Approches algorithmiques pour l'ordonnancement d'applications parallèles avec communications

Renaud Lepère

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE

pour obtenir le titre de

Docteur de l'INPG

Spécialité : **«Informatique : Systèmes et Communications »**

préparée au **Laboratoire Informatique et Distribution**

dans le cadre de l'**École Doctorale «Mathématiques et Informatique»**

présentée et soutenue publiquement

par

Renaud Lepère

le 6 octobre 2001

Approches algorithmiques pour l'ordonnancement d'applications parallèles avec communications

---

Directeur de thèse

M. Denis Trystram

Jury

M. Philippe Jorrand,	Président
Mme. Alix Munier,	Rapporteur
M. Klaus Jansen,	Rapporteur
M. Denis Trystram,	Directeur de thèse
M. Ed Coffman,	Examinateur
M. Frédéric Guinand,	Examinateur

Remerciements

Je remercie Denis Trystram pour sa confiance, son soutien et son suivi pendant toute ma thèse. Travailler avec lui a été pour moi une expérience très agréable, sympathique et fructueuse. Je remercie Brigitte Plateau pour m'avoir accueilli au sein du laboratoire ID-IMAG ainsi que tous les membres du laboratoire pour leur accueil chaleureux et les nombreuses discussions sympathiques à la Cafet.

Je remercie mes rapporteurs Alix Munier et Klaus Jansen ainsi que Frédéric Guinand pour leurs lectures attentives et leurs suggestions. Je remercie également les autres membres du jury Ed Coffman et Philippe Jorrand pour avoir partagé avec moi l'aboutissement de mon travail de thèse. Je les remercie aussi pour l'intérêt qu'ils ont manifesté pour ce travail.

Je tiens à remercier également les autres personnes avec qui j'ai pu collaboré au cours de cette thèse et notamment Borut Robič pour m'avoir accueilli en Slovénie, Christophe Rapine qui était toujours partant pour aller travailler dans un café, Gerhard Woeginger pour nos collaborations électroniques fructueuses, enfin Gregory Mounié et Rémi Revire pour le travail agréable que nous avons mené ensemble. J'ai eu beaucoup de plaisir à travailler avec eux.

Je souhaiterais remercier mes collègues de bureau Ekbel, François, Gilles, Anne et Arnaud et plus généralement toutes les personnes que j'ai côtoyées au labo pour leur gentillesse et pour les nombreuses discussions que nous avons partagé. Merci aussi à tous les joueurs avec qui j'ai eu l'occasion de m'amuser pendant ces trois années et notamment aux poseurs de bombes Alfredo, Chris, Gerson, Gustavo, Greg, Olivier et Mathias, aux joueurs de Go Alexis, Timothée et surtout Gilles pour nous avoir initiés à ce jeu, et aux amateurs de Starcraft, pour ne pas faire de délations (certains ont préféré garder l'anonymat) je ne citerais que leurs pseudos (Shhht, Crics, Geo, Remi, Simon, Gilles, Pfd, God, Alex, Mat). Un dernier mot enfin pour ma petite famille qui a supporté patiemment les soirées de rédaction et qui m'a encouragé jusqu'au bout.

Table des matières

Introduction

Mener à bien des simulations numériques, des prévisions météorologiques ou encore créer des images de synthèse nécessitent des ressources de calculs et de mémoire sans cesse croissantes. La précision et la complexité des différents modèles sur lesquels ces applications sont basées augmentent régulièrement. Depuis longtemps, le parallélisme apparaît comme une solution pour répondre à cette forte demande de performances. Bien que les performances des machines séquentielles doublent tous les 18 mois, l'utilisation de machines parallèles pourrait permettre de gagner plusieurs ordres de grandeurs de performances. Intuitivement, on peut penser qu'un programme pourra s'exécuter m fois plus rapidement sur m machines que sur une seule. Cependant le développement d'applications sur des machines parallèles reste délicat.

Les machines parallèles évoluent rapidement, et les supports d'exécution ont des propriétés distinctes. Les machines peuvent, par exemple, disposer d'une mémoire partagée physique pour permettre aux différentes unités de calculs de coopérer. Dans d'autres cas, les machines parallèles sont construites autour d'unités de calcul dotées de leur propre mémoire. Ces unités communiquent entre elles par échange de messages ; on parle aussi de machines à mémoire distribuée. Aujourd'hui, on assiste au développement rapide de machines parallèles de ce type : les grappes. Elles sont construites à partir de l'interconnexion de composants «standard» donc bon marché, comme les stations de travail ou les ordinateurs personnels (PC). Ces architectures, pouvant être constituées de plusieurs centaines de processeurs, offrent des performances potentiellement exceptionnelles pour un prix modeste. Cependant, leur exploitation reste très délicate et pose encore de très nombreux problèmes.

Les difficultés sont multiples et de natures différentes. Indépendamment de l'hétérogénéité et de l'évolution rapide des machines parallèles, les difficultés se situent au niveau algorithmique, au niveau des langages de programmation parallèle pour exprimer le parallélisme d'une application, ainsi qu'au niveau de l'ordonnancement et du placement des différents calculs d'une application parallèle sur les unités de calculs, les processeurs.

Actuellement, il n'existe aucune approche générale pour développer une application parallèle. En pratique, sur une architecture à mémoire distribuée, l'approche la plus répandue consiste à utiliser directement une bibliothèque bas niveau d'échange de messages (comme MPI ou PVM). Dans ce cas, la décision d'ordonnancement reste entièrement à la charge du programmeur : il doit explicitement spécifier l'ordonnancement de son application, c'est-à-dire où et quand exécuter les différents calculs.

Même si ce type d'approche reste tout à fait envisageable pour des applications régulières ou des applications basées sur un parallélisme de données, il apparaît très souhaitable que la tâche complexe de l'ordonnancement incombe à un compilateur ou à un environnement d'exécution. Dans cette optique, plusieurs langages et bibliothèques de haut niveaux ont été développés [Fu97, CDGR98]. L'application parallèle est alors décrite indépendamment de l'architecture sous-jacente et une représentation fine de l'application peut être construite. Cette construction est habituellement appelée extraction du parallélisme, elle peut avoir lieu avant l'exécution ou au cours de celle-ci. La représentation fine obtenue décrit les différents calculs effectués par l'application qui sont aussi appelés tâches et elle offre une vision des dépendances entre ces calculs. Elle est habituellement décrite à l'aide d'un *graphe de précédence* ou d'un *graphe de flot de données*. Un ordonnanceur fin tirant parti de cette connaissance peut alors être utilisé, soit dynamiquement à l'exécution, soit statiquement à la compilation. Cet ordonnanceur doit réaliser le lien nécessaire avec la machine cible. Il doit décider où et quand exécuter les différentes tâches de l'application, tout en tenant compte des ressources et des spécificités de la machine cible.

L'une des caractéristiques des architectures actuelles comme les grappes de PC est le temps important des communications, et notamment le temps des latences. Lors de l'exécution d'une application parallèle, les unités de calculs ont souvent besoin de communiquer entre elles pour partager des données ou des résultats. La prise en compte de ces communications dans la décision d'ordonnancement apparaît comme un facteur clé pour permettre des exécutions efficaces sur ce type d'architecture.

Pour prendre en compte ces communications, différents modèles ont été proposés. Un point essentiel pour la réalisation d'algorithmes d'ordonnancement utilisables en pratique est la validité de ces modèles. En effet, le modèle doit être suffisamment réaliste pour que l'ordonnancement obtenu puisse être efficacement exécuté sur la machine cible. Les surcoûts imprévus par le modèle doivent donc être limités. Pour prendre en compte les communications, deux grandes approches sont envisageables. La première consiste à modéliser les coûts directs qu'induisent les communications. Le plus connu, mais aussi le plus simple, de ces modèles est le *modèle délai*, il permet de prendre en compte une latence de communication lorsque deux tâches désirant échanger un résultat sont placées sur des processeurs différents. La seconde approche, plus récente est fondée sur une prise en compte implicite des communications. Elle s'appuie sur un modèle d'application où les tâches qui la composent sont elles-mêmes des activités parallèles pouvant s'exécuter sur un nombre variable de processeurs avec une efficacité variable dépendant notamment des coûts de communications et du parallélisme intrinsèque de la tâche. Il s'agit du modèle des *tâches malléables*. Ce modèle se base sur une vision structurée et à deux niveaux d'une application parallèle.

Dans le cadre de cette thèse, je me suis intéressé aux problèmes d'ordonnancement sous ces différents modèles. L'objectif est de placer et d'ordonner judicieusement les différentes tâches afin de minimiser le temps d'exécution de l'application. Il s'agit

d'éviter de trop nombreuses communications et de trouver un bon compromis entre l'utilisation des ressources de calculs et les communications. Le problème de l'ordonnancement dans le contexte du calcul parallèle est ainsi un problème d'optimisation. Sauf dans des cas bien particuliers, les problèmes d'ordonnancement sous les modèles prenant en compte les communications (que ce soit les modèles délai ou le modèle des tâches malléables) sont $\mathcal{NP}$-difficiles. Il apparaît donc très peu probable de pouvoir trouver un algorithme polynômial pour résoudre optimalement ceux-ci. Une solution naturelle consiste à se tourner vers des heuristiques qui calculent une «bonne» solution réalisable mais non nécessairement optimale.

L'évaluation de la qualité d'une heuristique apparaît alors comme un point important. Une première approche consiste à utiliser des jeux d'essais. Différentes instances sont générées, et l'heuristique peut alors être comparée à d'autres sur la base de ces instances. Cependant la réponse apportée n'est alors que partiellement satisfaisante, la validité des jeux d'essais restant souvent très subjective. La notion *d'algorithmes d'approximation* apporte une réponse intéressante au problème de la qualité d'une heuristique. Un algorithme est une k-approximation (d'un problème de minimisation) si pour toute instance du problème la solution réalisable obtenue n'excède pas la solution optimale d'un rapport au plus k. On parle aussi de la *garantie de performance* de l'algorithme. Dans le cadre de cette thèse, j'ai essayé de privilégier la recherche d'algorithmes d'approximation.

Le plan de ce document est le suivant. Dans le premier chapitre, nous décrivons en détail les notions, les concepts et les modèles que nous utiliserons. Nous décrivons comment représenter finement une application parallèle et notamment la notion de graphe de précédence. Il s'agit d'une représentation fine d'une application que nous utiliserons tout au long de cette thèse. Puis, nous décrivons les machines parallèles sur lesquelles nous envisageons d'exécuter des applications parallèles. Nous étudions en détail les modèles à communications explicites et le modèle des tâches malléables, qui permettent une prise en compte des communications dans la décision d'ordonnancement. Nous nous intéressons notamment à la validité pratique de ces modèles.

Dans le second chapitre, nous présentons un état de l'art des différents résultats obtenus pour les problèmes d'ordonnancement prenant en compte les communications. Nous nous intéressons à la complexité de ces problèmes, aux algorithmes d'approximation pour ces problèmes et aux heuristiques utilisées dans la pratique. Nous présentons notamment les résultats obtenus pour les problèmes d'ordonnancement sous les modèles de type délai car ceux-ci ont été largement étudiés. Puis nous terminons par les résultats connus pour l'ordonnancement sous le modèle des tâches malléables et sous des modèles proches.

Les trois chapitres suivants présentent mes contributions au domaine. Le troisième chapitre décrit nos travaux autour de l'ordonnancement sous le modèle des tâches malléables. Nous nous sommes intéressés au problème de l'ordonnancement avec des contraintes de précédences sous le modèle des tâches malléables et sur un nombre fixé de processeurs m. Nous avons utilisé une approche en deux phases. Dans

un premier temps, nous calculons une allocation, c'est à dire un nombre de processeurs pour calculer chacune des tâches malléables. Dans un deuxième temps, nous utilisons un algorithme d'ordonnancement de tâches multiprocesseurs (tâches parallèles devant s'exécuter sur un nombre fixé de processeurs) pour calculer dans quel ordre exécuter les différentes tâches. La première phase est basée sur une relaxation du problème initial. La contrainte sur le fait qu'au plus m processeurs travaillent à un instant donné est relâchée, et une nouvelle contrainte plus faible est introduite ; elle exprime le fait que le travail total ne doit pas être trop important. Cette approche nous a permis d'obtenir des algorithmes d'approximation offrant une garantie constante pour ce problème et pour des graphes de précédence de type «arbres» mais également pour des graphes de précédence quelconques. Ces résultats sont le fruit d'un travail mené en collaboration avec Gregory Mounié, Denis Trystram, Borut Robic et Gerhard Woeginger.

Dans les deux derniers chapitres, nous nous sommes intéressés aux problèmes d'ordonnancement sous des modèles à communications explicites. Le modèle que nous avons considéré est un modèle délai similaire à celui introduit par Papadimitriou et Yannakakis [PY90]. Toutes les tâches ont une même durée unitaire et les communications sont modélisées par un délai uniforme ρ potentiellement grand. Si deux tâches x et y liées par une relation de précédence sont ordonnancées sur des processeurs différents, alors la tâche y ne pourra commencer à s'exécuter que ρ unités de temps après la fin de la tâche x. Par contre si les tâches x et y sont placées sur le même processeur alors la tâche y pourra s'exécuter immédiatement après la terminaison de la tâche x. L'ordonnancement sous ce modèle est aussi appelé «problème d'ordonnancement avec grand temps de communication».

Le quatrième chapitre est une approche originale de ce problème, elle est basée sur une *décomposition récursive* du graphe de précédence. Nous réfléchissions au départ sur le problème de la construction du graphe de précédence décrivant une application sous le modèle des tâches malléables. Notre décomposition récursive du graphe de précédence est fondée sur une découpe consistant à trouver deux groupes de tâches indépendants (c'est-à-dire tels que chacune des tâches de l'un n'ait aucune relation de précédence avec chacune des tâches de l'autre) et tel que ces groupes de tâches soient les plus grands possibles. Ces deux groupes de tâches peuvent être exécutés de manière efficace en parallèle. Cette idée permet ainsi de construire une décomposition du graphe de précédence. Nous n'avons pas réussi à trouver de garantie de performances pour cette approche. Cependant nous avons expérimenté cette nouvelle heuristique sur quelques graphes d'applications réelles, en la comparant avec un algorithme classique pour le problème du regroupement DSC (Dominent Sequence Clustering) [GY92]. Les premiers résultats sont particulièrement encourageants. Au vu de ceux-ci, nous souhaitons maintenant valider notre approche sur des applications réelles. En collaboration avec Rémi Revire, un travail dans cette voie est en cours actuellement. Il s'agit d'intégrer notre approche et d'autres heuristiques d'ordonnancement statiques dans Athapascan1, l'environnement de programmation parallèle développé au ssin du projet Apache.

Dans le dernier chapitre, nous nous sommes intéressés au problème de l'ordonnancement avec grand temps de communication et avec *duplication*. La duplication, c'est-à-dire l'exécution d'une même tâche sur des processeurs différents, peut permettre de limiter les communications et apparaît comme une approche intéressante pour les problèmes d'ordonnancement avec délai de communications. En effet, si une tâche a plusieurs successeurs (c'est à dire si une tâche doit communiquer une donnée à beaucoup d'autres tâches pour qu'elles puissent s'exécuter), alors il peut être utile d'exécuter cette tâche sur différents processeurs afin de pouvoir commencer à exécuter des tâches successeurs plus tôt. Dans ce cadre, nous avons montré qu'il était possible de réduire le problème initial à celui de l'ordonnancement d'une succession de «petits graphes». Cette approche nous a permis de construire un algorithme d'approximation avec une garantie asymptotique en $\mathcal{O}(\log(\rho))$ pour ordonnancer une application décrite par un graphe de précédence arbitraire. Ce résultat améliore sensiblement les résultats déjà connus. Il est le fruit d'une étroite collaboration avec Christophe Rapine.

CHAPITRE

1

Motivation, modèles et concepts

Dans ce chapitre, nous allons présenter et détailler les notions, les concepts et les modèles que nous utiliserons au long de cette thèse. Nous présentons d'abord différentes façons de modéliser une application parallèle dont les graphes de précédence et les graphes de flot de données et introduisons le vocabulaire associé à ces représentations. Nous présentons ensuite les différentes architectures parallèle et notamment les architectures émergentes comme les grappes de PC. Nous verrons que celles-ci sont souvent caractérisées par des temps de communication importants. L'ordonnancement est la décision qui consiste à choisir où et quand exécuter les calculs effectués par une application. Prendre en compte les coûts des communications dans la décision d'ordonnancement apparaît comme un facteur clé pour l'exploitation efficace des architectures parallèles actuelles. Nous détaillons donc les différents modèles qui ont été proposés pour prendre en compte ces coûts de communications.

Sommaire

Ce chapitre a pour objectif de clarifier les notions, les concepts et les modèles que nous utiliserons tout au long de cette thèse. Il permet également de bien comprendre l'intérêt pratique des problèmes d'ordonnancement que nous abordons dans cette thèse.

Tout d'abord, nous présentons comment modéliser une application parallèle. Nous détaillons notamment les notions de *graphe de précédence* et de *graphe de flot de données* ainsi que le vocabulaire associé à ces représentations. Ces représentations offrent une vision fine des calculs effectués par une application. Les problèmes auxquels nous nous intéressons dans cette thèse consistent à choisir un lieu (processeur) et un ordre d'exécution pour ces calculs. C'est cette décision que nous appelons *ordonnancement*. Dans notre cadre, l'objectif de l'ordonnancement est de prendre une décision judicieuse afin de minimiser la durée totale de l'exécution de l'application parallèle. Nous nous intéresserons plus particulièrement aux algorithmes d'ordonnancement statiques, c'est à dire ceux qui tirent parti de la connaissance fine de l'application à l'aide d'une représentation appropriée de type graphe de précédence ou graphe de flot de données. Ces représentations fines d'une application correspondent en général au résultat d'une phase dite d'extraction du parallélisme. Nous nous intéressons également aux questions liées à la construction d'une telle représentation.

Pour réaliser un ordonnancement efficace d'une application, il apparaît aussi nécessaire d'avoir une description fine des architectures cibles, afin d'identifier les paramètres clés de ces architectures. Par la suite, nous présentons les différentes caractéristiques des machines parallèles sur lesquelles on peut envisager d'exécuter des applications parallèles. Nous nous intéressons notamment à l'évolution de celles-ci. Nous verrons que très souvent les architectures actuelles comme les grappes de PC sont caractérisées par des temps de communication liés aux échanges de messages pouvant être très importants. Il apparaît nécessaire de prendre en compte ces coûts de communication dans la décision d'ordonnancement. Pour se faire, différents modèles ont étés proposés. Deux approches sont principalement envisageables. La première consiste à modéliser explicitement les communications en considérant les coûts directs qu'elles induisent. La seconde approche est fondée sur une prise en compte implicite des communications. Elle s'appuie sur un modèle d'application où les tâches qui la composent sont elles-mêmes des activités parallèles pouvant s'exécuter sur un nombre variable de processeurs avec une efficacité variable dépendant notamment des coûts de communications et du parallélisme intrinsèque de l'application. Il s'agit du modèle des *tâches malléables*. Nous présentons en détail ces deux modèles.

1. Modélisation d'une application

Afin de pouvoir paralléliser et ordonnancer une application parallèle, il est nécessaire d'utiliser une représentation de cette application. Cette représentation doit exprimer le parallélisme présent dans cette application. Nous détaillons ci-dessous deux représentations très couramment utilisées, d'une part le graphe de précédence et d'autre part le graphe de flot de données qui offre une vision plus fine de l'application. Par la suite nous verrons quand et comment construire ces représentations.

1.1. Graphe de précédence

Dans la représentation par graphe de précédence, l'application est divisée en un ensemble fini d'unités élémentaires, les tâches. Ces tâches peuvent correspondre à des blocs d'instructions ou à des fonctions. Une tâche dispose en entrée d'un ensemble de données et produit un ensemble de résultats en sortie. Lorsque toutes les données nécessaires à l'exécution d'une tâche sont disponibles, cette tâche peut s'exécuter. Cette exécution dure un temps fini, sans interaction avec l'extérieur, et produit un ensemble de résultats. Lorsqu'une tâche a besoin du résultat d'une autre tâche pour s'exécuter on dit que les deux tâches sont liées par une relation de précédence. Un graphe de précédence $G = (V, E)$ permet de décrire ces relations ; il s'agit d'un graphe orienté et acyclique dans lequel les sommets représentent les tâches de l'application et les arcs représentent les relations de précédence entre les différentes tâches. La figure 1 illustre le graphe de précédence de l'algorithme 1, un algorithme de programmation dynamique très classique qui calcule la distance entre les deux mots A et B. Il s'agit d'un exemple jouet mais qui permet de bien visualiser la notion de graphe de précédence.

En général, la description d'une application par un graphe de précédence peut être plus ou moins fine. Cette notion correspond à la notion de *granularité* du graphe. On dit qu'il s'agit d'un graphe à *grain fin*, lorsque les tâches correspondent à des blocs d'instructions de courte durée d'exécution, comme par exemple quelques instructions. Lorsque les tâches sont de plus grandes durées comme par exemple dans le cas de tâches représentant des programmes entiers, on parle de graphe à *gros grain*.

Dans toute la suite de cette thèse, par souci de simplicité, nous confondrons la tâche et le sommet qui représente cette tâche dans le graphe de précédence. En pratique les tâches d'un graphe de précédence peuvent être et sont souvent étiquetées par une durée d'exécution. Se pose alors le problème de calculer et évaluer avec

Algorithme 1 `CompareMot`(A, B)

```
// distance[i, j] est la distance entre les mots A[1···i] et B[1···j]
// Initialisation de distance[i, 0] et distance[0, j]
pour i = 1 à Longueur(A) faire
  pour j = 1 à Longueur(B) faire
    distance[i, j] = min(  distance[i − 1, j − 1] + CompareLettre(A[i], B[j]),
                           distance[i − 1, j] + CoutAjout(A[i]),
                           distance[i, j − 1] + CoutAjout(B[j]));
  fin pour
fin pour
return distance[Longueur(A), Longueur(B)];
```

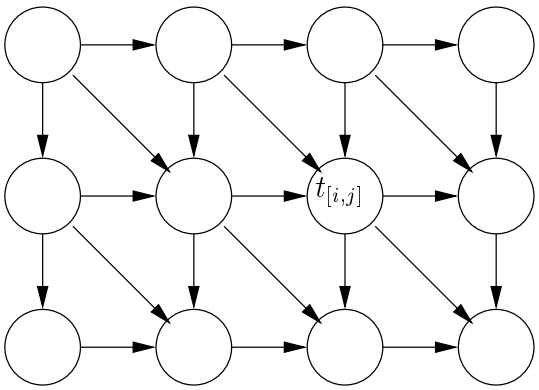

Figure 1 – Le graphe de précédence de l'algorithme 1 pour un mot de longeur 3 et un mot de longueur 4. Chaque sommet $t_{[i,j]}$ de ce graphe représente un calcul de $distance[i, j]$. Les arcs représentent les dépendances dus aux accès aux données. Ici pour calculer $distance[i, j]$, il faut avoir déja calculé $distance[i, j-1]$, $distance[i-1, j]$ et $distance[i-1, j-1]$.

précision cette durée. Les arcs d'un graphe de précédence peuvent être également étiquetés pour représenter les tailles des données. Pour un graphe de précédence G, nous adoptons le vocabulaire suivant :

Prédécesseur Une tâche x est un prédécesseur d'une tâche y si et seulement si il existe un arc (x, y) dans le graphe G.

Successeur Une tâche x est un successeur d'une tâche y si et seulement si la tâche y est un prédécesseur de la tâche x.

Ancêtre Une tâche x est un ancêtre de la tâche y si et seulement si il existe un chemin orienté reliant x à y dans le graphe G.

1.2. Graphe de flot de données

Une application peut être représentée plus finement grâce à un graphe de flot de données. Celui-ci est également basé sur un découpage de l'application en un ensemble de tâches. Cependant, un graphe de flot de données prend en compte plus finement les accès réalisés sur les données partagées. Un graphe de flot de données G est un triplet (V, D, E). V est un premier ensemble de sommets, il correspond aux tâches. D est un second ensemble de sommets, il correspond aux données partagées. L'ensemble E est un ensemble d'arcs, il décrit les accès réalisés par les tâches sur les données partagées. S'il existe un arc entre une tâche x et une donnée d, alors cela signifie que la tâche x réalise un accès en lecture à la donnée d. S'il existe un arc entre une donnée d et une tâche x, alors cela signifie que la tâche x réalise un accès en écriture à la donnée d. Ainsi E ne décrit pas les relations de précédence entre les tâches, cependant celles-ci peuvent se déduire à partir des accès réalisés sur les données. La figure 5 page 25 représente deux graphes de flot de données différents qui correspondent au même graphe de précédence.

L'utilisation d'une représentation fine, avec graphe de flot de données d'une application parallèle, pour l'ordonnancement n'est utile que sous des modèles complexes comme LogP, nous le verrons dans le paragraphe 3.1.2. Cela explique pourquoi les algorithmes d'ordonnancement proposés dans la littérature se fondent très souvent sur des graphes de précédence.

1.3. Construction de la représentation d'une application

En pratique, une question importante concerne la construction d'une représentation d'une application parallèle. Cette représentation est essentielle pour pouvoir ordonnancer finement l'application. Elle est souvent le résultat d'une phase dite d'*extraction du parallélisme*. La détection du parallélisme est obtenue par analyse des dépendances de données, et comme nous allons le voir, elle peut être effectuée soit par compilation, soit dynamiquement au cours de l'exécution.

Une première approche consiste à rechercher directement le parallélisme présent dans un code séquentiel. On parle dans ce cas de parallélisation automatique. La parallélisation automatique d'un code séquentiel est un problème difficile qui n'est pas pleinement résolu à ce jour [Fea95, DRV00]. Pour l'instant cette technique est surtout utilisée pour des structures de boucles imbriquées dont les indices sont des fonctions affines, ces structures de boucle sont appelées les nids de boucle [BDSV98].

Une seconde approche est d'utiliser des langages permettant de décrire explicitement cette représentation. Dans ce cas là granularité des tâches est choisie par le programmeur. PYRROS est, par exemple, un outil de programmation parallèle dont l'objectif est de générer un code MPI utilisable sur des architectures distribuées. PYRROS utilise un langage permettant de décrire explicitement des graphes de précédence dont la taille est fixée à la compilation [YG92]. Le programme `make`, outil standard (IEEE) utilisé pour la compilation modulaire, est un autre exemple. Les fichiers «Makefile», permettent de décrire une structure de graphe de précédence à gros grain. Cette structure permet d'une part, à l'outil `make` de n'effectuer que la recompilation des modules nécessaires après une modification, d'autre part il permet d'effectuer des compilations parallèles sur des machines multiprocesseurs.

La faiblesse de cette approche basée sur une description explicite du graphe de précédence de l'application est liée à son caractère statique. En effet, elle ne permet pas de décrire une application dont le parallélisme dépend des données en entrées, comme cela est souvent nécessaire. Ainsi le graphe de précédence d'une élimination de Gauss dépend de la taille des matrices ; le parallélisme présent dans une factorisation de matrice creuse dépend lui aussi de la structure de cette matrice.

Une dernière approche consiste à laisser à un système d'exécution le soin de découvrir le parallélisme d'une application au cours de son exécution. Plusieurs langages et librairies rentrant dans ce cadre ont été proposés. C'est la cas de Jade [RL98] ou d'Athapascan1 [CDGR98]. Je détaille maintenant plus précisément quelques idées développées dans Athapascan1, la librairie C++ parallèle développée au sein du projet Apache. En Athapascan1, la granularité des tâches et des données est explicite. Le programme principal et les tâches elles-mêmes peuvent récursivement créer de nouveaux objets partagés et de nouvelles tâches. Les tâches sont des objets dont une fonction membre permet l'exécution. Le parallélisme lui est implicite. Les relations de précédence entre les différentes tâches sont déduites des accès (lecture, écriture) effectués par les tâches sur les objets partagés. Le graphe de flot de données de l'application est alors construit dynamiquement et évolue avec l'exécution de l'application. Il offre une vision fine du futur de l'exécution. Il est ainsi possible de «dérouler» en partie l'application pour utiliser ensuite un ordonnanceur statique, c'est à dire un ordonnanceur tirant partie de la connaissance fine de l'application. Doreille a montré dans sa thèse que cette approche était très prometteuse [Dor99]. Il l'a notamment illustrée pour une factorisation numérique creuse de Cholesky dont le parallélisme dépendait fortement de la structure de la matrice creuse. Les questions que posent cette approche sont liées aux sur-coûts à l'exécution engendrés par cette analyse.

2. Description des différentes machines parallèles

Pour ordonnancer finement une application parallèle, il est également nécessaire de bien identifier les paramètres clés qui reflètent les architectures parallèles sur lesquelles on envisage d'exécuter des applications. L'objectif est de proposer ensuite des modèles réalistes prenant en compte ces paramètres. Pour cela, nous allons d'abord décrire les caractéristiques importantes des architectures parallèles. Historiquement, l'une des premières classifications des machines parallèles est celle proposée par Flynn [Fly66]. Il classe les machines parallèles selon le fait qu'elles disposent de un ou plusieurs *flots d'exécution* et de un ou plusieurs *flots de données*. Il distingue ainsi les machines SISD, SIMD, MIMD qui en anglais signifient respectivement «single instruction single data», «single instruction multiple data» et «multiple instruction multiple data». Une machine de type MIMD est constituée d'un ensemble d'unités de calcul. Chaque unité de calcul exécute des instructions qui lui sont propres.

Cette classification qui a eu son intérêt en son temps, est aujourd'hui obsolète, la plupart des machines parallèles sont du type MIMD. On distingue aujourd'hui d'une part les architectures à mémoire partagée et d'autre part les architectures à mémoire distribuée. Les architectures à mémoire partagée de type multiprocesseurs symétriques (SMP) sont des machines aujourd'hui très courantes et très populaires. Elles offrent une mémoire partagée dont le temps d'accès est uniforme (UMA). Cela facilite grandement la programmation, puisqu'il suffit simplement de lire ou écrire en mémoire pour échanger des données en prenant soin tout de même d'utiliser des primitives de synchronisation, sans a priori se soucier de la localité des données. L'interface de programmation Posix fournit un ensemble de primitives pour la programmation efficace de ce type de machine [Pth95]. Cependant la conception technique de machines SMP de grande taille pose des difficultés, notamment au niveau de la gestion de la cohérence de cache [Ste90]. D'autre part, dans un tel dispositif, la mémoire apparaît au centre du dispositif et l'accès à celle-ci est alors plus coûteux. Certaines machines parallèles très haut de gamme comme les serveurs SGI Origin 3000 proposent des mécanismes physiques pour implanter des mémoires partagées «scalables» à temps d'accès non uniforme (NUMA). L'ordre de grandeur en temps d'un accès en mémoire distante est dans ce cas inférieur à la micro-seconde [SGI00] ; cependant cela se fait souvent au dépend des temps d'accès en mémoire locale.

A l'opposé, les architectures à mémoire distribuée sont constituées d'un ensemble de nœuds composés chacun d'une mémoire et d'un processeur, connectés entre eux par des réseaux. La mise en oeuvre d'applications parallèles sur ce type d'architecture est délicate ; le programmeur doit gérer explicitement les accès locaux et tenir compte du fait que les accès à distance sont souvent beaucoup plus coûteux et ce notamment du fait des latences réseaux. En pratique ces accès à distance sont réalisés grâce à une

librairie d'échanges de message comme MPI [MPI95, MPI97]. Le programmeur doit dans ce cas explicitement appeler une fonction MPI «send» pour envoyer un message à un nœud avec lequel il désire communiquer. Ce nœud récepteur doit explicitement appeler une fonction MPI «receive» pour recevoir le message envoyé par l'émetteur. Il est également possible d'utiliser une mémoire virtuelle partagée comme Treadmarks [KDCZ94] ou SCIos [KHCR99]. Dans ce dernier cas, on parle aussi de machine de type NUMA, car cette couche fournit une mémoire partagée dont le temps d'accès est non uniforme (NUMA).

Du point de vue des machines parallèles disponibles, ces dernières années ont connues une évolution très rapide. Les constructeurs de machines parallèles spécialisées sont en perte de vitesse. On assiste aujourd'hui à une montée en puissance des solutions basées sur l'interconnexion massive de composants standards, on parle aussi de *grappes*. Il est aujourd'hui possible de construire des systèmes très haute performance à coût réduit en interconnectant des composants standards. Les performances offertes par les solutions PC - notamment celles équipées de processeurs récents, Intel Pentium4 et AMD Athlon - ont largement rattrapé leur retard sur les solutions basées sur des processeurs RISC traditionnels [Spe00]. Cette évolution peut s'illustrer sur les machines qui étaient disponibles au laboratoire ID-IMAG dans lequel j'ai effectué ma thèse. Quand, j'ai commencé ma thèse le laboratoire disposait d'une machine parallèle spécialisée : un IBM SP1 à 32 processeurs (RS6000-66mhz). Aujourd'hui le laboratoire ID-IMAG en collaboration avec HP dispose d'une grappe de 225 PC d'entrée de gamme connectés entre eux par un réseau Fast-Ethernet, et envisage d'acheter une grappe d'une centaine de PC SMP bi-processeur reliées entre eux par une réseau plus performant.

Du point de vue des réseaux d'interconnexion, les nœuds des grappes peuvent être reliées entre eux soit par des réseaux ordinaires de type Fast Ethernet, soit par des réseaux à faible latence comme par exemple Myrinet. Cependant, les latences réseaux restent importantes. Un réseau de type Fast Ethernet offre une latence au mieux de l'ordre de 100 μs [Die98] et une bande passante théorique de 100 Mbit/sec. Les réseaux comme Myrinet offrent des performances supérieures avec des latences de l'ordre de 10 μs et une bande passante de l'ordre de 1 Gbit/sec [CR98]. Ces chiffres sont à comparer au temps de cycle processeur 1 ns pour un processeur à 1 Ghz, ou aux temps d'accès à une mémoire qui sont de l'ordre de 50 ns. La bande passante d'une mémoire est elle de l'ordre de 16.8 Gbit/sec (bande passante pour une mémoire de type DDR que l'on trouve dans les PC de bureaux). D'autre part ces latences sont celles potentiellement offertes par le support matériel, cependant le surcoûts logiciels sont souvent très importants. Ils sont dus à mauvaise interaction entre le système, les noyaux de communication et les noyaux de multiprogrammation légère. Ce problème est particulièrement critique dans l'utilisation de réseaux rapides. Il convient souvent de repenser ces interfaces afin de ramener ce surcoût logiciel à un niveau compatible avec les faibles latences des réseaux, des interfaces comme les messages actifs permettent cela [vCGS92].

Si l'on opte pour une description d'une application basée sur un graphe à grain fin, il apparaît donc nécessaire de prendre en compte le temps de ces communications qui peuvent être largement supérieurs aux temps d'exécution d'une tâche. La latence d'un réseau Ethernet correspond par exemple à 10^5 cycles processeurs d'un processeur cadencé à 1 Ghz.

Les grappes de PC sont souvent constituées de processeurs homogènes ayant les mêmes vitesses (ou tout au moins des vitesses du même ordre de grandeur), c'est pourquoi dans cette thèse nous avons choisi de ne pas considérer l'hétérogénéité de vitesses comme un paramètre clé. D'autre part les latences d'accès entre des machines différentes d'une grappe semblent peu sensibles à la localisation exacte des machines, et à la topologie du réseau. Cependant, dans une optique «méta-computing» ; et avec l'apparition de réseaux haut-débits et longue distance comme VTHD [VTH99], il est maintenant envisageable d'interconnecter plusieurs grappes de PC pour exécuter une même application parallèle. Dans un tel cas, la topologie du réseau et notamment son caractère hiérarchique apparaît essentiel. Cette hiérarchie est également importante lorsque l'on considère des interconnexions de machines SMP dans une grappe.

Prendre en compte les temps des communications dans la décision d'ordonnancement est l'un des facteurs clés pour l'exploitation efficace des architectures actuelles. D'autre part, ce facteur semble d'autant plus important que l'on envisage l'interconnexion de machines distantes, il a une importance capitale dans les architectures récentes comme les grappes de PC.

3. Modélisation des communications

Pour prendre en compte les communications, deux approches sont principalement envisageables. La première consiste à modéliser explicitement les communications en considérant les coûts directs qu'elles induisent. Les problèmes d'ordonnancement qui en résultent sont proches des problèmes d'ordonnancement classiques. Ils consistent très souvent à trouver pour chaque tâche où l'exécuter (choix d'un placement) et à quelle date l'exécuter. La seconde approche est fondée sur une prise en compte implicite des communications. Elle s'appuie sur un modèle d'application où les tâches qui la composent sont elles-mêmes des activités parallèles pouvant s'exécuter sur un nombre variable de processeurs avec une efficacité variable dépendant notamment des coûts de communications. Il s'agit du modèle des *tâches malléables*. Nous allons maintenant présenter en détail ces deux approches.

3.1. Les modèles à communications explicites

La première façon de prendre en compte les communications est de modéliser explicitement les coûts qu'elles induisent. Ainsi, si l'on considère une application décrite à l'aide d'un graphe de précédence et deux tâches de cette application liées par une relation de précédence, on peut alors introduire par exemple un surcoût dû à l'envoi d'un message ou bien une latence induite par le réseau si ces deux tâches sont placées sur deux processeurs différents. D'autres paramètres comme la congestion du réseau peuvent aussi être pris en compte. Deux critères apparaissent alors antagonistes dans le choix de cette modélisation comme d'ailleurs dans de nombreuses autres modélisations : la simplicité du modèle et le réalisme du modèle. La simplicité d'un modèle de communication permet d'obtenir et de construire des algorithmes d'ordonnancement efficaces notamment d'un point de vue théorique. Si le modèle est trop complexe il est alors extrêmement difficile de trouver un algorithme d'ordonnancement prenant en compte les différentes subtilités du modèle. Cependant, cette simplicité se fait souvent au détriment du réalisme de la modélisation des communications, et des coûts pourtant importants peuvent être négligés. Nous allons maintenant passer en revue les différents modèles qui ont été proposés.

3.1.1. Les modèles délai

L'un des premiers modèles proposé pour prendre en compte les communications est le modèle délai UECT (Unit Execution and Communication Time). Il a été introduit par Rayward-Smith [RS87]. Comme son nom l'indique, ce modèle considère que toutes les tâches ont une même durée unitaire. D'autre part ce modèle suppose que si une donnée doit être transmise entre deux tâches x et y placées sur des processeurs différents, alors un délai de communication de durée unitaire est nécessaire, c'est à dire que la tâche y ne peut commencer à s'exécuter qu'une unité de temps après la terminaison de la tâche x. La figure 2 illustre le modèle UECT. Elle présente d'une part un graphe de précédence avec le placement des tâches, et d'autre part un *diagramme de Gantt* représentant un ordonnancement valide sous le modèle délai UECT. Il s'agit d'une représentation très classique où l'abscisse représente le temps et l'ordonnée représente les processeurs.

Les modèles délais considèrent comme l'illustre la figure2, qu'il est possible d'exécuter des tâches pendant l'envoi et la réception des messages. Cela revient à supposer que l'envoi d'un message ne nécessite pas l'utilisation du processeur. En pratique ce recouvrement des communications par des calculs peut être approché par l'exécution concurrente de plusieurs processus légers (threads) sur chaque processeur [Gin97],

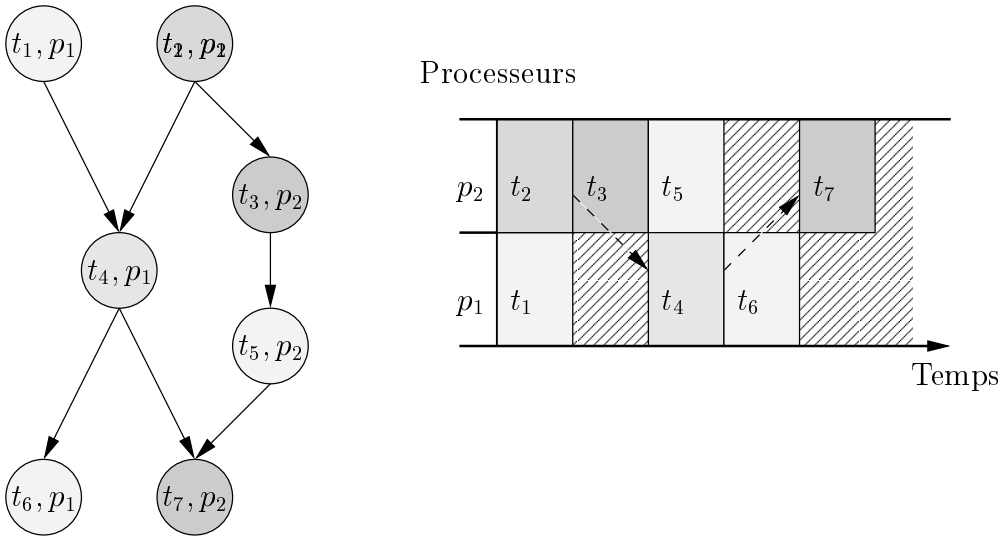

Figure 2 – A gauche, un graphe de précédence et un placement des tâches sur deux processeurs p_1 et p_2. A droite un diagramme de Gantt d'un ordonnancement valide sous le modèle délai UECT. Les zones hachurées représentent les périodes d'inactivité des processeurs. Les lignes en pointillé symbolisent le délai de communication. Les tâches t_3 et t_6 sont exécutées pendant l'envoi de messages, il y a recouvrement des calculs par les communications.

ou par l'utilisation de primitives de communications non bloquantes.

Le modèle délai UECT a par la suite été étendu. L'une des principales faiblesses du modèle UECT est son très faible degré de réalisme. Il n'offre en effet aucune latitude pour prendre en compte la variabilité du coût des communications entre les différentes architectures. Il suppose aussi que l'utilisateur choisisse pour décrire son application une granularité, c'est à dire un temps d'exécution pour les tâches de son application, égale au temps des communications. Cela parait très délicat. Papadimitriou et Yannakakis [PY90] ont notamment étendu le modèle délai. Le modèle qu'ils ont proposé est toujours basé sur un temps d'exécution unitaire pour les différentes tâches, mais ils considèrent que le délai de communication est cette fois un délai ρ variable et potentiellement grand. Nous appellerons par la suite LCT (Large communication time) ce nouveau modèle. La figure 3 représente le diagramme de Gantt d'un ordonnancement valide sous ce modèle.

Ce modèle permet de modéliser, au moins partiellement la différence des coûts de communication sur différentes architectures parallèles. Plus précisément il modélise efficacement des temps de latence réseau. Cependant il suppose de disposer d'un réseau parfait, sans congestion. Il suppose également soit de disposer d'un réseau avec une bande passante infinie, soit de supposer que les données devant être transmises ont toutes la même taille ou tout au moins le même ordre de grandeur. En effet, le

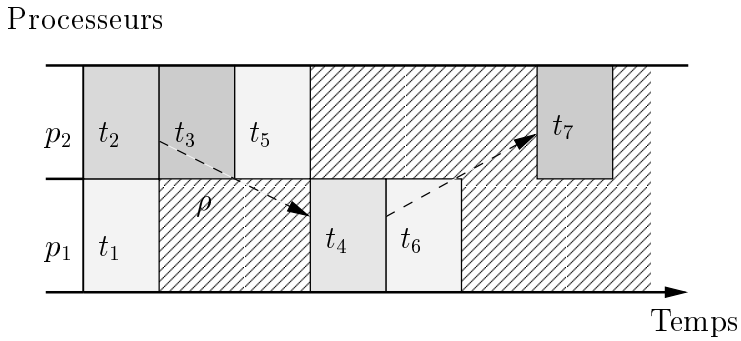

Figure 3 – Un diagramme de Gantt d'un ordonnancement valide sous le modèle délai LCT avec un délai $\rho = 2$ pour le graphe de l'application de la figure 2.

délai de communication ρ ne dépend pas de la taille des données à transmettre. Des modèles délai plus fins ont notamment été proposés par Hwang et al. [ACHL89]. Ils ont proposé de considérer un délai de communication dépendant d'une part de la taille des données à transmettre, et d'autre part de la topologie du réseau. Le délai de communication dépend alors aussi de la paire de processeurs qui communiquent.

3.1.2. Des modèles plus fins

Comme nous le verrons dans le chapitre suivant, les modèles délai ont été largement étudiés. Cependant ils négligent certains aspects pouvant être en pratique importants comme notamment les surcoûts à l'envoi de message ou la congestion du réseau. En effet pour effectuer des communications, il est parfois nécessaire d'effectuer localement plusieurs copies des régions mémoires contenant les données à transmettre. Ces copies peuvent être en pratique coûteuses, de nombreuses études cherchent d'ailleurs à éviter ces copies [TK95, WBT99]. Le modèle LogP [CKP+96] permet de mieux prendre en compte ces coûts. Dans celui-ci, le coût des communications est alors modélisé à l'aide de trois paramètres L, o et g. Le paramètre L modélise la latence réseau, le paramètre o (pour overhead en anglais) permet de prendre en compte un sur-coût local à l'envoi et la réception des messages. Enfin, le paramètre g (pour gap en anglais) décrit le temps minimal entre deux opérations de communication sur un même processeur, cela permet de limiter le nombre de messages en transit sur le réseau. Ainsi les modèles avec délais de communication sont des cas particuliers du modèle LogP, car il suffit de prendre $o = g = 0$. La figure 4 permet de bien comprendre le modèle LogP.

La première définition du modèle LogP ne considère que des petits messages de taille constante, cependant il a été étendu par la suite, pour prendre en considération des tailles de messages variables [AISS97, ELW97]. Les paramètres L, o et g sont

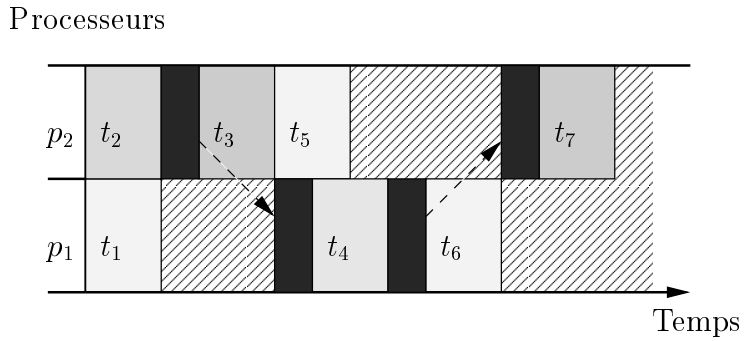

Figure 4 – Un diagramme de Gantt d'un ordonnancement sous le modèle LogP avec un délai $L = 1$ et un sur-coût $o = 1/2$ pour le graphe de l'application de la figure 2. Les zones grisées sombres représentent les surcoûts dus à l'envoi et à la réception de messages.

alors des fonctions affines de la taille du message à communiquer. Dans ce cas, il est alors nécessaire d'utiliser une modélisation très fine de l'application. Un graphe de précédence ne permet pas de décrire suffisamment précisément l'application et il est alors nécessaire d'utiliser une modélisation plus fine d'une application comme un graphe de flot de données. En effet, si une tâche produit deux résultats et communique ces résultats à deux tâches placées sur un processeur différent, alors les coûts des communications sont différents de ceux intervenant dans le cas ou un même résultat serait communiqué à ces tâches. La figure 5 illustre cela.

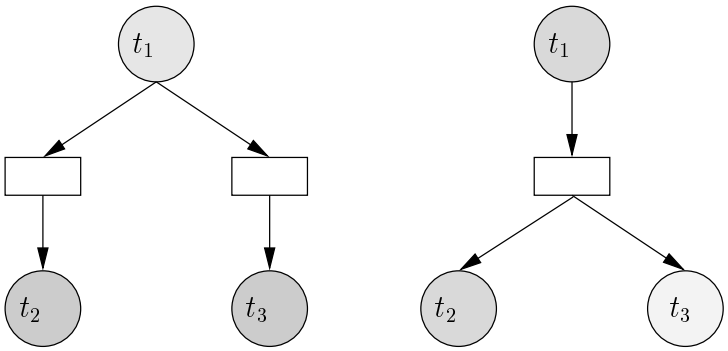

Figure 5 – Deux graphes des flots de données différents correspondant au même graphe de précédence. Les «ronds» représentent les tâches, et les «rectangles» représentent les données.

Il existe également d'autres modèles prenant en compte les communications comme BSP [Val90]. L'idée principale de ce modèle est de séparer les phases de calcul des phases de communications, en considérant une succession de super-étapes

(de l'anglais super-step). Les opérations de communication utilisées sont alors principalement des synchronisations globales et des communications de type «tous vers tous». Ce modèle est à rapprocher d'un paradigme de programmation. Nous avons choisi de ne pas nous intéresser à ce type de modèle dans cette thèse.

3.2. Le modèle des tâches malléables

A l'opposé des modèles à communications explicites, le *modèle des tâches malléables* est une autre approche pour la parallélisation d'application. L'idée principale du modèle des tâches malléables est de prendre en compte les communications de manière implicite en grossissant le grain de calcul. Le modèle des tâches malléables a été introduit par [TWY92] ; il est issu du *modèle de tâches parallèles* [DL89] dont il est très proche. Intuitivement, une *tâche malléable* est le regroupement d'un ensemble de tâches séquentielles. Une tâche malléable peut être exécutée en parallèle sur un nombre de processeurs variable avec un temps d'exécution dépendant de ce nombre de processeurs. Ce nombre de processeurs sera choisi par la politique d'ordonnancement mais il reste constant au cours de l'exécution d'une tâche malléable. Les différentes tâches malléables qui composent une application peuvent être ou non liées par des relations de précédence. On parle alors de tâches malléables indépendantes, et de tâches malléables avec relation de précédence. Ces relations de précédence sont alors modélisées classiquement à l'aide d'un graphe de précédence.

Le modèle des tâches malléables permet une parallélisation à deux niveaux. En effet on peut exploiter le parallélisme à l'intérieur d'une tâche malléable. Le temps d'exécution d'une tâche malléable dépend du nombre de processeurs qui exécutent cette tâche. Ce temps est fourni par l'utilisateur qui connaît son application ; il peut être issu par exemple d'une mesure lors d'une exécution précédente. Ce temps d'exécution prend en compte les temps des communications qui se produisent lorsque l'on exécute une tâche malléable sur plusieurs processeurs. Il dépend également du parallélisme intrinsèque de la tâche. D'autre part, le modèle des tâches malléables permet d'exploiter le parallélisme entre différentes tâches malléables. Dans le cadre du modèle des tâches malléables les temps de communications entre différentes tâches malléables sont très souvent négligés. Cela peut se justifier par la taille du grain, c'est à dire par le fait que le temps d'exécution d'une tâche malléable est relativement important par rapport aux communications, car il s'agit d'un regroupement de tâches séquentielles.

Le modèle des tâches malléables offre donc une vision structurée d'une application. Cette vision reflète très souvent sa structure logique. Elle s'adapte donc très bien à des applications tels que la décomposition de domaine. Au cours de son travail de thèse, Grégory Mounié a évalué l'utilité pratique du modèle des tâches malléables [BDMT99]. Il a notamment montré l'utilité de celui-ci pour la parallélisation d'une

application d'océanographie à grande échelle. Pour mieux saisir les problèmes que nous considérons, nous avons choisi d'illustrer ceux-ci sur cette application d'océanographie. Celle-ci permet de modéliser la circulation océanique sur un domaine tel que l'océan Atlantique Nord. Elle est fondée sur la résolution approchée des équations de Navier-Stokes à l'aide d'un schéma aux différences finies sur un maillage adaptatif [BD99]. Le schéma d'approximation numérique est basé sur une discrétisation de l'océan en espace et en temps. Le maillage est constitué d'un ensemble de grilles. Chacune d'elles modélise une zone de l'océan. En fonction des besoins, des turbulences ou des évolutions locales au cours du temps, on souhaite avoir une description plus fine de certaines grilles. Une grille fille imbriquée dans une grille parent correspond à un raffinement en espace et en temps de cette dernière. A partir de cet arbre de raffinement, il est possible d'en déduire un graphe de précédence décrivant l'application. Dans ce cas, le graphe de précédence est série-parallèle.

La figure 6 représente un arbre de raffinement et le graphe de précédence associé. Chaque calcul d'intégration sur une grille peut être calculé sur plusieurs processeurs, l'utilisation du modèle des tâches malléables se prête donc très bien à la parallélisation de cette application.

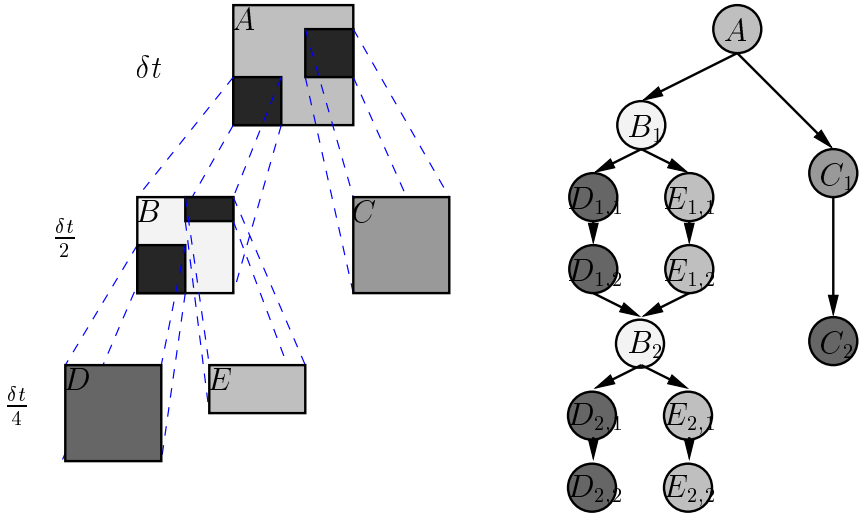

Figure 6 – Un arbre de raffinement, et le graphe de précédence associé.

Dans le cadre du modèle des tâches malléables, le problème de l'ordonnancement consiste alors à trouver sur quel ensemble de processeurs exécuter chaque tâche et dans quel ordre. Une interprétation géométrique permet de mieux comprendre le problème. La figure 7 représente à l'aide d'un diagramme de Gantt un exemple d'ordonnancement valide pour le graphe de précédence précédent.

Le modèle des tâches malléables offre donc une alternative aux modèles plus classiques basés sur des communications explicites tels que les modèles délai. En effet,

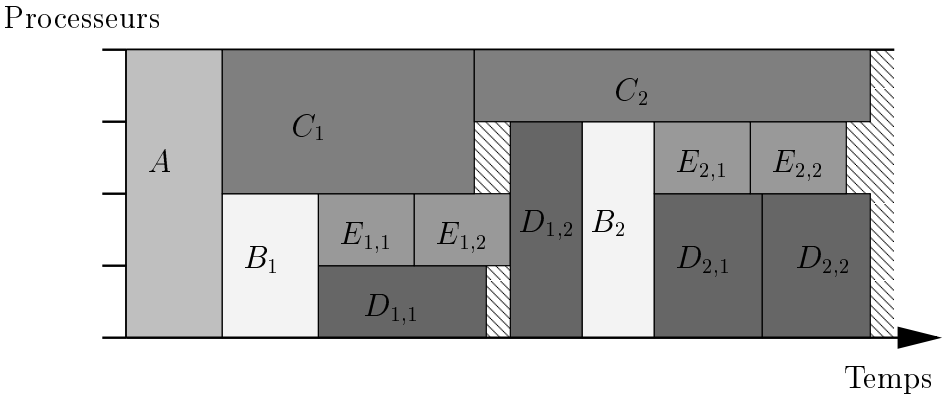

Figure 7 – Un ordonnancement sous le modèle des tâches malléables du graphe de précédence de l'application décrite figure 6.

ceux-ci ne prennent souvent en compte que partiellement le coût des communications. De plus l'ordonnancement sous ces modèles est un problème très délicat. Comme nous allons le voir dans le chapitre suivant, les algorithmes proposés dans la littérature ne sont que partiellement satisfaisants.

4. Conclusion partielle

Dans ce chapitre, nous avons présenté différentes approches pour modéliser une application parallèle et notamment les graphes de précédence, que nous utiliserons tout au long de cette thèse. En pratique, une question importante est celle de la construction de cette représentation. Les langages parallèles de haut niveau comme Athapascan1 apportent une réponse à ce problème. En construisant dynamiquement le graphe de flot de données de l'application, ils permettent l'utilisation d'algorithmes d'ordonnancements statiques dans un cadre large.

Nous avons ensuite abordé la description des machines parallèles. L'un des points clés est le coût très important lié aux communications. Une première solution consiste à utiliser des réseaux à faible latence ainsi que des interfaces logiciels adaptées à ces réseaux rapides, afin de réduire l'impact de ces coûts. Une seconde direction complémentaire consiste à prendre en compte ces coûts de communication dans la décision d'ordonnancement. L'objectif est alors d'utiliser la connaissance fine de l'application pour placer judicieusement les différents calculs de façon à minimiser le nombre de communications réellement effectuées.

Différents modèles ont été proposés pour prendre en compte ces communications. Nous avons distingué parmi ceux-ci deux grandes familles de modèles : les modèles à communications explicites et le modèle des tâches malléables. Dans le chapitre suivant, nous présentons les différents résultats obtenus pour l'ordonnancement sous ces modèles.

CHAPITRE

2

Ordonnancement avec prise en compte des communications

Dans le cadre du calcul parallèle, l'ordonnancement est la décision qui consiste à choisir où et quand exécuter les différentes tâches d'une application. Sur les architectures à mémoire distribuée, les coûts des communications peuvent souvent être très importants. Prendre en compte ces coûts de communications dans la décision d'ordonnancement apparaît donc comme une nécessité pour obtenir une exécution efficace. Différents modèles ont été proposés pour prendre en compte ces communications. Deux grandes familles ont été envisagées : les modèles à communications explicites et les modèles des tâches malléables. Ils sont décrits dans le chapitre précédent. Les modèles à communications explicites modélisent directement plus ou moins finement les coûts des communications. Le plus connu des modèles de ce type est le modèle délai. Le modèle des tâches malléables s'appuie sur un modèle d'application où les tâches qui la composent sont elles-mêmes des activités potentiellement parallèles. Ces tâches peuvent s'exécuter sur un nombre variable de processeur avec une efficacité variable dépendant notamment des coûts de communications. Dans ce chapitre, nous nous intéressons aux différents problèmes d'ordonnancement induit par ces modèles. Nous présenterons notamment les résultats obtenus pour les problèmes d'ordonnancement sous les modèles de type délai car ceux-ci ont été largement étudiés. Puis nous terminerons par les résultats connus pour l'ordonnancement sous le modèle des tâches malléables et sous des modèles proches.

Sommaire

1. Ordonnancement sous le modèle délai

Dans le cadre des modèles délai, les problèmes d'ordonnancement consistent à choisir pour chaque tâche, d'une part un placement, c'est-à-dire un processeur sur lequel elle sera exécutée, et d'autre part une date d'exécution. Pour que l'ordonnancement soit valide, il faut qu'il respecte les contraintes imposées par la modélisation des communications. Lorsque le problème d'ordonnancement suppose un nombre non borné de processeurs, nous parlerons par la suite de *problème de regroupement*[1]. Le nom regroupement s'explique par le fait que le problème consiste alors à former des groupes ordonnés de tâches qui seront exécutées sur un même processeur. Lorsque l'on prend en compte des délais de communication, il peut être intéressant d'exécuter plusieurs fois la même tâche sur des processeurs différents. Cela peut être utile lorsqu'une tâche a de nombreuses tâches successeurs, dupliquer cette tâche sur plusieurs processeurs permet alors d'exécuter les tâches successeurs plus tôt. Dans le cas de la duplication, la décision d'ordonnancement consiste à choisir plusieurs processeurs et plusieurs dates d'exécution pour chaque tâche.

Dans ce paragraphe, nous allons présenter tout d'abord la complexité des problèmes d'ordonnancement sous les modèles avec délai de communication. Nous verrons que très souvent les problèmes d'ordonnancement résultants sont $\mathcal{NP}$-difficiles. Face à l'impossibilité de résoudre ces problèmes en temps polynômial et raisonnable (sous réserve que $\mathcal{P} \neq \mathcal{NP}$), une solution souvent adoptée consiste à se tourner vers les *algorithmes d'approximation*. Nous présenterons donc les différents algorithmes d'approximation proposés pour ces problèmes. Enfin nous terminerons par différentes heuristiques utilisées en pratique.

1.1. Complexité

Pour désigner les différents problèmes d'ordonnancement, nous utilisons dans toute la suite de ce paragraphe la notation à trois champs $\alpha \mid \beta \mid \gamma$ introduite par Lawler et al. [Law82] et étendu ensuite par Veltman et al. [VLL90] pour prendre en compte les problèmes avec délai de communication. Le premier champ α décrit les ressources du système. Dans notre cas, cela correspond au nombre de processeurs. Le premier champ peut être égal à $P\infty$, P ou Pm, pour désigner respectivement un nombre infini de processeurs, un nombre borné de processeurs ou un nombre fixé de processeurs. Le deuxième champ β décrit la structure du graphe de précédence (*prec*

[1] clustering

dans le cas d'un graphe quelconque, *tree* dans le cas d'un arbre, ...). Le deuxième champ précise également le délai de communication considéré noté c et le temps des tâches noté p. Si par exemple, le deuxième champ est $tree, c = 1, p = 1$ alors cela correspond à l'ordonnancement d'un arbre sous le modèle UECT qui a été présenté dans le chapitre précédent. Ce champ précise aussi si l'on considère la duplication, elle est alors notée *dup* ou la préemption notée *pmtn*. Enfin le troisième champ γ correspond à l'objectif du problème. Au cours de cette thèse nous nous intéressons exclusivement à minimiser la durée total d'exécution d'un ordonnancement [2] ; ce critère est habituellement noté C_{max}.

L'un des premiers résultats de complexité connu concerne le problème de l'ordonnancement sous un modèle UECT sur un nombre borné de processeurs [RS87]. Le problème d'ordonnancement $P \mid prec, p = 1, c = 1 \mid C_{max}$ est $\mathcal{NP}$-difficile. Ce résultat a par la suite été étendu au cas d'un nombre non borné de processeurs [Pic95]. Nous pouvons remarquer que ce dernier problème était facile dans le cas où les délais de communications étaient négligés. Cependant si l'on considère la possibilité de dupliquer des tâches alors un résultat original a été trouvé par Colin et Chrétienne [CC91], ils ont notamment montré que le problème de regroupement $P\infty \mid prec, p_j = 1, c_j = 1, dup \mid C_{max}$ était polynomial. Un autre résultat sur la complexité a été démontré par Hoogeven et al. : «décider si le problème $P \mid biparti, p_j = 1, c_j = 1 \mid C_{max}$ admet un ordonnancement de durée inférieure ou égale à 4» est un problème $\mathcal{NP}$-complet [HLV94], cela implique qu'il n'existe pas d'algorithme d'approximation avec une garantie de performance meilleure que 5/4 pour le problème $P \mid prec, p_j = 1, c_j = 1 \mid C_{max}$ si $\mathcal{P}$ est différent de $\mathcal{NP}$.

Si l'on considère des modèles prenant en compte des grands délais de communications comme le modèle LCT présenté dans le chapitre précédent, alors les problèmes d'ordonnancement semblent plus complexes. D'un point de vue de leurs complexités, ils sont très souvent $\mathcal{NP}$-difficiles, et ce même pour des structures de graphes simples. Ainsi, Jakoby et al.. ont montré que le problème de regroupement $P\infty \mid binary\,tree, p_j = 1, c \mid C_{max}$ était $\mathcal{NP}$-difficile [JR92]. Récemment, Afrati et al.. ont montré que ce même problème d'ordonnancement sur 2 processeurs (le problème $P2 \mid binary\,tree, p_j = 1, c \mid C_{max}$) était «déjà» $\mathcal{NP}$-difficile [ABFM99]. Enfin, dans le cas de grand temps de communication, même si l'on considère la duplication, Papadimitriou et Yannakakis ont montré que le problème $P\infty \mid prec, p = 1, c, dup \mid C_{max}$ était $\mathcal{NP}$-difficile [PY90]. Un état de l'art plus complet de la complexité des problèmes d'ordonnancement sous les modèles délais a été publié par Chrétienne et Picouleau [CCLL95].

[2] makespan

1.2. Algorithmes d'approximation

Nous avons vu que les problèmes d'ordonnancement sous des modèles avec délai de communication sont très souvent $\mathcal{NP}$-difficiles. Face à l'impossibilité de résoudre ceux-ci en temps polynômial, une solution intéressante consiste à se tourner vers les *algorithmes d'approximation*. Intuitivement l'idée consiste à «sacrifier» une partie de la qualité de la solution obtenue en échange d'un temps de calcul polynômial. Un algorithme est une k-approximation si pour toute instance du problème la solution obtenue n'excède pas la solution optimale d'un rapport au plus k. On parle aussi de la *garantie de performance* de l'algorithme. Le cadre des algorithmes d'approximation dépasse largement les problèmes d'ordonnancement, un excellent état de l'art de ceux-ci peut être trouvé dans le livre de Hochbaum [Hoc96].

Historiquement l'un des premiers algorithmes d'approximation proposé l'a été dans le contexte de l'ordonnancement. Il s'agit des algorithmes de liste introduits par Graham [Gra66]. Remarquons qu'à cette époque la notion de problème $\mathcal{NP}$-complet n'existe pas encore ; elle sera introduite peu après par Cook [Coo71]. Les algorithmes de liste, introduits par Graham, sont l'une des premières heuristiques proposées pour les problèmes d'ordonnancement de tâches avec relation de précédence sur un nombre m de processeurs. Les algorithmes de liste sont basés sur une allocation gloutonne des tâches aux processeurs. L'idée consiste à ne pas laisser un processeur inoccupé à un instant t si il peut exécuter une tâche à cet instant. Dans le modèle considéré, une tâche peut être exécutée à l'instant t si et seulement si tous ses prédécesseurs ont déjà été exécutés auparavant ; on dit aussi que la tâche est prête. Graham a notamment montré le résultat suivant.

Proposition 1 [Garantie de Graham]. La durée d'un ordonnancement calculé à l'aide d'un algorithme de liste ω_{liste} est inférieure à $\frac{\omega_1}{m} + (1 - \frac{1}{m})\omega_\infty$, où ω_1 désigne le temps d'exécution sur un seul processeur et ω_∞ désigne le temps d'exécution sur une infinité de processeurs.

Comme $\frac{\omega_1}{m}$ et ω_∞ sont des bornes inférieures triviales du temps d'exécution d'un ordonnancement, on en déduit le résultat très connu suivant. Les algorithmes de liste sont des algorithmes d'approximation dont la garantie de performance est $2 - 1/m$.

De nombreux résultats d'approximation ont été proposés pour des modèles prenant en compte des petits délais de communication [LKV96, HM97, Gol99]. Intuitivement cela s'explique par le fait que ces modèles sont proches des modèles d'ordonnancement sans coût de communication. Il est facile de construire, à partir d'une k-approximation dans un modèle sans communication, une $2k$-approximation dans un modèle avec petit délai de communication. Il suffit de considérer que la

durée d'exécution de chaque tâche augmente du délai de communication. Dans ce contexte, l'objectif des différentes études a souvent été d'obtenir des algorithmes d'approximation avec une meilleure garantie. Hanen et Munier ont par exemple proposé un algorithme d'approximation dont la garantie est $2 - 1/m$ pour le problème $P \mid prec, p = 1, c = 1, dup \mid C_{max}$, c'est-à-dire la même garantie que pour les algorithmes de liste dans le contexte sans communication [HM97]. Pour les problèmes d'ordonnancement sous des modèles plus réalistes prenant en compte des grands temps de communications comme le modèle LCT, les algorithmes d'approximation proposés, en dehors de ceux dont la garantie dépend linéairement du délai de communication ρ sont beaucoup plus rares. Nous allons maintenant les présenter. Nous reviendrons par la suite sur les algorithmes dont la garantie dépend linéairement de ρ.

Dans le cas d'un grand délai de communication, Papadimitriou et Yannakakis ont les premiers proposé un *algorithme d'approximation* de garantie constante. Ils ont proposé pour le problème du regroupement avec duplication $P\infty \mid prec, p = 1, c, dup \mid C_{max}$ un algorithme avec une garantie de 2. Celui-ci est basé sur l'introduction d'une borne inférieure fine et définie inductivement de la date d'exécution d'une tâche. Plusieurs auteurs ont par la suite étendu ce résultat au cas d'un délai de communication variable, dépendant des tâches qui communiquent [JCL96, AK98, AD98]. Ces résultats d'algorithmes d'approximation constante pour les problèmes d'ordonnancement avec grand délai de communication et des structures de graphe de précédence arbitraires sont à notre connaissance les seuls connus. Malheureusement, ils ne sont pas utilisables en pratique. Le «travail» qu'ils engendrent de par la duplication n'est pas borné relativement à la taille du graphe. Le nombre de tâches réellement exécutées peut être très important. Dans ce cas il n'est alors pas possible de «replier» efficacement le regroupement obtenu sur un nombre borné de processeurs pour obtenir un ordonnancement qui puisse être utilisé en pratique.

Pour des structures générales de graphes, et pour le même problème d'ordonnancement mais sur un nombre borné de processeurs $P \mid prec, p = 1, c, dup \mid C_{max}$, Rapine a proposé dans sa thèse une extension originale des algorithmes de liste [Rap99]. Il a montré qu'il était possible d'obtenir un algorithme dont la garantie de performance dépendait de $\mathcal{O}(\sqrt{\rho})$. La technique utilisée consiste à limiter le nombre de tâches qu'il faut dupliquer pour exécuter une tâche. Dans le cas de structures de graphe particulières un résultat très intéressant a été proposé par Munier. Elle a proposée un algorithme d'approximation constante pour le problème du regroupement sans duplication $P\infty \mid out - tree, p = 1, c \mid C_{max}$ [Mun99]. Ce résultat est le premier résultat d'approximation constante pour le problème de regroupement sans duplication et avec grand délai de communication.

Ainsi, à notre connaissance, dans le cas de structures de graphes générales, il n'existe pas d'algorithme d'approximation constante ni pour le problème du regroupement sans duplication ni pour le problème de l'ordonnancement avec duplication sur un nombre borné de processeurs sous des modèles prenant en compte des com-

munications coûteuses. Il n'existe pas non plus de résultat qui prouve que cela soit impossible [3]. Cependant dans le cas où l'on considère que le graphe n'est pas connu à l'avance et est découvert au fur et à mesure de l'exécution (on-line scheduling) un résutat négatif a été montré par Dang et al.. [DKM99]. Dans ce cadre, ils ont montré qu'il n'était pas possible d'obtenir un algorithme d'approximation dont la garantie de performance est meilleure que $\mathcal{O}(\rho/\log(\rho))$.

1.3. Heuristiques

En pratique, le problème d'ordonnancement prenant en compte des temps de communication est très important, et de nombreuses heuristiques ont été proposées. Très souvent ces heuristiques sont appliquées sur des graphes de précédence composés de plusieurs milliers voire plusieurs millions de tâches. Cette remarque est particulièrement vraie, lorsque l'on considère des graphes dont la granularité est fine, c'est-à-dire des graphes dont le temps d'exécution des tâches est faible. En pratique, une attention particulière a donc été portée sur la recherche d'algorithmes dont la complexité soit faible et pas seulement polynomiale. Les heuristiques proposées peuvent être classées en trois catégories, les algorithmes de liste, les algorithmes basés sur une analyse du chemin critique et les algorithmes basés sur une décomposition du graphe de précédence.

1.3.1. Les algorithmes de listes revisités

Les algorithmes de liste ont été adaptés afin de prendre notamment en compte des délais de communications [ACHL89, WG90, CMPS97]. L'heuristique la plus connue de cette catégorie est ETF pour *Earliest Task First* [ACHL89]. ETF est un algorithme glouton dont le principe est de ne pas laisser un processeur inoccupé à un instant donné si une tâche qui n'a pas encore été exécutée peut l'être. La durée d'un ordonnancement calculé à l'aide d'un algorithme de liste dans ce contexte ω_{etf} est inférieure à $\frac{\omega_1}{m} + (1 - \frac{1}{m})\omega_\infty + C$, où ω_1 désigne le temps d'exécution sur un seul processeur, ω_∞ désigne le temps d'exécution sur une infinité de processeurs sans communication et C désigne la plus grande somme des délais de communication en suivant un chemin du graphe de précédence. D'autres extensions des algorithmes de liste ont été proposées. Doreille a montré que les algorithmes de liste pouvaient être étendus dans le cas où le graphe n'est pas entièrement connu à l'avance, tout en offrant le même type de garantie [Dor99]. Kort et Trystram ont également montré qu'ils pouvaient être étendus à des modèles de communications plus fins comme

[3] C'est le cas pour certains problèmes comme la clique max. On dit aussi que ces problèmes ne sont pas APX

LogP [KKT00]. Il est à noter qu'il s'agit à notre connaissance du seul algorithme d'ordonnancement qui a été proposé sous le modèle LogP pour des structures de graphes arbitraires. D'autres résultats d'ordonnancement sous le modèle LogP ont cependant été proposés pour des structures très particulières [LTZ01, BR97, Ver00].

1.3.2. Les heuristiques basées sur une analyse du chemin critique

Les heuristiques basées sur une analyse du chemin critique forment une autre classe très populaire d'heuristiques pour le problème du regroupement sans duplication [Sar89, BK88, GY94a, LP96]. Le problème du regroupement correspond au problème de l'ordonnancement sur un nombre non borné de processeurs. Il s'agit de former un ensemble de groupes de tâches qui seront exécutées sur le même processeur. Historiquement, la première heuristique à avoir été proposée est celle de Sarkar [Sar89]. Sarkar considère le problème du regroupement comme la phase préliminaire à l'ordonnancement sur m processeurs, et nomme cette phase «Internalisation Pre-Pass». Son heuristique consiste à fusionner, c'est-à-dire placer sur le même processeur deux tâches communiquantes, et à itérer ce schéma jusqu'à ce qu'une telle fusion n'améliore plus la durée du regroupement obtenu.

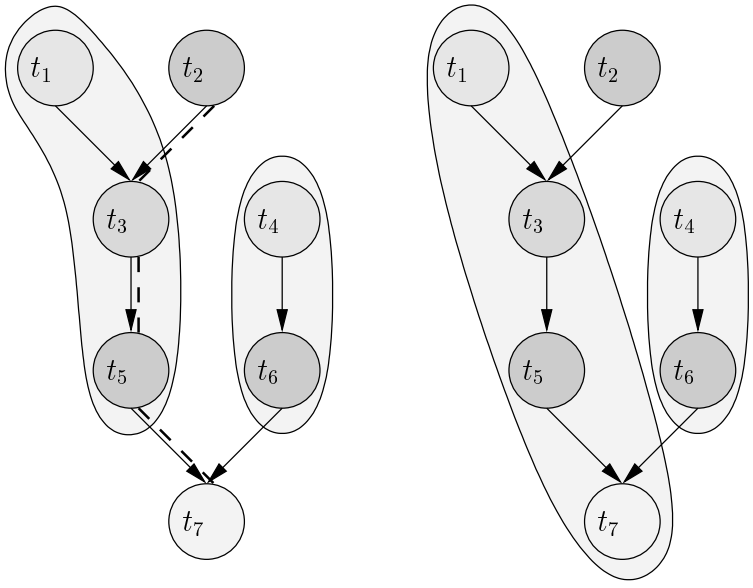

Figure 1 – Un exemple d'itération de l'algorithme DSC pour un modèle LCT avec un délai de communication $\rho = 2$. A gauche, un graphe partiellement regroupé, la ligne en pointillé représente la séquence dominante. A droite le graphe obtenu après une itération

Actuellement, la plus connue des heuristiques pour le problème du regroupement

est DSC qui signifie Dominant Sequence Clustering. L'algorithme DSC est un algorithme itératif qui a été proposé par Gerasoulis et Yang. L'idée de celui-ci est de fusionner à chaque étape deux groupes de tâches de la séquence dominante. La séquence dominante est le plus long chemin du graphe partiellement regroupé. La figure 1 illustre le fonctionnement de l'algorithme DSC. La conception de l'algorithme DSC a été guidée par le souci d'obtenir une faible complexité. Ainsi l'algorithme proposé évite de recalculer la séquence dominante à chaque itération. Cela a permis d'obtenir une complexité $\mathcal{O}((e+v).\log(v))$ pour l'algorithme, où e désigne le nombre d'arêtes et v le nombres de tâches dans le graphe de précédence. En pratique, les solutions obtenues à partir du problème du regroupement sans duplication peuvent être repliées efficacement sur un nombre borné de processeurs. Sarkar a proposé une heuristique pour placer les différents groupes de tâches obtenus sur les différents processeurs [Sar89]. Le système PYRROS basé sur DSC utilise lui un système d'équilibrage de charge pour adapter le regroupement produit par DSC [GY92].

1.3.3. Décomposition du graphe de précédence

Une autre famille d'heuristiques qui a été beaucoup moins étudiée est basée sur la décomposition de graphe. McCreary et Gill ont proposé une heuristique basée sur une décomposition récursive en «clan» [GM89]. Intuitivement un groupe de tâches est un «clan» si et seulement si toutes les tâches de ce groupe ont les mêmes tâches prédécesseurs et les mêmes tâches successeurs en dehors de ce groupe. La figure 2 illustre cette notion de «clan».

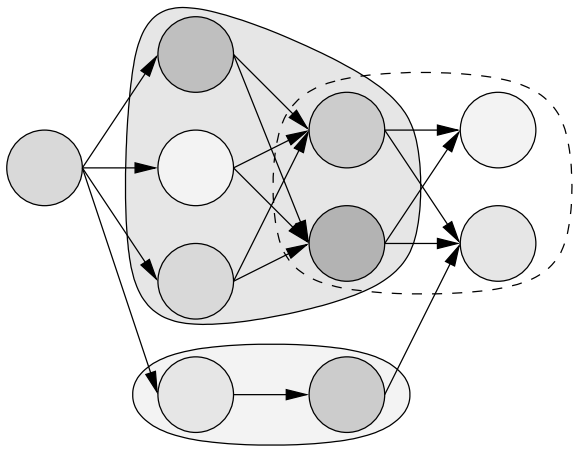

Figure 2 – Un graphe de précédence. Les deux groupes grisés de tâches forment des clans, par contre le groupe de tâches en pointillé n'en est pas un car toutes les tâches n'ont pas les mêmes prédécesseurs.

L'idée qu'ils ont proposée est fondée sur l'unicité de la décomposition récursive d'un graphe de précédence en «clan». Une comparaison de différentes heuristiques a par la suite été publiée et a montré que cette approche semblait prometteuse [KMJ94]. Cependant, l'un des freins à l'utilisation d'un tel algorithme est la complexité de calcul de la décomposition qui est de $\mathcal{O}(v^3)$. Récemment cette complexité a été ramenée à $\mathcal{O}(v^2)$ [EGMS94]. Nous pouvons noter que la décomposition récursive proposée a également eu des applications dans le domaine des représentations et dessins de graphes orientés [CMS98].

2. Ordonnancement sous le modèle des tâches malléables

Le modèle des tâches malléables s'appuie sur un modèle d'application dans lequel les tâches peuvent s'exécuter sur un nombre variable de processeurs. Une tâche malléable peut être vue comme le regroupement d'un ensemble de tâches séquentielles. La durée d'exécution d'une tâche malléable dépend du nombre de processeurs sur lesquels elle est exécutée. Cette durée variable peut être mesurée expérimentalement et elle permet de prendre notamment en compte le temps des communications. Le lecteur pourra trouver une description plus complète du modèle des tâches malléables dans le chapitre précédent. L'ordonnancement sous le modèle des tâches malléables consiste à choisir pour chaque tâche d'une part un nombre de processeurs pour exécuter cette tâche, et d'autre part une date d'exécution pour cette tâche. La figure 7 page 28 représente le diagramme de Gantt d'un ordonnancement sous le modèle des tâches malléables.

Nous allons maintenant présenter les différents travaux déjà réalisés sur l'ordonnancement avec le modèle des tâches malléables. Nous présenterons également les résultats pour des modèles proches de celui des tâches malléables comme le modèle des tâches continues [PM91] ou le modèle des tâches multiprocesseurs [GGJ78, BDW86]. Dans le modèle des tâches continues, le nombre de processeurs exécutant chaque tâche est un nombre réel qui sera choisi par la politique d'ordonnancement. Dans le modèle des tâches multiprocesseurs, c'est un entier fixé a priori. Un très bon article de synthèse sur le sujet a été écrit par Maciej Drozdowski [Dro96].

La complexité du problème de l'ordonnancement des tâches malléables a été étudiée par Du et Leung dans [DL89], qui les nommaient *système de tâches parallèles*. Ils ont montré que le problème de l'ordonnancement de tâches malléables indépendantes était un problème $\mathcal{NP}$-difficile au sens fort pour un nombre de processeurs égal à 5, et que le même problème avec des relations de précédence était fortement $\mathcal{NP}$-difficile à partir de 2 processeurs. Ce résultat implique notamment qu'il n'existe pas de schéma d'approximation pleinement polynomial pour ces problèmes.

2.1. Tâches indépendantes

D'un point de vue théorique, le problème de l'ordonnancement de tâches malléables indépendantes avec un nombre de processeurs fixé admet un *schéma d'approximation* (polynomial time approximation scheme) [JP98]. Cela signifie qu'il existe une famille d'algorithmes d'approximation $(A_\epsilon)_{\epsilon>0}$ où ϵ est aussi petit qu'on le souhaite, telle que la garantie de performance de l'algorithme A_ϵ est inférieure ou égale à $1+\epsilon$. Cependant cette famille d'algorithme n'est pas *pleinement polynomiale*, c'est-à-dire que la complexité des algorithmes n'est pas un polynôme en $\frac{1}{\epsilon}$. D'autre part, bien que le schéma de Klaus Jansen soit linéaire en fonction du nombre de tâches, la complexité dépend d'une constante multiplicative en $\mathcal{O}(m^m)$ qui fait que cet algorithme n'est pas utilisable en pratique.

En pratique, les tâches malléables indépendantes sont très souvent ordonnancées en deux phases. La première phase consiste à choisir une *allocation*, c'est-à-dire choisir pour chaque tâche le nombre de processeurs qui seront chargés de son exécution. La deuxième phase consiste à résoudre le problème d'ordonnancement de tâches multiprocesseurs indépendantes résultant du choix de cette allocation. Deux voies complémentaires ont étés proposées pour résoudre le problème en mettant l'accent soit sur la première phase, soit sur la seconde phase.

Turek, Wolf et Yu [TWY92] ont montré le résultat suivant. A partir d'un algorithme avec une garantie λ pour le problème de l'ordonnancement de tâches multiprocesseurs indépendantes, il est possible de construire un algorithme avec une garantie λ pour le problème de l'ordonnancement de tâches malléables indépendantes. Pour, le choix de l'allocation ils utilisent les deux bornes inférieures classiques du temps d'exécution d'un ensemble de tâches multiprocesseurs : *le chemin critique* (qui correspond ici au temps de la plus longue tâche) et *le travail moyen*. L'allocation de Turek et al. réalise un compromis entre ces deux critères. Elle consiste à réduire la plus longue tâche en lui allouant plus de processeurs, tant que cela n'engendre pas trop de travail. Le problème l'ordonnancement de tâches multiprocesseurs indépendantes induit est très proche des problèmes de strip-packing bi-dimensionnel [BCR80, CGJT80], ainsi que des problèmes d'ordonnancement avec contraintes de ressources [GG75]. En utilisant un algorithme de strip-packing tel que celui de Steinberg [Ste97], il est alors possible d'obtenir une garantie de performance de 2.

Cependant, il est possible d'obtenir de meilleures garanties de performance en mettant l'accent sur la première phase du calcul de l'allocation. Mounié, Rapine et Trystram ont ainsi montré qu'il était possible d'obtenir une garantie de $\sqrt{3}$ [MRT99]. Ils ont ensuite amélioré ce résultat pour obtenir une garantie de $3/2$ [MRT01]. Le choix de l'allocation est basé sur l'utilisation d'un problème de sac à dos entier de petite taille. Celui-ci permet de partitionner en deux ensembles les tâches. La seconde

phase consiste dans ce cas simplement à ranger les tâches sur deux étagères, comme l'illustre la figure 3.

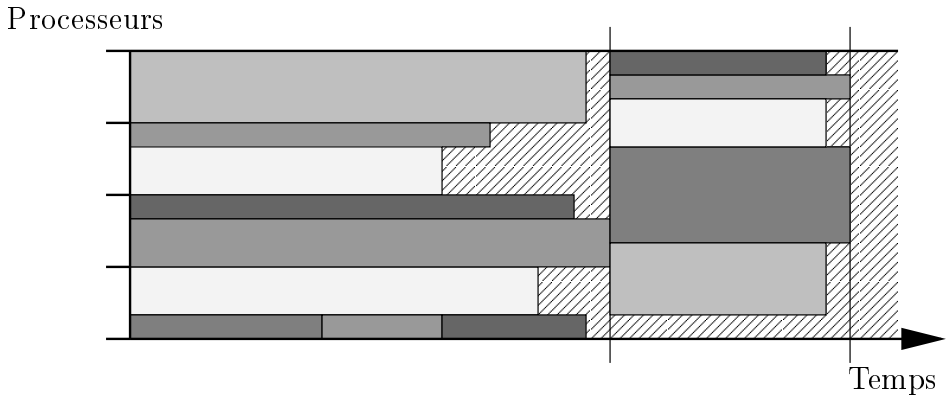

Figure 3 – Ordonnancement avec deux étagères

2.2. Tâches avec relation de précédence

Il existe peu de résultats pour l'ordonnancement de tâches parallèles prenant en compte des relations de précédences. Prasanna et Musicus se sont intéressés à un modèle de tâches continues, pour lesquelles les facteurs d'accélérations (rapport entre les temps d'exécution sur p processeurs et sur un seul processeur) considérés ont une forme particulière. Le facteur d'accélération est en p^α où p est la fraction de processeurs allouée à la tâche [PM91], et $0 \leq \alpha \leq 1$. Dans ce cadre, ils ont montré qu'il était possible d'utiliser la théorie du contrôle optimal. Ils ont proposé un algorithme d'ordonnancement optimal pour le problème qui en résulte. Ils ont également montré qu'il était possible d'appliquer ces résultats à des problèmes d'algèbre linéaire [PM96]. Cependant, une restriction importante liée à la forme du facteur d'accélération impose que toutes les tâches aient des facteurs d'accélération identiques.

Un autre résultat intéressant a été proposé par Feldmann et al. [FKS93]. Ils considèrent un modèle intermédiaire entre le modèle des tâches malléables et le modèle des tâches multiprocesseurs. Ils considèrent que chaque tâche doit s'exécuter sur un nombre de processeurs p fixé, cependant ils s'autorisent la possibilité de «virtualiser» cette exécution, c'est-à-dire la possibilité d'exécuter la tâche sur un nombre de processeurs inférieur à p avec un travail identique. Cela peut être vu comme un modèle de tâches malléables avec une parallélisation parfaite jusqu'à un certain seuil qui dépend de la tâche. Dans ce cadre, ils ont proposé un algorithme d'approximation avec une garantie $\frac{3+\sqrt{5}}{2}$.

3. Conclusion

Les modèles à communications explicites et notamment le modèle délai ont été très étudiés. La complexité de ces problèmes a notamment été l'objet de nombreuses études. Cependant ces études n'offrent aucune réponse au problème du monde réel de la parallélisation d'application. Sous des modèles délai à petit temps de communication, comme par exemple le modèle UECT, de nombreux algorithmes d'approximation ont été proposés et la difficulté du problème semble bien maîtrisée. Cependant, comme nous l'avons vu dans le chapitre précédent ces modèles semblent peu adaptés pour décrire les coûts des communications des machines actuelles.

Dans le cas de modèles plus proches de la réalité comme les modèles à grand temps de communication ou des variantes plus sophistiquées, la difficulté des problèmes qui en découlent n'est pas encore maîtrisée. Quelques algorithmes d'approximation ont bien été proposés, mais leur intérêt reste limité de par le modèle considéré ou de par le fait qu'ils soient restreints à des structures particulières. En pratique, les algorithmes utilisés n'offrent eux souvent qu'une garantie de performance très médiocre en $\mathcal{O}(\rho)$. Ce pire cas peut souvent être atteint en considérant un graphe de précédence aussi simple qu'une grille 2D !

Le modèle des tâches malléables est une approche plus récente qui visait à répondre à ces problèmes. Pour l'instant, l'ordonnancement sous ces modèles reste moins étudié. Le cadre imposé par le modèle des tâches malléables est plus restrictif. Il ne permet qu'une prise en compte implicite des communications et il nécessite que l'utilisateur fournisse une vision structurée de son application. Ce modèle semble cependant particulièrement bien adapté pour des applications telles que la décomposition de domaine ou dans le cadre de l'utilisation concurrente de programmes parallèles sur une même machine parallèle. De nombreux algorithmes d'approximation ont déjà été proposés, et les algorithmes d'ordonnancement sous ce modèle semblent pouvoir être applicable plus facilement.

CHAPITRE

Ordonnancement de tâches malléables

3

Le modèle des tâches malléables est une approche originale pour paralléliser une application. Il s'appuie sur un modèle d'application dans lequel les tâches sont elles mêmes des activités potentiellement parallèles. Une tâche malléable peut ainsi être exécutée sur un nombre variable de processeurs, et sa durée d'exécution dépend de ce nombre de processeurs. Dans ce chapitre nous nous intéressons au problème de l'ordonnancement de tâches malléables avec contraintes de précédence, c'est un problème fondamental qui reste encore peu étudié. Il s'agit de choisir la date et le nombre de processeurs pour exécuter les différentes tâches de l'application tout en respectant ces contraintes. Nous présentons une approche en deux phases pour ce problème. Dans un premier temps, nous calculons une allocation, c'est à dire un nombre de processeurs pour calculer chacune des tâches malléables. Dans un deuxième temps, nous utilisons un algorithme d'ordonnancement de tâches multiprocesseurs (tâches parallèles devant s'exécuter sur un nombre fixé de processeurs) pour calculer dans quel ordre exécuter les différentes tâches. La première phase est basée sur une relaxation du problème initial. La contrainte sur le fait qu'au plus m processeurs travaillent à un instant donné est relâchée, et une nouvelle contrainte plus faible est introduite ; elle exprime le fait que le travail total ne doit pas être trop important. Le problème relaxé peut ensuite être approximé en utilisant soit des techniques d'approximation duales, soit la programmation linéaire en nombre entier selon la structure du graphe de précédence. Cette approche nous a permis d'obtenir des algorithmes d'approximation offrant une garantie constante pour ce problème dans le cas de graphes de précédence de type «arbres» mais également dans le cas de graphes de précédence quelconques.

Sommaire

Dans ce chapitre, nous nous sommes intéressés au problème de l'ordonnancement sous le modèle des tâches malléables avec contraintes de précédence. Le modèle des tâches malléables ainsi que son intérêt pratique sont présentés en détail dans le paragraphe 3.2 du chapitre 1. Les différents algorithmes d'ordonnancement proposés sous ce modèle sont présentés plus en détail dans le paragraphe 2 du chapitre 2. Les résultats présentés dans ce chapitre sont le fruit d'un travail mené en collaboration avec Grégory Mounié, Denis Trystram et Gerhard Woeginger.

1. Notations et définitions

Nous considérons un ensemble de tâches malléables T. Les relations de précédences entre ces différentes tâches sont décrites de façon classique à l'aide d'un graphe de précédence G. Nous notons $\prec$ l'ordre partiel sur les tâches induit par ce graphe de précédence. Le temps d'exécution d'une tâche x sur q processeurs est noté $p_{x,q}$. Le comportement habituel des programmes parallèle nous permet de faire les hypothèses naturelles suivantes sur les temps d'exécution.

Hypothèse 1 [Monotonie].

1. Le temps d'exécution $p_{x,q}$ d'une tâche malléable x est une fonction décroissante du nombre de processeurs q qui l'exécutent.
2. Le travail $w_{x,q} \doteq q.p_{x,q}$ d'une tâche malléable x est une fonction croissante du nombre de processeurs q qui l'exécute.

La figure 1 illustre ces hypothèses. En pratique, l'hypothèse 1 est vérifiée au moins jusqu'à un certain seuil. A partir de ce seuil, l'ajout de processeurs ne fait qu'augmenter le temps d'exécution. Dans un tel cas il est possible d'exécuter la tâche sur un nombre moindre de processeurs pour conserver la validité de l'hypothèse.

L'hypothèse 2 traduit simplement le fait que le surcoût de gestion du parallélisme augmente avec le nombre de processeurs. Visuellement, le travail d'une tâche malléable est simplement l'aire de sa représentation dans un diagramme de Gantt. L'hypothèse signifie que cette aire augmente lorsque l'on augmente le nombre de processeurs qui calculent cette tâche. Cette hypothèse implique que l'ajout d'une fraction de processeurs ne peut diminuer le temps d'exécution plus que cette fraction. Cette hypothèse est très souvent vérifiée sauf pour de problèmes de grandes tailles dans lesquels les effets de la pagination ou de la gestion de cache entrent en ligne de compte.

Nous définissons maintenant plus formellement ce qu'est un ordonnancement sous le modèle des tâches malléables.

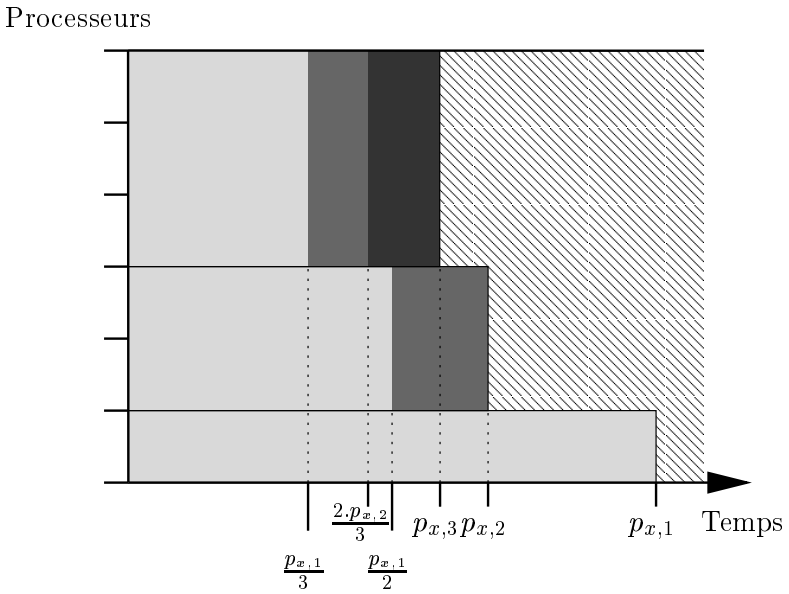

Figure 1 – Exécution d'une tâche malléable x sur un, deux et trois processeurs. La surface gris clair représente le travail sur un processeur $w_{x,1}$, elle est bien sûr constante quel que soit le nombre de processeurs qui exécutent la tâche. La surface gris moyen représente le surcoût de parallélisme dû à l'exécution sur 2 processeurs c'est à dire $w_{x,2} - w_{x,1}$. Enfin la surface gris foncé représente le surcoût de parallélisme dû à l'exécution sur 3 processeurs, c'est à dire $w_{x,3} - w_{x,2}$.

Définition 2 [Ordonnancement]. Un ordonnancement est un couple de fonctions (σ, π). La fonction σ associe à chaque tâche une date de début d'exécution, nous dirons que σ est une *datation*. La fonction π associe à chaque tâche un nombre de processeurs pour exécuter cette tâche, nous dirons que π est une *allocation*.

Étant donné un ordonnancement (σ, π), par souci de concision nous notons σ_x la date d'exécution de la tâche x, et π_x le nombre de processeurs qui l'exécutent. D'autre part nous notons ${c_x}^{(\sigma,\pi)}$ la date de terminaison de la tâche x dans l'ordonnancement (σ, π), plus formellement ${c_x}^{(\sigma,\pi)} = \sigma_x + p_{x,\pi_x}$. Lorsqu'il n'y a pas de confusion possible sur l'ordonnancement considéré, nous notons cette date de terminaison c_x.

Nous nous sommes intéressés aux ordonnancements *valides*. Ils respectent les contraintes de précédence et sont tels que à tout instant au plus m processeurs sont impliqués dans un calcul. Les contraintes de précédence sont respectées si et si seulement pour toute tâche x, et pour tout tâche y successeur de la tâche x, ${c_x}^{(\sigma,\pi)} \leq \sigma_y$. D'autre part si l'on note $X(t)$ l'ensemble des tâches qui sont en cours d'exécution à l'instant t, $X(t) = \{x \in T \mid \sigma_i \leq t < {c_x}^{(\sigma,\pi)}\}$, alors à tout instant t, il y a au plus m processeurs impliqués dans des calculs est équivalent au prédicat suivant : $\forall t, \sum_{x \in X(t)} \pi_x \leq m$.

La *temps d'exécution* d'un ordonnancement (σ, π) noté $\omega_{(\sigma,\pi)}$ est la date de terminaison de la tâche qui finit en dernier : $\omega_{(\sigma,\pi)} = \max_{x \in T} c_x{}^{(\sigma,\pi)}$. Formellement, le problème auquel nous nous intéressons est le suivant :

Définition 3 [Problème de l'ordonnancement de tâches malléables]. Trouver un ordonnancement valide (σ, π) dont le temps d'exécution $\omega_{(\sigma,\pi)}$ soit minimum.

Nous notons dans toute la suite (σ^*, π^*) un ordonnancement optimal, son temps d'exécution est noté ω^*. Nous pouvons notamment remarquer qu'étant donnée une allocation π, le problème de trouver une datation σ pour exécuter les différentes tâches tel que le temps d'exécution de l'ordonnancement (σ, π) soit minimum correspond au problème classique de l'ordonnancement de tâches multiprocesseurs. Une allocation π permet en effet de fixer le temps d'exécution des différentes tâches.

Du point de vue de la complexité, les problèmes de l'ordonnancement de tâches malléables et de tâches multiprocesseurs même indépendantes sont tous deux des problèmes $\mathcal{NP}$-difficile au sens fort [DL89, BDW86]. Du et Leung ont montré notamment que le problème de l'ordonnancement de tâches malléables indépendantes était un problème $\mathcal{NP}$-difficile lorsque le nombre de processeurs est fixé à 5. De plus, les problèmes de l'ordonnancement de tâches malléables et de tâches multiprocesseurs avec contraintes de précédence sont des problèmes pour lesquels il ne peut exister de schéma polynomial d'approximation. En effet, le problème de l'ordonnancement de tâches séquentielles avec précédence est «déjà» $\mathcal{APX}$-complet : il ne peut exister d'algorithme d'approximation dont la garantie soit inférieure à 4/3 [LK78]. Sous cet angle, l'une des principales questions est de savoir s'il existe des algorithmes d'approximation constante pour ces problèmes.

Etant donnée une allocation π et un chemin dans le graphe de précédence G, nous définissons la longueur de ce chemin sous l'allocation π comme la somme des temps d'exécution des tâches sur ce chemin. Le plus long chemin du graphe G est appelé le chemin critique de G. Dans toute la suite, nous notons ω_π^∞ la longueur de ce chemin. Le choix de cette notation vient du fait qu'il s'agit aussi du temps d'exécution de l'ensemble des tâches multiprocesseurs induites par l'allocation π sur un nombre infini de processeurs. D'autre part, nous notons W_π le travail de l'allocation π défini comme $W_\pi = \sum_{x \in T} w_{x,\pi_x} = \sum_{x \in T} \pi_x . p_{x,\pi_x}$. Nous nous sommes intéressés au problème suivant :

Définition 4 [Problème de l'allocation]. Trouver une allocation π tel que le maximum entre la longueur du chemin critique ω_π^∞ et le travail moyen $\frac{W_\pi}{m}$ soit minimum.

La proposition suivante permet de clarifier le rapport entre les solutions optimales du problème de l'ordonnancement de tâches malléables et du problème de l'allocation.

Proposition 5 . La valeur optimale du problème de l'allocation noté $\tilde{\omega}^*$ est plus

petite que le temps d'exécution optimal du problème de l'ordonnancement de tâches malléables ω^* .

Preuve. La preuve de ce résultat est extrêment simple. Comme $\tilde{\omega}^*$ est une solution optimale, pour toute allocation π nous avons $\tilde{\omega}^* \leq \max(\omega_\pi^\infty, \frac{W_\pi}{m})$. En particulier si (σ^*, π^*) désigne un ordonnancement optimal pour le problème de l'ordonnancement de tâches malléables ; on obtient alors :

$$\tilde{\omega}^* \leq \max(\omega_{\pi^*}^\infty, \frac{W_{\pi^*}}{m}) \tag{1}$$

Un ordonnancement valide doit d'une part obéir aux contraintes de précédences donc $\omega_{\pi^*}^\infty \leq \omega^*$, et d'autre part à tout instant il doit y avoir moins que m processeurs impliqués dans des calculs donc $W_{\pi^*} \leq m.\omega^*$. Finalement on obtient :

$$\max(\omega_{\pi^*}^\infty, \frac{W_{\pi^*}}{m}) \leq \omega^* \tag{2}$$

En combinant les équations 1 et 2, on obtient le résultat souhaité. □

Le problème de l'allocation peut être vu, comme une relaxation du problème de l'ordonnancement de tâches malléables. En effet, la contrainte sur le fait qu'il ne doit pas y avoir plus de m processeurs impliqués dans le calcul est supprimée. Par contre une nouvelle contrainte plus faible (car elle est conséquence directe de la précédente) est introduite, il faut que le travail W_π soit plus petit que $m\tilde{\omega}_\pi$.

Dans la partie 2, nous verrons comment construire un algorithme d'approximation pour le problème de l'ordonnancement de tâches malléables à partir d'un algorithme d'approximation pour le problème de l'allocation. Puis dans la partie 3, nous verrons comment approximer ce problème de l'allocation pour le cas d'un graphe de précédence structuré en arbre mais également pour le cas d'un graphe de précédence quelconque. Pour l'instant, nous faisons un petit aparté sur les représentations et sur la contiguïté des ordonnancements malléables.

1.1. Contiguité des ordonnancements

Un ordonnancement, tel que nous le définissons, ne précise pas sur quels processeurs seront exécutées les tâches mais informe seulement du nombre de processeurs qui calculent chacune des tâches. Ainsi la représentation d'un ordonnancement à l'aide d'un diagramme de Gantt n'est pas unique. Jusqu'alors nous avons toujours utilisé des représentations contiguë des ordonnancements, c'est à dire telles que toutes les tâches soient représentées par un rectangle.

S'il est souvent possible de trouver à partir d'un ordonnancement une représentation contiguë, cela n'est pas toujours possible. La figure 2 illustre le cas d'un

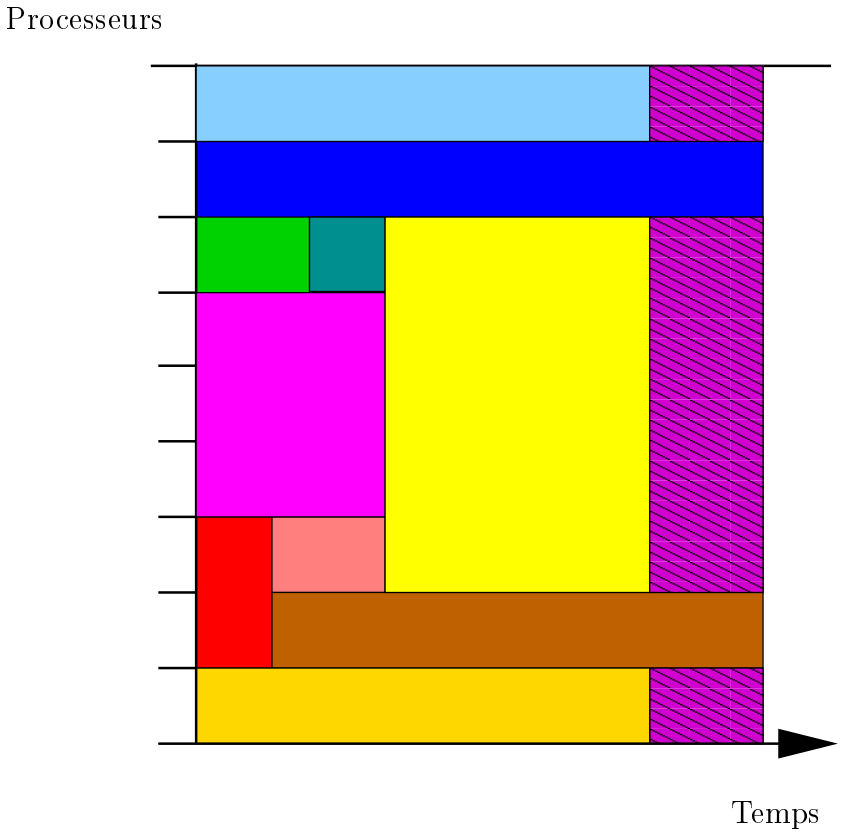

Figure 2 – La tâche qui termine en dernier n'est pas exécutée sur des processeurs contigus et il n'existe pas d'ordonnancement équivalent tel que toutes les tâches soient exécutées sur des processeurs contigus.

ordonnancement pour lequel il n'existe pas de représentation contigu. Une question délicate que l'on peut alors se poser est de savoir ce que l'on peut perdre si on se limite aux ordonnancements pour lequels il existe une représentation contigu. Cette question n'est pas seulement ludique et peut trouver un intérêt pratique si l'on considère la topologie du réseau sur lequel s'exécute l'application.

2. De l'allocation vers l'ordonnancement

Pour construire un algorithme d'approximation pour le problème de l'ordonnancement de tâches malléables, nous allons tout d'abord étudier ce problème à allocation

fixée : c'est le problème de l'ordonnancement de tâches multiprocesseurs. Dans toute la suite de ce paragraphe, nous notons $\bar{\pi}$ l'allocation fixée et $\bar{\sigma}$ la datation que l'on recherche. Le temps d'exécution de l'ordonnancement $(\bar{\sigma}, \bar{\pi})$ est lui noté $\bar{\omega}$ par souci de concision. Nous allons maintenant montrer un résultat préliminaire intéressant : sous certaines hypothèses sur l'allocation $\bar{\pi}$ considérée, il est possible d'ordonnancer l'ensemble de tâches multiprocesseurs induit avec une garantie constante en utilisant un algorithme de liste.

2.1. Ordonnancement de tâches multiprocesseurs

Les algorithmes de liste ont été introduit par Graham [Gra66]. Leur principe est extrêmement simple ; il consiste à ne pas laisser des processeurs inoccupés si une tâche qui n'a pas encore été ordonnancée peut l'être. L'algorithme 2 décrit plus formellement les algorithmes de liste. Il construit itérativement et chronologiquement la datation $\bar{\sigma}$. La variable *Pret* désigne l'ensemble des tâches prêtes c'est à dire l'ensemble des tâches dont tous les prédécesseurs ont déjà été ordonnancés. La variable *Ordonnance* désigne l'ensemble des tâches qui ont déjà été ordonnancées. La fonction $\texttt{Date}(x, (\bar{\sigma}, \bar{\pi}))$ retourne la date au plus tôt à partir de laquelle la tâche x peut être ordonnancée à partir de l'ordonnancement partiel $(\bar{\sigma}, \bar{\pi})$ dans lequel tous les prédécesseurs de x sont déjà ordonnancés. La fonction $\texttt{Pred}(x)$ retourne tous les prédécesseurs de la tâche x.

Algorithme 2 `AlgorithmeDeListe`$(\bar{\pi})$

$Pret = \{x \in T \mid \texttt{Pred}(x) = \emptyset\}$;
$Ordonnance = \emptyset$;
tant que $Pret \neq \emptyset$ **faire**
  $DateMini = \min\{\texttt{Date}(x, (\bar{\sigma}, \bar{\pi})) \mid x \in Pret\}$;
  $y = \texttt{Choix}(\{x \in Pret \mid \texttt{Date}(x, \sigma, \pi) = DateMini\})$;
  $\bar{\sigma}_y = DateMini$;
  $Ordonnance = Ordonnance \cup \{y\}$
  $Pret = \{x \in T \mid \texttt{Pred}(x) \subset Ordonnance\}$
fin tant que

Nous étudions maintenant le comportement d'un algorithme de liste, en fonction du nombre de processeurs maximum nécessaire pour exécuter une tâche multiprocesseur. Nous notons ce nombre de processeurs $\delta = \max_{x \in T} \bar{\pi}_x$. Dans la suite de ce paragraphe nous notons $\bar{\omega}$ le temps d'exécution d'un ordonnancement obtenu avec un algorithme de liste.

Proposition 6 [Garantie de performance d'un algorithme de liste]. Le temps d'exécution $\bar{\omega}$ d'un ordonnancement obtenu grâce à un algorithme de liste vérifie :

$$\bar{\omega} \leq \frac{W_{\bar{\pi}} + (m - \delta).\omega_{\bar{\pi}}^{\infty}}{m - \delta + 1}$$

Preuve. La preuve est une simple généralisation de l'analyse de Graham. On partitionne l'intervalle de temps $[0, \bar{\omega}]$ en deux ensembles I^+ et I^- qui correspondent respectivement aux instants où strictement plus de $m - \delta$ sont occupés et moins de $m - \delta$ processeurs sont occupés. Nous notons $|I^+|$ et $|I^-|$ la durée des périodes I^+ et I^-.

Le travail total effectué pendant la période I^+ est supérieur à $(m - \delta + 1).|I^+|$ et celui effectué pendant la période I^- est supérieur à $|I^-|$. Nous obtenons donc facilement l'équation :

$$W_{\pi} \geq (m - \delta + 1).|I^+| + |I^-| \tag{3}$$

Nous allons maintenant borner la durée de la période I^-. Pour cela, nous construisons un chemin $\mathcal{P}$ dans la fermeture transitive du graphe G. La dernière tâche du chemin $\mathcal{P}$ noté x_1 est l'une des tâches qui termine en dernier. Supposons que les i dernières tâches du chemin $\mathcal{P}$ soient définies. Ces tâches sont notées $x_1, \cdots x_i$ et par définition $x_i \prec \cdots \prec x_1$. S'il n'existe pas d'instant appartenant à I^- avant la date de début d'exécution de la tâche x_i notée σ_{x_i} alors la construction du chemin $\mathcal{P}$ est terminée.

Dans le cas contraire, nous notons t le plus grand de ces instants $t = \sup\{t \in I^- \mid t < \bar{\sigma}_{x_i}\}$. Il existe nécessairement un ancêtre de la tâche x_i qui s'exécute ou termine son exécution à l'instant t. En effet dans le cas contraire la tâche x_i aurait pu être exécutée plus tôt, ce qui est contradictoire avec un algorithme de liste. Nous définissons x_{i+1} comme l'un de ces ancêtres.

Par construction, à tout instant de I^-, il existe une tâche du chemin $\mathcal{P}$ qui s'exécute. Nous pouvons donc très facilement borner la durée de la période I^-, par la longueur du chemin $\mathcal{P}$ et comme $\omega_{\bar{\pi}}^{\infty}$ est une borne supérieure de la longueur de ce chemin, on en déduit l'équation suivante :

$$\omega_{\bar{\pi}}^{\infty} \geq |I^-| \tag{4}$$

Par construction $\bar{\omega} = |I^+| + |I^-|$, en multipliant l'équation 4 par $m - \delta$ et en ajoutant le résultat à l'equation 3 on obtient facilement le résultat souhaité. □

On peut déduire de la proposition précédente le corollaire suivant dont la preuve est immédiate.

Corollaire 7 La garantie de performance d'un algorithme de liste pour l'ordonnancement de tâches multiprocesseurs est $\frac{2m-\delta}{m-\delta+1}$.

Dans le cas de tâches séquentielles, nous pouvons remarquer que cette garantie correspond à la borne bien connue d'un algorithme de liste. En effet, si $\delta = 1$, la garantie vaut $2 - \frac{1}{m}$. Lorsque le nombre de processeurs qui exécutent une tâche n'est pas limité c'est à dire si $\delta = m$, la garantie obtenue est mauvaise. La borne du temps d'exécution de l'ordonnancement correspond dans ce cas au travail de l'allocation. Si l'on considère des tâches malléables, le travail de l'allocation est nécessairement plus grand que le temps d'exécution séquentielle (d'après les hypothèses de monotonies).

Cependant, si $\delta = \frac{m}{2}$, la garantie obtenue est intéressante, elle peut être bornée par la constante 3. Pour construire un algorithme d'approximation pour le problème de l'ordonnancement de tâches malléables, nous utilisons ce «bon» comportement des algorithmes de liste pour l'ordonnancement de tâches multiprocesseurs.

2.1.1. Difficultés de l'ordonnancement de tâches multiprocesseurs

Il nous est apparu impossible, lorsqu'aucune condition sur le nombre maximum de processeurs n'est spécifiée, d'obtenir des garanties relativement à une borne basée sur le maximum entre le travail moyen et le chemin critique. Je pense qu'il s'agit là de l'une des principales difficultés du problème de l'ordonnancement de tâches multiprocesseurs.

Proposition 8 . Pour tout entier k, il est possible de trouver une instance c'est à dire un graphe G, une allocation π et un nombre de processeurs m tel que $\omega_\pi^* \geq k.\max(\omega_\pi^\infty, W_\pi/m)$ où ω_π^* désigne la durée d'exécution de l'ordonnancement optimal basé sur l'allocation π.

Preuve. Le graphe de précédence que nous considérons est composé de n chaînes, et nous supposons que le nombre de processeurs m est égal au nombre de chaînes. La figure 3 illustre le graphe de précédence et l'allocation que nous considérons pour $m = n = 3$. L'instance est telle que $\omega_\pi^\infty = W_\pi/m = 1 + \epsilon$.

Lorsque l'une tâche de la i^{eme} chaîne de durée $1/2^i$ est exécutée au plus une tâche de chacune des tâches suivantes sont exécutées en même temps, et le travail effectué pendant que cette tâche est exécutée est au plus de $1/2^{(i-1)}$. On en déduit que pour exécuter toutes les tâches il faut au moins $(n+1)/2$ unités de temps. Le résultat en découle de manière immédiate. □

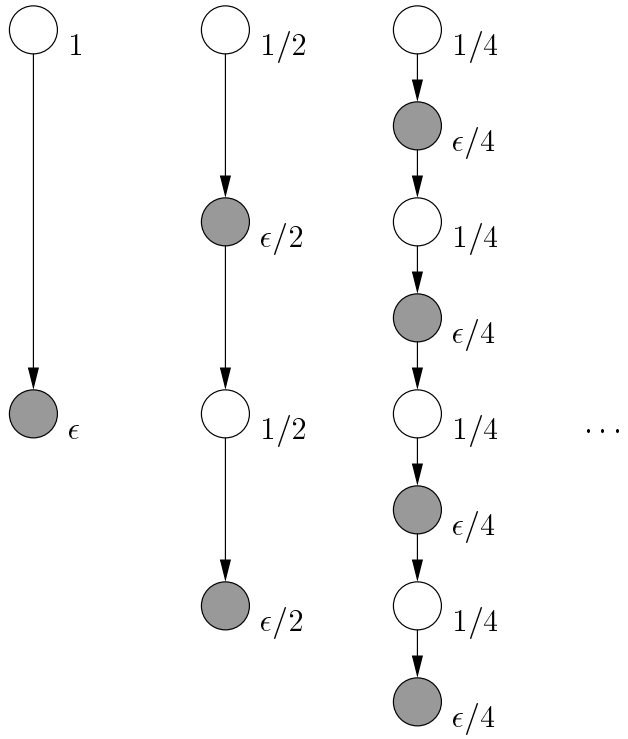

Figure 3 – Ce graphe de précédence est composé de n chaînes, les tâches grisées doivent s'exécuter sur m processeurs et les autres doivent s'exécuter sur un processeur. Les durées des tâches sont notées à coté de celles-ci.

2.2. Choix d'une bonne allocation

Pour construire un algorithme d'approximation pour le problème de l'ordonnancement de tâches malléables, nous utilisons le «bon» comportement des algorithmes de liste pour l'ordonnancement de tâches multiprocesseurs lorsque le nombre maximum de processeurs est peu important. Notre idée est simple, elle consiste à limiter le nombre maximum de processeurs qui sont utilisés pour calculer les différentes tâches. Cette idée permet d'aboutir à la proposition suivante :

Proposition 9 . S'il existe un algorithme de k-approximation pour le problème de l'allocation alors il est possible de construire un algorithme de $4k$-approximation pour le problème de l'ordonnancement de tâches malléables.

Preuve. Soit π la solution du problème de l'allocation obtenue par l'algorithme de k-approximation. D'après la proposition 5, $\max(\omega_\pi^\infty, \frac{W_\pi}{m}) \leq k.\omega^*$, où ω^* désigne le

temps d'exécution d'un ordonnancement optimal. Nous construisons à partir de π une allocation $\bar{\pi}$ dans laquelle aucune tâche ne s'exécute sur plus de δ processeurs. Celle-ci est définie formellement par $\bar{\pi}_x = \pi_x$ si $\pi_x \leq \delta$ et $\bar{\pi}_x = \delta$ sinon.

Un algorithme de liste peut être utilisé pour ordonnancer les tâches multiprocesseurs induites par l'allocation $\bar{\pi}$. D'après la proposition 6, si ω désigne le temps d'exécution de l'ordonnancement obtenue nous avons $\omega.(m - \delta + 1) \leq W_{\bar{\pi}} + (m - \delta).\omega^{\infty}_{\bar{\pi}}$. D'après l'hypothèse de monotonie 2 décrite dans le paragraphe 1 le travail de l'allocation $\bar{\pi}$ est plus petit que le travail de l'allocation π, c'est-à-dire $W_{\bar{\pi}} \leq W_{\pi}$. D'après la même hypothèse pour toute tâche x nous avons aussi $\delta.p_{x,\bar{\pi}_x} \leq m.p_{x,\pi_x}$. On peut en déduire $\omega^{\infty}_{\bar{\pi}} \leq \frac{m}{\delta}.\omega^{\infty}_{\pi}$

En tenant compte du fait que π est une solution obtenue par un algorithme de k-approximation, nous obtenons finalement :

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

Ce rapport est minimum pour $2\delta = m + 1$. Lorsque m est impair si nous choisissons $\delta = \frac{m+1}{2}$ nous obtenons $\frac{\omega}{\omega^*} \leq \frac{4k}{(1+1/m)^2}$. Si m est pair en choisissant $\delta = \frac{m}{2}$, on obtient $\frac{\omega}{\omega^*} \leq \frac{4k}{1+2/m}$ et dans tous les cas, nous avons $\frac{\omega}{\omega^*} \leq 4$ □

2.3. Vers une meilleure garantie

Il est possible d'obtenir une meilleure garantie à partir d'une solution au problème de l'allocation en analysant plus finement la situation. L'idée consiste à combiner l'analyse de la garantie d'un algorithme de liste avec le procédé qui limite le nombre de processeurs pour exécuter chaque tâche. Cela nous permet d'aboutir à la proposition suivante :

Proposition 10 . S'il existe un algorithme de k-approximation pour le problème de l'allocation alors il est possible de construire un algorithme de $r(m).k$-approximation pour le problème de l'ordonnancement de tâches malléables, où $r(m)$ est une fonction du nombre de processeurs m bornée par $\frac{3+\sqrt{5}}{2}$.

Preuve. La preuve est très proche de celle décrite précédemment mais légèrement plus technique. Soit π l'allocation obtenue par l'algorithme de k-approximation pour le problème de l'allocation. Nous définissons l'allocation $\bar{\pi}$ par $\bar{\pi}_x = \pi_x$ si $\pi_x \leq \delta$ et $\bar{\pi}_x = \delta$ sinon, où δ désigne une constante. Nous étudions maintenant le comportement d'un algorithme de liste basé sur l'allocation $\bar{\pi}$ en fonction de δ. Nous notons $\bar{\omega}$ la durée de l'ordonnancement obtenu. Nous partitionnons l'intervalle de temps $[0, \bar{\omega}]$ en trois ensembles I^+, I et I^- qui correspondent respectivement aux instants où

strictement plus de $m - \delta$ sont occupés, entre $m - \delta$ et δ processeurs sont occupés et strictement moins de δ processeurs sont occupées. Nous notons $|I^+|$, $|I|$ et $|I^-|$ la durée des périodes I^+, I et I^-.

Par définition, nous avons $\bar{\omega} = |I^-| + |I| + |I^+|$. D'autre part, pendant la période I^- au moins un processeur exécute une tâche, pendant la période I au moins δ processeurs exécutent des tâches et pendant la période I^+ au moins $(m - \delta + 1)$ processeurs exécutent des tâches. On en déduit donc l'équation suivante :

$$(m - \delta + 1).\bar{\omega} \leq W_{\bar{\pi}} + (m - \delta)|I^-| + (m - 2\delta + 1)|I| \tag{5}$$

Nous allons maintenant montrer un résultat intermédiaire qui nous sera très utile pour borner le second terme de l'inégalité précédente :

$$|I^-| + \frac{\delta}{m}|I| \leq \omega_{\pi}^{\infty} \tag{6}$$

Pour prouver que cette inéquation est bien vérifiée nous construisons un chemin $\mathcal{P}$ dans le graphe G de la même façon que dans la preuve de la proposition 6. La dernière tâche du chemin $\mathcal{P}$ notée x_1 est l'une des tâches qui termine en dernier. Supposons que les i dernières tâches $x_i \prec \cdots \prec x_1$ du chemin $\mathcal{P}$ soient définies. S'il n'existe pas d'instant de $(I^- \cup I)$ avant la date de début d'exécution de la tâche x_i alors la construction du chemin $\mathcal{P}$ est terminée. Dans le cas contraire, nous notons t le plus grand de ces instants $t = \sup\{t \in (I^- \cup I) \mid t < \sigma_{x_i}\}$. Il existe nécessairement un ancêtre de la tâche x_i qui s'exécute ou termine son exécution à l'instant t. En effet dans le cas contraire la tâche x_i aurait pu être exécutée plus tôt, ce qui est contradictoire avec le fonctionnement d'un algorithme de liste. Nous définissons x_{i+1} comme l'un de ces ancêtres. Par construction, à tout instant de $(I^- \cup I)$, il existe une tâche du chemin $\mathcal{P}$ qui s'exécute. Nous calculons maintenant la longueur du chemin $\mathcal{P}$ relativement à l'allocation initiale π.

Toute tâche x qui est exécutée pendant la période I^- est exécuté sur strictement moins de δ processeurs, dans ce cas $p_{x,\bar{\pi}_x} = p_{x,\pi_x}$. D'autre part le temps d'exécution de toute tâche x exécutée pendant I vérifie $\delta.p_{x,\bar{\pi}_x} \leq m.p_{x,\pi_x}$ d'après les hypothèses de monotonies. On en déduit que la longueur du chemin $\mathcal{P}$ relativement à l'allocation π est plus grande que $|I^-| + \frac{\delta}{m}|I|$. Comme ω_{π}^{∞} est une borne supérieure de la longueur du chemin $\mathcal{P}$ relativement à l'allocation π, nous aboutissons à l'équation 6.

Pour terminer l'analyse, nous distinguons deux cas :

- Cas 1 : $m.(m - 2\delta + 1) \leq \delta.(m - \delta)$
  L'équation 5 est équivalente à $(m-\delta+1).\bar{\omega} \leq W_{\bar{\pi}} + (m-\delta)(|I^-| + \frac{\delta}{m}|I|)$. D'après les hypothèses de monotonies et l'équation 6 que nous venons de montrer, on obtient : $(m - \delta + 1).\bar{\omega} \leq W_{\pi} + (m - \delta)\omega_{\pi}^{\infty}$. Comme π a été obtenu par un

algorithme de k-approximation nous avons $\bar{\omega}/\omega^* \leq k.(2m-\delta)/(m-\delta+1)$ mais aussi $\bar{\omega}/\omega^* \leq km/\delta$ par hypothèse sur le cas considéré.

- Cas 2 : $m.(m-2\delta+1) > \delta.(m-\delta)$
  L'équation 5 est équivalente à $(m-\delta+1).\bar{\omega} \leq W_{\bar{\pi}} + (m-2\delta+1)(|I^-| + \frac{\delta}{m}|I|)$. En utilisant les mêmes arguments que dans le 1er cas on obtient aussi $\bar{\omega}/\omega^* \leq \max(km/\delta, k.(2m-\delta)/(m-\delta+1))$.

Si nous choisissons δ de façon à minimiser $r(m,\delta) = \max(m/\delta, (2m-\delta)/(m-\delta+1))$, et que nous posons $r(m) = \min_{1\leq\delta\leq m} r(m,\delta)$ alors nous obtenons un algorithme de $k.r(m)$-approximation.

La fonction $\alpha \to m/\alpha$ est décroissante entre 1 et m et la fonction $\alpha \to (2m-\alpha)/(m-\alpha+1)$ est croissante sur cet intervalle. La fonction $\alpha \to r(m,\alpha)$ définie sur $[1,m]$ est donc minimum lorsque α est tel que $m.(m-2\alpha+1) = \alpha.(m-\alpha)$, c'est à dire pour $\alpha = (m/2)(3-\sqrt{5-4/m})$. La valeur qu'il faut choisir pour δ est donc soit l'entier immédiatement au dessus ou immédiatement au dessous de α, et lorsque m tend vers l'infini le rapport δ/m tend vers $(3-\sqrt{5})/2 \simeq 0.38$. En évaluant l'expression $r(m,\delta)$ pour $\delta = \alpha$ et $\delta = \alpha+1$, intervalle sur lequel $r(m)$ est croissante on en déduit :

$$r(m) \leq \frac{3+\sqrt{5}}{2} \simeq 2.62$$

□

3. Approximation du problème de l'allocation

Dans la partie précédente nous avons montré que le problème de l'allocation apparaissait comme un problème clé pour obtenir des algorithmes d'approximation pour le problème de l'ordonnancement de tâches malléables. Nous pouvons remarquer que le problème de l'allocation est un problème $\mathcal{NP}$-difficile au sens fort dans le cas général cela découle des arguments de Lenstra et Rinnooy Kan [LK78]. D'autre part, pour le cas d'une chaîne le problème est $\mathcal{NP}$-difficile au sens faible, il est très facilement réductible à un problème de type sac à dos. Dans cette partie, nous montrons comment approximer ce problème d'abord dans le cas de graphes de précédence de type arbres puis dans le cas général.

3.1. Cas des arbres

La version de décision du problème de l'allocation consiste à répondre à la question suivante : « Etant donné un entier λ, existe t-il une allocation π tel que $\max(\omega_\pi^\infty, \frac{W_\pi}{m}) \leq \lambda$? » Dans le cas d'un graphe de précédence de type «arbres» nous allons montrer qu'il est possible de construire un algorithme pseudo-polynomial, c'est-à-dire dépendant polynomialement de la taille de l'instance et de λ, pour répondre à cette question. Nous montrerons ensuite comment cet algorithme nous permet de construire un schéma d'approximation pleinement polynomial pour le problème de l'allocation (fully-polynomial time approximation scheme).

3.1.1. Programmation dynamique

Nous considérons un entier λ fixé et nous supposons que le graphe de précédence G est de type arbre entrant et qu'il est composé de n sommets. (le cas des arbres sortants est très similaire). Les sommets du graphe sont numérotés de 1 à n avec un ordre compatible avec l'ordre partiel du graphe G. Nous notons respectivement $w_{i,q}$ et $p_{i,q}$ le travail et le temps d'exécution de la tâche d'indice i sur q processeurs. Nous désignons par $W(i,t)$ le travail minimum pour exécuter les i tâches correspondant aux sommets numérotées de 1 à i en une durée inférieure à t sur une infinité de processeurs, et par $prec(i)$ l'ensemble des indices des tâches prédécesseurs de la tâche d'indice i.

Algorithme 3 Algorithme de programmation dynamique.

pour tout $t \geq 0$ **faire**
  $W(0,t) = 0$;
  $W(0,-t) = +\infty$;
fin pour
pour tout $t \in [1,\lambda]$ **faire**
  pour tout $i \in [1,n]$ **faire**
    $W(i,t) = min_{q=1..m} w_{i,q} + \sum_{j \in prec(i)} W(j, t - p_{i,q})$
  fin pour
fin pour

L'algorithme de programmation dynamique 3 calcule $W(i,t)$ pour i compris entre 1 et n et t compris entre 0 et λ, il permet de répondre au problème de décision posé. Sa complexité dépend de $\mathcal{O}(n.m.\lambda)$, il est donc pseudo-polynomial. Il serait tout à fait possible de retourner une allocation π en utilisant un algorithme de marquage et en gardant la même complexité.

3.1.2. Vers un schéma d'approximation polynomial

Nous allons maintenant montrer comment «transformer» l'algorithme pseudo-polynomial que nous venons de construire en un schéma pleinement polynomial d'approximation pour le problème de l'allocation. L'idée que nous proposons est inspirée des techniques d'approximations duales introduites par Schmoys et Hochbaum [HS87].

Dans notre cadre, un algorithme de k-approximation dual répond, en temps polynomial, à la question « Existe t-il une allocation de valeur inférieure à λ ? » soit par non, soit par une allocation π tel que $\max(\omega_\pi^\infty, \frac{W_\pi}{m}) \leq k\lambda$. D'autre part, d'après les hypothèses de monotonies la valeur optimale du problème de l'allocation est compris entre 0 et le temps d'exécution séquentiel de l'ensemble des tâches $W_1 = \sum_{x \in T} p_{x,1}$. En effectuant une recherche binaire sur l'intervalle $[0, W_1]$ et en appliquant à chaque itération un algorithme de k-approximation dual, nous pouvons donc trouver un algorithme de k-approximation pour le problème de l'allocation.

Etant donné un réel positif ϵ, nous allons construire un algorithme de $(1+\epsilon)$-approximation dual dont la complexité soit en $\mathcal{O}(1/\epsilon)$. Cela nous permettra d'obtenir un schéma d'approximation pleinement polynomial pour le problème de l'allocation. L'idée consiste à utiliser notre algorithme de programmation dynamique sur une instance approximée et moins précise. Soit une instance I du problème de l'allocation, et supposons que $\tilde{\omega}^*$ soit une borne supérieure de la valeur d'une allocation optimale. Notre objectif est de répondre en temps polynomial en la taille de l'instance et en $\mathcal{O}(1/\epsilon)$ à la question suivante :

$$\text{Trouver une allocation } \pi \text{ telle que } \max(\omega_\pi^\infty, \tfrac{W_\pi}{m}) \leq \tilde{\omega}^*(1+\epsilon) \,?$$

Pour répondre à cette question nous construisons une nouvelle instance I' à partir de l'allocation I. L'instance I' est composée du même nombre de tâches n avec les mêmes relations de précédence mais les temps d'exécution des tâches sont différents. Ils sont définis par $p'_{i,q} = \lfloor \frac{p_{i,q}}{k} \rfloor$ où k désigne une constante que nous fixerons par la suite. Par construction l'instance I' admet une solution optimale dont la valeur est inférieure ou égale à $\lfloor \frac{\tilde{\omega}^*}{k} \rfloor$. En choisissant $\lambda' = \frac{\tilde{\omega}^*}{k}$, l'algorithme pseudo-polynomial 3 permet donc de trouver une allocation que nous noterons $\tilde{\pi}$. Dans le cas où l'algorithme 3 échoue c'est que notre hypothèse initiale était fausse et que $\tilde{\omega}^*$ était plus petit que la valeur optimale (la réciproque est bien entendue fausse !). Considérons maintenant un chemin $\mathcal{P}$ dans le graphe G, nous avons la relation suivante :

$$\sum_{x \in \mathcal{P}} p_{x,\tilde{\pi}_x} \leq \sum_{x \in \mathcal{P}} k.(p'_{x,\tilde{\pi}_x} + 1) \leq kn + k. \sum_{x \in \mathcal{P}} p'_{x,\tilde{\pi}_x} \tag{7}$$

Comme $\tilde{\pi}$ est optimale pour l'instance I', nous avons $k.\sum_{x\in\mathcal{P}} p'_{x,\tilde{\pi}_x} \leq \tilde{\omega}^*$. D'après l'équation 7 nous en déduisons que $\omega^\infty_{\tilde{\pi}} \leq kn + \tilde{\omega}^*$. D'autre part, nous avons $W_{\tilde{\pi}} \leq k.\sum_{x\in T} \tilde{\pi}_x(p'_{x,\tilde{\pi}_x}+1)$. En considérant l'optimalité de $\tilde{\pi}$ pour l'instance I', nous obtenons $W_{\tilde{\pi}} \leq knm + m\tilde{\omega}^*$. Finalement, nous obtenons :

$$\max(\omega^\infty_{\tilde{\pi}}, \frac{W_{\tilde{\pi}}}{m}) \leq kn + \tilde{\omega}^*$$

En choisissant k tel que $kn = \epsilon.\tilde{\omega}^*$, on obtient le résultat souhaité. Il reste à vérifier que le temps de calcul de l'algorithme de programmation dynamique reste bien polynomial. Sa complexité est de $\mathcal{O}(n.m.\lambda)$ où $\lambda = \tilde{\omega}^*/k = n/\epsilon$, sa complexité est donc $\mathcal{O}(\frac{m.n^2}{\epsilon})$.

3.2. Cas d'un graphe général

Le problème de l'allocation est très proche d'un problème bien connu en gestion de projet : le problème du compromis coût/temps [1] [DDGW97]. Une instance de ce problème est décrite par différentes procédures qui doivent être réalisées. Ces procédures sont liées entre elles par des contraintes de précédence et peuvent être réalisées de différentes façons avec un coût et une durée variable. Un projet consiste à choisir pour chaque procédure une façon de la réaliser parmi les différentes alternatives. Un projet permet ainsi de fixer le coût total ainsi que la durée de réalisation (longueur de la plus longue chaîne). Ce problème est un problème bi-critère puisqu'il s'agit de trouver un projet qui minimise à la fois le coût et la durée du projet. Dans la variante «budget», il s'agit étant donné un budget B de trouver un projet minimisant la durée de réalisation tout en ne dépassant pas le budget alloué, nous notons $D^*(B)$ la durée de réalisation d'un projet optimal ne dépassant pas le budget B. En arrondissant la solution d'une relaxation d'un programme linéaire, Skutella [Sku98] a proposé un algorithme polynomial permettant de trouver un projet dont le coût total est au plus $2B$ et dont la durée est au plus $2D^*(B)$.

Ce résultat permet de construire facilement un algorithme de 2-approximation pour le problème de l'allocation. En effet une tâche malléable correspond à une procédure pouvant être exécutée de m façons différentes. Le temps d'exécution d'une tâche correspond alors au temps de réalisation de la procédure associée, et son travail correspond à son coût. Une allocation π correspond alors à un projet dont la durée total est ω^∞_π et dont le coût est W_π. En effectuant une recherche binaire et en appliquant à chaque itération l'algorithme de Skutella nous pouvons obtenir une allocation $\tilde{\pi}$ qui vérifie $\max(m\omega^\infty_{\tilde{\pi}}, W_{\tilde{\pi}}) \leq 2m\tilde{\omega}^*$ où $\tilde{\omega}^*$ désigne la valeur de l'allocation optimale. Nous obtenons ainsi un algorithme d'approximation de garantie 2

[1] «Discrete time-cost tradeoff problem» en anglais

pour le problème de l'allocation. D'après la proposition 10, nous obtenons donc un algorithme d'approximation de garantie $3 + \sqrt{5}$ dans le cas général.

4. Conclusion et perspectives

Dans ce chapitre nous avons présenté une approche en deux phases pour l'ordonnancement de tâches malléables avec des contraintes de précédences. Dans un premier temps, nous proposons de calculer une allocation à partir d'une relaxation du problème de l'ordonnancement de tâches malléables. Puis nous adaptons cette allocation afin qu'elle puisse être utilisée efficacement par un algorithme de liste qui ordonnance des tâches multiprocesseurs. Cette approche nous a permis d'obtenir des algorithmes d'approximation constante aussi bien dans le cas de graphes particuliers tels que les arbres que dans le cas de graphes quelconques, c'est l'un des premiers résultats dans ce sens. Nous avons ainsi proposé un schéma d'algorithmes polynomiaux de garanties $(\frac{3+\sqrt{5}}{2}) + \epsilon$ dans le cas des arbres et un algorithme d'approximation de garantie $3 + \sqrt{5}$ dans le cas d'un graphe quelconque. Il serait sans doute encore possible d'améliorer ces garanties mais il semble plus intéressant d'essayer de valider pratiquement ces heuristiques.

Dans un premier temps, une validation en «moyenne» du comportement de ces heuristiques par des simulations pourrait être entreprise. En effet, les résultats obtenus ici sont des bornes au pire du comportement des algorithmes proposés. Ces bornes ne sont sans doute atteintes que dans de rares cas (si elles le sont). De plus l'ordonnancement obtenu doit souvent pouvoir être corrigé a posteriori afin d'obtenir un meilleur résultat. Dans certains ordonnancements obtenus, il par exemple possible d'allouer plus de processeurs à certaines tâches sans pour autant retarder l'exécution d'autres tâches ! La principale difficulté pour effectuer une telle analyse statistique reste d'obtenir une bonne borne inférieure du temps d'exécution optimal.

Dans un second temps, ces algorithmes pourrait être utilisés pour paralléliser des applications se prêtant bien au modèle des tâches malléables. Dans ce cadre, il serait sans doute intéressant de ne conserver que la partie allocation (le nombre de processeurs alloués à chaque tâche) et de laisser un système d'exécution réguler dynamiquement la charge afin de corriger les incertitudes probables sur les durées d'exécutions et les éventuelles problèmes dues à la charge des machines. La garantie offerte serait la même si le système de régulation est basé sur un algorithme de liste, c'est-à-dire s'il ne laisse pas des processeurs inoccupés si une tâche peut être exécutée. La parallélisation offerte serait probablement plus robuste.

CHAPITRE

4

Regroupement

Dans ce chapitre nous nous intéressons au problème du regroupement avec grand temps de communication, c'est à dire au problème de l'ordonnancement sur un nombre infini de processeurs. Nous proposons une approche originale de ce problème basée sur une décomposition récursive du graphe de précédence. Le cœur de cette décomposition est une découpe consistant à trouver deux grands groupes de tâches indépendants, c'est-à-dire non liés par des relations de précédence. Ces deux groupes de tâches peuvent s'exécuter efficacement en parallèle et sans communication. Cette découpe est alors appliquée récursivement sur les groupes de tâches induits tant que cela s'avère intéressant. Notre approche est validée expérimentalement par des simulations sur des graphes d'applications réelles. Enfin, nous terminons par les perspectives de ce travail et les travaux actuellement en cours autour de cette approche.

Sommaire

1. Introduction

L'exécution efficace d'une application parallèle sur une architecture à mémoire distribuée est étroitement liée à la décision de l'ordonnancement des tâches qui constituent cette application. En effet, les temps dus aux échanges de données sur de telles architectures peuvent souvent être très importants par rapport aux temps de calculs. Pour obtenir une exécution efficace, il apparaît souvent nécessaire de regrouper plusieurs tâches à grain fin pour obtenir un rapport plus équilibré entre les calculs et les communications [Sar89]. Ce problème est connu en anglais sous le nom de «task clustering», nous l'appellerons par la suite le problème du regroupement. Il peut être vu comme un problème d'ordonnancement sur un nombre non borné de processeurs. En pratique, il est souvent possible de «replier» l'ordonnancement obtenu sur un nombre fini de processeurs, ou bien également d'utiliser une technique de vol de travail sur les groupes de tâches obtenus, nous verrons ce point en détail par la suite. Le problème du regroupement est un problème qui a été largement étudié. Cependant il n'existe pas de résultat pleinement satisfaisant, notamment lorsque les temps de communications sont importants. Nous renvoyons le lecteur intéressé au chapitre 2 qui présente un état de l'art du sujet.

Nous proposons dans ce chapitre une heuristique basée sur une décomposition récursive du graphe de précédence. Dans la littérature, on peut trouver un article basé sur une idée similaire [GM89]. La décomposition considérée est une décomposition récursive en «clan». Intuitivement un groupe de tâches est un «clan» si et seulement si toutes les tâches de ce groupe ont les mêmes tâches prédécesseurs et les mêmes tâches successeurs en dehors de ce groupe. L'idée proposée est fondée sur une décomposition récursive canonique d'un graphe de précédence en «clan». Cependant, cette décomposition ne permet pas toujours de détecter du parallélisme car il peut exister des «clans terminaux» (ie. ne pouvant plus être décomposé) de grande taille. Un graphe structuré en grille-2D par exemple ne peut être décomposé en clan bien qu'il soit tout à fait possible de paralléliser efficacement une application décrite par ce graphe. Le principe de la décomposition à laquelle nous nous sommes intéressés est fondé sur une découpe répondant au problème suivant : trouver deux groupes de tâches indépendants les plus grands possibles. Intuitivement deux groupes de tâches sont indépendants s'ils peuvent s'exécuter en parallèle sans communication. En appliquant cette découpe aux deux groupes de tâches obtenus, ainsi qu'aux groupes de tâches qui forment d'une part leurs prédécesseurs, et d'autre part leurs successeurs, cette idée permet de construire une décomposition récursive du graphe de précédence. Cette décomposition offre alors une vision structurée de l'application ; et il est possible d'ordonnancer l'application en utilisant cette décomposition et en s'arrêtant à un seuil de découpe qui dépend de la modélisation des communications.

Le plan de chapitre est le suivant. Nous commençons par présenter plus formellement le problème auquel nous nous intéressons, nous introduisons ensuite quelques notions sur les regroupements, et notamment les regroupements convexes qui sont la structure des regroupements que nous pouvons obtenir avec notre algorithme de décomposition récursive. Nous montrons que sous un modèle de communication de type modèle délai, ces derniers sont 2-*dominants*, c'est à dire qu'il existe un regroupement convexe dont le temps d'exécution est inférieur à 2 fois celui d'un regroupement optimal. Nous présentons ensuite notre algorithme de décomposition en montrant pourquoi le problème de partitionnement de graphe orienté acyclique est intéressant pour ce problème. Puis nous étudierons des heuristiques pour ce problème de partitionnement de graphe orienté acyclique. Enfin, nous simulons cette nouvelle approche sur quelques graphes d'applications réelles, en la comparant avec un algorithme classique pour le problème du regroupement : DSC Dominant Sequence Clustering [GY94a]. Finalement, nous concluons par quelques perspectives de ce travail. Une version préliminaire de ce travail a été présentée à la conférence RenPar'13 [Lep01].

2. Regroupement

Dans toute la suite, nous considérons un graphe de précédence que nous notons $G = (V, E)$. Nous désignons par $\prec$ l'ordre partiel sur les tâches V induit par le graphe G, par $\not\prec$ la relation complémentaire, et par $\sim$ la relation d'équivalence définie par $x \sim y$ si et seulement si $x \not\prec y$ et $y \not\prec x$. Nous nous plaçons dans un cadre simplifié du modèle délai, la durée d'exécution de toutes les tâches est supposée unitaire. Si deux tâches communicantes x et y (ie. (x, y) appartient à E) sont placées sur des processeurs différents alors la tâche y ne peut commencer à s'exécuter que ρ unités de temps après la date de terminaison de la tâche x. Ce délai de communication ρ est constant et indépendant des tâches x et y. Nous nous intéressons au problème des grands délais de communication c'est-à-dire que ρ est grand devant 1. Ce modèle correspond au modèle à communication explicite LCT que nous avons décrit dans le chapitre 1.

Définition 1 [Regroupement]. Un regroupement $R = \{(V_i, \prec_i)\}_i$ est une partition de l'ensemble des tâches, telle que chaque groupe de tâches est muni d'un ordre total d'exécution $\prec_i$ compatible avec l'ordre $\prec$.

Les groupes de tâches V_i d'un regroupement peuvent être vus comme les tâches qui seront exécutées sur un même processeur. Il est possible de construire à partir d'un regroupement R, un *graphe induit* noté G_R^{induit} en introduisant les arcs correspondant aux contraintes introduites par le séquencement sur les groupes de tâches,

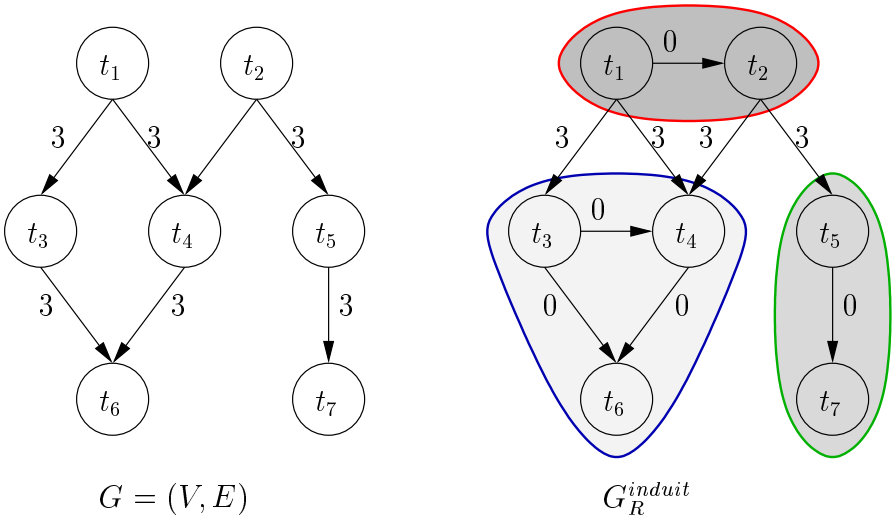

Figure 1 – A gauche, le graphe initial. A droite, le graphe induit correspondant au regroupement $R = \{\{t_1 \prec_1 t_2\}, \{t_3 \prec_2 t_4 \prec_2 t_6\}, \{t_5 \prec_3 t_7\}\}$. La longueur d'un plus long chemin du graphe induit ((t_1, t_3, t_4, t_6) ou bien (t_1, t_2, t_4, t_6)) est $\omega_R = 7$ (le temps des tâches est unitaire.

et en mettant à 0 les latences de communications internes aux différents groupes de tâches. Dans la suite, nous appelons temps d'exécution d'un regroupement R, la durée d'exécution sur une infinité de processeurs de l'ordonnancement induit par R où chaque groupe de tâches est exécuté sur un processeur différent . Nous noterons ω_R le temps d'exécution d'un regroupement R, il correspond aussi au plus long chemin du graphe induit, comme l'illustre la figure 1.

Le problème auquel nous nous intéressons est le suivant :

Définition 2 [Problème du Regroupement]. Trouver un regroupement $R = \{(V_i, \prec_i)\}_i$ du graphe G, tel que le temps d'exécution ω_R soit minimal.

Ce problème est très proche du problème noté $P\infty|prec, p_j = 1, c > 1|C_{max}$ avec la notation à trois champs de la théorie de l'ordonnancement introduite par Lawler et al [Law82] et étendu ensuite par Veltman et al [VLL90] : la notion de date d'exécution pour les tâches est remplacée par une notion d'ordre d'exécution sur les taches exécutées sur un même processeur. Nous noterons par la suite le temps d'exécution optimal ω^*, c'est-à-dire le temps d'exécution d'un regroupement dont le temps d'exécution est minimum.

2.1. Regroupement et groupe de tâches convexes

Une regroupement $R = \{(V_i, \prec_i)\}$ est dit linéaire si et seulement si les ordres $(\prec_i)_i$ sont tous inclus dans l'ordre $\prec$. Intuitivement cela signifie qu'il n'y pas deux tâches indépendantes dans un même groupe de tâches. Les regroupements linéaires ont été largement étudiés [GY93]. En effet les regroupements linéaires sont dominants lorsque les délais de communication sont petits. C'est à dire que si le délai de communication ρ est inférieur ou égal à 1, il existe un regroupement linéaire optimal. Cependant, ce résultat n'est bien évidemment pas vrai si les délais de communication sont supérieurs à 1 même pour des graphes de précédence simples comme les arbres binaires complets, la figure 14 page 79 permet de s'en convaincre. Nous allons maintenant introduire la notion de regroupement convexe. La décomposition récursive que nous allons présenter par la suite permet de construire des regroupements convexes. Mais tout d'abord, nous allons définir formellement ces regroupements, puis nous allons montrer qu'ils sont $2-$*dominants* lorsque les temps de communication sont importants, c'est à dire qu'il existe un regroupement convexe dont le temps d'exécution est inférieur à 2 fois celui d'un regroupement optimal.

Etant donné un regroupement $R = \{(V_i, \prec_i)\}$, alors nous pouvons construire un graphe regroupé G_R^{groupe} défini de la façon suivante : les sommets sont les groupes de tâches du regroupement R c'est-à-dire les ensembles de tâches $\{V_i\}_i$. Dans le graphe G_R^{groupe}, il existe une arête entre V_i et $V_j \neq V_i$ si et seulement si il existe une tâche $x \in V_i$ et une tâche $y \in V_j$ tels que $(x, y) \in E$. D'autre part, à chaque arc est associé le poids ρ, et à chaque sommet V_i est associé le poids $|V_i|$. La figure 2, illustre le graphe regroupé associé au regroupement R défini dans la figure 1.

Définition 3 [Regroupement convexe]. Un regroupement R est convexe si et seulement si le graphe regroupé G_R^{groupe} est acyclique. Nous définissons alors ω_R^{groupe} comme le plus long chemin dans le graphe G_R^{groupe}.

Nous pouvons remarquer que le graphe regroupé d'un regroupement R est indépendant de l'ordre sur les différents groupes de tâches de ce regroupement. D'autre part le lemme suivant nous permet de clarifier le lien entre le plus long chemin dans le graphe induit ω_R et le plus long chemin dans le graphe regroupé ω_R^{groupe}.

Lemme 4 Pour tout regroupement convexe R, le plus long chemin du graphe regroupé ω_R^{groupe} est une borne supérieure de son temps d'exécution ω_R.

Preuve. Soit $R = \{(V_i, \prec_i)\}$ un regroupement, et soit $C = (x_1, \cdots, x_m)$ un chemin dans le graphe induit G_R^{induit}. Soit $(W_i)_i$ les groupes de tâches tels que $x_i \in W_i$ pour tout i dans $[1, m]$. Remarquons que si $W_i = W_k$ alors pour tout j dans $[i, k]$,

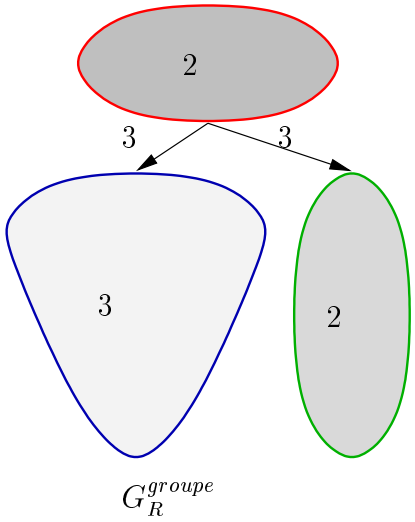

Figure 2 – Le graphe regroupé G_R^{groupe} associé au regroupement $R = \{\{t_1 \prec_1 t_2\}, \{t_3 \prec_2 t_4 \prec_2 t_6\}, \{t_5 \prec_3 t_7\}\}$. La longueur du plus long chemin est $\omega_R^{groupe} = 8$.

$W_i = W_j = W_k$, sinon il y aurait un cycle dans le graphe induit. Le chemin D dans le graphe regroupé G_R^{groupe} définit comme les différents groupes de tâches traversés par le chemin C, a par définition une longueur plus longue que celle de C. Comme cela est vrai pour tout chemin C, le résultat du lemme en découle. □

Nous pouvons également caractériser les groupes de tâches d'un regroupement convexe. Pour cela, nous allons introduire la notion de groupe de tâches convexes.

Définition 5 [Groupe de tâche convexe]. Un groupe de tâches T est convexe si et si seulement pour tout couple de tâches (x, z) de T, toute tâche y tel que $x \prec y \prec z$ est dans T.

La définition d'un groupe de tâches convexes est cohérente avec la définition mathématique habituelle de la convexité.

Proposition 6 [Propriété des groupes de tâches d'un regroupement convexe]. Etant donné un regroupement convexe $R = \{(V_i, \prec_i)\}$, alors pour tout groupe de tâches V_i de ce regroupement V_i est un groupe de tâches convexe.

Preuve. Par l'absurde supposons que V_i ne soit pas convexe, alors il existe x dans V_i, z dans V_i et y dans $V_j \neq V_i$ tel que $x \prec y \prec z$. Comme $x \prec y$ il existe un chemin entre V_i et V_j, et comme $y \prec z$, il existe un chemin entre V_j et V_i. Cela contredit le fait que R est un regroupement convexe, car le graphe G_R^{groupe} est acyclique si et seulement si R est un regroupement convexe. □

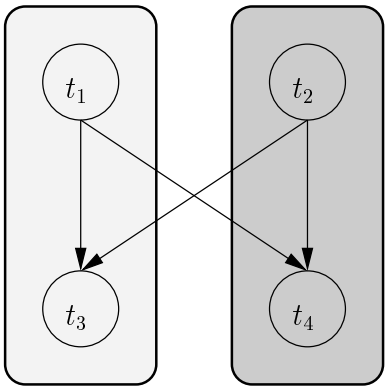

Figure 3 – Un ensemble de groupes de tâches convexes, mais un regroupement non convexe !

Cette proposition justifie le terme «regroupement convexe» que nous avons choisi. Cependant il faut voir qu'il ne s'agit pas d'une condition suffisante. En effet des groupes de tâches convexes, n'implique pas nécessairement un regroupement convexe. La figure 3 permet de s'en convaincre.

Proposition 7 [2-Dominance des regroupements convexes]. Il existe un regroupement convexe R tel que $\omega_R \leq 2.\omega^*$.

Preuve. L'idée consiste simplement à partitionner l'intervalle $[0, \omega^*]$ en sous intervalles de temps ρ et à insérer des temporisations pour les communications entre chacun des intervalles. Considérerons $R^* = \{(V_i^*, \prec_i^*)\}_{i\in[1,k]}$ un regroupement optimal. Nous notons t_x^* la date à laquelle est ordonnancée la tâche x dans R^* ; il s'agit du plus long chemin menant à x dans le graphe $G_{R^*}^{induit}$. Soit q et r, le quotient et le reste de la division euclidienne de ω_R^* par ρ : $\omega_R^* = q.\rho + r$.

Nous construisons alors la famille de groupes de tâches $(V_{i,j})_{i\in[1,k],j\in[0,q]}$ définie par : $V_{i,j} = V_i^* \cap \{x \in V \mid t_x^* \in [j\rho, (j+1)\rho[\}$. Par construction $(V_{i,j})_{i,j}$ défini bien une partition de l'ensemble des tâches V, en effet $(V_{i,j})_j$ défini une partition de V_i. Soit $\prec_{i,j}$ l'ordre sur $V_{i,j}$ défini comme étant la restriction de $\prec_i^*$ à $V_{i,j}$. Le regroupement $R = (V_{i,j}, \prec_{i,j})$ est convexe. En effet, par construction si il existe une arête entre deux groupes de tâches différents V_{i_1,j_1} et V_{i_2,j_2}, alors $j_1 < j_2$ car il faut une durée ρ pour effectuer la communication nécessaire. Soit $\omega_{i,j}^{groupe}$, la longueur du plus long chemin qui mène à $V_{i,j}$ dans le graphe G_R^{groupe}. Comme $|V_{i,j}| \leq \rho$ par récurrence immédiate, nous avons : $\omega_{i,j}^{groupe} = 2.\rho.j + |V_{i,j}|$. De plus comme, $|V_{i,q}| \leq r$, nous obtenons finalement $\omega_R^{groupe} = max_{i\in[1,k]}\omega_{i,q}^{groupe} \leq 2.q + r \leq 2.\omega^*$. Finalement d'après le lemme 4, nous obtenons $\omega_R \leq 2.\omega^*$. □

Nous pouvons remarquer que cette proposition est également vraie pour le problème de l'ordonnancement sur un nombre borné de processeurs avec grand temps

de communication, la preuve est très similaire à celle-ci. D'autre part cette proposition montre aussi comment replier facilement un ordonnancement sur une infinité de processeurs (c'est-à-dire un regroupement) sur un nombre borné de processeurs. Le procédé décrit ici montre qu'il est possible de construire un algorithme d'approximation de garantie constante pour le problème d'ordonnancement sur un nombre borné de processeur à partir d'un algorithme d'approximation de garantie constante pour le problème du regroupement.

Cette proposition justifie le choix que nous avons fait de nous intéresser aux regroupements convexes ainsi qu'aux groupes de tâches convexes pour construire une décomposition.

2.2. Notations

Nous allons maintenant introduire quelques notations sur les groupes de tâches convexes qui nous seront utiles par la suite. Soit T un groupe de tâches convexe, nous notons :

$T^{>}$: les tâches successeurs d'une tâche de T et qui ne sont pas dans T. Plus formellement nous définissons $T^{>}$ comme $\{x \in (V \setminus T) \mid \exists y \in T, y \prec x\}$.

$T^{<}$: les tâches prédécesseurs d'une tâche de T et qui ne sont pas dans T. Plus formellement nous définissons $T^{<}$ comme $\{x \in (V \setminus T) \mid \exists y \in T, x \prec y\}$.

$\widetilde{T}$: les tâches indépendantes de toutes les tâches dans le groupe de tâches T et qui ne sont pas dans T. Plus formellement $\widetilde{T}$ est défini par $V \setminus (T \cup T^{<} \cup T^{>})$.

$\overline{T}$: la fermeture convexe du groupe de tâches T. Formellement $\overline{T}$ est défini par $T \cup \{y \in V \mid \exists x \in T, \exists z \in T, x \prec y \prec z\}$.

Nous pouvons remarquer que les différents groupes de tâches $T^{<}$, $T^{>}$, $\widetilde{T}$ et $\overline{T}$ forment des groupes de tâches convexes.

3. Algorithme de décomposition récursive

Nous recherchons une décomposition récursive du graphe de précédence, formé de groupes de tâches convexes. Pour cela, intuitivement, nous commençons par rechercher un parallélisme à gros grain dans le graphe de précédence. Plus précisément, nous recherchons d'abord deux groupes de tâches convexes et indépendants (ie. non liés par des relations de précédence) A et B tels que le plus petit des deux soit le

plus grand possible, ces 2 groupes de tâches pourront s'exécuter en parallèle et sans communication. Si une telle découpe permet d'obtenir un regroupement tel que le plus long chemin de son graphe regroupé ω_G^{groupe} soit plus petit que le temps d'exécution séquentiel alors nous appliquons ce schéma récursivement afin de trouver du parallélisme à grain plus fin.

Le problème consistant à trouver les deux groupes de tâches A et B est central dans notre approche, et nous l'étudierons en détail par la suite. Nous l'appellerons le *problème du partitionnement de DAG*. Nous pouvons tout de suite remarquer que, sans perte de généralité, nous pouvons choisir $B = \widetilde{A}$. En effet les tâches du groupe de tâche B sont par définition incluses dans $\widetilde{A}$.

3.1. Schéma général de l'algorithme

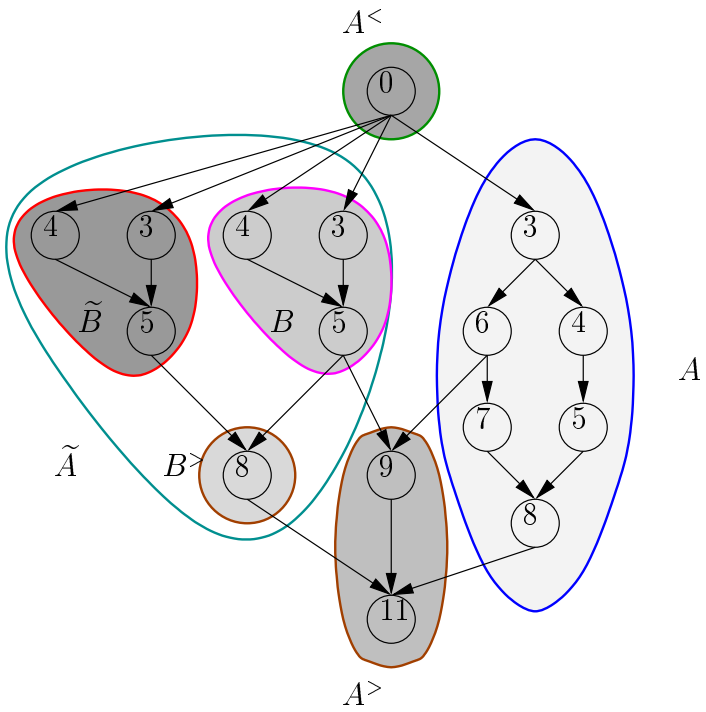

Figure 4 – Décomposition récursive d'un graphe composé de 16 tâches pour $\rho = 2$. Le graphe est d'abord décomposé en 4 groupes de tâches $A^<$, A, $\widetilde{A}$ et $A^>$. Ici, seul le graphe induit par le groupe de tâche $\widetilde{A}$ peut encore être décomposé. Les dates d'exécution sont inscrites sur les tâches. Le regroupement obtenu est composé des groupes de tâches $\{A^<, A, B, \widetilde{B}, B^>, A^>\}$. Le temps d'exécution est 12.

Nous considérons un groupe de tâches A solution du problème de partitionnement de DAG. Les quatre groupes de tâches convexes $A^<$, A, $\widetilde{A}$ et $A^>$ définissent une partition de notre graphe de précédence initial . Soit maintenant R un regroupement basé sur cette partition et ordonné selon un ordre arbitraire compatible avec l'ordre $\prec$. Le regroupement R permet aux groupes de tâches A et $\widetilde{A}$ de s'exécuter en parallèle. Par construction nous avons $\omega_R^{groupe} \leq |A^<| + \rho + max(|A|, |\widetilde{A}|) + \rho + |A^>|$, il n'y a pas égalité si l'un des groupes de tâches $A^<$ ou $A^>$ est vide. D'autre part, nous avons $max(|A|, |\widetilde{A}|) + min(|A|, |\widetilde{A}|) = |A| + |\widetilde{A}|$. Nous pouvons donc en déduire que $\omega_R^{groupe} \leq |V| + 2.\rho - min(|A|, |\widetilde{A}|)$. De plus d'après le lemme 4, $\omega_R < \omega_R^{groupe}$, un tel regroupement est donc intéressant si $min(|A|, |\widetilde{A}|)$ est supérieur à $2.\rho$. Ce résultat intuitivement naturel explique pourquoi nous considérons le problème de partitionement de DAG, en effet ce dernier par définition cherche à maximiser $min(|A|, |\widetilde{A}|)$.

Si le délai de communication est suffisamment petit par rapport à $|A|$ et $|\widetilde{A}|$, le temps d'exécution du regroupement R est plus petit que celui d'une exécution séquentielle. Dans ce cas, nous pouvons appliquer récursivement ce même algorithme sur les graphes induits par les quatre groupes de tâches $A^<$, A, $\widetilde{A}$ et $A^>$. Dans le cas contraire, nous exécutons séquentiellement les groupes de tâches $A^<$, A, $\widetilde{A}$ et $A^>$, selon un ordre total compatible avec $\prec$.

Algorithme 4 $\texttt{Regroupe}_\rho(G(V,E))$

$A = \texttt{Partition}(G)$;
si $\texttt{Min}(|A|, |\widetilde{A}|) \geq 2.\rho$ **alors**
  $R_A = \texttt{Regroupe}_\rho(\texttt{Induit}(G, A))$;
  $R_{\widetilde{A}} = \texttt{Regroupe}_\rho(\texttt{Induit}(G, \widetilde{A}))$;
  $R_{A^>} = \texttt{Regroupe}_\rho(\texttt{Induit}(G, A^>))$;
  $R_{A^<} = \texttt{Regroupe}_\rho(\texttt{Induit}(G, A^<))$;
  return $R_{A^<} \uplus R_A \uplus R_{\widetilde{A}} \uplus R_{A^>}$;
sinon
  return $\{(V, \overline{\prec})\}$;
fin si

La figure 4 illustre le principe de cette décomposition récursive. L'algorithme 4 le décrit. Dans ce dernier, la fonction $\texttt{Regroupe}_\rho(G)$ retourne un regroupement pour le graphe G. L'opérateur $\uplus$ désigne la fusion de plusieurs regroupements, il s'agit d'un opérateur d'union ensembliste mais qui préserve l'ordre sur les regroupements. La fonction $\texttt{Induit}(G, A)$ retourne le graphe induit par le groupe de tâche A. La fonction $\texttt{Partition}(G)$ retourne un groupe de tâches A du graphe G, elle cherche à résoudre le problème de partitionement de DAG. Enfin $\overline{\prec}$, désigne un ordre total (arbitrairement choisi) compatible avec l'ordre $\prec$.

3.2. Le problème de partitionement de DAG

Nous allons maintenant nous intéresser au problème qui consiste à trouver la découpe que nous effectuons à chaque appel récursif de l'algorithme. Cela correspond à la fonction `Partition`(G) de l'algorithme 4. Ce problème peut se formuler de la façon suivante :

Définition 8 [Problème de partitionnement de DAG]. Etant donné un graphe $G = (V, E)$, trouver un ensemble de tâches $A \subset V$, qui maximise $min(|A|, |\widetilde{A}|)$.

On peut noter que ce problème est relativement différent des problèmes classiques de partitionnement de graphes non orientés. Pour ces derniers, il s'agit souvent de partitionner l'ensemble des sommets d'un graphe non orienté en des groupes de sommets de taille égale (ou égale à un facteur près) de façon à minimiser le nombre d'arêtes entre les différents groupes. Un état de l'art des problèmes de partitionnement de graphe non orienté peut être trouvé dans l'article [Pot95]. Ce problème classique de partitionnement de graphe est un problème $\mathcal{NP}$-difficile. Nous ne connaissons pas actuellement la complexité de ce problème de partitionnement de DAG.

3.2.1. Une heuristique naïve

Nous allons tout d'abord considérer une première heuristique pour ce problème. Celle-ci consiste à construire deux groupes de tâches convexes indépendants A et B d'une manière symétrique et gloutonne en faisant «grandir» ces deux ensembles.

Nous partons de deux ensembles A et B réduits chacun à une tâche, tels que ces ensembles soient indépendants. Nous ajoutons un ensemble de tâches au plus petit de ces deux ensembles. Pour simplifier considérons que A est le plus petit de ces ensembles, nous ajoutons à A une tâche x indépendante du groupe de tâche B (ie $x \in \widetilde{B}$), puis nous effectuons une fermeture convexe sur l'ensemble $A \cup \{x\}$ afin de conserver un groupe de tâches convexe. L'idée consiste alors à choisir la tâche x de manière gloutonne, c'est-à-dire que nous choisissons la tâche x qui maximise $\overline{A \cup \{x\}}$. Puis nous recommençons cette itération jusqu'à ce qu'il ne soit plus possible d'ajouter de tâche dans le plus petit groupe de tâches.

Cependant une telle heuristique fait rapidement ressortir ses limites. En effet la complexité de celle-ci est très importante. Nous allons maintenant l'étudier. Nous désignons par $e = |E|$ le nombre d'arcs dans le graphe de précédence et par $v = |V|$ le nombre de tâches dans ce graphe. La complexité d'une opération de fermeture convexe est bornée par $\mathcal{O}(e + v)$. En effet, étant donné un groupe de tâches T, nous

pouvons utiliser un algorithme de marquage sur les tâches pour calculer les ensembles $T^<$, $T^>$ et par conséquence nous pouvons également calculer $\widetilde{T}$ et $\overline{T}$ en $\mathcal{O}(e)$. Nous pouvons aisément borner le nombre d'opérations de fermetures convexes par $\mathcal{O}(v^2)$. A chaque itération de l'algorithme nous effectuons moins de v fermetures convexes, et il y a moins de v itérations en tout. Si l'on considère les v^2 paires d'ensembles réduits à une tâche (cela correspond au choix initial des groupes de tâches A et B) alors nous pouvons borner la complexité de cette heuristique par $\mathcal{O}(e.v^4)$. Il serait sans nul doute possible d'obtenir une meilleure borne pour la complexité de cette heuristique. Cependant, lors des expérimentations nous n'avons pas réussi à utiliser cette heuristique pour des graphes de plus d'une centaine de tâches. Aussi, nous avons réfléchi à une heuristique de plus faible complexité.

3.2.2. Une heuristique aléatoire

L'objectif des algorithmes de regroupement est double. D'une part ceux-ci doivent permettre d'obtenir des regroupements dont le temps d'exécution est faible, c'est à dire que le regroupement obtenu doit être de bonne qualité. D'autre part, la complexité des algorithmes de regroupement doit être «relativement faible», car ils doivent pouvoir être appliqués à des graphes de plusieurs milliers voir plusieurs millions de tâches. C'est ce point que nous allons maintenant étudier. Il est particulièrement important notamment si l'on envisage d'utiliser un algorithme de regroupement à la volée. C'est à dire si l'algorithme de regroupement est utilisé au cours de l'exécution d'un programme parallèle pour déterminer où et dans quel ordre seront exécutées les différentes tâches. Dans le cadre de notre approche, il nous faut donc trouver une heuristique de partitionnement de DAG dont la complexité soit faible. D'autre part une telle heuristique permettrait d'observer le comportement d'un algorithme de regroupement par décomposition récursive pour des graphes de tâches de grande taille. L'heuristique que nous avons trouvée, est fondée sur le lemme suivant :

Lemme 9 Soit deux tâches indépendantes x et y ; les deux groupes de tâches $\{x\}^> \setminus \{y\}^>$ et $\{y\}^> \setminus \{x\}^>$ sont deux groupes de tâches indépendants.

Preuve. Nous raisonnons par l'absurde. Supposons donc qu'il existe deux tâches z_1 et z_2 appartenant respectivement à $\{x\}^> \setminus \{y\}^>$ et à $\{y\}^> \setminus \{x\}^>$ tels qu'il existe un chemin entre z_1 et z_2 dans le graphe de précédence G, c'est-à-dire tel que $z_1 \prec z_2$. Par définition de z_1 il existe un chemin entre x et z_1 c'est-à-dire $x \prec z_1$, donc $x \prec z_2$. Mais cela contredit la définition de z_2 car $z_2 \notin \{x\}^>$. Par symétrie, on en déduit immédiatement le résultat. La figure 5 permet de bien comprendre la situation. □

Nous pouvons remarquer que ce lemme admet une version «symétrique» dans laquelle nous considérons les tâches prédécesseurs, ainsi $\{x\}^< \setminus \{y\}^<$ et $\{y\}^< \setminus \{x\}^<$ sont deux groupes de tâches indépendants.

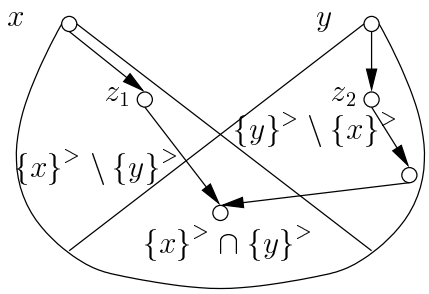

Figure 5 – Les différents groupes de tâches considérés

D'autre part la complexité du calcul de deux groupes de tâches indépendants est faible. En effet, nous avons déjà remarqué qu'étant donnée une tâche x, le calcul de $\widetilde{x}$, celui de $x^<$ et celui de $x^>$ pouvaient être réalisés efficacement en utilisant un algorithme de marquage avec une complexité en $\mathcal{O}(e+v)$. Nous rappelons que $e = |E|$ désigne le nombre d'arcs dans le graphe de précédence et que $v = |V|$ désigne le nombre de tâches dans ce graphe.

Il reste à déterminer les tâches x et y que nous sélectionnons : une première idée pourrait être de choisir parmi tous les couples de tâches (x, y) tels que $x \sim y$, celui qui maximise : $min(\{x\}^> \setminus \{y\}^>, \{y\}^> \setminus \{x\}^>)$. Cependant la complexité serait alors $\mathcal{O}(e.v^2)$ et serait donc encore une fois trop importante pour pouvoir être utilisée sur des graphes de grande taille.

C'est pourquoi, nous avons opté pour un algorithme aléatoire. Ainsi nous choisissons aléatoirement une tâche x. Si l'ensemble $\widetilde{\{x\}}$ n'est pas vide alors nous choisissons y aléatoirement dans ce groupe de tâche. Nous recommençons ce procédé K fois, en retournant le meilleur résultat. K désigne une constante pouvant être choisie expérimentalement. Intuitivement notre idée peut se justifier par le fait suivant. Définissons une tâche bloquante x comme une tâche tel que $\widetilde{\{x\}} = \emptyset$; si une majorité des tâches sont bloquantes alors notre algorithme ne permettra pas de trouver du parallélisme dans l'application. Cependant, le graphe de précédence est alors «proche» d'une chaîne, une structure de graphe non parallélisable.

La complexité de l'heuristique que nous avons décrite pour le problème de partitionnement de DAG est donc finalement en $\mathcal{O}(K.e)$. Si l'on suppose que cette heuristique partitionne le graphe en des ensembles de tailles voisines alors un algorithme de regroupement basé sur cette approche a une complexité en $\mathcal{O}(K.e.\log(v))$. Malheureusement l'heuristique de partitionnement que nous avons proposée est particulièrement inefficace lorsque le graphe de précédence est composé d'un grand nombre de composantes connexes. Aussi en pratique dans le cas d'un graphe non connexe, nous partitionnons les différentes composantes connexes en deux parties. Ce problème est très proche d'un problème de partition, pour lequel il existe des schémas

d'approximation très efficaces.

4. Simulations

Nous avons implémenté l'algorithme de décomposition récursive ainsi que l'heuristique décrite précédemment pour le problème de partitionnement de DAG. Nous avons ensuite simulé notre algorithme de regroupement sur des graphes d'applications régulières tout en faisant varier les délais de communication.

Pour l'heuristique de partitionnement de DAG nous avons fixé la constante K à 10. Nous avons choisi d'utiliser des graphes de précédence d'applications réelles. Ces graphes de précédence sont réguliers et donc faciles à générer. Les simulations ont étés conduites sur des graphes du type : Jacobi, FFT, Elimination de Gauss et Grille 3D. Ces graphes sont composés respectivement de 62500, 53248, 31623 et 64000 tâches. Ces graphes de précédence sont représentées dans les figures 6,7, 8 et 9.

Nous avons comparé notre algorithme de regroupement à l'algorithme de regroupement classique DSC [GY94a]. Les figures 10, 11, 12, 13 représentent les résultats de la simulation. L'abscisse représente le logarithme en base 2 du délai de communication et l'ordonnée le logarithme en base 2 de l'accélération. Cette dernière est définie comme le rapport entre le temps d'exécution séquentielle c'est-à-dire ici le nombre de tâches et le temps d'exécution du regroupement R obtenu ω_R. Ainsi la droite des figures correspond au cas où les délais de communication sont très importants, et sont aussi important que le temps d'exécution de l'application en séquentielle, dans ce cas il n'y a plus aucun parallélisme exploitable. La gauche des figures correspond à un petit délai de communication $\rho = 1$.

Pour le cas du graphe Jacobi (figure 10), nous avons représenté la moyenne, le moins bon et le meilleur des temps d'exécution des regroupements obtenus pour 10 expériences, pour les autres figures il s'agit toujours d'une moyenne sur 10 expériences. Nous avons représenté ces trois courbes car nous voulions observer l'influence du caractère aléatoire de l'heuristique de partitionnement de DAG. Les trois courbes dans la figure sont proches bien que le facteur K soit seulement égal à 10, le comportement de l'algorithme est donc relativement stable (il faut bien garder en tête qu'il s'agit d'une échelle logarithmique et qu'une petite différence est importante en valeur absolue). Pour les graphes de Jacobi et Grille 3D (figure 13), on peut remarquer que DSC est légèrement meilleur pour de petits délais de communication $\rho < 16$, alors que lorsque le délai de communication est plus grand, notre algorithme se comporte beaucoup mieux. Le temps d'exécution d'un regroupement calculé par l'algorithme DSC est alors jusque 8 fois plus grand. De plus nous avons remarqué que ce facteur

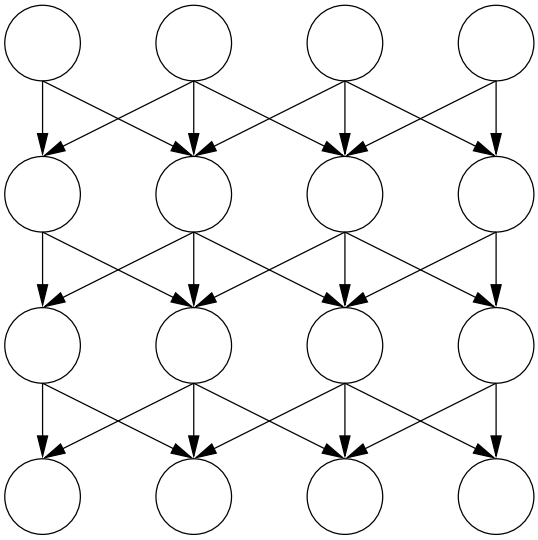

Figure 6 – Graphe de type Jacobi.

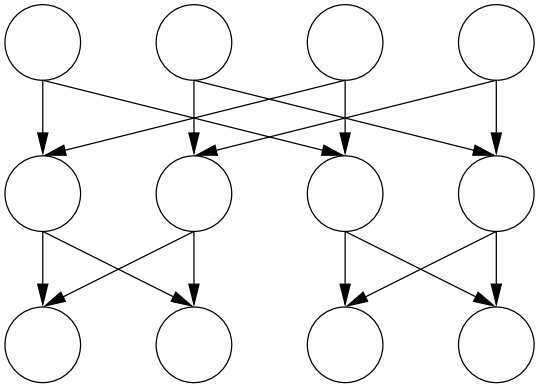

Figure 7 – Graphe de type FFT.

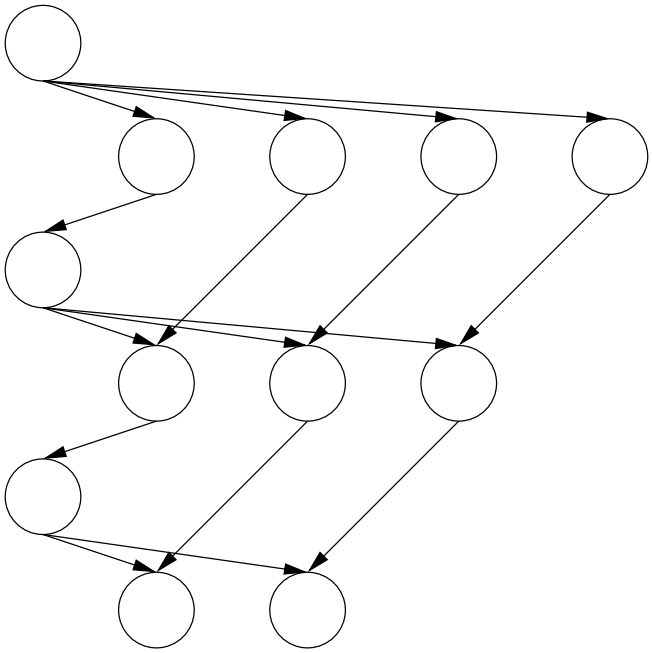

Figure 8 – Graphe de type Elimination de Gauss.

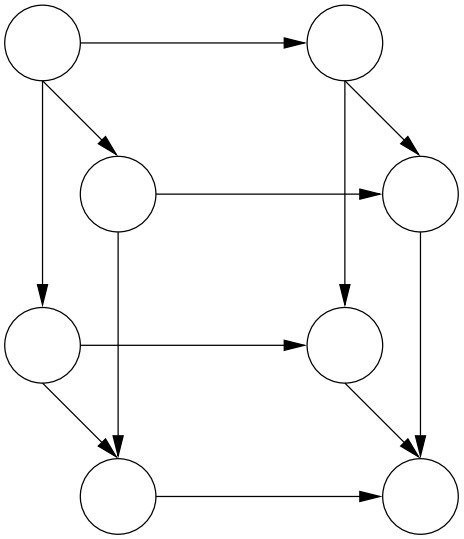

Figure 9 – Graphe de type grille 3D

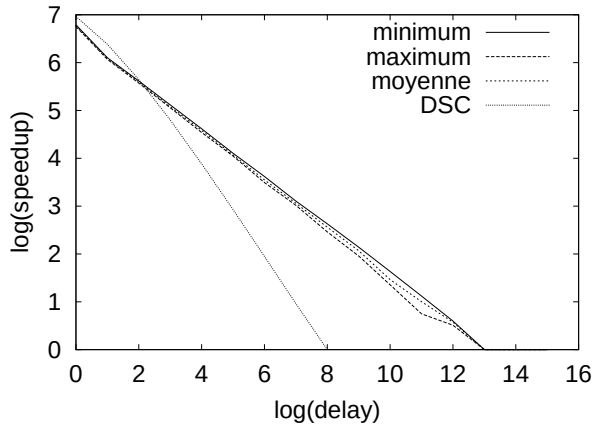

Figure 10 – Résultat pour un graphe jacobi composé de 62500 tâches.

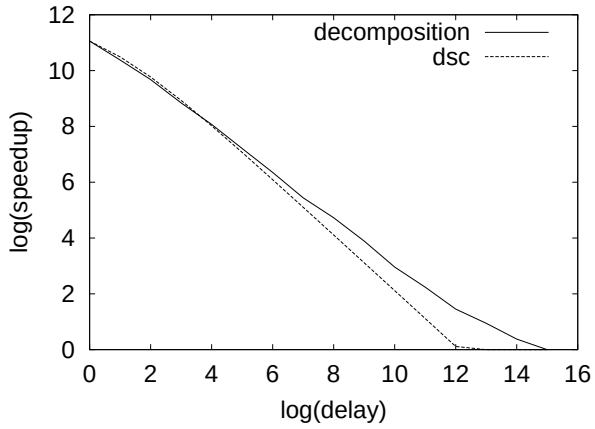

Figure 11 – Résultat pour un graphe FFT composé de 53248 tâches.

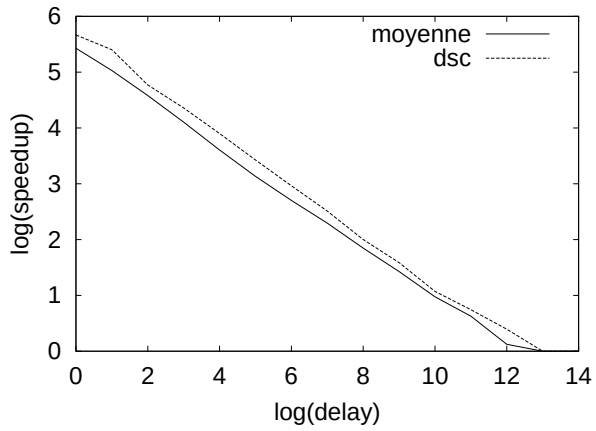

Figure 12 – Résultat pour un graphe Elimination de Gauss composé de 31623 tâches.

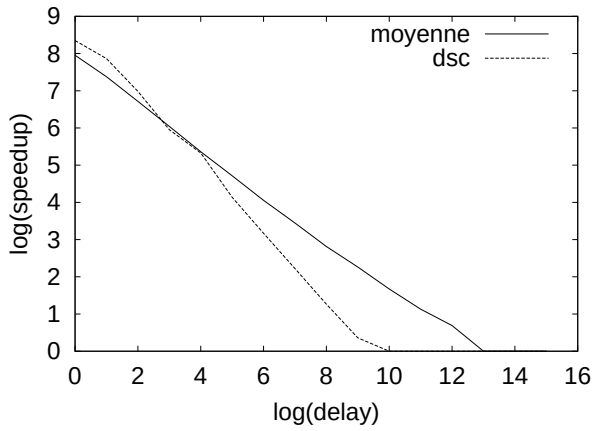

Figure 13 – Résultat pour un graphe grille 3D composé de 64000 tâches.

est encore plus important lorsque le graphe de précédence est plus grand. Par contre pour le graphe d'élimination de Gauss (figure 12) l'algorithme DSC est toujours légèrement meilleur quel que soit le délai de communication. Ce bon comportement de l'algorithme de DSC sur les graphes d'élimination de Gauss et de Gauss-Jordan est connu et a été analysé dans [GY94b]. Enfin pour le graphe de type FFT (figure 11), l'application est très parallèle (la longueur du chemin critique est en $\mathcal{O}(\log(n))$ et le nombre de tâches est de l'ordre de $\mathcal{O}(n.\log(n))$), notre algorithme ne se comporte mieux que DSC que lorsque le délai de communication devient très important.

Le nombre de regroupements que nous avons dû généré pour cette simulation est de 40. Nous avons utilisé le fait que seul le seuil d'arrêt de la découpe récursive dépend du délai de communication. Il suffit donc d'effectuer une décomposition récursive du graphe de précédence jusqu'au grain le plus fin ; puis de parcourir la décomposition générée jusqu'à une certaine profondeur pour construire les différents regroupements lorsque l'on fait varier le délai de communication. D'autre part, il y a 4 graphes différents et chaque courbe est une moyenne de 10 expérimentations. Le temps de calcul de ces 40 regroupements a nécessité moins d'une demi-heure sur un pentium3 à 533 Mhz, soit moins d'une minute pour calculer un regroupement. Ce temps de calcul est nettement supérieur à celui de l'algorithme DSC. Cependant il serait tout à fait possible d'aller plus vite car le prototype a été écrit afin de pouvoir s'adapter très facilement à d'autres algorithmes de partitionnement de DAG. D'autre part, nous n'avons pas pu comparer notre algorithme avec DSC pour des graphes de plus grande taille car la version disponible sur le web ne permettait pas de traiter des graphes de précédence de plus de 10^5 tâches. Enfin nous avons remarqué que le facteur limitatif de notre heuristique de regroupement était principalement lié à la quantité de mémoire disponible. Le processus commence à utiliser le fichier d'échange [1] vers 10^6 tâches (sur 128 Mo de mémoire de vive). Le temps de la première itération de partitionnement de DAG devient alors rapidement très long. Cela s'explique par les nombreux parcours itératifs d'une grande partie de la structure du graphe qu'il faut alors charger à partir du disque dur, lors de l'utilisation de l'algorithme de marquage. Dans ce contexte, il pourrait être intéressant d'utiliser des représentations paramétriques et donc peu gourmande en mémoire, pour représenter le graphe de précédence [CJ99].

5. Travaux en cours et perspectives

Les premiers résultats obtenus lors des simulations sont très encourageants et ce surtout lorsque les temps de communication sont importants. L'approche par décom-

[1] «swap» en anglais

position récursive semble ouvrir de très nombreuses perspectives pour les problèmes d'ordonnancement sous des modèles avec des communications coûteuses en temps. Il reste cependant de nombreux points à éclaircir et d'autres points peuvent sans aucun doute être améliorés. Aussi, je continue à travailler activement sur cette approche. Dans ce dernier paragraphe, je vais essayer de faire un tour d'horizon de différentes perspectives ce travail, et je vais présenter les différents travaux qui sont en cours actuellement.

5.1. Quelle garantie espérer pour une telle approche ?

L'une des premières questions que l'on peut de se poser est la garantie de performance que peut offrir une telle approche. Comme nous l'avons vu dans le chapitre 2, la garantie des algorithmes «classiques» pour le regroupement dépend très souvent linéairement du délai de communication ρ. En supposant que l'on sache résoudre le problème de partitionnement de DAG (ou tout au moins que l'on dispose d'un algorithme d'approximation pour ce problème), on peut se poser la question de la garantie de performance que l'on peut obtenir pour le problème du regroupement. Ce travail semble difficile. Cependant des avancées récentes ont eu lieu dans le domaine de l'analyse des garanties de performances des algorithmes basées sur le paradigme diviser pour régner [2]. Le livre [Hoc96] et plus particulièrement le chapitre 5 de Shmoys en présente un résumé très instructif.

Nous nous sommes cependant intéressés aux cas qui semblent poser problème. Le pire cas que nous connaissons actuellement est celui de l'arbre binaire complet. Considérons un arbre binaire complet de hauteur $2.\log\rho$, il est composé de $\rho^2 - 1$ tâches. Et il est possible de l'ordonnancer en $3.\rho - 2$ unités de temps, il suffit d'ordonnancer les tâches de hauteur plus petite que $\log\rho$ ensemble. Si l'on considère un algorithme de partitionnement de DAG optimal celui-ci coupe à chaque étape simplement l'arbre en 2 et répète ce schéma. La durée d'exécution du regroupement obtenu est donc de $(\rho+1).log\rho + (\rho-1)$. La meilleur garantie que l'on peut espérer obtenir lorsque ρ est grand est donc de l'ordre de $\mathcal{O}(\log\rho)$. La figure 14 illustre les deux types de regroupements que l'on a ainsi construit.

5.2. Etude du problème de partitionnement de DAG

Le problème de partitionnement de DAG est le problème clé de notre approche, aussi une étude approfondie de celui-ci apparaît nécessaire. Nous travaillons actuellement en collaboration avec András Sebõ sur la complexité de ce problème. Nous

[2] «Divide and Conquer» en anglais

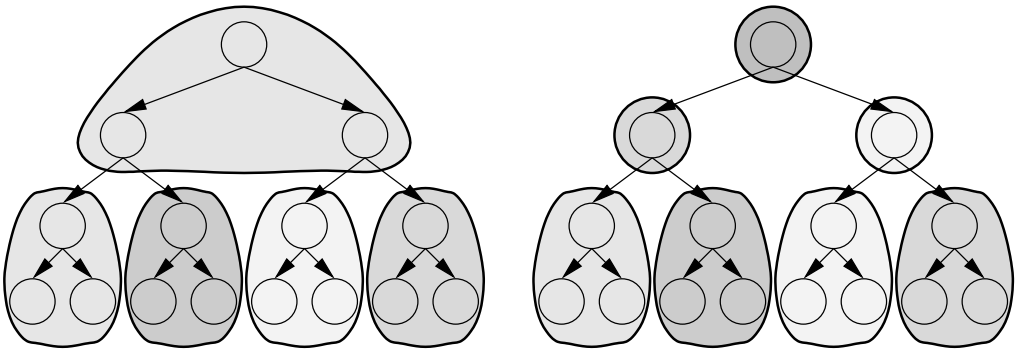

Figure 14 – A gauche, un exemple de «bon» regroupement pour $\rho = 2$ et à droite le regroupement obtenu par notre approche.

pensons que le problème est $\mathcal{NP}$-difficile. Il est équivalent au problème de sous graphe équilibré biparti complet [GJ79] [3] dans un graphe de co-comparabilité (ie. complémentaire de la fermeture transitive d'un graphe de précédence). Ce dernier problème est un problème $\mathcal{NP}$-complet dans le cas d'un graphe quelconque. D'autre part le problème apparaît également comme très proche de problèmes tel que le «sommet séparateur minimum» [4] [CK94].

Nous pensons que ce problème est $\mathcal{NP}$-difficile. Dans un tel cas, il serait également intéressant d'étudier s'il est possible de trouver des algorithmes d'approximation pour ce problème. D'autre part une étude d'heurisitiques efficaces tant du point de vue de la qualité du partitionnement obtenu que du point de vue de leur complexité pourrait être envisagée afin de compléter le travail entrepris.

5.3. Vers d'autres modèles de communication.

Un autre point qui mérite d'être abordé est l'indépendance de l'algorithme de partitionnement de DAG et plus généralement de l'approche envisagée vis à vis de la modélisation des communications. La décomposition récursive du graphe de précédence ne dépend pas par exemple du délai de communication que nous avons choisi. Seul le seuil d'arrêt de la décomposition dépend de ce délai. Ainsi, il apparaît possible d'étendre notre approche pour des modèles de communication plus fins tels que ceux prenant un compte un sur-coût à l'envoi des messages comme par exemple LogP. De plus, on peut alors aussi imaginer regrouper les communications entre les groupes de tâches qui communiquent si cela s'avère intéressant dans le modèle envisagé.

[3] «balanced complete bipartite subgraph» en anglais
[4] «Minimum B-vertex separator» en anglais

D'autre part, la vision structurée de l'application parallèle offerte par notre approche semble bien se marier avec les nouvelles architectures parallèles, tels que les grappes de PC ou bien même des réseaux de grappes connectées entre elles par des réseaux haut débits comme VTHD [VTH99]. Ces nouvelles architectures sont caractérisées notamment par des communications hiérarchiques. Les communications entre des machines situées sur deux grappes différentes doivent par exemple traverser un réseau national ; elles sont beaucoup plus coûteuses que les communications à l'intérieur d'une même grappe. On peut dans ce cas imaginer de s'arrêter à un certain seuil de décomposition afin d'exploiter un parallélisme à gros grain au niveau inter-grappes par exemple, et de décomposer plus finement l'application afin d'exploiter un parallélisme à grain plus fin au niveau grappe ou bien même au niveau SMP. A notre connaissance, il n'existe pas encore d'heurisitiques pour ces nouveaux problèmes d'ordonnancement qui émergent. Une étude théorique préliminaire sur la complexité des problèmes d'ordonnancement sous des modèles de type petit délai et à deux niveaux de hiérarchie a cependant été menée dans sa thèse par Giroudeau [Gir00].

5.4. Intégration dans Athapascan1

D'un point de vue pratique, et au vu des résultats encourageants des simulations, nous souhaitons maintenant valider notre approche sur des applications réelles. En collaboration avec Rémi Revire, un travail dans cette voie est en cours actuellement. Il s'agit d'intégrer notre approche et d'autres heuristiques d'ordonnancement statique dans Athapascan1.

Athapascan1 est une librairie parallèle simple d'utilisation développée au sein du projet Apache. En substance, Athapascan1 propose une interface de programmation parallèle de haut niveau indépendante de l'architecture. La granularité est explicite (tâche et objet partagé), mais le parallélisme est implicite les relations de précédence entre les différentes tâches sont déduites des accès (lecture, écriture) effectués par les tâches sur les objets partagés. Cette mécanique permet de construire dynamiquement tout ou partie du graphe de flot de données associé à l'application. Le graphe de flot de données est un outil très riche à partir duquel on peut notamment construire le graphe de précédence.

Dans le cadre de notre collaboration ; nous avons réalisé une interface générique de parcours de ces deux types de graphes qui sont les entrées d'un algorithme d'ordonnancement statique. D'autre part, nous avons défini une interface objet pour des ordonnanceurs statiques. Plusieurs algorithmes d'ordonnancement ont été implantés ou adaptés au dessus de cette interface. Nous intégrons actuellement ETF, DSC ainsi que notre heuristique en ayant effectué un repliement sur un nombre borné de processeurs. Il serait également possible d'utiliser le regroupement obtenu comme base pour

un algorithme de régulation de charge basé sur le vol de travail. Celui-ci pourrait alors voler des groupes de tâches entier. Un tel mécanisme pourrait par exemple permettre de réguler de faibles hétérogénéités de vitesse, ou d'amortir les imprécisions sur les temps de calcul des tâches (considérés ici comme tous identiques).

Notre objectif est maintenant de valider et de comparer ces différents ordonnanceurs sur quelques applications telles que des applications de programmation dynamique (combinaison par blocs, comparaison de deux mots), ainsi que sur des applications de programmation numérique (Elimination de Gauss, factorisation de Cholesky creux).

CHAPITRE

Ordonnancement avec duplication

5

Dans ce chapitre nous nous intéressons au problème de l'ordonnancement avec duplication et avec un grand délai de communication ρ sur un nombre borné de processeurs. Nous introduisons tout d'abord une borne inférieure de la date d'exécution d'une tâche. Puis nous montrons qu'il est possible de réduire le problème initial à celui de l'ordonnancement d'une succession de «petits graphes». Enfin, nous montrons comment une telle approche permet de construire un algorithme avec une garantie asymptotique en $\mathcal{O}(\log(\rho))$ pour ordonnancer une application décrite par un graphe de précédence arbitraire.

Sommaire

1. Introduction

Dans cette partie nous nous intéressons au problème de l'ordonnancement avec *duplication* et avec un grand *délai de communication*. Le modèle que nous considérons ici est similaire à celui introduit par Papadimitriou et Yannakakis [PY90]. De façon classique, l'application parallèle est décrite par un graphe de précédence. Toutes les tâches ont une même durée, que nous choisissons égale à 1. D'autre part, les communications sont modélisées par un délai uniforme ρ. Cela signifie que si deux tâches x et y liées par une relation de précédence sont ordonnancées sur des processeurs différents alors la tâche y ne pourra commencer à s'exécuter que ρ unités de temps après la fin de la tâche x. Par contre si x et y sont placées sur le même processeur alors la tâche y pourra s'exécuter immédiatement après la terminaison de la tâche x. Sous un modèle de ce type, il peut être intéressant de dupliquer certaines tâches. En effet, si une tâche a plusieurs successeurs, il peut être utile d'exécuter cette tâche sur différents processeurs afin de pouvoir commencer à exécuter des successeurs plus tôt. La figure 1 illustre très brièvement le modèle et l'intérêt de la duplication.

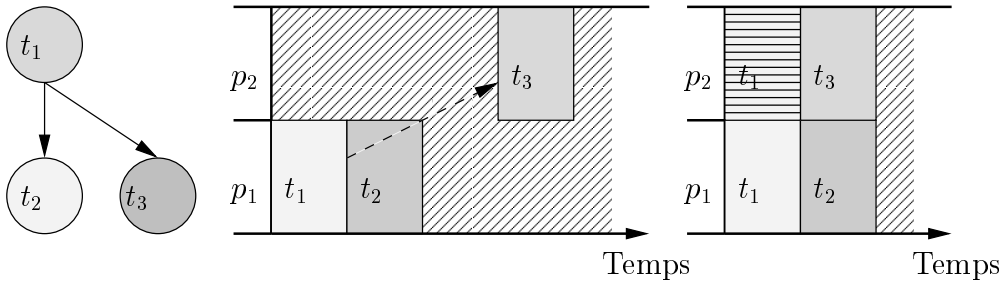

Figure 1 – Un graphe de précédence et deux diagrammes de Gantt représentant des ordonnancements avec et sans duplication. La tâche t_1 hachurée horizontalement est un duplicata.

La prise en compte de la duplication a permis d'obtenir des garanties de performance indépendantes du délai de communication ρ. Ainsi, sous ce modèle, Papadimitriou et Yannakakis ont proposé un *algorithme d'approximation* avec une garantie de 2 pour le problème de l'ordonnancement sur un nombre non borné de processeurs. Ce résultat est l'un des rares résultats d'algorithmes d'approximation constante pour les problèmes d'ordonnancement avec grand délai de communication. Cependant, il n'est pas utilisable en pratique : le «travail» qu'il engendre, de par la duplication (c.-à-d. le nombre de tâches réellement exécutées), n'est pas borné relativement à la taille du graphe. Il n'est donc pas possible de «replier» efficacement le regroupement obtenu sur un nombre borné de processeurs. Une description plus complète des

modèles à communications explicites ainsi que les principaux résultats obtenus pour l'ordonnancement sous ces modèles a été présentée dans le chapitre 2.

Dans ce chapitre, nous nous intéressons à ce même problème sur un nombre borné m de processeurs. Pour ce problème, nous proposons un algorithme d'approximation avec une garantie asymptotique en $\mathcal{O}(\log(\rho))$. Intuitivement l'idée consiste à ordonnancer une succession de «petits graphes». Cette notion de «petits graphes» est étroitement liée à la date d'exécution au plus tôt et sur un nombre non borné de processeurs des différentes tâches constituant ce graphe . Dans le premier paragraphe nous introduisons une borne inférieure de la date d'exécution d'une tâche, borne qui nous permet de définir formellement ce que nous entendons par «petits graphes». Dans le paragraphe 3.1, nous présentons plus formellement l'algorithme. Puis, nous analysons celui-ci dans le paragraphe 3.2, en supposant connu un algorithme pour l'ordonnancement de «petits graphes». Enfin dans le paragraphe 4, nous montrerons comment ordonnancer de «petits graphes». Ce travail a été réalisé en collaboration avec Christophe Rapine.

2. Une borne inférieure de la date d'exécution d'une tâche

Déterminer pour une tâche x sa date d'exécution au plus tôt sur un nombre non borné de processeurs est un problème NP-difficile car le *problème du regroupement* l'est également. C'est pourquoi nous introduisons maintenant la *hauteur* d'une tâche x dans un graphe G. Nous montrons par la suite que cette hauteur est une borne inférieure de la date d'exécution de la tâche x sur un nombre non borné de processeurs.

Définition 1 [Hauteur d'une tâche]. La hauteur d'une tâche x dans un graphe G notée $h_G(x)$ est définie de façon inductive par :

- Si x est une racine du graphe G alors $h_G(x) = 0$.
- Sinon, soit $A(x) = \{x_1, \ldots, x_{|A(x)|}\}$ les ancêtres d'une tâche x ordonnés par hauteur décroissante, et soit p la plus petite valeur entre $|A(x)|$ et $\rho + 1$; alors nous posons :

$$h_G(x) = \max_{i \in \{1, \cdots, p\}} h_G(x_i) + i$$

Lorsqu'il n'y a pas d'ambiguïté sur le graphe G considéré, nous notons la hauteur d'une tâche x, $h(x)$. Nous pouvons remarquer tout de suite que la hauteur d'une tâche est une généralisation directe de la notion de chemin critique dans un graphe. En effet, la longueur du chemin critique d'une tâche x dans un graphe G noté, $\omega_G^\infty(x)$ peut se définir inductivement d'une façon similaire. Dans le cas où x n'est pas une racine, et

si les ancêtres de x sont triés par longueur de chemin critique décroissante alors la longueur du chemin critique vaut par définition $\omega_G^\infty(x) = \omega_G^\infty(x_1) + 1$. De même la hauteur d'une tâche est une généralisation de la borne introduite par Papadimitriou et Yannakakis [PY90], notée $e(x)$. Si x n'est pas une racine du graphe G alors $e(x) = e(x_p) + p$ où les ancêtres de x sont classés par $e(.)$ décroissant et où p est défini comme dans la définition 1.

Nous allons maintenant montrer que la hauteur d'une tâche x est une borne inférieure de sa date d'exécution au plus tôt.

Lemme 2 Une tâche x ne peut pas être ordonnancée avant la date $h(x)$.

Preuve. La preuve se fait par induction sur la structure du graphe. Soit x une tâche. Si x est une racine alors le résultat est immédiat. Dans le cas contraire, on se donne un ordonnancement réalisable. Supposons que le résultat soit vrai pour tous les ancêtres de la tâche x. Nous notons ces ancêtres $A(x) = \{x_1, \ldots, x_{|A(x)|}\}$ et nous supposons que ceux-ci sont triés par hauteur décroissante. Nous allons montrer que le résultat est également vrai pour la tâche x.

Considérons une tâche x_i avec i compris entre 1 et $p = \min(|A(x)|, \rho + 1)$. Pour toutes les tâches x_j avec $j \leq i$, nous avons $h(x_j) \geq h(x_i)$. Or si la tâche x est ordonnancée strictement avant $h(x_i) + 1 + \rho$, alors toutes les tâches x_j avec $j \leq i$ sont ordonnancées sur le même processeur que la tâche x (ces tâches peuvent avoir déjà été exécutées sur d'autres processeurs et être alors dupliquées). Comme cet ensemble est composé de i tâches, la tâche x ne peut alors pas être ordonnancée avant $h(x_i) + i$. Par définition $i \leq \rho + 1$, par l'absurde on démontre donc que la tâche x ne peut pas être ordonnancée avant la date $h(x_i) + i$. Finalement nous pouvons conclure que la tâche x ne peut être ordonnancée avant la date $h(x)$. □

Définition 3 [Hauteur d'un graphe]. La hauteur d'un graphe G notée $h(G)$ est définie par :

$$h(G) = \max_{x \in G} h_G(x) + 1$$

D'après le lemme 2, il apparaît de façon évidente que $h(G)$ est une borne inférieure de la durée d'un ordonnancement du graphe de tâches G sur un nombre non borné de processeurs et en dupliquant éventuellement certaines tâches. La notion de «petit graphe» correspond à un graphe dont la hauteur $h(G)$ est plus petite que $\rho + 1$. Intuitivement cela correspond aux graphes tels que toutes les tâches ont moins que ρ prédécesseurs (ce résultat découle de la définition de la hauteur d'une tâche). Un petit graphe est donc un graphe qui peut être ordonnancé sur un nombre non borné de processeurs et en dupliquant certaines tâches, en moins de $\rho + 1$ unités de temps. Il suffit alors d'exécuter les différentes «feuilles» du graphe (tâches sans successeur) sur des processeurs distincts. Sur chaque processeur, il faut alors exécuter tous les

prédécesseurs d'une tâche feuille. Ces prédécesseurs peuvent être dupliqués un grand nombre de fois.

3. Vers l'ordonnancement de graphes quelconques

Dans ce paragraphe, nous nous intéressons à un algorithme pour l'ordonnancement des graphes de hauteur quelconque. Il s'agit d'un algorithme par phases qui consiste simplement à ordonnancer successivement des «petits graphes». Nous allons notamment étudier la garantie que peut offrir un tel algorithme en fonction de l'algorithme d'ordonnancement pour les «petits graphes».

3.1. Un algorithme par phases

Etant donné un graphe $G = (V, E)$, nous notons par la suite G_ρ le sous-graphe induit par l'ensemble des tâches dont la hauteur est plus petite que ρ : $\{x \in V \mid h_G(x) \leq \rho\}$. Par définition, nous avons $h(G_\rho) \leq \rho+1$, G_ρ est donc un «petit graphe». Pour ordonnancer un graphe quelconque, l'algorithme que nous proposons consiste simplement à ordonnancer le sous-graphe G_ρ, à attendre une durée de ρ unités de temps pour permettre aux communications d'avoir lieu, puis à recommencer avec le graphe restant. L'algorithme 5 décrit cette démarche plus formellement. Il prend en entrée un graphe G, un algorithme d'ordonnancement pour des «petits graphes» $\mathcal{A}$, et un délai de communication ρ. La fonction `Induit`(G, T) renvoie le sous-graphe induit par l'ensemble de tâches T du graphe G, et la fonction `Taches`(G) retourne l'ensemble des tâches du graphe G.

Algorithme 5 `Ordonnanceur_grand`$(G, \mathcal{A}, \rho)$

tant que `Taches`$(G) \neq \emptyset$ **faire**
  $T_\rho = \{x \in G \mid h_G(x) \leq \rho\}$;
  $G_\rho =$ `Induit`(G, T_ρ);
  Ordonnancer G_ρ avec l'algorithme $\mathcal{A}$;
  Attendre ρ unités de temps ;
  $G =$ `Induit`$(G,$ `Taches`$(G) \setminus T_\rho)$;
fin tant que

3.2. Analyse de l'algorithme

Nous allons maintenant étudier les garanties de performances que peut offrir un tel algorithme en fonction de celles offertes par l'algorithme d'ordonnancement pour des petits graphes $\mathcal{A}$. Pour cela nous allons d'abord étudier le rapport entre les hauteurs $h(G)$ et $h(G')$, où $h(G)$ désigne la hauteur du graphe au début d'une itération et G' désigne le graphe restant après cette itération, en notant $G' = G \setminus G_\rho$. Le lemme suivant est le point clé de l'analyse. En effet il va permettre de borner le nombre d'itérations de l'algorithme.

Lemme 4 Si la graphe restant G' est non vide alors la hauteur du graphe décroît au moins de $\rho + 1$ unités :

$$h(G) - h(G') \geq \rho + 1$$

Preuve. Cette preuve est un peu technique. Dans cette preuve, pour alléger les notations, nous noterons $h'(.)$ la hauteur dans le graphe G' c'est-à-dire $h_{G'}(.)$. Nous allons alors montrer que pour toute tâche x dans G', $h(x) - h'(x) \geq \rho + 1$. Le lemme est une conséquence immédiate de ce résultat. Soit donc une tâche x dans le graphe G'.

La preuve se fait par induction sur la structure du graphe G'. Dans le cas où x est une racine dans G' le résultat est évident. Considérons une tâche x dans le graphe G' qui ne soit pas une racine. Soit $A'(x) = \{x'_1, \cdots, x'_{|A'(x)|}\}$ les ancêtres de la tâche x dans le graphe G' classés par hauteur décroissante dans ce graphe.

Nous pouvons remarquer que les ensembles $A'(x)$ et $A(x) \setminus \{y \mid h(y) \leq \rho\}$ sont égaux. Soit $\{x_1, \cdots, x_{|A'(x)|}\}$, les éléments de ce dernier ensemble classés par hauteur décroissante dans le graphe initial G. Les $(x'_i)_i$ et les $(x_i)_i$ ne sont pas nécessairement classés selon le même ordre. Plus précisément il existe une permutation σ sur l'ensemble $\{1, \cdots, |A'(x)|\}$ tel que $x_i = x'_{\sigma(i)}$. Nous notons σ^{-1} la permutation inverse.

Nous allons commencer par montrer un résultat intermédiaire : pour tout i dans $\{1, \cdots, |A'(x)|\}$ nous avons $h(x_i) - h'(x'_i) \geq \rho + 1$. En effet, soit i dans $\{1, \cdots, |A'(x)|\}$, nous pouvons distinguer deux cas :

- Si $i \geq \sigma(i)$ alors $h'(x_i) = h'(x'_{\sigma(i)}) \geq h'(x'_i)$. D'après notre hypothèse d'induction, nous avons aussi $h(x_i) - h'(x_i) \geq \rho + 1$. En sommant ces deux inégalités nous obtenons donc : $h(x_i) - h'(x'_i) \geq \rho + 1$.
- Si $i < \sigma(i)$ alors de par un argument de flux, il existe $j > i$, tel que $\sigma(j) \leq i$. La figure 2 permet de bien comprendre pourquoi. Les flèches en trait plein illustrent les trois équations que nous allons maintenant exhiber. Comme $x'_{\sigma(j)} = x_j$, nous avons $h'(x_j) \geq h'(x'_i)$ et $h(x_i) \geq h(x_j)$. D'après notre hypothèse d'induction $h(x_j) - h'(x_j) \geq \rho + 1$ et finalement en sommant ces trois dernières

équations on obtient également : $h(x_i) - h'(x'_i) \geq \rho + 1$.

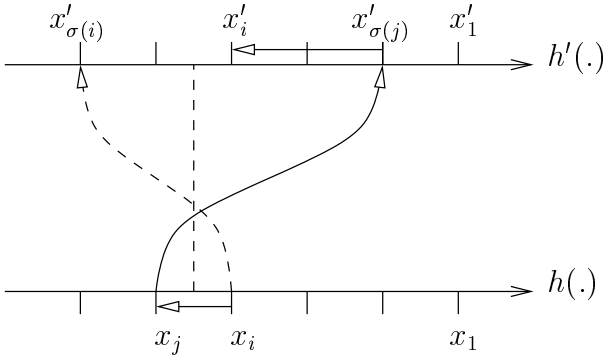

Figure 2 – Autant de flèches traversent le mur en pointillé dans un sens que dans l'autre.

Considérons maintenant l'entier j compris entre 1 et $\min(\rho+1, |A'(x)|)$ qui maximise $h'(x'_j)+j$; par définition de $h'(x)$, nous avons : $h'(x) = h'(x'_j)+j$. Donc d'après le résultat précédent nous avons : $(h(x_j) + j) - h'(x) \geq \rho + 1$. En considérant la définition de $h(x)$ nous pouvons aisément conclure que $h(x) - h'(x) \geq (\rho + 1)$. □

Dans la suite, par souci de simplicité, pour tout graphe G nous désignons par $|G|$ le nombre de tâches du graphe G. D'autre part, nous désignons en général par G des graphes de hauteur quelconque, et par H des «petits» graphes.

Corollaire 5 Supposons que l'algorithme $\mathcal{A}$ retourne pour tout graphe H de hauteur plus petite que $\rho + 1$ un ordonnancement sur m processeurs de durée inférieure ou égale à $\alpha.\frac{|H|}{m}+\beta.(\rho+1)$. Alors pour tout graphe G de hauteur quelconque, l'algorithme proposé retourne un ordonnancement sur m processeurs de durée inférieure ou égale à $\alpha.\frac{|G|}{m} + (\beta + 1).h(G) + \beta.(\rho + 1)$.

Preuve. La preuve est très simple. En effet, supposons que l'algorithme 5 termine en k itérations et soit $H_1, \cdots H_k$ les graphes ordonnancés au cours des différentes itérations. Alors comme $\sum_{i=1,\cdots,k} \frac{|H_i|}{m} = \frac{|G|}{m}$, on en déduit que la durée totale de l'ordonnancement est inférieure ou égale à $\alpha.\frac{|G|}{m}+k.\beta.(\rho+1)+(k-1).\rho$. Or le lemme précédent permet de borner le nombre d'itérations de l'algorithme. En effet d'après celui-ci, à chaque itération la hauteur du graphe restant décroît de $\rho + 1$, le nombre d'itérations k de l'algorithme est donc inférieure ou égal à $\lceil\frac{h(G)}{\rho+1}\rceil$. La durée totale de l'ordonnancement est donc finalement inférieure à : $\alpha.\frac{|G|}{m} + h(G).(\beta + 1) + \beta.(\rho + 1)$. □

Ce résultat est très intéressant. Il montre que si l'on sait ordonnancer un «petit graphe» efficacement (au sens défini par le lemme), alors on peut ordonnancer efficacement un graphe de hauteur quelconque.

4. Ordonnancement de graphes de petites tailles

Il apparaît donc clairement que le problème de l'ordonnancement d'un graphe H de hauteur inférieure ou égale à $\rho+1$ est le problème clé pour obtenir des algorithmes avec une bonne garantie de performance. Dans la suite nous considérons le problème de l'ordonnancement d'un graphe H de hauteur inférieure ou égale à $\rho+1$ et de structure quelconque. Pour toute tâche x de ce graphe $h(x) \leq \rho$, par définition nous avons donc $|A(x)| \leq \rho$. Le graphe H peut donc être ordonnancé sur un nombre non borné de processeurs en moins de $\rho+1$ unités de temps de façon immédiate. Le problème auquel nous nous intéressons maintenant consiste à ordonnancer ce graphe sur un nombre borné de processeurs.

4.1. Un autre algorithme par phase

Pour cela, nous proposons à nouveau d'utiliser un algorithme par phase. Chaque phase de l'algorithme est constituée d'une phase de travail de durée $\rho+1$, et d'une phase de communication de durée $\rho-1$ Dans la phase de travail, nous sélectionnons un ensemble S de tâches «feuilles», c'est-à-dire de tâches sans successeur. Puis nous ordonnançons chacune de ces tâches avec tous leurs prédécesseurs pendant cette phase. Dans la suite, étant donné un ensemble de tâches X, nous noterons $A(X)$ les ancêtres de X au sens large, c'est-à-dire $A(X) = \cup_{x \in X} A(x) \cup X$, et $W(X)$ le cardinal de cet ensemble $W(X) = |A(X)|$ (le choix de cette notation est lié au fait qu'il s'agit aussi d'un travail car chaque tâche a une durée unitaire). Par construction $W(S)$ désigne le nombre de tâches originales éxécutées pendant une phase de travail. Nous détaillons maintenant plus formellement l'algorithme. La fonction `Taches`(H) retourne l'ensemble des tâches du graphe H, la fonction `Induit`(H,T) permet de construire un sous-graphe induit. La fonction `Selection`(H,n) sélectionne un ensemble de n tâches feuilles dans le graphe H, et la fonction `Ordonnance`(H,S,t) ordonnance l'ensemble des tâches S ainsi que leur prédécesseurs dans le graphe H à partir de la date t en $\rho+1$ unitées de temps. Nous détaillerons ces deux dernières fonctions dans les deux sections suivantes 4.2 et 4.3.

Au cours de la première phase, m tâches feuilles sont sélectionnées ; aussi comme chaque tâche feuille a moins de ρ prédécesseurs, ordonnancer l'ensemble de ces m tâches ainsi que leurs prédécesseurs en $\rho+1$ unités de temps est facile. Il suffit d'exécuter chaque tâche feuille selectionnée et ses prédecesseurs sur un processeur. Comme une tâche peut être predécesseur de plusieurs tâches feuilles selectionnées, il

Algorithme 6 `Ordonnanceur_petit`(H, m, ρ)

$k = 0$, $t = 0$
tant que `Taches`$(H) \neq \emptyset$ **faire**
  $S =$ `Selection`$(H, 2^k.m)$
  `Ordonnancer`(H, S, t)
  si $W(S) \leq m.\frac{\rho+1}{2}$ **alors**
    $k = k + 1$
  fin si
  $t = t + 2.\rho$
  $H =$ `Induit`$(H,$ `Taches`$(H) \setminus A(S))$
fin tant que

peut alors être necessaire de dupliquer certaines tâches. Nous pouvons aussi remarquer qu'une phase de communication de durée $\rho - 1$ unités de temps est suffisante. En effet les tâches qui sont exécutées en dernier sont des tâches feuilles, qui n'ont pas de successeurs.

Cependant au cours d'une autre phase, il y a $2^k.m$ tâches feuilles qui sont sélectionnées. Et celles-ci doivent être ordonnancées en moins de $\rho + 1$ unités de temps. Pour que cela soit possible nous devons préciser comment nous sélectionnons les tâches feuilles que nous souhaitons exécuter.

4.2. Selection des tâches

Au cours d'une phase nous cherchons simplement à maximiser le nombre de tâches qui vont être éxécutées, c'est-à-dire $W(S)$. Cependant comme ce problème est NP-difficile, nous sélectionnons simplement un ensemble de tâches S qui est localement optimal. C'est-à-dire que si nous échangeons une tâche feuille y non sélectionnée avec une tâche x dans S alors le nombre total de tâches qui seront exécutées $W(S)$ n'augmente pas. Plus formellement, nous voulons la propriété suivante :

Pour toute tâche feuille x dans S et pour tout tâche feuille y non dans S

$$W(S) \geq W(S \setminus \{x\}) \cup \{y\}) \tag{1}$$

Construire un ensemble de tâches S qui satisfasse cette propriété est facile et peut être réalisé par un algorithme polynomial. Une procédure 2-Opt permet par exemple de le faire. Elle peut être décrite de la façon suivante : étant donné un ensemble de tâches feuilles arbitraire X de taille $2^k.m$, nous pouvons par exemple considérer toutes les paires de tâches feuilles $(x, y) \in (X \times \bar{X})$ et échanger les tâches x et y dans l'ensemble X si cela permet de faire augmenter $W(X)$ (nous désignons

ici par $\bar{X}$ le complémentaire de X dans l'ensemble des tâches feuilles de H). Cette opération peut alors être répétée jusqu'à ce qu'il n'existe plus de tel échange.

Un algorithme de sélection localement optimal garantit que si peu de tâches sont exécutées au cours d'une phase de travail alors la hauteur du graphe restant a diminué de façon significative. Cette idée est la clé de notre algorithme d'ordonnancement de petits graphes ; elle est clarifiée dans le lemme suivant.

Lemme 6 Considérons une phase de l'algorithme. Soit S l'ensemble des feuilles sélectionnées au cours de cette phase et soit H' le graphe restant à ordonnancer après cette phase. Nous avons :

$$h(H') \leq \frac{W(S)}{|S|}$$

Preuve. Etant donnée une tâche feuille z dans S, nous notons $w_S(z) = W(S) - W(S \setminus \{z\})$. Intuitivement, $w_S(z)$ correspond au nombre de tâches qui sont prédécesseurs de la tâche z et d'aucune autre tâche dans S. De manière triviale nous avons donc $W(S) \geq \sum_{z \in S} w_S(z)$. En utilisant le principe de la «cage à pigeons» nous pouvons affirmer qu'il existe une tâche x dans S telle que :

$$w_S(x) \leq \frac{W(S)}{|S|} \tag{2}$$

Considérons maintenant une tâche feuille y n'appartenant pas à S. La tâche y ne sera pas ordonnancée au cours de la phase considérée. Par construction, nous avons $h_{H'}(y)+1 = W(S \cup \{y\}) - W(S)$. D'autre part, pour tout ensemble de tâches feuilles X et Y, nous avons $W(X \cup Y) - W(X) = W(Y) - |A(X) \cap A(Y)|$. En appliquant cette équation d'une part avec $X = S$ et $Y = \{y\}$ et d'autre part avec $X = (S \setminus \{x\})$ et $Y = \{y\}$, nous obtenons facilement : $W(S \cup \{y\}) - W(S) \leq W((S \setminus \{x\}) \cup \{y\}) - W(S \setminus \{x\})$. Or l'ensemble des tâches sélectionnées est localement optimal donc d'après l'équation 1 : $W((S \setminus \{x\}) \cup \{y\}) - W(S \setminus \{x\}) \leq W(S) - W(S \setminus \{x\})$. Finalement nous obtenons l'équation suivante :

$$h_{H'}(y) + 1 \leq w_S(x) \tag{3}$$

En combinant cette dernière équation avec l'équation 2, nous obtenons que pour toute tâche feuille y non sélectionnée $h_{H'}(y) + 1 \leq \frac{W(S)}{|S|}$. Comme cela est valable pour toute tâche feuille y et donc pour tout tâche du graphe H', nous pouvons conclure. □

4.3. Ordonnancement des tâches sélectionnées

Au cours de chaque phase, nous sélectionnons un ensemble de $|S|$ tâches avec $|S| = 2^k.m$, il faut maintenant ordonnancer ces tâches ainsi que tous leurs prédécesseurs en $\rho + 1$ unités de temps. Nous pouvons d'abord remarquer qu'une durée de $\rho - 1$ unités de temps pour les communications est suffisante car les tâches S sont des tâches feuilles.

Corollaire 7 Soit H' le graphe restant après une itération de l'algorithme, soit S l'ensemble des tâches sélectionnées et k la valeur de la variable k au début de cette itération.
Si $W(S) \leq \frac{\rho+1}{2}.m$, nous avons $h(H') \leq \frac{\rho+1}{2^{k+1}}$

Ce corollaire est une conséquence du lemme 6. Nous pouvons remarquer qu'un tel résultat peut également être obtenu avec un algorithme de recouvrement (covering) tel que celui analysé par Hochbaum et Pathria [HP98]. L'idée de celui-ci consiste à sélectionner les tâches feuilles en utilisant un algorithme glouton. Plus précisément la tâche x tel que $W(S \cup \{x\}) - W(S)$ soit maximum est ajoutée à l'ensemble S. Cette itération est alors répétée jusqu'à ce que $|S| = 2^k.m$.

Le résultat de ce corollaire peut s'interpréter de la façon suivante :

- La condition correspond au fait que le nombre de tâches originales exécutées au cours d'une phase est plus petit que la moitié de la surface d'une phase. La surface d'une phase correspond au nombre maximal de tâches pouvant être exécutées au cours d'une phase.
- Le résultat de l'implication correspond au fait que le graphe restant peut être ordonnancé sur un nombre non borné de processeurs en une durée de $\frac{\rho+1}{2^{k+1}}$ unités de temps.

Par induction sur k, nous pouvons donc aisément déduire du corollaire précédent 7 le résultat suivant.

Corollaire 8 Si H désigne le graphe au début d'une itération et k désigne la valeur de la variable k au début de la même itération, alors :

$$h(H) \leq \frac{\rho + 1}{2^k}$$

On en déduit que chaque tâche feuille peut être ordonnancée avec tous ces prédécesseurs sur un processeur en une durée inférieur à $\frac{\rho+1}{2^k}$. Il est donc facile d'ordonnancer les $2^k.m$ tâches feuilles, sur m processeurs en $\rho + 1$ unités de temps.

4.4. Garantie de performance

Nous avons maintenant toutes les clés pour analyser la garantie de performance de notre algorithme pour ordonnancer des petits graphes.

Lemme 9 L'algorithme 6 peut ordonnancer tout graphe H de hauteur plus petite ou égale à $\rho+1$ en une durée inférieure ou égale à :

$$4.\frac{|H|}{m} + \big(1 + 2.\log(\rho+1)\big).(\rho+1)$$

Preuve. Pour prouver ce résultat, nous pouvons distinguer deux types de phases :
- Les phases où plus de $\frac{\rho+1}{2}.m$ tâches originales sont exécutées. Le nombre w de ces phases peut être facilement borné à l'aide d'un argument de travail, on obtient alors $w \leq \frac{2}{\rho+1}.\frac{|H|}{m}$.
- Les phases où strictement moins de $\frac{\rho+1}{2}.m$ tâches originales sont exécutées. Le nombre de ces phases k correspond aussi à la valeur de la variable k à la fin de l'algorithme. Considérons la dernière phase de ce type, soit F le graphe à ordonnancer au début de cette itération et k la valeur de la variable à la fin de cette itération. Comme la hauteur du graphe F est non nul et d'après le corollaire 8, nous obtenons $1 \leq h(F) \leq \frac{\rho+1}{2^{k-1}}$, donc $k \leq \log(\rho+1)+1$.

Par définition $w+k$ correspond au nombre total de phases, et comme chaque phase a une durée de $2.\rho$, la durée totale de l'ordonnancement vaut moins que $(w+k).(2.\rho)-\rho$. On obtient finalement :

$$\omega \leq \frac{4.\rho}{\rho+1}.\frac{|H|}{m} + \big(1 + \frac{2.\rho}{\rho+1}.\log(\rho+1)\big).(\rho+1)$$

□

Si l'on combine maintenant le corollaire 5 et le lemme 9 alors nous pouvons énoncer le résultat important de ce chapitre.

Lemme 10 Les algorithmes 5 et 6 permettent, lorsqu'ils sont combinés, d'ordonnancer tout graphe en une durée inférieur ou égale à :

$$4.\frac{|G|}{m} + 2.\big(\log(\rho+1)+1\big).h(G) + \big(2.\rho.\log(\rho+1)+1\big).(\rho+1)$$

Comme $\frac{|G|}{m}$ est une borne inférieure triviale du temps d'exécution d'un ordonnancement ω et comme $h(G)$ est aussi une borne inférieure de la durée d'un ordonnancement d'après le lemme 2, on peut en déduire le corollaire suivant.

Corollaire 11 Si la durée optimale d'un ordonnancement ω^* du graphe G est superieure au délai de communication ρ, alors les algorithmes proposés permettent d'ordonnancer le graphe G avec une garantie de performance de l'ordre de $\mathcal{O}(\log(\rho))$.

5. Conclusion

Dans ce chapitre, nous avons proposé et étudié une façon naturelle d'aborder le problème de l'ordonnancement avec un grand délai de communication. Elle consiste à découper le graphe initial en un ensemble de graphes pouvant être ordonnancés en $\rho+1$ unités de temps sur une infinité de processeurs avec duplication. Nous avons ainsi montré et isolé l'une des principales difficultés : le problème de l'ordonnancement de «petit graphe» sur un nombre borné de processeurs.

Cependant, bien que nos résultats améliorent très significativement les précédents résultats autour de l'ordonnancement avec grand temps de communication, ils ne répondent pas à la question de l'existence d'un algorithme d'approximation constante pour ce problème. Aussi de nombreuses questions restent ouvertes : notre approche permet-elle de trouver un algorithme d'approximation constante ? Existe-t-il un algorithme polynomial pour ordonnancer efficacement tout «petit graphe» ? Ou plus précisément, pouvons nous trouver un algorithme polynomial qui puisse ordonnancer tout petit graphe G en une durée proportionnelle à une constante près à $\frac{|G|}{m} + \rho$ sur m processeurs ? Cette borne est-elle suffisante ? Ou bien, au contraire, existe t'il une suite d'instance $(G_i, \rho_i, m_i)_{i \in I\!N}$ telle que la durée d'un ordonnancement optimal ω_n^* du «petit graphe» G_i sur m_i processeurs avec un délai ρ_i soit supérieure à $n.(\frac{|G_i|}{m_i} + \rho_i)$? Nous pensons que, tout au moins pour des structures de graphes particulières telles que des graphes d'arité bornée, il doit être possible de construire des algorithmes d'approximation constante pour ordonnancer avec duplication tout graphe sur un nombre borné de processeurs. Je continue à travailler activement sur ce problème en collaboration avec Christophe Rapine.

D'un autre coté, il est apparu que le problème d'ordonnancement avec duplication d'un graphe de très petite hauteur ($h(G) \ll \rho$) était très difficile. En effet, pour de telles instances une seule phase de communication peut être déjà trop long. Il convient de partitionner les différentes feuilles du graphe sur les processeurs tout en exécutant également les ancêtres associés sur chaque processeur. Ce problème est très proche des de problèmes sacs à dos avec précédences, des problèmes de recouvrement d'ensemble ou des problèmes de gestion de machines et outils [CK99], ces deux derniers problèmes étant connus pour leur non-approximabilité. Il semble donc illusoire de chercher un algorithme d'approximation constante pour des graphes de hauteur inférieure à ρ. Cependant d'un point de vue pratique, ce problème n'est

pas du tout gênant ! En effet, nous avons toujours négligé un coût de lancement des différentes tâches et cela nécessite au moins une communication. On peut donc toujours considérer que la durée d'un ordonnancement est supérieur à ρ.

Enfin, les résultats que nous avons établis sont valables pour un modèle de type délai uniforme, on peut se demander s'il est possible d'étendre ces résultats pour des modèles plus fins comme des modèles à délais variables. Cela parait cependant délicat car les algorithmes que nous avons utilisé jusqu'ici sont des algorithmes par phase de longueur proportionnelle à un délai.

Conclusion

Lors de l'exécution d'une application parallèle sur une architecture à mémoire distribuée, les unités de calculs sont très souvent amenées à communiquer entre elles pour partager des données ou des résultats. La prise en compte du coût de ces communications dans la décision d'ordonnancement et de placement des calculs apparaît comme un facteur clé pour obtenir des exécutions efficaces. J'ai étudié dans le cadre de ma thèse des problèmes d'ordonnancement sous des modèles permettant une prise en compte efficace de ces communications : le modèle des tâches malléables et le modèle délai à grand temps de communication.

Le modèle des tâches malléables s'appuie sur un modèle d'application où les tâches qui la composent sont elles-mêmes des activités parallèles pouvant s'exécuter sur un nombre variable de processeurs. Dans ce cadre, nous avons proposé des algorithmes d'approximation avec garantie constante pour le problème d'ordonnancement de tâches malléables avec des contraintes de précédence. Plus précisément, pour des graphes de précédence de type arbre nous avons obtenu une garantie de performance de $\frac{3+\sqrt{5}}{2}$ et de $3+\sqrt{5}$ pour des graphes quelconques. Il serait sans nul doute encore possible d'améliorer encore ces résultats en proposant des algorithmes d'approximation offrant de meilleures garanties ; mais il me semble plus intéressant d'étudier en pratique le comportement de ces heuristiques dans des applications réelles qui se prêtent bien à une parallélisation avec le modèle des tâches malléables comme la décomposition de domaine. Ces heuristiques pourraient alors être comparées à des heuristiques plus classiques comme le «gang» (heuristique qui consiste à exécuter toutes les tâches sur tous les processeurs) ou des heuristiques de type placement par niveaux, qui n'offrent elles aucune garantie de performance.

Nous nous sommes ensuite intéressés au problème plus classique de l'ordonnancement avec délai de communication. Nous avons choisi de considérer ce problème dans le cadre des grands délais de communication, car ce modèle est beaucoup plus réaliste que les modèles avec petits temps de communication. Les problèmes d'ordonnancement avec grand temps de communication sont des problèmes délicats pour lesquels il n'existe pas encore de solution satisfaisante.

Dans un premier temps, nous avons étudié ce problème sur une infinité de processeurs : le problème du regroupement. Il s'agit de regrouper des tâches entre elles pour éviter de trop nombreuses communications. Nous avons caractérisé une famille de regroupements 2-dominants : les regroupements convexes. En d'autres termes,

cela signifie qu'il existe un regroupement convexe dont la durée d'exécution est plus petite que deux fois la durée d'exécution d'un regroupement optimal. En plus de simplifier grandement la recherche d'un algorithme d'approximation pour ce problème, nous avons également montré que les regroupements convexes pouvaient être très utiles pour permettre un repliement sur un nombre borné de processeurs.

Nous avons ensuite proposé une approche originale pour construire des regroupements convexes. Notre approche est basée sur une décomposition récursive du graphe de précédence. Pour la construire nous utilisons une découpe du graphe de précédence consistant à trouver deux groupes de tâches indépendants et les plus grands possibles. Ce problème original que nous avons appelé *le problème de partitionnement de DAG* est au cœur de notre approche, nous avons commencé son étude théorique et proposé des heuristiques efficaces pour le résoudre. Nous avons validé pratiquement notre approche sur des simulations à partir de graphes de précédence d'applications réelles ; les premiers résultats sont très encourageants.

Ces recherches méritent d'être poursuivies, car les perspectives de ce travail tant du point de vue théorique que pratique sont nombreuses. Dans un premier temps, nous souhaitons valider notre approche dans le cadre d'un environnement de programmation d'applications parallèles Athapascan1. Un travail dans cette voie est en cours actuellement. Il s'agit d'intégrer notre approche et d'autres heuristiques d'ordonnancement statiques dans cet environnement. D'autre part, notre algorithme de décomposition récursive offre une vision structurée et hiérarchique des calculs effectués par une application parallèle. Cette vision me semble primordiale pour prendre en compte les communications importantes et hiérarchiques des architectures émergentes comme les interconnexions de grappes de multiprocesseurs.

Nous avons également étudié le problème de l'ordonnancement avec grand temps de communication sur un nombre fixé de processeurs en considérant la possibilité de dupliquer certaines tâches. La duplication permet de réduire les sur-coûts dus aux communications coûteuses. Notre objectif initial était de trouver un algorithme d'approximation constante pour ce problème. Si nous n'avons pas réussi, nous avons cependant nettement amélioré les résultats existants.

Nous avons proposé et étudié une façon naturelle d'aborder ce problème d'ordonnancement. Elle consiste à découper le graphe initial en un ensemble de petits graphes relativement au délai de communication et à ordonnancer successivement chacun de ces petits graphes. Nous avons ainsi montré et isolé l'une des principales difficultés : le problème de l'ordonnancement de ces petits graphes relativement au délai de communication sur un nombre borné de processeurs. Cette approche nous a permis d'obtenir un algorithme d'approximation dont la garantie est $\mathcal{O}(\log(\rho))$ où ρ désigne le délai de communication pour des graphes de précédences quelconques. Nous pensons que ces résultats pourront encore être améliorés en considérant des graphes de précédences plus spécifiques, comme par exemple des graphes d'arité bornée. Les recherches dans cette direction doivent être poursuivis.

Bibliographie

[ABFM99] Afrati, Bampis, Finta, and Milis. 2 processors scheduling with large communication delays. personnal communication. submitted., 1999.

[ACHL89] F.D. Anger, Y-C. Chow, J-J. Hwang, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2) :244–257, apr 1989.

[AD98] D. P. Agrawal and S. Darbha. Optimal scheduling algorithm for distributed-memory machines. *IEEE Transactions on parallel and distributed systems*, 9(1) :87–95, January 1998.

[AISS97] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP : Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1) :71–79, 1997.

[AK98] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on parallel and distributed systems*, 9(9), 1998.

[BCR80] Baker, Coffman, and Rivest. Othogonal packings in two dimensions. *SIAM journal on Computing*, 1980.

[BD99] Eric Blayo and Laurent Debreu. Adaptive mesh refinement for finite difference ocean model : some first experiments. *Journal of Physical Oceanography*, 29(6) :1239–1250, 1999.

[BDMT99] E. Blayo, L. Debreu, G. Mounié, and D. Trystram. Dynamic load balancing for ocean circulation model with adaptive meshing. In Patrick et al. Amestoy, editor, *Euro-Par' 99 Parallel Processing - 5th International Euro-Par Conference*, number 1685 in Lecture Notes in Computer Science, pages 303–312, August 31 - September 3 1999.

[BDSV98] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms : From parallelism extraction to code generation. *Parallel Computing*, 24(3–4) :421–444, 1998.

[BDW86] Jacek Blazewicz, Mieczyslaw Drabowski, and Jan Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers*, 35(5) :389–393, 1986.

[BK88] J. C. Browne and S. J. Kim. A general approach to the mapping of parallel computations upon multiprocessor architectures. In *Proc. of the 1988 International Conference on Parallel Processing*, volume III,

Algorithms and Applications, pages 1–8, University Park, Penn, 1988. Penn State.

[BR97] C. Boeres and V. E. F. Rebello. Versatile task scheduling of binary trees for realistic machines. *Lecture Notes in Computer Science*, 1300, 1997.

[CC91] J. Colin and P. Chrétienne. CPM scheduling with small communication delays and task duplication. *Operations Research*, 39(4), 1991.

[CCLL95] P. Chrétienne, E.G. Coffman, J.K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*, chapter Scheduling with Communication Delays : A Survey. Wiley, New York, 1995.

[CDGR98] Gerson Cavalheiro, Mathias Doreille, François Galilée, and Jean-Louis Roch. Athapascan-1 : On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, pages 88–95, Paris, France, October 1998.

[CGJT80] Coffman, Garey, Johnson, and Tarjan. Performance bounds for level-oriented two dimensional packing algorithms. *SIAM journal on Computing*, 1980.

[CJ99] Michel Cosnard and Emmanuel Jeannot. Compact DAG representation and its dynamic scheduling. *Journal of Parallel and Distributed Computing*, 58(3) :487–514, 1999.

[CK94] P. Crescenzi and V. Kann. A compendium of $\sqrt{}$ optimization problems, 1994. www.nada.kth.se/ viggo/problemlist/compendium.html.

[CK99] Y. Crama and J.J. van de Klundert. The approximability of tool management problems. *Naval Research Logistics*, 46 :445–462, 1999.

[CKP+96] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP : A practical model of parallel computation. *Communications of the ACM*, 39(11) :78–85, 1996.

[CMPS97] H. Choo, J. Marquis, G.-L. Park, and B. Shirazi. Decisive path scheduling : A new list scheduling method. In *Proc. of the Int. Conf. on Parallel Processing,*, pages 472–480, 1997.

[CMS98] R. Chapman, C. McCreary, and F.-S. Shieh. Using graph parsing for automatic graph drawing. *IEEE Transactions on Systems, Man and Cybernetics*, 28(5), 1998.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the* 3^{rd} *Annual ACM Symposium on Theory of Computing,* Shaker Heights, Ohio, pages 151–158, 1971.

[CR98] Franck Cappello and Olivier Richard. Architectures parallèles à partir de réseaux de stations de travail : réalités, opportunités, enjeux. *Calculateurs Parallèles*, 10(1), 1998.

[DDGW97] P. De, E.J. Dunne, J.B. Gosh, and C.E. Wells. Complexity of the discrete time-cost tradeoff problem for project networks. *Operations Research*, 45, 1997.

[Die98] Hank Dietz. Linux parallel processing howto, 1998. www.linuxdoc.org/HOWTO/Parallel-Processing-HOWTO.html.

[DKM99] X. Deng, E. Koutsoupias, and P. MacKenzie. Competitive implementation of parallel programs. *Algorithmica*, 23(1) :14–30, 1999.

[DL89] J. Du and J.Y-T. Leung. Complexity of scheduling parallel tasks systems. *SIAM Journal on Discrete Mathematics*, 2(4) :473–487, nov 1989.

[Dor99] Mathias Doreille. *Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. PhD thesis, Institut National Polytechnique de Grenoble, 1999.

[Dro96] M. Drozdowski. Scheduling multiprocessor tasks - an overview. *European Journal of Operational Research*, 94 :215–230, 1996.

[DRV00] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser Verlag, 2000.

[EGMS94] Andrzej Ehrenfeucht, Harold N. Gabow, Ross M. McConnell, and Stephen J. Sullivan. An $o(n^2)$ divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs. *Journal of Algorithms*, 16(2) :283–294, 1994.

[ELW97] Joern Eisenbiegler, Welf Loewe, and Andreas Wehrenpfennig. On the optimization by redundancy using an extended logp model. In *Advances in Parallel and Distributed Computing (APDC'97)*, pages 149–155. IEEE Computer Society Press, 1997.

[Fea95] Paul Feautrier. Compiling for massively parallel architectures : a perspective. *Microprocessing and Microprogramming*, 41 :425–432, 1995.

[FKS93] A. Feldmann, M-Y. Kao, and J. Sgall. Optimal online scheduling of parallel jobs with dependencies. In *25th Annual ACM Symposium on Theory of Computing*, pages 642–651, San Diego, California, 1993. url : http ://www.ncstrl.org, CS-92-189.

[Fly66] M. J. Flynn. Very high-speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, 1966.

[Fu97] C. Fu. *Scheduling and runtime support for Irregular Computations*. PhD thesis, University of California Santa Barbara, 1997.

[GG75] M. R. Garey and R. L. Graham. Bounds on multiprocessor scheduling with resource constraints. *SIAM J. Comput.*, 4 :187–200, 1975.

[GGJ78] M.R. Garey, R.L. Graham, and D.S. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1) :3–21, jan 1978.

[Gin97] Ilan Ginzburg. *Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications*. PhD thesis, Institut National Polytechnique de Grenoble, 1997.

[Gir00] Rodolphe Giroudeau. *L'impact des délais de communications hiérarchiques sur la complexité et l'approximation des problèmes d'ordonnancement*. PhD thesis, Université d'Évry Val d'Essonne, 2000.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability : A guide to the theory of NP-complete ness.* W.H. Freeman, New York, 1979.

[GM89] H. Gill and C. McGreary. Automatic determination of grain size for efficient parallel processing. *Communications of the ACM*, 32(5) :1073–1078, 1989.

[Gol99] Alfredo Goldman. *Impact des modèles d'exécution pour l'ordonnancement en calcul parallèle.* PhD thesis, Institut National Polytechnique de Grenoble, 1999.

[Gra66] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell Systems Technical Journal*, 45 :1563–1581, 1966.

[GY92] A. Gerasoulis and T. Yang. PYRROS : static scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 428–437. ACM, jul 1992.

[GY93] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6) :686–701, 1993.

[GY94a] A. Gerasoulis and T. Yang. DSC : Scheduling parallel tasks on an unbounded number of processo rs. *IEEE Transaction on Parallel and Distributed Systems*, 5 :951–967, 1994.

[GY94b] Apostolos Gerasoulis and Tao Yang. Performance bounds for column-block partitioning of parallel Gaussian elimination and Gauss-Jordan methods. *Applied Numerical Mathematics : Transactions of IMACS*, 16(1–2) :283–297, 1994.

[HLV94] 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.

[HM97] C. Hanen and A. Munier. Using duplication for scheduling unitary tasks on m processors with unit communication delays. *Theoretical Computer Science*, 178(1–2) :119–127, 1997.

[Hoc96] D. Hochbaum, editor. *Approximation Algorithms for $\mathcal{NP}$-Hard Problems.* Pws, September 1996.

[HP98] D. Hochbaum and A. Pathria. Analysis of the greedy approach in covering problems. *Naval Research Quarterly*, 45 :615–627, 1998.

[HS87] 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.

[JCL96] David S.L. Wei Jing-Chiou Liou, Michael A. Palis. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on parallel and distributed systems*, 7(1), 1996.

[JP98] 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.

[JR92] A. Jakoby and R. Reischuk. The complexity of scheduling problems with communication delays for trees. *Lecture Notes in Computer Science*, 621 :165–177, 1992.

[KDCZ94] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks : Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[KHCR99] P. T. Koch, J. S. Hansen, E. Cecchet, and X. Ronsset de Pina. SciOS : An SCI-based software distributed shared memory. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, 1999.

[KKT00] Tomasz Kalinowski, Iskander Kort, and Denis Trystram. List scheduling of general task graphs under LogP. *Parallel Computing*, 26(9) :1109–1128, July 2000.

[KMJ94] A. A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. Technical Report CSE94-02, 14, 1994.

[Law82] E. Lawler. Recent results in the theory of machine scheduling. a. bachem, 1982., 1982.

[Lep01] R. Lepère. *Approches algorithmiques pour l'ordonnancement d'applications parallèles avec communications.* PhD thesis, Institut National Polytechnique de Grenoble, 2001. to appear.

[LK78] J. K. Lenstra and A. H. G. Rinnooy Kan. The complexity of scheduling under precedence constraints. *Operations Research*, 26 :22–35, 1978.

[LKV96] E. Lawler, T. Kailath, , and V. Roychowdhury T. Varvarigou. Scheduling in and out forests in the presence of communication delays. *IEEE Transactions on parallel and distributed systems*, 7(10), 1996.

[LP96] J. Liou and M. Palis. An efficient task clustering heuristic for scheduling dags on multiprocessors. In *Symposium of Parallel and Distributed Processing*, 1996.

[LTZ01] W. Löwe, D. Trystram, and W. Zimmermann. On scheduling send-graphs and receive-graphs under the logp model. *Information Processing Letters*, 2001. à paraitre.

[MPI95] Mpi : A message-passing interface standard, 1995. www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[MPI97] Mpi-2 : Extensions to the message-passing interface, 1997. www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[MRT99] 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.

[MRT01] G. Mounié, C. Rapine, and D. Trystram. A $\frac{3}{2}$ approximation algorithm for scheduling independant malleable tasks. *SIAM Journal on Computing*, 2001.

[Mun99] Alix Munier. Approximation algorithms for scheduling trees with general communication delays. *Parallel Computing*, 25(1) :41–48, January 1999.

[Pic95] C. Picouleau. Two new $\mathcal{NP}$-complete scheduling problems with communication delays and unlimited number of processors. *Discrete Applied Mathematics*, 60 :331–342, 1995.

[PM91] G. N. Srinivasa Prasanna and B. R. Musicus. Generalised multiprocessor scheduling using optimal control. In *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 216–228. ACM, 1991.

[PM96] 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.

[Pot95] Alex Pothen. Graph partitioning algorithms with applications to scientific computing. In D. E. Keyes, A. H. Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*. Kluwer Academic Press, 1995.

[Pth95] POSIX standard 1003.1c (pthread). IEEE Computer Society, 1995.

[PY90] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2) :322–328, 1990.

[Rap99] Christophe Rapine. *Algorithmes d'approximation garantie pour l'ordonnancement de tâches*. PhD thesis, Institut National Polytechnique de Grenoble, 1999.

[RL98] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3) :483–545, May 1998.

[RS87] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18 :55–71, 1987.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, 1989.

[SGI00] SGI 3000 family reference guide. Silicon Graphics, Inc., 2000. www.sgi.com/origin/3000/datasheet.html.

[Sku98] M. Skutella. Approximation algorithms for the discrete time-cost tradeoff problem. *Mathematics of Operations Research*, 23(4) :909–929, 1998.

[Spe00] Spec cpu 2000. Standard Performance Evaluation Corporation, 2000. www.spec.org/osg/cpu2000/results/.

[Ste90] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6) :12–24, June 1990.

[Ste97] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2) :401–409, 1997.

[TK95] M Thadani and Khalidi. An efficient zero-copy i/o framework for unix. Technical Report SMLI TR95, Sun Microsystems Lab, Inc., 1995.

[TWY92] J. Turek, J. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, 1992.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, 1990.

[vCGS92] T. vonEicken, D. Culler, S. Goldstein, and K. Schauser. Active messages : a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, 1992.

[Ver00] Jacques Verriet. Scheduling outtrees of height one in the logp model. *Parallel Computing*, 26(9) :1065–1082, 2000.

[VLL90] B. Veltman, B. J. Lageweg, and J. K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16(2–3) :173–182, dec 1990.

[VTH99] Projet RNRT VTHD, 1999. Plate-forme d'expérimentation IP/WDM Vraiment Très Haut Débit pour applications de l'Internet de nouvelle génération. www.vthd.org.

[WBT99] T. Warschko, J. Blum, and W. Tichy. On the design and semantics of userspace communication subsystems. In *Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'99)*, pages 2344–2350, 1999.

[WG90] Min-You Wu and Daniel D. Gajski. Hypertool : A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3) :330–343, 1990.

[YG92] T. Yang and A. Gerasoulis. PYRROS : Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, 1992.