pour les applications dont les coûts de calcul sont très grands devant les coûts de communication et pour des exécutions se déroulant sur des machines parallèles à mémoire partagée.

Un ordonnancement σ doit respecter la contrainte suivante :

$$\forall j \in succ(i), Term(i) \leq Date(j)$$

.

Ordonnancer un graphe sans tenir compte des coûts de communication est relativement facile. De très bonnes garanties de performance peuvent être obtenues avec des heuristiques simples, par exemple, en commençant à exécuter une des tâches prêtes sur le premier processeur qui devient disponible. Quel que soit l'ordre dans lequel les tâches sont placées, l'exécution dure moins de 2 fois plus longtemps que le meilleur ordonnancement [60]. Pour être précis, la garantie est au pire de $2 - \frac{1}{m}$, où m est le nombre de processeurs de la machine.

En fonction du graphe de précédences, de bien meilleures approximations peuvent être trouvées. Par exemple, si toutes les tâches sont indépendantes, en plaçant la plus grosse tâche d'abord, on obtient un rapport de performance avec le meilleur ordonnancement de $4/3$. L'exemple des tâches indépendantes est en fait *pleinement approximable*. Pour tout ϵ, un ordonnancement s'exécutant en moins de $1 + \epsilon$ fois le temps du meilleur ordonnancement peut être construit en temps polynômial (polynôme en n, le nombre de tâches, et $\frac{1}{\epsilon}$). Les détails de cet algorithme d'ordonnancement et de son analyse se trouvent par exemple dans le chapitre 1 de [67].

Pratiquement, ce modèle est réaliste dans deux cas :

- la mémoire de la machine parallèle est physiquement partagée. Il n'y a pas de communications car les données sont toujours "locales";
- le coût de calcul de chaque tâche est très grand devant le coût d'une communication. Les communications n'influencent pas réellement le temps d'exécution si l'algorithme est déterministe.

Un premier modèle plus général consiste à prendre comme modèle du temps d'acheminement des données de la mémoire d'un processeur à un autre, une fonction de la taille des données.

1.2.3 Le modèle délai

La première extension possible est de considérer un délai d constant, lors de la transmission d'un message dans le réseau entre deux tâches situées sur des processeurs différents [102, 24]. Ce délai est une fonction de la taille des données à acheminer. Les processeurs peuvent calculer librement sans être "gênés" par les communications. Il n'y a pas, dans ce modèle, de contention (embouteillage) sur le réseau.

Un ordonnancement σ doit donc respecter la contrainte de précédence suivante :

$$\begin{aligned} \forall j \in succ(i), \quad & \text{tel que } Proc(i) \neq Proc(j) \\ & Term(i) + d \leq Date(j) \end{aligned}$$

L'ordonnancement dans ce modèle est généralement plus difficile que l'ordonnancement avec coût de communication nul [70]. La difficulté d'ordonnancer un graphe de tâches dans ce modèle dépend du rapport entre le plus petit coût de calcul d'une tâche et le plus grand coût de communications entre deux tâches.

1. Dans les problèmes dits à petit temps de communications, le coût de communication est plus petit que le coût calcul. Ce sont des problèmes relativement simples car ils sont proches des algorithmes sans communication [64].
2. Dans les problèmes dits à grand temps de communication, le coût de communication est plus grand que le coût calcul. Trouver une solution proche de la solution optimale devient plus difficile dans le cas général. Il est possible de limiter le nombre de communications en dupliquant des tâches [94, 65, 1].

Lorsqu'une application parallèle est considérée à son grain le plus fin, le coût des communications est souvent bien supérieur à celui des quelques calculs locaux.

En pratique, des algorithmes de regroupement linéaire comme DSC [54] sont utilisés pour ordonnancer ces graphes. Ils consistent à diviser le graphe en chaînes critiques. Une chaîne critique est un chemin dans le graphe avec des communications de coût important. Une chaîne est exécutée par le même processeur. Le problème revient alors à distribuer ces chaînes sur les processeurs en essayant de minimiser le coût des communications entre chaînes. Ceci peut être fait par un algorithme de partitionnement de graphe non-orienté. Malheureusement, ces algorithmes ont des garanties de performance qui dépendent du coût de la plus grande chaîne de communication [71].

En utilisant la duplication, il est possible de construire des ordonnancements fonction de la racine carrée du plus grand rapport entre le coût d'une tâche de calcul et celui d'une communication [100].

Approximations faites dans le modèle délai

En fait, le modèle délai néglige deux aspects importants de la modélisation des communications d'une application parallèle :

- le surcoût d'exécution dû à la gestion des communications : Pile de protocole à l'envoi et à la réception, interruptions, etc. À cela, on peut encore ajouter les éventuelles copies de mémoires lors des communications. Dans le modèle délai, un grand nombre de communications peuvent être faites sans surcoût, tant qu'elles sont recouvertes par du calcul.
- la contention due aux goulots d'étranglements du réseau. Dans les algorithmes par phases, tous les processeurs ont tendance à envoyer et recevoir leurs messages en même temps. Suivant l'architecture et la performance du réseau, cela peut entraîner un ralentissement important dans la vitesse d'acheminement d'un message.

Ces deux aspects sont partiellement pris en compte par deux autres modèles, *logP* et *BSP*.

1.2.4 Les modèles $logP$ et BSP

Ces deux modèles sont un raffinement du modèle délai pour être plus proche du comportement matériel : $logP$ modélise plus finement le coût d'une communication tandis que BSP est en fait un modèle de machine et d'exécution.

Le modèle $logP$

Le modèle $logP$ [28] est une extension du modèle délai plus proche du véritable comportement d'une machine parallèle. Comme pour le modèle délai, les paramètres de la machine parallèle sont le nombre de processeurs (le P de $logP$) et le temps de transmission du message d'un processeur à l'autre (le l). Mais le modèle $logP$ tient compte de deux paramètres supplémentaires : le surcoût en calcul d'une communication (o) et le débit du réseau (g), borne inférieure du temps qu'il faut attendre avant de pouvoir traiter le prochain message.

Les contraintes de précédences à respecter dans ce modèle sont plus complexes. Un ordonnancement σ est valide s'il est la restriction d'un ordonnancement σ' valide dans le modèle délai (avec $d = l$) du graphe $G'(V', E')$ construit à partir de $G(V, E)$ comme suit. V' et E' sont des sur-ensembles de V et E. Pour

tout $e_{i,j} \in E$, tel que $Proc(i) \neq Proc(j)$, on ajoute les tâches $o_{i,j}$ et $o_{j,i}$ telles que :

$$t_{o_{i,j}} = t_{o_{j,i}} = o$$

$$o_{i,j} \in succ(i), o_{j,i} \in succ(o_{i,j}), j \in succ(o_{j,i})$$

De plus l'ordonnancement σ' doit respecter la contrainte :

$$\begin{aligned} \forall i,j,k,l, \quad & \text{tel que } Proc(o_{i,j}) = Proc(o_{k,l}) \\ & Date(o_{i,j}) + g \leq Date(o_{k,l}) \\ & \text{ou } Date(o_{k,l}) + g \leq Date(o_{i,j}) \end{aligned}$$

En pratique, sur les réseaux locaux actuels à fort débit, g n'est pas un facteur limitant car il est plus petit que o. o dépend fortement des contraintes matérielles et logicielles. Le protocole de communication utilisé pour une transmission influence fortement sa valeur. Si le matériel du réseau garantit, par exemple, une transmission sans perte et sans erreur, le protocole de communication peut être allégé. La librairie BIP [99] permet un gain notable sur la latence d'une communication dans un réseau Myrinet par rapport à l'utilisation du protocole IP sur ce même réseau. Le même gain est obtenu pour la bibliothèque de communication d'IBM sur les machines SP, lors de l'utilisation du réseau rapide de la machine par un unique processus sur chaque noeud.

Le modèle BSP

Le modèle BSP [111] est à la fois un modèle de programmation et d'exécution. Il prend à sa charge la résolution des problèmes de contention dans une application. L'idée est de séparer les calculs et les communications. Un algorithme est vu comme une succession de phases synchrones (un superstep) de calculs locaux puis de communications globales (de coût C).

Les contraintes de précédences sont ici aussi un peu plus complexes. Soit S_n l'ensemble des tâches exécutées dans le n-ième superstep.
$Date(S_n) = \min_{i \in S_n} Date(i)$ et $Term(S_n) = \max_{i \in S_n} Term(i) + C$. Bien sur

$$\forall k,n, Term(S_n) \leq Date(S_k) \text{ ou } Term(S_k) \leq Date(S_n)$$

Les contraintes de précédences sont alors les suivantes :

$$\begin{gathered} \forall i \in S_n, j \in S_k \text{ et } j \in succ(i), \\ Term(j) + C \leq Term(S_n) \text{ et } n = k \text{ et } Proc(i) = Proc(j) \\ \text{ou } Term(S_n) \leq Date(j) \text{ et } n < k \end{gathered}$$

Cette séparation permet d'implanter des algorithmes de communications tirant efficacement parti du réseau [58]. Ces algorithmes sont capables de gérer efficacement la contention en découpant les messages en paquets et en les acheminant par plusieurs chemins en même temps. Il existe d'autres modèles proches de BSP, comme CGM [19].

Ordonnancement dans les modèles $logP$ et BSP

Quelques travaux existent sur l'ordonnancement dans ces modèles, par exemple [82] pour $logP$ ou [56] pour BSP. Néanmoins il est difficile d'obtenir des ordonnancements avec une bonne garantie de performance.

Pour $logP$, l'ajout du paramètre o pose un nouveau problème. Comme dans le modèle délai, on essaie de recouvrir les temps de communications par du calcul, mais il faut en plus minimiser le nombre de communications, ce qui est bien sur antagoniste des problèmes d'équilibrage de la charge de calcul. De plus les problèmes à grand temps de communication ne sont pas plus faciles à résoudre dans logP car le modèle délai n'est qu'une restriction de logP au cas ou o et g sont nuls.

L'ordonnancement dans le modèle BSP consiste à découper l'application en phases de calculs locaux et de communications globales. Si l'application n'est pas "naturellement" équilibrée, les phases de communications doivent être multipliées pour distribuer la charge. Ces étapes de communications sont synchrones, et donc les processeurs ne travaillent pas pendant cette phase. Le plus mauvais ordonnancement ajoute une phase de communication synchrone pour chaque communication dans le graphe. Cette contrainte de partition et de communication synchrone rend le problème d'ordonnancement plus difficile que le modèle délai. Pour citer un exemple, le problème de l'allocation de chaînes de tâches UET est résolu dans le modèle délai en temps linéaire sur processeurs identiques, mais est NP-Difficile au sens fort dans BSP [57].

Formellement, si l'on considère que le temps d'une communication délai est égal à celui d'une communication BSP globale, on peut transformer tout ordonnancement dans le modèle BSP vers le modèle délai sans allonger sa durée d'exécution. Par contre, transformer un ordonnancement dans le modèle délai en un ordonnancement dans le modèle BSP peut introduire des temps d'attentes égaux au coût de synchronisation de chaque communication, donc un surcoût qui peut être proportionnel au nombre de communications. L'ordonnancement optimal dans BSP s'obtient en minimisant le nombre de phases de communication et en équilibrant le travail. Ce problème se réduit à 3-DM (3 Dimensional Matching, problème NP-Difficile au sens fort) pour des graphes simples (chaînes UET) [57].

1.2.5 Conclusion partielle

Les graphes de tâches représentant une application sont difficiles à générer et difficiles à manipuler de part leurs grandes tailles.

L'ordonnancement de graphe est difficile à réaliser efficacement avec des garanties qui ne soient pas fonction du nombre et du coût des communications, même dans un modèle aussi simple que le modèle délai.

Si l'on utilise un modèle de communication plus proche de la machine mais plus complexe, comme logP ou BSP, l'ordonnancement devient alors encore plus difficile.

Il paraît donc difficile d'appréhender le problème de l'ordonnancement avec communications de façon efficace et réaliste à la fois. La plupart des utilisateurs des machines parallèles connaissent, en général, suffisamment bien leur application pour guider les premiers choix de parallélisation. Cette méthode empirique a permis l'émergence de nombreux travaux implantant efficacement des applications parallèles. Nous proposons dans ce mémoire une autre méthode d'ordonnancement, où les communications ne sont pas négligées mais ne sont prises en compte que de manière implicite. Plutôt que d'exprimer explicitement les communications, l'utilisateur doit guider l'analyse à partir des connaissances qu'il a de son code et exprimer le regroupement logique de calculs qui doivent s'exécuter ensemble sur la machine. Nous allons utiliser une mesure de la performance de l'exécution de chacun de ces groupes (tâches malléables) en fonction du nombre de processeurs qui l'exécute. Cette mesure va nous permettre d'effectuer un ordonnancement des groupes sans avoir à tenir compte explicitement des communications. Ce que nous discutons dans le chapitre suivant.

Chapitre 2

Le Modèle des Tâches Malléables

Contents

Un des objectifs de ce travail de thèse est de promouvoir le modèle des tâches malléables en montrant qu'il permet d'ordonnancer efficacement de véritables applications parallèles. Dans ce chapitre, le modèle des tâches malléables est présenté et comparé qualitativement aux modèles classiques présentés dans le chapitre précédent. Nous proposons dans les chapitres suivants plusieurs algorithmes pour ordonnancer des tâches malléables indépendantes (cf chap. 3) ou en présence de relations de précédence (cf chap. 4) Nous montrons aussi comment utiliser ce modèle pour appréhender l'implantation d'une véritable application. Nous traitons l'exemple d'une simulation adaptative de courant océanique qui est détaillé au chapitre 5.
Dans un deuxième temps, nous abordons quelques aspects généraux de l'ordonnancement de Tâches Malléables. Nous rappelons certains résultats déjà acquis en ce qui concerne la complexité du problème général de l'ordonnancement de tâches malléables. Le modèle des tâches malléables est apparenté aux tâches multiprocesseurs et aux tâches divisibles. Les résultats de ces deux modèles peuvent être transposés dans le cas des tâches malléables. Nous étudions le passage d'une solution obtenue dans le modèle des tâches divisibles, où la ressource-processeur est continue, à une solution malléable. Nous présentons un algorithme de transformation qui permet d'obtenir une garantie constante de performance par rapport à la solution malléable optimale.

2.1 Présentation du modèle

Depuis les années 1980, de nombreux travaux ont été réalisés afin de concevoir, pour de véritables applications, des codes parallèles efficaces. Nous avons précédemment montré que l'un des problèmes majeurs dans la conception d'une application parallèle efficace est le choix, pour chaque calcul, du lieu et de la date de son exécution. Il faut donc résoudre ce qui est appelé usuellement un *problème d'ordonnancement*.

L'implantation d'une application est fondée sur une étude algorithmique. Cette étape dans la conception d'une application parallèle prend souvent en compte le problème d'ordonnancement.

Le choix du lieu d'exécution d'un calcul impose la présence en ce lieu de l'ensemble des données nécessaires à ce calcul. Ces données, si elles ne sont pas présentes localement, devront donc être acheminées à travers le réseau. Une communication sur un réseau est une opération lente à l'échelle d'un ordinateur : l'échelle de temps utilisée pour les calculs est maintenant de l'ordre de quelques nanosecondes ($10^{-9}s$). Le PowerPC G4 fournit une performance réelle d'un Gigaflops (1 milliard d'opérations flottantes par seconde). Si le débit des réseaux augmente lui aussi de façon impressionnante, le temps de transmission minimum (la latence) ne

diminue pas aussi vite. Une communication sur les meilleurs réseaux prendra au moins quelques microsecondes. Par exemple, il faut $13.10^{-6}s$ à un réseau Myrinet pour acheminer une donnée à travers le réseau. Le rapport entre le temps pour effectuer un calcul et le temps de faire une communication est donc de 10000. Si les données sont acheminées par un réseau "standard" (Ethernet 100), les écarts dans les ordres de grandeur sont encore plus importants.

L'ordonnancement consiste à distribuer équitablement le travail pour que les ressources de calcul de la machine parallèle soient toujours occupées. Le nombre et le volume de communications induites par cette distribution pour maintenir la cohérence des calculs ne doivent pas être trop importants afin de ne pas pénaliser les performances. L'ordonnancement essaie aussi de minimiser les problèmes de synchronisation. Un calcul sur un processeur ne peut commencer que lorsque les données sont localement disponibles en mémoire. Des temps d'attente peuvent apparaître si un processeur est inoccupé jusqu'à l'arrivée d'une communication avec des données.

Ces fluctuations dans le matériel ou dans la taille des problèmes traités ont une influence sur la performance. Elles influent aussi sur la qualité pour un support donné des différents ordonnancements possibles. Même en se restreignant à la même classe de machines, un ordonnancement peut être à la fois meilleur qu'un autre sur une machine particulière et moins bon sur une autre. Choisir l'ordonnancement au moment de la conception de l'application n'est donc pas forcément une approche efficace. Lorsque l'on sépare l'algorithme de son ordonnancement, il est alors possible d'adapter cet ordonnancement afin qu'il tienne compte de la réalité physique de la machine parallèle qui exécute l'application. La séparation entre l'ordonnancement et l'algorithme permet de tester différents algorithmes pour choisir le plus efficace.

Cette séparation entre l'algorithme et l'ordonnancement peut être faite lors de l'écriture de l'application par l'utilisation d'outils de programmation de haut niveau comme Cilk [17], Jade [103], NESL [16], Athapascan-1 [49].

Le modèle des tâches malléables est une façon de séparer la programmation de l'application de l'ordonnancement final de cette application. Il ne contraint pas le choix de l'outil de programmation parallèle, mais guide la conception de l'implantation.

Il existe une très large littérature considérant le problème de l'ordonnancement des tâches d'un programme parallèle (pour ne citer que les ouvrages généraux ou articles de synthèse : [23, 67, 35]). L'éventail des travaux réalisés va des travaux théoriques sur des modèles abstraits, aux outils pratiques comme les implantations d'applications sur la plupart des plateformes d'exécution parallèle ou distribuée.

Dans le cadre de cette thèse, nous ne nous intéressons qu'à la minimisation du temps d'exécution de l'ordonnancement (le *makespan*), c'est-à-dire la plus grande date de terminaison d'une tâche du graphe. D'autres critères peuvent être utilisés

comme la date moyenne de terminaison des tâches.

Parmi les diverses approches possibles pour résoudre le problème d'ordonnancement, la plus commune est de considérer le programme à son grain le plus fin et d'appliquer un algorithme de regroupement (en anglais *clustering*) pour essayer de diminuer le surcoût dû aux communications [54]. Le principal inconvénient de cette approche est que les communications sont prises en compte explicitement. Cela entraîne deux problèmes évoqués au chapitre précédent :

- Les communications sont souvent exprimées à l'aide d'un modèle simplifié de l'architecture sous-jacente. Cette modélisation ne correspond pas forcément au comportement asynchrone et non-déterministe des communications de l'application.
- La gestion des communications rend le problème d'ordonnancement souvent plus difficile à résoudre [70], y compris pour des modèles simples comme le modèle *délai*.

Un nouveau modèle d'exécution a récemment été proposé : le *système de tâches parallèles* [36] (en anglais, *parallel task system*) dont le *modèle des tâches malléables* [110] est issu.

Une tâche malléable est le regroupement d'un ensemble de tâches séquentielles exécutées en parallèle par un nombre de processeurs choisi par la stratégie d'ordonnancement. Le modèle des tâches malléables s'apparente à deux autres modèles de tâches exécutées par plusieurs processeurs : le modèle des tâches multiprocesseurs [50] et le modèle des tâches divisibles [107, 14]. Les trois modèles diffèrent dans la liberté de choix du nombre de processeurs qui vont exécuter une tâche.

Dans le modèle des tâches multiprocesseurs, ce nombre est un entier naturel fixé entre 1 et m, où m désigne le nombre total de processeurs de la machine. Dans le modèle des tâches divisibles, l'ordonnancement détermine un nombre réel entre 0, exclu, et m, définissant la "quantité" de processeurs alloués. Le nombre de processeurs exécutant une tâche malléable est un compromis entre ces deux modèles : c'est un entier, choisi par la politique d'ordonnancement entre 1 et m.

2.1.1 Motivation

Une application parallèle est traditionnellement modélisée par un graphe pondéré. Les sommets représentent les tâches, les arcs, les contraintes de dépendance ou de précédence [25]. Le poids des sommets et des arcs représente respectivement les coûts de calcul et de communication. Cette modélisation est parfois étendue. Par exemple, un graphe de flot de données est un graphe biparti où des sommets supplémentaires s'ajoutent à ceux des tâches de calculs, pour modéliser plus finement les données échangées.

En pratique, le graphe est difficile à manipuler explicitement dans son entier à cause de sa grande taille. Par exemple, une simple multiplication de matrices 1000x1000 représente déjà un million de multiplications et d'additions. Quelques approches existent pour réduire cette taille, comme l'utilisation d'une représentation paramétrisée [26].

Pour la plupart des problèmes, l'ordonnancement avec délai de communication non nul est reconnu généralement pour être plus difficile que l'ordonnancement du même problème où ce délai est négligé [70]. C'est-à-dire que les garanties de performances sont plus mauvaises, ou que la même garantie est obtenue au prix d'une complexité accrue. Cette différence est accentuée lors de l'utilisation de modèle de communication plus fin que le modèle délai comme LogP [28].

D'un coté, pour que tous les processeurs soient occupés tout le temps, il faut distribuer la charge de calcul et donc distribuer les données entre les processeurs. De l'autre, la gestion de ces données distribuées en parallèle demande des communications et des synchronisations (attentes). Augmenter la distribution augmente donc le volume de communications. La production d'un ordonnancement efficace implique la résolution de problèmes d'optimisation difficiles afin de trouver un compromis entre ces deux aspects.

Il est difficile de prévoir l'efficacité de l'exécution parallèle d'un ensemble de tâches. De ce fait, l'impact du surcoût de parallélisation est généralement négligé dans les ordonnancements. Mais si le comportement de l'exécution parallèle d'un ensemble de tâches séquentielles communicantes est difficile à prévoir, en revanche, il est souvent aisément mesurable. En particulier, lorsque le déroulement de l'exécution des tâches (algorithme et ordonnancement) est déterministe. L'exécution d'un grand nombre d'applications parallèles dépend des données en entrée. Par contre, ces applications sont souvent formées de différentes parties déterministes. Par exemple, les schémas de calculs itératifs convergent souvent au bout d'un nombre d'itérations rarement prédictible, mais chacune de ces itérations est composée des mêmes opérations. Le comportement moyen global de chaque sous-ensemble de tâches composant l'application peut être mesuré en fonction du nombre de processeurs l'exécutant, et de son ordonnancement interne.

L'utilisateur ou l'environnement fournissent le graphe de tâches malléables avec les informations sur l'efficacité de l'exécution des tâches. Le modèle des tâches malléables utilise ces informations, facilement accessible, pour essayer de «mieux» ordonnancer l'application parallèle. Les décisions d'ordonnancement d'une application se fondent alors sur les performances réelles en exécution de parties de l'application plutôt que sur un modèle négligeant les communications.

Le modèle des tâches malléables est décrit formellement dans la section 2.2. Intuitivement c'est un moyen de concevoir une application parallèle efficace pour

plusieurs raisons :

- Il permet de refléter la structure logique de beaucoup d'applications, comme par exemple la décomposition de domaines, ou d'autres méthodes hiérarchiques. Ces applications sont la composition de plusieurs opérateurs parallèles comme par exemple des opérateurs de résolution de systèmes linéaires. Ces opérateurs sont connus du programmeur de l'application. L'ordonnancement d'un de ces opérateurs est souvent assez direct, de même que la répartition des données. Le modèle des tâches malléables manipule l'application à son grain "naturel".
- Les tâches malléables simplifient l'expression du problème car l'utilisateur n'a plus à manipuler explicitement les communications lors du calcul de l'ordonnancement. Par contre l'utilisateur doit fournir des outils génériques capables de mettre en place les communications nécessaires au bon déroulement de l'algorithme.
- Le même formalisme peut être employé à plusieurs niveaux de granularité.
- Le modèle peut être enrichi pour tenir compte des possibilités dépendant de l'architecture ou des capacités du système parallèle comme l'hétérogénéité des processeurs ou du réseau.
- Les tâches malléables reflètent aussi la hiérarchie à deux niveaux d'ordonnancement des applications parallèles apparaissant sur les nouvelles architectures. Ces nouvelles architectures sont les grappes (réseaux de machines à mémoire partagée). Le parallélisme se situe à deux niveaux : le parallélisme local, en mémoire partagée, avec différents flots d'exécution (*threads*) dont l'ordonnancement est sous le contrôle du système et le parallélisme en mémoire distribuée qui utilise, par exemple, une bibliothèque de communication comme MPI [45]. Des outils sont maintenant disponibles pour faciliter la programmation de telles applications [91, 47, 21].

Une tâche malléable est un objet de plus gros grain qu'une tâche séquentielle classique puisqu'elle est le regroupement de plusieurs tâches séquentielles. Ce grain plus gros nous permet de négliger plus facilement le coût des communications entre deux tâches malléables. Le problème de l'ordonnancement à grand temps de communication est relativement difficile (il n'y a pas d'approximation connu) [1]. De plus, le graphe malléable est plus petit que le graphe entier et donc il est plus facilement manipulable. Il est à noter que, même si une tâche malléable définit un groupe de tâches séquentielles ordonnancées ensemble, cela n'impose pas d'autres contraintes sur l'ordonnancement interne à la tâche malléable. Il peut être fait au mieux, en essayant par exemple de recouvrir les communications entre tâches malléables par du calcul.

Le modèle des tâches malléables tient compte des communications, mais seulement de manière implicite, en utilisant un critère simple de mesure de la per-

formance de l'exécution d'une tâche malléable. Ce paramètre est nommé facteur d'inefficacité dans le reste de ce document et est présenté dans la section 2.3.

2.2 Définition formelle

Soit $G(V, E)$, un graphe orienté où V représente l'ensemble de tâches malléables de l'application et E représente l'ensemble des contraintes de précédence entre les tâches. Soit m, le nombre de processeurs de la machine cible.

Nous définissons le problème du calcul d'un ordonnancement malléable σ sur m processeurs identiques comme :

Définition 3 *Un ordonnancement malléable σ d'un graphe $G(V, E)$ est une paire de fonctions $(Date, Alloc)$, avec $Date : V \rightarrow R^+$ et $Alloc : V \rightarrow 1, 2, ..., m$ qui assignent à chaque tâche, respectivement, sa date de début d'exécution et le nombre de processeurs qui l'exécute.*

Le temps requis pour exécuter une tâche i sur $q = Alloc(i)$ processeurs, est noté $t_{i,q}$. La date de terminaison de l'exécution d'une tâche est donc $Term(i) = Date(i) + t_{i,q}$.

Définition 4 *Le temps requis pour exécuter une tâche i sur $q = Alloc(i)$ processeurs, est $t_{i,q}$. Le travail d'une tâche sur q processeurs est $W_{i,q} = q\, t_{i,q}$*

Un ordonnancement $\sigma = (Date, Alloc)$ est valide s'il respecte les contraintes suivantes :

$$\forall j \in succ(i), Term(i) \leq Date(j)$$
$$\forall j \in V, \sum_{\substack{\forall i, \\ Date(i) \leq Date(j) \\ Term(i) > Date(j)}} Alloc(i) \leq m$$

L'ordonnancement doit donc respecter les contraintes de précédence et au plus m processeurs sont impliqués dans les calculs des tâches à un instant donné.

Définition 5 *Le temps d'exécution ω d'un ordonnancement σ est la plus grande date de terminaison,*

$$\omega = \max_{i \in V} Term(i)$$

Dans ce mémoire nous traitons de la minimisation de ω, le *makespan* (C_{max}). Nous cherchons un ordonnancement valide de temps d'exécution ω^* minimum.

2.3 Le facteur d'inefficacité

Nous proposons de tenir compte implicitement du surcoût dû aux communications et aux synchronisations à l'intérieur d'une tâche malléable par le *facteur d'inefficacité*. Il est à rapprocher d'autres mesures de l'efficacité d'une application parallèle, comme les fonctions d'iso-efficacité de Kumar, Grama, Karypis et Gupta [83]

Dans un monde idéal, un système parallèle avec m processeurs devrait être capable de terminer l'exécution d'une application m fois plus vite qu'une machine avec un seul processeur. Dans la réalité, un système à m processeurs ne permet pas une telle accélération car les communications entre processeurs sont beaucoup plus lentes que les calculs, en particulier sur toutes les machines à mémoire physiquement distribuée.

Le temps d'exécution d'une tâche malléable sur m processeurs est donc souvent plus grand que le temps d'exécution de la tâche malléable sur 1 processeur, divisé par m. Le rapport entre l'accélération obtenue et l'accélération idéale théorique dépend principalement du nombre m de processeurs et de la taille N des données à traiter. Ce rapport est le facteur d'inefficacité que nous utilisons. Il sera noté $\mu(m, N)$. L'idée est de prendre en compte grossièrement (implicitement) le surcoût de la gestion du parallélisme à l'intérieur d'une tâche i.

Nous rappelons que nous notons $t_{i,q}$ le temps nécessaire au calcul d'une tâche i sur q processeurs.

Définition 6 *Le facteur d'inefficacité $\mu_{i,q}$ de la tâche i sur q processeurs est :*

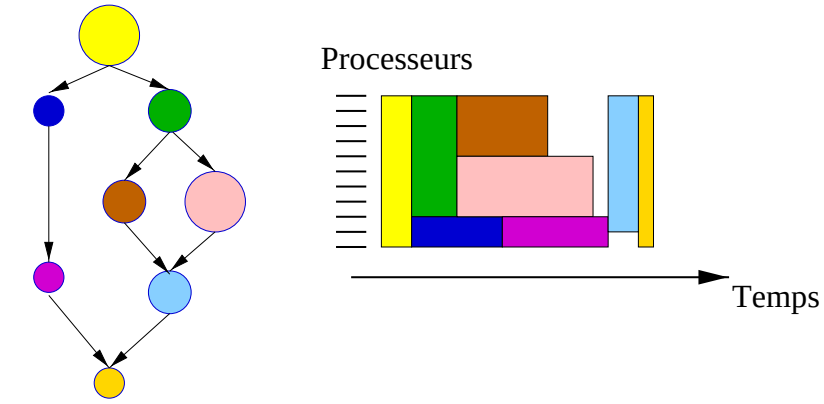
$$\mu_{i,q} = \frac{q\ t_{i,q}}{t_{i,1}} \tag{2.1}$$

Généralement le facteur d'inefficacité augmente avec le nombre de processeurs et diminue lorsque la taille des données augmente. Ce n'est pas toujours le cas, comme nous le verrons dans la section suivante. Une interprétation géométrique du problème est une façon pratique d'aborder l'ordonnancement des tâches malléables et de mieux en appréhender leurs comportements.

2.3.1 Interprétation géométrique

Une tâche multiprocesseurs, et donc une tâche malléable peut être représentée géométriquement par un rectangle. La longueur d'un des cotés du rectangle correspond au nombre de processeurs utilisés. L'autre dimension du rectangle correspond au temps d'exécution de la tâche. Un ordonnancement consiste à placer un ensemble de rectangles dans une boîte en respectant les contraintes de précédence.

La boîte de rangement est limitée dans une des dimensions par le nombre total de processeurs disponibles, le temps d'exécution de l'ordonnancement étant l'autre dimension. Le but est de minimiser la taille, en temps, de la boîte.



Graphe de Tâches Malléables

Ordonnancement du graphe

FIG. 2.1 – *Représentation de l'ordonnancement d'un graphe de tâches malléables*

La représentation de l'ordonnancement d'un graphe de tâches malléables est donnée en exemple dans la figure 2.1.

Ce type de représentation nécessite une hypothèse de *continuité*. C'est-à-dire que les rectangles sont placés sur un ensemble de processeurs adjacent. Cette hypothèse peut parfois être levée sans pénalité. Il suffit que les coûts de communications, entre deux processeurs quelconques du réseau, soient toujours les mêmes. C'est souvent le cas lorsque les noeuds du réseau sont des monoprocesseurs (que ce soit pour les réseaux de stations de travail ou les machines parallèles). Lorsque la machine est homogène, c'est surtout la façon de faire les regroupements des calculs et données qui est déterminante. La distribution de ces groupes sur les noeuds de la machine n'a que peu d'influence.

Pour les noeuds multiprocesseurs, il y a deux niveaux de communications (locale et distante). Dans un réseau hétérogène, le comportement de l'exécution d'une tâche malléable devient plus complexe. Cette complexité se retrouve alors dans la fonction d'inefficacité qui doit tenir compte de cette hétérogénéité.

Plus le nombre de processeurs exécutant une application est grand, plus le nombre, et souvent le volume, de communications gérant la cohérence des données distribuées, augmentent. Le même phénomène s'applique aussi aux tâches malléables. Chaque partie d'une tâche malléable doit donc exécuter plus de travail.

Propriété 1 $\mu_{i,q}$ *est une fonction croissante de* q.

Si une tâche s'exécute sur 1 seul processeur en un temps $t_{i,1} = T$ (cf Fig. 2.2), elle ne s'exécute pas en un temps plus petit que $T/2$ sur deux processeurs (cf Fig. 2.2), mais en un temps $t_{i,2} = \mu_{i,2}\ T/2$ (cf Fig. 2.2). Sur 4 processeurs, le temps d'exécution devient $t_{i,4} = \mu_{i,4}\ T/4$. Géométriquement si $\mu_{i,2} < \mu_{i,4}$, cela signifie que la surface du surcoût est plus grande dans une exécution sur 4 processeurs que sur 2.

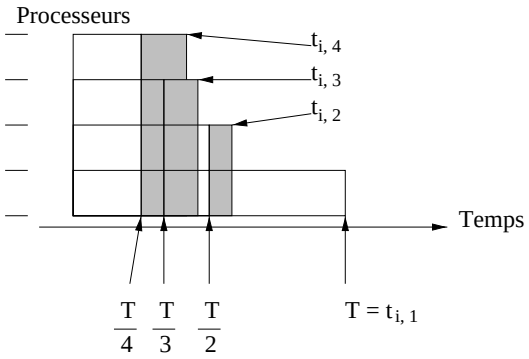


FIG. 2.2 – *Influence du facteur d'inefficacité sur le temps d'exécution d'une tâche*

Compte tenu du comportement typique d'une application parallèle, la forme générale du facteur d'inefficacité $\mu_{i,q}$ peut être décomposée, en fonction du nombre q de processeurs utilisés, en trois intervalles consécutifs, fonctions du nombre de processeurs (cf fig. 2.3).

1. Le premier intervalle (I) correspond au surcoût de la mise en place du parallélisme : par rapport à un programme séquentiel, le programme parallèle doit initialiser les bibliothèques de communications et mettre en place la gestion distribuée des données et des calculs. En plus de ces aspects constants de l'inefficacité, il faut ajouter les aspects dépendants du découpage : volume et nombre de communication.
   Typiquement, cet intervalle correspond au passage de l'exécution de la tâche malléable de 1 à 2 processeurs.
2. Le second intervalle (II) de processeurs correspond aux régions où l'ajout d'un processeur se fait à faible coût car les mécanismes nécessaires à la parallélisation sont déjà en place. L'augmentation du travail nécessaire est proportionnelle au nombre de processeurs. Le facteur d'inefficacité est alors quasiment linéaire. En effet, chaque ajout d'un processeur introduit de nouveaux calculs (distribution et nouvelles communications). Le seul facteur diminuant, est le volume des communications. Mais, il n'est pas, souvent, le facteur limitant.
3. Le dernier intervalle (III) apparaît lorsque les calculs distribués ne sont plus

suffisants pour maintenir efficacement les processeurs occupés. Les processeurs passent leur temps à s'attendre mutuellement. Le facteur d'inefficacité croît alors de façon importante. Cette dégradation de la performance s'accroît. Après l'ajout d'un certain nombre de processeurs, l'ajout d'un processeur augmente le temps, au lieu de le réduire.

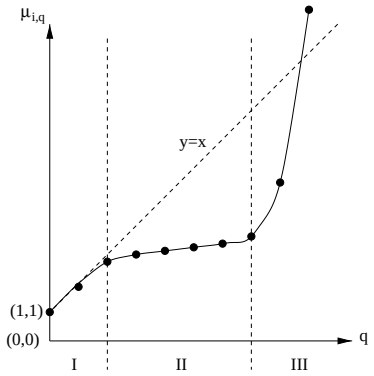


FIG. 2.3 – *Facteur d'inefficacité typique*

Un certain nombre d'hypothèses qualitatives peuvent être faites sur le facteur d'inefficacité. En fonction de ces hypothèses, différentes approximations du problème d'ordonnancement peuvent être obtenues.

2.3.2 Hypothèses sur le facteur d'inefficacité

Un certain nombre d'hypothèses sur le facteur d'inefficacité μ peuvent être faites en fonction de l'application parallèle modélisée. Ces hypothèses vont permettre d'obtenir de bonnes garanties de performance de l'ordonnancement du graphe de tâches malléables.

Une première hypothèse, concerne l'augmentation du travail réalisé par une tâche malléable si l'on augmente le nombre de processeurs l'exécutant.

Hypothèse 1 *μ est une fonction croissante du nombre de processeurs exécutant la tâche malléable :*

$$\frac{\partial \mu_{i,q}}{\partial q} \geq 0$$

Cette hypothèse permet de pouvoir plus facilement borner le travail à accomplir par une tâche malléable. Pour minimiser le travail, il suffit de minimiser le nombre de processeurs exécutant les tâches malléables.

Pour des raisons pratiques ou théoriques, cette hypothèse n'est pas toujours vérifiée. Les effets de cache peuvent provoquer des accélérations super-linéaires

si l'ajout d'un processeur à la découpe du problème permet à chacun de ces morceaux de tenir dans le cache mémoire. L'ordonnancement interne d'une tâche malléable n'est pas forcément toujours proche de la valeur du travail divisée par le nombre de processeurs. Observons l'ordonnancement d'une tâche malléable composée de trois tâches séquentielles indépendantes de mêmes durées T. L'exécution de cette tâche malléable va terminer en $2\ T$ sur deux processeurs et en T, sur 3 (cf fig. 2.4). Le facteur d'inefficacité passe alors de $\frac{4}{3}$ à 1.

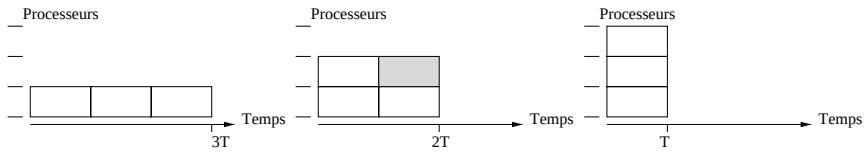


FIG. 2.4 – *Passage de l'exécution d'une tâche malléable de 1 à 2 puis 3 processeurs où l'hypothèse 1 est fausse*

Le but du parallélisme est de permettre l'exécution plus rapide de l'application. Le temps d'exécution d'une tâche malléable peut être décroissant si le nombre de processeurs augmente. Du point de vue du travail, il augmente donc moins vite que l'ajout de processeurs. Cela se traduit pour le facteur d'inefficacité par l'hypothèse suivante :

Hypothèse 2 *μ est une fonction du nombre de processeurs exécutant la tâche malléable, de pente inférieure ou égale à $\frac{1}{q}\mu_q$:*

$$\frac{\partial \mu_{i,q+}}{\partial q} \leq \frac{1}{q}\mu_q$$

Cette hypothèse est souvent vérifiée, mais jusqu'à une certaine valeur limite du nombre de processeurs. Au-delà de cette valeur limite, le nombre de calculs à réaliser pour chacun des processeurs devient petit devant le coût des communications. L'ajout d'un processeur à l'allocation augmente le temps d'exécution de la tâche malléable.

Cette hypothèse permet de concevoir des algorithmes pour lesquels l'ajout de processeurs à une allocation ne va pas augmenter le temps d'exécution. Cela permet de borner le travail qu'une tâche malléable réalise.

La combinaison de ces deux hypothèses indique que le temps d'exécution d'une tâche est une *fonction décroissante* du nombre de processeurs et que le travail réalisé par une tâche malléable est une *fonction croissante* du nombre de processeurs. Cette combinaison d'hypothèses est appelée *hypothèse de monotonie*

Hypothèse 3 *Un problème d'ordonnancement de tâches malléables est dit monotone si μ remplit les hypothèses 1 et 2.*

Définition 7 *Un* sous-ensemble d'allocation monotone *d'une tâche malléable est un sous-ensemble de* $\{1 \dots m\}$ *tel que les allocations associées vérifient les hypothèses de monotonie.*

Ce sous-ensemble est construit comme suit : une allocation sur q processeurs de la tâche malléable fait partie du sous-ensemble d'allocation monotone si

1. $\forall n \geq q, W_{i,n} \geq W_{i,q}$, c'est-à-dire que le travail accompli par la tâche exécutée par n processeurs est plus grand que le travail accompli par la tâche exécutée par q;
2. $\forall n \leq q, t_{i,n} \geq t_{i,q}$, c'est-à-dire que le temps d'exécution sur q est plus petit que le temps d'exécution sur n

Il y a au moins une allocation possible, celle de travail minimal. En effet, par définition, il n'y a pas d'allocation avec un travail plus petit, donc la première condition est satisfaite. De plus toute allocation, sur un nombre plus petit de processeurs, a, elle aussi, un travail plus grand et donc, par définition, un temps plus grand (puisque $t_{i,q} = \frac{W_{i,q}}{q}$), ce qui satisfait la deuxième condition.

Ces restrictions sont utilisées par certains algorithmes d'ordonnancement comme prétraitement afin de simplifier la gestion des cas généraux d'allocation. Mais la plupart des nouveaux algorithmes que nous proposons dans les chapitres suivants utilisent l'hypothèse de monotonie sans ces restrictions.

Des hypothèses plus restrictives sur le facteur d'inefficacité peuvent être faites. Elles ne concernent alors que certaines classes d'applications.

Le facteur d'inefficacité μ peut être convexe. Dans ce cas, la parallélisation devient de moins en moins efficace, la variation de cette efficacité est elle aussi monotone.

Hypothèse 4 *μ est une fonction strictement croissante et convexe (de dérivée croissante) du nombre de processeurs.*

Cette hypothèse est plus forte que la simple hypothèse 3 de monotonie.

Une fonction particulière peut être choisie pour le facteur d'inefficacité.

Hypothèse 5 *$\mu(p) = p^{\alpha}$, ou tout autre fonction de p.*

Cette hypothèse est utilisée dans un modèle proche de celui des tâches malléables, le modèle des tâches divisibles. Nous présentons dans la section 2.4.4 les différences entre ces deux modèles.

Enfin, pour certaines applications, la même fonction μ est appliquée à toutes les tâches malléables du graphe, ou à chaque sous-ensemble de ces tâches malléables. Il peut s'agir de toutes les tâches appartenant à une structure de précédence particulière, comme une chaîne.

Hypothèse 6 *La même fonction μ s'applique à chaque sous-ensemble de tâches malléables.*

Nous utilisons ces dernières hypothèses en liaison avec les algorithmes issus de modèles proches de celui des tâches malléables des chapitres suivants.

2.3.3 Hypothèses sur l'exécution des tâches malléables

Dans ce mémoire, nous avons utilisé l'hypothèse suivante :

Hypothèse 7 *Une tâche malléable s'exécute en même temps sur tous les processeurs qui la calculent et chaque processeur exécute au plus une tâche malléable à la fois.*

Un certain nombre d'hypothèses peuvent aussi avoir un sens pratique suivant les fonctionnalités fournies par le support d'exécution et la facilité à contrôler l'exécution d'une tâche malléable.

Une première possibilité concerne l'arrêt de l'exécution d'une tâche malléable, c'est-à-dire qu'une tâche peut être interrompue à tout moment pour être ensuite redémarrée par les mêmes processeurs. Cela nécessite que le support d'exécution soit capable de sauvegarder un contexte d'exécution distribuée cohérent d'une tâche malléable. Une implantation peut être de partager chaque noeud entre plusieurs processus (lourds ou légers) de manière préemptive. Les mécanismes de contrôles de l'exécution des processus sont alors facilement utilisables : signaux, variables d'exclusion mutuelle, sémaphore, etc.

Une extension de cette hypothèse concerne la possibilité de reprendre la tâche avec des processeurs différents sans changer leur nombre. Cela demande de pouvoir migrer le contexte d'exécution d'une tâche malléable, mais surtout d'être capable de reconstruire son espace d'état des communications. En particulier, il faut gérer les communications qui étaient en cours au moment de l'arrêt (réception ou redirection). Dans le cadre du parallélisme, c'est une hypothèse réaliste car il est possible d'implanter des mécanismes distribués d'enregistrement de l'état du système (en anglais, *snapshot*).

Enfin, si une tâche est capable de reconstruire finement son environnement à partir du *snapshot* de l'exécution, il est alors possible d'imaginer qu'une tâche continue son exécution sur un nombre différent de processeurs.

Si la continuation d'une tâche est relativement complexe à implanter, il est possible d'utiliser simplement des mécanismes de redémarrage de la tâche depuis le début de son exécution. Cela signifie que les opérations de communications entre les tâches malléables peuvent être répétées sans provoquer d'erreur ou bien il faut regrouper les communications pour les faire uniquement en fin et en début d'exécution des tâches.

2.3.4 Discussion pratique sur les accélérations super-linéaires

Des anomalies d'accélération sont courantes pour les problèmes d'optimisation combinatoire où l'ajout de processeurs change en fait l'ordre de parcours de l'espace de recherche. Nous avons aussi déjà évoqué les anomalies d'ordonnancement. Mais un facteur d'inefficacité ayant une pente négative et donc une accélération super-linéaire, peut apparaître comme irréaliste dans le contexte du calcul numérique parallèle.

En particulier, on pourrait penser qu'il suffit d'utiliser la meilleure implantation séquentielle. Elle peut être une émulation d'un programme parallèle, avec, par exemple, les changements de contextes ou le parcours de la mémoire qui assurent la plus grande localité des références. C'est faux lorsque l'on travaille à ressources bornées, comme souvent, sur un noeud d'une machine parallèle.

En pratique, un tel comportement peut se présenter si les calculs utilisent d'autres ressources que la simple puissance de calcul des processeurs impliqués. La principale ressource utilisée en plus de la puissance de calcul est la mémoire. Dans tous les ordinateurs modernes, cette mémoire est organisée comme une hiérarchie de couches successives de support physique de mémoire dont la taille est inversement proportionnelle au temps d'accès. Cela va des registres du processeur, en passant par les un ou deux niveaux de mémoire cache, la RAM et la mémoire virtuelle (swap) sur disques, jusqu'à la copie des fichiers, des disques sur bandes.

Nous allons maintenant décrire deux applications réelles sur lesquelles des accélérations super-linéaires peuvent être atteintes : la factorisation de Cholesky dense et la simulation de dynamique moléculaire.

Effet de cache dans la factorisation de Cholesky dense

Les opérations simples d'algèbre linéaire sont les fonctions de bases de la plupart des applications scientifiques. Elles utilisent pleinement les pipelines arithmétiques flottants, à haute performance, des processeurs. Le principal problème est de maintenir ce pipeline alimenté lors de ces opérations.

Pour illustrer ce point nous avons mesuré les performances d'une factorisation de Cholesky [59]. C'est l'un des noyaux numériques les plus utilisés. Pour les petites tailles de matrices, la vitesse de calcul dépend beaucoup du fait que la matrice tienne en mémoire cache, ou pas. Le lecteur pourra trouver dans [32] une discussion de l'importance des caches dans ce cadre. Si l'on augmente le nombre de processeurs entre deux exécutions en parallèle, de façon à ce que les blocs de la matrice tiennent dans le cache, l'accélération obtenue est alors plus grande que la fraction de processeurs ajoutée [7]. On a donc une accélération super-linéaire, du moins en théorie, car les matrices sont de petites tailles. Cet effet se retrouve aussi pour les grandes matrices lorsqu'elles ne tiennent plus entièrement dans la

mémoire physique et que des portions de ces matrices sont envoyées sur disque par le mécanisme de mémoire virtuelle.

La figure 2.5 présente le temps d'exécution atteint par la routine de factorisation de Cholesky de la bibliothèque parallèle Scalapack [32] sur un IBM SP, pour une matrice de taille 7000x7000.

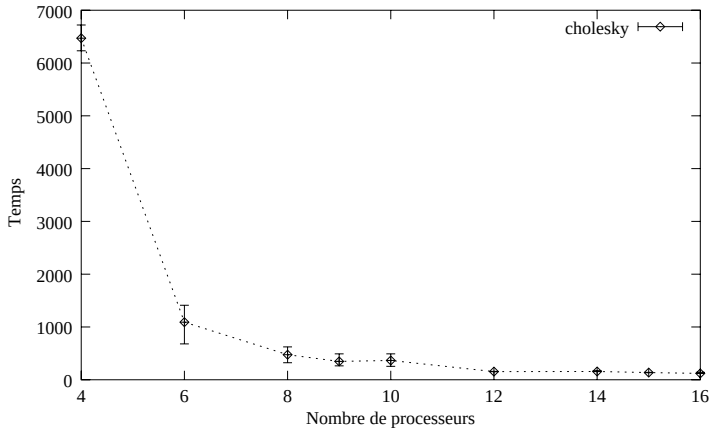


FIG. 2.5 – *Temps d'exécution (moyenne, min. et max. en seconde pour 30 tirages par point) du Cholesky dense de scalapack, pour une matrice 7000x7000 avec des blocs de taille 100x100, sur une IBM SP-1*

En premier lieu, il faut noter que les calculs ne s'exécutent pas sur moins de 4 noeuds.

La SP permet un ensemble de mesures très stables lorsque les effets de cache ne sont pas en jeu (cf fig 2.7).

Le temps plus important de la première expérimentation s'explique par un effet de chargement de l'exécutable et des bibliothèques de fonction sur l'ensemble des noeuds (Il s'agit de la première expérience effectuée pour tous les tests).

Lorsque la place mémoire occupée par les données dépasse les capacités de la machine, le mécanisme de mémoire virtuelle swappe les données, mais il swappe aussi le code. Les temps d'exécution deviennent alors beaucoup plus variables. La figure 2.8 présente un histogramme des variations des temps pour les exécutions sur 6 et 8 processeurs.

Pour tenir compte de cette variabilité, nous avons choisi de représenter aussi les minima et les maxima pour chaque point. Malgré ces variations, les temps reportés dans la figure 2.5 montrent une accélération super-linéaire. Ce point particulier est encore plus visible sur la figure 2.6 : le travail est le produit du temps d'exécution par le nombre de processeurs utilisés. Sur cette figure, il est décroissant lorsque le nombre de processeurs augmente. Utiliser plus de processeurs diminue donc le nombre de "calculs" effectués.

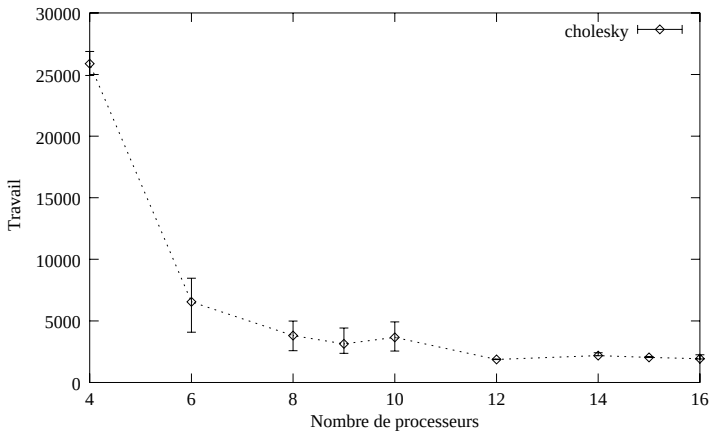


FIG. 2.6 – *Travail (temps multiplié par le nombre de processeurs, moyenne, min. et max. pour 30 tirages par point) du Cholesky dense de Scalapack pour une matrice 7000x7000 avec des blocs de taille 100x100, sur une IBM SP1*

Dynamique moléculaire

La simulation de dynamique moléculaire peut être modélisée comme un problème de N-corps. Les calculs sont irréguliers lorsque les calculs des interactions sont limités spatialement (rayon de coupure). Un calcul efficace nécessite l'utilisation de techniques avancées pour recouvrir les communications par des calculs. Pour simuler le comportement de protéines, les interactions sont calculées entre des centaines de milliers d'atomes [10]. Les calculs utilisent donc une grande quantité de mémoire.

Pour certains ordinateurs parallèles haut de gamme, comme le Cray T3E, dans le but d'optimiser les communications et de simplifier le matériel, il n'y a pas de gestion de la mémoire virtuelle. La mémoire disponible pour une application est donc fortement limitée. Si une instance du problème ne tient pas dans la mémoire physique, l'exécution ne peut avoir lieu directement. Deux solutions sont alors possibles : implanter un mécanisme de mémoire virtuelle à la main ou diminuer l'utilisation mémoire de l'algorithme. Réimplanter un mécanisme de mémoire virtuelle consiste à sauvegarder sur disques les résultats intermédiaires de calculs. Bien sûr, cela augmente le temps d'exécution. Lorsque le nombre de processeurs et les mémoires associées deviennent suffisants pour stocker le problème en entier, une accélération super-linéaire devrait être observée.

Nous allons montrer un exemple où un algorithme s'adapte à la mémoire disponible. La figure 2.9 présente les accélérations obtenues pour une application de dynamique moléculaire en fonction du nombre de processeurs alloués pour différentes tailles de problèmes : de 11615 à 413039 atomes [9]. La ligne de référence $y = x$ est aussi représentée. La plus grande simulation ne s'exécute pas

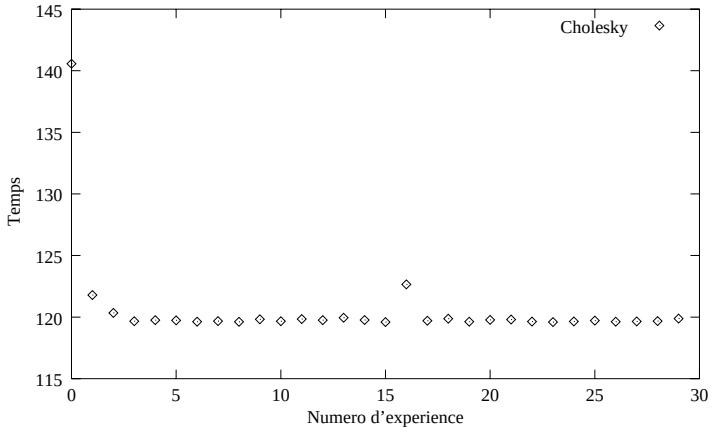


FIG. 2.7 – *30 mesures sur 16 processeurs du Cholesky dense de Scalapack pour une matrice 7000x7000 avec des blocs de taille 100x100, sur une IBM SP1*

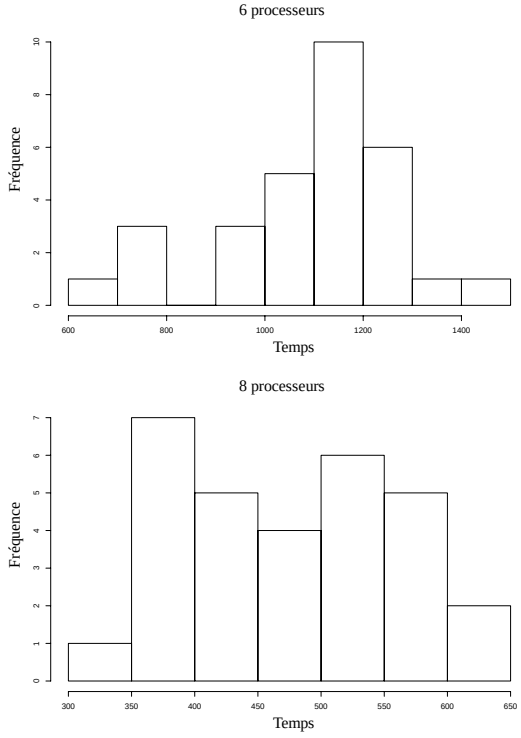


FIG. 2.8 – *Histogrammes de 30 mesures pour 6 et 8 processeurs du Cholesky dense de Scalapack pour une matrice 7000x7000 avec des blocs de taille 100x100, sur une IBM SP1*

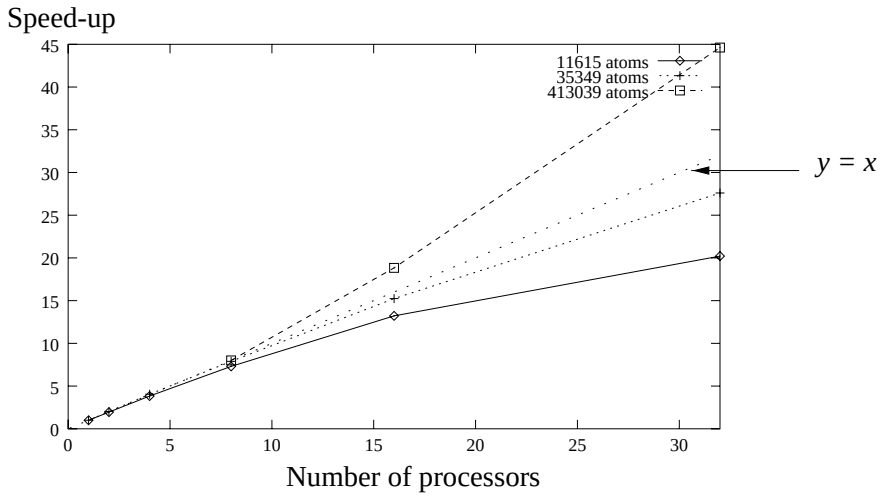


FIG. 2.9 – *Dynamique moléculaire : accélérations vs. le nombre de processeurs*

sur un nombre de processeurs inférieur à 8. Des listes d'interactions doivent être conservées de manière à pouvoir accélérer les calculs. Ces listes ne tiennent pas complètement en mémoire sur 8 processeurs. Cela induit des calculs supplémentaires. Au fur et à mesure que le nombre de processeurs augmente, de plus en plus d'interactions sont stockées en mémoire, ce qui accélère les calculs au-delà de la fraction de processeurs ajoutés. Cette accélération super-linéaire n'apparaît que jusqu'à 32 processeurs. Après cette limite, le surcoût de la parallélisation et les déséquilibres de la charge ne permettent plus une telle accélération.

Nous venons donc de présenter deux applications réelles de calculs numériques dans lesquelles une accélération super-linéaire des calculs peut être rencontrée. En pratique ces applications ne sont pas les plus fréquentes.

2.3.5 Evaluation du facteur d'inefficacité

Pour toute application, l'un des problèmes est l'obtention d'une bonne estimation du facteur d'inefficacité pour pouvoir efficacement ordonnancer le graphe de tâches malléables.

- Pour certaines applications, l'exécution d'une seule tâche malléable est aisée. Le facteur d'inefficacité peut alors être mesuré empiriquement. Cette méthode permet de connaître concrètement le temps moyen d'exécution d'une tâche malléable. La mesure du facteur d'inefficacité est décrite au chapitre 5, section 5.4.4 pour une application de simulation océanique.
- Pour d'autres applications, le facteur d'inefficacité peut être évalué analytiquement en analysant la complexité des opérations. Cela devrait être pos-

sible par exemple pour une factorisation de Cholesky creuse parallèle [37].

Après avoir décrit tout au long de ce chapitre le modèle des tâches malléables et la façon dont il peut être utilisé pour ordonnancer une application, nous présentons dans la suite de ce chapitre et les chapitres suivants quelques stratégies et techniques d'ordonnancement des graphes de tâches malléables.

2.4 Généralités sur l'Ordonnancement de Tâches Malléables

2.4.1 Résultats de complexité

Du et Leung [36] ont d'abord montré que le modèle des tâches malléables (sous le nom de *Parallel Task System*) est une extension de modèles classiques. Nous détaillons maintenant leurs preuves pour leurs intérêts pédagogiques. En effet nous utilisons des idées similaires pour nos algorithmes.

Théorème 1 *Les problèmes d'ordonnancement classiques discrets, comme l'ordonnancement de tâches monoprocesseur et de tâches multiprocesseurs, peuvent être transformés en un problème d'ordonnancement de tâches malléables.*

Preuve :
Le nombre de processeurs alloués à chaque tâche dans le problème initial est nommé *allocation initiale*. La transformation du problème initial, proposée par Du et Leung, utilise un facteur d'inefficacité μ approprié qui "force" le nombre de processeurs utilisé pour l'exécution de chaque tâche. Le facteur d'inefficacité est construit de façon à ce que :

- Le temps d'exécution d'une tâche est infini, si son allocation en processeurs est plus petite que l'allocation initiale.
- Le temps d'exécution d'une tâche est égal à son temps d'exécution dans le problème initial si son allocation en processeurs est supérieure ou égale à son allocation initiale.

Trivialement, tout ordonnancement avec un temps d'exécution plus petit que l'infini utilise au moins le nombre de processeurs de l'allocation initiale pour exécuter chaque tâche.

De plus puisque l'allocation initiale est la plus petite allocation possible pour une tâche, tout ordonnancement ayant des tâches allouées sur plus de processeurs que leur allocation initiale peut être transformé en un ordonnancement utilisant uniquement les allocations initiales en laissant inoccupés les processeurs supplémentaires pendant l'exécution des tâches.

Toute solution du problème malléable permet donc de construire une solution du problème multiprocesseurs initial. □

Cette transformation d'un problème à allocation fixée en un problème malléable montre clairement que la résolution du problème malléable général est aussi difficile que la résolution du problème multiprocesseurs.

Théorème 2 *Le problème de l'ordonnancement malléable est NP-Complet pour tout nombre de processeurs* $m \geq 2$*. Il est NP-Complet* au sens fort *pour les problèmes avec contraintes de précédence, et pour* $m \geq 5$ *pour les problèmes sans relation de précédence.*

Preuve :
Le problème multiprocesseurs est NP-Complet pour un nombre de processeurs $m \geq 2$ [13].

La NP-Complétude *au sens fort* a été prouvée par Du et Leung [36] en utilisant des réductions de *3-partition*[51]. □

Théorème 3 *Le problème de l'ordonnancement de tâches malléables sans relation de précédence peut être résolu en temps pseudo-polynomial pour 2 et 3 processeurs*

Preuve :
Les tâches de la même allocation sont groupées ensemble, les unes après les autres sur les mêmes processeurs. Du et Leung [36] placent ces groupes suivant des structures d'ordonnancement particulières. Ils ont montré que tout ordonnancement peut être transformé en un ordonnancement respectant ces contraintes et ayant le même temps d'exécution. Ils définissent M comme la somme des temps des tâches pour des exécutions de chaque tâche sur 1 processeur pour $m = 2$ ou le maximum des sommes pour 1 ou 2 processeurs pour $m = 3$. Il ne reste alors qu'à répartir les différentes tâches dans les différents groupes. Ils utilisent la programmation dynamique pour calculer en temps $O(nM^2)$ ($m = 2$) et $O(nM^5)$ ($m = 3$) cette répartition. M est la somme des temps des tâches pour des exécutions de chaque tâche sur 1 processeur pour $m = 2$ ou le maximum des sommes pour 1 ou 2 processeurs pour $m = 3$ (Ils ne se limitent pas au cas monotone). Pour trouver le meilleur ordonnancement, il suffit donc de trouver la meilleure répartition. □

Enfin, Ils ont aussi étudié le cas préemptif, que nous ne considérons pas dans ce mémoire. Dans ce modèle une tâche peut être exécutée par un nombre de processeurs différents après sa préemption. Nous allons étudier le cas non-préemptif, c'est-à-dire qu'une tâche malléable est exécutée sans interruption par le même sous-ensemble de processeurs. Il semble clair que la préemption pourrait réduire le temps d'exécution, comme dans le cas d'un ordonnancement classique, mais à un prix trop lourd pour sa mise en oeuvre pratique !

2.4.2 Précédents résultats pour les tâches malléables indépendantes

Le problème des tâches malléables indépendantes est *approximable* pour n'importe quel ensemble de tâches [75]. Cela signifie qu'il existe une famille d'algorithmes polynômiaux qui permet de résoudre le problème avec une précision ϵ aussi petite que l'on veut. Mais cette famille d'algorithmes n'est pas *pleinement approximable* : la complexité des algorithmes n'est pas un polynôme en $\frac{1}{\epsilon}$. En pratique, le facteur constant de la complexité de l'algorithme d'ordonnancement est trop grand pour que cet algorithme soit utilisable concrètement.

Turek, Wolf et Yu [110] ont proposé plusieurs algorithmes pratiques pour ordonnancer les tâches malléables. Chacun de ces algorithmes est composé de deux phases successives.

L'algorithme de Mounié, Rapine et Trystram [90, 100] permet de construire un ordonnancement qui est, au plus, à un facteur $\sqrt{3} + \epsilon$ de l'ordonnancement optimal.

Les algorithmes en deux phases

L'idée est d'utiliser deux algorithmes l'un après l'autre. Le premier choisit le nombre de processeurs alloués à chaque tâche, tandis que le second algorithme ordonnance le système multiprocesseurs ainsi obtenu.

Chacune de ces deux phases peut être plus ou moins compliquée et les deux phases peuvent être plus ou moins interdépendantes :

Turek et al. [110] ont introduit les algorithmes à deux phases pour résoudre le problème des tâches malléables indépendantes. La première phase consiste à trouver, en temps polynomial, une allocation des processeurs proche de l'allocation optimale.

Deux bornes inférieures classiques du temps d'exécution d'un ensemble de tâches (malléables ou non) sont utilisées : le chemin critique, c'est-à-dire le temps de la plus longue tâche et la somme des travaux des tâches. L'allocation de Turek et al. est un compromis entre ces deux critères. Elle consiste à réduire les plus longues tâches en leur allouant plus de processeurs, tant que cela n'engendre pas trop de travail.

Une fois que l'allocation de processeurs a été choisie, Turek et al. utilisent un algorithme de strip packing à 2 dimensions pour résoudre le placement (cf fig. 2.10).

La garantie de performance de l'algorithme est celle du strip packing à 2 dimensions. À notre connaissance, l'algorithme avec la meilleure garantie absolue est l'algorithme de Steinberg [108]. Cet algorithme a une garantie de performance

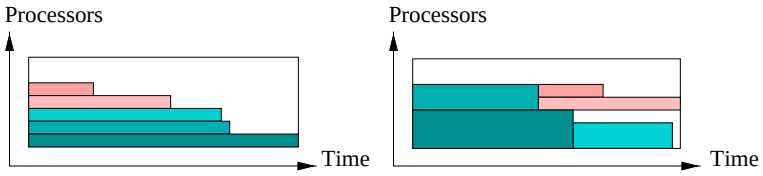


FIG. 2.10 – *Exemple d'ordonnancement de tâches malléables indépendantes*

de 2.

2.4.3 Ordonnancement en deux phases

Nous résolvons le problème, de l'ordonnancement d'un graphe orienté de tâches malléables, défini à la section 2.2.

Résoudre le problème de l'ordonnancement des tâches malléables, c'est donc résoudre les deux problèmes suivants :

- Le choix du nombre de processeurs $Alloc(i)$ qui vont exécuter la tâche i; c'est ce que nous nommons, *le problème d'allocation*.
- La réalisation de *l'ordonnancement* des tâches multiprocesseurs définies par l'allocation $Alloc(i)$.

La plupart des algorithmes et des heuristiques travaillant sur ce problème traitent chacun de ces deux problèmes séparément, en *deux phases*. La qualité de l'ordonnancement obtenu dépend de la qualité de chacun des deux algorithmes individuellement et de la qualité de la combinaison de ces algorithmes.

Lors de l'élaboration de l'algorithme, deux stratégies peuvent donc être employées :

1. Choisir deux algorithmes indépendants tel que chacun fournit une bonne garantie. Soit l'algorithme A_1 qui fournit une allocation $Alloc_1$ pour les tâches du graphe. Soit un nombre réel g_1 (la garantie de l'algorithme) tel que

$$\max(\sum_{i \in V} W_{i,Alloc_1(i)}, CP_{Alloc_1(i)}) \leq g_1 \min_{Alloc} \max(\sum_{i \in V} W_{i,Alloc(i)}, CP_{Alloc})$$

   CP_{Alloc} est le chemin critique induit par l'allocation $Alloc$.

   Soit A_2 un algorithme d'ordonnancement de graphe multiprocesseurs, de garantie g_2, tel que le makespan ω_2 vérifie :

$$\omega_2 \leq g_2 \ \min(\sum_{i \in V} W_{i,Alloc(i)}, CP_{Alloc})$$

La combinaison de ces deux algorithmes permet d'obtenir la garantie suivante :

$$\omega_2 \leq g_2\ g_1\ \min_{\forall Alloc} \max(\sum_{i \in V} W_{i,Alloc(i)}, CP_{Alloc})$$

L'ordonnancement malléable obtenu est donc de garantie $g_1\ g_2$ par rapport au meilleur ordonnancement malléable.

2. Choisir deux algorithmes imbriqués, c'est-à-dire tels que l'allocation des tâches malléables du graphe permet un ordonnancement multiprocesseurs simple. Le calcul de garantie de performance pour ces algorithmes doit être capable de borner le travail et le chemin critique d'un tel algorithme d'ordonnancement malléable. Cela peut être fait avec des arguments implicites fournis par une "prédiction" du temps d'exécution, et du travail, du meilleur ordonnancement malléable.
   C'est cette approche que nous avons choisie dans la section 3.3 pour ordonnancer efficacement des tâches malléables indépendantes.

Nous présentons et analysons dans la section 2.4.4 comment construire une solution malléable avec garantie de performance à partir d'une solution obtenue dans un modèle proche en modifiant les allocations des tâches.

2.4.4 Transformation d'ordonnancement d'un modèle proche vers les Tâches Malléables

Nous montrons dans cette section qu'il est possible de raisonner sur des ordonnancements dans les modèles proches pour obtenir un bon ordonnancement dans le modèle des tâches malléables.

Définition 8 *Une tâche multiprocesseurs est exécutée par un nombre entier fixé de processeurs.*

Le problème des tâches multiprocesseurs semble assez difficile à appréhender efficacement. Quelques résultats existent et sont présentés dans [35], mais aucun d'eux n'est général. Il est clair que tout résultat sur les tâches multiprocesseurs peut être directement utilisé pour ordonnancer un groupe de tâches malléables dont on aurait choisi, pour chaque tâche malléable, l'allocation en nombre de processeurs. Nous verrons dans la section 4.2.1 que le problème posé par la difficulté de l'ordonnancement de tâches multiprocesseurs peut être contourné efficacement par une nouvelle méthode.

Le modèle considéré dans cette partie sera le modèle des tâches divisibles, où une tâche peut utiliser un nombre fractionnaire de processeurs, c'est-à-dire toutes

les valeurs réelles de l'intervalle $]0; m]$.

Définition 9 *Une tâche divisible est exécutée par un nombre réel de processeurs choisi par la politique d'ordonnancement.*

Les travaux sur l'ordonnancement de graphes de tâches dans le modèle des taches divisibles [14, 107] utilisent la notion de vitesse d'exécution d'une tâche, ce qui correspond à la version continue du facteur d'inefficacité des tâches malléables, divisé par le nombre de processeurs. La fonction vitesse utilisée dans ces travaux est en fait l'inverse du facteur d'inefficacité que nous avons défini.

Pour un ensemble de tâches indépendantes, des solutions optimales simples existent si le facteur d'inefficacité est convexe ou concave [14].

Dans le cas concave, le facteur d'inefficacité devient plus petit lorsque le nombre de processeurs exécutant la tâche malléable augmente, c'est-à-dire que l'exécution d'une tâche est d'autant plus efficace que l'on ajoute des processeurs. L'accélération de l'application est dite super-linéaire. Le meilleur ordonnancement consiste à exécuter toutes les tâches les unes après les autres sur tous les processeurs, l'une après l'autre. Nous avons présenté quelques exemples dans la section 2.3.4 même si ce type d'application est relativement peu fréquent.

Dans le cas convexe, l'exécution devient moins efficace au fur et à mesure de l'ajout de processeurs. Nous détaillons et utilisons un résultat bien connu [14] dans le contexte malléable.

Il est à noter que les contraintes sur la forme du facteur d'inefficacité ne concernent que les algorithmes de la littérature pour les tâches divisibles. Dans le cadre de cette thèse, une simple hypothèse de monotonie suffit : le temps d'exécution des tâches est décroissant, et le travail croissant, en fonction de q, la fraction de processeur utilisée.

Théorème 4 *Dans le cas monotone, le meilleur ordonnancement consiste à exécuter toutes les tâches en même temps sur une fraction de processeurs afin qu'elles finissent toutes à la même date.*

Preuve :
Soit ω, le makespan de l'ordonnancement. Si une tâche est ordonnancée par une fraction de processeurs plus petite, elle va s'exécuter en temps supérieur à ω, ce qui augmente donc le makespan.

Si toutes les tâches sont exécutées sur une fraction au moins aussi grande de processeurs, sa surface augmente. La somme des surfaces des tâches augmente donc aussi. la surface devient donc plus grande que $m\ \omega$, ce qui augmente donc le makespan.

Donc l'allocation des tâches ne peut être changée sans augmenter le makespan. L'ordonnancement est donc optimal.

□

La seule difficulté est de trouver la date d'exécution pour savoir quelle fraction allouer à chaque tâche. Le calcul de cette date peut être résolu exactement par des méthodes d'optimisation non linéaire, ou alors on peut utiliser la méthode d'approximation duale [68] présentée dans la section 3.2. Elle consiste ici à effectuer une dichotomie sur la date limite, et à affecter à chaque tâche la fraction de processeurs nécessaire pour qu'elle s'exécute dans le temps imparti. La dichotomie converge vers la date pour laquelle la somme des fractions est égale au nombre de processeurs.

Quelques résultats existent pour les graphes plus généraux et des fonctions μ qui ont une forme particulière en q^α où q est la fraction de processeurs allouée à la tâche [107].

Cette contrainte permet d'obtenir, en temps polynômial, des ordonnancements divisibles optimaux pour des structures spécifiques comme des chaînes ou des arbres. Les algorithmes d'ordonnancement utilisent le fait que deux tâches de vitesse q_1^α et q_2^α placées en série ou en parallèle, peuvent être remplacées par une seule tâche équivalente de vitesse q_3^α. Cette stratégie peut être employée pour des graphes généraux, mais nécessite de résoudre des problèmes d'optimisation non-linéaires [97, 98].

Cette fonction vitesse q^α peut être une bonne approximation du comportement d'une tâche lorsque $\alpha < 1$ et que la tâche est exécutée par un nombre q de processeurs supérieur à 1. Cela correspond à un facteur d'inefficacité monotone (en fait, un facteur d'inefficacité convexe). Pour transformer une allocation divisible en une allocation monotone, il suffit d'arrondir la quantité de processeurs à un nombre entier. Par exemple, en arrondissant à la partie entière inférieure, le temps d'exécution n'est pas augmenté d'un facteur supérieur à 2 et le travail n'est pas augmenté. De plus, le placement étant déjà choisi par l'ordonnancement divisible, le temps d'exécution malléable est au plus à un facteur 2 de l'optimal.

Par contre, lorsque le nombre de processeurs est compris entre 0 et 1, cette fonction vitesse correspond à des accélérations superlinéaires. Ces algorithmes sur les tâches divisibles peuvent donc produire des ordonnancements divisibles qui peuvent être aussi loin que l'on souhaite de l'ordonnancement malléable. Prenons l'exemple d'un ensemble de $n > m$ tâches indépendantes de temps d'exécution 1 sur 1 processeur, avec la même fonction de vitesse $q^\alpha, \alpha < 1$. Chaque tâche va se voir allouer $\frac{m}{n}$ processeurs. Donc chaque tâche va s'exécuter en temps $\frac{1}{\left(\frac{m}{n}\right)^\alpha}$. Dans le modèle malléable chaque tâche va s'exécuter sur au moins 1 processeur et le travail total ne peut pas diminuer par rapport à cette allocation. Les n tâches sont donc ordonnancées en temps au moins égal à $\frac{n}{m}$ (cf figure 2.11).

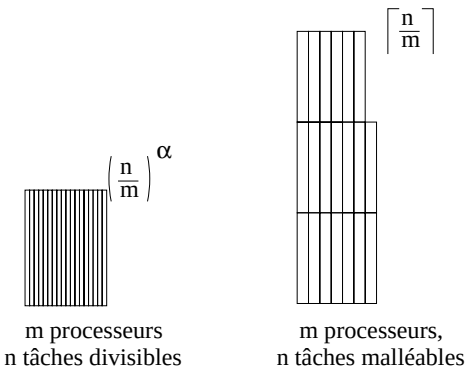


FIG. 2.11 – *Ordonnancement divisible et malléable de tâches indépendantes*

Le rapport des temps d'exécution entre les deux est alors plus grand que $\frac{n}{m}\ \left(\frac{m}{n}\right)^{\alpha}$, c'est-à-dire :

$$\left(\frac{n}{m}\right)^{1-\alpha}$$

Si n grandit, ce rapport peut être aussi grand que l'on veut. Il n'existe donc pas de stratégie d'ordonnancement général pour passer d'une solution dans le modèle divisible, avec vitesse en q^{α}, vers le modèle malléable avec une solution qui est à un coût constant de la valeur l'ordonnancement optimal divisible.

Par contre nous allons montrer comment construire, à partir des indications de la solution divisible, un ordonnancement malléable avec une garantie de performance par rapport à l'ordonnancement optimal malléable.

Algorithme de transformation

Même si la solution continue ne donne pas d'indication sur le temps d'ordonnancement de la solution malléable, il est possible de faire une transformation dont on peut garantir qu'elle est à une garantie constante de l'optimal malléable dans le cadre d'une fonction vitesse concave. En effet, l'exécution d'une tâche par un nombre de processeurs inférieur à 1, dans notre cadre du modèle des tâches malléables, n'a pas de sens. La surface minimum d'une tâche malléable est atteinte pour une exécution sur 1 seul processeur.

Les temps d'exécution d'une tâche sont définis comme suit : $t_{i,q} = t_{i,1}q^{\alpha}$ avec $-1 < \alpha < 0$. L'algorithme est relativement simple. Toute tâche i dont l'allocation $q_i^C \geq 1$ dans la solution continue est exécutée par $\lfloor q_i^C \rfloor$ processeurs. Les tâches dont l'allocation $q_i^C < 1$ sont allouées sur 1 processeur. L'ordonnancement consiste à commencer à la date 0 toutes les tâches utilisant un nombre de processeurs supérieur à 1, puis à effectuer un algorithme d'ordonnancement au plus tôt

des tâches séquentielles.

Algorithm 1 Algorithme de production d'un ordonnancement de tâches malléables indépendantes à partir d'une solution pour les tâches continues indépendantes

for $i \in V$ **do**
  $q_i^M = \lfloor q_i^C \rfloor$
  if $q_i^C < 1$ **then**
    $q_i^M = 1$
  end if
end for
for $i \in V / q_i^M > 1$ **do**
  $Date(i) = 0$
end for
for $i \in V / q_i^M == 1$ **do**
  $Date(i) = DateAuPlusTot()$
end for

Théorème 5 *L'algorithme 1 produit un ordonnancement malléable ayant une garantie de performance de 3.*

Preuve :
Soit ω_c, le makespan de la solution au problème des tâches continues. La solution continue consiste à exécuter toutes les tâches en même temps sur une fraction des processeurs afin qu'elles terminent toutes à la date ω_c, en même temps. La solution continue réalise le compromis optimal entre la surface de travail totale et la longueur des tâches. La valeur de la solution continue est donc une borne inférieure de ω_m, le makespan de la solution malléable.

Toutes les tâches dont l'allocation continue $q_i^C \geq 1$ voient leur allocation diminuer d'un facteur inférieur à $1/2$ (le passage de $2-\epsilon$ à 1). Puisque le travail est croissant, la durée de leur exécution est donc augmentée d'un facteur inférieur à 2. Par contre, leur surface est diminuée. De plus la somme des processeurs alloués à ces tâches est plus petite que m dans l'allocation continue et donc cette somme est plus petite que m après la transformation.

Les tâches dont l'allocation $q_i^C < 1$ passent à 1, voient leur temps d'exécution diminué mais leur surface augmentée. Par contre cette surface est la surface minimum que peuvent avoir ces tâches dans un ordonnancement malléable quelconque. Ces tâches-là ont donc une surface plus petite que leur surface dans l'ordonnancement malléable.

Les tâches de durée comprise entre ω_c et $2\ \omega_c$ peuvent être exécutées sur moins de m processeurs à la date 0. Les autres tâches ont une durée inférieure à ω_c et leur surface totale est inférieure à $m\ \omega_m^*$.

La somme des surfaces des tâches W est inférieure à $m\ \omega_c + m\ \omega_m^*$. Une analyse semblable à celle de Graham [60] peut s'appliquer : la dernière tâche placée débute à un instant où tous les processeurs sont occupés (sinon elle aurait pu débuter plus tôt). Donc le temps d'exécution est plus petit que $max(2\ \omega_c, \frac{m\ \omega_c + m\ \omega_m^*}{m} + \omega_c) = 2\ \omega_c + \omega_m^*$

Or puisque $\omega_c \leq \omega_m$, on obtient donc pour ω_m, le makespan de l'ordonnancement transformé :

$$\omega_m \leq \omega_m^* + 2\ \omega_c$$

.

□

Cette première approche permet d'obtenir une garantie de performance de 3. Mais en modifiant très légèrement l'algorithme, une garantie de 2 est facilement atteignable.

Amélioration de la garantie

En effet, la garantie indique que

$$\omega_m \leq \omega_m^* + 2\omega_c$$

donc si $\omega_m^* \geq 2\omega_c$ on obtient directement une garantie de 2. Le point délicat à traiter est donc le cas où $\omega_m^* \leq 2\ \omega_c$.

La modification de l'algorithme est la suivante : les tâches initialement allouées sur plus de 1 processeur voient leur allocation réduite jusqu'à obtenir un temps d'exécution égal à $2\ \omega_c$. Puisque $2\ \omega_c$ est plus grand que ω_m^*, l'allocation choisie pour ces tâches est plus petite que l'allocation de ces tâches dans l'ordonnancement optimal malléable.

La somme des travaux de toutes les tâches est alors plus petite que celle de l'optimal est donc on obtient la borne suivante

$$\omega_m \leq \frac{m\ \omega_m^*}{m} + \omega_c$$

Avec cette modification nous obtenons bien alors le fait que

$$\omega_m \leq 2\ \omega_m^*$$

.

Si la solution continue peut-être obtenue en temps raisonnable, cet algorithme est peu coûteux. Mais, pour obtenir de meilleures garanties que 2, nous devons tenir compte plus finement du fait du caractère discrêt du problème.

Chapitre 3

Ordonnancement de tâches malléables indépendantes

Contents

Dans ce chapitre, nous nous intéressons à l'ordonnancement de n tâches malléables indépendantes, c'est-à-dire sans relation de précédences, sur m processeurs. Nous approfondissons certains résultats précédents concernant le problème de l'ordonnancement de tâches malléables indépendantes. Dans un deuxième temps, nous présentons notre apport sur le sujet. Nous postulons que les algorithmes efficaces se divisent essentiellement en les deux grandes catégories présentées. Nous abordons dans cette deuxième partie un ensemble d'algorithmes où l'allocation des tâches malléables est choisie de manière à obtenir un placement multiprocesseurs simple. Nous avons précédemment utilisé ces stratégies avec succès dans le cadre de l'ordonnancement de tâches malléables indépendantes monotones [90]. Nous proposons ici un nouvel algorithme permettant d'obtenir une garantie de $3/2 + \epsilon$ pour les tâches malléables indépendantes avec une complexité $O(n\ m)$. C'est le meilleur résultat avec une complexité raisonnable que nous connaissons.

3.1 Précédents résultats

Nous nous intéressons à des algorithmes d'ordonnancement de tâches malléables indépendantes, utilisables en pratique, c'est-à-dire de complexité raisonnable.

Jansen et Porkolab [75] ont proposé un schéma d'algorithmes polynômiaux d'approximation pour l'ordonnancement de tâches malléables indépendantes. Cet algorithme ne pose pas de contrainte sur le facteur d'inefficacité des tâches. Il est donc possible d'obtenir une solution avec une garantie de performance de $(1 + \epsilon)$ avec ϵ aussi petit que l'on veut. Malheureusement, ce schéma n'est pas un schéma d'approximation pleinement polynômial, c'est-à-dire que la complexité de l'algorithme n'est pas polynômiale en $\frac{1}{\epsilon}$, ni en m non plus d'ailleurs. Ce qui signifie qu'en pratique, cet algorithme est coûteux et non utilisable dans un contexte d'implantation d'une application parallèle.

Le problème général de l'ordonnancement de tâches indépendantes a été montré NP-Difficile au sens fort à partir de 5 processeurs [36]. Du coût, il est impossible de trouver un schéma d'approximation pleinement polynômial.

3.1.1 Étape 1 : choix d'une allocation

Turek, Wolf et Yu [110] ont proposé une méthode polynômiale, pour le cas sans restriction sur le facteur d'inefficacité, pour le choix de l'allocation avec un coût en $O(nmA_{pack})$. A_{pack} est le coût de l'algorithme utilisé pour le placement de tâches multiprocesseurs indépendantes, comme par exemple un algorithme de strip-packing. Turek et al. appliquent l'algorithme de strip-packing à

chaque pas. La complexité de cette méthode a été améliorée par Ludwig [88, 86] en $O(n\ m + A_{pack})$ en utilisant le fait que la garantie de A_{pack} est une fonction de deux paramètres : la longueur de la plus grande tâche et la somme des surfaces des tâches, c'est-à-dire les mêmes paramètres que ceux optimisés par les choix de l'allocation.

L'idée est de générer le placement de nm allocations particulières. La meilleure solution fournie une garantie de performance pour le problème malléable égale à la garantie de A_{pack} pour le problème de placement multiprocesseurs.

L'algorithme part de l'allocation ayant la surface minimum puis alloue plus de processeurs à la tâche de plus grand temps d'exécution pour l'accélérer en augmentant le moins possible la surface. L'algorithme est présenté dans une version simplifiée dans l'algorithme 2. La simplification apportée pour les tâches monotones (cf hypothèse 3, page 36), est qu'il est inutile de chercher la "prochaine" allocation sélectionnée de surface minimum de la tâche.

L'idée est que le makespan ω^* de l'allocation $Alloc^*$ du meilleur ordonnancement respecte les deux bornes inférieures de l'ordonnancement malléable que sont la somme des travaux et le chemin critique (la longueur de la plus grande tâche). C'est-à-dire :

$$\exists Alloc^*, \omega^* \geq \frac{W_{Alloc^*}}{m} \text{ et } \omega^* \geq CP_{Alloc^*}$$

.

Algorithm 2 Algorithme simplifié (tâches monotones) de l'allocation de [110]

$\forall t, Alloc(t) = 1$
while $CP_{Alloc} > \frac{W_{Alloc}}{m}$ **do**
  Soit t la tâche courante la plus longue
  Allouer un processeur supplémentaire à t
end while

Initialement un seul processeur est alloué à chaque tâche. Tant que la quantité de travail total à fournir, divisée par le nombre de processeurs, n'est pas plus grande que le temps d'exécution de la plus grande tâche dans l'allocation courante, l'algorithme alloue un processeur de plus à cette tâche.

Lorsque les tâches ne sont pas monotones, l'algorithme général exclut les allocations "gênantes" en se restreignant au sous-ensemble d'allocations monotones, c'est-à-dire en excluant une allocation quand il en existe une autre utilisant moins de processeurs avec un plus petit temps ou utilisant plus de processeurs avec une plus petite surface (cf. discussion sur l'hypothèse 3, page 36).

Lorsque les tâches sont monotones, Ludwig et Tiwari [88] ont montré que la recherche de l'allocation peut se faire en temps dans $O(n \log^2(m))$ en utilisant une dichotomie sur l'intervalle des allocations possibles de chaque tâche. Les détails de l'algorithme complet peuvent être trouvés dans [87, 86].

L'algorithme construit finalement une allocation, où la taille du chemin critique est proche de la somme du travail total divisée par le nombre de processeurs.

Théorème 6 *Si les tâches sont monotones, c'est-à-dire si l'hypothèse 3, page 36, est satisfaite, la somme des travaux des tâches de l'allocation $Alloc$ fournie par l'algorithme 2 est minimum par rapport au chemin critique choisi.*

Preuve :
L'algorithme ajoute des processeurs uniquement aux plus grandes tâches qui définissent le chemin critique. La monotonie des tâches garantit que l'ajout de processeurs augmente le travail. L'allocation de chaque tâche est le minimum permettant que son temps d'exécution soit plus petit que le chemin critique. Toute autre allocation permettant la même garantie sur le chemin critique demanderait plus de processeurs. De par les hypothèses de monotonie, cela augmente donc le travail accompli. □

Corollaire :
La valeur de l'ordonnancement malléable optimal est à un facteur plus petit que 2 de cette borne inférieure.

Cela découle de ce que nous allons présenter maintenant.

3.1.2 Étape 2 : empilement

Turek, Wolf et Yu [110] proposent l'utilisation d'un algorithme de *strip-packing* [3] pour placer les tâches (rectangles) allouées par l'algorithme précédent. Le problème du strip-packing consiste à mettre des rectangles dans une seule boîte (1 bin 2D) ayant une dimension fixée. Le but est de placer tous les rectangles en minimisant l'autre dimension de la boîte.

En fait, ils proposent l'utilisation de deux types d'algorithmes de strip-packing suivant que l'on impose, ou non, une contrainte de contiguïté des processeurs. Nous nous limitons dans ce chapitre aux algorithmes respectant ces contraintes de contiguïté.

La garantie de performance de l'ordonnancement est la garantie de performance de l'algorithme du strip-packing 2D. Toute amélioration dans les algorithmes de packing 2D est directement applicable. La garantie considérée n'est pas le rapport de compétitivité par rapport au placement optimal mais par rapport aux deux bornes inférieures que sont la taille maximale des rectangles (chemin critique) et la somme des surfaces des rectangles (le travail total).

À notre connaissance, l'algorithme de strip-packing 2D ayant la meilleure garantie de performance absolue pour cette application est celui de Steinberg [108]. Il a une garantie de performance de 2 par rapport à la taille de la plus grande tâche et par rapport à la surface totale des tâches.

Mais avant d'analyser l'algorithme de Steinberg, nous pouvons noter qu'il existe d'autres algorithmes pour le strip-packing. En particulier, il existe des schémas d'approximation asymptotique pleinement polynômiaux de strip-packing [78] [43]. C'est-à-dire qu'il existe des algorithmes produisant des placements de hauteur ω, tels que $\omega \leq \omega^*(1+\epsilon)+C$, où C est une constante. Par exemple dans [79], C vaut $\frac{2(2+\epsilon)^2}{\epsilon^2}+1$

Ces algorithmes sont polynômiaux en n, le nombre de rectangles, et en $\frac{1}{\epsilon}$, la qualité de l'approximation. Néanmoins, nous pensons que ces algorithmes ne sont pas forcément les plus adéquats pour notre problème et ce pour deux raisons :

1. De plus la garantie de $(1+\epsilon)$ n'est accessible qu'à une constante près. Dans notre problème, le nombre de tâches n'est pas forcément très grand et donc il est courant que ω^* soit proche de la longueur de la plus grande tâche. Lorsque ω^* est proche de la valeur de C et donc la garantie obtenue est également de l'ordre de 2.
2. Ces algorithmes fournissent un placement avec une garantie de performance qui est une fonction de l'optimal. L'algorithme de Steinberg fournit sa garantie en fonction de critères de bornes inférieures : la taille et la surface des rectangles.

   L'emploi de ces algorithmes impose de tester plusieurs (toutes) allocations possibles afin de trouver celle qui donne le meilleur résultat.

Le principal inconvénient de cette approche est son utilisation du strip-packing. Si le meilleur algorithme connu a une garantie de 2 par rapport aux bornes inférieures, l'ordonnancement malléable obtenu est limité à la même garantie.

L'algorithme de Steinberg [108] n'est pas un algorithme d'ordonnancement, mais un algorithme fondé sur des empilements ((cf fig. 3.1)). En pratique son comportement en moyenne n'est pas forcément adapté. Cet algorithme est récursif. Il applique différentes stratégies de placement suivant la taille de la boîte et les tailles des rectangles en entrée. Chaque stratégie peut être appliquée dans chacune des deux dimensions.

Si un rectangle est plus long que la moitié d'un coté de la boîte, il est placé contre ce côté. L'algorithme continue le placement des autres rectangles dans la plus grande sous boîte rectangulaire privée de la portion occupée par le rectangle placé. Sinon si il existe une tâche, telle que les autres tâches représentent moins du quart de la surface de la boîte, elle est placée contre le bord de sa plus grande

dimension. Enfin, si il existe une paire de rectangles dont chacun des cotés fait plus d'un quart de la dimension de la boîte, ils sont empilés contre le bord.

La force et l'idée originale de l'algorithme sont de traiter efficacement les cas non triviaux, en séparant la liste des rectangles en deux sous listes et en divisant simultanément la boîte en deux. Cela est fait de façon à ce que chacune des sous-listes puisse être placée dans sa sous-boîte en respectant la garantie de performance de 2. La liste est coupée en deux listes, L_1 et L_2 de façon à ce que :

- la surface des m premières tâches, L_1, est plus petite que $3/8$ de la surface de la boîte,
- la surface des tâches restantes, L_2, est plus petite que $1/4$ de la surface de la boîte,
- la taille de la $m+1$-ième tâche (Nous rappelons que les tâches sont triées) est plus petite que $1/4$ de la dimension de la boîte.

Il est alors possible de couper le rectangle en deux parties en fonction de la surface totale de L_1, pour respecter la contrainte de la garantie de 2 pour chacune des nouvelles sous boîte de la boîte initiale.

La récursion sur chacune des sous-listes finit par converger vers les cas triviaux.

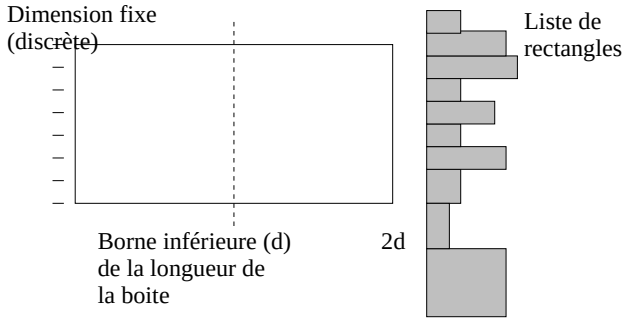


FIG. 3.1 – *Strip packing (Steinberg): boîte et liste de rectangles*

Cette division de l'espace (la boîte) est un problème. Par exemple, Lorsque l'algorithme divise l'espace (cf fig. 3.2), il peut "trancher" un processeur en deux, ce qui, pour notre problème avec une dimension discrète, signifie qu'un processeur reste inutilisé (cf fig. 3.3). Cette division est effectuée lorsqu'un cas trivial n'existe pas. Un empilement de tâches peut alors survenir alors qu'un simple algorithme de liste les aurait placées plus efficacement.

Ces deux défauts influencent le comportement moyen de l'algorithme de Steinberg. L'algorithme choisit en fonction des entrées parmi plusieurs stratégies de placement possible. Nous pensons qu'il est assez proche de son comportement

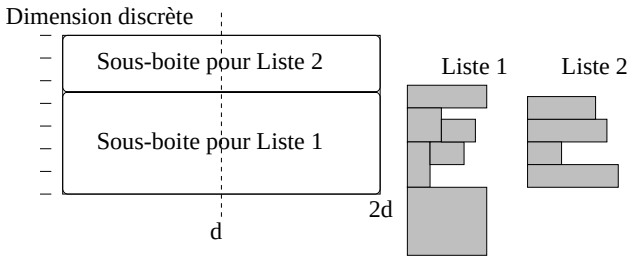


FIG. 3.2 – *Strip packing (Steinberg): coupe en 2 la liste et la boîte*

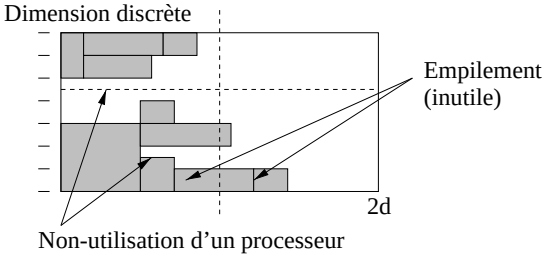


FIG. 3.3 – *Strip packing (Steinberg): défauts de l'algorithme de Steinberg en pratique*

au pire du point de vue de l'ordonnancement produit. Ce point a été confirmé expérimentalement (cf section 3.4).

3.2 Approximation duale

Dans le reste du mémoire, nous allons abondamment utiliser la technique de l'approximation duale. Nous allons illustrer son usage sur le problème déjà présenté du calcul de l'allocation de tâches malléables.

L'approximation duale a été introduite par Shmoys et Hochbaum [68]. Elle consiste à choisir une date λ et à construire des algorithmes utilisant cette date comme si elle était la durée de l'ordonnancement optimal. La connaissance de cette date supposée optimale permet de déduire un certain nombre de propriétés.

En fonction de cette date λ et d'un nombre réel $k \geq 1$, un algorithme de k-approximation duale répond à la question : *Existe-t-il une solution de valeur inférieure ou égale à λ ?*

- soit en répondant correctement *Non*, si il n'en existe pas,
- sinon en fournissant une solution de valeur inférieure à $k\lambda$

On utilise l'algorithme de k-approximation duale pour diriger une dichotomie sur la date λ. La dichotomie permet de trouver un intervalle de temps $[\Lambda..(1+\epsilon)\Lambda]$ tel que l'algorithme répond "Non !" pour la date Λ et produit une solution pour $(1+\epsilon)\Lambda$. La valeur de l'optimal est donc plus grande que λ_1 et la valeur de la solution est plus petite que $k(1+\epsilon)\Lambda$. La solution est donc au plus à un facteur $k(1+\epsilon)$ de la valeur de la solution optimale.

Pour initialiser la dichotomie, il suffit de partir de la borne inférieure pour trouver une solution de valeur ω_0. On peut alors trouver la date Λ après $\lceil \log_2(\frac{1}{\epsilon}) \rceil$ itérations de la dichotomie.

3.2.1 Application à l'ordonnancement de tâches malléables indépendantes

Nous allons illustrer l'approximation duale sur le problème du calcul de l'allocation de tâches malléables indépendantes. L'approximation duale permet de travailler sur des algorithmes en utilisant des arguments de surface ou de longueur de chemin critique par rapport à l'optimal supposé.

Nous construisons maintenant une légère variante de l'algorithme de Turek et al. en utilisant une approximation duale pour un problème de tâches malléables indépendantes monotones.

Choisissons une date λ. Si λ est le makespan de l'ordonnancement optimal, chaque tâche de cet ordonnancement est allouée sur le nombre de processeurs qui lui permet de s'exécuter en moins de λ. Si chaque tâche monotone est allouée sur le plus petit nombre de processeurs possible, le travail de chaque tâche est minimum. Donc la somme des travaux des tâches est minimum.

Si la somme des travaux est plus grande que $m\lambda$, il ne peut exister d'ordonnancement malléable de durée inférieure à λ.

La dichotomie sur la valeur de λ permet de trouver une borne inférieure de l'ordonnancement malléable optimal.

Si nous appliquons l'algorithme de strip-packing de Steinberg de garantie absolue 2 pour ordonnancer ces taches multiprocesseurs de la dernière allocation, nous obtenons une solution de garantie $2+\epsilon$.

3.2.2 Autres travaux sur l'ordonnancement de tâches indépendantes

Nous avons présenté dans la section 3.1 l'algorithme de Turek, Wolf et Yu [110] pour l'ordonnancement de tâches malléables indépendantes. Il consiste à choisir d'abord une allocation proche de la meilleure allocation possible du point de vue du travail et du chemin critique du graphe généré. Ensuite, un algorithme

de strip-packing est utilisé pour placer les tâches. Mais comme nous l'avons vu, le problème est que la garantie de performance est limitée par celle de l'algorithme de placement. Pour les tâches indépendantes, le meilleur algorithme de strip-packing 2d, à notre connaissance, a une garantie de performance de 2.

Nous allons maintenant aborder une autre classe d'algorithmes dont l'approche est différente. Ces algorithmes essayent de trouver une allocation dont l'ordonnancement est *simple*. De cette façon, leur garantie est fonction de la garantie obtenue par le choix et la structure de l'allocation et non plus par celui de l'ordonnancement multiprocesseurs.

Nous montrons d'abord les idées derrière un algorithme que nous avons conçu avec Christophe Rapine de Denis Trystram [90]. Il s'appuie sur les principes de l'approximation duale présentée dans la section 3.2. Les démonstrations complètes figurent dans la thèse de Christophe Rapine[100]. Nous ne présentons ici que le principe.

Supposons que la limite λ de l'approximation duale soit la durée d'exécution de l'ordonnancement réalisable. Cela signifie que toutes les tâches malléables sont exécutées en un temps plus petit que λ.

Définition 10 *L'allocation minimale de chaque tâche malléable afin qu'elle s'exécute en temps inférieur à λ est son* allocation canonique *(cf fig. 3.4).*

Définition 11 *La surface occupée par les tâches à leur allocation canonique est la* surface canonique.

La surface canonique est plus petite que $\lambda\ m$.

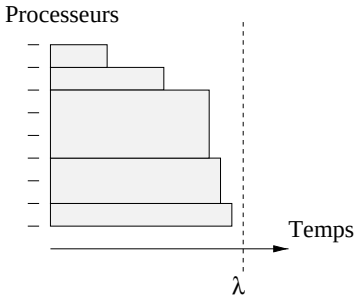


FIG. 3.4 – *Surface canonique*

Observons l'allocation canonique. Soit S_m (cf fig. 3.5), la quantité de travail effectué par les tâches de plus grand temps d'exécution sur m processeurs. En utilisant des arguments de surface, il est possible de prouver que la généralisation de l'algorithme *Plus Grande Tâche d'abord* [61] (LPTF, Largest processing Time

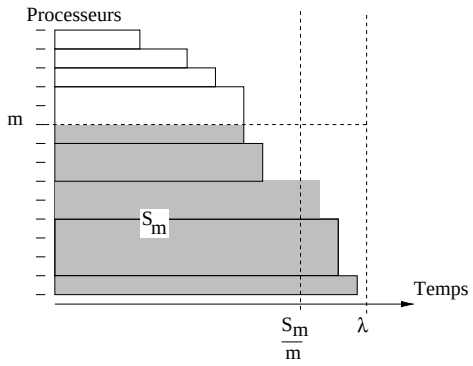


FIG. 3.5 – S_m

First) aux tâches malléables place les tâches dans leur allocation canonique en un temps plus petit que $2\ S_m/m$ (cf fig. 3.5).

Lorsque la surface S_m est petite par rapport à $\lambda\ m$, cet algorithme fourni une solution de bonne qualité. Lorsque la surface S_m devient grande, il est possible de résoudre deux problèmes sac-à-dos entiers, essayant de placer les tâches dans deux étagères (de hauteur λ et β) (cf fig. 3.6), trouvent une solution réalisable.

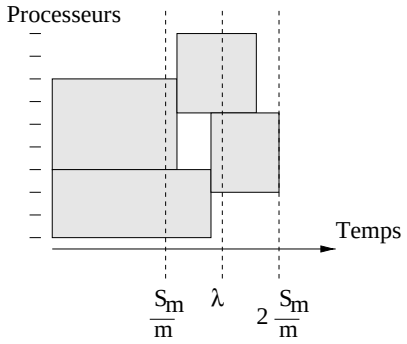


FIG. 3.6 – *LPTF sur l'allocation canonique*

Il y a un compromis à trouver entre S_m et β pour choisir le meilleur des deux algorithmes, LPTF ou 2 Étagères. Dans [90] ce compromis a été montré égal à $\sqrt{3}$, d'où une garantie de $\sqrt{3}+\epsilon$. Cela améliore la borne de 2, obtenue par Ludwig [86] en appliquant l'algorithme de strip-packing de Steinberg [108] à la stratégie d'allocation de Turek, Wolf et Yu [110]. Ce premier succès nous a conduits à étudier d'autres structures d'ordonnancement pour atteindre une meilleure garantie.

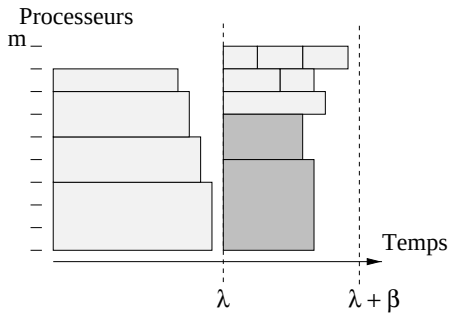


FIG. 3.7 – *ordonnancement en deux étagères de λ et β.*

3.3 Ordonnancement de tâches indépendantes en deux étagères

Dans cette partie nous allons présenter un nouvel algorithme d'ordonnancement de tâches malléables indépendantes monotones. Sa garantie de performance est de $3/2 + \epsilon$ seulement.

Nous allons utiliser une approximation duale. La limite λ sera le temps d'exécution de l'optimal supposé. Pour simplifier les notations, sans perte de généralité, nous normalisons l'instance afin que λ soit égal à 1.

L'idée est de séparer l'ensemble des tâches malléables en trois sous-ensembles : les *grandes*, les *petites* et les *petites séquentielles*. À partir de cette partition nous montrons comment ordonnancer ces différentes classes de tâches pour obtenir une solution réalisable.

3.3.1 Structure de l'ordonnancement optimal

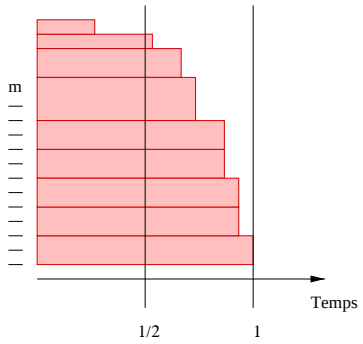


FIG. 3.8 – *Allocation canonique*

La limite 1, (λ), du makespan supposé de l'optimal nous fournit des indications sur l'allocation des tâches. D'abord le temps d'exécution de chaque tâche est plus petit que 1 (cf fig. 3.8). Ensuite la somme des travaux est plus petite que m, ($\lambda\ m$). La succession de deux tâches dans l'ordonnancement impose que l'une des deux tâches soit plus petite que 1/2. Autrement dit, il ne peut y avoir dans l'ordonnancement optimal plus d'une tâche affectée à un processeur de temps strictement supérieur à 1/2.

Formellement, dans l'ordonnancement optimal, les tâches s'exécutant en un temps compris dans l'intervalle $]1/2..1]$, n'utilisent pas plus de m processeurs au total. Et donc toutes les autres s'exécutent en temps $]0..1/2]$.

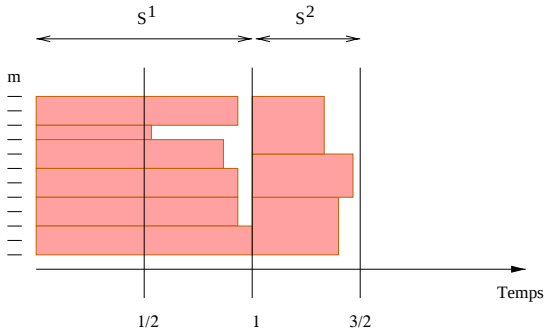


FIG. 3.9 – *Deux étagères : 1 + 1/2*

Nous allons donc rechercher sur un ordonnancement en deux étagères (cf fig. 3.9) : la première de hauteur 1 (notée S^1), la seconde de hauteur 1/2 (notée S^2).

3.3.2 Partition des tâches et monotonie

La première étape consiste à séparer l'ensemble de tâches en ces deux catégories : les taches s'exécutant en $]1/2..1]$ dans S^1 et celle s'exécutant en $]0..1/2]$ dans S^2. Nous les nommerons simplement dans la suite les tâches de S^1 et les tâches de S^2.

La monotonie des tâches (cf hypothèse 3, page 36) fournit des bornes sur les variations de temps d'exécution et de travail des tâches. En effet, ces variations ne peuvent être plus importantes que les variations de processeurs choisis pour l'allocation.

Observons les informations apportées par la monotonie pour le calcul de l'allocation canonique. Nous rappelons que l'allocation canonique d'une tâche est le plus petit nombre de processeurs qui doivent lui être alloués pour qu'elle s'exécute en temps plus petit que 1. Il y a deux cas à distinguer suivant que cette allocation est l'allocation séquentielle ou que la tâche s'exécute sur $q > 1$ processeurs. Dans

ce dernier cas, par définition, le temps d'exécution de la tâche sur $q-1$ processeurs est plus grand que 1. La monotonie indique que le rapport des temps ne peut être plus grand que le rapport des processeurs $\frac{q-1}{q}$ et donc, chaque tâche a un temps d'exécution plus grand que $1 \times \frac{q-1}{q}$. De la même manière, toutes les tâches s'exécutant sur plusieurs processeurs dans S^2 ont un temps d'exécution compris dans l'intervalle $]1/4..1/2]$.

Cette relation indique donc que toutes les tâches qui s'exécutent dans $]0..1/2]$, avec une allocation canonique, sont séquentielles. Ces tâches séquentielles ne gênent pas l'algorithme et nous précisons dans la section 3.3.3 comment les insérer entre les deux étagères sans perturber l'ordonnancement.

Définition 12 *W_{seq} est la somme des surfaces des tâches s'exécutant sur un processeur en temps $]0..1/2]$ dans leurs allocations canoniques.*

3.3.3 Insertion des petites tâches séquentielles

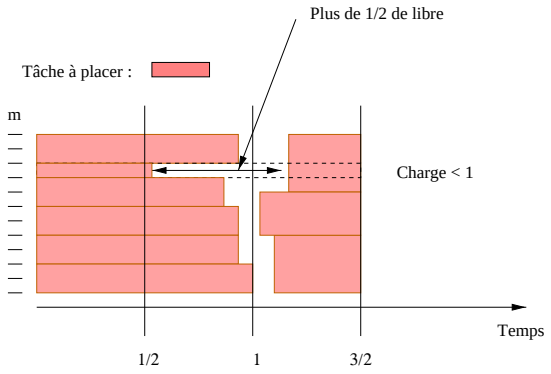


FIG. 3.10 – *Insertion des petites tâches séquentielles*

Si l'on a construit une allocation répartie en deux étagères, telle que sa surface soit plus petite que $m - W_{seq}$, il est toujours possible d'insérer une tâche séquentielle s'exécutant en temps $]0..1/2]$ en conservant un makespan inférieur à $3/2$. En effet, si la surface de toutes les tâches est plus petite que m, alors l'un des processeurs exécute des calculs pendant moins de 1, c'est-à-dire que la somme des temps des tâches que ce processeur exécute est plus petite que 1. Si les tâches de l'étagère S^1 commencent à l'instant 0 et que les tâches de l'étagère S^2 finissent à l'instant 3/2 alors, il y a entre les tâches des deux étagères de ce processeur plus de 1/2 temps d'inactivité et donc une tâche séquentielle de temps d'exécution $]0..1/2]$ peut être exécutée par ce processeur entre les deux étagères (cf fig. 3.10).

La condition initiale, $m - W_{seq}$, reste vraie au fur et à mesure que les tâches séquentielles sont retirées de W_{seq} et insérées, sinon la surface totale serait plus grande que m. L'insertion de toutes les tâches de W_{seq} est donc possible.

Dans la suite, les petites tâches séquentielles sont retirées des listes de tâches et, sans perte de généralité, ne sont pas toujours évoquées dans les analyses. Par exemple, elles sont implicitement comptabilisées dans les sommes de travaux.

3.3.4 Séparation en deux classes de tâches

Après la mise à l'écart des petites tâches séquentielles, nous allons résoudre un sac-à-dos entier de petite dimension par programmation dynamique, pour séparer les tâches restantes en deux classes : les tâches allouées respectivement à S^1 et S^2.

En plus de l'allocation canonique, c'est-à-dire l'allocation telle que chaque tâche s'exécute en temps plus petit que 1, nous utilisons aussi pour chaque tâche son allocation telle que son temps d'exécution soit inférieure à $1/2$. S'il n'existe pas une telle allocation, la surface de travail de cette tâche est considérée comme infinie dans ce cadre.

Le problème est équivalent à un problème de sac-à-dos où la valeur d'une tâche va être sa surface, suivant qu'elle est placée dans l'étagère S^1 ou S^2. Le poids d'une tâche correspond au nombre de processeurs qu'elle utilise dans S^1, 0 si elle est placée dans S^2. La contrainte porte sur le nombre total de processeurs disponibles, m, pour exécuter les tâches de S^1.

Nous allons utiliser la fonction $\Gamma(i, d)$ qui fournit pour la tâche i, le nombre minimum de processeurs lui permettant de s'exécuter en temps inférieur à d. S'il n'existe pas d'allocation inférieure à m pour la tâche i afin qu'elle s'exécute en temps inférieur à d, $\Gamma(i, d)$ vaut $+\infty$. W_{seq} est la somme des travaux des petites tâches séquentielles mises à l'écart.

Le but est de minimiser la surface notée W^*. Formellement :

$$W^* = \min_{S^1, S^2} \left(\sum_{i \in S^1} W_{i,\Gamma(i,1)} + \sum_{i \in S^2} W_{i,\Gamma(i,1/2)} + W_{seq} \right)$$

sous la contrainte

$$\sum_{i \in S^1} \Gamma(i, 1) \leq m$$

L'équation de récurrence de la programmation dynamique est :

$$\bar{W}(i, q) = min \left(\begin{array}{l} \bar{W}(i-1, q) + W_{i,\Gamma(i,1/2)}, \\ \bar{W}(i-1, q - \Gamma(i, 1)) + W_{i,\Gamma(i,1)} \end{array} \right)$$

où i est le numéro de la ième tâche et q le nombre de processeurs disponibles dans S^1. Comme le problème est en nombre entier, l'algorithme a un temps d'exécution de $O(n\ m)$

Par construction, et comme nous l'avons expliqué dans la section 3.3.1, si $W^* > m$ alors il n'existe pas de solution ayant un makespan inférieur ou égal à 1 et l'algorithme répond "Non !" à l'approximation duale. Sinon nous allons pouvoir construire un ordonnancement avec un temps d'exécution inférieur à $3/2$.

La surface W^* est strictement plus petite que la surface de travail de l'ordonnancement optimal de l'ensemble des tâches.

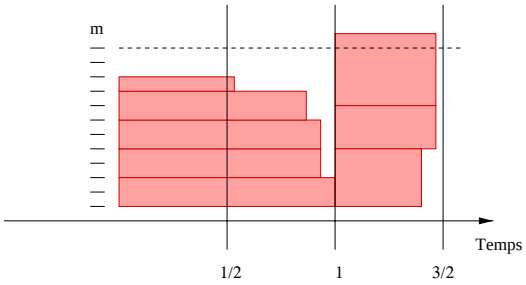


FIG. 3.11 – *Sac à dos partitionnant l'ensemble des tâches*

3.3.5 Transformation de l'ordonnancement

L'algorithme précédent construit une solution réalisable si les tâches placées dans S^2 utilisent au total moins de m processeurs. En effet par construction, les tâches placées dans S^1 respectent cette contrainte et nous avons déjà montré qu'il était toujours possible d'insérer les petites tâches séquentielles entre les deux étagères.

Lorsque la solution fournit au problème d'allocation ne permet pas de construire directement un ordonnancement réalisable, c'est que la somme des processeurs utilisés dans S^2 est supérieure à m. Nous allons transformer cette allocation. Pour cela nous allons utiliser les propriétés de monotonie des tâches. En particulier nous allons utiliser la proposition suivante.

Propriété 2 *Les tâches exécutées dans S^2 ont un temps d'exécution compris dans l'intervalle $]1/4..1/2]$*

Preuve :
Les petites tâches séquentielles de durée inférieure à $1/2$ ont été mises à part et ne sont pas placées dans l'étagère. Les tâches placées ont donc un temps d'exécution canonique plus grand que $1/2$ et sont donc exécutées par deux processeurs au moins. De par les hypothèses de monotonie, cela induit que leur temps d'exécution est plus grand que $\frac{1}{2}\frac{1}{2}$, c'est-à-dire $1/4$.

□

L'idée, pour construire l'ordonnancement réalisable, est d'utiliser le fait que chaque processeur exécutant une tâche de S^2, effectue au moins $1/4$ de travail. Si S^1 est occupée au moins jusqu'à $3/4$, ce processeur effectue plus de 1 en travail. Or puisque la surface totale est plus petite que m, Il n'y a pas plus de m processeurs utilisés par étagère. Pour garantir l'occupation d'un processeur au-delà de 1, nous renonçons, sur certains processeurs, à trouver une solution en deux étagères.

Pour cela nous allons utiliser un certain nombre de transformations. Il est important de noter que chacune d'elles diminue le travail des tâches sur lesquelles elle s'applique et donc maintient une surface totale de travail plus petite que la surface de l'ordonnancement optimal.

Empiler des tâches de S^1

Lorsque deux tâches séquentielles de S^1 sont de durée inférieure à $3/4$, il est possible de les empiler sur un seul processeur (cf fig. 3.12) en moins que $3/2$.

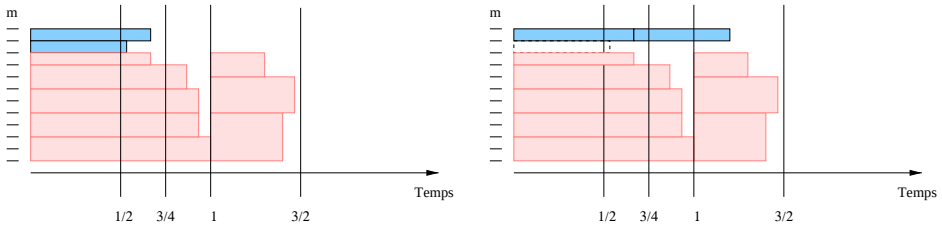


FIG. 3.12 – *Empilement de deux tâches séquentielles inférieures à* $3/4$

Cet empilement occupe un processeur pendant au moins 1 puisque les tâches sont plus grandes que $1/2$.

Diminuer l'allocation de tâches de S^1

Les tâches de S^1 sont allouées à leur allocation canonique. Cela signifie que, si elles ne sont pas séquentielles, le fait de diminuer leur allocation d'un processeur, induit un temps d'exécution de ces tâches plus grand que 1. D'un autre coté, si ces tâches s'exécutent en temps inférieur à $3/4$ cela signifie que diminuer leur allocation n'induit pas un temps d'exécution plus grand que $\frac{3}{4}\frac{2}{1}$ c'est-à-dire $3/2$ (cf fig. 3.13). Les processeurs exécutant cette tâche sont utilisés au-delà de 1

Ces deux transformations peuvent s'appliquer à toutes les tâches de S^1 de durée inférieure à $3/4$. Il ne peut donc rester au plus qu'une seule tâche séquentielle de durée inférieure à $3/4$ que l'on n'a pu empiler.

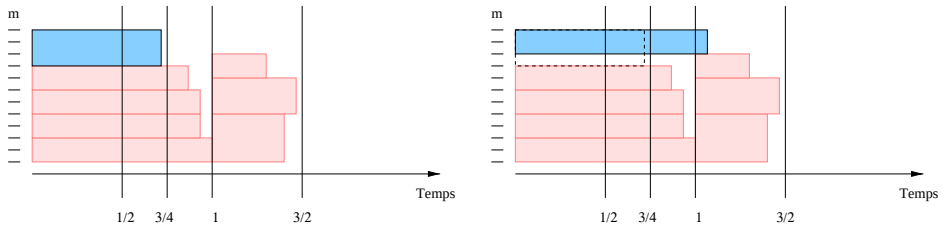


FIG. 3.13 – *Diminuer l'allocation de tâches parallèles inférieures à 3/4*

Replacer une tâche de S^2 dans S^1

Les deux transformations précédentes diminuent le nombre de processeurs utilisés dans S^1. Soit q, le nombre de processeurs libres dans l'étagère S^1, il est possible qu'une des tâches de S^2 puisse être replacée, dans son allocation canonique de S^1 (cf fig. 3.14). Une des deux précédentes transformations pourrait éventuellement être utilisée sur cette tâche, si son temps d'exécution canonique est plus petit que $3/4$.

Il est important de noter que cette transformation respecte la contrainte que moins de m processeurs soient utilisés dans S^2.

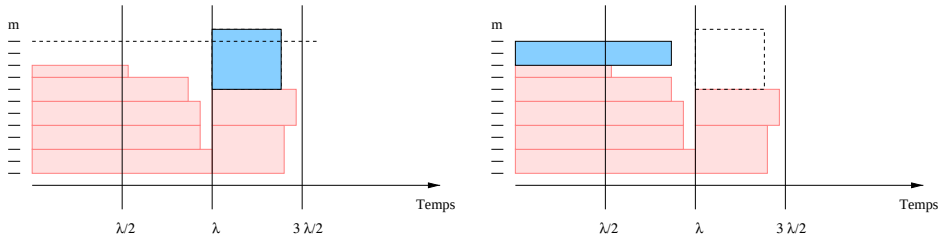


FIG. 3.14 – *Replacer une tâche de S^2 dans S^1*

Il est aussi possible qu'une des tâches de S^2 s'exécute en temps supérieur à 1 mais inférieur à $3/2$ sur moins de q processeurs (cf fig. 3.15). Après transformation, dans ce cas-là, les processeurs sont occupés à plus de 1 par une seule tâche.

Une dernière transformation

Si

– aucune des transformations précédentes n'est possible

– l'ordonnancement n'est toujours pas réalisable

– $q > 0$

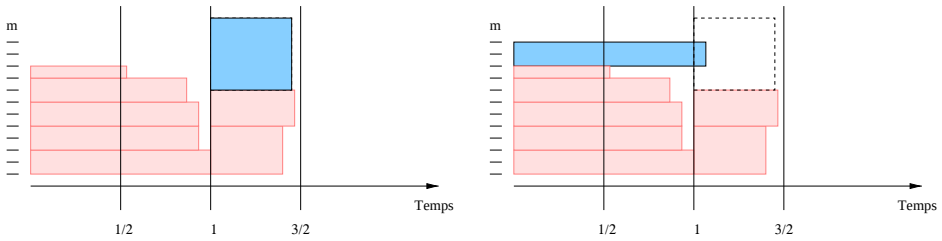


FIG. 3.15 – *Replacer une tâche de S^2 en $]1..3/2]$*

– il existe une tâche séquentielle a de durée inférieure à $3/4$ dans S^1,

nous pouvons appliquer la transformation suivante (cf fig. 3.16).

a est empilé sur une autre tâche de S^1 si la somme de leur temps d'exécution est inférieure à $3/2$. Nous montrerons par la suite que si cette transformation est possible, alors, l'une des tâches de S^2 peut être maintenant ordonnancée en temps inférieur à $3/2$ sur les $q+1$ processeurs libres. L'ordonnancement devient alors réalisable.

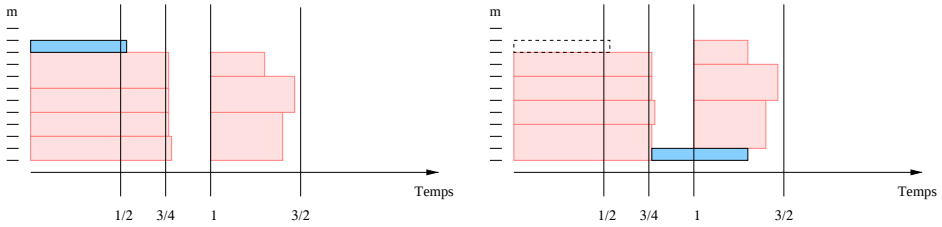


FIG. 3.16 – *Empilement de la dernière tâche de durée inférieure à 3/4*

3.3.6 L'algorithme

Une version simplifiée de l'algorithme est présentée par l'algorithme 3.

La complexité du sac à dos est de $O(n\ m)$.

Chaque tâche ne peut être déplacée ou redimensionnée plus de deux fois par les transformations de la boucle **while**. Une des difficultées est de trouver une tâche sur laquelle s'applique une des 4 transformations en temps $O(m)$. En effet il suffit pour cela de trier les tâches en $3m$ listes, suivant le nombre de processeurs qu'elles utilisent pour les temps d'exécutions inférieurs à 3/2, 1 et 1/2. Ces 3 m-partitions se font en temps $O(n)$. Les tâches entrant dans un état d'allocation stable sont éliminées des listes. La boucle s'effectue donc $O(n)$ fois, au pire, une opération de coût $O(m)$.

Algorithm 3 Algorithme d'ordonnancement en deux étagères de tâches malléables indépendantes. Il renvoie vrai, si il est possible d'ordonnancer l'ensemble des tâches de T en temps 1 sur m processeurs, faux sinon

boolean **Function** DeuxEtageres(T, m)
T_{seq} = Extraction des tâches séquentielles inférieures à 1/2
$T = T \backslash T_{seq}$
$T1, T2 = Sac\_a\_dos(T, m)$ {Partition de T}
if SurfaceTotale($T1, T2, T_{seq}$) >m **then**
  return faux
end if
while Ordonnancement_Non_Realisable() **do**
  if Une des trois premières transformations est possible **then**
    Appliquer une des 3 premières transformations {Dans un ordre quelconque}
  else
    Appliquer la dernière transformation
  end if
end while
Insertion des tâches de T_{seq}
return true

Le processeur le moins chargé peut être trouvé en temps $O(m)$. L'insertion des petites tâches séquentielles est donc au pire en $O(n\ m)$.

Le coût de l'algorithme de placement est donc en $O(n\ m)$.

3.3.7 Solution réalisable

Les transformations précédentes sont appliquées jusqu'à ce que la solution devienne un ordonnancement réalisable.

L'application des transformations précédentes construit une solution qui a la forme suivante (cf fig. 3.17) :

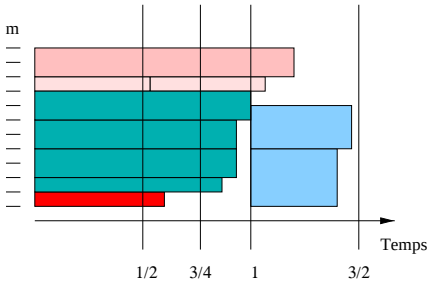


FIG. 3.17 – *Forme de la solution réalisable*

Un certain nombre de processeurs sont occupés à plus que 1 en exécutant une seule tâche ou un empilement des deux tâches séquentielles. Les autres processeurs exécutent un ordonnancement en deux étagères. S^1 exécute des tâches dont le temps d'exécution est compris dans $]3/4..1]$ sauf (peut-être) pour une seule tâche séquentielle de durée $]1/2..3/4]$. Il reste aussi éventuellement des processeurs inutilisés pendant l'exécution de S^1. Nous rappelons que nous notons q le nombre de processeurs inutilisés de S^1. S^2 est occupée par des tâches s'exécutant en $]1/4..1/2]$. Nous allons maintenant montrer que, lorsque les transformations ne peuvent plus être appliquées, la solution produite est un ordonnancement réalisable.

Nous allons utiliser pour cela une preuve par contradiction et supposer que S^2 "déborde". Pour cela nous allons évaluer finement la surface totale de travail en utilisant les informations fournies par la structure de l'ordonnancement.

En fait les processus occupés à plus de 1 ne nous gênent pas. Par souci de clarté et sans perte de généralité, nous ne présentons pas la prise en compte de ces processeurs dans les analyses suivantes. En effet, nous n'utilisons que des arguments de surface. La restriction de l'analyse aux seuls processeurs occupés à moins de 1 est donc suffisante.

Deux cas se présentent suivant qu'il y a ou non des processeurs libres dans S^1.

Le cas où $q = 0$

Dans ce cas, il n'y a pas de processeur libre dans S^1. Tous les processeurs, sauf un, sont occupés strictement à plus de 1, en exécutant soit une seule tâche, soit un empilement de deux tâches séquentielles, soit en exécutant les deux étagères. Dans ce dernier cas, ils exécutent des tâches de temps d'exécution supérieur à $3/4$ dans S^1 et supérieur à $1/4$ dans S^2.

Un seul processeur, au plus, exécute dans S^1 une tâche séquentielle de durée comprise dans l'intervalle $]1/2..3/4[$. Il est donc strictement occupé strictement plus de $1/2 + 1/4$, donc strictement plus de $3/4$.

La surface totale occupée est donc strictement supérieure à $m - 1/4$. Pour que S^2 déborde, il faut occuper au moins un processeur supplémentaire sur cette étagère. Pour occuper un processeur supplémentaire, il faut au moins $1/4$ de travail supplémentaire puisque chaque tâche de S^2 s'exécute en temps strictement supérieur à $1/4$.

Il est donc impossible que S^2 déborde sans que la surface totale soit *strictement* plus grande que $m - 1/4 + 1/4$. Par contradiction puisque le sac-à-dos et les transformations garantissent que cette surface est plus petite que m (cf fig. 3.18), l'ordonnancement est réalisable.

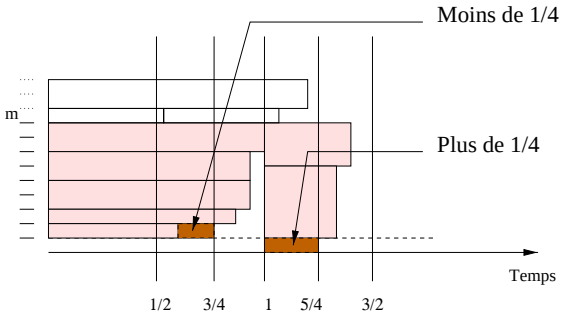


FIG. 3.18 – *Aucun processeur libre dans S^1*

Le cas où $q > 0$

S'il y a q processeurs libres dans S^1 (cf fig. 3.19), une information supplémentaire est disponible. En effet, cela signifie qu'aucune tâche de S^2 ne tient en temps $]1/2..3/2]$ sur q processeurs, sinon la troisième transformation aurait replacé cette tâche dans S^1.

Les hypothèses de monotonie impliquent que toutes les tâches de S^2 sont exécutées par, strictement, plus de $3\ q$ processeurs donc au moins $3\ q + 1$ processeurs.

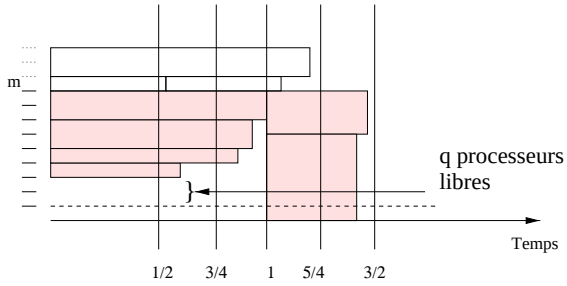


FIG. 3.19 – *Il y a plusieurs processeurs libre dans* S^1

Sinon en diminuant l'allocation de ces tâches à q processeurs, leur temps d'exécution ne pourrait être multiplié par plus de trois, de moins de $1/2$ à plus de $3/2$.

Les hypothèses de monotonie indiquent aussi que le temps d'exécution d'une tâche dans S^2 est au moins $\frac{1}{2}\frac{3\ q}{3\ q+1}$, sinon elle pourrait être exécutée en temps inférieur à $1/2$ sur $3\ q$ processeurs, ce qui serait contradictoire avec le choix fait pour l'allocation de cette tâche.

Deux cas se présentent suivant que S^1 est occupé plus de $\frac{3}{4}(m-q)$ ou non. C'est pour la gestion de ce dernier cas que nous avons introduit la dernière transformation.

S^1 est occupé plus de $\frac{3}{4}(m-q)$ Nous pouvons maintenant exprimer une borne inférieure de somme totale du travail effectué par les tâches. Cette borne doit être plus petite que m. Pour cela nous allons utiliser à nouveau la fonction $\Gamma(i,d)$ qui fournit l'allocation minimum de la tâche i, telle que son temps d'exécution soit plus petit que d.

$$m > \frac{3}{4}(m-q) + \sum_{i \in S^2} W_{i,\Gamma(i,1/2)} \tag{3.1}$$

Il y a au moins $m+1$ processeurs qui sont occupés dans S^2. La somme des surfaces des tâches de S^2 peut s'exprimer de la manière suivante :

$$\sum_{i \in S^2} W_{i,\Gamma(i,1/2)} \geq \sum_{i \in S^2} \frac{1}{2} \frac{\Gamma(i,1/2)-1}{\Gamma(i,1/2)}\ \Gamma(i,1/2)$$

donc

$$\sum_{i \in S^2} W_{i,\Gamma(i,1/2)} \geq \sum_{i \in S^2} \frac{1}{2}\Gamma(i,1/2) - \sum_{i \in S^2} \frac{1}{2}$$

Notons k, le nombre de tâches placées dans S^2. Puisqu'il y a au moins $m+1$ processeurs occupés dans S^2, on obtient :

$$\sum_{i \in S^2} W_{i,\Gamma(i,1/2)} \geq \frac{1}{2}(m+1) - \frac{1}{2}k \tag{3.2}$$

La somme des surfaces des tâches de S^2 peut aussi s'exprimer d'une autre manière. Chacune des k tâches de S^2 ne peut être placée en temps inférieur à $3/2$ sur les q processeurs libres. La surface de chaque tâche de S^2 est donc supérieure à $\frac{3}{2}q$. On a donc une deuxième relation :

$$\sum_{i \in S^2} W_{i,\Gamma(i,1/2)} \geq \frac{3}{2}qk \tag{3.3}$$

En combinant les équations 3.1 et 3.2, on obtient :

$$\frac{3}{4}(m-q) + \frac{1}{2}(m+1) - \frac{1}{2}k < m$$

ou

$$\frac{1}{4}m + \frac{1}{2} < \frac{1}{2}k + \frac{3}{4}q$$

c'est-à-dire

$$m < 3q + 2k - 2 \tag{3.4}$$

En combinant les équations 3.1 et 3.3, on obtient :

$$\frac{3}{4}(m-q) + \frac{3}{2}qk < m$$

ou

$$\frac{3}{4}q(2k-1) < \frac{1}{4}m$$

c'est-à-dire

$$3q(2k-1) < m \tag{3.5}$$

On tire des deux résultats (3.5 et 3.4) la relation suivante :

$$3q(2k-1) < 3q + 2k - 2$$

c'est-à-dire

$$3q(2k-2) < (2k-2)$$

Cela implique $q = 0$. Il y a donc contradiction avec l'hypothèse courante $q > 0$. Il n'était donc pas possible que S^2 déborde.

Application réussie de la dernière transformation Si la dernière transformation a été appliquée avec succès, les processeurs occupés de S^1 le sont à plus de $3/4$ et donc la démonstration du cas précédent s'applique.

S^1 est occupé moins de $3/4(m-q)$ Pour que S^1 soit occupé moins de $3/4(m-q)$ et que l'ordonnancement ne soit toujours pas réalisable, il faut que S^1 contienne encore un processeur occupé par une tâche séquentielle de durée inférieure à $3/4$. Donc la dernière transformation a échoué.

On peut d'abord montrer que $m > q+1$. En effet, si $m = q$ alors la somme des surfaces des tâches de S^2 est inférieure à q. Donc les tâches de S^2 ont un temps d'exécution, sur q processeurs, inférieur à 1, on aurait donc descendu une des tâches de S^2 dans S^1. Si $m = q+1$, il n'y a qu'une seule tâche de durée $]1/2..3/4[$ dans S^1. Les tâches de S^2 (il y en a au moins une) ne peuvent pas s'exécuter en temps inférieur à $3/2$ sur q processeurs. La relation sur la somme des surfaces des tâches est donc :

$$\frac{1}{2} + \frac{3}{2}q < q+1$$

ce qui mène à :

$$\frac{1}{2}q < \frac{1}{2}$$

et donc $q = 0$ (contradiction !)

On a donc :

$$m \geq q+2$$

Soit $h = 3/4 + x$ la hauteur minimum des processeurs occupés au-delà de $3/4$ de S^1. Soit $h' = 1/2 + x'$ la hauteur du processeur de S^1 occupée strictement moins de $3/4$. Puisque la dernière transformation a échoué, on a $h + h' > \frac{3}{2}$. Cela entraîne que $x + x' > \frac{1}{4}$.

Nous pouvons évaluer la surface occupée de S^1.

$$\sum_{i \in S^1} W_i \geq (m-q-1)(\frac{3}{4}+x) + (\frac{1}{2}+x')$$

Comme $(m-q-1) \geq 1$

$$\sum_{i \in S^1} W_i \geq (m-q-1)\frac{3}{4} + \frac{1}{2} + x + x'$$

et comme $x + x' > \frac{1}{4}$,

$$\sum_{i \in S^1} W_i > \frac{3}{4}(m-q)$$

Il y a donc contradiction avec le fait que la surface occupée de S^1 est inférieure à $\frac{3}{4}(m-q)$.

Occurence de la dernière transformation Cette transformation n'intervient que dans les cas où :

- Une tache séquentielle proche de $1/2$ et des tâches proches de $3/4$ se partagent S^1,
- Il n'y a qu'une seule tâche dans le S^2 considéré,
- La tâche de S^2 est de la "bonne" taille.

Ces instances devraient être relativement peu fréquentes.

3.3.8 Conclusion

Cette heuristique montre l'impact important pour ce type d'ordonnancement de la structure que l'algorithme essaie de construire. Nous avons donc trouvé une structure d'ordonnancement qui permet d'obtenir une garantie de performance de $3/2 + \epsilon$ en $O(n\ m\ \log_2(\frac{1}{\epsilon}))$. Cette heuristique est plus simple que celle que nous utilisions pour obtenir la borne de $\sqrt{3} + \epsilon$. Sa grande simplicité nous laisse à penser que de meilleures garanties sont atteignables avec d'autres structures. Ces structures complexes devraient rendre ce problème plus difficile.

En moyenne, par contre, il n'est pas forcément évident que les ordonnancements seront meilleurs que celui de $\sqrt{3}$ car les tâches peuvent s'exécuter en temps 3/2. Pour l'algorithme en $\sqrt{3}$ les tâches s'exécutent toutes en temps inférieur à 1.

3.4 Comparaison en moyenne des différents algorithmes

Le comportement en moyenne des algorithmes n'est pas forcément lié avec la complexité au pire. Nous avons donc implanté les différents algorithmes et nous avons comparé leurs performances sur des instances tirées au hasard.

3.4.1 Tirage des instances

Nous avons construit nos instances de la manière suivante. Nous tirons selon une loi de probabilité les temps d'exécutions des tâches.

Le temps d'exécution sur $q + 1$ processeurs est une fonction du temps d'exécution de la tâche sur q processeurs.

$$t_{i,1} = X$$

$$t_{i,q+1} = \frac{q + X_{[0..1]}()}{q + 1}\ t_{i,q}$$

X est une variable aléatoire. $X_{[0..1]}$ est la restriction de X à l'intervalle $[0..1]$.

Les deux fonctions que nous avons utilisées ont une distribution gaussienne $Normale$ ou une distribution uniforme $Uniforme$.

En centrant la gaussienne de $Normale_{[0..1]}$ sur 0 ou 1, nous obtenons des instances avec des tâches très parallèles, ou au contraire, des tâches dont le temps varie très peu. Une plus grande variabilité entre les tâches de l'instance est obtenue en utilisant un tirage uniforme.

Nous avons reporté la moyenne de 250 tirages en faisant varier le nombre de tâches de 1 à 150 sur 32 processeurs.

3.4.2 Les différents algorithmes

Nous avons tester 4 algorithmes.

- Notre algorithme de garantie de performance $3/2$, présenté dans la section 3.3. Nous sélectionnons la meilleure solution après avoir testé plusieurs tailles pour la seconde étagère. Cela nous permet d'essayer d'améliorer la garantie obtenue. Les tailles choisies pour S^2 étaient donc $1/2$, $1/4$ et $1/8$. La garantie est uniquement pour $3/2$. Les autres sont uniquement des choix heuristiques.
- La version de Ludwig [87] de l'algorithme de Turek Wolf et Yu [110] en utilisant l'algorithme de Steinberg [108] comme algorithme de strip-packing.
- L'algorithme de liste bien connu, Largest Processing Time First (LPTF) [61], pris comme référence. Un seul processeur est alors alloué à chaque tâche. L'algorithme alloue en priorité les plus grandes tâches.
- L'algorithme [90] de garantie de performance $\sqrt{3}$. Nous avons également choisi plusieurs tailles pour la seconde étagère, $1-\sqrt{3}$, $1/2$, $1/4$ et $1/8$.

3.4.3 Expérimentations

Toutes les mesures reportées présentent le rapport entre la solution produite par chaque algorithme et la borne inférieure du temps d'exécution fournie par l'algorithme de Turek, Wolf et Yu.

Dans toutes les mesures que nous avons faites, nous avons observé que l'usage de l'algorithme de Steinberg pour le strip-packing conduit à un ratio de performance proche de 2 en moyenne. La figure 3.20 est obtenue avec une distribution gaussienne des tirages, centrée sur 1/2 et un écart type de 1/2.

Par contre nos deux algorithmes sont très proches de 1, c'est-à-dire que le ratio de performance obtenu en moyenne est bien meilleur que $3/2$ ou $\sqrt{3}$. La figure 3.21 représente un zoom de la figure précédente.

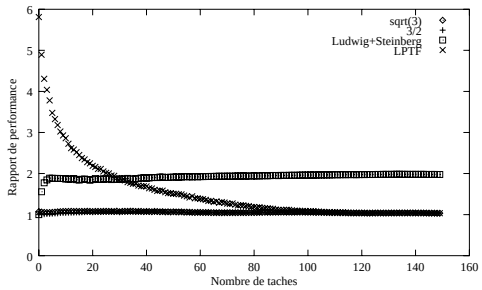


FIG. 3.20 – *Moyenne des ratios de performance pour 250 tirages par point pour les 4 algorithmes*

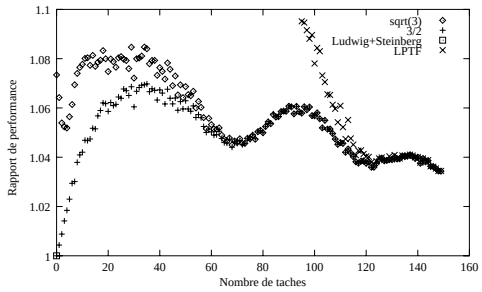


FIG. 3.21 – *Zoom sur le ratio de performance moyen pour 250 tirages par point pour les 4 algorithmes*

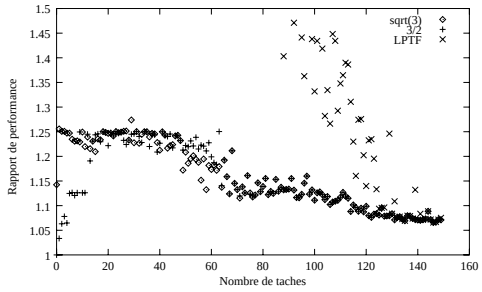


FIG. 3.22 – *Valeur au pire du ratio de performance pour 250 tirages par point des 4 algorithmes*

Le comportement au pire, sur 250 tirages par point, de notre algorithme est présenté par la figure 3.22. Au pire, le ratio de performance obtenu, pour 250 tirages, de nos deux algorithmes n'est pas plus mauvais que $1 + 1/4$ pour ces instances là.

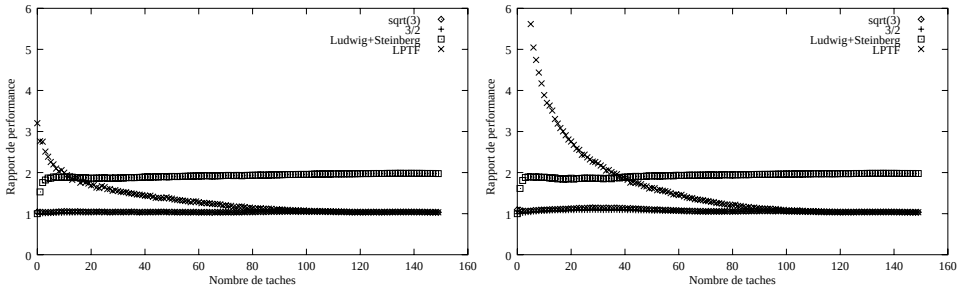


FIG. 3.23 – *Moyenne du ratio de performance pour 250 tirages par point des 4 algorithmes pour des tâches peu parallèles (à droite) et très parallèles (à gauche)*

Lorsque le tirage du facteur d'inefficacité est changé, le comportement des algorithmes varie un peu. La figure 3.23 présente les moyennes des ratios de performance pour un écart type de 1 et des distributions centrées sur 1 et 0, c'est-à-dire respectivement des tâches peu parallèle et très parallèle.

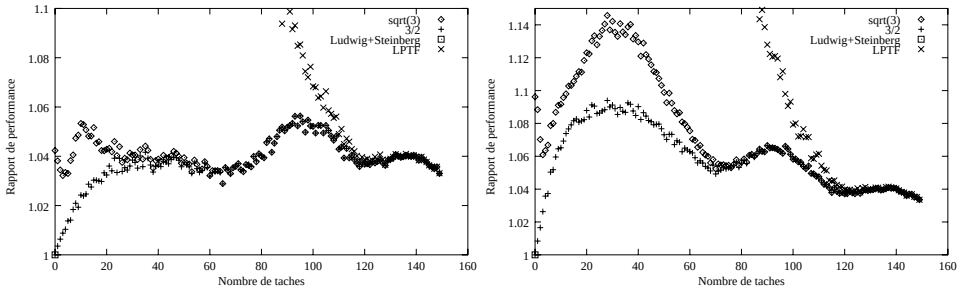


FIG. 3.24 – *Zoom sur la moyenne du ratio de performance pour 250 tirages par point des 4 algorithmes pour des tâches peu parallèles (à droite) et très parallèles (à gauche)*

Un zoom (cf fig. 3.24) permet de montrer que plus les tâches sont parallèles, plus nos algorithmes ont du mal à obtenir de bons ratios de performance.

Pour des tâches parfaitement parallèles, nous obtenons des ratios de performance, en moyenne, inférieurs à 1.2 pour les deux algorithmes (cf fig. 3.25). Il est à noter que, sur ces instances et pour un petit nombre de tâches, l'algorithme

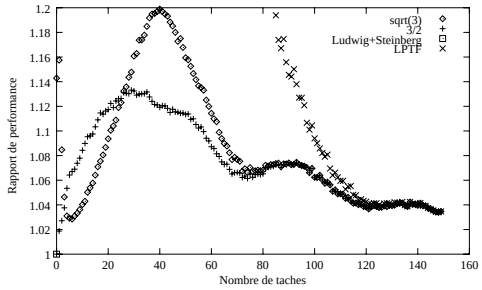


FIG. 3.25 – *Zoom sur la moyenne des ratios de performance pour 250 tirages par point des 4 algorithmes pour des tâches parfaitement parallèles*

de garantie $\sqrt{3}$ est meilleur que notre algorithme en $3/2$. Sur ces instances, il faut rappeler que l'ordonnancement optimal consiste à exécuter chaque tâche sur tous les processeurs, l'une après l'autre. Cette structure est donc relativement différente de celle qu'essaie de construire notre algorithme en 2 étagères. Cela se ressent particulièrement pour les petits nombres de tâches, où les pertes dues aux déséquilibres de charge peuvent être importantes.

On remarque que, pour l'algorithme LPTF, il faut jusqu'à 4 fois le nombre de processeurs avant que les performances des trois algorithmes deviennent équivalentes.

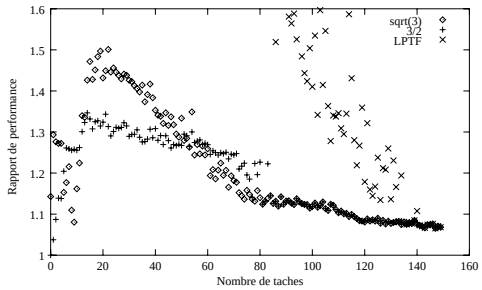


FIG. 3.26 – *Zoom sur la pire valeur du ratio de performance pour 250 tirages par point des 4 algorithmes pour des tâches parfaitement parallèles*

Pour ces instances dont les tâches sont parfaitement parallèles, nos algorithmes obtiennent des ratios de performance de l'ordre de 1.5 pour l'algorithme de garantie $\sqrt{3}$ et de 1.35 pour notre algorithme de garantie $3/2$ (cf fig 3.26).

Enfin nous avons aussi mesuré les performances obtenues pour des distributions uniformes sur $[0..1]$ du temps d'exécution et de l'inefficacité (cf fig 3.27). Ce cas est relativement favorable.

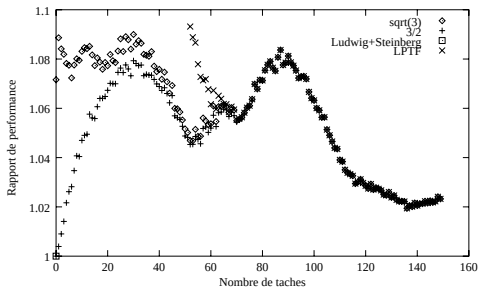


FIG. 3.27 – *Zoom sur la moyenne de la garantie pour 250 tirages par point des 4 algorithmes pour des tirages uniformes dans* $[0..1]$

3.4.4 Conclusion sur les expérimentations

L'utilisation de l'algorithme de Steinberg pour le strip-packing conduit à des ordonnancements multiprocesseurs coûteux avec des ratios de performance de l'ordre de 2.

Une prise en compte plus fine, du fait que le problème est discret, permet d'obtenir des ordonnancements dont le ratio de performance, en moyenne, est proche de 1. Ce ratio est bien meilleur, en moyenne, que la garantie au pire de $1/2$.

Chapitre 4

Ordonnancement de tâches malléables avec précédence

Contents

Dans ce chapitre nous allons présenter les différents travaux déjà réalisés autour des modèles proches des tâches malléables incluant des relations de précédences. Ensuite nous proposons un algorithme d'ordonnancement de garantie constante, pour une structure de graphe simple : les chaînes.

4.1 Introduction - Motivation

Le chapitre est organisé de la manière suivante : Nous allons tout d'abord présenter les différents travaux déjà réalisés autour des modèles proches des tâches malléables incluant des relations de précédences. Ensuite nous proposons un algorithme d'ordonnancement de garantie constante, pour une structure de graphe simple : les chaînes. Cet algorithme est un algorithme en deux phases. La première consiste à calculer une allocation, i.e le nombre de processeurs alloués à chaque tâche; elle est fondée sur un compromis entre le chemin critique et le travail total. Ensuite nous ordonnançons le graphe de tâches multiprocesseurs induit.

Nous montrons dans la section 4.2.2, comment utiliser la résolution d'un problème de sac-à-dos pour le calcul de l'allocation pour les chaînes et les arbres. Comme pour le problème de l'ordonnancement de tâches indépendantes, l'algorithme est fondé sur une recherche d'une allocation qui réalise un compromis entre l'ordonnancement multiprocesseurs, la somme des travaux et le chemin critique du graphe.

Le problème des tâches multiprocesseurs semble assez difficile à appréhender efficacement. Quelques résultats existent et sont présentés dans l'article de synthèse [35], mais aucun d'eux n'est général. Il est clair que tout résultat sur les tâches multiprocesseurs peut être directement utilisé pour ordonnancer un groupe de tâches malléables dont on aurait choisi, pour chaque tâche malléable, le nombre de processeurs qui l'exécutent. Nous montrerons par la suite, dans la section 4.2.1, que le problème posé par la difficulté de l'ordonnancement de tâches multiprocesseurs, peut être contourné efficacement.

4.1.1 Précédents travaux sur les tâches multiprocesseurs avec précédence

Il existe une autre approche modifiant le nombre de processeurs exécutant une tâche multiprocesseurs. Il s'agit de simuler sur un petit nombre de processeurs l'ensemble des processeurs demandés par la tâche. On parle alors de virtualisation des processeurs. Dans le cadre de ce modèle, les travaux de Feldman, Kao, Sgal et Teng [42] ont montré que l'on ne peut obtenir une bonne allocation avec des algorithmes *on-line* sans changer (ici, virtualiser) le nombre de processeurs

utilisés par les tâches. Ils obtiennent une garantie pour le problème du placement multiprocesseurs virtualisé, de $2 + \phi$ ($\phi \approx 0.618$, le nombre d'or).

Mais ce résultat ne s'applique pas aux tâches malléables. La différence, entre les deux modèles, se situe dans le comportement de la virtualisation. Le travail de la tâche ne diminue pas lorsque le nombre de processeurs l'exécutant diminue. Cela limite donc inférieurement le temps d'exécution virtualisé. Pour un problème malléable monotone, par contre, son travail peut diminuer jusqu'à un facteur m. La garantie de performance de Feldman et al. n'est donc pas une garantie de performance pour les tâches malléables, sauf pour les instances avec des accélérations (super-)linéaires. Il pourrait donc s'appliquer à des tâches non monotones.

4.1.2 Hypothèses sur les tâches malléables

Nous faisons pour notre problème, les deux mêmes hypothèses réalistes sur le comportement des tâches malléables que celles discutées dans l'hypothèse 3, page 36.

4.2 Ordonnancement d'un ensemble de chaînes de tâches malléables

Nous proposons un algorithme en deux phases pour l'ordonnancement d'un ensemble de chaînes de tâches malléables monotones. En effet, nous allons tout d'abord montrer un résultat préliminaire intéressant. Sous certaines hypothèses, il est possible d'ordonnancer un ensemble de tâches multiprocesseurs pour un graphe de précédence quelconque avec une garantie constante. Puis, nous montrerons que, dans le cas des chaînes de tâches malléables, un choix judicieux de l'allocation, c'est-à-dire du nombre de processeurs alloués à chacune des tâches malléables du graphe, nous permet de nous ramener à ce problème.

4.2.1 Ordonnancement de tâches multiprocesseurs

Dans cette partie nous allons étudier l'ordonnancement d'un graphe de tâches multiprocesseurs $G = (V, E)$. Une tâche multiprocesseurs est une tâche qui nécessite plusieurs processeurs pour s'exécuter mais contrairement aux tâches malléables, ce nombre est fixé. Nous notons respectivement p_i et t_i, le nombre de processeurs et le temps d'exécution d'une tâche i, ω^∞, le chemin critique (plus long chemin) dans le graphe G (le makespan d'un ordonnancement optimal sur une infinité de processeurs), et W_V le travail des tâches de V ($W_V = \sum_{i=1..n} t_{i,q_i}$, où l'allocation q_i est fixée).

Nous allons étudier le comportement d'un algorithme de liste [60] (cf algo. 4.2.1), en fonction du nombre maximum de processeurs nécessaires pour exécuter une tâche multiprocesseurs que nous noterons $\delta = \max_{i \in V} p_i$.

Algorithm 4 Algorithme de liste pour l'ordonnancement de tâches multiprocesseurs

$Ready =$ Calculer l'ensemble des tâches prêtes
while $Ready \neq \emptyset$ **do**
  for all $i \in Ready$ **do**
    Calculer $date(i)$, la date d'exécution au plus tôt de la tâche i.
  end for
  Trouver la première tâche de L telle que sa date de début d'exécution soit minimale.
  Exécuter cette tâche et recalculer l'ensemble $Ready$ des tâches prêtes .
end while

Théorème 7 *Le makespan ω_{Liste} de l'ordonnancement obtenu grâce à un algorithme de liste est borné par $\frac{W_V+(m-\delta)\omega^\infty}{m-\delta+1}$.*

Preuve :
Le principe est une généralisation du résultat bien connu de Graham [60]. On partitionne l'intervalle $[0, \omega_{Liste}]$ en I^+ et I^-, où I^+ correspond aux instants où, strictement, plus de $m - \delta$ processeurs sont occupés et I^- correspond aux instants où plus de δ processeurs sont inoccupés. On note par la suite $|I|$ la durée en temps de la période I. Sur la figure 4.1, nous avons représenté I^+ et I^-, avec $\delta = 4$ pour 8 processeurs.

Le travail total effectué pendant la période I^+ est plus grand que

$$(m - \delta + 1)|I^+|$$

, puisqu'il y a plus de $m - \delta$ processeurs occupés. Le travail effectué pendant la période I^- est plus grand que $|I^-|$ puisqu'il y a au moins un processeur occupé. On en déduit donc l'inégalité suivante :

$$W_V \geq |I^+|.(m - \delta + 1) + |I^-|$$

Nous allons maintenant essayer de borner la durée de l'intervalle I^-. Pour cela nous allons construire récursivement un chemin dans la fermeture transitive du graphe G. Soit x_1, une des tâches qui finissent en dernier. Supposons x_i défini. Si il existe au moins une tâche t s'exécutant dans $I^- \backslash [date(x_i), \omega_{Liste}]$, on définit

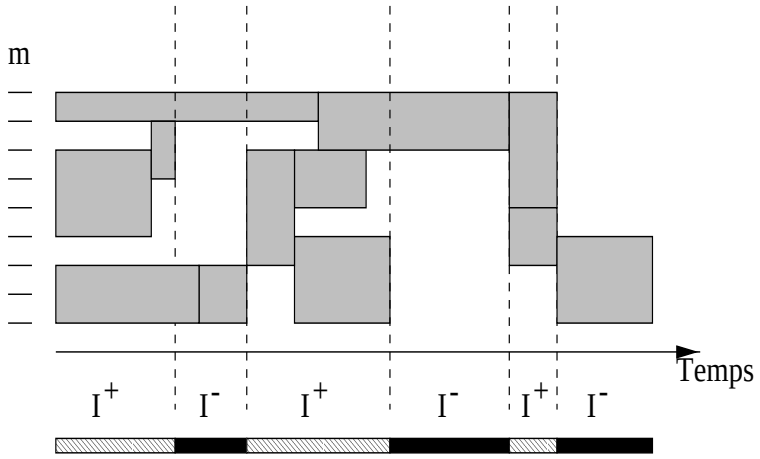


FIG. 4.1 – *Ordonnancement d'un graphe de tâches multiprocesseurs par un algorithme de liste.*

récursivement x_{i+1} de la façon suivante : Soit d_{x_i,I^-} la plus grande date de I^- plus petite que $date(x_i)$. Soit X_{i+1} l'ensemble des tâches qui s'exécutent à l'instant d_{x_i,I^-}. Il existe forcément au moins une tâche $t \in X_{i+1}$ telle que $t \prec x_i$, dans la fermeture transitive, sinon la tâche x_i, ou une autre tâche précédant x_i, aurait pu être exécutée plus tôt puisque son allocation est plus petite que δ. Nous choisissons pour x_{i+1} l'une de ces tâches. Cela nous permet de construire un chemin $C = (x_k, \cdots, x_1)$, tel que, par construction, pour tout instant $d \in I^-$, il existe une tâche du chemin C qui est en exécution à l'instant d.

Nous avons représenté les deux cas possibles, fonction de d_{x_i,I^-}, dans la figure 4.2.

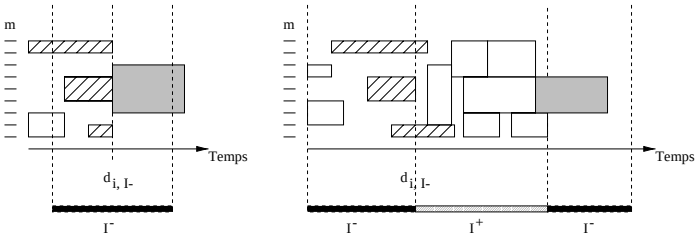


FIG. 4.2 – *Il existe une contrainte de précédence (un chemin dans le graphe de tâches) entre la tâche grisée et une des tâches hachurées*

Sur la figure 4.1, nous avons représenté un chemin possible.

Tous les chemins du graphe sont plus petits que le chemin critique ω^∞. On en

déduit donc l'inégalité suivante :

$$|I^-| \leq \omega^\infty$$

En combinant les deux équations on obtient finalement:

$$\omega_{Liste} = |I^+| + |I^-| \leq \frac{(W_V + (m-\delta)\omega^\infty)}{m-\delta+1}$$

□

Corollaire :
La garantie de performance d'un algorithme de liste pour l'ordonnancement multiprocesseurs est bornée par $\frac{(2m-\delta)}{m-\delta+1}$.

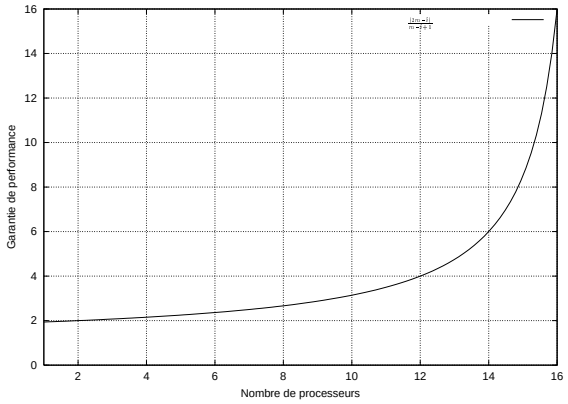


FIG. 4.3 – *Garantie de performance pour l'ordonnancement de tâches multiprocesseurs pour* $m = 16$ *en fonction de* δ

Nous pouvons remarquer que cette garantie correspond à la borne d'un algorithme de liste pour des tâches séquentielles lorsque $\delta = 1$, elle est alors égale à $2 - 1/m$, où l'on retrouve la borne bien connue. Cette garantie est mauvaise lorsque $\delta = m$, cependant si $\delta = m/2$ alors la garantie est bornée par 3. Nous avons tracé la borne de la garantie de performance d'un algorithme de liste en fonction de δ pour $m = 16$ sur la figure 4.3.

Corollaire :
Si $\delta = m/2$ *la garantie devient*

$$w_{Liste} \leq \frac{2}{m+2} W_V + \frac{m}{m+2}\omega^\infty$$

ou grossièrement

$$w_{Liste} \leq 2\ W_V + \omega^\infty$$

Ce dernier corollaire est la base du choix de notre stratégie d'allocation. Nous allons choisir une allocation afin d'obtenir une garantie de performance constante.

4.2.2 Choix d'une bonne allocation

Nous allons maintenant, nous intéresser au cas d'un graphe $G = (V, E)$ de tâches malléables. L'idée principale consiste à choisir une allocation du nombre de processeurs aux tâches, afin que le travail total, divisé par le nombre de processeurs, et le chemin critique soient bornés par rapport au makespan d'un ordonnancement malléable optimal. Puis nous utilisons le résultat de la section précédente pour ordonnancer le problème induit. Pour résoudre ce problème du choix de l'allocation, nous introduisons un nouveau problème qui consiste à trouver pour une longueur de chemin critique fixé, l'allocation qui minimise le travail total. Dans le cas d'un graphe formé de chaînes nous proposons un schéma complet d'algorithmes d'approximation pour le résoudre, c'est à dire une famille d'algorithmes polynômiaux avec une garantie de performance arbitrairement proche de 1.

Nous rappelons que $t_{i,q}$ désigne le temps d'exécution de la tâche i sur q processeurs. Nous notons par la suite G_{Alloc} le graphe de tâches multiprocesseurs induit par l'allocation $Alloc$, ω^{∞}_{Alloc} la longueur du plus long chemin de G_{Alloc}, et W_{Alloc}, le travail correspondant. Nous nous intéressons au problème suivant :

Définition 13 (Problème de l'allocation) *Trouver une allocation $Alloc$, sur l'ensemble des tâches T, telle que le maximum entre la longueur du chemin critique ω^{∞}_{Alloc} de G_{Alloc} et le travail moyen W_{Alloc}/m soit le plus petit possible.*

Nous allons tout d'abord montrer que si nous pouvons approximer ce problème alors il est possible d'approximer le problème de l'ordonnancement du graphe de tâches malléables G. Puis nous montrerons par la suite comment nous pouvons résoudre le problème de l'allocation pour le cas d'un ensemble de chaînes.

La proposition suivante permet de clarifier le rapport entre les solutions optimales du problème de l'allocation, et la solution optimale pour le problème de l'ordonnancement d'un graphe de tâches malléables.

Théorème 8 *La valeur à l'optimal du problème de l'allocation est plus petite que le makespan optimal du problème de l'ordonnancement du graphe de tâches malléables : $\omega^* \geq max(\omega^{\infty}_{Alloc^*}, W_{Alloc^*}/m)$, où $Alloc^*$ désigne la solution optimale du problème de l'allocation.*

Preuve: Soit $Alloc$ l'allocation de l'ordonnancement optimal du problème de l'ordonnancement du graphe de tâches malléables. Il apparaît clairement que : $\omega* \geq max(\omega^{\infty}_{Alloc}, W_{Alloc}/m)$. Or, $Alloc^*$ est par définition la solution optimale du problème de l'allocation : $max(\omega^{\infty}_{Alloc}, W_{Alloc}/m) \geq max(\omega^{\infty}_{Alloc^*}, W_{Alloc^*}/m)$, d'où le résultat.

Nous allons maintenant montrer comment construire à partir d'une approximation pour le problème de l'allocation, une approximation pour le problème de

l'ordonnancement d'un graphe de tâches malléables. Puis nous montrerons ensuite comment il est possible d'obtenir une bonne approximation pour le problème de l'allocation.

Théorème 9 *S'il existe une k-approximation pour le problème de l'allocation alors il existe une $4k$-approximation pour le problème de l'ordonnancement d'un graphe de tâches malléables.*

Preuve :
L'idée consiste à utiliser le résultat du théorème 7 (page 88) qui fournit une bonne approximation pour le problème si le nombre de processeurs alloués à chaque tâche n'est pas trop grand. Pour cela, considérons $Alloc$ la solution du problème d'allocation qui est par hypothèse une k-approximation, nous allons à partir de cette allocation construire une nouvelle allocation $Alloc'$. $Alloc'$ est définie par $Alloc'(t) = Alloc(t)$ si $Alloc(t) < \delta$ et par $Alloc'(t) = \delta$ sinon. La notation δ désigne un entier entre 1 et m représentant le nombre maximum de processeurs sur lequel on a choisi d'allouer les tâches dans $Alloc'$. Le makespan d'un algorithme de liste pour l'ordonnancement du graphe multiprocesseurs $G_{Alloc'}$ est donc, d'après le théorème 7, plus petit que :

$$\frac{W_{Alloc'} + (m-\delta)\omega^{\infty}_{Alloc'}}{m-\delta+1}$$

Or, d'après l'hypothèse de monotonie, nous avons : $\omega^{\infty}_{Alloc'} \leq \frac{\delta}{m}\omega^{\infty}_{Alloc'}$ et $W_{Alloc'} \leq W_{Alloc}$. D'autre part, d'après le théorème 8, $\omega^{\infty}_{Alloc} \leq k.\omega^*$ et $W_{Alloc}/m \leq k.\omega^*$, on obtient finalement la garantie de performance suivante :

$$\frac{\omega_{Alloc'}}{\omega^*} \leq \frac{k.m^2}{\delta\ (m-\delta+1)}$$

□

Nous avons représenté celle-ci sur la figure 4.4, pour $m = 16$. Cette garantie de performance est minimum pour $\delta = \lfloor (m+1)/2 \rfloor$, dans ce cas elle est alors égale à $\frac{4.k}{(1+1/m)^2} \leq 4.k$.

Finalement, nous allons maintenant voir comment nous pouvons trouver une approximation pour résoudre le problème de l'allocation pour le cas d'un graphe formé de chaînes. Pour cela nous allons utiliser une k-approximation duale, technique introduite par Shmoys et Hochbaum [68]. Étant donné un nombre réel λ, nous nous intéressons au problème suivant :

Définition 14 (Problème contraint de l'allocation) *Trouver une allocation, $Alloc$, sur l'ensemble des tâches T telle que la longueur du chemin critique ω^{∞}_{Alloc} est inférieure à λ et que le travail moyen W_{Alloc}/m soit le plus petit possible.*

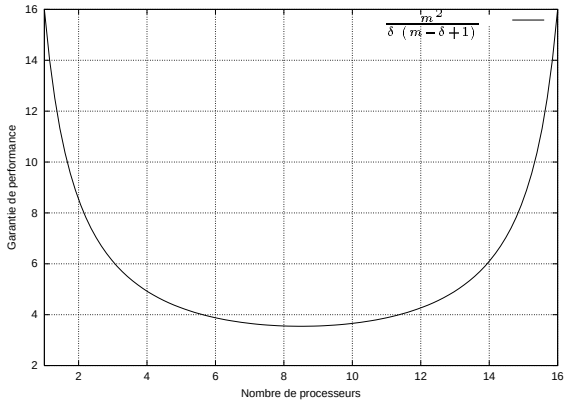


FIG. 4.4 – *Garantie de performance pour l'ordonnancement de tâches malléables, pour $m = 16$ en fonction de δ, si l'on sait résoudre exactement le problème de l'allocation*

Supposons que l'on connaisse une k-approximation pour ce problème qui fournisse une solution $Alloc$. Nous montrons par la suite un schéma pleinement complet d'approximation pour ce problème (i.e. $k = 1 + \epsilon$). Si $W_{Alloc}/m > k.\lambda$, alors il n'existe pas de solution pour le problème d'allocation de valeur inférieure à λ. D'autre part, si $W_{Alloc}/m < k.\lambda$, alors $Alloc$ est une solution réalisable du problème de l'allocation de valeur $k.\lambda$. Ces deux remarques permettent de diriger une dichotomie sur la date λ. Cela permet d'aboutir à la proposition suivante :

Théorème 10 *S'il existe une k-approximation pour le problème contraint de l'allocation alors il existe une $k.(1+\epsilon)$-approximation pour le problème de l'allocation.*

Preuve :
D'après les hypothèses de monotonie $\omega_1/m \leq \eta^* \leq \omega_1$, où $Alloc^*$ est la solution optimale du problème de l'allocation de valeur η^* par définition égale à $\max(\omega^{\infty}_{Alloc^*}, W_{Alloc^*}/m)$, et ω_1 est le temps de calcul du graphe sur 1 processeur. Nous cherchons η tel que $\eta^* \leq \eta \leq (1+\epsilon)\eta^*$, considérons un découpage de l'intervalle $[\omega_1/m, \omega_1]$ en $(m-1)/\epsilon$ sous-intervalles de longueur $\frac{\omega_1.\epsilon}{m}$. Par dichotomie on peut trouver en $\log_2((m-1)/\epsilon+1)$ étapes l'intervalle dans lequel se trouve η^*. Si on choisit η comme étant la borne supérieure de cet intervalle, on obtient alors $\frac{\eta}{\eta^*} \leq \frac{\eta^*+\omega_1.\epsilon/m}{\eta^*} \leq 1+\epsilon$. Or l'algorithme fournit une solution qui est une k-approximation de la solution donc une solution avec une garantie de $k.(1+\epsilon)$. □

Nous nous intéressons maintenant à la résolution approchée du problème contraint de l'allocation dans le cas d'un graphe formé de chaînes. Si l'on considère

un graphe G formé de k chaînes $(C_1, \cdots, C_k)$, alors on peut remarquer que ce problème peut être résolu en cherchant pour chaque chaîne la solution optimale à ce même problème. La solution optimale étant alors l'allocation formée des k allocations de chacune des chaînes. C'est pourquoi par la suite nous nous restreignons au cas d'une seule chaîne $C = (t_1, \cdots, t_n)$.

Nous montrerons tout d'abord que le problème contraint de l'allocation peut être résolu par un algorithme de programmation dynamique. En effet ce problème s'apparente en fait à un problème de sac-à-dos. À partir de celui-ci nous construirons un schéma d'approximation pleinement polynômial (*fully polynomial approximation scheme*) en utilisant les techniques connues [93, 51].

4.2.3 Algorithme de programmation dynamique

Nous allons calculer grâce à un algorithme de programmation dynamique $T(i, w)$ qui est le temps minimum pour exécuter les i premières tâches de la chaîne en travaillant moins que w. Nous notons par la suite $t_{i,q}$ et $w_{i,q}$ respectivement le temps et le travail de la tâche i si elle est exécutée sur q processeurs. Toutes les valeurs (temps et donc travail) sont considérées ici comme entières. Il est possible de se ramener à ce problème entier tant que les temps d'exécutions sont des nombres rationnels. Nous noterons $T(i, w) = +\infty$ si il n'est pas possible d'exécuter les i premières tâches en travaillant moins que w. Le schéma de calcul est donné par l'algorithme 4.2.3.

Algorithm 5 Algorithme de programmation dynamique pour le problème contraint d'allocation

$T(0, w) = -\infty$ si w négatif et $T(0, w) = 0$ sinon.
for all $i \in [1, n]$ **do**
  for all $w \in [1, W_m]$ **do**
    $T(i, w) = min_{q=1..m} T(i-1, w - w_{i,q}) + t_{i,q}$
  end for
end for

La solution au problème contraint d'allocation est le plus petit W^* tel que $T(n, W^*) < \lambda$. Malheureusement, la complexité du calcul de cet algorithme de programmation dynamique est en $O(n\ m\ W_m)$ et une implantation simple utilise un espace mémoire de $O(n\ W_m)$, où W_m est la somme des travaux des n tâches sur m processeurs. Or W_m est potentiellement très grand, ce résultat n'est donc pas directement utilisable en pratique, cependant il va nous permettre de construire un schéma complet d'approximation.

4.2.4 Schéma d'approximation complet

Un schéma complet d'approximation est une famille d'algorithmes (A_ϵ) telle que (A_ϵ) est un algorithme d'approximation de garantie $1+\epsilon$, dont la complexité est $P(k,\epsilon)$ où k est la taille de l'instance et P un polynôme.

Pour obtenir un schéma complet d'approximation, nous allons utiliser une méthode classique qui consiste à diviser ici la quantité de travail de chacune des tâches par un facteur K, (ce qui peut se voir comme un changement d'échelle) afin d'obtenir un nouveau problème. Puis nous allons étudier l'allocation optimale de ce nouveau problème par rapport au problème initial. Plus formellement, les travaux des différentes tâches dans ce problème sont définis par $w'_{i,q} = \left\lceil \frac{w_{i,q}}{K} \right\rceil$, les temps d'exécutions restant inchangés. La complexité du calcul de ce nouveau problème est bornée par $n\ m\ \frac{W_m}{K}$.

Remarquons tout d'abord que, puisque nous n'avons pas modifié les temps d'exécution, toute allocation réalisable de l'un des deux problèmes est une allocation réalisable de l'autre. Notons $Alloc'$ une solution optimale du second problème, nous allons maintenant chercher à borner le travail de l'allocation optimale $Alloc'$ pour le problème initial :

$$W_{Alloc'} = \sum_{i=1,\cdots,n} w_{i,Alloc'(i)} \leq k \sum_{i=1,\cdots,n} \left\lceil \frac{w_{i,Alloc'(i)}}{k} \right\rceil$$

Or $Alloc'$ est par définition la solution optimale du second problème donc :

$$W_{Alloc'} \leq k \sum_{i=1,\cdots,n} \left\lceil \frac{w_{i,Alloc^*(i)}}{k} \right\rceil$$

Donc nous en déduisons :

$$W_{Alloc'} \leq k\ n + \sum_{i=1,\cdots,n} w^*_{i,Alloc^*(i)} = k\ n + W_{Alloc^*}$$

L'erreur absolue commise sur le travail est donc bornée par $k\ n$. La dernière étape est le choix du coefficient K. Pour faire ce choix, nous allons maintenant utiliser l'hypothèse de monotonie. Celle-ci implique : $T_1 \leq W^* \leq m\ T_1$, où T_1 est le temps de calcul de la chaîne sur un processeur, c'est-à-dire $T_1 = \sum_{i=1,\cdots,n} t_{i,1}$. Si nous choisissons $K = \frac{T_1\ \epsilon}{n}$, on obtient donc $W_{Alloc'} \leq T_1\ \epsilon + W_{Alloc^*}$ c'est-à-dire : $W_{Alloc'} \leq (1+\epsilon)\ W_{Alloc^*}$. D'autre part, la complexité de l'algorithme est en $O(\frac{n\ m\ W_m}{K})$, elle est donc bornée par $O(\frac{n^2\ m^2}{\epsilon})$.

Finalement, pour tout ϵ, nous avons donc un algorithme de garantie de performance $(1+\epsilon)$ dont la complexité est polynômiale en la taille de l'instance et en $\frac{1}{\epsilon}$. C'est-à-dire un schéma d'approximation pleinement complet.

Ce résultat nous permet de conclure, d'après la proposition 10, qu'il est possible de trouver une solution au problème de l'allocation, dans le cas d'un graphe formé de chaînes, à ϵ près de l'optimal, en temps $\mathcal{O}\left(\log\left(\frac{m}{\epsilon}\right)\frac{n^2\,m^2}{\epsilon}\right)$. D'autre part, d'après la proposition 9, si l'on sait résoudre le problème de l'allocation avec une garantie k alors on peut résoudre le problème de l'ordonnancement de tâches malléables avec une garantie $4.k$. On peut donc résoudre le problème de l'ordonnancement d'un graphe de chaînes de tâches malléables avec une garantie $4+\epsilon$.

4.3 Conclusion et perspective

Nous avons présenté, pour le modèle des tâches malléables, un algorithme d'ordonnancement pour un problème incluant des contraintes de précédences. Par rapport aux travaux précédents utilisant uniquement des tâches indépendantes, nous avons montré que, pour des graphes composés de chaînes de tâches malléables, il est possible de calculer un ordonnancement avec une garantie $4\ (1+\epsilon)$ pour une complexité en $\mathcal{O}\left(\log\left(\frac{m}{\epsilon}\right)\frac{n^2\,m^2}{\epsilon}\right)$.

Nous espérons que ces premiers résultats théoriques ouvriront la voie vers l'ordonnancement avec garantie de graphes plus généraux, en particulier les arbres.

4.4 Perspectives

Pour cette première approche de l'ordonnancement avec contraintes de précédence, nous utilisons un algorithme relativement simple pour l'ordonnancement de tâches malléable avec relation de précédence. Cet algorithme est un algorithme en deux phases et il se heurte à la même difficulté que les algorithmes en deux phases, pour tâches indépendantes, à savoir l'ordonnancement multiprocesseurs.

La garantie de performance obtenue est également fonction de notre algorithme multiprocesseurs. Forts du résultat positif obtenu pour les tâches indépendantes, nous pensons que de meilleures garanties peuvent aussi être obtenues en utilisant une approche intégrée, comme nous l'avons fait pour les tâches indépendantes. Le problème est alors de trouver une structure pour le problème multiprocesseurs qui permette à la fois un ordonnancement simple et une allocation efficace.

Nous avons montré qu'il est possible d'obtenir des ordonnancements avec une garantie de performance constante en utilisant des algorithmes qui ne sont pas trop coûteux. Il nous reste maintenant à tester cela sur des problèmes pratiques.

Chapitre 5

Un exemple d'application : la simulation de courants océaniques

Contents

Cette partie du mémoire discute de la conception, de l'implantation, des performances de l'équilibrage de charge d'une simulation de circulation océanique à grande échelle. Nous allons d'abord décrire quelques aspects des modèles océaniques. Ces aspects sont présents dans la méthode d'adaptation du maillage utilisée et sont utiles à la compréhension du reste du mémoire. Nous allons ensuite montrer comment le modèle des tâches malléables s'imbrique dans la conception de l'application. Enfin, quelques expérimentations numériques préliminaires seront discutées. Certains aspects techniques sont reportés dans les annexes A et B.

5.1 Introduction à la simulation de courants océaniques

5.1.1 Contexte général

La modélisation numérique des circulations océaniques a débuté dans les années soixante pour se développer jusqu'à nos jours. C'est un problème important avec de nombreux débouchés (pêche, navigation, météo, etc.). Aujourd'hui, la modélisation des océans à deux buts principaux :

- Le premier but est l'étude du climat. Les prédictions sur l'évolution du climat, sur des échelles allant de quelques mois à quelques années, est un problème crucial. Les conséquences humaines et économiques de phénomènes comme *El Niño* ou de la montée des océans dues au réchauffement global sont importantes. Elles motivent le développement de modèles climatiques efficaces. L'océan est avec l'atmosphère l'une des parties essentielles du système climatique. Les modèles océaniques doivent donc être développés et couplés aux modèles atmosphériques pour obtenir un modèle climatique global.
- le deuxième but de la modélisation océanique est d'obtenir des prévisions d'évolution du "temps" semblable aux prévisions obtenues en météorologie. C'est le but du programme international GODAE (Global Ocean Data Assimilation Experiment) [105].

Le problème majeur rencontré lors de l'utilisation des modèles généraux de circulations océaniques est le très grand coût de calcul. Ces coûts sont plus grands que ceux obtenus lors de l'utilisation des modèles atmosphériques géographiquement correspondant. En effet, les mouvements océaniques typiques comme les tourbillons sont cinq à dix fois plus petits (de l'ordre de quelques dizaines de kilomètres) que les tourbillons atmosphériques. La résolution horizontale (c'est-à-dire la distance à l'échelle entre deux points de discrétisation) des modèles doit donc

permettre la prise en compte de ces structures. Cela implique des besoins importants, en puissance de calcul et en espace mémoire. Les résolutions actuelles étant typiquement de $1/6°$ à $1/10°$ (entre 10 et 15 kilomètres), plusieurs millions de points de grille doivent être calculés. De plus, les échelles de temps dans l'océan sont plus grandes que dans l'atmosphère, ce qui nécessite l'intégration des modèles numériques sur des périodes de temps plus longues, de quelques semaines à quelques dizaines d'années avec un pas de temps variant de 1 à 30 minutes. Ces modèles tournent sur des machines parallèles ou vectorielles car ils demandent plusieurs centaines ou plusieurs milliers d'heures de calcul.

Il est à noter qu'aujourd'hui les modèles tendent à utiliser des méthodes d'assimilation de données qui permettent de mélanger les prévisions du modèle et des observations [104]. Cela augmente encore le coût de calcul d'un ou deux ordres de magnitude.

5.1.2 L'adaptation du maillage

Les techniques de maillage adaptatif sont de plus en plus populaires. Elles consistent à ne placer un maillage fin qu'aux endroits où cela est nécessaire, au moment où cela est nécessaire. Il est clair que l'adaptation du maillage est d'un grand intérêt car elle permet de réduire le coût de calcul des modèles en tirant avantage de l'hétérogénéité spatiale des courants océaniques. De plus, cela peut permettre de faire des zooms locaux sur des zones d'intérêts.

Les techniques d'éléments finis permettent d'utiliser des grilles non-uniformes sur le domaine de calcul. Cette approche est utilisée par exemple pour la modélisation des côtes, mais la plupart des modèles globaux utilisent des méthodes de différences finies sur des grilles structurées.

Blayo et Debreu [12] ont développé une bibliothèque Fortran 90 qui permet de transformer n'importe quel modèle océanique séquentiel en différences finies, en un modèle adaptatif, permettant un zoom local. La méthode est fondée sur le schéma d'adaptation proposé par Berger et Oliger [8]. Il consiste en un ensemble de grilles, à différentes échelles de résolution en temps et en espace, qui interagissent. Comme l'adaptation du maillage évolue avec le temps, le nombre de grilles et leurs tailles varient pendant la simulation. Une stratégie d'équilibrage de la charge est donc nécessaire pour implanter efficacement cette stratégie sur un ordinateur parallèle. Dans ce travail de thèse, nous avons collaboré avec une équipe de numériciens du LMC-IMAG (Grenoble), spécialisée en océanographie. Il nous a donc fallu comprendre les problèmes pour pouvoir écrire une maquette parallèle d'une simulation océanique. À terme, le but est de pouvoir transposer les résultats obtenus dans cette thèse pour permettre une exécution parallèle efficace de la transformation adaptative d'un modèle océanique parallèle non adaptatif initial.

5.2 Modèles océaniques adaptatifs

5.2.1 À propos des modèles océaniques

La circulation océanique est modélisée par un ensemble d'équations décrivant les conservations des moments (équations de Navier-Stokes), de la masse (équations de continuité), de la chaleur, de la salinité et par une équation d'état exprimant la densité de l'eau en fonction de la température, de la salinité et de la pression.

Pour simplifier les équations, deux hypothèses sont généralement faites : les variations de densité sont petites (approximation de Boussinesq) ce qui induit une vélocité qui ne diverge pas, et l'approximation hydrostatique (le gradient de pression ne varie que par gravité et densité du fluide, selon l'axe Z). Cela donne un ensemble d'équations qui sont appelées en océanographie les *équations primitives* [95]. Ce sont elles qui sont résolues par la plupart des modèles de circulations océaniques généraux.

Il est possible de simplifier encore les équations en faisant encore d'autres hypothèses pour obtenir des modèles encore plus simples, comme le modèle quasi-géostrophique [11] ou le modèle *shallow-water*, modèle où le fluide est considéré comme homogène et inclus dans une couche mince.

Donc, un modèle océanique peut être écrit de façon symbolique comme:

$$\frac{\partial X}{\partial t} = F(X)$$

ou t est le temps, X est une variable d'état (incluant par exemple la vitesse, la température, la salinité et la pression pour les modèles utilisant les équations primitives) et F est un opérateur non-linéaire. Le schéma de discrétisation en temps est généralement explicite, ce qui mène souvent à des équations de la forme

$$X(t + \delta t) = G(X(t), X(t - \delta t))$$

ou δt est le pas de temps. Les opérations algébriques impliquées par l'opérateur G sont des versions discrètes d'opérateurs simples comme le gradient ($\vec{\nabla} f$), ou le laplacien ($\vec{\nabla}.\vec{\nabla} f$) (avec $\vec{\nabla} = \vec{i}\frac{\delta}{\delta x} + \vec{j}\frac{\delta}{\delta y} + \vec{k}\frac{\delta}{\delta z}$). Elles ne demandent que des calculs locaux, c'est-à-dire que les calculs en un point nécessitent uniquement de connaître les valeurs des points de son voisinage immédiat. Néanmoins, quelques modèles utilisent des schémas implicites sur une des équations, l'équation de mouvement barotrope (mouvement moyen dans le sens de la hauteur). Cela permet d'éviter l'utilisation de très petits pas de temps pour garantir la stabilité numérique dans les vagues de gravité externes. Pour ces schémas implicites, les équations à résoudre, à chaque pas de temps, forment un système linéaire de la forme (symbolique) :

$$A\ X(t + \delta t) = H(X(t), X(t - \delta t))$$

.

5.2.2 À propos de la parallélisation

La parallélisation d'un modèle océanique utilise les techniques de décomposition de domaine (e.g. [15] [112]). Les domaines géographiques sont divisés en sous-domaines, chacun d'eux étant affecté à un processeur.

Le calcul des termes explicites est principalement local, il ne nécessite que l'échange des valeurs des variables du modèle aux interfaces entre deux sous-domaines au début du pas de temps. Par contre, la résolution des systèmes linéaires pour les termes implicites n'est pas locale. Elle correspond à la forme discrétisée d'équations elliptiques. La résolution de ces systèmes peut se faire en parallélisant des méthodes classiques comme SOR, un gradient conjugué préconditionné [106] ou par des techniques de décomposition de domaine [29] [63].

Un des points importants à noter est que les modèles océaniques sont des applications régulières. Leurs volumes de calculs peuvent être estimés précisément comme une fonction de la taille de la grille du domaine et du nombre de processeurs impliqués. Leur temps d'exécution en parallèle peut être mesuré par un jeu de tests approprié.

5.2.3 À propos de l'adaptation du maillage

Comme nous l'avons vu, le principe de l'adaptation d'un maillage et de raffiner ou de déraffiner le maillage en fonction de critères mathématiques, comme une estimation de l'erreur de calcul, ou physiques, comme l'activité des tourbillons. Ces techniques sont assez souvent employées dans des codes d'éléments finis mais rarement dans des codes de différences finies car l'adaptation du maillage mène à des grilles non-homogènes, ce qui complique le code.

Berger et Oliger [8] ont proposé un algorithme d'adaptation qui évite cet inconvénient. Le raffinement n'est pas fait à l'aide d'une unique grille non-homogène mais d'une hiérarchie de grilles. La hiérarchie de grilles est composée d'un ensemble de grilles homogènes, composant des domaines imbriqués de résolution croissante. Ces grilles interagissent les unes sur les autres.

Le principe de l'algorithme (cf. [8] ou [12]) est le suivant : la racine de la hiérarchie de grilles est une grille qui recouvre tout le domaine avec une résolution grossière Δh_0 et un pas de temps Δt_0. Si l'on fixe le degré de raffinement par niveau r, toutes les grilles filles de cette grille parente ont une résolution plus fine $\Delta h_1 = \Delta h_0/r$ et un pas de temps plus petit $\Delta t_1 = \Delta t_0/r$. Mais les grilles ne recouvrent que quelques parties du domaine initial. Cette structure est récursive jusqu'à atteindre un nombre de niveau maximum $l_{\max}$ (cf fig. 5.1). Des structures

similaires d'arbres de grilles représentant un même "domaine" sont utilisées dans d'autres domaines comme l'imagerie (*quad tree*).

L'intégration en temps démarre sur la grille racine. Un pas de temps est calculé sur la grille grossière. Les solutions à l'instant t et $t + \Delta t_l$ sont utilisées pour calculer les conditions aux limites des grilles filles du niveau suivant. Lorsque ces grilles auront avancé de r pas de temps Δt_{l+1}, elles fourniront une solution plus précise à leur grille parente des régions qu'elles recouvrent. Ce processus est résumé dans l'algorithme récursif suivant :

Algorithm 6 Procédure d'intégration récursive des grilles niveau par niveau

Procedure INTEGRATION(ν) {Le niveau croit avec le degré de raffinement. La grille la plus grossière est au niveau 0}
if ν == 0 **then**
  $nbpas = 1$
else
  $nbpas = r$
end if
for i = 1 à nbpas **do**
  Faire un pas de temps sur toutes les grilles du niveau ν
  if $\exists$ niveau $\nu + 1$ **then**
    Calculer les conditions aux limites des grilles du niveau $\nu + 1$
    INTEGRATION(niv + 1)
    Mise-à-jour des grilles du niveau ν avec les solutions des grilles du niveau $\nu + 1$
  end if
end for

La hiérarchie de grilles doit rester pertinente tout au long de la simulation. Un critère est donc évalué pour chaque point de grille pour déterminer si la précision locale semble suffisante. Les différentes grilles de la hiérarchie sont alors éventuellement déplacées, créées, détruites ou redimensionnées.

L'un des intérêts de cette méthode d'adaptation est qu'elle peut être utilisée sans avoir à modifier le modèle. Ce modèle peut être vu comme une boîte noire calculant un pas de temps sur une grille.

Il faut quand même fournir au modèle les «bons paramètres» : domaine de calcul, taille de la grille, pas de temps, conditions initiales et condition aux limites.

Dans la version parallèle de l'application, le modèle sera aussi utilisé comme une boîte noire mais avec un paramètre supplémentaire : les processeurs qui doi-

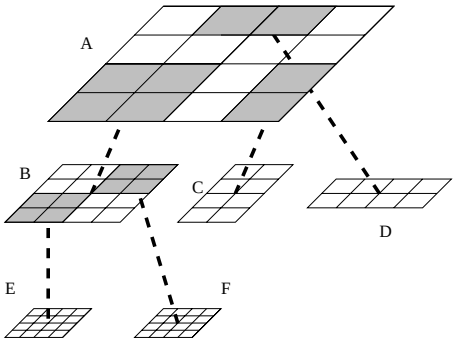


FIG. 5.1 – *Exemple de hiérarchie de grilles*

vent calculer la grille. Le problème revient alors à trouver le meilleur placement des grilles sur les processeurs de la machine parallèle.

Comme la hiérarchie de grilles évolue avec l'avancée de la simulation, ce placement doit être décidé à chaque fois que l'on change la hiérarchie de grilles. Le modèle des tâches malléables va servir à résoudre ce problème d'ordonnancement.

Une autre approche possible de l'ordonnancement serait l'utilisation d'ordonnancement à la volée (*on-line*). Elle pose plusieurs problèmes de performance, en particulier à cause du surcoût de migration des données, ou d'ordre purement technique, comme la capacité du code à pouvoir migrer en cours d'exécution. Nous n'avons pas traité cette voie.

5.3 Implantation parallèle

5.3.1 Le graphe des tâches malléables

Comme nous l'avons rappelé, une application parallèle peut être décrite par un graphe de tâches à exécuter [27]. Les tâches malléables permettent une description du graphe de l'application à son grain naturel. Il est déterminé par l'utilisateur (expert), développeur de l'application à paralléliser. Dans l'application de simulation océanographique que nous présentons, une tâche malléable correspond à la résolution des équations d'une des grilles pour un pas de temps donné. Le coût de calcul associé à cette tâche dépend de la taille de la grille en x et y, mais aussi de sa hauteur selon l'axe vertical (le nombre de couches dans le modèle). La structure d'arbres de grilles n'est utile qu'a l'ordonnancement des tâches et à l'adressage des communications. Chaque calcul est répété pour chaque pas de temps de la grille associée.

Le graphe de tâches malléables est identique au graphe de l'exécution séquen-

tielle récursive. Mais ce graphe n'est pas connu à l'avance. Le raffinement du maillage étant dirigé par les calculs, le graphe évolue au cours de la simulation. De nouvelles tâches malléables sont introduites. D'autres changent de taille ou disparaissent. Le problème d'ordonnancement est donc en ce sens dynamique. Néanmoins sa structure évolue en général relativement lentement au cours du temps.

La figure 5.2 représente le graphe de tâches malléables engendré par la hiérarchie de grilles de la figure 5.1. Le graphe est présenté sous deux formes. la partie gauche représente le graphe de précédence des calculs. La partie droite représente les flots de communication de données entre tâches malléables nécessaires aux calculs. Les arcs pleins représentent les communications dont le volume est de l'ordre de la taille des grilles (les mises à jour des grilles parentes par les grilles filles et la transmission des valeurs d'une grille). Les arcs hachurés représentent les communications dont la taille est de l'ordre de la racine carrée de la taille d'une grille (les données nécessaires aux calculs des conditions aux limites sur la frontière). Il est important de noter qu'un arc représente un ensemble de communications. Dans notre cas, c'est la distribution des grilles des tâches malléables émettrices et réceptrices sur l'ensemble des processeurs qui détermine les communications utiles.

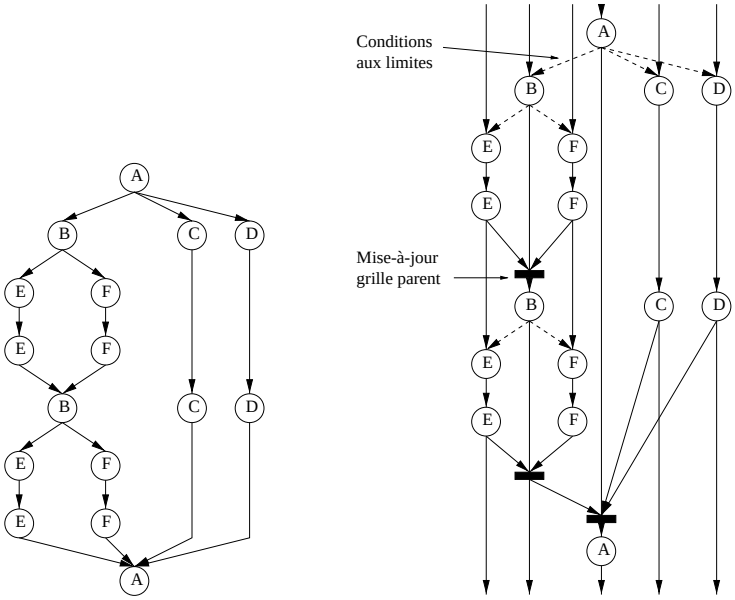


FIG. 5.2 – *Graphe de précédence et Graphe de flot de données de tâches malléables.*

Le but est de minimiser le temps total de l'exécution de la simulation. Pour cela il faut minimiser les temps d'attente et éviter de trop distribuer le calcul de

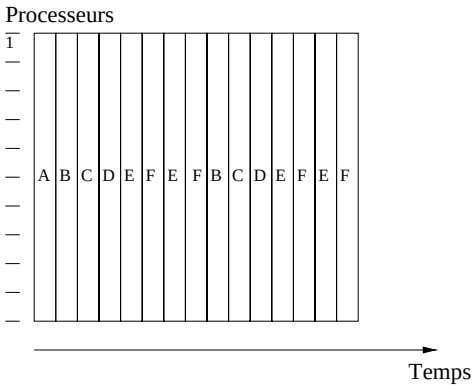


FIG. 5.3 – *Ordonnancement en gang*

chacune des tâches pour ne pas engendrer trop de communications. Le modèle océanographique est utilisé comme une boîte noire. Nous n'allons donc pas autoriser l'interruption de l'exécution du calcul d'une tâche malléable. Nous n'allons pas non plus autoriser la migration des grilles. Suivant les résultats obtenus, la préemption ou la migration pourraient être envisagées, même si cela rend le problème d'implantation plus complexe.

5.3.2 Les algorithmes d'ordonnancement

Pour ordonnancer le graphe, parmi les différents choix d'algorithmes possibles, nous avons choisi deux familles d'algorithmes : l'ordonnancement en gang et l'ordonnancement niveau par niveau. Il s'agit de montrer sur un exemple réel que le modèle des tâches malléables peut être utilisé avec profit. Une étude plus profonde devrait être menée pour utiliser des algorithmes plus sophistiqués tels que ceux développés dans cette thèse. Nous allons discuter les raisons de ces choix du point de vue de la simulation océanique adaptative et détailler les aspects techniques.

L'ordonnancement en Gang

Cette politique est très populaire en système [73, 101]. Elle consiste juste à ordonnancer chaque tâche malléable sur tous les processeurs à la fois, l'une après l'autre. La figure 5.3, montre l'ordonnancement en gang du graphe de la figure 5.2. Dans le cadre de notre application, cet ordonnancement présente des inconvénients :

– la qualité de cet ordonnancement devrait fortement dépendre de la valeur du facteur d'inefficacité,

– un grand nombre de communications vont être nécessaires pour acheminer les données entre les grilles. Cela pourrait remettre en cause notre hypothèse négligeant les communications entre tâches malléables.
– plus le nombre de processeurs exécutant une tâche malléable augmente, plus le coût de mise en place des communication devient important.

Mais il a aussi deux avantages :

– l'équilibrage de la charge de calcul est parfait,
– le volume de chaque communication est réduit et surtout elles sont distribuées entre les processeurs, ce qui permet de maximiser l'utilisation du réseau de communication.

Cet ordonnancement se calcule en temps linéaire et est très facile à mettre en oeuvre. Il permet un équilibrage parfait de la charge (calcul, mémoire) entre les différents processeurs. Sa performance est liée au facteur d'inefficacité. Pour être précis, sa garantie de performance est égale à la somme des facteurs d'inefficacités, divisée par le temps d'exécution de l'ordonnancement.

Théorème 11 *Pour des tâches monotones, l'ordonnancement en Gang a une garantie G qui vérifie :*

$$G \leq \frac{\sum_{i \in V} \mu_{i,m} t_{i,1}}{\sum_{t \in V} t_{i,1}} \leq \mu_{max}$$

Preuve :
Par définition du facteur d'inefficacité, le temps d'exécution sur m processeurs d'une tâche malléable monotone est égal à $t_{i,m} = \mu_{i,m} t_{i,1}$. Le temps d'exécution d'un ordonnancement en Gang est donc $\frac{\sum_{i \in V} \mu_{i,m} t_{i,1}}{m}$. Le minimum de la somme des travaux est atteint lorsque toutes les tâches sont exécutées par un seul processeur. Cette somme divisée par le nombre de processeurs est une borne inférieure du temps d'exécution. □

Dans des environnements d'exécution asynchrones qui permettent de recouvrir les calculs par des communications en exécutant en même temps toutes les tâches prêtes, cette stratégie peut s'avérer relativement efficace.

L'ordonnancement niveau par niveau

L'ordonnancement niveau par niveau est un compromis entre les temps d'attente dûs aux déséquilibres de la charge et les gains en efficacité obtenus en évitant d'allouer un trop grand nombre de processeurs par tâche malléable sur «trop» de processeurs.

Un exemple d'ordonnancement niveau par niveau du graphe 5.2 est montré par la figure 5.4.

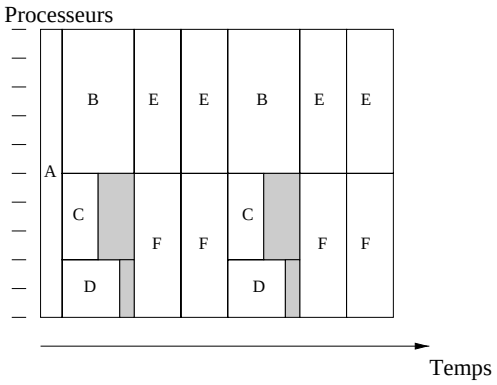


FIG. 5.4 – *Ordonnancement niveau par niveau*

L'ordonnancement niveau par niveau présente plusieurs avantages :

– il tient compte des "petits problèmes", dont le facteur d'inefficacité est important. Un "petit problème" correspond aux calculs sur une grille de petite taille par rapport aux nombres de processeurs. Elles peuvent apparaître à différents niveaux de raffinement pendant la simulation.
– il peut s'appliquer efficacement sur des instances ordonnées sur un très grand nombre de processeurs.
– les tâches situées sur un même niveau sont indépendantes les unes des autres. Il est donc possible d'utiliser pour l'ordonnancement de ce paquet de tâches malléables les heuristiques efficaces d'ordonnancement de tâches malléables indépendantes que nous avons présentées dans le chapitre 3. De plus, le temps passé par l'application à calculer l'ordonnancement des tâches malléables est aisément adaptable. Pour minimiser ce surcoût de l'ordonnancement, des politiques très simples d'ordonnancement des niveaux peuvent être employées, tandis que, pour les cas où le temps d'exécution d'une tâche malléable est très grand devant celui du calcul de l'ordonnancement, il est possible d'améliorer la garantie de l'ordonnancement proposé.

Niveau par niveau

Une politique d'équilibrage de charge niveau par niveau peut être adaptée aux tâches malléables. Une présentation complète de cette heuristique peut être trouvée dans [83]. Elle a été proposée dans le cadre du modèle classique de tâches monoprocesseur sans délai de communication. L'idée de base consiste à considérer chaque niveau du graphe de manière indépendante. Un niveau i est constitué de l'ensemble des tâches situées à une distance (longueur maximale des chemins) i du noeud racine (les arcs ne sont pas pondérés). Toutes les tâches d'un niveau

sont indépendantes (sinon, si a précède b, alors le chemin passant par a et allant à b est de longueur $i+1$ et donc b est au niveau $i+1$).

Cette stratégie consiste donc à ordonnancer l'ensemble des tâches indépendantes. La figure 5.4 détaille un exemple d'ordonnancement d'un graphe de tâches malléables niveau par niveau.

L'ordonnancement de chacun des niveaux peut être fait suivant de nombreuses stratégies :

- choisir en fonction de la taille des tâches une fraction des processeurs qui va exécuter chaque tâche malléable, sur une même étagère. Cette stratégie ne fonctionne que lorsque les temps d'exécution des tâches sont du même ordre et que le nombre de tâches est petit devant le nombre de processeurs.
- choisir une des heuristiques que nous avons présentées précédemment pour ordonnancer un ensemble de tâches malléables indépendantes : l'heuristique polynomiale de garantie 2 de [87, 86], notre heuristique de garantie $3/2$ ou même le schéma d'approximation polynomial de [75].

Autre ordonnancement possible : le placement proportionnel

Une autre politique relativement simple, que nous n'avons pas mise en oeuvre, est le placement proportionnel. C'est une stratégie classique d'ordonnancement d'un arbre de tâches. Il est utilisé par exemple pour l'ordonnancement de factorisation de Cholesky parallèle de matrices creuses [83]. L'idée du placement proportionnel est d'attribuer à chaque sous-arbre une quantité (discrète) de processeurs, proportionnelle au travail qu'il doit accomplir.

5.3.3 Discussion sur l'implantation des tâches malléables

L'utilisation du modèle des tâches malléables dans la conception d'une application influence son implantation. Dans notre cas, une tâche malléable est l'ensemble des calculs nécessaires au calcul d'un pas de temps sur une grille. Chaque tâche malléable est conçue comme si elle était un petit programme parallèle indépendant. L'implantation d'une tâche malléable est donc simple et relativement modulaire. Il suffit d'implanter une version parallèle des calculs effectués sur une grille. Le point difficile de la programmation est d'éviter l'introduction de synchronisations inutiles entre tâches malléables. Le programmeur n'est pas non plus déchargé de l'implantation des communications entre tâches malléables.

Dans la suite de ce chapitre, nous allons aborder en détail le problème de l'implantation. Nous présentons aussi les résultats pratiques que nous avons déduits de notre utilisation du modèle des tâches malléables pour l'application de simulation océanique.

5.4 Expérimentations

5.4.1 Le modèle océanique

Pour valider nos algorithmes d'ordonnancement et illustrer la faisabilité de l'utilisation des tâches malléables dans un contexte pratique, nous avons implanté une maquette de simulation océanique adaptative parallèle. Le modèle océanique utilisé dans les expérimentations est un modèle simple : le modèle quasi géostrophique. Le but est ici de valider expérimentalement notre approche et nous aider à concevoir de bonnes heuristiques d'ordonnancement pour une future simulation d'un modèle opérationnel. Cette (lourde) tâche est en cours. Le modèle quasi-géostrophique a été largement utilisé dans la communauté de la modélisation océanique. Il est reconnu comme prototype simple des activités tourbillonnaires à grande échelle autour des latitudes moyennes [11]. Debreu et Blayo [12] ont implanté une méthode de raffinement de maillage sur une version multi-couches de ce modèle. Ils ont montré que l'usage de cette méthode permet un gain significatif en temps de calcul (jusqu'à un facteur 3). Elle conserve les principales caractéristiques numériques du modèle dans un intervalle de 10 à 20 % par rapport à une solution obtenue sur une grille uniforme à haute résolution. Cette méthode donne aussi de meilleures prédictions locales que l'utilisation de techniques avec des grilles imbriquées, quelle que soit la région d'intérêt et ce, pour un coût de calcul comparable.

Dans cette étude, nous avons utilisé une version barotropique (c'est-à-dire à une seule couche d'eau) du modèle. Une description détaillée du modèle peut être trouvée dans [95]. L'équation qui dirige les calculs peut s'écrire comme suit :

$$\frac{\partial \Delta\psi}{\partial t} + J(\psi, \Delta\psi) + \beta\frac{\partial \psi}{\partial x} = \text{ rot } \tau - r\Delta\psi + A\Delta^2\psi \qquad (5.1)$$

où ψ est la fonction de courant, J est l'opérateur Jacobien, β est le gradient méridional du paramètre, à la latitude moyenne du domaine. τ est la tension horizontale du vent à la surface de l'océan, r est le coefficient de friction au fond, et A est le coefficient de viscosité latérale.

La discrétisation est faite sur une grille horizontale uniforme. Le laplacien discret est l'approximation standard à cinq points. Le Jacobien discret est écrit en suivant le schéma de discrétisation d'Arakawa [2]. La discrétisation en temps de l'équation de vorticité (cf l'équation 5.1) est explicite. Une résolution de l'équation de Poisson $\xi = \Delta\psi$ est effectuée à chaque pas de temps pour calculer la fonction de courant ψ, en utilisant une méthode d'analyse de Fourier avec une réduction cyclique [69]. Une évolution des isovaleurs de la fonction de courant est illustrée dans la figure 5.5.

L'exemple consiste en la simulation d'un rectangle océanique située aux latitudes moyennes avec un vent constant soufflant au milieu du rectangle. Dans

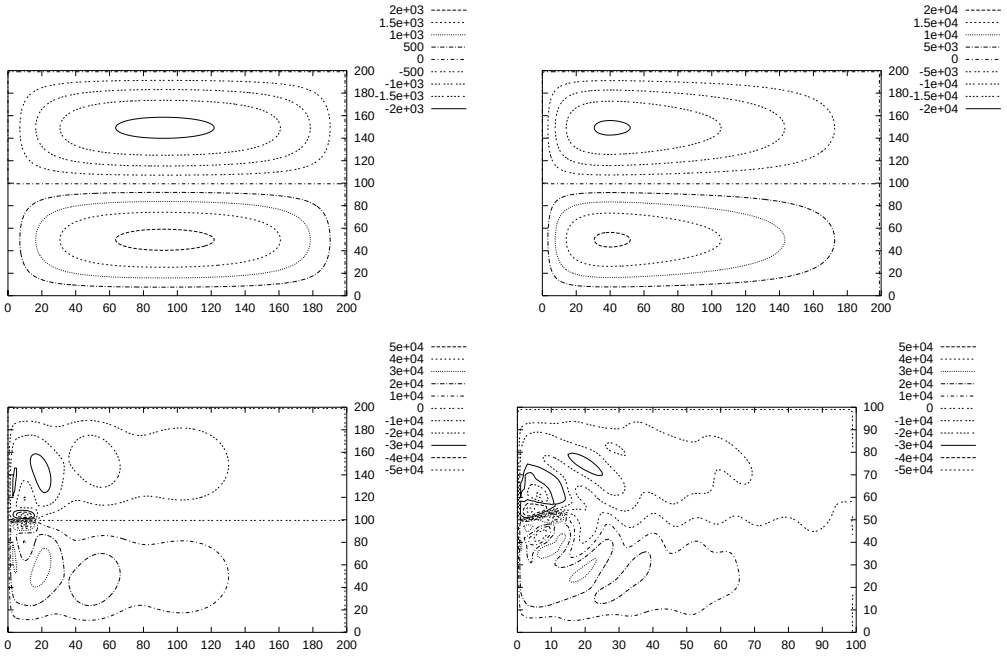


FIG. 5.5 – *Évolution des isovaleurs de la fonction de courant ψ pendant les pas d'initialisations (10, 100, 1000 et 50000 pas)*

cet exemple, la rupture de la symétrie et la création de tourbillons sont dues aux instabilités numériques. L'instant de cette rupture de symétrie peut être employé comme une méthode de test de la stabilité numérique des algorithmes. Par exemple, le comportement de la simulation avec l'algorithme adaptatif est proche de celui avec un pas de grilles fin. Nous allons utiliser cette évolution et ce déplacement des zones où les courants sont les plus forts pour valider notre modèle d'ordonnancement.

5.4.2 Contraintes d'implantation

Lors de l'implantation, certaines contraintes sont apparues. Comme pour toute application, le système sur lequel l'application va s'exécuter influence les possibilités de programmation. Nous souhaitons pouvoir appliquer nos algorithmes sur une version adaptative d'un modèle de simulation océanique existant. Les résultats des travaux exposés ici sont donc directement transposables à l'ordonnancement d'un modèle océanique complet. L'ensemble de ces aspects étant très technique, nous les avons consignés dans l'annexe A.

5.4.3 Transfert de données entre tâches malléables

Le modèle des tâches malléables décide de la distribution des calculs et des données au niveau du graphe de tâches malléables. L'ordonnancement de chaque tâche malléable est à la charge du programmeur. Dans notre cas, il s'agit de paralléliser les opérateurs appliqués aux données de chaque grille.

Dans notre modèle, nous négligeons le temps de communications entre tâches malléables lors du calcul de l'ordonnancement. Néanmoins, nous avons montré que, pour certains ordonnancements, la minimisation de ce coût est un problème de partition de graphe. Cette minimisation peut être calculée par des outils de partitionnement de graphes appropriés. Néanmoins, des communications seront toujours nécessaires. L'une des difficultés de la parallélisation de l'application réside dans la gestion de ces flots de communications. Plusieurs approches sont possibles :

– Gérer les communications à l'aide d'outils spécifiques aidant à la gestion de la distribution des tableaux de données comme POOMA [77], CHAOS PARTY [38], PETSc [4], etc.. Nous n'avons pas retenu cette solution car cela nécessite de profonds changements dans le code parallèle de calcul des itérations sur chacune des grilles. L'utilisation d'un tel outil ajouterait de plus des problèmes d'interactions et de mesure des performances pour ce qui concerne la gestion des données. Par contre, l'utilisation des ordonnancements de tâches malléables dans de tels outils pourrait être facilement automatisable.
– La méthode de gestion des données la plus simple est une gestion centralisée : les données issues d'une tâche malléable sont rassemblées avant d'être rediffusées vers les tâches malléables suivantes. On peut imaginer d'appliquer les interpolations des calculs de conditions aux limites lors de ce regroupement.

  Cette gestion est la plus facile à implanter dans les environnements de programmation parallèle de haut niveau comme Cilk ou Athapascan-1 qui ne possèdent pas encore d'opérateurs spécifiques pour la gestion des données distribuées. Il suffit pour cela d'utiliser les variables partagées du langage et d'utiliser les synchronisations adéquates en lecture et écriture de ces variables. Par contre ces deux langages ne fournissent pas (encore) d'objets partagés distribués et donc les données sont concentrées sur un noeud (éventuellement elles sont même dupliquées).

  Cette centralisation des données est pénalisante pour deux raisons. Premièrement, l'ensemble des données d'une grille peut ne pas tenir dans la mémoire physique d'un seul processeur ou provoquer un effondrement des performances à cause des mécanismes de mémoire virtuelle. Pour atténuer ce

problème, il est possible de centraliser les différentes données composant une grille sur différents processeurs. Deuxièmement, la centralisation des données est une forme de synchronisation des processeurs impliqués. La tâche malléable émettrice doit finir ses calculs avant que tous les processeurs calculant la tâche malléable suivante puissent commencer à travailler. Cela introduit donc des délais supplémentaires, qui auraient pu être évités avec des communications personnalisées entre les processeurs exécutant les deux tâches malléables.

– Puisque l'ordonnancement du graphe de tâches malléables est connu, la distribution de chaque tâche malléable détermine entièrement le flot de communication nécessaire à l'algorithme. Les communications sont en fait fonction du découpage (bandes et blocs réguliers) de l'ensemble des structures de données (tableaux, vecteurs). Ces découpages étant relativement simples, nous avons décidé de gérer explicitement toutes les communications entre tâches malléables.

La gestion explicite des communications entre tâches malléables est une des difficultés de l'implantation parallèle. C'est un point important qui doit être approfondi avant d'utiliser nos stratégies d'ordonnancement dans un véritable modèle opérationnel. C'est également une des questions ouvertes pour faciliter l'écriture de telles applications dans un langage de programmation parallèle : il faut exprimer de manière générique et efficace la distribution d'une entité.

5.4.4 Évaluation de l'inefficacité

Le facteur d'inefficacité μ correspond à un facteur de pénalité, du temps parallèle d'exécution, dû au surcoût de gestion du parallélisme. L'intérêt des tâches malléables est que le modèle tient compte implicitement les communications. La question fondamentale est donc la mesure de l'impact de ces communications au mieux, le plus précisément possible. Modéliser les communications de l'application est relativement difficile car il ne s'agit pas dans notre application de mesurer de simples communications point-à-point mais aussi de tenir compte d'éventuels problèmes de congestion. Par exemple, la résolution de système linéaire employée dans la maquette implique des échanges personnalisés de toute portion d'une tâche malléable vers toutes les autres.

Nous avons choisi comme mesure de base le temps de calcul d'un pas de temps sur une grille. Cela mesure assez précisément le temps de calcul d'une tâche malléable. Par contre, cela ne tient pas du tout compte des communications et calculs associés entre tâches malléables. Le calcul des conditions aux limites d'une tâche malléable va nécessiter d'acheminer les données correspondantes de

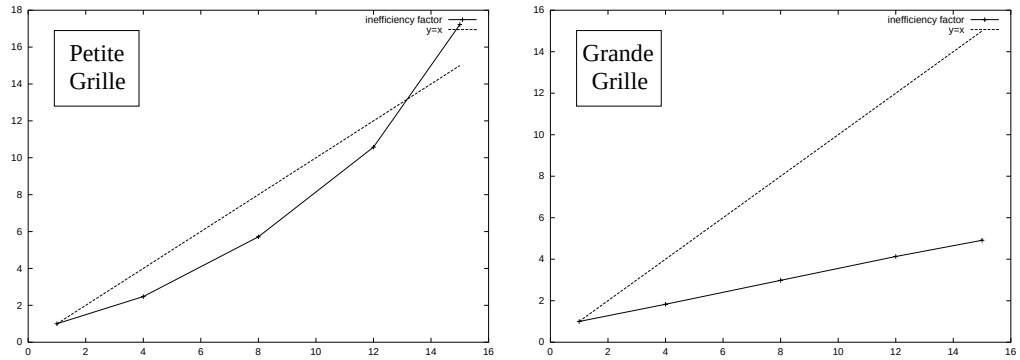


FIG. 5.6 – *Fonction d'inefficacité en fonction du nombre de processeurs. A gauche : une grille de taille 50x50; A droite : une grille de taille 500x500. La courbe de référence $y = x$ est également dessinée.*

la grille parente, et de faire des interpolations. Les mises-à-jour des grilles parentes peuvent engendrer un flot de données important.

Le temps d'exécution du code parallèle a donc été mesuré dans le but de déterminer empiriquement une approximation du facteur d'inefficacité pouvant être utilisée dans les stratégies d'ordonnancement.

Le facteur d'inefficacité a été mesuré en utilisant l'exécution parallèle des calculs sur une seule grille pendant 20 itérations. Comme pour les mesures de la section 2.3.4, tant que la mémoire physique du noeud n'est pas dépassée, et que les caches ont été remplis, les temps d'exécutions sont extrêmement stables sur la SP (moins de 10% de variation dans les temps d'exécution).

Les variations des courbes de la figure 5.6 en fonction du nombre de processeurs montrent un même comportement global. Deux aspects ont été étudiés :

– Dans la figure 5.6 on retrouve le comportement typique du facteur d'inefficacité μ en fonction du nombre de processeurs pour une petite et une grande grille. Pour une grande grille, le facteur d'inefficacité croît linéairement. Plus la taille de la grille diminue, plus la courbe du facteur d'inefficacité s'incurve. Après une certaine limite, le facteur d'inefficacité correspondant aux petites grilles croît plus vite que le nombre de processeurs. Par conséquent, l'utilisation d'un plus grand nombre de processeurs augmente le temps d'exécution de la tâche malléable au lieu de la faire décroître.
– La variation de μ en fonction de la taille de la grille est représentée dans la figure 5.7).

  La forme générale des courbes reste la même quel que soit le nombre de processeurs. Plus la taille de la grille est petite et le nombre de processeurs grand, plus l'inefficacité de l'exécution devient importante.

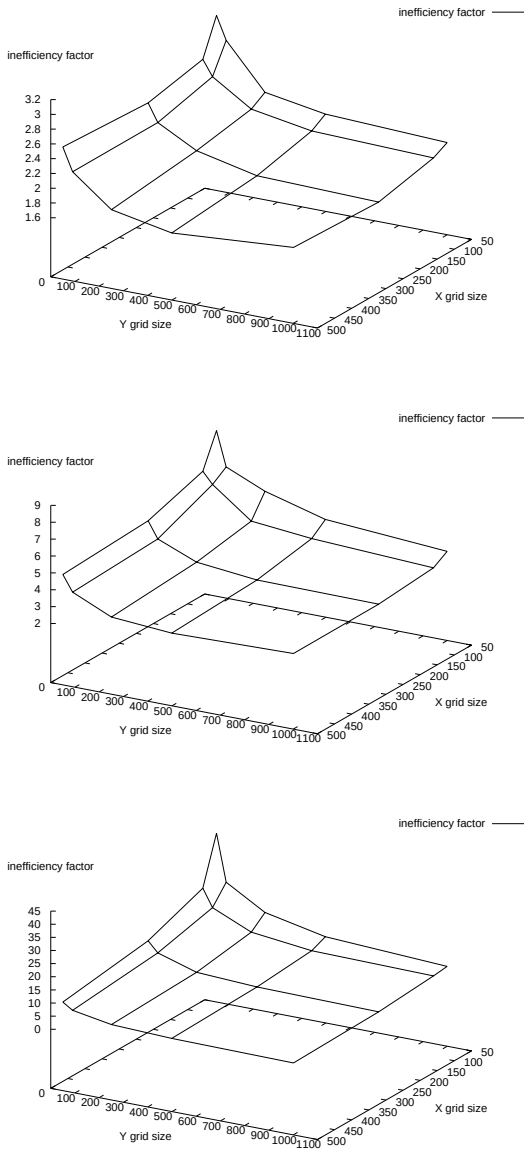


FIG. 5.7 – *Facteur d'inefficacité en fonction des tailles de grille dans les directions x et y. De haut en bas, pour 4, 8 et 15 processeurs*

Le facteur d'inefficacité que nous mesurons est extrêmement régulier. Son comportement est proche de celui présenté dans la section 2.3. Le surcoût important de la parallélisation provient de la résolution du système linéaire qui implique, dans ce code, deux transpositions de matrices et donc des échanges totaux entre les noeuds calculant la grille.

5.4.5 Résultats expérimentaux

Nous avons développé les expérimentations validant notre étude sur une machine parallèle IBM-SP2 en utilisant 16 noeuds connectés par un réseau rapide.

Le modèle océanique choisi est relativement simple. La série d'expériences qui suit est une expérimentation pour comparer les politiques d'ordonnancement. L'objectif est aussi de démontrer la validité de l'utilisation du modèle des tâches malléables pour ordonnancer de véritables applications. Il sert aussi de test pour trouver une stratégie destinée à être implantée dans le code réel.

Les simulations présentées ici concernent des exécutions où la grille parente à une petite taille (100x100) et une taille un peu plus grande (200x200). Nous avons mesuré le temps d'exécutions de 10 pas de temps sur la grille grossière sans raffinement des grilles. Les expérimentations ont été répétées plusieurs fois et nous présentons ici la moyenne des mesures. Sur l'axe des abscisses de toutes les courbes, on trouve reporté le nombre de processeurs. La structure de la grille est une homothétie de celle choisie par l'algorithme de raffinement au début de l'exécution séquentielle pour la taille 100x100 (pour éviter des problèmes de divergence entre les raffinements aux deux échelles). La figure 5.8 reporte le raffinement.

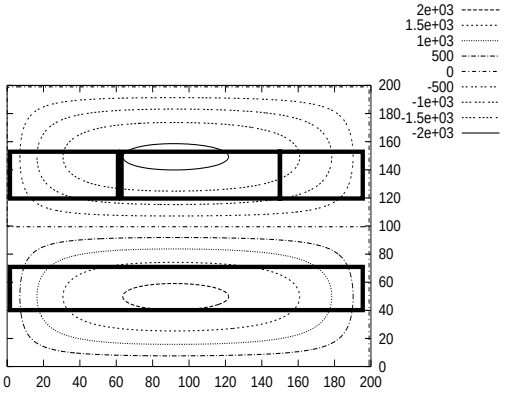


FIG. 5.8 – *Raffinement initiale*

Comme le nombre d'expérience par point est relativement petit, pour illus-

trer la stabilité, ou l'instabilité, des mesures, nous avons aussi reporté l'intervalle entre le minimum et le maximum de chacun des points de mesures. Elles sont relativement stables, sauf dans le cas de l'ordonnancement en Gang. Ces variations sont dues à la plus grande sensibilité de ces exécutions à l'état du réseau à chaque instant, d'autant que ces problèmes de synchronisations s'accumulent.

L'accélération et le travail sont des mesures de performances classiques. L'*accélération* désigne le rapport entre le temps d'exécution et le nombre de processeurs employé, tandis que le *travail* est le produit du temps d'exécution par le nombre de processeurs.

Une heuristique utilisant les informations du facteur d'inefficacité peut décider de limiter le nombre de processeurs qu'elle utilise pour la simulation. Il est donc nécessaire de définir la notion de *travail réellement effectué*, produit du temps d'exécution par le nombre de processeurs que l'algorithme d'ordonnancement a choisi d'utiliser. C'est ce travail qui est reporté dans la figure 5.13.

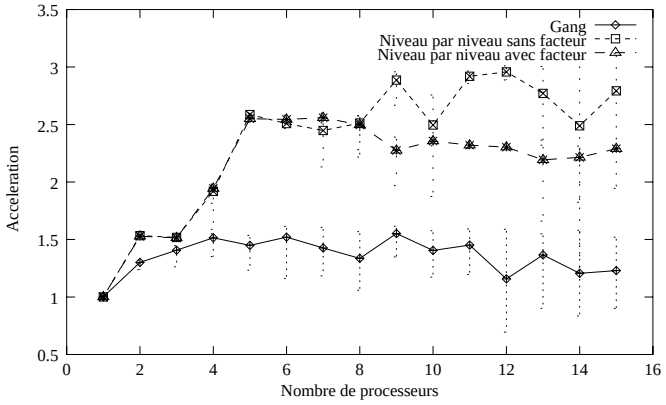


FIG. 5.9 – *Accélération obtenue en fonction du nombre de processeurs pour une grille de petite taille (100x100) et des tâches ordonnancées en* Gang*, et avec l'heuristique* Niveau par Niveau *avec et sans facteur d'inefficacité*

Les deux premières figures (cf fig. 5.9 et 5.10) concernent l'accélération obtenue pour les exécutions des 10 pas de simulation. Étant donné la faible taille des grilles et la relative importance des communications dans notre problème, les accélérations atteignables sont faibles.

Néanmoins nous pouvons observer quand même les points suivants :

– En règle générale l'accélération obtenue par l'ordonnancement en Gang est nettement plus faible que celle obtenue par les ordonnancements niveau par niveau.
– Sur un petit nombre de processeurs, il y a peu de différence entre les différents ordonnancements. L'ordonnancement en Gang est même parfois un

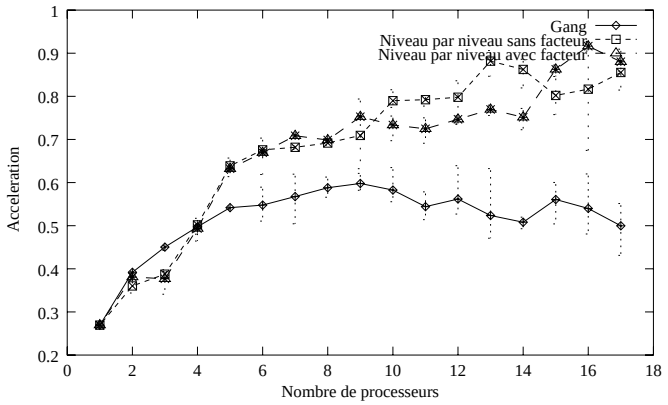


FIG. 5.10 – *Accélération obtenue en fonction du nombre de processeurs pour une grille de taille moyenne (200x200) et des tâches ordonnancées en* Gang, *et* Niveau par Niveau *avec et sans facteur d'inefficacité*

peu meilleur que les ordonnancements niveau par niveau lorsque le nombre de processeurs est petit.

Cela met en évidence que le coût des communications est important dans notre problème lorsque les grilles ne sont pas de grande taille. De plus, dans les cas où le nombre de processeurs est relativement petit, ce coût de communication plus grand peut être masqué par les gains induits par l'équilibrage de charge et le recouvrement des communications par les calculs. Lorsque le nombre de processeurs augmente, les déséquilibres de charges diminuent. Il n'y a plus non plus de recouvrement suffisant des communications et il est alors clairement plus avantageux de diminuer le coût des communications.

C'est pour cela que les ordonnancements niveau par niveau fournissent des ordonnancements avec des accélérations plus importantes.

Les figures 5.11 et 5.12 représentent les temps d'exécution en seconde des 10 pas de temps pour les grilles grossières de tailles 100x100 et 200x200. Les points suivants peuvent être notés :

– Les temps d'exécutions des simulations avec les différents temps d'exécution diminuent jusqu'à atteindre un palier à partir duquel les gains en temps deviennent beaucoup plus petits.
– Bien que le comportement global, des simulations sur les grilles petites ou moyennes, est globalement le même, les paliers ne sont pas atteints aux mêmes valeurs pour les deux types de simulations.

Dans notre problème, il y a donc des effets de palier à partir desquels les ajouts de processeurs, la ressource, n'ont que peu d'influence sur les temps d'exécutions.

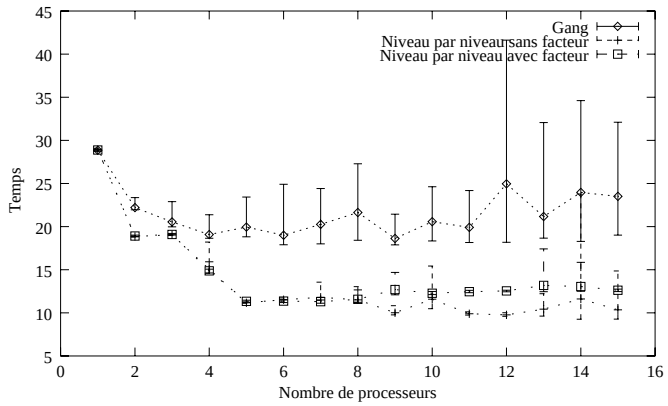


FIG. 5.11 – *Temps d'exécution pour une simulation de petite taille (100x100), avec des tâches ordonnancées en* Gang*, et* Niveau par Niveau *avec et sans facteur d'inefficacité, en fonction du nombre de processeurs.*

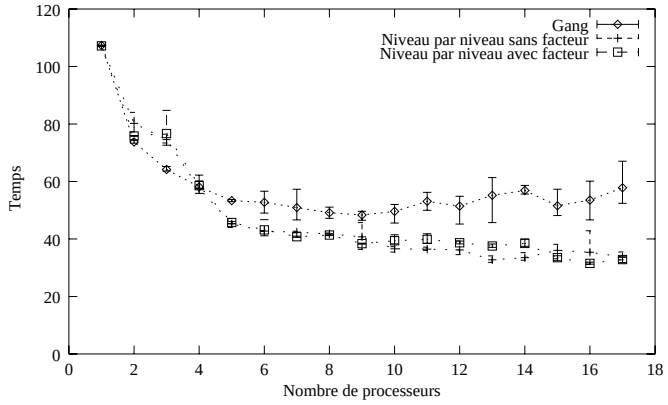


FIG. 5.12 – *Temps d'exécution pour une simulation de taille moyenne (200x200), avec des tâches ordonnancées en* Gang*, et* Niveau par Niveau *avec et sans facteur d'inefficacité, en fonction du nombre de processeurs*

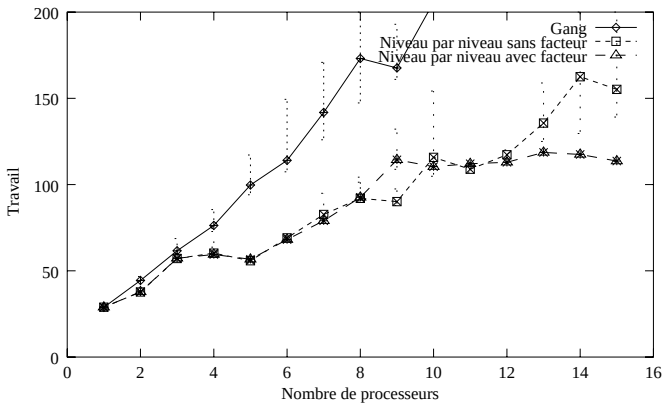


FIG. 5.13 – *Travail réellement effectué par une simulation de petite taille, ordonnancé en* Gang, *et* Niveau par Niveau *avec et sans facteur d'inefficacité, en fonction du nombre de processeurs.*

L'autre point important illustré par ces courbes est le fait que les paliers dépendent de la granularité des opérations (le rapport entre coût de calcul et coût de communication), et que cette granularité dépend de la taille des tâches.

Dans la figure 5.13, nous voyons apparaître un comportement plus fin de l'application qui va permettre de différencier les ordonnancements niveau par niveau avec et sans facteur d'inefficacité.

- Le travail fourni par les ordonnancements croît avec le nombre de processeurs utilisé.
- L'algorithme d'ordonnancement avec facteur d'inefficacité, contrairement aux autres algorithmes, Gang et Niveau par Niveau sans facteur d'inefficacité, ne dépasse pas un certain palier de travail.

Dans la figure 5.13, nous avons reporté le travail réel, c'est-à-dire le produit du temps d'exécution par le nombre de processeurs réellement utilisés par l'application. Pour l'ordonnancement en Gang et l'ordonnancement niveau par niveau sans facteur d'inefficacité, ce nombre est bien sûr identique au nombre de processeurs disponibles.

Pour l'algorithme d'ordonnancement niveau par niveau avec facteur d'inefficacité, ce n'est pas vrai. En effet, il n'utilise pas plus de 9 processeurs pour exécuter la simulation sur la petite grille. Cela signifie que les ressources employées pour la simulation sont bornées. De plus cette borne n'introduit pas un surcoût trop important dans le temps d'exécution de l'algorithme par rapport à la politique d'ordonnancement sans facteur d'inefficacité qui continue à utiliser toujours plus de processeurs.

5.4.6 Conclusions sur les expérimentations

La bonne efficacité de l'algorithme d'ordonnancement niveau-par-niveau avec facteur d'inefficacité nous permet de penser que ces résultats sont extensibles à de futurs travaux utilisant un plus grand nombre de processeurs. En particulier, les résultats montrent clairement que cet algorithme peut ordonnancer une simulation plus efficacement qu'un ordonnancement en Gang. De plus cet algorithme évite l'utilisation inutile de ressources sans réelle perte de performance. Il permet donc de gérer de manière naturelle le cas des problèmes de petites tailles qui peuvent apparaître au cours de la simulation en fonction du raffinement dynamique des grilles au cours d'une véritable simulation.

Il est à noter que la simulation adaptative utilise le modèle océanographique comme une boîte noire et elle est donc intrinsèquement synchrone, les communications entre tâches malléables n'ayant lieu qu'avant ou après les calculs. Le temps d'exécution est relativement sensible aux perturbations dans la synchronisation entre tâches malléables. C'est particulièrement vrai pour l'ordonnancement en gang qui induit des flots de communications relativement larges (c'est-à-dire impliquant un grand nombre de noeuds du réseau). Le temps d'exécution est sensible à la quantité de communications, ce qui explique la relative inefficacité de l'ordonnancement en Gang.

Enfin, les expérimentations ont aussi montré que, dans une certaine mesure, les communications de cette application peuvent être recouvertes efficacement par du calcul lorsque le travail à accomplir est important par rapport au nombre de processeurs impliqués. L'utilisation, de modèles de programmation plus asynchrones pour cette application, pourrait donc avoir un impact important sur les performances de l'application.

5.5 Perspectives sur la prise en compte de la localité

Cette section est une réflexion prospective qui met en évidence des questions potentiellement importantes pour l'implémentation réelle. Dans le modèle des tâches malléables, nous avons toujours négligé jusqu'ici les communications entre tâches malléables. Néanmoins, dans une application réelle, même si cette approximation est justifiée par le grain plus important des tâches malléables, il vaut mieux éviter de faire trop de communications. En effet, faire moins de communication entre tâches malléables permet d'éviter de perturber le comportement des communications internes des tâches malléables en occupant le réseau. Dans le cas où notre approximation est plus contestable, tenir compte de la localité pour diminuer le nombre et le volume des communications permet d'augmenter le rapport entre calcul et communication et donc de grossir le grain des tâches malléables.

Nous allons montrer maintenant que cette minimisation des communications, dans le cadre de l'ordonnancement niveau par niveau, peut être appréhendée et résolue comme un problème de partitionnement de graphe. Nous présentons brièvement notre expression du problème de minimisation des communications sous forme d'un partitionnement de graphe, et nous revenons sur certains détails après avoir présenté différents algorithmes de partitionnement de graphe. En effet, dans ce cas, des outils haut niveau de partitionnement peuvent être utilisés.

Le graphe des communications Le flot de communication qui est échangé entre deux tâches malléables est défini par le nombre de processeurs exécutant chacune de ces deux tâches, par leurs ordonnancements, et la distribution des données internes à chacune des tâches.

Définition 15 *Une* portion *d'une tâche malléable est l'ensemble des données et des calculs d'une tâche malléable placé par l'ordonnancement interne de la tâche malléable sur le même processeur.*

Sur une machine parallèle homogène, chaque portion peut être placée sur n'importe quel processeur, cela n'influence pas leur temps d'exécution. Par contre, la répartition respective des portions de tâches malléables sur les processeurs va influencer le schéma des communications nécessaires entre tâches malléables (cf fig. 5.14).

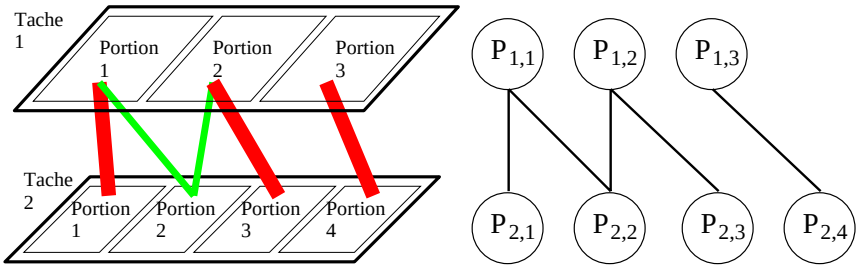


FIG. 5.14 – *Les flots de communications et le graphe associé entre portions de 2 tâches malléables*

La figure 5.14 illustre ces flots de communications entre portions (la grosseur du flot est proportionnelle à la taille des données qu'il transmet). S'il y a des communications, il y a alors des contraintes de précédences entre les deux tâches. Elles ne sont pas exécutables en même temps. Pour diminuer le nombre et le volume des communications il apparaît sur le schéma que les deux portions 1 doivent être exécutées sur le même processeur. Il en est de même pour les paires de portions respectivement de la tâche 1 et de la tâche 2, $\{2,3\}$ et $\{3,4\}$. La

portion 2 de la tâche ne peut pas alors être placée sur le même processeur que les portions 1 et 3 de la deuxième tâche.

Les portions vont être les noeuds du graphe. Chacun des flots va définir les arêtes du graphe que nous allons ensuite partitionner pour trouver une disposition des portions qui minimise le nombre et le volume de communications. La pondération des arêtes dépend du volume de communication dans le flot.

La principale difficulté est de garantir que l'algorithme d'ordonnancement partitionne le graphe de façon à ce qu'il n'y ait qu'une seule portion par processeur, par niveau. Il faut éventuellement gérer des groupes de portions de différentes tâches, empilées sur le même processeur par les décisions de l'algorithme d'ordonnancement (cf fig 5.15).

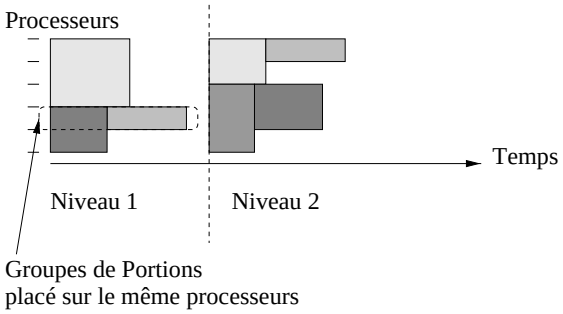


FIG. 5.15 – *Les groupes de portions*

Le partitionnement de graphe consiste à partager les noeuds du graphe (en fonction de leur pondération) en plusieurs parties de même poids, tout en minimisant le coût des arêtes coupées. Nous montrons dans la section 5.5 la présentation des différents algorithmes de partitionnement, comment pondérer les noeuds du graphe pour prendre en compte les décisions de l'algorithme d'ordonnancement.

En plus des aspects structurels du graphe, il est aussi possible d'associer à chaque portion des informations "géométriques" comme les indices des données qu'elles manipulent, ou la position dans un monde "réel", de la fraction des données sur laquelle elles travaillent.

Nous discutons, dans l'annexe B, des différents algorithmes de partitionnement de graphe utilisables en pratique.

Application du partitionnement de graphe pour minimiser les flots de communications

Lors d'un ordonnancement niveau par niveau, on peut optimiser la répartition des taches malléables sur les processeurs pour tenir compte des communications entre tâches malléables.

D'un côté, il est imaginable d'implanter une heuristique d'optimisation propre pour faire la partition, voir de chercher l'optimal lorsque le nombre de tâches/allocation/processeurs n'est pas trop grand. Mais nous montrons maintenant qu'il est possible d'utiliser les algorithmes de partitionnement qui respectent des critères d'équilibrage fixé.

Le graphe est construit comme indiqué précédemment, c'est-à-dire qu'un noeud est associé à chaque portion (cf fig. 5.15). Les arêtes correspondent aux flots de communications entre portions.

Les niveaux sont optimisés l'un après l'autre. Les tâches d'un même niveau sont indépendantes. Le premier niveau est placé arbitrairement sur les processeurs. Les autres niveaux sont placés l'un après l'autre. L'ordonnancement des tâches du niveau et les positions des portions précédentes définissent un graphe de communication.

Les entrées du problème sont :

– Le graphe des portions

– L'ordonnancement des tâches malléables

Le but est de calculer le placement $Place(i, j)$ de chaque portion j de chaque tâche i sur l'un des processeurs de la machine.

S'il n'y a qu'une seule étagère de tâches (cf fig. 5.16), le placement est relativement facile. Il n'est pas nécessaire que le partitionnement tienne compte des décisions d'ordonnancement. Il suffit que toutes les portions soient pondérées identiquement. Cela revient à calculer une simple permutation des processeurs (cf fig. 5.16.

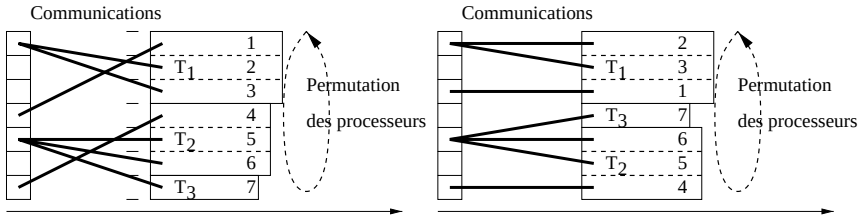


FIG. 5.16 – *Partitionnement du graphe à une seule étagère*

S'il y a plusieurs étagères dans l'ordonnancement des tâches du niveau, il faut que l'algorithme de partitionnement prenne en compte cette structure d'empilement décidée par l'algorithme d'ordonnancement (cf fig.5.17).

Nous allons construire un graphe dont la partition équilibrée va être une solution à notre problème.

Pour cela, nous allons affecter un poids $P_{i,j}$ à chaque portion j de chaque tâche i. Nous allons aussi ajouter m (le nombre de processeurs) noeuds "résidu"

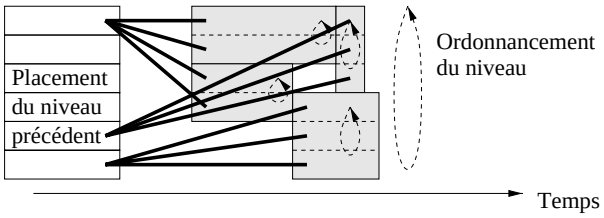


FIG. 5.17 – *Partitionnement du graphe sur plusieurs niveaux*

de poids P_q^r. q représente le numéro d'un processeur virtuel de l'ordonnancement malléable du niveau. L'algorithme 7 permet de calculer ces poids.

Algorithm 7 calcul du poids de chaque noeuds

{ affectation du poids de chaque portion}
for $1 \leq i \leq n$ **do**
  for $1 \leq j \leq Alloc(i)$ **do**
    $P_{i,j} = 2^{i+1}$
  end for
end for
{ affectation du poids des tâches fantômes}
for $1 \leq q \leq m$ **do**
  $P_q^r = 2^{n+2} - 1 - \sum_{i \in PlaceSur(q)} P_{i,1}$
end for

Théorème 12 *La partition équilibrée du graphe est une solution au problème de placement des portions.*

Preuve :
Nous avons en fait construit un codage binaire de l'occupation de chacun des processeurs virtuels de l'ordonnancement initial. La somme totale des noeuds à placer est $m\ (2^{n+2} - 1)$. Une partition équilibrée place donc une somme poids de noeuds de $2^{n+2} - 1$. Or Pour chaque tâche i, il y a $Alloc(i)$ noeuds de durée 2^{i+1} et $m - alloc(i)$ noeuds résidu de durée supérieure à 2^{i+1}. Il y a exactement un seul de ses noeuds par processeur sinon la charge d'un des processeurs devient plus grande que 2^{n+2}. Tous les processeurs sont maintenant occupés à strictement plus de 2^{n+1}. Le même argument s'applique donc pour la tâche $n - 1$ de poids 2^n. □

La figure 5.18 illustre ces partitions. Nous avons éclaté la représentation des tâches fantômes que nous avons reliées par une arête pour illustrer le fait que toute

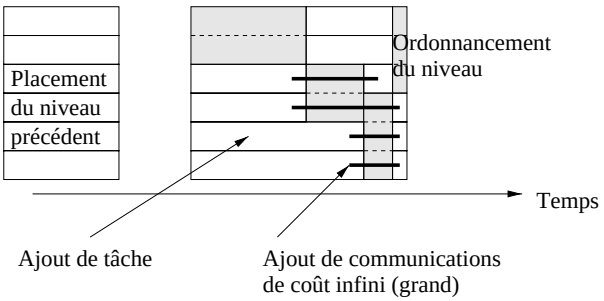


FIG. 5.18 – *Equilibrage et forcage du partitionnement a plusieurs niveaux*

partition équilibrée, grâce à l'introduction de noeuds fantômes, permet d'éviter l'exécution d'une portion sur un processeur où la tache malléable n'est pas assignée. Le partitionnement peut néanmoins permuter les processeurs et permuter les portions des tâches entre les différents processeurs exécutant la tâche comme montré dans la figure 5.17.

Chapitre 6

Conclusion et Perspectives

Nous avons présenté dans ce travail, une nouvelle manière de concevoir la parallélisation d'une application. Elle est fondée sur le modèle des tâches malléables avec facteur d'inefficacité. Nous avons discuté quelques-unes des propriétés fondamentales des tâches malléables et du facteur d'inefficacité.

Nous avons ensuite montré que le modèle des tâches malléables permet un certain nombre de résultats théoriques d'ordonnancement. Nous avons proposé de répartir les heuristiques d'ordonnancement en deux catégories.

La première calcule une allocation multiprocesseurs avec des garanties sur la quantité de travail et la longueur du chemin critique. Nous avons montré comment construire un ordonnancement efficace à partir de cette allocation. Nous avons proposé une heuristique pour le problème de l'ordonnancement de chaînes de tâches malléables monotones.

La deuxième classe d'algorithmes regroupe les heuristiques choisissant l'allocation pour faciliter le problème de l'ordonnancement multiprocesseurs. En plus de quelques heuristiques connues que nous avons adaptées aux tâches malléables, nous avons proposé un nouvel algorithme d'ordonnancement de tâches indépendantes ayant une garantie absolue de $3/2$, pour une complexité linéaire en le nombre de tâches et le nombre de processeurs.

Nous avons enfin évoqué l'utilisation des outils classiques de partitionnement de graphe pour optimiser le placement de chaque tâche malléable sur la machine afin d'optimiser les communications.

Le modèle des tâches malléables permet donc :

- de construire des ordonnancements avec une garantie de performance constante pour des problèmes incluant des relations de précédences,
- d'obtenir des garanties meilleures qu'une approche utilisant les algorithmes généraux de strip-packing en utilisant l'aspect discret de l'allocation des tâches et la monotonie de leur comportement.

Nous avons utilisé le modèle des tâches malléables dans la conception d'une application parallèle efficace. Pour cette validation expérimentale de notre approche, nous avons implanté une simulation adaptative à grande échelle de courants océaniques. Nous avons comparé les performances de plusieurs des algorithmes proposés dans l'ordonnancement de notre application.

Les expérimentations présentées dans le chapitre 5 ne sont pas exhaustives et doivent être complétées, mais elles montrent plusieurs avantages dans l'utilisation du modèle des tâches malléables :

- Le modèle des tâches malléables permet de manipuler facilement l'ordonnancement de l'application. La connaissance des performances d'une grille autorise de bien meilleures performances que l'utilisation de stratégies aveugles comme l'ordonnancement en Gang.
- Les changements des stratégies d'ordonnancement dans l'implantation utilisant le modèle des tâches malléables sont faciles, car ils se limitent à changer le calcul des allocations et de l'ordonnancement.
- Le cas des petits problèmes est pris en compte de façon simple sans surcoût.
- Les ressources (processeurs, mémoire, etc.) sont utilisées efficacement :
  - en limitant les ressources utilisées sans un grand surcoût dans le temps d'exécution
  - ou en distribuant les ressources plus efficacement pour limiter l'usage de la mémoire ou des communications.

Un certain nombre de points n'ont pas été traités dans cette thèse et constitueraient des prolongements intéressants de ce travail.

D'abord, le modèle des tâches malléables pourrait être étendu aux machines hétérogènes. Cela nécessite d'augmenter le nombre de paramètres du facteur d'inefficacité afin qu'il reflète les caractéristiques de la machine. En particulier il faudrait étudier la définition des paramètres afin de faciliter la mesure ou la prédiction efficace du facteur d'inefficacité.

Ensuite une deuxième piste plus théorique serait d'adapter les schémas d'approximation de strip-packing à notre problème malléable.

Il serait aussi intéressant d'étudier l'impact de l'aspect discret de l'une des dimensions pour les algorithmes de strip-packing dans le but d'obtenir de meilleures garanties absolues.

D'autres heuristiques pourraient aussi être appliquées au problème des tâches malléables. En particulier, des stratégies simples (calcul d'allocation ou placement proportionnel) permettraient de calculer de bons placements d'arbres.

Un certain nombre de points pratiques peuvent être étudiés. En particulier, il serait intéressant de mesurer l'écart en termes de performance, entre notre version de la simulation et une version asynchrone (communications et multiples

flots d'exécution). Cela permettrait d'évaluer deux aspects du coût d'utilisation du modèle des tâches malléables pour l'ordonnancement :

- le coût de communication entre tâches malléables,
- le coût de la synchronisation implicite induit par le fait de privilégier les activités liées à une tâche plutôt qu'à une autre.

D'autres applications parallèles, comme la factorisation de Cholesky creuse, pourraient avantageusement tirer parti du modèle des tâches malléables pour calculer leur ordonnancement. En effet, la résolution de systèmes linéaires par des méthodes directes est une opération coûteuse en temps et non triviale dans sa parallélisation [37]. La factorisation est effectuée généralement en trois étapes : le calcul d'une permutation pour réduire le remplissage [62], une factorisation logique pour calculer ce remplissage puis enfin la factorisation numérique. Pour maximiser l'efficacité des calculs, les colonnes ayant la même structure sont groupées. Cela permet l'utilisation des BLAS de niveaux 3 [33].

L'arbre d'élimination représente le graphe de tâches. Un noeud ne peut être calculé que si tous ses fils dans l'arbre ont déjà été factorisés. La découpe des colonnes de la matrice puis la gestion distribuée des "gros" noeuds du sommet de l'arbre pourraient être appréhendées efficacement et simplement avec le modèle des tâches malléables, sans qu'il y ait besoin de différencier ces deux étapes.

D'autres applications s'adapteraient facilement à ce modèle. Ces premiers résultats nous laissent espérer que le modèle des tâches malléables pourra servir de base à d'autres travaux de parallélisation.

Annexe A

Contraintes d'exécution

Cette annexe présente les détails techniques de l'implantation de la simulation adaptative parallèle que nous avons réalisée.

A.1 Modèle d'exécution

L'algorithme récursif 6 de la page 102 est celui qui est employé par la version adaptative séquentielle. La transformation faite par Laurent Debreu et Éric Blayo du code de simulation océanique non-adaptatif en un code adaptatif est fondée sur l'utilisation des *POINTER* Fortran 90.

A.1.1 Du modèle séquentiel non-adaptatif vers un modèle séquentiel adaptatif

La plupart des codes océaniques sont écrits en FORTRAN (77 ou 90). Les données manipulées par le code non-adaptatif sont, en général, un ensemble de variables globales (*/COMMON/*).

Les *POINTER* FORTRAN 90 sont des références sur les objets pointés, plutôt que des pointeurs, avec une arithmétique, au sens C. L'avantage est que la syntaxe d'utilisation d'un objet ou du *POINTER* sur cet objet reste la même.

Debreu et Blayo utilisent les *POINTER* pour sauvegarder et manipuler le contexte de chacune des grilles du modèle. Les variables globales sont remplacées par des *POINTER* sur le même type de données. Il suffit alors d'allouer autant de données que le nombre de grilles à calculer. Pour effectuer des calculs sur les données d'une grille, il suffit d'avoir préalablement affecté chacun des *POINTER* pour qu'ils référencent les données du bon contexte.

Le calcul d'un pas de temps sur une grille $G1$ dans l'algorithme 6 devient :

```
Installer_Contexte(G1)
Faire_1_pas_de_temps(G1)
```

La transmission de données d'une grille est aisée. Le calcul des conditions aux limites des grilles filles peut se faire en interpolant les données de la grille parente. Les mises à jour de la grille parente par ces grilles filles sont de simples copies de données.

A.1.2 Du modèle séquentiel non-adaptatif vers le modèle parallèle non adaptatifs

La parallélisation d'un code de simulation océanique non adaptatif est relativement simple. Ce type de code est une succession d'opérateurs appliqués aux données (laplacien, résolution de systèmes linéaires, etc.). Il suffit de paralléliser chacun de ces opérateurs. Une parallélisation *SPMD* est relativement facile à mettre en place. Il suffit de choisir la manière dont les données sont distribuées. Les données sont, dans notre cas, des tableaux à deux dimensions. Nous avons choisi une distribution en bandes. Pour tous les opérateurs explicites, la parallélisation est très simple puisqu'il suffit de mettre à jour les frontières de chacune des bandes. La version parallèle de l'opérateur de résolution de système linéaire de notre modèle a besoin de faire deux transpositions de matrices pour les communications. Puisque nous avons décidé de découper les données en bandes, la transposition consiste à faire un *échange total personnalisé* entre les différents processeurs impliqués.

A.1.3 Vers une version parallèle du modèle adaptatif

La version parallèle adaptative est le produit des deux versions précédentes. Elle tient compte du mélange des deux aspects prépondérants dans les versions précédentes.

- Le contrôle de l'exécution. Pour pouvoir calculer une itération d'une grille, il faut avoir positionné son contexte avant de lancer les calculs. Il faut aussi que les données qui vont servir à l'interpolation des conditions aux limites soient présentes localement. Comme nous allons programmer cette application en utilisant le paradigme du passage de messages, il faut prendre garde à ne pas provoquer un interblocage entre deux processeurs s'attendant mutuellement.
- Les données doivent être échangées entre tâches tenant compte de la distribution de chacune des grilles, pour amener les bonnes données aux bons

processeurs. Une alternative pourrait être de dupliquer toutes les données, partout. Mais elle est coûteuse en espace mémoire et en coût de communication. Elle est impraticable sur un grand nombre de processeurs.

A.2 Contraintes venant des modèles océaniques

Notre objectif dans ce travail est de démontrer l'utilité du modèle des tâches malléables pour l'ordonnancement d'une version adaptative parallèle des modèles océaniques opérationnels.

L'implantation des modèles océaniques étant réalisée en FORTRAN, plusieurs contraintes doivent être prises en compte. En effet, pour être portable d'une plateforme d'exécution à une autre, l'implantation doit tenir compte de certaines particularités des compilateurs et bibliothèques FORTRAN. Contrairement au compilateur C, le compilateur FORTRAN a une grande latitude dans sa gestion de la mémoire du programme en exécution. Nous allons montrer l'influence de ces contraintes sur la programmation de l'application.

A.2.1 Contraintes sur la gestion du contrôle de l'application

Une méthode naturelle pour l'implantation de l'application est l'utilisation des flots d'exécutions asynchrones (*threads*). L'association d'un flot à l'exécution de chaque grille garantit l'absence d'interblocage et permet le recouvrement des communications par les calculs. Cependant l'utilisation de flots d'exécutions asynchrones n'est pas possible pour deux raisons :

- L'implantation de l'adaptation utilise des variables globales *POINTER* pour gérer le contexte de calcul de chaque grille. Avant qu'un flot ne puisse effectuer les calculs concernant une grille, il faut donc que ces variables soient correctement positionnées. Cela empêche l'exécution concurrente simple de plusieurs flots d'exécutions. Il existe deux façons de contourner ce premier problème. Soit en protégeant l'utilisation de chacun des *POINTER* par un *mutex*, c'est-à-dire, un mécanisme d'exclusion mutuelle. Ces mécanismes doivent être manipulés avec précaution pour éviter l'introduction de nouveaux interblocages. Soit en modifiant le code initial afin que des variables globales ne soient plus utilisées. À la place, les données doivent être passées en paramètre pour chaque fonction. Il est possible de passer, par exemple, une structure FORTRAN 90 contenant l'ensemble du contexte d'exécution d'une grille. Cette opération s'avère difficile si le code du modèle de simulation océanique à modifier est de grande taille.
- En FORTRAN 77, il n'y a pas de fonctions récursives, car les variables locales de ces fonctions sont, en général, persistantes (statiques). Plusieurs

flots d'exécution ne peuvent donc pas exécuter la même fonction en même temps. FORTRAN 90 a introduit les fonctions récursives avec l'ajout d'un mot clé supplémentaire. Il est donc possible de modifier le code des fonctions pour que plusieurs flots d'exécution puissent l'exécuter en même temps. Malheureusement il n'en est pas de même pour les bibliothèques systèmes. Deux exemples : sous le système IBM AIX 4.2, la documentation de l'environnement de développement parallèle [72] indique qu'un seul flot d'exécution peut exécuter du code FORTRAN.

Ce problème s'inverse pour d'autres bibliothèques d'algèbre linéaire optimisées comme la bibliothèque SunPerf sous Solaris 2.7. Elle n'est pas seulement réentrante, mais les calculs peuvent être effectués en parallèle sur les machines multiprocesseurs à mémoire partagée. Pour en tirer pleinement parti, il faut donc que l'application elle-même soit aussi à multiples flots d'exécutions. Cela permet de recouvrir plus efficacement les calculs par les communications et d'éviter des échanges de messages.

A.2.2 Contraintes sur la gestion de la mémoire

La sémantique du langage FORTRAN définit le résultat des opérations mais ne définit pas vraiment la sémantique des opérations de manipulations de la mémoire. Le langage C est utilisé dans l'écriture de systèmes d'exploitation. Il est donc obligé de définir clairement les opérations de manipulation de la mémoire (arithmétique de pointeur, allocation mémoire contiguë). Un compilateur FORTRAN peut choisir la stratégie la plus efficace pour manipuler la mémoire. Mais il doit le faire de manière transparente pour un programme séquentiel. En FORTRAN, le passage des paramètres d'une fonction se fait par référence. Sur AIX, le compilateur FORTRAN s'autorise à recopier un tableau passé en paramètre à une fonction afin de ne passer que cette copie.

Dans un programme parallèle utilisant une bibliothèque d'échanges de message, le programmeur spécifie les échanges de données entre mémoires. Plus la mémoire est contrôlée finement, plus ces échanges peuvent être faits de façon efficace.

Tous les constructeurs de machines parallèles à mémoire physiquement distribuée fournissent une implantation de MPI comme bibliothèque à passage de message portable. Certaines implantations de cette interface peuvent réaliser des échanges de message très efficaces, sans copie inutile.

Dans les bibliothèques de passage de messages, des limitations dans les transmissions permettent de préserver la mémoire du récepteur ou les tampons des couches réseaux. Ce sont des mécanismes de contrôle de flux : lors de l'échange d'un message, si la mémoire sur le processus récepteur est saturée, les messages en provenance du processus émetteur sont bloqués. Pour éviter de bloquer, en

même temps, l'émetteur (et ainsi provoquer des risques d'interblocage), les messages sont acheminés de manière asynchrone. Pour cela, deux méthodes sont employées :

- Les données à envoyer sont copiées dans un tampon. Le «système» achemine ce tampon sur le récepteur lorsqu'il y a assez de place mémoire disponible. C'était la méthode employée par la bibliothèque PVM [52]. Le stockage intermédiaire des données permet de transformer ces données en utilisant dans les tampons un codage des données indépendant de l'architecture du processeur. Ce type de techniques est particulièrement utile dans les environnements hétérogènes comme les réseaux de stations de travail.

  Cependant, la copie de données a un coût non négligeable. Elle prend du temps, surtout si le codage des données est modifié à la volée. Elle occupe aussi de l'espace mémoire puisque l'espace mémoire utilisé par les communications est doublé.
- Pour pallier ces inconvénients, l'utilisateur de MPI [45] décrit à la bibliothèque la position en mémoire des données à envoyer. L'utilisateur teste ensuite la terminaison de l'envoi avant de pouvoir réutiliser ces données. Cela permet l'envoi des données sans copie.

  En conclusion, cette deuxième méthode asynchrone est la plus efficace si les données à envoyer ont une représentation mémoire simple, c'est-à-dire si elles sont groupées sous la forme de tableaux, et si l'environnement d'exécution est homogène.

Les deux méthodes que nous venons de décrire s'appliquent également à la réception de messages. Dans le cas de MPI, l'implantation de la bibliothèque définit le choix de l'une ou l'autre (ou les deux) de ces deux méthodes. Ce choix n'est pas accessible à l'utilisateur.

Un problème se pose : la copie des tableaux passés en paramètre, décidée par le compilateur FORTRAN. Pour effectuer un envoi asynchrone, il faut décrire la position des données en mémoire. Cela consiste à donner l'adresse mémoire de ces données. En Fortran, il n'y a pas d'opérateur pour connaître l'adresse d'une donnée. Par contre, les paramètres de chaque appel de fonction sont passés par références, c'est-à-dire par adresse. Pour que la bibliothèque de communication récupère l'adresse d'une donnée, il suffit de la passer en paramètre. Il existe pour cela des appels MPI spécifiques. Si l'environnement d'exécution copie cette donnée au moment du passage de paramètre, c'est l'adresse de la copie qui est alors mémorisée par la bibliothèque de communication. Le problème apparaît lorsque l'environnement décide de détruire la copie après le retour de la fonction appelée. Si la communication n'a pas pu encore être faite, la copie ne sera plus présente en mémoire lorsque la bibliothèque de communication voudra l'envoyer. C'est pour cette raison que la documentation du Parallel Operating Environment (2.3)

[72] “déconseille” l'usage des communications asynchrones en FORTRAN. Nous avons donc utilisé les communications “tamponnées” de MPI.

Annexe B

Les algorithmes de partitionnement de graphe

Nous résumons ici quelques-unes des principales méthodes de partition de graphes non orientés.

Nous limitons souvent la discussion des algorithmes au cas de la partition en deux parties (bissection). Pour effectuer une k partition, si l'extension de l'algorithme à une k partition n'est pas directe, il suffit d'exécuter récursivement l'algorithme de bissection en pondérant les partitions si k n'est pas une puissance de 2.

Les graphes seront notés $G(N, E, W_N, W_E)$, où N est l'ensemble des noeuds, E, l'ensemble des arêtes, W_N, l'ensemble des poids associés à chaque noeud (par défaut 1) et W_E, l'ensemble des poids associés à chaque arête.

Définition 16 *Un ensemble d'arêtes E_c dont la suppression déconnecte le graphe en deux partitions $N = N_1 \cup N_2$ est une* coupe.

Définition 17 *Un ensemble de noeuds N_c dont la suppression déconnecte le graphe en deux partitions $N = N_1 \cup N_c \cup N_2$ est un* séparateur.

Il est possible de passer facilement d'une solution à l'autre, en prenant pour E_c les arêtes incidentes aux noeuds du séparateur N_c et à l'une des deux partitions, ou en prenant pour N_c, les noeuds reliés par les arêtes de la coupe E_c et appartenant à l'une des deux partitions.

Calculer une partition équilibrée minimisant la coupe est un problème NP-Complet [51]. Lorsqu'il suffit de trouver la coupe minimum déconnectant le graphe, le problème *Min Cut/Max Flow* est, lui, polynômial.

Deux types d'informations sont utilisés pour décider du choix de la partition :

– une information géométrique, par exemple des coordonnées, associée aux noeuds.

– la structure du graphe

Ces deux types d'informations définissent deux classes d'algorithmes.

La qualité de la partition est jugée en fonction de deux critères : l'équilibre de la répartition du poids des noeuds dans chaque partition et la taille de la coupe. D'autres critères peuvent également être pris en compte [30] pour des k partitions : taille maximale de la coupe pour une seule partition, degré du graphe des partitions, critères de formes (comme le rapport entre le nombre de noeuds de la plus grande et de la plus petite frontière), critères de "surface".

Quelques résultats théoriques existent, en particulier pour l'existence et taille des séparateurs. Tarjan et Lipton [109] ont montré l'existence d'un séparateur d'une taille de l'ordre de $\sqrt{8\ N}$ pour les graphes planaires. Des généralisations à des espaces à d dimensions ont été étudiés par Miller, Teng, Thurston et Vavasis [89] et Cao, Gilbert et Teng [20].

B.1 Algorithmes de partitionnement géométrique

Il existe de nombreux algorithmes utilisant les "coordonnées" des noeuds. Ils s'appliquent principalement aux problèmes où un noeud est uniquement connecté à ses voisins. C'est souvant le cas de la plupart des maillages utilisés dans des simulations de phénomènes physiques. C'est aussi le cas de la simulation de circulation océanique que nous avons implanté.

Les principales méthodes utilisent souvent des projections. Les noeuds peuvent être séparés en fonction de la coordonnée de leur projection sur une droite [76] : un des axes du repère du modèle ou l'axe d'inertie des noeuds. Ce découpage est utilisé par exemple pour des problèmes de N corps avec rayon de coupure [9].

Il est aussi possible de tirer plusieurs plans de coupe au hasard. Ces plans passent par le point central, c'est-à-dire le point tel que tout plan passant par ce point sépare les noeuds en deux groupes de même taille. Une heuristique de calcul de ce point peut être trouvée dans [20]. La même méthode peut être employée après projection et normalisation du graphe de dimension d sur une sphère de dimension $d + 1$ [89].

Enfin, un autre algorithme standard consiste à calculer le diagramme de Voronoï des noeuds, à partir de k germes dont la position est tirée au hasard, ou bien calculée [41].

B.2 Les algorithmes de partitionnement structurel

Le partitionnement géométrique pourrait être envisagé comme algorithme d'équilibrage de charge avec localité. Malheureusement les algorithmes géométri-

ques n'utilisent pas les informations sur les arêtes du graphe, ce qui est le point qui nous intéresse vraiment, puisque nous essayons de trouver une approche pour tenir compte des coûts de communications.

Les méthodes utilisent les arêtes du graphe pour calculer la coupe. Une première méthode consiste à parcourir intelligemment les noeuds. Un parcours en largeur d'abord permet de faire des partitions niveau par niveau, tandis qu'une agglomération de noeuds [48, 40] autour d'un germe suivant différents critères [53].

Le partitionnement spectral est une approche complètement différente. Elle considère le graphe comme un ensemble de masse (les noeuds) relié par des ressorts (les arêtes). Les valeurs propres de la matrice d'adjacence du graphe correspondent aux fréquences de résonance du graphe. De même les vecteurs propres associés décrivent l'amplitude et la phase du mouvement des masses. La deuxième plus grande valeur propre correspond à une "vague". En observant le signe de chaque composante du vecteur propre associé à un noeud, on partitionne le graphe en deux entre les noeuds sur le sommet ou dans le creux de la vague.

Mais la plupart des algorithmes évoqués précédemment fournissent des solutions grossières au problème de la partition. Il est possible d'améliorer la coupe en utilisant un simple algorithme d'optimisation local comme l'algorithme de Kernighan et Lin [80], amélioré par Fiduccia et Mattheyses [44]. Il consiste simplement à échanger des noeuds se trouvant à la frontière si cela permet d'améliorer la coupe.

Les algorithmes précédents deviennent coûteux lorsque la taille du graphe devient importante. L'approche multi-niveaux permet d'accélérer ces algorithmes. Elle consiste à comprimer le graphe pour travailler sur une approximation grossière du graphe initial puis à utiliser les solutions obtenues sur ces graphes grossiers pour trouver une solution du graphe fin plus rapidement.

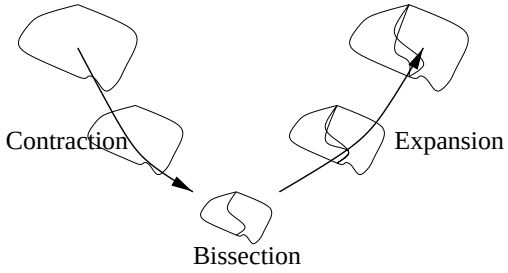


FIG. B.1 – *Méthodes multi-niveaux*

Ces algorithmes multi-niveaux ont été implantés avec succès, que ce soit pour les algorithmes de type Kernighan-Lin (par exemple dans les librairies METIS [62] et SCOTCH [96]) ou pour les méthodes spectrales dans CHACO [66] (CHACO est en fait une collection d'algorithmes de partitionnement) ou [6, 5].

Enfin, pour être complet, il faut noter que des légers raffinements permettent de minimiser des critères annexes, en particulier le degré de chaque partition. Pour cela on peut appliquer des algorithmes de propagation terminale, implantée dans CHACO [66], ou alors utiliser une approche complète de plongement du graphe initial dans un graphe représentant la machine (cf les travaux de François Pellegrini [96]).

Bibliographie

[1] F. Afrati, E. Bampis, L. Finta, and I. Milis. 2 processors scheduling with large communication delays. personnal communication. submitted.

[2] A. Arakawa. Computational design for long term integration of the equations of fluid motions. *J. Comp. Phys.*, 1:119–143, 1966.

[3] B. Baker, E. Coffman Jr., and R. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[4] S. Balay, W. Gropp, L. McInnes, and B. Smith. *Modern Software Tools in Scientific Computing*, chapter Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries, pages 163–202. Birkhauser Press, 1997.

[5] S. Barnard and H. Simon. A fast multilevel implementation of recursive spectral bissection. In *Proceeding of the 6th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1993.

[6] Stephen T. Barnard. Pmrsb: Parallel multilevel recursive spectral bissection. In *Supercomputing 95*, 1995.

[7] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1993. Obtainable at http://www.netlib.org/templates.

[8] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, 53:484–512, 1984.

[9] P.-E. Bernard. *Parallélisation et multiprogrammation pour une application irrégulière de dynamique moléculaire opérationnelle*. Mathématiques appliquées, Institut National Polytechnique de Grenoble, 1997.

[10] P-E. Bernard, T. Gautier, and Denis Trystram. Large scale simulation of parallel molecular dynamics. In *Proceedings of Second Merged Symposium IPPS/SPDP 13th International Parallel Proc essing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, April 1999.

[11] E. Blayo. *Développement d'un modèle quasi-géostrophique de circulation océanique. Application à l'océan Atlantique Nord*. PhD thesis, Université Joseph Fourier, Grenoble I, Laboratoire des Écoulements Géophysiques et Industriels, 1992.

[12] E. Blayo and L. Debreu. Adaptive mesh refinement for finite difference ocean models: first experiments. *J. Phys. Oceanogr.*, 29:1239–1250, 1998.

[13] J. Blazewicz, M. Drabowski, and J. Welgarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transaction on Computing*, C-35(5):389–393, 1986.

[14] J. Blazewicz, K. Ecker, G. Schmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing System*. Springer-Verlag, 1985.

[15] R. Bleck, S. Dean, M. O'Keefe, and A. Sawdey. A comparison of data-parallel and message passing versions of the miami isopycnic coordinate ocean model (micom). *Paral. Comp.*, 21:1695–1720, 1995.

[16] G. Blelloch. Programming parallel algorithms. *CACM*, 39(3):85–97, 1996. url: http://www.cs.cmu.edu/ scandal/nesl.html.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. C. E. Zhou, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, August 1995. url: http://supertech.lcs.mit.edu/cilk.

[18] D. Butenhof. *Programming with POSIX threads*. Addison-Wesley, Reading, MA, USA, 1997.

[19] E. Caceres, F. Dehne, A. Ferreira, and P. Flocchini. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *Lecture Notes in Computer Science*, 1256:390–400, 1997.

[20] F. Cao, J. R. Gilbert, and S.-H. Teng. Partitionning meshes with lines and planes. Technical report csl9601.ps, Parc Xerox,, janvier 1996. found on ftp://parcftp.xerox.com/pub/gilbert/index.html.

[21] A. Carissimi. *ath-0: exploitation de la multiprogrammation légère sur grappes de multiprocesseurs*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et Calcul/Informatique et Distribution, novembre 1999.

[22] P. Chretienne, E. G. Jr. Coffman, and J. K. Lenstra, editors. *Scheduling Theory and Its Applications*. John Wiley & Son Ltd, 1995.

[23] P. Chrétienne, E.G. Coffman, J.K. Lenstra, and Z. Liu. *Scheduling Theory and its Applications*. Wiley, New York, 1995.

[24] P. Chrétienne and C. Picouleau. The basic scheduling with interprocessor communication delays. Technical Report 91.6, MASI, Paris, June 1991.

[25] E.G. Coffman and P.J. Denning. *Operating System Theory*. Prentice Hall, 1972.

[26] M. Cosnard and E. Jeannot. Automatic coarse-grained parallelization techniques. In Kowalik Grandinetti, editor, *NATO workshop: Advances in High Performance Computing*. Kluwer academic, 1997.

[27] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles*. collection IIA. InterEditions, 1993.

[28] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[29] L. Debreu and E. Blayo. On the schwarz alternating method for oceanic models on parallel computers. *J. Comp. Phys.*, 141:93–111, 1998.

[30] R. Diekmann, D. Meyer, and B. Monien. Parallel decomposition of unstructured fem-meshes. In *LNCS*, volume 980. 1995.

[31] J. Dongarra, H.-W. Meuer, and E. Strohmaier. Top500 supercomputer sites. In *Supercomputing 99*, 1999.

[32] J. (Editor) Dongarra, L. Duff, D. Danny, C. Sorensen, and H. van der Vorst. *Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools)*. Society for Industrial & Applied Mathematics, 1999.

[33] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. Algorithm 679: A set of level 3 Basic Linear Algebra Subprograms: Model implementation and test programs. *Trans. on Mathematical Software*, 16(1):18–28, March 1990.

[34] M. Doreille. *Athapascan-1 : vers un modèle de programmation parallèle pour le calcul scientifique*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et Calcul/Informatique et Distribution - IMAG, décembre 1999. url: http://www-apache.imag.fr/.

[35] M. Drozdowski. Scheduling multiprocessor tasks - an overview. *European Journal of Operational Research*, 94:215–230, 1996.

[36] J. Du and J.Y-T. Leung. Complexity of scheduling parallel tasks systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, November 1989.

[37] B. Dumitrescu, M. Doreille, J.-L. Roch, and D. Trystram. Two-dimensional block partitionings for the parallel sparse Cholesky factorization. *Numerical Algorithms*, 16(1):17–38, 1997.

[38] G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz. Compiler and runtime support for programming in adaptive parallel environments. *Scientific Programming, 1997*, January 1997.

[39] A. Fagot. *Réexécution déterministe pour un modèle procédural parallèle basé sur les processus légers*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et Calcul - IMAG, juillet 1997.

[40] C. Farhat and H. D. Simon. Top/domec- a software tool for mesh partitionning and parallel processing. Technical Report RNR-93-011, NASA Ames, 1993.

[41] T. Feder and D. H. Greene. Optimal algorithms for approximate clustering. In *Symposium on Theory of Computing (STOC)*. ACM, 1988.

[42] A. Feldmann, M-Y. Kao, and J. Sgall. Optimal online scheduling of parallel jobs with dependencies. In 25*th Annual ACM Symposium on Theory of Computing*, pages 642–651, San Diego, California, 1993. url: http://www.ncstrl.org, CS-92-189.

[43] W. Fernandez de la Vega and V. Zissimopoulos. An approximation scheme for strip packing of rectangles with bounded dimensions. *Discrete Applied Mathematics*, 82:93–101, 1998.

[44] C. Fiduccia and R. Mattheyses. An effective heuristic procedure for partitionning graphs. Technical Report 82CRD130, General Electric Co., Corporate Research and Development Center, Schenectady, NY, 1982.

[45] MPI Forum. Mpi: A message-passing interface standard. url: http://www-unix.mcs.anl.gov/mpi/index.html, 1995.

[46] MPI Forum. Mpi-2: Extensions to the message-passing interface. electronic version, 1997.

[47] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[48] C. Frahat. A simple and efficient automatic fem domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[49] F. Galilée. *Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et Calcul/Informatique et Distribution - IMAG, septembre 1999. url: http://www-apache.imag.fr/.

[50] M.R. Garey, R.L. Graham, and D.S. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, January 1978.

[51] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman, New York, 1979.

[52] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[53] G. George Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, January 1999.

[54] A. Gerasoulis and T. Yang. DSC : Scheduling parallel tasks on an unbounded number of processors. *IEEE Transaction on Parallel and Distributed Systems*, 5:951–967, 1994.

[55] A. Goldman. *Impact des modèles d'exécutions pour l'ordonnancement en calcul parallèle*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et Calcul/Informatique et Distribution, novembre 1999.

[56] A. Goldman and G. Mounié. Un exemple d'ordonnancement sous le modèle bsp. In Guy-René Perrin Dominique Mery, editor, *10ème Rencontres Francophones du Parallélisme (Renpar'10)*, pages 239–242, june 1998.

[57] A. Goldman, G. Mounié, and D. Trystram. Near optimal algorithms for scheduling independent chains in bsp. In *Fifth International Conference on High Performance Computing (HIPC)*. IEEE, december 1998.

[58] A. Goldman, J. Peters, and D. Trystram. Exchange of messages of different size. In *Irregular'98*, number 1457 in LNCS, Berkeley, USA, september 1998.

[59] G. Golub and C. Van Loan. *Matrix computations*. Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

[60] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell Systems Technical Journal*, 45:1563–1581, 1966.

[61] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[62] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, May 1997.

[63] M. Guyon, M. Chartier, F.-X. Roux, and P. Fraunie. First considerations about modelling the ocean general circulation on mimd machines by domain decomposition method. In F.-X. Le Dimet, editor, *Proceedings of the NATO advanced research workshop on* high perfomance computing in the geosciences, NATO ASI Series C462, les Houches, France, June 1993. Kluwer Academic Publishers.

[64] C. Hanen and A. Munier. An approximation algorithm for scheduling dependant tasks on m processors for small communication delays. In *Proceedings of IEEE Symposium on Emerging Technologies and Factory Automation*, volume 1, pages 167–189. IEEE, October 1995.

[65] C. Hanen and A. Munier. Using duplication for scheduling unitary tasks on m processors with communication delays. *Theoretical Computer Science*, 178:119–127, 1997.

[66] B. Hendrickson and R. Leland. A multilevel algorithm for partitionning graphs. In *Proc. Supercomputing '95*, 1995.

[67] D. Hochbaum, editor. *Approximation Algorithms for Np-Hard Problems*. Pws, September 1996.

[68] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[69] R.W. Hockney. A fast direct solution of poisson's equation using fourier analysis. *J. ACM*, 12:95–113, 1965.

[70] J. Hoogeveen, J.-K. Lenstra, and B. Veltman. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters*, 16:129–137, 1994.

[71] Jing Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.

[72] IBM. *README for the Parallel Operating Environment fileset, ppe.poe, of IBM Parallel Environment for AIX, Version 2 Release 3*.

[73] I.D.Scherson, R.Subramanian, V. L. M. Reis, and L. M. Campos. Scheduling computationally intensive data parallel programs. In *Placement dynamique et répartition de charge : application aux systèmes parallèles et répartis (École Française de Parallélisme, Réseaux et Système)*, pages 107–129. Inria, July 1996.

[74] IEEE. *IEEE 1003.1c-1994: Standard for Information Technology — Portable Operating System Interfaces (POSIX) — Part 1: System Application Programm Interface (API) — Amendment 2: Threads Extension [C langage]*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994.

[75] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms (SODA 98)*, pages 490 – 498, San Francisco, CA USA, January 25 - 27 1998.

[76] Mark T. Jones and Paul E. Plassmann. Parallel algorithms for adaptative refinement and partitionning of unstructured meshes. In *Proceedings of the Scalable High Performance Computing Conference*, pages 478–485. IEEE Computer Society Press, 1994.

[77] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. Williams. Array design and expression evaluation in

pooma ii. In *Lecture Notes in Computer Science*, volume 1505, page 231. Springer-Verlag, 1998.

[78] C. Kenyon and E. Rémila. Approximate strip packing. In *37th Annual Symposium on Foundations of Computer Science*, pages 31–36, Burlington, Vermont, 14–16 October 1996. IEEE.

[79] C. Kenyon and E. Rémila. A near-optimal solution to a two-dimensional cutting stock problem. Technical Report 1998-48, Laboratoire de l'Informatique du Parallélisme, october 1998. URL : http://www.ens-lyon.fr/LIP/Pub/rr.html.

[80] B. Kernighan and S. Lin. An effective heuristic procedure for partitionning graphs. *The Bell System Techical Journal*, pages 291–308, feb 1970.

[81] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming With Threads*. SunSoft Press, Mountainview, CA, USA, 1995.

[82] I. Kort. *Ordonnancement et Modèles d'Exécution : cas de graphes spécifiques*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et Calcul - IMAG, juillet 1998.

[83] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[84] Bil Lewis and Daniel J. Berg. *Threads primer: a guide to multithreaded programming*. SunSoft Press, Mountainview, CA, USA, 1996.

[85] Bil Lewis and Daniel J. Berg. *Multithreaded programming with pthreads*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA, 1998.

[86] W. T. Ludwig. Algorithms for scheduling malleable and nonmalleable parallel tasks. Technical Report CS-TR-95-1279, University of Wisconsin, Madison, August 1995.

[87] W. T. Ludwig. *Algorithms for scheduling malleable and nonmalleable parallel tasks*. PhD thesis, University of Wisconsin - Madison, Department of Computer Sciences, 1995.

[88] W. T. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In Daniel D. Sleator, editor, *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 167–176, Arlington, VA, January 1994. ACM Press.

[89] G. Miller, S.-H. Teng, W. Thurston, and S. Vavasis. *Graph Theory and Sparse Matrix Computation*, chapter Automatic mesh partitionning. Springer-Verlag, 1993.

[90] G. Mounié, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Eleventh ACM Symposium on*

Parallel Algorithms and Architectures (SPAA'99), pages 23–32. ACM, juin 1999.

[91] R. Namyst and J.-F. Méhaut. PM^2: Parallel multithreaded machine. A computing environment for distributed architectures. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, editors, *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*, pages 279–285, Amsterdam, February 1996. Elsevier, North-Holland.

[92] S Oaks and H. Wong. *Java Threads*. Java Series. O'Reilly & Associates, 2nd edition, 1999.

[93] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[94] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, April 1990.

[95] J. Pedlosky. *Geophysical fluid dynamics*. Number 710. Springer-Verlag, 1987.

[96] F. Pellegrini. *Application de méthodes de partition à la résolution de problèmes de graphes issus du parallélisme*. PhD thesis, LaBRI, Bordeaux, France, janvier 1995.

[97] G. N. Srinivasa Prasanna and B. R. Musicus. Generalized multiprocessor scheduling and applications to matrix computations:. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):650–664, June 1996.

[98] G. N. Srinivasa Prasanna and B. R. Musicus. The optimal control approach to generalized multiprocessor scheduling. *Algorithmica*, 15(1):17–49, 1996.

[99] L Prylli, B Tourancheau, and R Westrelin. Bip: a new protocol designed for high performance networking on myrinet. In *IPPS/SPDP98*, Orlando, USA, 1998.

[100] C. Rapine. *Algorithmes d'approximation garantie pour l'ordonnancement de tâches. Application au domaine du calcul parallèle.* Thèse d'informatique, Institut National Polytechnique de Grenoble, janvier 1999.

[101] C. Rapine, I. Scherson, and D. Trystram. On-line scheduling of parallelizable jobs. *Lecture Notes in Computer Science*, 1470:322–327, 1998.

[102] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[103] M. Rinard. *The design, implementation and evaluation of Jade : a portable, implicitly parallel programming language*. PhD thesis, Stanford University, 1994. url: http://suif.stanford.edu/ martin/jade/index.html.

[104] A. Robinson, P. Lermusiaux, and N. Quincy Sloan. data assimilation. *The Sea*, 10:541–594, 1998.

[105] N. Smith and M. Lefebvre. The global ocean data assimilation experiment. In *International Symposium: Monitoring the Oceans in the 2000s: An Integrated Approach*, Biarritz, France, October 1998. URL:http://www.bom.gov.au/bmrc/mrlr/nrs/oopc/godae/biarritz.htm.

[106] R.D. Smith, J.K. Dukowicz, and R.C. Malone. Parallel ocean general circulation modeling. *Physica D*, 60:38–61, 1992.

[107] G. N. Srinivasa Prasanna and B. R. Musicus. Generalised multiprocessor scheduling using optimal control. In 3*rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 216–228. ACM, 1991.

[108] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, 1997.

[109] R. Tarjan and R. Lipton. A separator theorem for planar graphs. *SIAM Journal of Applied Mathematics*, 36:177–189, avril 1979.

[110] J. Turek, J. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In 4*th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, 1992.

[111] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[112] A.J. Wallcraft and D.R. Moore. The nrl layered ocean model. *Paral. Comp.*, 23:2227–2242, 1997.

[113] L. Zaslavsky, S. Kahan, B. Elton, K. Maschhoff, and L. Stern. A scalable approach for solving irregular sparse linear systems on the tera mta multithreaded parallel shared-memory computer. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, march 1999. url: http://www.tera.com/www/library.