Algorithmes d'ordonnancement pour les nouveaux supports d'exécution

Pierre-François Dutot

▶ To cite this version:

Pierre-François Dutot. Algorithmes d'ordonnancement pour les nouveaux supports d'exécution. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 2004. Français. NNT : . tel-00011516

HAL Id: tel-00011516
https://theses.hal.science/tel-00011516

Submitted on 1 Feb 2006

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

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

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

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

<u>THÈSE</u>

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : "Informatique : Systèmes et Communications"

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

présentée et soutenue publiquement

par

Pierre-François DUTOT

le 27 Août 2004

<u>Titre :</u>

Algorithmes d'ordonnancement pour les nouveaux supports d'exécution

---

Directeur de thèse : Denis TRYSTRAM

JURY

M. MICHEL COSNARD,	Président
M. VAN-DAT CUNG,	Examinateur
M. KLAUS JANSEN,	Rapporteur
M. YVES ROBERT,	Rapporteur
M. ARNOLD ROSENBERG,	Examinateur
M. DENIS TRYSTRAM,	Directeur de thèse

Remerciements

On ne peut certainement pas faire une bonne thèse seul, sans l'aide de ses collègues et le soutien de ses proches. Je ne déroge pas à cette règle et la liste des gens à remercier est longue, je vais donc essayer d'être organisé, et je m'excuse par avance auprès de ceux que j'ai oublié.

Avant les remerciements officiels, ma première pensée est pour Isabelle, qui a su me supporter et me soutenir dans les moments difficiles et particulièrement dans la dernière ligne droite, pendant les mois avant la soutenance. J'espère pouvoir t'aider de la même façon à supporter ce stress quand viendra ton tour.

J'aimerais également remercier ma famille de m'avoir soutenu dans mon choix de carrière depuis des années, et d'avoir contribué à la réussite du pot (merci Maman). Merci à Anne-Marie pour sa relecture et ses petits choux. Ceux qui ont aimé la mousse au chocolat peuvent écrire à Grégory (`mounie@imag.fr`) pour en redemander un saladier.

Pour la partie travail, tout d'abord je voudrais remercier le premier de mes collègues, c'est-à-dire[1] mon directeur de thèse Denis Trystram. Il m'a fait confiance en me prenant en thèse alors que mon stage de DEA était sur un sujet totalement différent, et j'ai bon espoir qu'il ne regrette pas. Discuter avec Denis a toujours été très enrichissant sur tous les sujets, en allant de l'informatique à la littérature en passant par de la philosophie, de la sociologie, de l'œnologie et de la politique.

Je voudrais également remercier les membres de mon jury, qui sont pour la plupart venus de loin pour assister à ma soutenance. Merci à Michel Cosnard qui a accepté de présider ce jury, et qui a excellé dans ce rôle. Merci à Yves Robert et Klaus Jansen, qui ont évalué la thèse et vérifié tous les petits détails dans les preuves les plus longues. Merci à Van-Dat Cung et Arnold Rosenberg, qui ont accepté d'être examinateurs de la thèse.

Je voudrais également remercier les deux coauteurs qui ont travaillé avec moi, Grégory et Lionel. Ils forment avec Guillaume le socle du groupe O2 tel

[1] Tu as vu Denis, maintenant j'arrive à écrire «c'est-à-dire» avec les tirets.

que je l'ai connu. Je souhaite à Lionel de réussir sa thèse, et j'espère pouvoir travailler à nouveau avec lui par la suite[2].

Il reste encore beaucoup de gens à remercier dans le laboratoire, à commencer par Emmanuel, Eiad et Florent, mes cobureaux par ordre chronologique qui m'ont supporté pendant ses années de thèse. Je souhaite à Manu et Eiad de réussir dans leurs nouvelles carrières respectives, et je souhaite à Florent de faire une bonne thèse sans trop de stress.

Une thèse c'est aussi près d'un millier de repas de midi (et la demi-heure qui suit), et des dizaines de barbecues et pots en tout genre. Pour tous ces bons moments passés en leur compagnie, je voudrais remercier tous les membres du laboratoire, avec un pensée particulière pour (retenez votre souffle avant de lire la liste) :

Adrien, Bruno, Corine, Cyril, Cyrille, Denis (l'autre Denis), Estelle, Feryal, Florent, Georges, Grégory, Guillaume, Jacques, Jean-Marc, Jean-Michel, Jeroslav, Jesus, Jonathan, Julien, Lionel, Loick, Luiz-Angelo, Marion, Mauricio, Maxime, Nicolas, Olivier, Philippe, Pierre, Pierre, Rémi, Samir, Stéphane et tous ceux que j'ai immanquablement oubliés.

[2] Je précise que je veux bien retravailler avec Denis et Grégory aussi, ne vous vexez pas !

Table des matières

Introduction **11**

I Tâches Malléables **19**

1 Introduction **21**
1.1 Modèle des tâches malléables 21
1.2 Ordonnancement par phases 22
1.3 Allocation minimale de temps t 23
1.4 Techniques de preuve 25
1.5 Approximation duale 25
1.6 Discussion géométrique 26

2 Complexité **29**
2.1 Chaînes de tâches identiques par phases 29
2.2 Exemple de déroulement de l'algorithme 31
2.3 Graphe de tâches identiques par phases 32
2.4 Chaînes de tâches différentes 33
2.5 Conclusion 34

3 Ordonnancement Hiérarchique **37**
3.1 Modifications du modèle de base 38
3.2 Structure de l'ordonnancement 41
3.2.1 Première partition des tâches 42
3.2.2 Réduction de l'allocation 43
3.3 Principe régissant l'ordonnancement 45
3.3.1 Ordonnancement de P_{plein} 45
3.3.2 Ordonnancement de P_1 46
3.3.3 Ordonnancement de P_2 47
3.3.4 Propriétés des étagères 48
3.4 Transformations 49
3.4.1 De P_2 vers P_1 ou P_{plein} 50

3.4.2 De P_1 vers P_{plein} 52
3.4.3 Superposition de deux tâches séquentielles de P_1 dans P_{plein} 53
3.5 Analyse 53
3.6 Résumé de l'algorithme 59
3.7 Cluster de biprocesseurs 60
3.8 Conclusion 61

4 Ordonnancement Multi-critère 63
4.1 Rappel des notations 63
4.2 Ordonnancement avec un critère unique 63
4.2.1 Optimisation du makespan 64
4.2.2 Optimisation de la somme des temps de complétion . . 65
4.3 Tour d'horizon du multi-critère 66
4.3.1 Première construction générique 67
4.3.2 Spécialisation d'un algorithme 68
4.4 Une famille d'algorithmes 71
4.4.1 Un meilleur algorithme MSWP 71
4.4.2 Une famille d'algorithmes 71
4.5 Introduction de l'aléatoire 74
4.6 Conclusion 77

II Tâches Divisibles 79

5 Chaînes de Processeurs 81
5.1 Définitions 82
5.2 Algorithme 85
5.3 Propriétés 87
5.4 Optimalité 89
5.5 Conclusion 90

6 Réseau en Pieuvre 91
6.1 Présentation de l'algorithme 93
6.2 Exemple 96
6.3 Conclusion 96

7 Arbres 99
7.1 Description des problèmes 100
7.1.1 3-Partition 100
7.1.2 POTMEAH 100

7.2 Réduction 101
7.2.1 Faire un ordonnancement à partir de 3-partition 102
7.2.2 D'un ordonnancement sur arbre à une solution de 3-Partition 104
7.3 Extensions du modèle 105
7.3.1 Avec latences 106
7.3.2 Avec un seul canal de communication 108
7.4 Conclusion 110

Conclusion et perspectives 113

Table des figures

1.1 À gauche une tâche totalement malléable, à droite une séquentielle . . . 22
1.2 Une suite de phases avec successivement 1, 2, 3, 4 et 9 tâches . 23
1.3 Ordonnancement sans la contrainte de phase et avec cette contrainte . . . 24
1.4 Allocation optimale et non contiguë. . . . 27
1.5 La seule allocation possible. . . . 28

2.1 Toutes les étapes du déroulement de l'algorithme où S_j et P_j sont mis à jour. . . . 33
2.2 Ordonnancement optimal . . . 34

3.1 Les tâches 1 et 2 sont placées idéalement, alors que la tâche 3 ne l'est pas ($m = 4$). . . . 40
3.2 Structure générale de l'ordonnancement. . . . 41
3.3 Exemple avec 4 tâches $m = 3$, $k = 4$. . . . 43
3.4 Réarrangement des tâches de la figure 3.3. . . . 44
3.5 Résultat de l'ordonnancement de P_{plein}. . . . 46
3.6 Ordonnancement de P_{plein} (gauche) et de P_1 (droite). Quelques processeurs peuvent être inutilisés. . . . 47
3.7 Ordonnancement de P_2. Les aires hachurées correspondent aux boîtes englobant des tâches allouées à $\frac{3}{4}k$ processeurs. . . . 48
3.8 Les processeurs inutilisés sont dans une meilleure configuration après le mélange des deux ensembles P_1 et P_{plein}. . . . 50
3.9 Seules quelques portions de tâches sont mélangées. . . . 51
3.10 Aspect de l'ordonnancement après les dernières transformations. 57
3.11 La tâche rouge est la seule mal alignée. . . . 60
3.12 Les tâches sont placées sur des processeurs contigus. . . . 61

4.1 Courbe du compromis C_{max} / $\sum_i \omega_i C_i$. . . . 68

4.2 Transformation d'un ordonnancement $O^*_{\sum C_i}$ en ordonnancement bi-critère (ici $k = 4$ et $\alpha = 2$). 72
4.3 Courbe de la famille d'algorithmes 74
4.4 Comparaison des différents algorithmes 77

5.1 Le premier noeud est la source des tâches. 83
5.2 Représentation graphique d'un ordonnancement. 84
5.3 L'algorithme en pseudo-code. 86
5.4 On peut toujours éviter de croiser les communications. 88

6.1 Une pieuvre 91
6.2 Transformation d'un nœud simple en nœud multiple. 92
6.3 Transformation de la figure 2.1. 93
6.4 Une petite pieuvre. 97
6.5 Ordonnancement optimal sur la chaîne de gauche. 97
6.6 Ordonnancement optimal sur la chaîne de droite 97
6.7 Arbre à un niveau induit par la transformation. 98
6.8 Ordonnancement optimal sur la pieuvre. 98

7.1 Un arbre (à gauche) et son ordonnancement (à droite) 99
7.2 Arbre utilisé dans la réduction 101
7.3 Ordonnancement de 8 tâches à partir d'une solution de 3-partition 103
7.4 Arbre avec communications de durée zéro. 108
7.5 Transformation d'un noeud. 109
7.6 Arbre modifié, sans lien de communication de temps 0. 110
7.7 Ordonnancement de 8 tâches sur l'arbre modifié. 111

Introduction

Depuis quelques années, la fin du progrès technologique pour les circuits intégrés est annoncée dans les conférences de parallélisme et présentée comme une des motivations du domaine de recherche en calcul parallèle. Chaque procédé de gravure des processeurs a ses limites, et les limites physiques paraissent absolues tant que le procédé suivant n'est pas inventé. Aujourd'hui cette fin tant annoncée n'a toujours pas eu lieu, mais cela ne rend pas le calcul parallèle inutile, bien au contraire. La démocratisation actuelle de l'informatique, liée à la baisse des prix du matériel (en particulier «l'entrée de gamme») a mis à la disposition du plus grand nombre une grande puissance de calcul sous la forme de grappes de processeurs. Cette évolution a rendu viable le modèle de plate-forme de calcul à bas prix.

Dans la dernière liste des 500 machines de calcul les plus puissantes publiée en novembre 2003 (voir [Top500]) 52 machines sont inscrites sous la dénomination de «réseaux de stations de travail» (NOW - *networks of workstations*) et 123 machines sont des clusters vendus par des grands constructeurs comme IBM, HP ou Dell. Les courbes fournies sur le site web de l'organisation Top500 [Top500] montrent que sur les 5 derniers classements ce nombre de clusters double a peu près chaque année.

Ce changement de paysage du calcul parallèle a créé une nouvelle demande d'algorithmes spécifiques pour tirer le meilleur parti de ces architectures. Pour concevoir ces algorithmes et démontrer leur efficacité, il faut s'appuyer sur des modèles qui tentent de décrire fidèlement le comportement réel des machines tout en restant suffisamment simples à manipuler.

Modèles

Les problèmes qui apparaissent avec ce nouveau type de plates-formes sont en fait des problèmes de gestion efficace de ressources. Il s'agit de déterminer le placement des calculs et des données sur une architecture répartie, ainsi que le déroulement dans le temps de l'exécution des programmes parallèles. Ces problèmes d'ordonnancement sont étudiés depuis longtemps déjà,

mais les caractéristiques des nouvelles grilles de calcul (résilience, hiérarchie, hétérogénéité) sont très différentes de celles des super-calculateurs dédiés du siècle dernier, d'où la nécessité d'élaborer de nouveaux modèles et de nouveaux algorithmes.

Pour une première introduction au domaine du parallélisme, plusieurs livres en français sont disponibles comme par exemple [CT93, LR03, MJT04].

Modèles classiques

On distingue généralement trois niveaux de modèles :

- Les modèles d'architecture
- Les modèles de calcul
- Les modèles de programmation

Les modèles d'architecture tentent de représenter mathématiquement le plus fidèlement possible le comportement global d'une plate-forme à l'aide de quelques paramètres. Les modèles de calcul se situent plus loin dans l'abstraction et prennent en compte certains aspects de la gestion des ressources (préemption, duplication, etc.). Les modèles de programmation sont utilisés pour exprimer le parallélisme des applications de la façon la plus précise possible pour en tirer le maximum de profit lors du déploiement de ces applications. La séparation entre ces trois niveaux de modèles est floue, les modèles de calcul pouvant aussi bien être également des modèles d'architecture (comme les PRAM [FW78],LogP [CKP+96]) ou des modèles de programmation (comme BSP [Val90]).

Les modèles étudiés dans cette thèse sont des modèles de calcul. Ces modèles récents, ainsi que les problèmes que j'ai étudié dans ces modèles, ont tous pour ancêtre le problème suivant, posé dans le modèle multiprocesseurs le plus simple qui soit [Ull75].

Définition 1 *On dispose de m machines identiques, reliées entre elles de façon complète par un réseau tellement rapide que les communications ont un temps nul. Il s'agit d'ordonnancer en un temps minimum sur cette machine un graphe $G = (V, E)$ de tâches identiques de temps d'exécution 1 reliées entre elles par des relations de précédence*[3].

En notant $\sigma(i)$ la date de début d'exécution de la tâche i, il s'agit de déterminer pour toutes les tâches une date de début d'exécution telle que :

1. *pour tout j, $|\{i/\sigma(i) = j\}| \leq m$*
2. *pour tout $e = (i, j) \in E$, $\sigma(i) + 1 \leq \sigma(j)$*
3. *$\max_i \sigma(i)$ soit le plus petit possible*

[3] Cette représentation des applications est utilisée depuis les années 70 [CD73].

La première condition est qu'à chaque instant au plus m machines sont utilisées, la deuxième indique que les relations de précédence doivent être vérifiées, et la dernière est l'objectif sur la longueur de l'ordonnancement. Ce problème d'apparence simple est en fait complexe à résoudre. Si m est une des entrées du problème, alors il est NP-dur au sens fort [Ull75], tandis que si m est fixé la complexité reste inconnue.

Le modèle délai étend ce modèle en rajoutant des temps de communications entre les tâches n'ayant pas lieu sur les mêmes processeurs. Les tâches peuvent également avoir des temps d'exécution quelconques, ainsi que des temps de communications quelconques. L'évolution suivante du modèle est de considérer des processeurs ayant des vitesses différentes, le temps de calcul d'une tâche sur un processeur étant alors le rapport entre le temps nominal de la tâche et la vitesse du processeur. Certaines études portent également sur les processeurs non liés, où chaque tâche a un temps d'exécution qui dépend du processeur sur lequel elle est exécutée.

Les temps de communications ont aussi fait l'objet d'études plus raffinées avec le modèle LogP par exemple, où l'on introduit les notions de latence, de surcoût et de *gap* (écart entre deux communications). Une étude intéressante sur plusieurs modèles est celle de Bampis *et al.* [BGT97].

Tous ces raffinements successifs du modèle de base visent à se rapprocher du comportement réel des machines. Mais la conception d'un modèle universel qui resterait utilisable par les théoriciens est un graal qui semble maintenant abandonné. Il paraît difficile d'adapter le modèle classique aux nouvelles caractéristiques des supports d'exécution. Ces nouveaux supports ont besoin de nouveaux modèles qui leur soient propres, et c'est dans cet esprit qu'ont été conçus et utilisés les deux modèles suivants.

Modèle des tâches malléables

Le modèle des tâches malléables [TWY92] a été créé pour séparer deux problèmes difficiles : d'une part la parallélisation fine au niveau de l'instruction, et d'autre part une parallélisation plus large au niveau supérieur, c'est-à-dire au niveau des fonctions ou même des programmes. Les tâches malléables sont donc des boîtes noires renfermant des procédures ou des programmes parallèles qui ont été traités individuellement et qu'il faut agencer sur une machine parallèle, ou une grille d'ordinateurs. La différence entre les tâches malléables et les tâches parallèles (qui sont également des tâches pouvant s'exécuter sur plusieurs processeurs) réside dans le choix du nombre de processeurs à réserver aux tâches. Dans le cadre des tâches parallèles classiques ce nombre est déterminé par l'utilisateur, tandis que pour les tâches malléables c'est l'ordonnanceur qui doit déterminer l'allocation des tâches.

Pour un aperçu des travaux existants sur le modèle des tâches parallèles, nous conseillons [Dro96] et [Dro04b].

Les communications à grain fin sont donc implicitement incluses dans le temps d'exécution des tâches, et les communications entre les tâches elles-mêmes sont supposées suffisamment petites pour pouvoir être négligées dans une première approche. Le temps d'exécution des tâches suivant le nombre de processeurs alloués peut être déterminé soit théoriquement par analyse du code et du graphe de tâche généré, soit plus pragmatiquement en analysant les traces d'exécution des programmes soumis à une grille de calcul [FR95].

Depuis les tout premiers articles [TWY92] dans lesquels le modèle apparaît, les différents groupes travaillant sur le sujet se sont surtout attachés soit à produire des heuristiques pour ordonnancer rapidement, en fournissant des garanties par rapport à une minoration de l'ordonnancement optimal [MRT99] (la plus récente pour les tâches indépendantes étant de 3/2), soit à donner des schémas d'approximations polynomiaux inapplicables pour les cas réels [JP02].

Les résultats les plus récents portent sur les problèmes avec contraintes de précédence [LTW01, LM00]. Là encore on a des garanties sur le rapport entre l'optimal et l'ordonnancement calculé par une heuristique (soit un rapport de $\frac{3+\sqrt{5}}{2} + \epsilon$ dans le cadre des arbres et des graphes séries/parallèles). Le modèle des chaînes de tâches est important à maîtriser car il représente bien l'exécution séquentielle d'un programme qui comporte des blocs d'instructions parallélisables. Plusieurs ouvrages reviennent sur l'ensemble des résultats du domaine, comme par exemple [Lud95] ou plus récemment [MT02].

Le modèle général des tâches malléables est actuellement utilisé pour paralléliser des applications réelles [BDMT99], de même que le modèle plus restreint des tâches identiques par phases [DK99].

Notre premier objectif ici est de cerner la frontière qui existe entre les problèmes *simples* que l'on peut résoudre en temps polynomial, et les problèmes plus complexes qui appartiennent à la classe des problèmes NP-difficiles au sens fort. La maîtrise de cette frontière permet de renforcer la légitimité des algorithmes d'approximation présentés dans la littérature.

Les problèmes étant souvent difficiles, le deuxième objectif est de fournir des heuristiques sur deux problèmes précis : l'ordonnancement sur une architecture hiérarchique et l'ordonnancement bi-critère.

Modèle des tâches divisibles

Le modèle des tâches divisibles a été créé pour modéliser le comportement d'applications ayant un très fort degré de parallélisme [CR88]. Par exemple, la recherche d'un mot dans une base de données peut être effectuée par

plusieurs processeurs en même temps, chaque processeur ayant accès à une partie de la base de données. Le partage de la charge de travail dans cet exemple se fait à un niveau de granularité très fin, en effet l'unité de base est le mot alors que la base de données en contient en général plusieurs millions. Ce type de comportement existe aussi pour beaucoup d'autres applications, comme l'analyse de données [SETI], la recherche des nombres premiers de Mersenne [Mers] ou des applications de bio-informatique comme l'étude du protéome [Dec] ou du repliement des protéines [Fold].

Les projets grand public cités ci-dessus utilisent actuellement la puissance de calcul fourni gracieusement par des volontaires par l'intermédiaire du réseau internet, sans utiliser d'algorithme évolué pour le placement des tâches. L'étude du déploiement de telles applications sur des grilles de calcul hétérogènes reste cependant un sujet intéressant. En effet de nombreuses applications peuvent être écrites de cette façon et elles ne peuvent pas toutes bénéficier de la publicité qu'ont eu les premières applications (donc elles ne peuvent attirer autant de participants bénévoles).

Dans le modèle général, le travail à effectuer est un volume de calculs qui peut être partitionné et distribué entre les processeurs de n'importe quelle façon. Le temps de calcul sur chaque processeur dépend linéairement de la taille du morceau qui lui est attribué et de sa vitesse, et similairement le temps de communication entre deux processeurs dépend linéairement du volume transmis ainsi que de la vitesse du lien. Il existe plusieurs sites qui regroupent des informations sur le domaine, ainsi qu'une bibliographie complète. Je conseille en particulier les sites de Maciej Drozdowski [Dro04a] et de Tom Robertazzi [Rob04].

Pour une première approche du sujet, je me suis concentré sur le cas particulier où le travail ne peut être partagé qu'en blocs de tailles égales (appelés tâches unitaires). Les ordonnancements optimaux pour le problème des tâches unitaires sont des approximations des ordonnancements généraux. En augmentant le nombre de tâches unitaires on peut théoriquement se rapprocher aussi près que l'on veut d'un ordonnancement optimal général, au prix d'un temps de calcul rapidement prohibitif. L'étude des tâches unitaires permet toutefois d'avoir des algorithmes d'approximation pour le cas général et de fixer des bornes de complexité.

Contributions

Le thème principal de ce travail a donc été l'étude de la gestion efficace de ressources, et en plus particulier l'ordonnancement de tâches sur des plateformes parallèles. Dans ce domaine encore très vaste, je me suis concentré sur

l'étude de deux modèles récents et prometteurs que sont les tâches malléables et le paradigme maître-esclave. Ce document est donc organisé en deux parties, la première aborde l'étude du modèle des tâches malléables, tandis que la seconde est centrée sur le modèle des tâches divisibles.

L'architecture de la première partie correspond globalement à l'ordre naturel qui va des problèmes les plus simples que j'ai rencontré en début de thèse jusqu'aux résultats les plus récents. Le premier chapitre présente une introduction au modèle, ainsi qu'aux différentes techniques de preuves utiles dans le cadre des tâches malléables.

Le deuxième chapitre est un premier regard sur les aspects les plus simples du modèle qui avaient jusqu'à présent été négligés par les précédents travaux. Les résultats portent sur la complexité du modèle en général et sur un cas particulier où toutes les tâches sont identiques et doivent s'exécuter par phases[4]. Ces travaux ont fait l'objet d'une publication [Dut02a] aux 14e Rencontres francophones du Parallélisme en 2002.

Le troisième chapitre est le plus proche de mon sujet de thèse original «Ordonnancement Hiérarchique». Il porte sur l'étude de l'ordonnancement des tâches malléables sur des processeurs ayant deux niveaux de communications. Pour les architectures les plus courantes, un algorithme efficace et garanti est donné. Cet algorithme [DT01] a été présenté à la 13e conférence SPAA (*Symposium on Parallel Algorithms and Architectures*) en 2001.

Le quatrième chapitre de cette partie présente les travaux les plus récents de cette thèse. Les contacts fréquents avec d'autres chercheurs du laboratoire ayant à leur charge l'administration de grilles de calcul a conduit à une réflexion sur les critères d'évaluation de nos algorithmes, ainsi qu'à la production d'algorithmes plus polyvalents qui optimisent simultanément plusieurs critères. La famille d'algorithmes présentés dans ce chapitre a fait l'objet d'un article [DT] actuellement soumis au journal Algorithmica. Un article lié à cette thématique a été accepté pour publication dans la 16e conférence SPAA [DEMT04], mais n'est pas reproduit ici. La principale différence entre les deux articles est le parti pris de départ. Dans l'article d'Algorithmica nous avons choisi de chercher des heuristiques garanties et de démontrer les valeurs des facteurs d'approximation, tandis que dans l'article de SPAA les algorithmes sont conçus pour être les plus efficaces et rapides possible dans le cas général en sacrifiant les garanties sur les cas pathologiques qui n'apparaissent pas dans l'utilisation normale d'une plate-forme de calcul.

La deuxième partie s'intéresse au modèle maître-esclave, qui permet d'étudier une autre classe d'applications, c'est-à-dire les applications forte-

[4]Les définitions sont données dans le premier chapitre.

ment parallèles ou paramétriques. Cette partie est présentée de façon chronologique, puisque c'est également le cheminement le plus naturel.

Le premier chapitre décrit un algorithme pour un réseau hétérogène simple : les chaînes de processeurs. Sur cette topologie, un algorithme optimal en temps polynomial existe. Cet algorithme a été présenté à l'école thématique sur la Globalisation des Ressources Informatiques et des Données [Dut02b] en décembre 2002.

Dans le chapitre suivant la topologie est légèrement plus complexe, dans le but (avoué) d'approcher les arbres. Cette topologie est celle des pieuvres, c'est-à-dire d'un ensemble de chaînes reliées au même noeud maître. Il y a ici aussi un algorithme optimal en temps polynomial, qui a fait l'objet d'une publication à IPDPS (*International Parallel and Distributed Processing Symposium*) en avril 2003.

Enfin le dernier chapitre de cette partie montre que pour une topologie plus complexe, le problème devient difficile à résoudre. En effet le problème de l'ordonnancement de tâches identiques en maître-esclave sur un arbre de processeurs est NP-dur au sens fort. Cette preuve [Dut04] est parue dans le journal EJOR (*European Journal of Operational Research*).

Première partie

Tâches Malléables

Chapitre 1

Introduction

1.1 Modèle des tâches malléables

Le modèle des tâches malléables tel qu'il a été introduit par Turek, Wolf et Yu [TWY92] repose sur les hypothèses suivantes. Une tâche i peut être allouée sur un nombre quelconque $p(i)$ de processeurs. Son temps d'exécution $t_i(p(i))$ dépend du nombre de processeurs sur lesquels elle est allouée. Cette allocation est exclusive, c'est-à-dire qu'un processeur ne peut travailler que sur une tâche à la fois.

Dans le cadre de ce travail, les tâches ne peuvent pas être préemptées, c'est-à-dire que l'exécution d'une tâche se déroule sans interruption sur tous les processeurs qui lui ont été alloués de façon exclusive. La préemption seule est rarement considérée en ce qui concerne les tâches malléables. Le modèle des tâches modelables est proche du modèle des tâches malléables, et prend en compte la préemption en permettant également le redéploiement, c'est-à-dire le changement de l'ensemble des processeurs exécutant la tâche. Le chapitre [DMT04] aborde plusieurs problèmes d'ordonnancement dans les deux modèles.

Une hypothèse fréquemment admise dans les travaux sur les tâches malléables (comme par exemple [MRT99, LTW01]) est que les tâches sont monotones. Le temps d'exécution est alors une fonction décroissante du nombre de processeurs, et le travail total $W_i(p(i))$ (produit du nombre de processeurs alloués à une tâche par son temps d'exécution) est alors une fonction croissante du nombre de processeurs, ce qui est conforme au comportement général des applications parallèles jusqu'à un certain point. Au delà de ce seuil où le temps d'exécution devient fortement lié aux temps de synchronisations et de communications, il suffit de laisser un certain nombre de processeurs inactifs pour garder le meilleur temps d'exécution possible.

Définition 2 *Hypothèse de monotonie*
Une tâche i est dite monotone si son temps d'exécution $t_i(j)$ vérifie les inégalités suivantes pour tout $j < m$, où m est le nombre de processeurs disponible.

$$\begin{aligned} t_i(j) &\geq t_i(j+1) \\ j \times t_i(j) = W_i(j) &\leq W_i(j+1) \end{aligned}$$

Pour la suite, il est utile de définir deux types de tâche ayant des profils complètement opposés (voir figure 1.1) :

Définition 3 *Totalement malléable*
On dit d'une tâche qu'elle est totalement malléable si son travail ne dépend pas du nombre de processeurs qui lui sont alloués.

$$\forall j \ W_i(j) = W_i(j+1)$$

Définition 4 *Séquentielle*
On dit d'une tâche qu'elle est séquentielle si son temps d'exécution ne dépend pas du nombre de processeurs qui lui sont alloués.

$$\forall j \ t_i(j) = t_i(j+1)$$

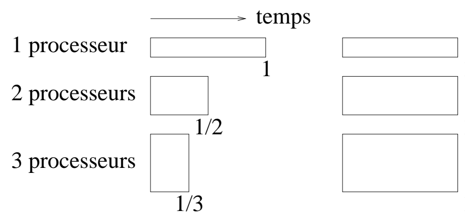

FIG. 1.1 – À gauche une tâche totalement malléable, à droite une séquentielle

1.2 Ordonnancement par phases

Dans la suite nous aurons besoin d'introduire le concept d'ordonnancement par phases (voir figure 1.2). Il s'agit en fait de coordonner l'exécution des tâches pour faciliter la gestion du système. Les tâches étant identiques, elles peuvent être groupées en phases, toutes les tâches d'une phase ayant leur début d'exécution synchronisé. Si les tâches ont la même allocation elles ont le même temps d'exécution, donc elles finiront toutes en même temps.

Cette synchronisation est nécessaire dans certains modèles comme le modèle BSP [Val90], où les calculs locaux et les phases de communication sont séparés, les programmes étant découpés en phases (généralement appelées «*Super Step*»). Ici le découpage calcul/communication n'est pas aussi absolu puisqu'il y a des communications à l'intérieur des tâches malléables. Mais les tâches étant identiques et synchronisées au début, on peut synchroniser les *super steps* des tâches entre elles.

Définition 5 *Ordonnancement par phases*
Un ordonnancement est dit "par phases" s'il y a une partition des tâches en groupes telles que pour toutes les tâches d'un groupe l'exécution commence au même instant, et que deux tâches appartenant à des groupes différents ne peuvent pas s'exécuter en même temps.

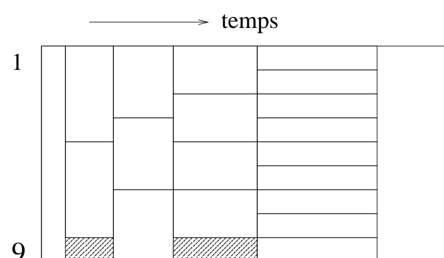

FIG. 1.2 – Une suite de phases avec successivement 1, 2, 3, 4 et 9 tâches

On peut exhiber quelques exemples pour lesquels la contrainte de l'ordonnancement par phases fait augmenter le temps de l'ordonnancement optimal. Le plus grand écart que l'on ait constaté est de 3/2 pour l'exemple décrit dans la figure 1.3. Il s'agit de deux chaînes, l'une de taille 2 et l'autre de taille 1. Les tâches sont toutes identiques, de longueur 1 sur 1 processeur, et de longueur 1/2 sur deux et sur trois processeurs. Le nombre total de processeurs est de trois. Il semblerait[1] que cette valeur de 3/2 est l'écart maximum possible dans le cas des chaînes.

1.3 Allocation minimale de temps t

L'allocation minimale de temps t de la tâche i (appelée *allocation canonique* dans la thèse de Grégory Mounié [Mou00]) est l'allocation ayant le plus

[1]Ce problème est encore ouvert à ma connaissance.

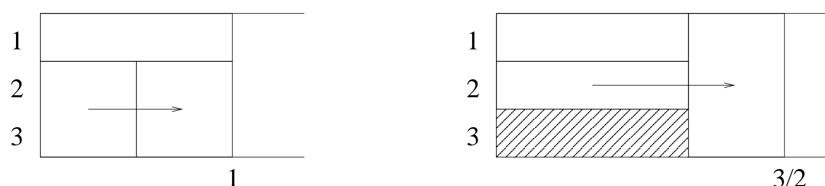

FIG. 1.3 – Ordonnancement sans la contrainte de phase et avec cette contrainte

petit nombre de processeurs telle que le temps d'exécution de la tâche $t_i(j)$ soit inférieur ou égal à t. On notera cette allocation $p_t(i)$.

Cette valeur n'est définie que quand t est plus grand que le temps d'exécution de la tâche sur toute la machine c'est-à-dire quand $t > t_i(m)$, m étant le nombre de processeurs de la machine.

Quand les tâches sont monotones, on peut montrer la propriété suivante sur le temps d'exécution dans l'allocation minimale de temps t.

Propriété 1 *Pour tout t tel que $p_t(i)$ est défini et différent de 1, on a :*

$$t_i(p_t(i)) > \frac{p_t(i) - 1}{p_t(i)} t$$

En effet si l'allocation est minimale et plus grande que 1, alors en enlevant un processeur à la tâche, son temps devient plus grand (strictement) que t. Le travail étant croissant avec le nombre de processeurs alloués, on a :

$$\begin{array}{rcl} W_i(p_t(i)) & > & W_i(p_t(i) - 1) \\ t_i(p_t(i)) \times p_t(i) & > & t_i(p_t(i) - 1) \times (p_t(i) - 1) \\ t_i(p_t(i)) \times p_t(i) & > & t \times (p_t(i) - 1) \end{array}$$

Une simple division donne alors le résultat énoncé. On peut remarquer que le résultat reste vrai si l'allocation minimale est réduite à un processeur puisque la propriété dans ce cas particulier traduit juste que le temps d'exécution sur un processeur est non nul[2].

[2] Il arrive qu'il soit théoriquement intéressant de considérer des tâches de temps d'exécution nul. Ce n'est pas le cas ici puisque les tâches sont soit indépendantes soit identiques.

1.4 Techniques de preuve

Certaines techniques de preuve revenant régulièrement dans cette partie sont présentées ici en détail. Examinons d'abord les preuves de garanties de performance.

La performance d'un algorithme d'ordonnancement sur un ensemble de tâches est le rapport entre la valeur du critère optimisé par l'algorithme et la meilleure valeur possible de ce critère pour l'ensemble de tâches considéré. Par exemple pour le *makespan* un algorithme qui finit toutes les tâches en 20 unités de temps quand l'optimal est 10 unités de temps a une performance de 2. De façon plus générale, on parle de garantie de performance, c'est-à-dire de la performance que l'on est sûr d'avoir avec un algorithme sur l'ensemble (généralement infini) des instances possibles.

La comparaison avec un ordonnancement optimal est compliquée par le fait que bien souvent connaître la valeur de cet optimal est déjà en soi un problème difficile. La comparaison est alors effectuée par rapport à une valeur que l'on sait inférieure à l'optimal. Si l'on a un algorithme qui obtient pour le critère considéré une valeur qui est le double d'une valeur inférieure à l'optimal, on est à moins de deux fois l'optimal.

Dans les preuves sur les ordonnancements de tâches multiprocesseurs indépendantes, les deux bornes inférieures classiques pour le *makespan* sont :

- Le temps de la tâche la plus longue[3].
- Le travail moyen effectué par les processeurs.

Pour les tâches malléables, ces deux valeurs dépendent évidemment de l'allocation choisie pour les tâches. À allocation fixée, elle restent utilisables. Sinon on considère l'allocation qui minimise la valeur de la borne, c'est-à-dire l'allocation sur toute la machine pour chaque tâche pour le temps de la tâche la plus longue et l'allocation sur un processeur pour le travail moyen.

1.5 Approximation duale

Une technique récente et particulièrement intéressante pour obtenir des algorithmes d'approximation garantis est la méthode d'approximation duale introduite par Hochbaum et Shmoys [HS87].

Un algorithme de ρ-approximation duale est un algorithme qui prend une valeur t et rend soit un ordonnancement de durée au plus ρt si t est plus petit que l'optimal, soit une erreur si t est plus grand que l'optimal. Avec un

[3] Quand les tâches sont liées par des relations de précédence, on utilise la longueur du plus long chemin dans le graphe de précédence.

algorithme de ρ-approximation, on peut effectuer une recherche dichotomique sur t pour approcher l'optimal aussi près que l'on veut.

Soit n le nombre de tâches. Le temps total d'exécution sur un processeur et le temps moyen d'exécution sur un processeur donnent deux bornes (une inférieure et une supérieure) du *makespan* séparées au plus de n. En $\log_2(n)+s$ étapes, on se trouve donc à moins de $1+2^{-s}$ de l'optimal.

Dans le cas de l'approximation duale, la connaissance de t permet de borner le travail effectué par l'optimal. Si t est plus grand que l'optimal, toutes les tâches ordonnancées par l'optimal ont une allocation supérieure ou égale à $p_t(i)$, $p_t(i)$ étant l'allocation minimale pour la tâche i en temps t. Donc le travail total (W^*) fait par l'optimal est au moins :

$$W^* \geq \sum_i (t_i(p_t(i)) \times p_t(i))$$

Une borne supérieure est également disponible, puisque le travail total de l'optimal est effectué dans le diagramme de Gantt dans un rectangle de taille m par t.

$$W^* \leq m \times t$$

On peut donc complètement encadrer la valeur du travail total.

Propriété 2

$$\sum_i (t_i(p_t(i)) \times p_t(i)) \leq W^* \leq m \times t$$

Cette propriété illustre bien une des deux bornes inférieures décrites ci-dessus. En divisant les deux termes de gauches par m on retrouve que le temps d'exécution moyen (W^*/m) est inférieur au temps de l'ordonnancement optimal (t). L'autre borne revient à écrire que t est plus grand que le plus grand des $t_i(p_t(i))$, ce qui ici est trivial par définition de $p_t(i)$.

1.6 Discussion géométrique sur les ordonnancements

À première vue, une fois l'allocation choisie, le modèle des tâches malléable devient très proche du problème de «*Strip packing*», c'est-à-dire du pavage de rectangle par des rectangles orientés. Ce problème étudié depuis les années 80 [BCR80], est encore aujourd'hui l'objet de nombreuses études [LMM02].

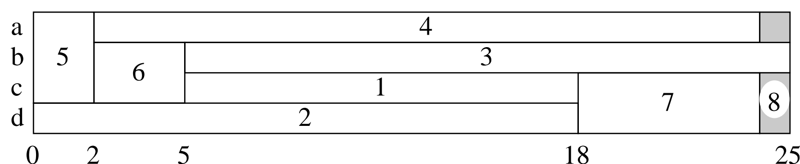

FIG. 1.4 – Allocation optimale et non contiguë.

Dans cette section nous allons montrer qu'il existe des instances pour lesquelles il n'est pas possible de trouver une représentation telle que chaque tâche soit représentée par un unique rectangle dans le diagramme de Gantt.

La figure 1.4 montre le diagramme de Gantt d'un ordonnancement optimal pour l'instance composée des huit tâches décrites dans le tableau 1.6 à ordonnancer sur quatre processeurs.

Tâche	1	2	3	4	5	6	7	8
1 proc.	13	18	20	22	6	6	12	3
2 proc.	13	18	20	22	3	3	6	1.5
3 proc.	13	18	20	22	2	3	6	1
4 proc.	13	18	20	22	2	3	6	1

TAB. 1.1 – Temps d'exécution des huit tâches de la figure 1.4

On peut facilement vérifier que ces tâches sont monotones (leur temps décroît et leur surface croît quand le nombre de processeurs augmente). La surface totale minimale est la somme des nombres de la première ligne (100). Cette valeur divisée par le nombre de processeurs disponibles (4) donne une borne inférieure au temps optimal de complétion des tâches (25). Ce temps est atteint avec l'ordonnancement présenté dans la figure 1.4, où la tâche 8 est allouée sur les processeurs a, c et d.

Nous allons maintenant démontrer qu'il n'y a pas pour cette instance d'ordonnancement contigu qui finisse en 25 unités de temps. On peut déjà constater que permuter les processeurs dans le diagramme de Gantt ne résout pas le problème.

Regardons d'abord quelles sont les allocations possibles pour les 8 tâches. Le profil des tâches 1 à 4 est tel que ces tâches doivent être effectuées sur un seul processeur. Elles sont également trop longues pour que deux d'entre elles soient exécutées l'une à la suite de l'autre sur un processeur. Numérotons donc les processeurs en fonction de la tâche séquentielle qui leur est allouée (processeur 1 pour la tâche 1 et ainsi de suite). Le temps libre restant dans

la limite des 25 unités de temps est de 12 unités de temps pour le processeur 1, 7 pour le processeur 2, 5 pour le 3 et 3 pour le 4.

La tâche 7 étant la plus grande de celles qu'il reste à allouer, c'est un bon point de départ pour une étude de cas.

- Si la tâche 7 est faite séquentiellement, la seule allocation possible fait qu'elle occupe tout le temps libre du processeur 1, ce qui laisse les processeurs 2, 3 et 4 avec respectivement 7, 5 et 3 unités de temps libre. La seule façon de faire 5 avec les temps d'exécution des tâches restantes est de mettre la tâche 5 sur trois processeurs et la tâche 6 sur deux, ce qui laisse pour les trois processeurs 2, 3 et 4 un temps libre de 2, 0 et 1. La huitième tâche ne peut donc pas être allouée correctement.
- Si la tâche 7 est faite sur plusieurs processeurs, son profil interdit d'en utiliser plus que deux (sinon le travail total augmenterait). Les seuls processeurs ayant assez de temps libre sont donc les processeurs 1 et 2, ce qui laisse après allocation de la tâche un temps libre de 6, 1, 5 et 3 unités de temps.

  La seule façon de remplir le processeur 2 est d'allouer la tâche 8 sur trois processeurs. Ce qui laisse trois possibilités pour le temps libre restant sur les quatre processeurs :
  - 5, 0, 4 et 3
  - 5, 0, 5 et 2
  - 6, 0, 4 et 2

  Avec seulement deux tâches, seul le cas 5, 0, 5 et 2 peut être parfaitement rempli. La tâche 5 étant alors allouée sur trois processeurs et la tâche 6 sur les processeurs 1 et 3.

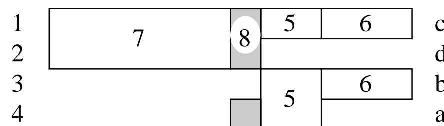

FIG. 1.5 – La seule allocation possible.

L'allocation du seul cas possible est décrit dans la figure 1.5. Cette figure représente juste l'allocation et pas les ordonnancements qui peuvent en résulter. Les tâches sont représentées sur les processeurs sur lesquels elles sont allouées. Les numéros sur la gauche de la figure sont les numéros des processeurs utilisés dans la discussion précédente. Les lettres à droite de la figure indiquent comment cette allocation correspond à l'ordonnancement de la figure 1.4. Comme nous l'avons dit précédemment, aucune permutation des processeurs ne peut rendre cette allocation contiguë. Il n'y a donc pas d'allocation contiguë qui soit optimale pour cette instance particulière.

Chapitre 2

Complexité

Dans ce chapitre, nous allons aborder une première série de problèmes d'ordonnancement pour les tâches malléables. Ces problèmes relativement simples permettent de se familiariser avec les tâches malléables. Dans tout le chapitre il s'agit d'ordonnancer des tâches malléables pour minimiser le temps total d'exécution (*makespan*), d'abord pour des tâches identiques avec des relations de précédence sous forme de chaînes, puis pour des tâches identiques avec des relations de précédence quelconques et enfin pour des chaînes de tâches quelconques.

2.1 Chaînes de tâches identiques par phases

L'algorithme que nous allons décrire pour les chaînes est dérivé de l'algorithme que l'on utilise dans le cas de tâches indépendantes [Dec00]. Cet algorithme utilise la programmation dynamique pour diminuer le coût et s'exécuter en temps polynomial. Dans toute la suite de ce chapitre, nous noterons m le nombre total de processeurs.

Dans le cas de tâches indépendantes, l'algorithme optimal est le suivant :

> On calcule l'ordonnancement optimal pour n tâches grâce aux ordonnancements que l'on connaît pour un nombre de tâches compris entre $n-1$ et $n-m$ (m étant le nombre maximum de tâches dans une phase, une par processeur). Dans chaque cas on ajoute une phase contenant le nombre nécessaire de tâches aux ordonnancements précédents, et l'on ne garde pour la solution que l'ordonnancement le plus intéressant. On itère ensuite ce procédé jusqu'à atteindre le nombre de tâches souhaité.

Dans le cas des chaînes, une tâche de chaque chaîne au plus peut être mise dans chaque phase, et donc chaque phase ne peut contenir plus de tâches qu'il n'y a de chaînes. Cela nous donne une limite sur le nombre d'ordonnancements précédents à considérer qui peut être plus petite que m. Il faut également garder pour chaque étape en mémoire quelles sont les longueurs de chaînes qui restent à ordonnancer.

Dans l'algorithme décrit ci-dessous, on notera S_i l'ensemble des phases utilisées pour l'ordonnancement de l'étape i (c'est-à-dire des i premières tâches considérées). Chaque phase dans S_i est représentée par le nombre de tâches qu'elle contient. Par exemple $S_4 = \{3, 1\}$ signifie qu'on ordonnance 4 tâches par une phase contenant trois tâches suivie d'une phase n'en contenant qu'une. En particulier on a toujours $S_0 = \emptyset$ et $S_1 = \{1\}$. S_2 est suivant les cas égal à $\{1, 1\}$ ou $\{2\}$, c'est-à-dire soit deux phases l'une à la suite de l'autre contenant une tâche chacune, soit une seule phase contenant deux tâches. On notera P_i l'ensemble de chaînes restant à ordonnancer après l'étape i. Chaque chaîne peut également être uniquement représentée par le nombre de tâches qui la composent. Ainsi, P_0 est l'ensemble initial de chaînes et P_1 est déduit du précédent en réduisant d'un le plus grand nombre (une tâche a été prise sur la chaîne la plus longue). $|P_i|$ est le nombre de chaînes à l'étape i. On note enfin par $|$ la concaténation d'ensembles, et $t(S_i)$ la somme des temps de chaque phase composant la suite de phases contenues dans S_i, c'est-à-dire, le temps total d'exécution des i tâches ordonnancées selon cette suite de phases. On pose par convention que si S_i n'est pas défini, $t(S_i)$ est égal à $+\infty$.

La fonction `Réduire`(P_k, i) utilisée ci-dessous enlève une tâche aux i plus grandes chaînes de P_k.

```
 Entrée : fonction t(), P_0
On initialise S_0 à ∅
Pour j = 1 à n faire
        S_j est indéfini
        Pour i = 1 à min(j,m) faire
                Si |P_{j-i}| ≥ i alors
                        Si t(S_j) > t(S_{j-i}|{i}) alors
                                S_j = S_{j-i}|{i}
                                P_j=Réduire(P_{j-i}, i)
Afficher S_n.
```

La concaténation du singleton $\{i\}$ à S_{j-i} dans l'algorithme correspond à la création d'une nouvelle phase contenant i tâches et rajoutée à la suite de phases S_{j-i}.

Il nous reste donc à prouver que le résultat de cet algorithme est bien un ordonnancement optimal pour le problème des chaînes de tâches identiques.

Théorème 1 *Le résultat de l'algorithme est un ordonnancement optimal pour le problème des chaînes de tâches identiques par phases.*

Preuve. La preuve se fait par induction sur le nombre de tâches de l'ordonnancement. Soit P^n un ensemble de chaînes ayant n tâches au total, et $S^*_{P^n}$ un ordonnancement optimal pour P^n tâches. Soit P^{n-j} l'ensemble de chaînes déduit du précédent en enlevant les tâches contenues par la dernière phase de $S^*_{P^n}$. Ces tâches sont nécessairement indépendantes et sans successeurs, ce sont les dernières de quelques chaînes.

On peut montrer que l'ordonnancement obtenu en enlevant la dernière phase $S^*_{P^n} - \{j\}$ est un ordonnancement équivalent à $S^*_{P^{n-j}}$. En effet, si l'optimal $S^*_{P^{n-j}}$ était plus petit, alors nous pourrions reconstruire un ordonnancement pour P^n plus petit que l'optimal $S^*_{P^n}$, ce qui est absurde.

Par hypothèse d'induction, notre algorithme calcule un ordonnancement optimal pour P^{n-j}. Sur l'entrée P^n, à la phase $n-j$, P^n_{n-j} contient j tâches finales, puisque ce sont les tâches qui ont la plus petite priorité. Avec quelques échanges simples on peut garantir que les j tâches restantes sont les mêmes que celles que l'on a retirées à $S^*_{P^n}$.

La dernière constatation à faire est que le résultat de l'algorithme à l'étape $n-j$ sur l'entrée P^n est identique au résultat rendu avec l'entrée P^{n-j}. Allonger certaines chaînes d'une tâche revient juste à leur donner une priorité plus forte par rapport aux chaînes de même longueur. Pour l'étape n, l'algorithme considère le résultat de l'étape $n-j$ plus une phase de taille j comme une des solutions possibles. Notre algorithme reconstruit donc bien un ordonnancement optimal pour P^n. □

On peut remarquer que les tâches étant identiques, nous pouvons changer l'ordre des phases à la fin de l'ordonnancement. Il existe donc toujours un ordonnancement pour lequel les phases sont triées par nombre croissant de tâches.

2.2 Exemple de déroulement de l'algorithme

Considérons une machine à 5 processeurs, et des tâches malléables à ordonnancer par phases ayant le profil suivant :

Nombre de processeurs	1	2	3	4	5
Temps d'exécution	6	3	2	2	2

P_0 étant composé d'une chaîne de trois tâches et une chaîne de deux tâches (donc $n = 5$). On note $P_0 = \{3, 2\}$

Le déroulement de l'algorithme est le suivant :
On commence par initialiser S_0 comme étant l'ensemble vide. On initialisera ensuite chaque S_i comme étant indéfini avant de les utiliser. Pour les premières itérations, on place d'abord une seule tâche $j = 1$ en une seule phase $S_1 = S_0|\{1\} = \{1\}$, puis deux tâches $j = 2$, d'abord en mettant les tâches en deux phases d'une tâche $S_2 = S_1|\{1\} = \{1, 1\}$, puis en prenant la meilleure option $(t(S_2) > t(S_0|\{2\}))$ de mettre les deux tâches en une seule phase $S_2 = S_0|\{2\} = \{2\}$.

L'étape suivante $j = 3$ est la première où le nombre de chaînes disponibles va limiter le nombre de tâches que l'on peut mettre par phase. On peut donc soit rajouter une phase d'une tâche à l'ordonnancement S_2 prévu pour deux tâches, soit rajouter une phase de deux tâches à l'ordonnancement S_1. Les deux ordonnancement ayant le même temps d'exécution, c'est le premier qui est choisi.

À l'étape $j = 4$ on essaie de mettre une phase d'une tâche sur S_3, puis on fait le meilleur choix de mettre une phase de deux sur S_2. Enfin pour la dernière étape, la meilleure solution est de mettre une phase d'une tâche sur S_4.

Sur la figure 2.1, on a représenté chaque étape de l'algorithme pendant laquelle S_j et P_j sont mis à jour. S_j est représenté par l'ordonnancement correspondant, et P_j par les chaînes, moins les tâches déjà ordonnancées entourées par des pointillés.

2.3 Graphe de tâches identiques par phases

Intéressons nous maintenant au cas plus général où le graphe de précédence est quelconque. Dans ce cas, si l'on a un algorithme pour ordonnancer de façon optimale les tâches identiques par phases, on peut utiliser cet algorithme dans le cas particulier de tâches séquentielles de durée unitaire. Notre algorithme résout donc également le problème de l'ordonnancement des graphes de tâches UET (P | prec, p_i=1 | C_{max}) qui est NP-difficile au sens fort [CK02].

Le problème d'ordonnancer des graphes de tâches identiques par phases est donc NP-difficile au sens fort.

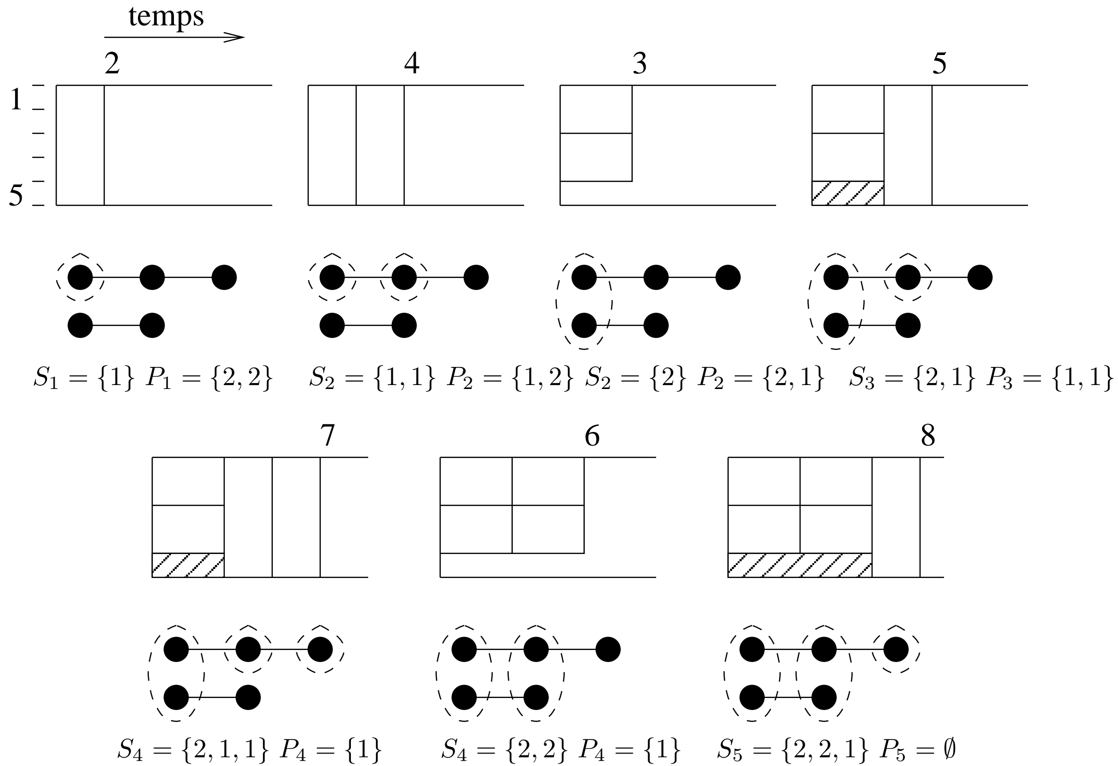

FIG. 2.1 – Toutes les étapes du déroulement de l'algorithme où S_j et P_j sont mis à jour.

2.4 Chaînes de tâches différentes

Si l'on s'intéresse maintenant au cas plus large de tâches différentes les unes des autres, le problème des chaînes devient NP-difficile au sens fort même quand le nombre total de processeurs m est fixé.

La preuve dérive d'une des preuves de Du et Leung dans [DL89]. La principale différence ici est que l'allocation des tâches est fixe dans le problème considéré par Du et Leung. Pour reprendre le même schéma de preuve, il faut considérer deux types de tâches. Le premier type est une chaîne de tâches tampons. Cette chaîne sert à créer des trous dans lesquels on peut placer de petites tâches correspondant à une instance de 3-PARTITION [GJ79]. Le deuxième type de tâches est formé par ces tâches qui traduisent un problème de 3-PARTITION.

Définition 6 *3-PARTITION*
Étant donnés z un entier, $(a_i)_{1 \leq i \leq 3z}$ un ensemble de $3z$ entiers de somme

totale zB, existe-t-il un découpage en z triplets tel que la somme des trois éléments de chaque triplet soit exactement B ?

Propriété 3 *3-PARTITION est NP-complet au sens fort.*

Revenons à nos tâches malléables. Soit A une tâche totalement malléable jusqu'à $m-1$ processeurs, et telle que $t_{m-1} = t_m = B$, et soit B une tâche totalement malléable de durée 1 sur m processeurs. Notre chaîne de tâches tampon est constituée de z tâches de type A et $z-1$ tâches de type B, chaque tâche de type B étant intercalée entre deux de type A. Le temps minimum d'exécution de la chaîne est donc de $zB + z - 1$. L'ordonnancement optimal représenté dans la figure 2.2 laisse z emplacements de taille B. Notons T_i les tâches séquentielles de temps d'exécution a_i. Trouver un ordonnancement pour l'ensemble de la chaîne et des tâches T_i revient à résoudre le problème 3-PARTITION.

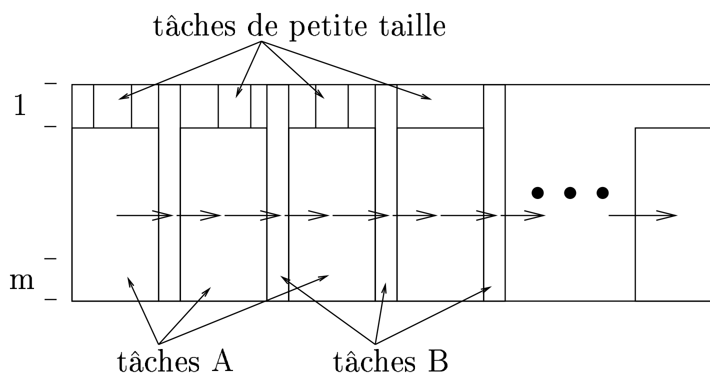

FIG. 2.2 – Ordonnancement optimal

Donc le problème d'ordonnancement de chaînes de tâches malléables différentes est NP-difficile au sens fort, même lorsque m est fixé supérieur ou égal à deux.

2.5 Conclusion

Nous nous sommes intéressés dans ce chapitre à l'ordonnancement sous le modèle des tâches malléables d'un problème frontière. D'une part nous avons montré qu'ordonnancer des chaînes de tâches identiques en ayant la contrainte des phases est un problème polynomial en fournissant un algorithme pour le résoudre, et d'autre part nous avons également montré que

prendre une structure de précédence quelconque, ou considérer des tâches différentes les unes des autres rendait tout de suite le problème beaucoup plus difficile. Les résultats peuvent être résumés dans le tableau suivant :

	chaîne	graphe quelconque
tâches identiques par phases	P	NP-difficile
tâches différentes	NP-difficile	NP-difficile

Nous avons donc cerné plus précisément la limite dans ce cadre entre les problèmes simples et les problèmes plus complexes. Reste à classer le problème d'ordonnancer des tâches malléables identiques en relaxant la contrainte de phase, puisque nous avons exhibé un exemple pour lequel l'ordonnancement par phases et l'ordonnancement sans phases était différents.

Chapitre 3

Ordonnancement Hiérarchique

Jusqu'à présent, nous avons étudié des problèmes sur une machine parallèle qui peut être soit une machine multiprocesseur, soit un réseau de machines monoprocesseurs identiques reliées entre elles par un réseau de communication. Une approche possible pour augmenter encore un peu plus la vitesse des plates-formes de calcul est d'utiliser plusieurs machines ayant chacune plusieurs processeurs. Cependant, les vitesses de communications entre deux processeurs d'une même machine ou entre deux machines sont très différentes. C'est ce problème de machines ayant deux niveaux de communications que nous allons étudier dans ce chapitre.

Bampis, Giroudeau et König ont beaucoup travaillé sur la version théorique du problème, où les tâches sont monoprocesseurs, ont un temps d'exécution unitaire et des communications unitaires entre multiprocesseurs mais nulle au sein d'une machine. Ils ont ainsi montré [BGK99b, Gir00] que pour des machines biprocesseurs, il est impossible[1] d'avoir un algorithme polynomial garanti à 5/4 de l'optimal. Dans un autre article [BGK99a], ils ont fourni un algorithme ayant une performance garantie de 8/5 pour le cas des biprocesseurs. La performance de cet algorithme tend vers 2 quand l'on augmente le nombre de processeurs au sein d'une même machine.

Il n'existe pas à ma connaissance de publications sur l'ordonnancement de tâches parallèles sur machine hiérarchique. Les recherches appliquées[2] actuelles se limitent aux problèmes niveau système comme la gestion des communications collectives [KdSF$^+$00] (comme par exemple le *broadcast*). Une des difficultés inhérentes au problème est que le modèle doit être adapté si l'on veut pouvoir utiliser plusieurs machines pour la même tâche. Sinon, en n'autorisant l'exécution d'une tâche qu'au sein d'un seul multiprocesseur, on revient à la superposition de deux problèmes : d'une part une partition

[1] À moins d'avoir $P = NP$.

[2] Par opposition aux recherches théoriques.

des tâches en autant de paquets qu'il y a de machines multiprocesseurs et d'autre part le problème classique d'ordonnancement sur chacun des multiprocesseurs.

3.1 Modifications du modèle de base

Le modèle des tâches malléables a été conçu pour séparer le problème de l'ordonnancement interne aux tâches du problème d'ordonnancement entre les différentes tâches. Ce qui fait la simplicité du modèle est le fait que sur la plupart des plates-formes de calcul parallèle les processeurs sont identiques. Pour un nombre de processeurs donné, le choix des processeurs ne change donc pas (ou de façon négligeable) le temps d'exécution de la tâche.

Sur une plate-forme de calcul ayant plusieurs niveaux de communications, ceci n'est malheureusement plus vrai. Le temps de communication entre deux processeurs partageant la même carte mère est jusqu'à mille fois plus rapide que le temps de communication entre deux processeurs distant [CSG99]. Il faut donc adapter les tâches malléables pour prendre en compte ce comportement de localité.

Plate-forme

L'ordonnancement sur plate-forme hiérarchique est déjà dur pour des tâches unitaires (s'exécutant sur un seul processeur), et des temps de communications unitaires inter-cluster, comme nous l'avons dit précédemment. Nous allons donc fournir une heuristique pour obtenir des ordonnancements garantis pour le problème de l'ordonnancement de tâches malléables sur une plate-forme hiérarchique.

Dans ce chapitre la plate-forme est composée de m multiprocesseurs[3], chacun de ces multiprocesseurs ayant k processeurs identiques. Ce travail est restreint aux cas où k est une puissance de deux, pour deux raisons : par souci de réalisme d'abord, puisque les machines multiprocesseurs sont généralement bi- ou quadri-processeurs, plus rarement avec huit processeurs (mais à ma connaissance rarement avec 47 processeurs), et également par souci de performance puisque la version à deux et quatre processeurs a la même garantie de performance de 3/2 que la version homogène, alors que la version avec un k quelconque sur laquelle nous avions commencé à réfléchir aurait eu une garantie de 2.

[3] On utilisera parfois l'abbréviation SMP qui vient de l'anglais «*Symmetric MultiProcessors*».

Dans un premier temps nous allons présenter la construction générale pour $k > 2$. Le cas particulier $k = 2$ fait l'objet d'une analyse séparée. Dans ce cas particulier, certaines propriétés générales ne sont pas conservées, mais le problème du placement est suffisamment simplifié pour rendre le problème facile.

Placement

Une variante des tâches malléables consiste à donner un temps d'exécution aux tâches qui dépend de l'ensemble de processeurs alloué. Cette variation, noté set_j dans la notation trois champs usuelle, est beaucoup plus complexe [DBBD95, DBS95]. Pour simplifier notre problème, nous adoptons une règle de dominance.

Définition 7 *(Règle de placement idéal)*
Pour un nombre donné de processeurs, nous dirons qu'une tâche est placée de façon idéale quand son temps d'exécution est minimal pour ce nombre de processeurs.

Intuitivement cette règle semble dire que dans l'optimal, toutes les tâches seront nécessairement placées de façon idéale. Malheureusement il n'en est rien, une tâche parallèle ayant peu de communications aura une faible pénalité, et peut se retrouver à boucher les trous dans l'ordonnancement optimal.

Cette règle est inspirée de l'expérience qui tend à montrer que plus on regroupe les processeurs effectuant la même tâche, plus l'exécution est rapide. Pour une tâche s'exécutant sur moins de $k + 1$ processeurs, le placement le moins coûteux sera certainement de prendre tous les processeurs sur le même multiprocesseur.

Pour étendre cette règle aux tâches utilisant plus de k processeurs, nous allons donc tenter de minimiser le nombre de multiprocesseurs impliqués dans le calcul de la tâche.

Hypothèse 1 *(Placement de pénalité minimale)*
Dans la suite du chapitre, nous allons supposer qu'un des placements idéaux pour une tâche T_i allouée sur $a_i k + b_i$ processeurs (avec $a_i \in [0; m]$ et $b_i \in [0; k-1]$) est d'avoir exactement a_i multiprocesseurs lui sont dédiés et les b_i processeurs restant sont situés sur le même multiprocesseur.

Cette hypothèse ne force pas la contiguïté des processeurs. En effet les b_i processeurs peuvent être choisis n'importe comment sur leur multiprocesseur, et les a_i multiprocesseurs dédiés ne sont pas forcément contigus. Dans la figure 3.1 deux tâches vérifiant l'hypothèse de placement minimal sont présentées, la troisième tâche étant mal placée puisqu'elle est à cheval sur deux

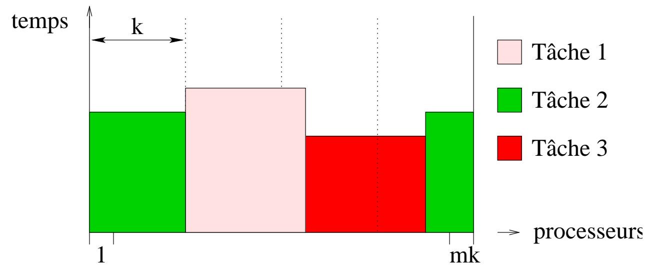

FIG. 3.1 – Les tâches 1 et 2 sont placées idéalement, alors que la tâche 3 ne l'est pas ($m = 4$).

multiprocesseurs. Exceptionnellement dans ce chapitre toutes les figures seront représentées avec les processeurs en abscisse et le temps en ordonnée. En effet dans les figures il sera plus important ici de détailler la disposition suivant les processeurs plutôt que le déroulement dans le temps.

L'ordonnancement que nous allons construire vérifiera cette propriété de placement de pénalité minimale. Cependant la garantie de performance compare le résultat de l'algorithme à un optimal où les tâches peuvent être placées n'importe comment. La seule hypothèse vraiment contraignante ici est que pour un nombre de processeurs donné, ces tâches soient au moins aussi rapides avec ce placement qu'avec un placement différent.

Propriétés

Les tâches étant monotones, les deux propriétés définies dans le chapitre 1 restent valides. Nous les redonnons ici, en introduisant la notation propre au chapitre, c'est-à-dire qu'il n'y a plus m machines homogènes, mais m clusters de k machines.

Propriété 4

$$t_i(p_t(i)) > \frac{p_t(i) - 1}{p_t(i)} t$$

Cette propriété est celle de l'allocation minimale (voir propriété 1).

Propriété 5 *Si le makespan optimal est plus petit que t, alors pour toute fonction d'allocation p telle que pour tout i on ait $p(i) \leq p_t(i)$, on a*

$$\sum_{i=1}^{n} p(i) t_i(p(i)) \leq m \times k \times t$$

Cette propriété est directement dérivée de la propriété 2.

Dans la suite de ce chapitre, nous allons considérer (sans perte de généralité) que la valeur t utilisée pour l'approximation duale est 1.

3.2 Structure de l'ordonnancement

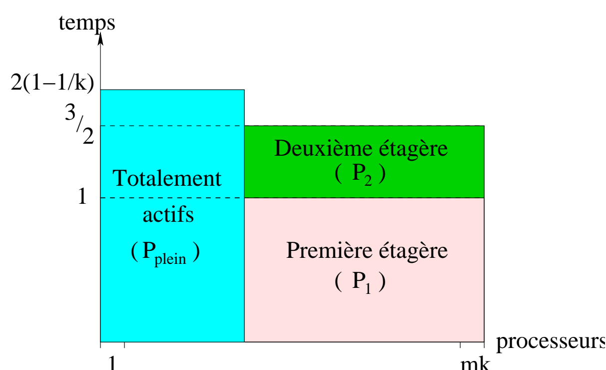

FIG. 3.2 – Structure générale de l'ordonnancement.

L'ordonnancement que l'on va construire est basé sur un découpage du diagramme de Gantt temps/processeurs qui est représenté dans la figure 3.2. Les tâches seront réparties en trois groupes appelés P_1, P_2 et P_{plein}, qui seront respectivement associés aux parties «Première étagère», «Deuxième étagère» et «Totalement actifs». L'algorithme d'ordonnancement est construit en deux phases.

1. Dans la première phase, nous commençons par séparer les tâches dans les deux ensembles P_1 et P_2 par programmation dynamique, comme dans l'article [MRT99], Puis nous allouons à chaque tâche de P_1 le nombre de processeurs correspondant à l'allocation minimale de temps 1, et à chaque tâche de P_2 l'allocation minimale de temps $1/2$. La troisième partie est vide pour le moment.

   Ensuite l'allocation de quelques tâches est réduite en suivant quelques règles de réduction, jusqu'à l'obtention d'un ordonnancement faisable (l'ordonnancement n'est pas fait à ce niveau, seuls certains critères sont vérifiés).

2. La deuxième phase est l'ordonnancement des tâches. Cet ordonnancement est finalement une version légèrement modifié du «*Strip packing*», c'est-à-dire du pavage d'un rectangle par des petits rectangles.

Les transformations de l'étape précédente nous donnent des propriétés sur l'allocation des tâches qui rendent ce problème simple.

Les propriétés de l'allocation et de l'ordonnancement permettent de garantir une exécution en temps $2(1 - 1/k)$. La structure étant celle présentée dans la figure 3.2.

Cet algorithme est en fait très largement inspiré d'un algorithme conçu pour le cas homogène, qui n'est pas encore paru à ce jour [MRT01].

3.2.1 Première partition des tâches

La première partition des tâches en deux ensembles se fait par programmation dynamique, en résolvant le problème de sac à dos suivant :

$$W^* = \min_{P_1, P_2} \left[\sum_{i \in P_1} W(i, p_1(i)) + \sum_{i \in P_2} W\left(i, p_{1/2}(i)\right) \right]$$

Où $p_t(i)$ est l'allocation minimale de temps t définie dans le chapitre 1.3.

Cette équation signifie que l'on souhaite minimiser le travail total des tâches quand on leur alloue le nombre minimal de processeurs pour qu'elles s'exécutent en temps 1 pour celles de la première étagère et en temps $1/2$ pour celles de la seconde.

On ajoute la contrainte qu'au plus mk processeurs sont utilisés sur la première étagère. Sans cette contrainte la meilleure solution serait de mettre toutes les tâches dans la première étagère.

$$\sum_{i \in P_1} p_1(i) \leq mk$$

Les équations de programmation dynamique qui permettent de résoudre ce système sont les suivantes :

$$\bar{W}(i, f) = min \begin{pmatrix} \bar{W}(i-1, f) + W\left(i, p_{1/2}(i)\right), \\ \bar{W}(i-1, f - p_1(i)) + W(i, p_1(i)) \end{pmatrix}$$

Avec i l'indice de la tâche et f le nombre de processeurs libres sur la première étagère. Comme ce problème est un problème en nombres entiers, la complexité est de $O(nmk)$ où n est le nombre de tâches.

L'analyse sur laquelle se base cette construction est la suivante. Même dans la solution optimale, il ne peut y avoir plus de m processeurs occupés pour plus de $1/2$ unité de temps. Les tâches qui durent plus de $1/2$ unité de temps sont parmi celles qui sont en train de s'exécuter à $t = 1/2$. Il y a

donc une partition naturelle dans l'optimal entre les tâches «longues» et les tâches «rapides». Cette distinction ne se fait pas sur le travail associé à une tâche quand elle s'exécute sur un processeur mais est plutôt liée au degré de parallélisme de la tâche.

La partition que l'on obtient avec le sac-à-dos n'est généralement pas celle de l'optimal, mais comme c'est la séparation qui a le plus petit travail total, le travail W^* est plus petit que le travail de l'optimal W^{opt}.

Le *makespan* de l'optimal étant de 1, la surface totale de l'optimal dans le diagramme de Gantt est bornée par le produit du nombre des processeurs par le temps c'est-à-dire $W^{opt} \leq mk$.

Donc si la valeur W^* obtenue par programmation dynamique est supérieure à cette borne mk, il y a une contradiction avec le fait que W^* est la plus petite surface possible. C'est donc dès le résultat de la programmation dynamique que l'on peut déterminer si la valeur de l'optimal est plus grande que celle que l'on a choisie dans le cadre de l'approximation duale.

Dans le cas contraire, les étapes suivantes finiront toujours par fournir un ordonnancement faisable en temps $2(1 - 1/k)$.

3.2.2 Réduction de l'allocation

Le problème d'ordonnancer un ensemble quelconque de tâches sur un cluster sans être en contradiction avec la règle de placement minimal étant complexe, nous allons faire une première réduction de l'allocation des tâches.

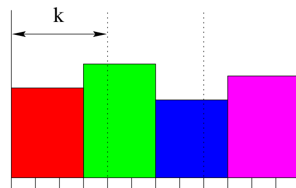

FIG. 3.3 – Exemple avec 4 tâches $m = 3$, $k = 4$.

Les propriétés de monotonie garantissent que la réduction de l'allocation n'augmente pas le travail d'une tâche. C'est pourquoi toutes les transformations que nous allons effectuer sur les tâches réduisent l'allocation. Nous conservons ainsi la propriété que le travail total est inférieur à celui que fait l'optimal.

Dans l'exemple de la figure 3.3 il suffit de réduire l'allocation d'une des tâches à un processeur pour avoir un ordonnancement qui satisfasse la propriété de placement. Cependant, cette solution n'est pas la meilleure du point

de vue du *makespan*, et est difficilement généralisable. Dans la figure 3.4 deux solutions différentes sont représentées. À gauche toutes les tâches ont été réallouées sur deux processeurs, et un multiprocesseur est laissé libre. À droite, les deux tâches les plus petites sont superposées. Dans la première solution c'est donc l'allocation qui est modifiée pour obtenir un ordonnancement simple alors que dans la seconde l'allocation est conservée mais l'ordonnancement se complexifie. C'est la première solution que nous avons généralisée dans notre algorithme.

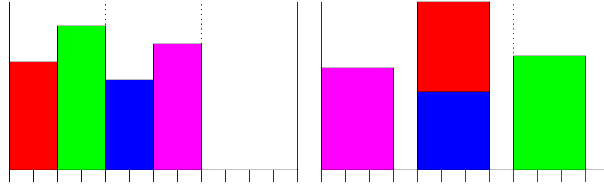

FIG. 3.4 – Réarrangement des tâches de la figure 3.3.

La généralisation se fait de la façon suivante :

Pour toutes les tâches T_i du premier ensemble de tâches P_1, l'allocation après la programmation est $p_1(i)$. Soit a_i et b_i tels que $p_1(i) = a_i k + b_i$ avec b_i plus petit que k. Si b_i est non nul, soit j_i le plus grand entier tel que $2^{j_i} \leq b_i$. La nouvelle allocation de la tâche T_i est alors de $a_i k + 2^{j_i}$ processeurs. Si b_i est nul, l'allocation reste la même que précédemment. Par exemple si $p_1(i) = 15$ et que k vaut 8, la tâche T_i est réallouée sur $12 = 8 + 4$ processeurs. Avec la même tâche, si k vaut 4 l'allocation devient $3 \times 4 + 2 = 14$ processeurs.

Dans la suite, on notera l'allocation $a_i k + 2^{j_i}$ sous la forme $a_i k + b_i^{\downarrow}$ si 2^{j_i} est la puissance de deux immédiatement inférieure à b_i. Par convention quand b_i est égal à 2^{j_i} ou que b_i est nul on a $b_i = b_i^{\downarrow}$.

Toutes les tâches dont l'allocation a été strictement réduite sont déplacées de P_1 dans P_{plein}. Nous pouvons maintenant prouver deux lemmes intéressants sur ces tâches.

Lemme 1 *Les tâches réduites s'exécutent en un temps inférieur ou égal à* $2(1 - {}^1\!/_k)$.

Preuve. Les hypothèses de monotonie impliquent que le travail décroît quand on réduit l'allocation. Nous avons donc quand r est plus petit que s :

$$t_i(r)r \leq t_i(s)s$$

L'allocation minimale de temps 1 étant $a_i k + b_i$, nous avons également $t_i(a_i k + b_i) \leq 1$ pour toute tâche T_i dans P_1.

En remplaçant ici s et r par les allocations avant et après réduction, on obtient l'inégalité suivante :

$$t_i(a_ik + b_i^{\downarrow})(a_ik + b_i^{\downarrow}) \leq a_ik + b_i$$

Ce qui amène au résultat final après quelques majorations simples :

$$\begin{aligned}
t_i(a_ik + b_i^{\downarrow}) &\leq \frac{a_ik + b_i}{a_ik + b_i^{\downarrow}} \leq \frac{b_i}{b_i^{\downarrow}} \\
t_i(a_ik + b_i^{\downarrow}) &\leq \frac{2b_i^{\downarrow} - 1}{b_i^{\downarrow}} \\
t_i(a_ik + b_i^{\downarrow}) &\leq \frac{k-1}{k/2} = 2(1 - \frac{1}{k})
\end{aligned}$$

□

Lemme 2 *Toutes les tâches de l'ensemble P_{plein} s'exécutent en un temps supérieur à 1.*

Preuve. Toutes les tâches de l'ensemble P_{plein} ont eu leur allocation réduite par rapport à l'allocation minimale de temps inférieur à 1. Leur temps d'exécution est donc strictement plus grand que 1. □

3.3 Principe régissant l'ordonnancement

Avant d'ordonnancer définitivement les tâches, il peut être nécessaire d'effectuer encore quelques transformations sur l'allocation des tâches. Avant de voir pourquoi de telles transformations sont parfois nécessaires et quelles sont ces transformations, nous allons présenter le schéma général de l'ordonnancement.

3.3.1 Ordonnancement de P_{plein}

Nous souhaitons d'abord ordonnancer les tâches de P_{plein} à gauche du diagramme de Gantt en utilisant le moins de processeurs possible (c'est-à-dire sans laisser de processeurs inutilisés) tout en respectant la contrainte de placement définie précédemment.

L'allocation d'une tâche T_i dans P_{plein} étant de la forme $p(i) = a_ik + b_i^{\downarrow}$, il est raisonnable de vouloir ordonnancer les deux parties de la tâche (quotient a_i et reste $b_i^{\downarrow}$) séparément. La première partie correspond à a_i multiprocesseurs dédiés à la tâche. La seconde partie doit être ordonnancée entièrement

sur un seul multiprocesseur. Pour s'assurer que les restes de toutes les tâches seront tous ordonnancés correctement, nous allons nous servir du fait que ce sont tous des puissances de 2, ainsi que k. Nous trions donc tous les restes par ordre décroissant et les allouons de gauche à droite sur les multiprocesseurs tel que présenté dans la figure 3.5.

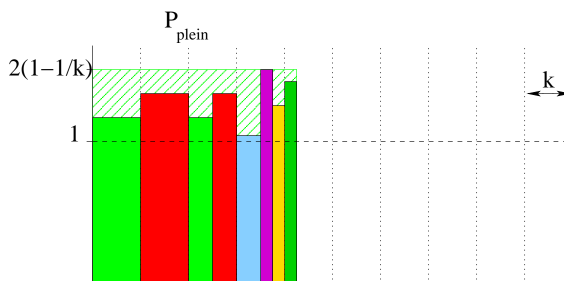

FIG. 3.5 – Résultat de l'ordonnancement de P_{plein}.

Lemme 3 *Cet ordonnancement respecte le critère de placement de pénalité minimale.*

Preuve. Il faut ici prouver que les restes des tâches s'exécutent bien à l'intérieur d'un seul multiprocesseur, et ce pour toutes les tâches de P_{plein}. Cela revient à prouver que les processeurs kl et $kl+1$ ne sont pas alloués au même reste de tâche.

Pour un reste de taille 2^j, toutes les parties de tâches ordonnancées à sa gauche sont soient sur un (ou plusieurs) multiprocesseur, soient sur un multiple de 2^j processeurs puisque les restes sont ordonnancés par taille décroissante. Il existe donc un entier l' tel que ce reste soit ordonnancé sur les processeurs $2^j l'+1$ à $2^j(l'+1)$. Soit d_j l'entier tel que $k = 2^j d_j$. Si $kl = 2^j d_j l \geq 2^j l'+1$ alors $d_j l > l'$ et donc $d_j l \geq l'+1$ et $kl+1 = 2^j d_j l + 1 \geq 2^j(l'+1)+1$. Ceci étant vrai pour tout l, le reste est dans un seul multiprocesseur. □

3.3.2 Ordonnancement de P_1

L'ordonnancement de P_1 se fait de façon très similaire, la seule différence notable étant que l'on remplit le diagramme de Gantt de la droite vers la gauche. Toutes les tâches sont allouées sur $a_i k + b_i^{\downarrow}$ processeurs et le même schéma peut être utilisé.

Lemme 4 *Le nombre de processeurs utilisés par les deux ensembles P_1 et P_{plein} est inférieur à mk.*

Preuve. Les tâches de P_{plein} étant des tâches de P_1 dont l'allocation a été réduite, la contrainte sur le nombre de processeurs de P_1 définie lors du partitionnement des tâches garantit que le nombre de processeurs utilisés par les deux ensembles est inférieur à mk. □

Ce lemme permet de constater qu'il n'y a pas de recouvrement entre les tâches de P_{plein} ordonnancées de gauche à droite et celle de P_1 ordonnancées de droite à gauche. À cette étape, l'ordonnancement ressemble à celui de la figure 3.6.

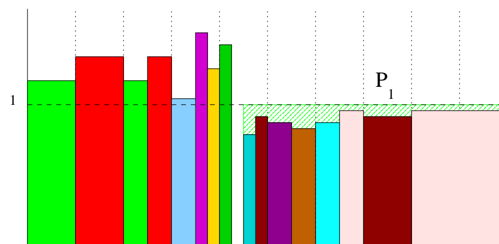

FIG. 3.6 – Ordonnancement de P_{plein} (gauche) et de P_1 (droite). Quelques processeurs peuvent être inutilisés.

3.3.3 Ordonnancement de P_2

Pour construire une solution bornée par $2(1 - 1/k)$, nous ne pouvons pas nous permettre de réduire l'allocation de toutes les tâches de P_2 pour utiliser la même technique d'ordonnancement. Cependant, nous pouvons laisser quelques processeurs inutilisés et garder un ordonnancement faisable. L'espace «en trop» entre 1 et $2(1 - 1/k)$ compense la perte que l'on va s'autoriser. Pour toute tâche T_i de P_2, allouée précédemment sur $a_i k + b_i$, on réserve des *boîtes* ayant $a_i k + 2^{j_i+1}$ processeurs, j_i étant défini comme précédemment. On notera $b_i^{\uparrow}$ cet arrondi à la puissance de 2 supérieure, avec pour convention $b_i^{\uparrow} = b_i$ quand $b_i^{\downarrow} = b_i$.

Notez la différence entre «réserver» et «allouer». Allouer plus de processeurs risquerait de faire augmenter le travail et donc de fausser tous les raisonnements sur le travail. Pour les raisonnements théoriques qui suivent les tâches restent donc allouées sur $a_i k + b_i$ processeurs, les autres restant inutilisés. Pour les implantations de l'algorithme, les processeurs réservés peuvent être tous utilisés pour accélérer l'exécution.

Les boîtes ayant des nombres de processeurs faciles à ranger, la technique précédente sera à nouveau utilisée. Un exemple complet est présenté dans la figure 3.7.

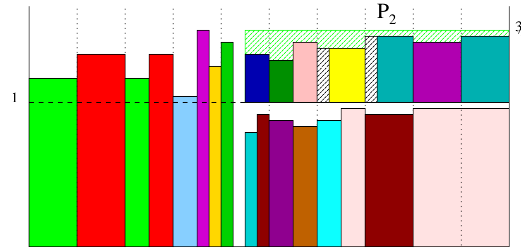

FIG. 3.7 – Ordonnancement de P_2. Les aires hachurées correspondent aux boîtes englobant des tâches allouées à $\frac{3}{4}k$ processeurs.

A cette étape, l'ordonnancement n'est pas forcément réalisable, puisque rien ne prouve que les tâches de la deuxième étagère n'empiètent pas sur celle de P_{plein}. Pour parer à ce problème nous allons procéder à quelques transformations dans la section 3.4.

3.3.4 Propriétés des étagères

Avant de continuer la description de l'algorithme, quelques propriétés utiles pour la démonstration de la garantie de l'algorithme peuvent déjà être présentées. L'analyse de la garantie de l'algorithme se basant principalement sur la quantité de travail, examinons d'abord ce que l'on peut en dire.

Dans la suite nous noterons s_1, s_2 et s_{plein} le nombre de processeurs utilisés respectivement par P_1, P_2 et P_{plein}. Après la première phase d'allocation nous avons : $s_1 + s_{plein} \leq mk$. Toutes les transformations que nous utiliserons garderont cette propriété vraie. Le lemme 2 montre que le travail de P_{plein} est plus grand que s_{plein}.

Comme pour le nombre de processeurs utilisés, nous noterons n_1, n_2 et n_{plein} le nombre de tâches de chaque ensemble.

Lemme 5 *Le travail de P_1 est au moins $s_1/2$.*

Preuve. Après le premier partitionnement, toutes les tâches de P_1 s'exécutent en plus de $1/2$ unité de temps. En effet dans le cas contraire, elles seraient dans l'ensemble P_2. Donc le travail de l'ensemble P_1 est au moins $s_1/2$. Les transformations que l'on propose dans la section suivante améliorent cette occupation moyenne à $3/4$ unité de temps.

Si toutes les tâches séquentielles de P_2 sont mises à l'écart pour être insérées à la fin, on peut montrer que l'occupation moyenne des tâches de P_2 est de $1/4$.

Lemme 6 *Une tâche T_i allouée en partie dans une boîte de taille 2^j a une aire au moins égale à la moitié de l'aire représentée par les processeurs réservés par la tâche pendant une demi-unité de temps. C'est-à-dire qu'elle utilise en moyenne les processeurs qui lui sont réservés pour au moins 0.25 unité de temps.*

Preuve. Soit T_i une tâche allouée en partie dans une boîte de taille 2^j. L'allocation minimale de temps $1/2$ de cette tâche est $a_ik + b_i$ avec $b_i > 2^{j-1}$. C'est par définition de l'allocation minimale la plus petite allocation telle que la tâche s'exécute en moins de $1/2$. Son travail est donc par monotonie supérieur à $1/2(a_ik + b_i - 1)$.

$$W_i > \frac{1}{2}(a_ik + b_i - 1) \geq \frac{1}{2}(a_ik + 2^{j-1}) \geq \frac{1}{4}(a_ik + 2^j)$$

Nous verrons dans la suite une version légèrement différente de ce lemme, où l'on garantit dans certaines configurations que l'utilisation moyenne des processeurs est au moins de $3/8$ unité de temps.

Les tâches séquentielles de P_2 seront insérées à la fin sur les processeurs ayant une charge totale inférieure à une unité de temps. Nous verrons que tant qu'il y a des tâches à insérer il reste des processeurs ayant une charge inférieure à 1, et que les tâches peuvent être insérées sans détériorer la qualité de l'ordonnancement.

On considère donc dans la majeure partie de la preuve qu'il n'y a pas de petites tâches dans l'ensemble P_2. Nous reviendrons à la fin de l'algorithme sur leur insertion. Toutefois il faut raisonner sur la quantité totale de travail. Ces tâches n'étant pas ordonnancées comme les autres de P_2, leur quantité de travail n'est pas compté dans W_2. Cette quantité est ajoutée à W_{plein}.

3.4 Transformations

Dans cette section, nous allons détailler les trois transformations qui permettent de déplacer une tâche d'un ensemble (P_2 ou P_1) vers un autre ensemble (P_1 ou P_{plein}). Ces transformations vont permettre de réduire le dépassement de l'ensemble P_2 en vue d'obtenir un ordonnancement faisable. Leur principale caractéristique est qu'elles n'augmentent pas la quantité de travail effectué. Elles peuvent être faites dans n'importe quel ordre. Pour que l'ordonnancement soit faisable il suffit que le nombre de processeurs utilisés par P_{plein} plus le nombre de processeurs utilisés par P_2 soit inférieur au nombre de processeurs disponibles mk. Nous verrons plus tard le détail du placement.

3.4.1 De P_2 vers P_1 ou P_{plein}

Soit f le nombre de processeurs inutilisés entre P_1 et P_{plein} sur la première étagère ($f = mk-(s_1+s_{plein})$). Soit T_i une tâche de P_2 pour laquelle $p_1(i)$ est égal à a_ik+b_i avec $a_ik+b_i^{\downarrow} \leq f$ (rappelons que $b_i^{\downarrow}$ est la plus grande puissance de 2 telle que $b_i^{\downarrow} \leq b_i$). On peut alors déplacer la tâche T_i de l'étagère P_2 à l'étagère P_1 ou P_{plein} suivant son temps d'exécution sur $a_ik + b_i^{\downarrow}$ processeurs, et toujours vérifier les propriétés démontrées dans les lemmes 1, 2, 4 et 5.

Dans deux cas particuliers, cette transformation doit être adaptée. Ces deux cas correspondent en fait aux cas où les autres transformations ne sont pas possibles et il ne reste qu'une ou deux tâches dans P_2. On verra que si l'ordonnancement n'est toujours pas faisable et qu'il ne reste qu'une ou deux tâches dans P_2, il peut arriver qu'elles soient trop grandes pour avoir leur allocation réduite comme décrit ci-dessus sans pour autant dépasser la limite $2(1-1/2^k)$ fixée. Dans ces deux cas, la transformation va être appliquée à une tâche de P_2 sans pour autant réduire son allocation $p1(i)$ à une allocation de type $a_ik + 2^{j_i}$. Il faut donc trouver une façon de placer la tâche sur tous les processeurs libres tout en respectant la contrainte de placement.

Quand il ne reste qu'une tâche à descendre de P_2 vers un des deux autres ensembles, il suffit de mixer les autres tâches de P_1 et P_{plein} comme sur la figure 3.8. Les processeurs libres sont alors tous regroupés à droite du diagramme de Gantt, et la dernière tâche peut tous les utiliser et être bien placée avec une allocation $a_ik + b_i$.

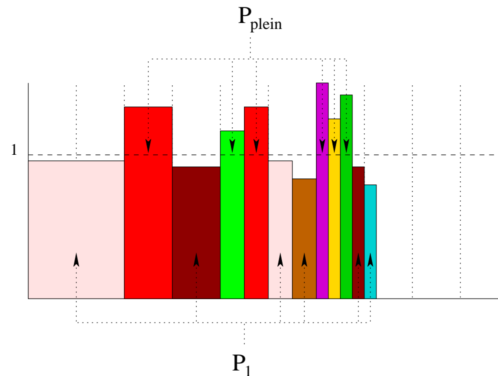

FIG. 3.8 – Les processeurs inutilisés sont dans une meilleure configuration après le mélange des deux ensembles P_1 et P_{plein}.

Quand il reste deux tâches, on montrera qu'il suffit d'en descendre une pour rendre l'ordonnancement faisable. Puisqu'il reste alors une tâche dans l'ensemble P_2, le mélange présenté ci-dessus n'est pas possible. Heureusement

nous pouvons montrer que peu de processeurs sont nécessaires pour la tâche restant dans P_2. Il suffit de mélanger les portions de tâches qui sont sur les deux SMPs partiellement utilisés comme indiqué dans la figure 3.9. Nous montrerons dans l'analyse que l'espace dégagé pour la dernière tâche de la deuxième étagère est suffisant.

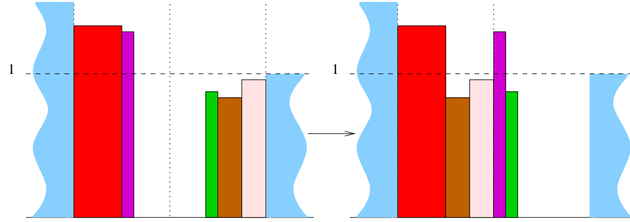

FIG. 3.9 – Seules quelques portions de tâches sont mélangées.

Après cette transformation, nous pouvons montrer la version plus forte du lemme 6 qui nous servira dans l'analyse :

Lemme 7 *Quand cette transformation n'est pas possible et qu'un des processeurs est libre sur la première étagère, on peut garantir que les tâches de* P_2 *utilisent en moyenne les processeurs qui leur sont réservés pendant* 3/8 *unité de temps (en réduisant le nombre de processeurs réservés si nécessaire).*

Preuve. Quand k est supérieur ou égal à 8, les tâches ne remplissant pas assez les boîtes qui leur ont été allouées peuvent voir leur allocation réduite sans dépasser la limite que l'on s'est fixé. En effet, si elles occupent les processeurs en moyenne pendant moins de 3/8 unité de temps, leur travail dans cette allocation $a_ik + b_i^{\uparrow}$ est inférieur à $3/8(a_ik + b_i^{\uparrow})$. L'allocation autorisée immédiatement inférieure n'ayant pas été choisie, il est par contre supérieur à $1/2(a_ik + b_i^{\downarrow})$ si $b_i^{\uparrow}$ est non nul, et supérieur à $a_ik/2$ si $b_i^{\uparrow}$ est nul. Dans tous les cas, réserver l'allocation autorisée immédiatement inférieure à celle qui était prévue mène à des temps d'exécution inférieurs à 3/4 unité de temps. Avec des tâches commençant leur exécution au temps 1, on reste en dessous de la limite $2(1 - 1/k)$ pour $k \geq 8$.

Pour $k = 2$, le cas est traité à part. Il reste donc à considérer $k = 4$, pour lequel on ne peut pas réduire l'allocation des tâches. C'est là que l'hypothèse du processeur inutilisé est utile. Si aucune tâche de P_2 n'a pu être déplacée, et qu'il reste un processeur de libre entre P_1 et P_{plein}, cela signifie qu'il n'y a pas de tâches dans P_2 qui ait un travail inférieur à 3/2 (sinon elle aurait pu être descendue par la transformation précédente). Donc les tâches ont toutes une

allocation dans P_2 de plus de 6 processeurs. Or s'ils occupent moins de 3/8 unité de temps en moyenne, c'est que leur travail est inférieur à $3/8(a_ik+b_i^\uparrow)$. Si $b_i^\uparrow = b_i^\downarrow = b_i$, alors le travail de la tâche T_i est supérieur à $(a_ik + b_i - 1)$. Ce qui nous amène aux inégalités suivantes :

$$\frac{1}{2}(a_ik + b_i - 1) < W_i < \frac{3}{8}(a_ik + b_i)$$

$$a_ik + b_i < 4$$

D'où $a_i = 0$ et $b_i < 4$, ce qui est exclu puisqu'il reste un processeur libre sur la première étagère. Si $b_i^\uparrow > b_i > b_i^\downarrow$, alors on a $b_i^\uparrow = 2b_i^\downarrow$ par définition. Ce qui permet d'écrire :

$$\frac{1}{2}(a_ik + b_i^\downarrow) < W_i < \frac{3}{8}(a_ik + b_i^\uparrow)$$

$$a_ik < 2b_i^\downarrow$$

Puisque le processeur libre impose $a_i \geq 1$ et que k vaut 4, cela impose $b_i^\downarrow > 2$ donc $b_i^\downarrow = 4$, ce qui contredit $b_i > b_i^\downarrow$.

3.4.2 De P_1 vers P_{plein}

Pour appliquer la transformation précédente, il faut bien entendu qu'il y ait suffisamment de processeurs libres entre les deux ensembles P_1 et P_{plein}. Pour augmenter le nombre de processeurs libres, on peut réduire l'allocation d'une tâche de P_1. Il faut cependant faire attention à ne pas trop augmenter le temps d'exécution de la tâche pour ne pas dépasser la limite de $2(1-1/k)$ unités de temps.

Cette réduction ne peut donc être appliquée qu'à un petit sous-ensemble des tâches de P_1. Les tâches ayant un temps d'exécution de moins de $(1-1/k)$ peuvent avoir leur allocation réduite de moitié. Pour la preuve, nous aurons uniquement besoin de réduire l'allocation des tâches de P_1 ayant un temps d'exécution inférieur à 3/4.

La réduction en elle-même consiste à transformer une allocation $a_ik+2^{j_i}$ en $a_ik+2^{j_i-1}$ quand cela est possible. Cette réduction ne peut évidemment pas s'appliquer aux tâches de P_1 s'exécutant sur un seul processeur. Ces tâches seront traitées par la troisième transformation.

Lemme 8 *Toutes les tâches de P_1 allouées à plus d'un processeur ont un temps d'exécution supérieur à 3/4 quand cette transformation n'est plus possible.*

Preuve. Tout d'abord, regardons par monotonie quelle est l'allocation des tâches s'exécutant en 3/4 unité de temps ou moins.

L'allocation minimale nous garantit que $t_i(p_1(i)-1) > 1$. Par monotonie du travail nous avons :

$$p_1(i)t_i(p_1(i)) \geq (p_1(i)-1)t_i(p_1(i)-1)$$

D'où l'on peut conclure :

$$t_i(p_1(i)) > 1 - \frac{1}{p_1(i)}$$

Donc les tâches s'exécutant en 3/4 unité de temps ou moins sont allouées sur moins de 4 processeurs. Le cas $k = 2$ étant traité à part, les tâches dans P_1 ne peuvent être allouées sur 3 processeurs. Donc seules les tâches sur deux processeurs sont concernées. En devenant séquentielles, leur temps d'exécution est au plus doublé, elles finissent donc en 3/2 unités de temps dans P_{plein}. □

3.4.3 Superposition de deux tâches séquentielles de P_1 dans P_{plein}

Si deux tâches allouées à un seul processeur s'exécutent en un temps inférieur à $1 - 1/k$, on peut les effectuer sur un même processeur en moins de $2(1-1/k)$ unités de temps. Les tâches séquentielles de temps inférieur à 1/2 ayant été exclues précédemment, les nouvelles tâches formées par cette superposition ont un temps d'exécution supérieur à 1.

Lemme 9 *Le travail total effectué dans P_1 est supérieur à $\frac{3}{4}(s_1 - 1) + \frac{1}{2}$ quand cette transformation et la précédente ne sont plus possibles.*

Preuve. Quand la superposition n'est plus possible, il reste au plus une tâche séquentielle dans l'ensemble P_1. Cette tâche a un temps d'exécution strictement supérieur à 1/2. Tous les autres processeurs sont occupés par des tâches multiprocesseurs qui vérifient le lemme 8.

3.5 Analyse

À cette étape de l'algorithme tous les ordonnancements ou presque sont faisables. Nous allons commencer la preuve qui démontre leur faisabilité pour débusquer les quelques cas pathologiques qui nécessitent un traitement supplémentaire.

Supposons que nous sommes dans un de ces cas pathologiques. Aucune des trois transformations précédentes ne peut être appliquée, et l'ordonnancement n'est toujours pas faisable (c'est-à-dire qu'il y a toujours trop de tâches dans l'ensemble P_2 par rapport au nombre de processeurs disponibles). Nous allons raisonner sur la quantité de travail à faire et sur celle faite dans chaque partie. Voici donc d'abord quelques propriétés et rappel de propriétés sur le travail.

Nous avons déjà défini $f = mk - s_1 - s_{plein}$ le nombre de processeurs libres entre P_1 et P_{plein}. Le travail total est découpé en la somme des travaux de chaque partie :

$$W_{total} = W_{P_1} + W_{P_2} + W_{P_{plein}}$$

Propriété 6 $W_{total} < mk$ puisque 1 est le temps d'exécution optimal de l'instance (c'est l'hypothèse de l'approximation duale).

Propriété 7 $W_{P_{plein}} > s_{plein}$ puisque les tâches de P_{plein} durent toutes plus de 1 unité de temps (voir lemme 2).

Pour éviter d'écrire deux fois la preuve, suivant s'il y a ou non dans P_1 une dernière tâche séquentielle plus petite que 3/4, nous noterons x la variable booléenne associée. La valeur de x est 1 s'il y a une petite tâche dans P_1, et 0 sinon.

Propriété 8 $W_{P_1} > \frac{3}{4}s_1 - \frac{x}{4}$ Ceci est une réécriture plus générale du lemme 9.

Sachant que le travail total est la somme du travail des différentes parties, et connaissant les trois inégalités précédentes, on peut en tirer une majoration du travail restant à effectuer dans la partie P_2 de l'ordonnancement :

$$f + \frac{s_1 + x}{4} > W_{P_2} \tag{3.1}$$

Toutefois, si nous supposons que l'ordonnancement n'est pas faisable, cela signifie qu'avec le lemme 6 qui donne l'occupation moyenne des processeurs dans P_2, nous pouvons également minorer le travail restant à faire dans P_2 :

$$W_{P_2} > \frac{f + s_1 + 1}{4}$$

De ces deux inégalités concernant W_2, nous pouvons déduire qu'il y a au moins un processeur libre sur la première étagère. En effet, un rapide calcul nous donne :

$$\begin{aligned} f + \frac{s_1 + x}{4} &> \frac{f + s_1 + 1}{4} \\ \frac{3}{4}f &> \frac{1 - x}{4} \geq 0 \end{aligned}$$

Puisqu'il y a un processeur libre on peut utiliser le lemme 7 pour minorer le travail de P_2 par une borne plus grande que la précédente :

Propriété 9 $W_{P_2} > \frac{3(f+s_1+1)}{8}$

Et puisque l'ordonnancement n'est toujours pas faisable, c'est que la deuxième transformation n'est pas possible.

Propriété 10 $W_{P_2} > n_2 \frac{f+1}{2}\frac{3}{2}$

Preuve. Ceci signifie juste qu'aucune des n_2 tâches de P_2 ne pouvait être descendue sur la première étagère. S'il y en avait une avec un travail inférieur à $\frac{f+1}{2}\frac{3}{2}$, elle pourrait être allouée dans P_{plein} avec un temps d'exécution inférieur à 3/2. En effet s'il y a f processeurs libres, la plus grande allocation de forme autorisée $(a_i k + b_i^{\downarrow})$ est supérieure à $(f+1)/2$.

Plus généralement, nous avons vu dans le paragraphe 3.4.1 que s'il restait une seule tâche dans P_2, on pouvait utiliser tous les processeurs libres en mélangeant P_1 et P_{plein}. Donc dans tous les cas, on peut écrire $W_{P_2} > \frac{3}{2}f$.

Avec ces deux minorations de W_{P_2} et la majoration proposée dans l'équation 3.1, on peut déduire deux nouvelles inégalités intéressantes :

$$\begin{aligned} f + \frac{s_1 + x}{4} > \quad & W_{P_2} > \frac{3(f+s_1+1)}{8} \\ 8f + 2s_1 + 2x \quad > & \quad 3f + 3s_1 + 3 \\ 5f + 2x - 3 \quad > & \quad s_1 \end{aligned} \tag{3.2}$$

Et :

$$\begin{aligned} f + \frac{s_1 + x}{4} > \quad & W_{P_2} > n_2 \frac{f+1}{2}\frac{3}{2} \\ 4f + s_1 + x \quad > & \quad 3n_2(f+1) \\ s_1 + x \quad > & \quad 3n_2(f+1) - 4f \end{aligned} \tag{3.3}$$

En combinant les deux (3.2 et 3.3) , on obtient une borne sur le nombre de tâches dans P_2 quand l'ordonnancement n'est toujours pas faisable :

$$\begin{aligned} 5f + 3x - 3 > \quad & s_1 + x > 3n_2(f+1) - 4f \\ 3f + x - 1 \quad > & \quad n_2(f+1) \\ 3 \quad > & \quad n_2 \end{aligned} \tag{3.4}$$

Donc quand à ce stade l'ordonnancement n'est pas faisable, il ne peut rester qu'une ou deux tâches dans l'ensemble P_2. Il y en a forcément au moins

une, sinon l'ordonnancement serait faisable (le seul problème possible étant justement que les tâches de P_2 dépassent).

Nous allons donc maintenant examiner séparément le cas où il reste une tâche et le cas où il en reste deux.

Cas $n_2 = 1$

Tout d'abord, s'il reste une tâche dans P_2 et que l'ordonnancement n'est pas faisable, nous savons qu'elle a besoin d'occuper plus de processeurs que $mk - s_{plein} = s_1 + f$. Donc son travail est supérieur à $(s_1 + f)/2$. L'équation 3.1 nous donnant une majoration de W_{P_2}, on obtient le résultat suivant :

$$\begin{aligned} f + \frac{s_1 + x}{4} &> \frac{f + s_1}{2} \\ 2f + x &> s_1 \end{aligned} \tag{3.5}$$

La variante de la propriété 10 $W_{P_2} > \frac{3}{2}f$ et l'équation 3.1 nous donnent une inégalité similaire :

$$\begin{aligned} f + \frac{s_1 + x}{4} &> \frac{3f}{2} \\ s_1 + x &> 2f \end{aligned} \tag{3.6}$$

Les inégalités 3.5 et 3.6 étant strictes, x vaut nécessairement 1 et s_1 est égal à $2f$. Il y a donc une tâche séquentielle dans P_1 dont le temps d'exécution est compris (strictement) entre 1/2 et 3/4 qui n'a pas pu être superposée à une autre tâche séquentielle. Pour rendre l'ordonnancement faisable il suffit dans ce cas de superposer cette tâche à la deuxième plus petite tâche de P_1 et d'ordonnancer la tâche de P_2 sur les $f + 1$ processeurs libres.

Montrons d'abord que le placement de la séquentielle sur la plus petite tâche multiprocesseur de P_1 n'est pas en contradiction avec la borne $2(1-1/k)$ que l'on s'est fixée. Pour cela il faut commencer par montrer $W_{P_1} < \frac{3}{4}s_1$. Par l'absurde, si $W_{P_1} \geq \frac{3}{4}s_1$ on peut remplacer la propriété 8 par cette inéquation dans le calcul de l'équation 3.1, ce qui nous donne comme majoration du travail $f + s_1/4 \geq W_{P_2}$. Avec $s_1 = 2f$, ceci est en contradiction avec l'hypothèse que la tâche restant dans P_2 ne peut être ordonnancée sur f processeurs ($W_{P_2} > \frac{3}{2}f$).

Soit r_1 et r_2 deux réels strictement positifs définis tels que la dernière tâche séquentielle de temps inférieur à 3/4 dans P_1 ait pour temps d'exécution $1/2+r_1$ et que la deuxième plus petite tâche de P_1 ait pour temps d'exécution $3/4 + r_2$.

$$W_{P_1} \geq \frac{1}{2} + r_1 + (s_1 - 1)(\frac{3}{4} + r_2)$$

$$\begin{aligned} \frac{3}{4}s_1 &> W_{P_1} \\ \frac{1}{4} &> r_1 + (s_1 - 1)r_2 > r_1 + r_2 \end{aligned}$$

La superposition des deux tâches aura donc un temps d'exécution inférieur à 3/2.

Il reste donc à montrer qu'il existe une deuxième tâche dans P_1. S'il n'y en avait qu'une, l'équation 3.1 s'écrirait $f + \frac{1}{2} > W_{P_2}$, ce qui avec la variante de la propriété 10 nous donne :

$$f + \frac{1}{2} > W_{P_2} > \frac{3}{2}f$$

Soit $f = 0$, ce qui contredit l'existence d'au moins un processeur libre.

Pour finir, on peut montrer que la tâche de P_2 peut être ordonnée sur les $f + 1$ processeurs en un temps inférieur à 3/2. En effet avec $s_1 = 2f$ et l'équation 3.1 on a $\frac{3}{2}(f + 1) > f + \frac{s_1+1}{4} > W_{P_2}$. L'ordonnancement obtenu est similaire à celui présenté dans la figure 3.10[4].

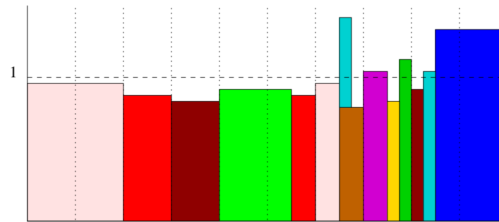

FIG. 3.10 – Aspect de l'ordonnancement après les dernières transformations.

Cas $n_2 = 2$

Dans ce cas il reste deux tâches dans P_2. Pour terminer l'ordonnancement, il faut ordonnancer une des deux sur une partie des f processeurs libres, et montrer que la disposition des tâches permet de placer la deuxième tâche sur la deuxième étagère en respectant le critère de placement.

Nous allons commencer par montrer par l'absurde que la plus grande tâche peut être allouée sur f processeurs sans que le temps d'exécution ne dépasse 3/2. Si la grande tâche a un travail supérieur à $\frac{3}{2}f$, sachant que la

[4]Je vous promets que cette figure correspond à un cas possible.

petite a un travail supérieur à $\frac{3}{4}(f+1)$ (voir propriété 10) on a les inégalités suivantes :

$$\begin{aligned} W_{P_2} &> \frac{3}{2}f + \frac{3}{4}(f+1) = \frac{9f+3}{4} \\ f + \frac{s_1+x}{4} &\geq W_{P_2} \\ 4f + s_1 + x &> 9f+3 \\ s_1 + x &> 5f+3 \end{aligned}$$

Ce qui contredit l'inéquation 3.2.

La plus grande tâche peut donc être placée sur les f processeurs libres. Étudions maintenant la disposition des processeurs occupés pendant moins de une unité de temps pour le placement de la dernière tâche de P_2.

1. Si la tâche que l'on vient d'ordonnancer sur les f processeurs libres a un temps d'exécution supérieur à 1, son travail est plus grand que f. Nous pouvons alors borner le travail de la petite tâche grâce à l'inéquation 3.1. Ce travail est inférieur à $\frac{s_1}{4}$. Cette tâche a donc besoin d'au plus $\frac{s_1}{2}$ processeurs pour être ordonnancée. Si s_1 est plus grand que k, il y a un (ou des) SMP(s) entièrement réservé(s) à des tâches de P_1 et au plus un partiellement utilisé par P_1 avant le réarrangement décrit dans la figure 3.9. Utiliser le (ou les) SMP(s) réservé(s) suffit pour placer la dernière tâche de P_2. Si s_1 est plus petit que k, lors du mélange décrit dans la figure 3.9 le SMP ayant le plus processeurs utilisés par des tâches de P_1 en a au moins $s_1/2$. Donc la deuxième tâche peut toujours être placée.
2. Si la tâche que l'on ordonnance sur les f processeurs dure moins de 1 unité de temps, les f processeurs qu'elle utilise peuvent également être utilisés pour la deuxième tâche. En majorant différemment le terme de gauche de l'inéquation 3.1, grâce à l'inéquation 3.3 utilisée avec $n_2 = 2$, on a :

$$\begin{aligned} f + \frac{s_1+x}{4} = \frac{f+s_1}{2} + \frac{2f-s_1+x}{4} &< \frac{f+s_1}{2} \\ W_{P_2} &< \frac{f+s_1}{2} \end{aligned}$$

   Donc la petite tâche a un travail inférieur à $\frac{f+s_1}{4}$, elle utilise au plus $\frac{f+s_1}{2}$ processeurs. Comme dans le cas précédent, il y a toujours assez de processeurs dans une bonne configuration après le mélange présenté dans la figure 3.9.

3.6 Résumé de l'algorithme

L'algorithme est donc constitué des étapes suivantes :

1. Faire une partition des tâches entre P_1 et P_2 par le sac à dos (par programmation dynamique).
   Complexité : $O(mnk)$
2. Si le poids total minimum est plus grand que mk alors l'optimal est plus grand que 1. Il faut réessayer avec une valeur plus grande.
3. On enlève les tâches séquentielles de P_2.
   Complexité : $O(n_2)$
4. On réduit les tâches de P_1 à leur allocation $a_ik + b_i^{\downarrow}$, en mettant dans P_{plein} toutes celles qui sont réellement réduites.
   Complexité : $O(n_1)$
5. Tant que l'ordonnancement n'est pas faisable et qu'une des trois transformations est possible, on fait une transformation.
   Complexité : $O(2n_2 + n_1)$
6. Si l'ordonnancement n'est toujours pas possible, on fait la transformation qui correspond au cas particulier $n_2 = 1$ ou $n_2 = 2$ suivant le nombre de tâches dans P_2.
   Complexité : $O(1)$
7. On place définitivement les tâches en insérant les petites tâches enlevées à l'étape 3.
   Complexité : $O(n)$ pour le placement $O(n_{petites}mk)$ pour l'insertion.[5]

Cette insertion se fait entre les tâches de la première et celles de la deuxième étagère. Puisque le travail total est inférieur à mk, il y a des processeurs qui font moins de 1 unité de travail. On peut ajouter à ces processeurs une petite tâche (inférieure à $1/2$) sans pour autant dépasser la charge maximale de $3/2$. Avant insertion, les processeurs travaillent sur au plus deux tâches. En plaçant la (ou les) tâche(s) insérée(s) entre ces deux tâches en décalant la deuxième tâche si nécessaire, on est sûr de ne pas terminer la dernière tâche après l'instant $3/2$.

La complexité de l'algorithme est de l'ordre de $O(mnk)$, la taille de l'instance étant du même ordre puisque chaque tâche a mk temps d'exécution en fonction du nombre de processeurs alloués.

Théorème 2 *L'algorithme d'ordonnancement présenté ici offre une garantie de performance de* $2(1 - 1/k)$ *pour le problème de l'ordonnancement de*

[5] $n_{petites}$ étant le nombre de petites tâches.

tâches malléables monotones indépendantes sur une machine composée de m multiprocesseurs identiques ayant chacun k processeurs, où k est une puissance de deux supérieure ou égale à 4.

Par construction la taille de la boîte contenant les tâches de l'ensemble P_{plein} est exactement $2(1-1/k)$, et l'on a montré que pour toutes les instances il était possible d'ordonnancer les tâches sans dépasser cette limite.

3.7 Cluster de biprocesseurs

Pour $k = 2$ la borne $2(1-1/k)$ vaut 1. Le meilleur algorithme connu pour le cas homogène étant de $3/2$ (voir [Mou00]) et le problème étant NP-complet au sens fort, nous allons nous contenter[6] ici de faire un ordonnancement ayant une garantie de $3/2$.

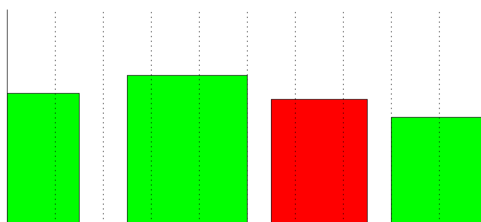

FIG. 3.11 – La tâche rouge[7] est la seule mal alignée.

L'idée dans le cas des biprocesseurs est de prendre le résultat de l'algorithme homogène et de changer légèrement le placement des tâches pour garantir le placement de pénalité minimale pour toutes. Le seul problème de placement que l'on puisse avoir, quand k vaut 2 et que les allocations sont contiguës comme dans le cas homogène, est d'avoir une tâche ayant un nombre de processeurs alloués pair qui ne soit pas alignés avec les SMPs (voir figure 3.11). Pour éviter d'avoir ce problème, il suffit de placer toutes les tâches ayant un nombre pair de processeurs sur les bords du diagramme de Gantt comme sur la figure 3.12. Ce qui revient simplement à mettre les tâches paires de P_{plein} sur la gauche et celle de P_1 et P_2 sur la droite.

Les lecteurs attentifs remarqueront que dans l'article original, une dernière transformation similaire à celle que nous avons présentée dans le cas $n_2 = 1$ peut parfois être nécessaire. S'il reste des tâches dans P_2, leur placement peut être perturbé par cette tâche de P_1 qui se termine entre $5/4$ et

[6] Si on avait réussi à faire mieux que $3/2$, le résultat ferait un chapitre de plus.
[7] Elle apparaît foncée sur les impressions noir et blanc.

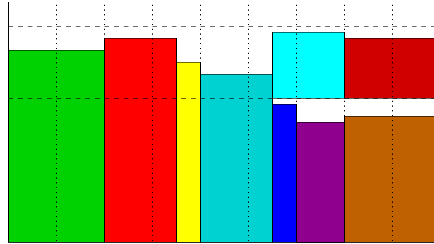

FIG. 3.12 – Les tâches sont placées sur des processeurs contigus.

3/2. Heureusement on peut toujours prouver que cette transformation n'est utile que lorsque n_2 vaut 1 comme précédemment.

3.8 Conclusion

L'algorithme d'approximation garanti présenté ici est à ma connaissance le premier à permettre d'utiliser une structure hiérarchique pour des tâches malléables. La garantie obtenue pour les petites valeurs de k étant la même que dans le cas homogène, il paraît difficile de faire mieux sans changer radicalement d'approche. Il serait plus intéressant de s'intéresser au cas où les SMPs sont de tailles différentes, ou dans un contexte hétérogène. Les modifications faites au modèle ne sont malheureusement plus suffisantes dans ce cas, et l'on atteint les limites du modèle des tâches malléables.

Chapitre 4

Ordonnancement Multi-critère

4.1 Rappel des notations

Comme dans les chapitres précédents, nous considérons n tâches à ordonnancer sur m processeurs. Le temps d'exécution de la tâche T_i allouée sur $p(i)$ processeurs est toujours noté $t_i(p(i))$ et la date à laquelle la tâche T_i commence à s'exécuter (sur tous les processeurs qui lui sont alloués) est toujours noté $\sigma(i)$. La date de fin d'une tâche est $C_i = \sigma(i) + t_i(p(i))$. Le travail est toujours noté $W_i(p(i))$.

4.2 Ordonnancement avec un critère unique

Le problème de l'ordonnancement tel qu'il est défini dans le chapitre 1, est la minimisation sous certaines contraintes (non recouvrement des tâches, nombre limité de processeurs, ...) d'une fonction objective. Une des fonctions les plus utilisées est le *makespan*. C'est d'ailleurs celle que l'on a utilisé dans les chapitres précédents. Cette fonction permet de minimiser le temps total d'occupation de la machine si l'on se place du point de vue d'un «locataire» qui loue une machine multiprocesseur pendant un certain temps. Elle permet aussi de minimiser le temps d'attente du résultat quand la machine est utilisée par un unique utilisateur qui est intéressé par le résultat global de tous ses calculs mais pas particulièrement par une sous partie.

Il y a cependant d'autres situations pour lesquelles ce critère est moins intéressant, voire contre productif. Par exemple, un utilisateur ayant programmé 100 tâches sur une machine peut vouloir obtenir rapidement les résultats des 10 premières pour vérifier qu'il ne s'est pas trompé, et pour pouvoir planifier les 100 tâches suivantes qu'il soumettra (en modifiant certains paramètres en fonction des premiers résultats par exemple). Dans ce

cas, il faut trouver une autre fonction d'estimation qui colle mieux à la réalité, c'est-à-dire qui satisfasse les utilisateurs. Il existe plusieurs critères qui ont été proposés dans la littérature, parmi lesquels nous avons choisi le makespan et la somme des temps de complétion pour leurs caractères antagonistes et leur fréquente utilisation.

Le critère de la somme des temps de complétion est mathématiquement défini comme $\Sigma_i C_i$ où C_i est le temps de fin de la tâche i. Minimiser ce critère revient souvent à favoriser l'exécution des petites tâches par rapport aux tâches plus importantes. Avec un seul processeur, l'ordonnancement optimal consiste à trier les tâches par temps d'exécution croissant et à les exécuter dans cet ordre, avantageant ainsi les tâches les plus petites. Dans un contexte de tâches malléables, ce comportement est encore vrai quand les tâches sont toutes très peu parallèles ou au contraire toutes très parallèles. Quand elles sont toutes très parallèles, l'optimal est alors de toutes les allouer sur m processeurs et de reprendre le schéma d'ordonnancement à un processeur.

Pour ne pas pénaliser les grandes tâches, ou plus simplement pour modéliser une gamme de problèmes plus large, un poids w_i est parfois associé à chaque tâche[1]. On cherche alors à minimiser la somme pondérée des temps de complétion $\Sigma_i w_i C_i$. On peut remarquer que l'optimisation de la somme des temps de complétion est équivalente à la minimisation de la moyenne des temps de complétion, puisque la seule différence entre les deux valeurs est la division par n, le nombre de tâches.

4.2.1 Optimisation du makespan

Comme on l'a vu depuis le début de cette partie, l'ordonnancement se ramène souvent à savoir «où» et «quand» une tâche va être calculée. Avec les tâches malléables la question «où» peut être scindée en deux : «combien de processeurs» et «lesquels». Les premiers algorithmes d'ordonnancement de tâches malléables ont donc abordés les deux premières parties séparément, pour que les questions «lesquels» et «quand» puissent être résolues avec un algorithme classique d'ordonnancement de tâches multiprocesseurs (parfois appelées «rigides»). Cette partie est souvent résolue avec une approximation par un algorithme de *strip packing*, c'est-à-dire d'ordonnancement de rectangles.

En 1992, Turek Wolf et Yu [TWY92] ont introduit l'idée de spécialiser la première phase («combien») pour minimiser le critère global. Leur algorithme

[1]Attention à ne pas confondre w_i le poids associé à la tâche i avec le W_i le travail de la tâche i.

essaye de minimiser dans cette phase deux bornes inférieures du makespan pour le problème de tâche rigide, le travail total et le plus long chemin.

- Le travail total est la somme des travaux de toutes les tâches. Le temps total d'exécution est plus grand que la charge moyenne des processeurs, donc plus grand que le travail total divisé par le nombre de processeurs.
- Le plus long chemin est le temps minimum d'exécution de toutes les tâches quand le nombre de processeurs est infini et que les communications sont négligeables. Ce temps minimum est le temps d'exécution d'un chemin du graphe des tâches (quand il y a des relations de précédence entre les tâches) ou d'une tâche seule quand il n'y a pas de relations de précédence.

Leur algorithme arrive au compromis en partant d'une allocation séquentielle de toutes les tâches, en augmentant à chaque fois le nombre de processeurs alloués à la plus grande tâche jusqu'à ce que le travail total divisé par le nombre de processeurs soit plus grand que le temps d'exécution de la plus grande tâche. La garantie de performance est alors fixée par la garantie de l'algorithme utilisé pour ordonnancer les tâches rigides. Le meilleur algorithme connu ayant une garantie de 2, l'algorithme complet a une garantie de 2.

Pour obtenir une meilleure approximation, la solution adoptée par G. Mounié dans sa thèse [Mou00] est de favoriser la deuxième phase en acceptant de perdre un peu sur la première phase. Les allocations étant calculées de manière à préparer l'ordonnancement qui va suivre, la garantie est bien meilleure $(3/2 + \epsilon)$. C'est sur ces travaux que repose le chapitre 3.

4.2.2 Optimisation de la somme des temps de complétion

La minimisation de la somme (parfois pondérée) des temps de complétion est également étudiée depuis très longtemps. Les premiers algorithmes polynomiaux optimaux remontent aux années 50 [Smi56] pour le problème à une machine avec ou sans pondération, et la complexité du problème à plusieurs machines a été prouvée dans le papier original de Graham [Gra69]. Le problème à une machine est simplement résolu en ordonnant les tâches suivant une règle de dominance appelée règle de Smith, qui est dans le cas pondéré d'exécuter les tâches triées dans l'ordre croissant de leur rapport[2] $\frac{t_i}{w_i}$.

Cette règle de dominance simple dans le cas de tâches indépendantes a été étendue pour obtenir des algorithmes optimaux pour plusieurs classes

[2] À nouveau t_i est le temps d'exécution et w_i le poids de la tâche.

de graphes de précédence. Pour les problèmes NP-difficiles, plusieurs algorithmes reposent également sur un tri des tâches (généralement par résolution d'un programme linéaire relaxé), suivi d'une politique inspirée de l'ordonnancement de liste [Que93]. Ce travail a ensuite été étendu au problème multiprocesseurs [HSSW97], toujours avec des tâches s'exécutant sur un seul processeur.

Pour les tâches malléables, l'approche géométrique est celle qui a donné les meilleurs résultats. Il s'agit généralement de construire un ordonnancement comme une succession d'étagères (comme celles utilisées dans le chapitre précédent), en triant ces étagères comme dans la règle de Smith, suivant le rapport de leur taille et du poids des tâches qui les composent.

L'algorithme «smart SMART» de Schwiegelshohn et al. [SLW+98] basé sur ce principe, a une garantie de performance de 8 dans le cas non pondéré et de $8,53$ dans le cas pondéré. C'est à ma connaissance le meilleur algorithme publié avant cette thèse.

4.3 Tour d'horizon du multi-critère

L'objectif de cette section est de présenter rapidement les principes de l'ordonnancement multi-critère et les idées importantes du domaine. Le lecteur intéressé peut lire par exemple le livre de T'kindt et Billaut [TB02], ou celui de Collette et Siarry [CS02] pour un résumé plus complet.

Comme nous l'avons dit au-dessus, les ordonnancements qui minimisent le temps total d'exécution et ceux qui minimisent la somme des temps d'exécution donnent des résultats très différents, et généralement mauvais pour l'autre critère. Si les deux critères sont importants, comme par exemple quand plusieurs utilisateurs ayant des points de vues différents partagent une machine, ou qu'une équipe loue une machine (minimisation du makespan) mais veut quand même mettre des priorités aux différentes tâches (minimisation de la somme pondérée des temps), il faut concevoir et utiliser des algorithmes plus généraux. Ces algorithmes peuvent dans l'absolu être moins bons sur chaque critère que d'autres algorithmes spécialisés, tout en étant meilleurs sur l'autre. Ce qui compte réellement, c'est que ces algorithmes ne soient pas plus mauvais qu'un autre algorithme connu sur les deux critères en même temps.

Nous allons maintenant présenter deux approches différentes pour concevoir des algorithmes bi-critères.

4.3.1 Première construction générique

Nous allons d'abord présenter une construction applicable à tous les types de problèmes d'ordonnancement pour lesquels on connaît à la fois un algorithme pour le makespan $\mathcal{A}_{C_{max}}$ et un pour la somme des temps de complétion $\mathcal{A}_{\sum C_i}$ (présenté dans [PSTW97]). On notera $\rho_{C_{max}}$ la garantie de l'algorithme $\mathcal{A}_{C_{max}}$ et $\rho_{\sum C_i}$ la garantie de l'algorithme $\mathcal{A}_{\sum C_i}$.

Deux phases, deux algorithmes ($\mathcal{A}_{\sum C_i}$; $\mathcal{A}_{C_{max}}$)

Proposition Il est possible de mixer les algorithmes $\mathcal{A}_{\sum C_i}$ et $\mathcal{A}_{C_{max}}$ en un algorithme ayant à la fois une garantie de $2\rho_{\sum C_i}$ sur la somme des temps de complétion et une garantie de $2\rho_{C_{max}}$ sur le makespan.

On peut d'abord remarquer que si l'on retarde l'exécution de l'ordonnancement fourni par l'algorithme $\mathcal{A}_{C_{max}}$ par un temps constant $\lambda\rho_{C_{max}}C^*_{max}$, on augmente le facteur de garantie de $\lambda\rho_{C_{max}}$. Le principe du nouvel algorithme est donc de commencer par utiliser l'ordonnancement fourni par l'algorithme $\mathcal{A}_{\sum C_i}$ puis de changer au bout de $\rho_{C_{max}}C^*_{max}$ unités de temps pour passer à l'ordonnancement $\mathcal{A}_{C_{max}}$ pour ordonnancer les tâches n'étant pas encore finies. Les tâches finissent donc toutes avant la date $2\rho_{C_{max}}C^*_{max}$. La garantie sur le makespan est donc $2\rho_{C_{max}}$. Pour montrer la garantie sur la somme des temps de complétion, il suffit de constater que seules les tâches finissant dans l'ordonnancement produit par $\mathcal{A}_{\sum C_i}$ après le temps $\rho_{C_{max}}C^*_{max}$ ont leur temps de complétion augmenté. Ces tâches finissant dans le nouvel ordonnancement avant la date $2\rho_{C_{max}}C^*_{max}$, la somme des temps de complétion est au plus doublée. La garantie sur la somme des temps de complétion est donc de $2\rho_{\sum C_i}$.

Corollaire Avec les meilleurs algorithmes spécialisés connus pour le makespan et la somme des temps de complétion (voir sections 4.2.1 et 4.2.2), l'algorithme bi-critère obtenu a une garantie de $3 + \epsilon$ sur le makespan et de 16 sur la somme des temps de complétion.

Le nouvel ordonnancement étant obtenu en enlevant des tâches à deux ordonnancements faisables, il est potentiellement plein de trous. On peut en pratique améliorer le résultat en exécutant les tâches dès que possible, mais cela n'étant pas quantifiable le résultat théorique (au pire des cas) reste le même.

Réglage fin des paramètres

Avec ce schéma d'ordonnancement, il est possible de diminuer la garantie sur un critère en augmentant celle de l'autre critère. Il suffit pour cela de changer d'ordonnancement plus ou moins tôt. Si la date où l'on passe à

l'ordonnancement généré par $\mathcal{A}_{C_{max}}$ est $\lambda\rho_{C_{max}}C^*_{max}$, la garantie sur le makespan devient alors $(1+\lambda)\rho_{C_{max}}$, et celle de la somme des temps de complétion devient alors $\frac{1+\lambda}{\lambda}\rho_{\sum C_i}$. Pour équilibrer les deux rapports, on peut prendre $\lambda = 5,33$ ce qui donne une garantie de $9,5$ sur les deux critères. La courbe complète est représentée dans la figure 4.1.

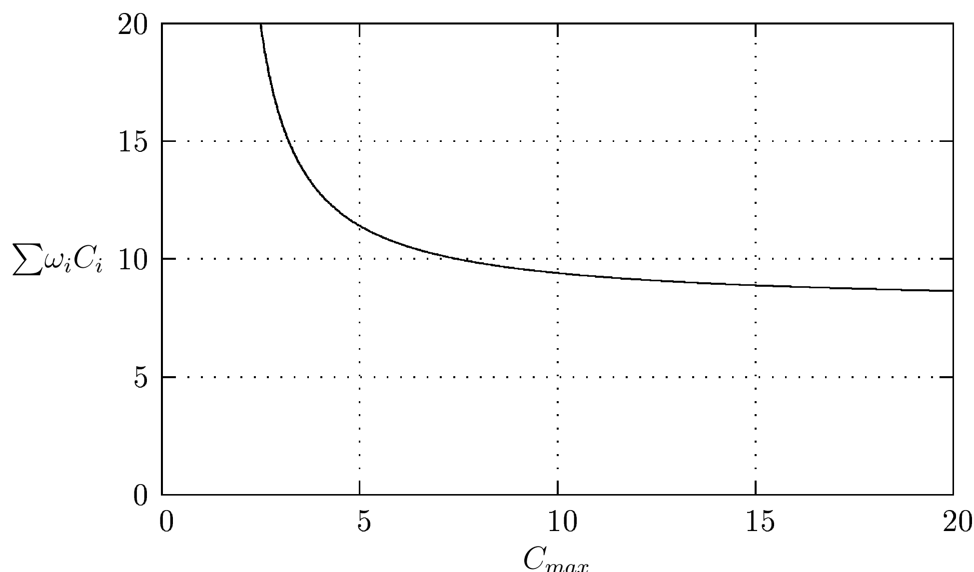

FIG. 4.1 – Courbe du compromis C_{max} / $\sum_i \omega_i C_i$.

4.3.2 Spécialisation d'un algorithme

Pour obtenir un ordonnancement bon sur deux critères différents, un autre type de construction est possible. Il s'agit d'utiliser un algorithme efficace sur l'un des critères comme brique de base d'un ordonnancement ayant une garantie sur l'autre critère. C'est cette approche que Hall *et al.* ont privilégiée (voir [HSW96] et [HSSW97]). Leur algorithme de base est un algorithme qui prend en entrée un ensemble de tâches pondérées et un temps D, et essaye de minimiser le temps nécessaire pour faire au moins autant de poids que l'optimal (pour le poids) en temps D. Ceci est proche de l'optimisation du makespan, puisque si le temps imparti à l'algorithme est C^*_{max}, l'algorithme rendra un ordonnancement ayant toutes les tâches (puisque l'optimal arrive à faire toutes les tâches en un temps C^*_{max}).

L'idée principale est d'utiliser cette brique de base pour ordonnancer des intervalles de temps ayant une taille à croissance géométrique. L'algorithme

de base étant garanti, on sait borner le temps total d'exécution. Nous allons présenter ici la construction avec les notations de [HSW96], puisque c'est cette construction dont nous nous sommes inspirés pour notre famille d'algorithmes.

L'algorithme de base est techniquement noté **MSWP** de l'anglais *Maximum Scheduled Weight Problem*[3]. Comme nous l'avons brièvement dit plus haut, il prend en entrée un ensemble de tâches S et un temps D, et rend un ordonnancement d'un sous-ensemble de S en temps ρD (ρ étant le facteur d'approximation de l'algorithme), tel que tous les sous-ensembles de S ayant un poids supérieur au sous-ensemble choisi aient un makespan optimal supérieur à D.

Partant de l'hypothèse que la plus petite tâche a un temps de complétion d'au moins 1 quand toutes les machines lui sont allouées[4], on coupe l'échelle de temps en lots de taille 2^l. On définit $\tau_1 = 1$ et $\tau_l = 2^{l-1}$ (pour tout l) comme les bornes du découpage temporel. L'ordonnancement est alors construit itérativement de la façon suivante. À chaque étape l, on considère l'ensemble J_l des tâches disponibles (et pas encore ordonnancées) au temps τ_l. L'algorithme de **MSWP** est alors invoqué avec cet ensemble de tâches et le temps $D = \tau_l$, et le résultat est placé entre $\rho\tau_l$ et $\rho\tau_{l+1}$.

Pour prouver que l'ordonnancement final sera garanti sur la somme pondérée des temps de complétion, il faut ici montrer un lemme de dominance. Soit $\bar{S}_l$ et $\bar{W}_l$ respectivement l'ensemble de tâches ordonnancées par l'algorithme **MSWP** à l'étape l et le poids de cet ensemble. On définit de la même façon S_l et W_l comme étant l'ensemble des tâches ordonnancées entre τ_{l-1} et τ_l dans un ordonnancement fixé, et le poids des tâches de cet ensemble. L'ensemble $S = (\cup_{k=1}^{l} S_k) - (\cup_{k=1}^{l-1} \bar{S}_k)$ est l'ensemble des tâches faites dans cet ordonnancement fixé avant la date τ_l mais que l'algorithme n'a pas fait avant $\rho\tau_l$. Comme ces tâches finissent toutes avant τ_l dans l'ordonnancement fixé, et qu'elles sont toutes disponibles à l'étape l de l'algorithme, le poids de l'ensemble choisi à l'étape l par l'algorithme est plus grand que le poids de S. Ce qui permet d'écrire :

$$\forall l, \quad \sum_{k=1}^{l} \bar{W}_k \geq \sum_{k=1}^{l} W_k$$

[3]Problème du Poids Ordonnancé Maximal. La définition est dans la suite du texte.

[4]Cette hypothèse n'est pas aussi anodine qu'on pourrait le penser. Non seulement elle donne la taille de départ des lots (*batches*) pour l'algorithme, mais elle permet en plus de s'affranchir de tous les scénarios d'adversaire utilisant des tâches de taille ϵ. La différence de tailles entre les différentes tâches ne peut donc se faire que par l'utilisation de tâches géantes, qui de toutes façons feront augmenter le temps de complétion général.

Lemme 10 *Le poids ordonnancé par l'algorithme entre* 0 *et* $\rho\tau_{l+1}$ *est au moins aussi grand que le poids fait par n'importe quel ordonnancement entre* 0 *et* τ_l.

Soit L tel que toutes les tâches d'un ordonnancement optimal pour la somme pondérée des temps d'exécution soient finies à la date τ_L. Les tâches de $\bar{S}_l$ étant toutes finies au temps $\rho\tau_{l+1}$, la somme pondérée des temps de complétion de l'ordonnancement est inférieure à $\sum_{l=1}^{L} \rho\tau_{l+1}\bar{W}_l$. Grâce au lemme précédent, au fait que $\tau_{l+1} = 4\tau_{l-1}$ et au fait que les tâches de S_l finissent toutes après le temps τ_{l-1}, on peut écrire la série d'inégalités suivantes, qui mènent à la garantie pour la somme pondérée des temps de complétion.

$$\sum_{l=1}^{L} \rho\tau_{l+1}\bar{W}_l \leq 4\rho \sum_{l=1}^{L} \tau_{l-1}\bar{W}_l \leq 4\rho \sum_{l=1}^{L} \tau_{l-1}W_l \leq 4\rho \sum_{j=1}^{n} \omega_j C_j^*$$

Pour la garantie sur le makespan, il suffit de constater que s'il reste des tâches à l'étape L, c'est qu'à l'étape $L-1$ toutes n'ont pas pu terminer. Donc $\tau_{L-1} < C_{max}^*$, or l'ordonnancement construit se finit au pire au temps $\rho\tau_{L+1}$, ce qui implique que la garantie sur le makespan est de 4ρ également.

Une modification rendant un paramètre aléatoire permet d'améliorer ces résultats et d'obtenir la garantie de $\frac{2}{\ln(2)}\rho$ pour les deux critères[5]. Comme nous réutiliserons cette technique, elle sera présentée en détail dans la suite de ce chapitre.

Optimisation hors-ligne

L'algorithme que nous venons de présenter a été conçu dans le cadre de la résolution de problèmes en ligne (*on-line*), c'est-à-dire où les tâches ne sont connues que lorsqu'elles deviennent disponibles. Puisqu'ici nous nous sommes uniquement intéressés aux problèmes hors-ligne (*off-line*), nous pouvons légèrement modifier l'algorithme pour gagner un facteur 2 supplémentaire sur le makespan. Il s'agit de trouver la valeur minimale t_{min} pour laquelle l'algorithme **MSWP** ordonnance toutes les tâches. Ce temps est une borne inférieure du makespan optimal, mais surtout on peut choisir la valeur de τ_1 pour que $\tau_L = t_{min}$. L'ordonnancement étant fini au temps $\rho\tau_{L+1}$, la garantie est alors de 2ρ.

[5]Attention, ceci est une garantie en moyenne, donc pas forcément atteignable pour les deux critères en même temps.

4.4 Une nouvelle famille d'algorithmes pour les tâches malléables

4.4.1 Un meilleur algorithme MSWP

Dans l'article [CPS+96], les auteurs utilisent un algorithme pour le problème **MSWP** ayant une garantie de $3 + \epsilon$ (pour le modèle des tâches malléables). Or pour les tâches malléables, le meilleur algorithme d'approximation pour les tâches indépendantes a une garantie de $3/2 + \epsilon$ (voir la thèse de Gregory Mounié [Mou00]). On peut donc gagner un facteur 2 en adaptant cet algorithme pour en faire un algorithme résolvant le problème **MSWP**. Cette modification se fait au niveau du sac à dos qui partitionne les tâches en tâches très parallèles et tâches moins parallèles (voir le chapitre précédent pour une description détaillée du problème de sac à dos et sa résolution par programmation dynamique). Au lieu de partitionner les tâches disponibles en deux ensembles (parallèles *vs.* non parallèles), nous allons les scinder en trois ensembles : parallèles, non parallèles et rejetées.

Le seul défaut de cette approche est que la complexité de l'algorithme est fortement augmentée. En effet en prenant des poids entiers, la complexité est multipliée par un facteur $n\omega_{max}$, où n est le nombre de tâches et ω_{max} est le poids maximum d'une tâche. Cependant dans les problèmes réels, les poids sont généralement entiers et l'écart n'est pas très important (avec des écarts très importants, l'ordre d'exécution des tâches est fixé par les poids et l'on peut approximer par des algorithmes d'ordonnancement avec contraintes de précédence).

On passe donc de $12 + \epsilon$ à $6 + \epsilon$ (et même $3 + \epsilon$ pour le makespan) pour l'algorithme présenté dans la section 4.3.2, qui devient alors meilleur que celui de la section 4.3.1.

4.4.2 Une famille d'algorithmes

Nous pouvons ici aussi rajouter un paramètre variable pour (comme dans la section 4.3.1) avoir une meilleure garantie sur un critère au détriment de l'autre. Pour introduire le paramètre, nous allons d'abord étudier la transformation d'un ordonnancement optimal pour la somme des temps de complétion (noté $O^*_{\sum C_i}$) en un ordonnancement $\bar{O}$ similaire à ceux produits par l'algorithme pour un paramètre fixé $\alpha > 1$.

1. Soit C^*_{max} le temps de complétion optimal de l'instance considérée. Soit k le plus petit entier tel qu'il n'y ait pas de tâches qui puissent s'exécuter en moins de $\frac{C^*_{max}}{\alpha^k}$.

2. Si l'on considère le début de l'ordonnancement $O^*_{\sum C_i}$ jusqu'au temps $\tau_1 = \frac{C^*_{max}}{\alpha^{k-1}}$, on a un ordonnancement pour un sous-ensemble de tâches. L'algorithme **MSWP** peut ordonnancer ce sous-ensemble de tâches en $\rho\frac{C^*_{max}}{\alpha^{k-1}}$. Ces tâches sont placées dans l'ordonnancement $\bar{O}$ entre $\bar{\tau}_1 = \rho\frac{C^*_{max}}{\alpha^{k-1}(\alpha-1)}$ et $\bar{\tau}_2 = \bar{\tau}_1 + \rho\frac{C^*_{max}}{\alpha^{k-1}} = \rho\frac{C^*_{max}}{\alpha^{k-2}(\alpha-1)}$.
3. On procède de même pour tous les intervalles $[\tau_{k-j}; \tau_{k-j+1}]$ (pour j allant de $k-2$ à 1), en ordonnançant avec **MSWP** les tâches restantes ayant un temps de complétion $C_i < \tau_{k-j} = \frac{C^*_{max}}{\alpha^j}$ en $\rho\frac{C^*_{max}}{\alpha^j}$ unités de temps, placées entre $\bar{\tau}_{k-j} = \rho\frac{C^*_{max}}{\alpha^j(\alpha-1)}$ et $\bar{\tau}_{k+1-j} = \rho\frac{C^*_{max}}{\alpha^{j-1}(\alpha-1)}$
4. Toutes les tâches restantes peuvent alors être ordonnancées en ρC^*_{max} unités de temps, et placées à la fin de l'ordonnancement.

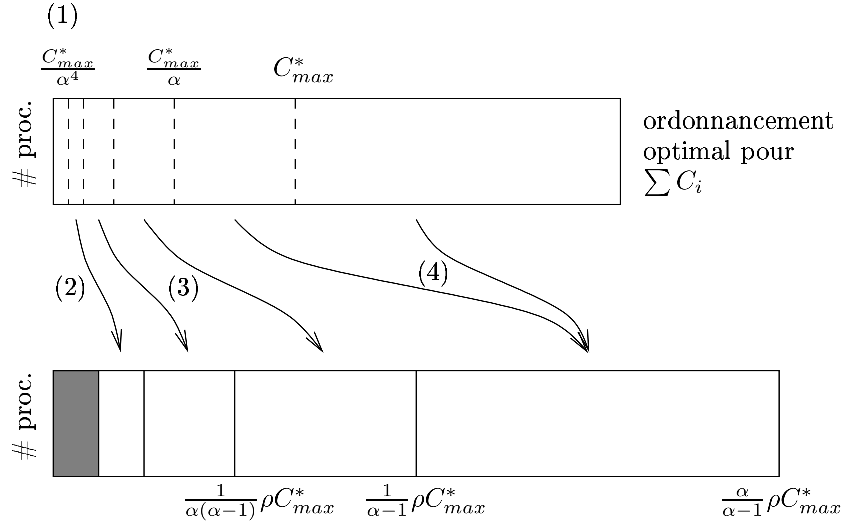

FIG. 4.2 – Transformation d'un ordonnancement $O^*_{\sum C_i}$ en ordonnancement bi-critère (ici $k = 4$ et $\alpha = 2$).

Cette transformation est illustrée dans la figure 4.2. Les nombres entre parenthèses sur la figure permettent d'identifier quelles tâches sont placées à quelle étape. Si l'on note $C_i^{\mathbf{s}}$ le temps de complétion des tâches avant la transformation et $C_i^{\mathbf{t}}$ le temps de complétion après la transformation, pour toute tâche i telle que $\frac{C^*_{max}}{\alpha^j} < C_i^{\mathbf{s}} \leq \frac{C^*_{max}}{\alpha^{j-1}}$ on a $\rho\frac{C^*_{max}}{\alpha^{j-1}(\alpha-1)} < C_i^{\mathbf{t}} \leq \rho\frac{C^*_{max}}{\alpha^{j-2}(\alpha-1)}$ après la transformation, d'où l'on peut déduire l'inégalité : $C_i^{\mathbf{t}} < \frac{\alpha^2}{\alpha-1}\rho C_i^{\mathbf{s}}$. Cette inégalité nous donne directement une garantie de $\frac{\alpha^2}{\alpha-1}\rho$ sur la somme des temps de complétion. La garantie sur le makespan se calcule en faisant la somme des temps d'exécution de tous les intervalles, c'est-à-dire $\frac{\alpha}{\alpha-1}\rho$.

Nous allons maintenant définir une famille d'algorithmes basés sur ce principe d'intervalles croissants.

Description de l'algorithme

Comme nous nous plaçons dans un contexte hors-ligne nous pouvons, avant de faire un quelconque ordonnancement, exécuter plusieurs fois l'algorithme **MSWP** pour obtenir par dichotomie la plus petite borne $\tilde{C}_{max}$ telle que **MSWP** puisse ordonnancer toutes les tâches en temps $\rho\tilde{C}_{max}$. Cette borne est une borne inférieure de $C^*_{max} + \epsilon$, où ϵ est une constante qui dépend de la taille de la dernière itération de la dichotomie.

Grâce à cette borne et au temps d'exécution des tâches, on peut déterminer le plus petit k tel qu'aucune tâche ne s'exécute en moins de $\frac{\tilde{C}_{max}}{\alpha^k}$ unités de temps. L'algorithme s'exécute itérativement, en utilisant à chaque étape j l'algorithme **MSWP** avec une date $D_j = \frac{\tilde{C}_{max}}{\alpha^{k-j}}$ pour remplir une étagère de taille $\rho\frac{\tilde{C}_{max}}{\alpha^{k-j}}$. Par définition de $\tilde{C}_{max}$, la dernière étape est certaine de contenir toutes les tâches n'ayant pas été ordonnancées dans les étapes précédentes.

Le temps de complétion total est donc au plus de $\frac{\alpha}{\alpha-1}\rho C^*_{max}$. Pour obtenir la garantie de performance sur la somme des temps de complétion, il faut à nouveau utiliser le lemme 10. Comme précédemment, la garantie se fait entre le début d'un intervalle et la fin de l'image du suivant. Les garanties de performance sont donc de $\frac{\alpha}{\alpha-1}\rho$ pour le temps de complétion et de $\frac{\alpha^2}{\alpha-1}\rho$ pour la somme des temps de complétion.

Analyse des résultats

Pour illustrer ces résultats, nous avons tracé dans la figure 4.3 l'ensemble des couples de garanties de performance que l'on peut obtenir quand α est plus grand que 1 (en prenant le meilleur **MSWP** possible, c'est-à-dire $\rho = 3/2$). Une simple étude de courbe montre que la garantie sur la somme des temps de complétion atteint son minimum pour $\alpha = 2$. Ce minimum est donc de 4ρ. En ce point la garantie de performance sur le makespan est de 2ρ. Les valeurs atteintes pour $\alpha > 2$ sont indiquées par un trait continu, le trait en pointillés indiquant les valeurs pour $\alpha < 2$. On peut voir sur la courbe que seuls les algorithmes correspondant à un paramètre α supérieur ou égal à 2 sont intéressants. En effet ce sont pour le moment les meilleurs algorithmes connus pour le problème, c'est-à-dire que pour chacun d'entre eux il n'existe pas encore d'algorithme ayant une meilleure garantie sur les deux critères en même temps.

Pour donner un point de comparaison nous avons ajouté sur la courbe les points représentant les algorithmes présentés dans [CPS+96]. Ces algorithmes

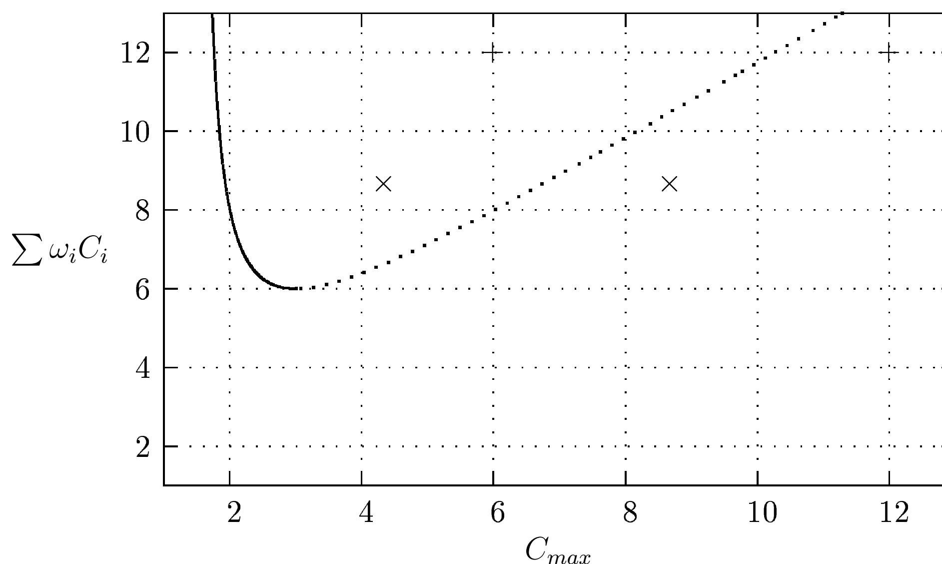

FIG. 4.3 – Courbe de la famille d'algorithmes

ayant été écrits pour la version hors-ligne du problème, nous avons divisé par deux leur garantie sur le makespan pour obtenir une comparaison honnête. Ceci correspond à la détermination d'une borne inférieure $\tilde{C}_{max}$ du makespan optimal par l'algorithme **MSWP**, qui permet de fixer ensuite la taille du premier intervalle de façon à terminer par un intervalle de taille $\rho\tilde{C}_{max}$. On fait en fait l'économie de l'incertitude sur la taille du dernier intervalle par rapport à $\tilde{C}_{max}$. Les deux points matérialisés par un $+$ correspondent donc au résultat (12 ;12) (par l'algorithme déterministe) amélioré en (6 ;12), tandis que les deux $\times$ correspondent au résultat (8,67 ;8,67) (par l'algorithme aléatoire) amélioré en (4,34 ;8,67).

4.5 Introduction de l'aléatoire

Nous allons maintenant améliorer un peu plus les résultats précédents en utilisant l'approche aléatoire présentée dans [CPS+96]. Le principe de base de cette approche est d'utiliser plusieurs fois un des algorithmes de la famille précédente, en faisant varier un paramètre aléatoire pour obtenir plusieurs ordonnancements d'une même instance et ne garder que le meilleur. Les garanties de performance telles qu'on les a présentées sont basées sur l'analyse du pire cas. Or les instances qui atteignent ces valeurs de pire cas

sont différentes suivant la valeur du paramètre que l'on va faire varier. En moyenne le résultat est donc bien meilleur.

Le paramètre que nous faisons varier aléatoirement est un facteur multiplicatif sur le temps. Les pires cas pour la somme des temps de complétion étant atteints quand une tâche qui finissait juste après une des bornes d'intervalle est ordonnancée de façon à finir juste avant la frontière de l'intervalle image, faire varier l'échelle de temps permet de faire passer la tâche en question du début d'un intervalle à la fin de l'intervalle précédent. Nous allons donc multiplier tous les temps par un facteur β choisi dans l'intervalle $]1/\alpha; 1]$. Ce facteur étant généralement inférieur à 1, il faudra rajouter une étagère de taille $\rho\tilde{C}_{max}$ à la fin de l'ordonnancement pour être sûr de bien ordonnancer la totalité des tâches. Rajouter ce lot nous fait malheureusement perdre le gain obtenu avec le passage vers les problèmes hors-ligne. En contrepartie, la version en ligne de notre algorithme devient quasiment aussi compétitive que la version hors ligne, et reste meilleure que les versions existantes.

Théorème 3 *En multipliant les variables temporelles de l'algorithme précédent par une variable aléatoire prise uniformément dans l'intervalle* $]1/\alpha; 1]$, *on obtient en moyenne une garantie de* $\frac{\alpha}{\ln(\alpha)}\rho$ *sur la somme des temps de complétion et de* $(1 + \frac{1}{\ln(\alpha)})\rho$ *sur le temps total de complétion.*

Comme dans la section 4.3.2, nous allons comparer l'ordonnancement obtenu par notre algorithme à un ordonnancement optimal pour la somme des temps de complétion. Les notations sont exactement les mêmes, à l'exception du fait que maintenant τ_l et $\bar{\tau}_l$ valent respectivement $\frac{\tilde{C}_{max}}{\alpha^{k-l}}$ et $\rho\frac{\tilde{C}_{max}}{\alpha^{k-l}(\alpha-1)}$. Le lemme 10 est également toujours valide ici.

Considérons donc un ordonnancement optimal pour la somme des temps de complétion. Soit $\tau_{fin(i)}$ la date de début de l'intervalle dans lequel se finit la tâche T_i (c'est-à-dire $\tau_{fin(i)} \leq C_i^* < \tau_{fin(i)+1}$). Comme dans la section 4.3.2, la somme des temps de complétion est bornée par $\sum_{l=1}^{L} \bar{\tau}_{l+1}\bar{W}_l$, et nous pouvons écrire les inégalités suivantes :

$$\sum_{l=1}^{L} \bar{\tau}_{l+1}\bar{W}_l \leq \alpha^2 \sum_{l=1}^{L} \bar{\tau}_{l-1}\bar{W}_l \leq \rho\frac{\alpha^2}{\alpha-1} \sum_{l=1}^{L} \tau_{l-1}\bar{W}_l \leq \rho\frac{\alpha^2}{\alpha-1} \sum_{l=1}^{L} \tau_{l-1}W_l$$

À partir de cette étape, l'analyse va être plus fine que dans la section 4.3.2. Nous n'allons pas majorer directement le terme $\sum_{l=1}^{L} \tau_{l-1}W_l$ par $\sum_{i=1}^{n} w_iC_i^*$, mais simplement l'écrire différemment sous la forme $\sum_{i=1}^{n} w_i\tau_{fin(i)}$, en remplaçant les sommes partielles W_l par les tâches qui les composent.

$$\sum_{i=1}^{n} w_i \bar{C}_i \leq \rho \frac{\alpha^2}{\alpha - 1} \sum_{i=1}^{n} w_i \tau_{fin(i)}$$

La dernière inégalité porte donc sur les dates de début des intervalles dans lesquels les tâches se terminent dans l'optimal. L'espérance étant une fonction linéaire, il suffit de connaître l'espérance de $\tau_{fin(i)}$ en fonction de C_i^* pour connaître une borne sur l'espérance de la somme des temps de complétion. On écrit β sous la forme α^{-X}, où X est une variable aléatoire uniformément distribuée sur l'intervalle $]0;1]$.

Soit $\gamma_i \in [0;1[$ et l_i entier tel que $C_i^* = \alpha^{-\gamma_i} \frac{\tilde{C}_{max}}{\alpha^{l_i}}$. Si $\beta = \alpha^{-X}$ est inférieur ou égal à $\alpha^{-\gamma_i}$ on a par définition $\tau_{fin(i)}$ qui est égal à $\beta \frac{\tilde{C}_{max}}{\alpha^{l_i}}$, et sinon $\tau_{fin(i)}$ vaut $\beta \frac{\tilde{C}_{max}}{\alpha^{l_i+1}}$. À partir de cette constatation un calcul simple nous permet d'évaluer l'espérance de $\tau_{fin(i)}$ en fonction de C_i^*.

$$\begin{aligned}
E[\tau_{fin(i)}] &= \int_0^1 \tau_{fin(i)} dX \\
&= \int_0^{\gamma_i} \alpha^{-X} \frac{\tilde{C}_{max}}{\alpha^{l_i+1}} dX + \int_{\gamma_i}^1 \alpha^{-X} \frac{\tilde{C}_{max}}{\alpha^{l_i}} dX \\
&= \int_0^{\gamma_i} \alpha^{-X} \alpha^{\gamma_i - 1} C_i^* dX + \int_{\gamma_i}^1 \alpha^{-X} \alpha^{\gamma_i} C_i^* dX \\
&= \alpha^{\gamma_i} C_i^* \left(\frac{1}{\alpha} \int_0^{\gamma_i} \alpha^{-X} dX + \int_{\gamma_i}^1 \alpha^{-X} dX \right) \\
&= \alpha^{\gamma_i} C_i^* \left(\frac{1}{\alpha} \left(-\frac{\alpha^{-\gamma_i}}{\ln(\alpha)} + \frac{1}{\ln(\alpha)} \right) + \left(-\frac{\alpha^{-1}}{\ln(\alpha)} + \frac{\alpha^{-\gamma_i}}{\ln(\alpha)} \right) \right) \\
E[\tau_{fin(i)}] &= C_i^* \left(-\frac{1}{\alpha \ln(\alpha)} + \frac{1}{\ln(\alpha)} \right) = \frac{\alpha - 1}{\alpha \ln(\alpha)} C_i^*
\end{aligned}$$

Ce qui nous permet de conclure sur l'espérance de la somme des temps de complétion :

$$E\left[\sum_{i=1}^{n} w_i \bar{C}_i \right] \leq \frac{\alpha \rho}{\ln(\alpha)} \sum_{i=1}^{n} w_i C_i^*$$

La garantie sur la somme des temps de complétion est donc $\frac{\alpha\rho}{\ln(\alpha)}$. La garantie sur le makespan dans le pire cas est augmentée à cause de l'étagère supplémentaire de taille fixe $\rho \tilde{C}_{max}$ et devient donc $\frac{\alpha}{\alpha-1}\rho + \rho = \frac{2\alpha-1}{\alpha-1}\rho$. On peut également définir une garantie en moyenne sur le makespan, puisque la

longueur totale d'exécution varie aussi en fonction de β. Cette valeur moyenne est de $\frac{\rho}{\ln(\alpha)} + \rho = (1 + \frac{1}{\ln(\alpha)})\rho$.

Il faut cependant souligner le fait qu'il peut très bien ne pas exister de valeur de β telle que l'ordonnancement obtenu soit aussi bon que les valeurs moyennes le suggèrent sur les deux critères en même temps. C'est pour cela que sur la figure 4.4 nous avons représenté à la fois la courbe des deux valeurs moyennes (trait épais) et celles de la valeur moyenne pour la somme des temps de complétion et de la valeur pour le pire cas du makespan (trait fin). Pour permettre une comparaison avec les résultats précédents, la courbe de la famille précédente est tracée en pointillés.

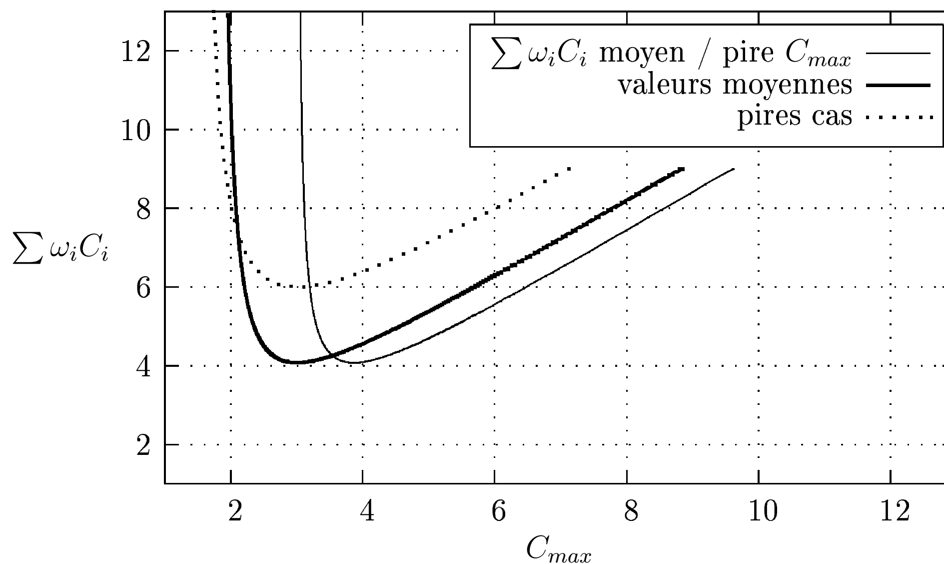

FIG. 4.4 – Comparaison des différents algorithmes

On peut noter que contrairement au cas précédent, le minimum des courbes pour la somme des temps de complétion est atteint pour la valeur $\alpha = e \simeq 2,72$ au lieu de $\alpha = 2$. La valeur minimale est alors de $e\rho$, qui vaut environ $4,08$ si l'on prend l'algorithme **MSWP** introduit dans ce chapitre ($\rho = 3/2$). La garantie sur le makespan vaut alors en moyenne 3, et au pire $3,88$ environ.

4.6 Conclusion

Les techniques utilisées ici ont été présentées avec les tâches malléables et l'algorithme de Mounié [Mou00] pour faciliter l'exposé. Cependant ce sont des

procédés génériques qui peuvent être étendus à d'autres problèmes. On peut tout à fait utiliser cette approche avec l'algorithme du chapitre précédent pour obtenir une famille d'algorithmes garantis sur les deux critères dans un cadre de plate-forme hiérarchique.

Ces algorithmes bi-critères ont une garantie beaucoup plus petite que les précédents, ouvrant la voie d'une utilisation pratique. En simplifiant le principe de ces algorithmes on obtient en effet des résultats très bons en moyenne [DEMT04] qui par contre ne sont plus théoriquement garantis.

Deuxième partie

Tâches Divisibles

Chapitre 5

Chaînes de Processeurs

L'application étudiée dans cette partie sera composée d'un seul ensemble de tâches identiques et indépendantes les unes des autres. Ce modèle, proche de certaines applications réelles comme SETI [SETI] ou la recherche de nombres premiers de Mersenne [Mers], est très similaire au modèle des tâches divisibles introduit par Cheng et Robertazzi [CR88]. Ce modèle correspond particulièrement aux applications de type paramétrique [Ciment], pour lesquelles un grand nombre de calculs indépendants sont fait à partir d'un jeu de données qui ne varie que par quelques paramètres.

Dans le modèle des tâches divisibles, l'ensemble du travail à effectuer peut être partitionné en blocs de n'importe quelle taille et est ensuite distribué à un ensemble de processeurs. Dans les premiers travaux de Cheng et Robertazzi [CR88], il s'agissait de partager l'analyse d'un grand ensemble de données entre plusieurs processeurs reliés en chaînes. La propagation des données se fait dans cet article en une seule fois, c'est-à-dire qu'il n'y a qu'une communication entre deux processeurs voisins. Le temps de calcul et le temps de communication étant liés au volume de données traité, l'ordonnancement optimal est obtenu par une formule analytique simple.

Beaucoup d'autres topologies ont été étudiés par la suite avec des processeurs homogènes, comme les arbres [CR90], ou avec des processeurs hétérogènes, comme les bus [SRL98] ou les étoiles [CRL00]. La différence entre ces deux architectures est que dans le cas du bus le temps de communication est le même pour tous les processeurs, alors que dans le cas de l'étoile les liens peuvent être de différentes natures, et donc avoir des vitesses différentes. Dans les deux cas les solutions étudiés dans ces articles correspondent à un déploiement en une seule fois, ce qui conduit à des ordonnancements relativement simples.

La centralisation des données traitées n'est généralement pas pris en compte dans les travaux sur les tâches divisibles. En effet, pour de nom-

breuses applications comme la recherche d'un mot dans une base de données, ce retour est beaucoup plus simple que l'envoi initial de données, donc beaucoup plus rapide. Il a également été récemment démontré qu'en gardant le même ordre pour le retour que pour le déploiement[1] on obtenait une solution asymptotiquement optimale [AGR03].

Cependant, on peut remarquer que le travail étant parfaitement divisible, il n'y a pas lieu de se limiter à une propagation des données en une seule communication. Scinder les communications en plusieurs étapes permet de commencer le travail sur les premières parties sans avoir besoin d'attendre la totalité du travail à faire localement.

Cette constatation a conduit plusieurs auteurs [YC03, BLR03] à considérer des ordonnancements composé de plusieurs distribution de travail (*multi-round*). Ces algorithmes sont asymptotiquement optimaux, c'est-à-dire que quand le nombre de tournées augmente, le rapport entre le temps de complétion optimal et le temps obtenu par l'algorithme tend vers 1.

Plutôt que de distribuer le travail en plusieurs tours, une autre solution est la distribution de ce travail en paquets de même taille. En effet cela permet une plus grande souplesse dans l'ordre dans lequel on envoie des données aux différents travailleurs. C'est le cadre de cette partie. Une très bonne bibliographie sur les tâches identiques dans le cadre maître esclave et sur les modèles proches est disponible en ligne [BLR02b].

Le critère d'optimisation étudié dans le cadre de ce travail est la minimisation du temps total de complétion (*makespan*). Ce critère est proche de la maximisation du nombre de tâches effectuée en régime permanent, qui a été étudié en détail dans la thèse d'Arnaud Legrand [Leg03].

Les réseaux étudiés dans ce chapitre se limiteront aux chaînes de processeurs hétérogènes. Résoudre le problème des tâches divisibles sur les chaînes de processeurs hétérogènes permet de résoudre le problème pour une classe de graphes ayant certaines propriétés particulières [Li02]. C'est également avec les arbres à un seul niveau (déjà traités dans la littérature [BLR02a]) un premier pas vers les structures plus complexes qui seront traitées dans les chapitres suivants.

5.1 Définitions

Dans ce chapitre, nous allons considérer une chaîne de processeurs identique à celle de la figure 5.1. Chaque processeur possède un lien de communication de latence c_i et un temps de cycle de w_i. Ce qui signifie qu'une tâche

[1] C'est-à-dire l'ordre FIFO (*First In First Out*).

mettra c_1 unités de temps pour arriver sur le premier processeur si le lien est inoccupé et w_1 unités de temps pour être exécutée si le processeur est libre. On appelle parfois « maître » la source de tâches qui est connectée au premier processeur.

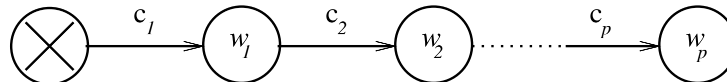

FIG. 5.1 – Le premier noeud est la source des tâches.

Les liens et les processeurs sont à utilisation exclusive. Il ne peut pas y avoir de partage d'un lien entre deux tâches ou de partage d'un processeur.

Mathématiquement pour définir un ordonnancement, on donne habituellement la date $T(i)$ de début d'exécution d'une tâche (notée $\sigma(i)$ dans la partie précédente), ainsi que le processeur $P(i)$ qui lui est alloué. Ici plusieurs contraintes font que plus d'informations sont nécessaires. D'abord les communications exclusives font qu'il faut ordonnancer aussi les transmissions. On peut également retarder dans chaque processeur le calcul ou la réémission d'une tâche (alors qu'on ne peut en aucun cas l'interrompre). Il faut donc préciser pour chaque tâche l'ensemble $C(i)$ des temps de début de transmission pour tous les liens de communication empruntés. Cet ensemble est décrit sous la forme d'un vecteur ayant pour longueur le nombre de liens traversés. L'élément k de ce vecteur, noté C_k^i, est la date d'émission[2] de la tâche i par le processeur $k-1$.

Définition 8 Ordonnancement. *L'ordonnancement de n tâches sur p processeurs est défini par la donnée de trois fonctions $P(i)$, $T(i)$ et $C(i)$ décrites ci-dessous. On note $1..p$ l'ensemble des entiers de 1 à p.*

$$\begin{aligned} P(i) &: 1..n \rightarrow 1..p \\ T(i) &: 1..n \rightarrow \mathbb{N} \\ C(i) &: 1..n \rightarrow \{\mathbb{N}^i | i \in 1..p\} \end{aligned}$$

Définition 9 Ordonnancement réalisable. *Un ordonnancement est réalisable s'il vérifie les quatre propriétés suivantes :*

$$\forall i \in 1..n\ \forall k \in 2..P(i) \qquad C_{k-1}^i + c_{k-1} \leq C_k^i \tag{5.1}$$

[2] Il s'agit en fait du début de la communication entre les processeurs $k-1$ et k. Le processeur maître a par convention le numéro 0.

$$\forall i \in 1..n \qquad C^i_{P(i)} + c_{P(i)} \leq T(i) \tag{5.2}$$

$$\forall i,j \in 1..n \qquad P(i) = P(j) \Longrightarrow |T(i) - T(j)| \geq w_{P(i)} \tag{5.3}$$

$$\forall i,j \in 1..n \ \forall k \in 1..p \qquad (k \leq P(i) and\ k \leq P(j)) \Longrightarrow |C^i_k - C^j_k| \geq c_k \tag{5.4}$$

La première condition traduit que la tâche est complètement reçue par un processeur avant d'être réémise. La deuxième en est le prolongement final, c'est-à-dire qu'il faut que la tâche soit complètement reçue avant d'être calculée. La troisième condition implique que deux tâches ne peuvent s'exécuter en même temps sur un processeur et la dernière que deux tâches ne peuvent pas être transmises en même temps sur un lien de communication.

Dans la figure 5.2, nous donnons un exemple de représentation graphique d'un ordonnancement assez simple sur une chaîne réduite à deux processeurs. Chaque trait vertical représente une unité de temps. Les flèches en diagonale sont les transmissions et les flèches horizontales sont les temps d'exécution. La deuxième tâche émise par le maître est retardée d'une unité de temps avant d'être exécutée pour ne pas empiéter sur l'exécution de la tâche précédente. Cette représentation permet de montrer l'ordonnancement des communications, et remplacera donc dans toute cette partie les diagrammes de Gantt présentés dans la première partie.

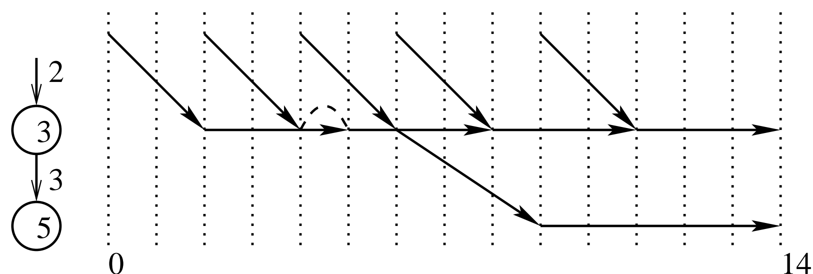

FIG. 5.2 – Représentation graphique d'un ordonnancement.

Définition 10 Temps total de calcul T_{max}. *Le temps total de calcul est la différence entre la date d'émission de la première tâche par le maître et la date de terminaison de la dernière exécution.*

Nous aurons besoin par la suite de comparer des vecteurs de communications. Pour cela nous allons utiliser un ordre lexicographique étendu dans le cas où les vecteurs sont de tailles différentes. Pour comparer deux vecteurs de tailles identiques, nous comparons les valeurs des composantes deux à deux,

la première différence indique lequel est le plus grand. Si les vecteurs sont de tailles différentes, soit les premiers éléments du plus long sont différents de l'autre vecteur et dans ce cas on fait la même comparaison que pour deux vecteurs de mêmes tailles, soit un vecteur est préfixe de l'autre et dans ce cas le plus long est inférieur au plus court[3].

Définition 11 Comparaison de vecteurs. *Soit* $A = \{a_1, \ldots, a_i\}$ *et* $B = \{b_1, \ldots, b_j\}$ *deux vecteurs de communications, on dit que* A *est inférieur à* B *(noté* $A \prec B$*) si et seulement si une des deux conditions suivantes est vérifiée :*

- $i > j$ *et* $\forall k \in 1..j \; a_k = b_k$
- $\exists k \in 1..min(i;j)$ *tel que* $a_k \neq b_k$ *et si* l *est le plus petit indice tel que* a_k *et* b_k *diffèrent, alors* $a_l < b_l$

5.2 Algorithme

- **Entrée :** n tâches et une chaîne $(c_i)_{i \in 1..p}$, $(w_i)_{i \in 1..p}$
- **Sortie :** Un ordonnancement réalisable pour les n tâches, c'est-à-dire les trois fonctions P, T et C.

L'algorithme qui donne l'ordonnancement optimal des tâches est complètement décrit dans la figure 5.3. Le principe de cet algorithme est de construire l'ordonnancement à partir de la fin du temps alloué, en remontant le temps et sans jamais remettre en cause les choix effectués. L'idée principale derrière cette construction est de réduire au maximum l'utilisation des ressources de communication pour pouvoir ordonnancer le maximum de tâches en un minimum de temps. En partant de la fin de l'ordonnancement, on peut en plaçant chaque tâche choisir l'allocation qui va permettre de laisser le plus de place aux tâches précédentes. Nous montrerons plus loin que cette solution est optimale.

Pour expliquer le déroulement de l'algorithme il faut introduire ici deux nouvelles définitions. On notera h l'enveloppe des communications (respectivement o l'occupation des processeurs) à une étape donnée de l'algorithme, c'est-à-dire le vecteur des temps à partir desquels les liens de communications (respectivement les processeurs) sont utilisés pour la première fois.

La description de l'algorithme en langage courant est la suivante. D'abord on définit une limite supérieure au temps total d'exécution qui correspond

[3]Contrairement aux apparences, ceci n'est pas l'ordre lexicographique sur les mots. Les huit sous ensembles de l'ensemble $\{1,2,3\}$ sont triés dans l'ordre $\{\} < \{1\} < \{1,2\} < \{1,2,3\} < \{1,3\} < \{2\} < \{2,3\} < \{3\}$ par l'ordre lexicographique sur les mots, alors que mon ordre est : $\{1,2,3\} < \{1,2\} < \{1,3\} < \{1\} < \{2,3\} < \{2\} < \{3\} < \{\}$.

```
// Calculer une borne supérieure du temps total
      T∞ = c1 + (n-1) * max(w1,c1) + w1
// Initialisation des vecteurs h et o.
      for i = 1 to p do
            hi = oi = T∞
      endfor
// Initialisation des C(i)
      for i = 1 to n do
            C(i) = {0;···;0}
      endfor

// Calcul des vecteurs de communications
      for i = n downto 1 do
            for k = p downto 1 do
                  kC^i_k = min(ok − wk − ck,hk − ck)
                  for j = k-1 downto 1 do
                        kC^i_j = min(kC^i_{j+1} − cj,hj − cj)
                  endfor
                  if C(i) ≺ kC(i) then C(i) = kC(i)
            endfor
            P(i) = length(C(i)) // Placement des tâches
            T(i) = o_P(i) − w_P(i)
            o_P(i) = T(i)
            for k = 1 to P(i)
                  hk = C^i_k
            endfor
      endfor

// Décalage temporel
      for i = n downto 1 do
            T(i) = T(i) − C^1_1
            for k = P(i) downto 1 do
                  C^i_k = C^i_k − C^1_1
            endfor
      endfor

return C,P and T
```

FIG. 5.3 – L'algorithme en pseudo-code.

au temps nécessaire pour effectuer toutes les tâches séquentiellement sur le premier processeur. L'ordonnancement va être ensuite calculé pour finir à cette date, puis on décalera l'échelle des temps pour faire coïncider l'origine des temps avec l'émission de la première tâche. On initialise ensuite h et o avec la valeur $(T_\infty, \ldots, T_\infty)$ puisqu'au départ aucune tâche n'est placée. La dernière initialisation est celle des vecteurs de communications de chaque tâche. Comme par la suite on prendra le plus grand parmi tous les vecteurs possibles, l'initialisation se fait avec le plus petit vecteur possible.

La dernière notation importante pour décrire l'algorithme est la notation du processeur destination. On note ${}^kC_l^i$ la date d'envoi de la tâche i par le processeur l si la tâche i va s'exécuter sur le processeur k. Cette notation permet de comparer entre eux plusieurs vecteurs de communications de la tâche i suivant le processeur où elle sera allouée. Par extension de la notation, on écrira ${}^kC(i)$ le vecteur complet tel que la tâche i est allouée sur le processeur k.

La partie principale de l'algorithme est constituée de deux boucles, l'une sur les tâches et l'autre sur les processeurs. Pour chaque tâche on essaie successivement de placer la tâche sur les p processeurs en créant à chaque fois le plus grand vecteur de communication possible étant données les tâches suivantes. On place effectivement la tâche sur le processeur qui maximise ce vecteur de communications suivant l'ordre défini précédemment. La dernière petite procédure effectue le décalage temporel nécessaire pour mettre le temps à 0 lors de l'émission de la première tâche.

La complexité de cet algorithme est de $O(np^2)$, la comparaison de vecteurs étant en $O(p)$.

5.3 Propriétés

Avant de nous lancer dans la preuve de l'optimalité de l'algorithme, il est intéressant de montrer quelques propriétés sur la structure des solutions que l'on construit.

Le premier lemme concerne les vecteurs de communications. On peut montrer que si l'on considère deux vecteurs de communications maximaux pour l'allocation d'une tâche sur deux processeurs différents et qu'on leur retire leurs q premiers éléments, alors l'ordre sur les deux vecteurs tronqués reste le même. L'idée sous-jacente est présentée dans la figure 5.4.

Lemme 11 *Soit h une enveloppe, o une occupation, k et l deux processeurs et i une tâche. Si ${}^kC(i) \prec {}^lC(i)$ alors pour tout $q \leq min(k,l)$ on a $\{{}^kC_q^i; \cdots ; {}^kC_k^i\} \prec \{{}^lC_q^i; \cdots ; {}^lC_l^i\}$.*

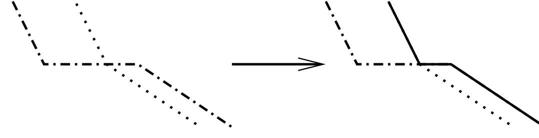

FIG. 5.4 – On peut toujours éviter de croiser les communications.

Les deux processeurs k et l étant différents, les deux vecteurs sont de tailles différentes. Si l'un des deux vecteurs est préfixe de l'autre, alors le lemme est vrai. Sinon soit r le plus petit indice pour lequel il y a une différence. ${}^kC_r^i < {}^lC_r^i$ puisque ${}^kC(i) \prec {}^lC(i)$.

Par construction si r est plus petit que k et l la différence entre ${}^kC_r^i$ et ${}^lC_r^i$ vient d'une différence entre $\min({}^kC_{r+1}^i - c_r, h_r - c_r)$ et $\min({}^lC_{r+1}^i - c_r, h_r - c_r)$, donc entre ${}^kC_{r+1}^i$ et ${}^lC_{r+1}^i$. Cette différence existe donc jusqu'à $r = min(k, l)$.

Il n'existe donc pas d'indice q plus grand que r tel que ${}^kC_q^i \geq {}^lC_q^i$, ce qui conclut la preuve. □

Le deuxième lemme montre que la structure de la solution pour une chaîne est comparable à la structure d'une solution sur un problème restreint.

Lemme 12 *Soit $(c_i)_{i \in 1..p}$, $(w_i)_{i \in 1..p}$ une chaîne de processeurs, et n tâches à ordonnancer sur cette chaîne. Soit n' le nombre de tâches telles que $P(i) \geq 2$ après notre algorithme. L'ordonnancement de ces n' tâches sur la sous-chaîne $(c_i)_{i \in 2..p}$, $(w_i)_{i \in 2..p}$ est le même que celui produit par notre algorithme si l'on lui fournit en entrée n' et la chaîne $(c_i)_{i \in 2..p}$, $(w_i)_{i \in 2..p}$ à un décalage temporel près.*

Plus formellement soit $(\sigma(i))_{i \in 1..n'}$ les n' tâches telles que $P(\sigma(i)) \geq 2$, ($\sigma(i)$ étant une fonction croissante de $1..n'$ dans $1..n$), et soit $T_{shift} = C_2^{\sigma(1)}$, si l'on note avec un chapeau toutes les fonctions concernant le nouveau problème de n' tâches sur $(c_i)_{i \in 2..p}$, $(w_i)_{i \in 2..p}$ on a :

$$\begin{aligned} \forall i \in 1..n' \quad & \hat{P}(i) = P(\sigma(i)) \\ \forall i \in 1..n' \quad & \hat{T}(i) = T(\sigma(i)) - T_{shift} \\ \forall i \in 1..n' \ \forall q \in 2..\hat{P}(i) \quad & \hat{C}_q^i = C_q^{\sigma(i)} - T_{shift} \end{aligned}$$

Par construction de notre algorithme, la valeur de ${}^kC_2^i$ ne dépend pas de la valeur de ${}^kC_1^i$ puisque la construction du vecteur se fait de k vers 1. Nous avons montré dans le lemme précédent que si ${}^kC(i) \prec {}^lC(i)$ alors $\{{}^kC_2^i, \cdots, {}^kC_k^i\} \prec \{{}^lC_2^i, \cdots, {}^lC_l^i\}$. Les tâches placées sur le premier processeur ne changent ni l'enveloppe ni l'occupation des processeurs situés après. L'exécution sur la chaîne $(c_i)_{i \in 2..p}$, $(w_i)_{i \in 2..p}$ est donc identique à l'exécution

sur la chaîne $(c_i)_{i\in 1..p}$, $(w_i)_{i\in 1..p}$ pour les n' tâches allouées sur les processeurs autres que le premier. Il n'y a que les références temporelles qui changent.□

5.4 Optimalité

Théorème 4 *L'algorithme présenté en figure 5.3 fournit une solution optimale.*

La preuve des cas limites comme $p = 1$ ou $n = 1$ est simple. Pour $p = 1$ notre algorithme charge le processeur sans temps morts et pour $n = 1$ notre algorithme teste toutes les possibilités avant de choisir une de celles où la date d'émission par le maître est la plus tardive.

La preuve générale de ce théorème est faite par l'absurde. On suppose qu'il existe une chaîne pour laquelle il existe un nombre de tâches tel que notre algorithme soit moins bon que l'optimal. Sans perte de généralité on peut considérer une des chaînes les plus petites telles que notre algorithme ne soit pas optimal ; pour cette chaîne on peut également considérer le plus petit nombre de tâches tel que notre algorithme ne soit pas optimal.

Soit $(c_i)_{i\in 1..p}$, $(w_i)_{i\in 1..p}$ cette chaîne et n ce nombre de tâches. Comme nous avons considéré la plus petite chaîne telle que notre algorithme ne soit plus optimal, notre algorithme est optimal sur $(c_i)_{i\in 2..p}$, $(w_i)_{i\in 2..p}$ quelque soit le nombre de tâches. De même, sur $(c_i)_{i\in 1..p}$, $(w_i)_{i\in 1..p}$ notre algorithme est optimal pour $n-1$ tâches.

L'idée principale de la preuve est d'étudier les différents cas possibles quand la première tâche de l'ordonnancement optimal est enlevée. On se retrouve avec un ordonnancement de $n-1$ tâches, donc nécessairement pas plus rapide que celui de notre algorithme. La comparaison de cet ordonnancement avec celui produit par notre algorithme amène une contradiction soit sur le fait que p est la plus petite taille de chaîne posant problème, soit que n est le plus petit nombre de tâches pour lequel il y a un problème.

Pour la preuve formelle, nous allons indicer à gauche les notations précédentes pour indiquer de quel cadre elles proviennent. Ainsi ${}_{opt(n)}C(i)$ est le vecteur de communication de la tâche i dans la solution optimale pour n tâches, et ${}_{alg(n)}C(i)$ est le vecteur de communication de la tâche i dans la solution optimale pour n tâches.

Par hypothèse $alg(n)$ n'est pas optimal donc ${}_{opt(n)}T_{max} < {}_{alg(n)}T_{max}$. De plus $alg(n-1)$ l'est, donc si l'on enlève une tâche de la solution optimale on obtient : ${}_{opt(n)}T_{max} - {}_{opt(n)}C_1^2 > {}_{alg(n-1)}T_{max} = {}_{alg(n)}T_{max} - {}_{alg(n)}C_1^2$. Ce qui implique ${}_{alg(n)}C_1^2 > {}_{opt(n)}C_1^2$. Par ailleurs on a ${}_{opt(n)}C_1^2 \geq c_1$ puisque les liens

sont exclusifs. Donc le lien du premier processeur est inoccupé pendant un certain laps de temps dans la solution fournie par notre algorithme entre la première et la deuxième tâche.

Nous allons maintenant considérer deux cas :

- ${}_{alg(n)}P(1) = 1$. Puisqu'il y a un temps d'inactivité du premier lien après la transmission de la tâche, on avait nécessairement $o_1 - w_1 - c_1 < h_1 - c_1$ lorsque ${}^{1}_{alg(n)}C^1_1$ a été calculé, ce qui signifie que le processeur 1 est pleinement chargé par notre algorithme durant les ${}_{alg(n)}T_{max}$ unités de temps. Comme l'optimal est plus rapide, il ne peut pas faire autant de tâches sur le premier processeur. Donc il en fait plus sur la sous-chaîne $(c_i)_{i \in 2..p}$, $(w_i)_{i \in 2..p}$. En ${}_{alg(n)}T_{max} - c_1 - 1$ unités de temps, notre algorithme n'est donc pas optimal sur la sous-chaîne de taille $p - 1$, ce qui contredit la minimalité de p.
- ${}_{alg(n)}P(1) > 1$. La tâche n'a donc pas pu être placée sur le premier processeur ce qui veut à nouveau dire qu'il est pleinement chargé. Donc à nouveau sur la sous-chaîne $(c_i)_{i \in 2..p}$, $(w_i)_{i \in 2..p}$ l'optimal fait au moins autant de tâches que notre algorithme en moins de temps. En effet le fait que le premier lien soit inactif après l'envoi de la première tâche montre que ${}_{alg(n)}C^1_2 = c_1$ (sinon la communication aurait été retardée). Notre algorithme utilise donc ${}_{alg(n)}T_{max} - c_1$ unités de temps alors que sur la sous-chaîne l'optimal en utilise au plus ${}_{opt(n)}T_{max} - c_1$. Cela contredit donc l'optimalité sur la sous-chaîne.

□

5.5 Conclusion

Le problème d'ordonnancer des tâches identiques sur des chaînes de processeurs identiques est donc polynomial en p et en n, avec un algorithme relativement simple. L'étape suivante est la résolution du même problème avec des arbres de processeurs. Les travaux publiés sur les graphes fork étant de conception très différente la jonction des deux semble être une tâche ardue. On verra dans le chapitre suivant qu'elle est possible pour une sous-classe d'arbre. Le chapitre d'après montre que toute tentative d'algorithme optimal polynomial pour le problème général des arbres est vouée à l'échec à moins que P ne soit égal à NP.

Chapitre 6

Réseau en Pieuvre

Nous allons dans ce chapitre présenter un algorithme pour le problème précédent sur un réseau plus complexe appelé «pieuvre».

On appelle «pieuvres» les arbres n'ayant qu'un seul nœud d'arité supérieure ou égale à deux. Ce nœud est le nœud maître de notre distribution de tâches. Un exemple simple est dessiné dans la figure 6.1. Comme dans le cas de la chaîne, un ordonnancement réalisable sur une pieuvre est un ordonnancement dans lequel les liens de communications et les processeurs ne sont pas utilisés par plus d'une tâche à la fois. La seule différence se situe au niveau du processeur maître, en effet la contrainte «1-port» implique le fait que le maître ne peut envoyer de tâches qu'à un seul de ses fils.

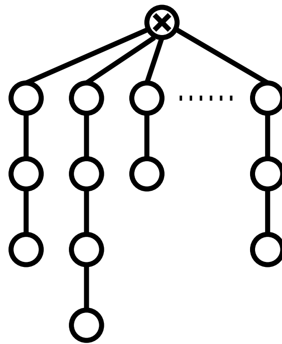

Fig. 6.1 – Une pieuvre

Pour obtenir un algorithme d'ordonnancement sur les pieuvres, il est intéressant de regarder ce qui existe déjà pour des formes plus simples de réseau. C'est en combinant un algorithme pour les arbres à un seul niveau (qui sont un cas particulier de pieuvre où tous les processeurs sont liés au maître),

et l'algorithme du chapitre précédent que nous avons obtenu l'algorithme présenté ici.

Rappelons d'abord brièvement l'algorithme d'ordonnancement sur les arbres à un niveau proposé par [BLR02a]. Cet algorithme ordonnance de façon optimale les tâches identiques sur un arbre hétérogène avec la politique suivante : privilégier les liens de communications les plus rapides (ce sont ceux qui bloquent le maître le moins longtemps) et ordonner les tâches suivant le temps d'exécution sur le processeur cible.

La première étape de cet algorithme est de créer pour chaque nœud (c_i,w_i) plusieurs nœuds à usage unique (voir la figure 6.2 avec $m_i = \max(c_i, w_i)$). Ces nœuds à usage unique ont le même lien de communication que le nœud d'origine, mais des vitesses de calcul différentes pour prendre en compte le fait que le nœud d'origine ne pouvait faire deux tâches en même temps. Les nœuds à usage unique sont ensuite sélectionnés pour l'ordonnancement en les triant par temps de communication croissant et insérés dans un ordonnancement par temps d'exécution décroissant. L'algorithme s'arrête lorsque l'insertion n'est plus possible, c'est-à-dire lorsque le temps total passé à envoyer les tâches ayant un temps de calcul plus grand plus le temps passé à envoyer et calculer la tâche courante dépasse le temps T_{lim} imparti à l'ordonnancement.

Une propriété intéressante pour conclure sur l'optimalité de l'algorithme est que chaque ordonnancement peut être transformé en un ordonnancement où les tâches sont envoyées par le maître triées par temps de calcul décroissant.

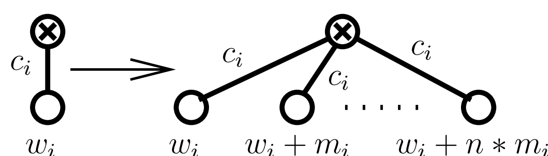

FIG. 6.2 – Transformation d'un nœud simple en nœud multiple.

Tout ceci est décrit plus longuement, avec le calcul précis de la complexité dans l'article original [BLR02a] que le lecteur intéressé est invité à consulter[1].

[1]La version française est disponible dans le chapitre 3 de la thèse d'Arnaud Legrand [Leg03]

6.1 Présentation de l'algorithme

Comme dans le cas de l'algorithme pour les arbres à un niveau, nous allons considérer qu'une des entrées du problème est le temps T_{lim} imparti à l'ordonnancement. Les notations sont reprises du chapitre précédent.

L'algorithme de chaîne tel qu'il est présenté dans le chapitre précédent n'utilise pas une limite de temps comme T_{lim}, mais un nombre de tâches. On peut facilement passer en entrée un temps limite en plus du nombre n de tâches, et ajouter des tâches jusqu'à ce que la limite de temps soit atteinte ou que n tâches aient été ordonnancées. Il suffit pour cela de changer T_∞ en T_{lim}, de changer la boucle `for` sur les vecteurs de communications en `while` et d'arrêter le calcul quand la limite ou le nombre de tâches est atteint.

L'algorithme sur les arbres à un niveau (qui sont des cas particuliers de pieuvres) reposait sur une transformation de chaque noeud en noeuds à usage unique. Cette décomposition permettait de décider quel noeud aurait des tâches à exécuter, ainsi que l'ordre dans lequel ces tâches seraient exécutées. Avec une chaîne à la place de chaque processeur dans l'arbre à un niveau, la puissance de calcul est moins facilement transformable en noeuds à usage unique. Cette transformation peut cependant être déduite de l'ordonnancement optimal sur la chaîne. Soit C_1^i (pour i de 1 à n) les différents temps d'émission des tâches par le processeur maître dans l'ordonnancement optimal pour la chaîne seule et c_1 le temps de communication du maître au premier esclave de la chaîne. On peut remplacer la chaîne par n processeurs à usage unique ayant tous un lien c_1 et ayant un temps d'exécution de $T_{lim} - C_1^i - c_1$. Cette transformation nous donne un arbre à un niveau pour lequel l'ordonnancement optimal de ν tâches, avec $\nu \in [|1; n|]$, a le même temps d'exécution que l'optimal sur la chaîne pour le même nombre de tâches.

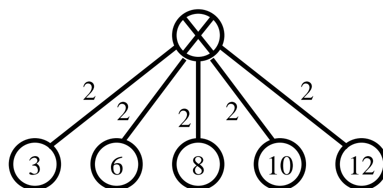

FIG. 6.3 – Transformation de la figure 2.1.

Le résultat de cette transformation appliquée à l'exemple de la figure 2.1 est décrit dans la figure 6.3. Dans cet exemple, la tâche qui était faite par le deuxième processeur correspond au processeur de temps d'exécution 8.

Toutes les chaînes reliées à un même maître peuvent être transformées de la même façon. Pour un temps limite T_{lim} donné, on calcule pour chaque chaîne un ordonnancement optimal, puis l'on crée comme indiqué un arbre à un niveau constitué de processeurs à usage unique.

Ces arbres à usage unique sont groupés, et l'on peut ainsi appliquer l'algorithme polynomial qui concerne les arbres à un niveau. Cet algorithme nous permet de choisir quels sont les noeuds qui devront effectivement être utilisés, c'est-à-dire où les tâches devront être réellement exécutées. Cette conversion des noeuds sélectionnés vers un ordonnancement de pieuvre est une correspondance directe entre les noeuds et le chemin dont ils sont issus.

Plus formellement, l'algorithme s'écrit de la façon suivante :

```
(1) Étant donnés T_lim, n et une pieuvre
(2) Pour chaque chaîne de la pieuvre
          calculer n, C, P, et T
(3) Créer l'arbre à un niveau associé
(4) Calculer l'optimal sur l'arbre à un niveau
(5) Convertir les résultats en ordonnancement de pieuvre
```

Théorème 5 *L'algorithme sur les pieuvres est polynomial en n le nombre de tâches et en p le nombre de processeurs.*

La complexité de la ligne 2 est $\sum_c np_c^2$, où p_c est la longueur de la chaîne c. On peut majorer la somme des carrés par le carré de la somme : $\sum_c np_c^2 < np^2$. La ligne 3 est une simple réécriture des résultats obtenus à l'étape précédente (il y a moins de p chaînes, donc moins de np noeuds créés). La ligne 4 est quadratique en le nombre de noeuds à usage unique, sa complexité est donc inférieur à n^2p^2. La dernière ligne correspond à une simple réécriture des résultats, elle est donc linéaire en la taille du résultat. La complexité globale est donc de l'ordre de $O(n^2p^2)$. □

Lemme 13 *Un ordonnancement réalisable pour l'arbre à un niveau peut être transformé en un ordonnancement réalisable pour la pieuvre.*

À chaque noeud de l'arbre à un niveau est associé une tâche et son allocation dans une chaîne de la pieuvre. Pour une chaîne donnée, nous pouvons considérer toutes les tâches ordonnancées sur cette chaîne qui ont été choisies. L'ordonnancement de chaîne obtenu en ne considérant que ces tâches est l'ordonnancement optimal privé de quelques tâches. La seule différence est que le temps d'émission par le maître peut être différent dans le cas de

l'arbre à un niveau, puisque les communications ont pu être faites plus tôt. Il ne peut cependant pas y avoir d'inversion dans l'ordre d'émission des tâches puisqu'elles sont triées par temps d'exécution décroissant par l'algorithme d'arbre.

La faisabilité est déterminée par le non recouvrement des communications vers les différents fils d'un même noeud, et le non recouvrement des exécutions des tâches. Le non recouvrement des communications est assuré par l'algorithme d'arbre à un niveau, puisque le noeud maître est le seul noeud à avoir plusieurs fils, et le non recouvrement des temps de calcul est garanti par l'algorithme de chaîne pour chacune des chaînes. □

Lemme 14 *Un ordonnancement de pieuvre peut être transformé en ordonnancement d'arbre à un niveau.*

Ce lemme peut paraître surprenant à première vue puisque l'arbre à un niveau a été construit à partir d'ordonnancements particuliers sur les différentes chaînes qui composent la pieuvre, et que l'on va montrer ici que n'importe quel ordonnancement de pieuvre peut être lié à un ordonnancement sur l'arbre plat.

Une propriété intéressante de notre algorithme de chaîne est que l'ordonnancement optimal pour n tâches est construit itérativement à partir de la solution pour $n-1$ tâches. Pour tout nombre i de tâches inférieur ou égal à n, l'ordonnancement des i dernières tâches est un ordonnancement optimal en temps.

Quand on considère un ordonnancement quelconque sur la pieuvre, chaque tâche ordonnancée sur une chaîne est donc reçue par le premier processeur de la chaîne plus tôt ou à la même date que dans l'ordonnancement généré par notre algorithme, donc suffisamment tôt pour pouvoir être liée au noeud à usage unique correspondant. Ceci est vrai pour toutes les chaînes de la pieuvre. □

Théorème 6 *L'ordonnancement obtenu par notre algorithme de pieuvre est optimal.*

La preuve de ce théorème est le simple enchaînement des lemmes précédents. Le lemme 14 montre que chaque ordonnancement de pieuvre peut être transformé en un ordonnancement pour l'arbre plat de même durée. Cela vaut donc pour un ordonnancement optimal sur la pieuvre en T_{lim} unités de temps. Cet ordonnancement obtenu par transformation sur l'arbre plat est réalisable, mais pas forcément optimal. Comme nous obtenons l'optimal sur l'arbre plat, nous faisons au moins autant de tâches que l'ordonnancement optimal considéré sur la pieuvre. En revenant à un ordonnancement de pieuvre

comme indiqué dans le lemme 13, on produit un ordonnancement réalisable de pieuvre qui contient au moins autant de tâches qu'un ordonnancement optimal. □

6.2 Exemple

Pour illustrer le déroulement de l'algorithme, voici un exemple détaillé sur une petite pieuvre avec T_{lim} égal à 20 unités de temps. La pieuvre est présentée dans la figure 6.4. Les ordonnancements optimaux fournis par l'algorithme de chaînes pour les deux chaînes sont respectivement dans les figures 6.5 et 6.6. L'arbre à un niveau qui en découle est donné dans la figure 6.7, avec les noeuds choisis par l'algorithme d'arbre soulignés en pointillés. On peut remarquer que l'algorithme sélectionne en priorité les noeuds qui ont un faible temps de communication.

L'ordonnancement final sur la pieuvre est enfin présenté dans la figure 6.8. Les tâches étant triées par temps décroissant d'exécution dans l'ordonnancement d'arbre à un niveau, les tâches de la branche gauche sont intercalées avec celles de la branche de droite.

6.3 Conclusion

Les structures plus générales posent tout de suite des problèmes trop difficiles pour être résolus par une approche directe comme on le verra dans le chapitre suivant. Il faut donc envisager des heuristiques, ou changer légèrement d'objectif pour avoir des résultats utilisables sur des clusters ayant une structure plus complexe ou sur des réseaux locaux de machines de bureaux qui ne sont généralement pas organisés au départ pour faire du calcul parallèle.

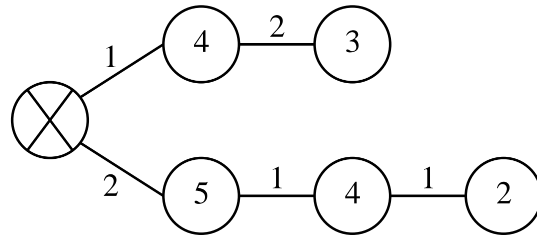

FIG. 6.4 – Une petite pieuvre.

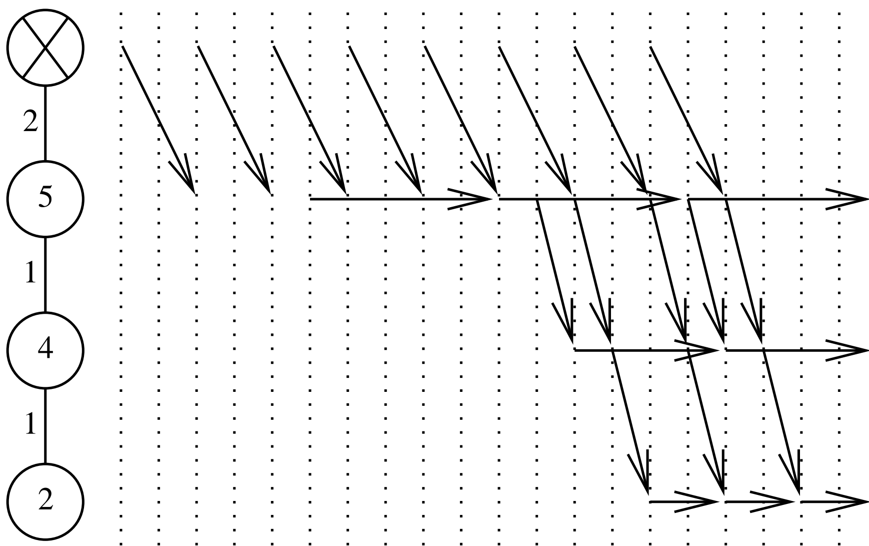

FIG. 6.5 – Ordonnancement optimal sur la chaîne de gauche.

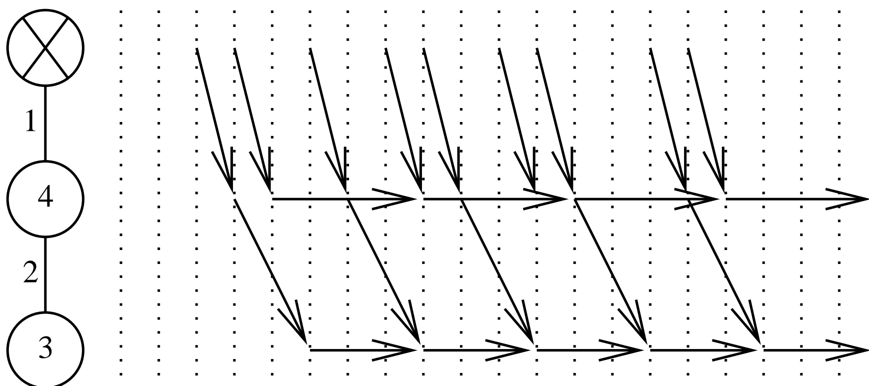

FIG. 6.6 – Ordonnancement optimal sur la chaîne de droite

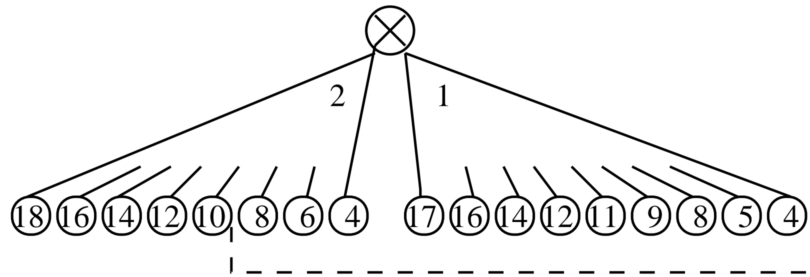

Fig. 6.7 – Arbre à un niveau induit par la transformation.

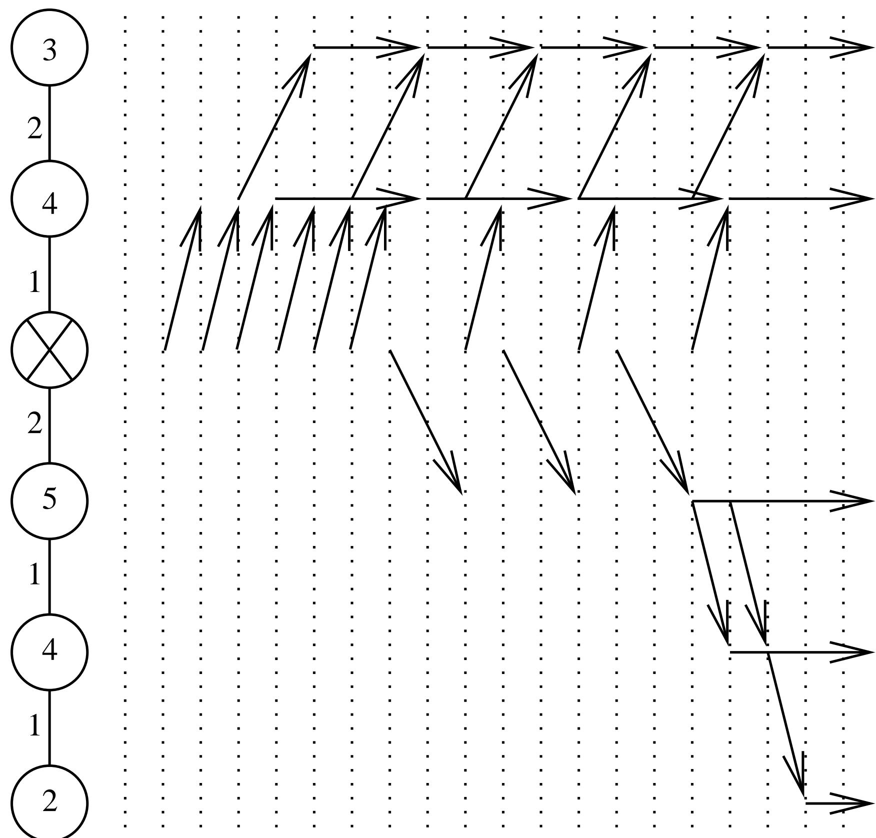

Fig. 6.8 – Ordonnancement optimal sur la pieuvre.

Chapitre 7

Arbres

Nous utiliserons à nouveau dans ce chapitre la représentation d'ordonnancements introduite dans les chapitres précédents. Sur la figure 7.1, l'ordonnancement d'un arbre est décrit. Les lignes verticales pointillées représentent les unités de temps, les traits pointillés horizontaux représentent les différents processeurs, et les flèches représentent les communications et les calculs de tâches. Utiliser cette représentation pour des arbres quelconques peut s'avérer fastidieux. Heureusement les arbres que l'on va utiliser dans ce chapitre restent suffisamment simples.

Les noeuds et les arcs de l'arbre sont valués, les nombres représentant le temps nécessaire pour respectivement calculer ou transmettre une tâche.

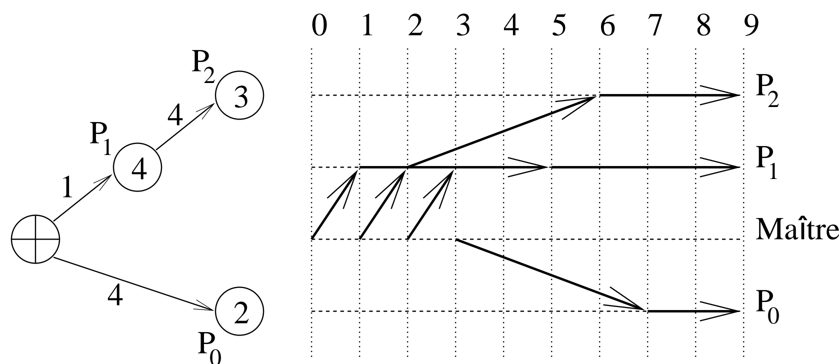

FIG. 7.1 – Un arbre (à gauche) et son ordonnancement (à droite)

Dans les chapitres précédents, il était implicitement admis que les noeuds avaient une mémoire suffisamment importante pour contenir tous les messages qu'on leur transmettait. Cette hypothèse est réaliste dans la plupart des

problèmes paramétriques, les données nécessaires aux calculs étant petites. Dans ce chapitre cette hypothèse n'est même plus nécessaire, une mémoire ne pouvant contenir qu'une seule tâche suffit.

7.1 Description des problèmes

7.1.1 3-Partition

Nous allons utiliser ici une version légèrement modifiée du problème classique 3-partition [GJ79] comme problème de référence pour notre démonstration de NP-complétude. Nous allons donc rappeler dans cette section les notations et les modifications faites au problème classique.

Définition 12 *3-partition*
Soit S et n deux entiers, et soit $(y_i)_{i\in 1..3n}$ une suite de $3n$ entiers tels que pour tout i, $\frac{S}{4} < y_i < \frac{S}{2}$.

La question du problème de décision 3-partition est «Peut-on partitionner l'ensemble des y_i en n triplets tels que la somme de chaque triplet soit exactement S ?».

Théorème 7 *Le problème de décision 3-partition est NP-complet au sens fort.*

Dans toute la suite, nous utiliserons les rationnels x_i définis par $x_i = \frac{1}{4}S + \frac{y_i}{8}$, où les y_i sont issus d'une instance de 3-partition. Si un triplet de y_i a pour somme S, le triplet de x_i correspondant a pour somme $\frac{7S}{8}$, et réciproquement. Une partition des y_i en triplets de somme S est donc équivalente à une partition des x_i en triplets de somme $\frac{7S}{8}$.

Cette modification permet en fait de garantir que les x_i sont contenus dans un intervalle plus petit que celui qui contenait les y_i. En effet les x_i sont compris (strictement) entre $\frac{9S}{32}$ et $\frac{5S}{16}$.

7.1.2 POTMEAH[1]

Nous allons maintenant présenter formellement le problème de l'ordonnancement de tâches maître-esclave sur un arbre de processeurs hétérogènes.

Définition 13 *POTMEAH*

[1]**P**roblème de l'**O**rdonnancement de **T**âches **M**aître-**E**sclave sur un **A**rbre de processeurs **H**étérogènes.

Soit T=(V,E) un arbre. Soit v_0 un noeud spécial appelé «Maître». Pour tous les autres noeuds v_i soit w_i le temps de calcul d'une tâche. À chaque arête e_i est associé le temps c_i de transmission des données de cette tâche. Finalement soit n un nombre de tâches et D une date limite.

La question associée au problème de décision POTMEAH est : «Est-il possible d'ordonnancer n tâches en temps D sur l'arbre T ?».

7.2 Réduction

L'arbre utilisé pour la réduction est présenté dans la figure 7.2. L'instance du problème POTMEAH que l'on va considérer est d'ordonnancer $4n$ tâches avec pour date limite $D = E + nS + \frac{S}{4}$, où E est un temps énorme par rapport au reste (pour la démonstration on fixe $E = (n+1)S$ pour garder des valeurs polynomiales par rapport aux données d'entrée). Le noeud maître est connecté à un noeud de distribution, auquel il envoie une tâche toutes les $\frac{S}{4}$ unités de temps. Le noeud de distribution est trop lent pour pouvoir calculer ne serait-ce qu'une tâche ($2E > D$). Il est donc obliger de distribuer les tâches aux noeuds inférieurs. Chaque noeud inférieur ne peut calculer qu'une seule tâche puisque les temps de calcul sont tous supérieurs ou égaux à E (et $2E > D$).

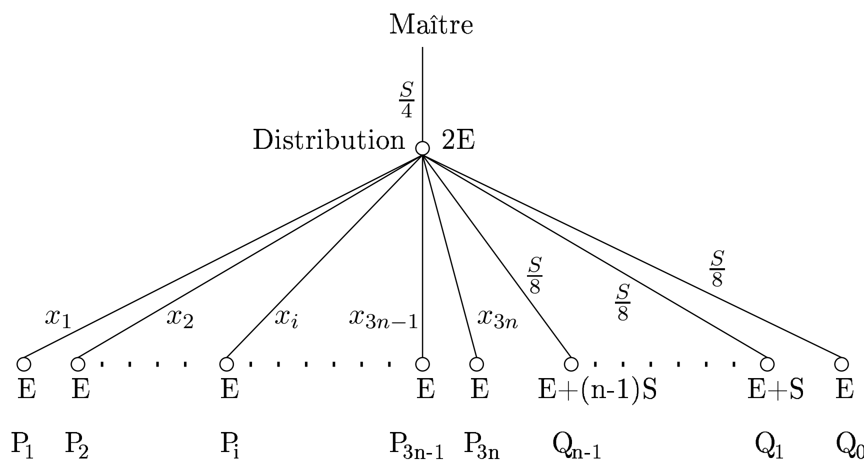

FIG. 7.2 – Arbre utilisé dans la réduction

Cette structure est construite pour reproduire le problème de 3-partition dans le cadre de l'ordonnancement de tâches maître esclave. Dans la réduction à 3-partition utilisée dans le chapitre 2, les entiers à regrouper par triplets étaient remplacés par des tâches de temps d'exécution différents. Ici

ce mécanisme n'est plus possible puisque les tâches sont identiques. Il faut donc trouver une autre façon d'avoir des entités valuées qui se partagent une ressource découpée en n parties de taille S. Comme précédemment cette ressource est le temps, mais cette fois le seul moyen de partager le temps entre plusieurs éléments concurrents est d'utiliser la contrainte 1-port. En effet, seules les communications sont en concurrence dans ce modèle. C'est donc le lien de communication sortant du noeud de distribution qui permet de recréer les x_i. Pour créer les n intervalles de taille S à remplir par les x_i il a fallu ajouter les noeuds Q_i et le lien de communication de temps $\frac{S}{4}$ reliant le maître au noeud de distribution.

7.2.1 Faire un ordonnancement à partir de 3-partition

Montrons d'abord qu'à partir d'une solution de 3-partition, nous pouvons construire un ordonnancement de $4n$ tâches faisable en temps $\frac{S}{4} + nS + E$.

Pour simplifier la lecture, les x_i seront ordonnés comme dans la solution du problème de 3-partition (c'est-à-dire $x_{3j+1} + x_{3j+2} + x_{3j+3} = \frac{7}{8}S$ pour $j \in 0..(n-1)$). L'ordonnancement est le suivant :

1. Utiliser le lien entre le maître et le distributeur le plus possible, en envoyant une tâche toute les $\frac{S}{4}$ unités de temps. Les $4n$ tâches sont donc envoyées en nS unités de temps.
2. Le distributeur envoie dès que possible les tâches dans l'ordre d'arrivée selon le principe suivant :
   - La tâche $4j+1$ sur le lien x_{3j+1} (pour $j \in 0..(n-1)$)
   - La tâche $4j+2$ sur le lien x_{3j+2} (pour $j \in 0..(n-1)$)
   - La tâche $4j+3$ sur le lien x_{3j+3} (pour $j \in 0..(n-1)$)
   - La tâche $4j+4$ sur le lien $\frac{S}{8}$ allant vers le processeur Q_{n-1-j} (pour $j \in 0..(n-1)$)

Le diagramme d'un exemple avec $n = 2$, $7S/8 = 15$ et les entiers $(x_1, x_2, x_3, x_4, x_5, x_6) = (4, 4, 7, 4, 5, 6)$ est donné dans la figure 7.3.

Comme la somme de chaque triplet $(x_{3j+1}, x_{3j+2}, x_{3j+3})$ est exactement $\frac{7S}{8}$, le temps total d'envoi d'un groupe de 4 tâches par le noeud de distribution est exactement S.

Comme les x_i sont plus grands que $\frac{S}{4}$, pendant que le distributeur envoie une tâche x_i il reçoit au moins une tâche du maître. Le groupe de 4 tâches étant transmis en S unités de temps, la transmission de la tâche vers un processeur Q_i se termine en même temps que l'arrivée de la tâche suivante sur le noeud de distribution. Le lien sortant du distributeur est donc rempli sans temps mort.

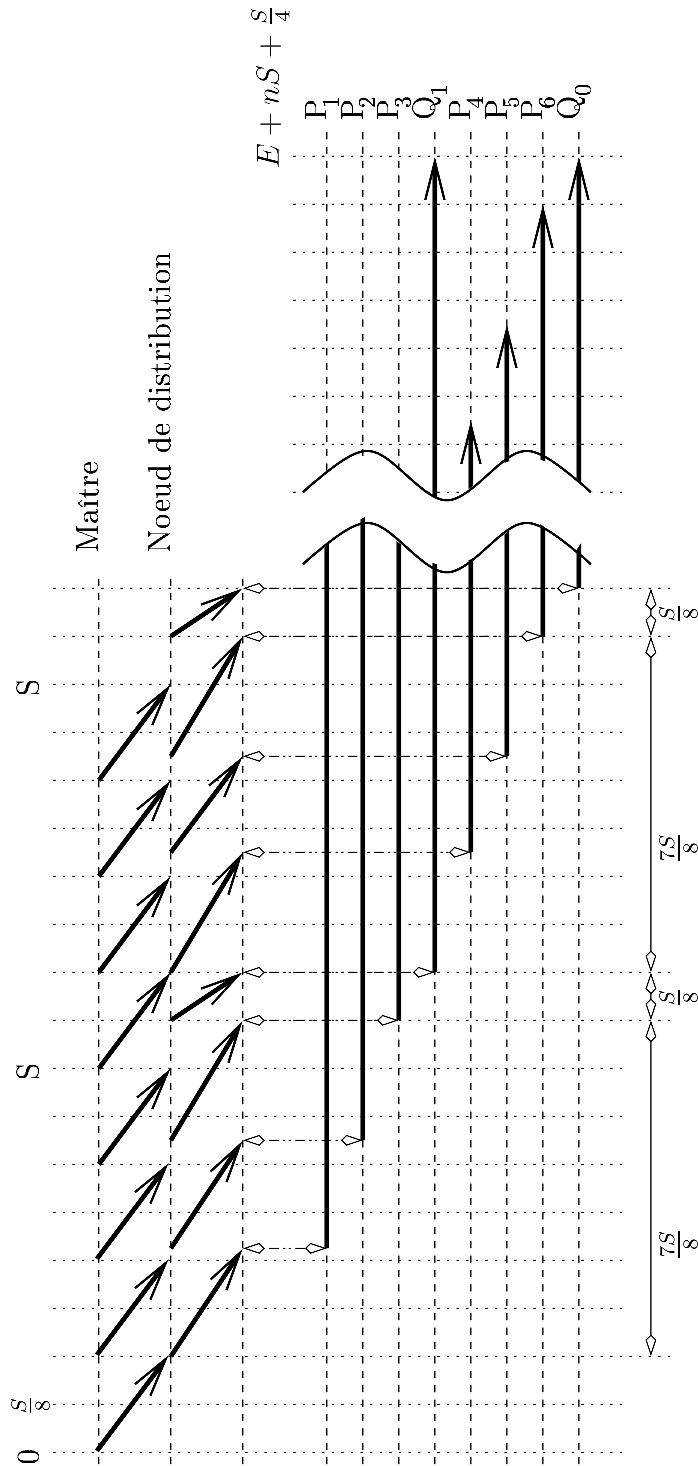

FIG. 7.3 – Ordonnancement de 8 tâches à partir d'une solution de 3-partition

La première tâche mettant $\frac{S}{4}$ unités de temps pour parvenir au distributeur, la dernière tâche à quitter le noeud distributeur arrive sur son noeud de calcul au temps $nS + \frac{S}{4}$, ce qui laisse juste le temps pour exécuter la dernière tâche sur le processeur Q_0.

De fait, tous les processeurs Q_i reçoivent leur tâche juste à temps pour finir leur exécution à la date D.

7.2.2 D'un ordonnancement sur arbre à une solution de 3-Partition

Nous allons maintenant montrer que tout ordonnancement de $4n$ tâches en D unités de temps est lié à une solution de 3-partition et que sa structure est similaire à celle présentée dans la figure 7.3.

Comme signalé précédemment, chaque noeud peut calculer au plus une tâche, à l'exception du noeud de distribution et du noeud maître. Étant donnés le nombre de noeud et le nombre de tâches, chaque noeud doit donc être utilisé. De plus, le temps minimal de calcul E et le temps total de communication après le noeud de distribution nS fait qu'il ne doit pas y avoir de temps mort entre deux envois du distributeur.

Montrons d'abord quelques propriétés sur le processeur Q_{n-1} et la tâche qui lui est associée.

Lemme 15 *La tâche associée au processeur Q_{n-1} est une des 4 premières à être envoyées par le noeud maître.*

Le temps de calcul du noeud Q_{n-1} étant de $E + (n-1)S$, la tâche s'exécutant sur ce noeud doit y arriver au plus tard à la date $S + \frac{S}{4}$. La cinquième tâche ne peut y arriver avant la date $\frac{5S}{4} + \frac{S}{8}$, puisqu'il faudrait envoyer 5 tâches du maître puis envoyer la cinquième sur le lien allant au processeur Q_{n-1}. Les tâches suivantes arrivent encore plus tard, la tâche calculée par Q_{n-1} est donc une des 4 premières. □

Lemme 16 *Les trois premières tâches sont allouées à des processeurs P_i.*

Comme on l'a dit plus haut, le lien quittant le distributeur doit être occupé en permanence par des émissions de tâches. Il faut donc toujours qu'il y ait une tâche disponible pour l'émission quand une transmission s'achève. Quand une tâche est envoyée sur un lien de temps x_i, le maître a le temps d'envoyer la suivante au distributeur pendant la transmission.

Par contre, si au moins une des trois premières tâches est envoyée à un processeur Q_i, le temps de communication des trois premières tâches est strictement inférieur à $\frac{S}{8} + \frac{5}{16}S + \frac{5}{16}S = \frac{3}{4}S$. Il y a donc nécessairement un temps

mort pour le lien sortant du distributeur entre la réception de la première tâche et l'émission de la quatrième. Cette pause rend l'ordonnancement de $4n$ tâches en temps D impossible comme nous l'avons déjà souligné. □

Le corollaire immédiat des deux lemmes précédents est que la quatrième tâche est allouée au processeur Q_{n-1}.

Lemme 17 *Les trois premières tâches émises par le distributeur utilisent son lien de communication pendant exactement $\frac{7}{8}S$ unités de temps.*

Le processeur Q_{n-1} ayant un temps d'exécution de $E + (n-1)S$, il doit recevoir sa tâche au plus tard au temps $\frac{5}{4}S$. Ce qui implique que les trois premières tâches sont émises en $\frac{7}{8}S$ unités de temps au plus.

Comme la cinquième tâche n'arrivera sur le distributeur qu'au temps $\frac{5}{4}S$, et que le lien sortant doit être utilisé en permanence, il faut que la tâche émise vers Q_{n-1} occupe le lien jusqu'à cette date. Donc il est nécessaire que les trois premières tâches utilisent le lien pendant au moins $\frac{7}{8}S$ unités de temps. □

Théorème 8 *Ordonnancer $4n$ tâches en un temps $D = \frac{S}{4} + nS + E$ unités de temps permet de reconstruire une instance au problème 3-partition associé.*

Nous venons de montrer que les trois premières tâches émises par le noeud maître formaient un triplet de somme exactement $\frac{7S}{8}$. Cette propriété peut être récursivement étendue aux autres triplets de tâches $4j+1$, $4j+2$ et $4j+3$, les tâches $4j+4$ étant ordonnancées sur les processeurs Q_{n-1-j}. Un ordonnancement de $4n$ tâches en temps D sur l'arbre considéré permet donc de reconstruire une partition de l'ensemble des x_i en triplets de somme $\frac{7}{8}S$. Ces triplets sont une solution du problème de 3-partition associé. □

7.3 Extensions du modèle

Le modèle utilisé jusqu'à présent peut être légèrement modifié pour prendre en compte des détails spécifiques à certaines architectures. Deux modifications classiques que nous étudierons rapidement ici sont d'une part l'ajout de latence pour les communications et d'autre part la prise en compte du modèle 1-port strict, où l'envoi et la réception simultanés sont impossibles.

Ces deux contraintes supplémentaires peuvent rendre les problèmes plus simples ou plus compliqués suivant les cas. Ici on peut étendre la preuve précédente pour montrer que l'ordonnancement sur les arbres reste NP-complet au sens fort.

7.3.1 Avec latences

Dans un modèle avec des latences, il est souvent judicieux de regrouper des communications pour partager le coût de la latence entre plusieurs messages. Pour pouvoir conserver la même architecture de preuve, il va falloir modifier légèrement les temps de communication pour inclure la latence, et montrer que les communications ne peuvent être groupées sans induire de retard dans l'ordonnancement.

Soit l le temps de latence des communications. Dans un premier temps, nous considérons que l est plus petit que $\frac{S}{16}$. Nous pouvons modifier l'arbre présenté dans la figure 7.2 pour soustraire l de tous les temps de communications c_i. Cette modification rend l'ordonnancement présenté dans la figure 7.3 directement réalisable avec les temps de latences.

Il faut donc prouver que ce type d'ordonnancement est toujours le seul possible, c'est-à-dire qu'une solution au problème POTMEAH est toujours liée à une solution du problème de 3-partition associé.

Nous allons maintenant montrer qu'il n'est pas possible de grouper des communications et de rester optimal en temps. Tout d'abord, les communications qui partent du noeud de distribution ne peuvent être groupées entre elles. En effet chaque noeud de calcul n'exécute qu'une seule tâche. Le seul problème est donc au niveau du noeud maître.

Lemme 18 *La première tâche est envoyée seule par le maître.*

Comme précédemment, la somme des temps de communication sur les liens sortant du noeud de distribution est nS. Le plus petit temps d'exécution est E. Donc le début des envois par le distributeur doit se faire au temps $\frac{S}{4}$. Ce qui implique que la première tâche à envoyer est disponible sur le noeud de distribution à cet instant là, donc qu'elle a été envoyée seule par le noeud maître.

Lemme 19 *La deuxième tâche est envoyée seule par le maître.*

La deuxième tâche doit être prête à l'envoi quand le distributeur termine l'émission de la première. Or l'émission de la première par le distributeur dure strictement moins de $\frac{5S}{16}$ unités de temps, puisque cette valeur borne les x_i. Soit j le nombre de tâches envoyées dans le lot contenant la deuxième tâche, on a :

$$l + j(\frac{S}{4} - l) \leq \frac{5S}{16}$$

et l'on peut minorer le terme de gauche par :

$$l + j(\frac{S}{4} - l) = j\frac{S}{4} - (j-1)l > j\frac{S}{4} - (j-1)\frac{S}{16} = j\frac{3S}{16} + \frac{S}{16}$$

On obtient alors une majoration de j :

$$j < \frac{16}{3S}(\frac{5S}{16} - \frac{S}{16}) = \frac{4}{3}$$

Cela montre que la deuxième tâche a été émise seule.

Lemme 20 *La troisième tâche est envoyée seule par le maître.*

On peut reprendre le même calcul que dans le lemme précédent, avec un temps d'émission de la part du distributeur de moins de $\frac{10S}{16}$ cette fois (ce qui correspond à $2x_i$). On a donc :

$$\frac{S}{4} + l + j(\frac{S}{4} - l) \leq \frac{10S}{16}$$

Ce qui avec la même minoration de j nous donne $j < 5/3$.

Lemme 21 *La quatrième tâche est envoyée seule par le maître.*

Le même calcul est encore réutilisable, avec un temps d'émission de la part du distributeur de moins de $\frac{15S}{16}$ (ce qui correspond à $3x_i$), pour atteindre cette fois la majoration $j < 2$. La quatrième tâche est donc également transmise seule.

On ne peut évidemment pas continuer indéfiniment de cette façon, mais heureusement ces quatre lemmes nous suffisent. En effet, avec ces quatre lemmes toutes les propriétés démontrées dans la section 7.2.2 sont à nouveau vraies. On peut donc à nouveau utiliser une récurrence comme dans le théorème 8 pour montrer que toutes les communications sur des liens de temps x_i se font par groupes de trois communications de somme exactement $\frac{7S}{8}$.

La condition préliminaire $l < \frac{S}{16}$ signifie que cette preuve n'est vraie que pour un modèle où la latence est moins grande que la moitié de la durée de la plus petite communication. On peut cependant la modifier pour augmenter le rapport entre la latence et la plus petite communication. En remplaçant[2] chaque noeud de calcul par un graphe *fork*, constitué d'un noeud père ne pouvant calculer ($w_i = 2E$) et de k fils ne pouvant faire qu'une tâche chacun (en réglant finement leur w_i et c_i) et chaque lien de communication c_i par $\frac{c_i - l}{k}$, on peut forcer le groupement des tâches en paquets de taille k entre le distributeur et les noeuds de calcul (pour n'avoir à payer qu'une fois la latence à ce niveau). Pour le lien entre le maître et le distributeur, on démontre des lemmes similaires aux quatre lemmes ci-dessus. Les nouveaux lemmes s'écrivent de la façon suivante :

[2]Les w_i et c_i utilisés ici sont ceux de la preuve originale sans latence.

1. Le premier groupe est composé d'exactement k tâches.
2. Le deuxième groupe a moins de $\frac{4k}{3}$ tâches.
3. Les deuxième et troisième groupes ont à eux deux moins de $\frac{8k}{3}$ tâches.
4. Les groupes deux, trois et quatre ont à eux trois moins de $4k$ tâches.
5. Les groupes deux, trois, quatre et cinq ont ensemble exactement $4k$ tâches.

La latence est ici toujours inférieure à $\frac{S}{16}$, mais le temps de transfert sur le lien le plus rapide ($\frac{S/8-l}{k}$) peut être aussi petit que l'on veut. La preuve complète étant fortement similaire, elle n'est pas reproduite ici.

7.3.2 Avec un seul canal de communication

Dans cette section nous allons étudier le modèle où les noeuds n'ont qu'un seul port de communication, c'est-à-dire qu'ils ne peuvent pas recevoir et envoyer de messages en même temps. Ce modèle est plus réaliste que le précédent car la plupart des ordinateurs ne disposent que d'une seule carte Ethernet. Cette contrainte supplémentaire ne rend pas le problème plus facile pour autant, puisque l'on va montrer qu'il reste NP-difficile au sens fort.

La preuve est très proche de celle présentée dans le cas général, l'arbre issu de la transformation étant légèrement adapté. Une transformation *ad hoc* est présentée dans la figure 7.4. Les noeuds insérés ne calculent pas. Le premier permet de recevoir du maître en permanence et le second de mettre les tâches à disposition du distributeur. Le deuxième noeud est essentiel, puisque les communications entrantes et sortantes sont rarement synchronisées, comme on peut le constater sur le diagramme 7.3.

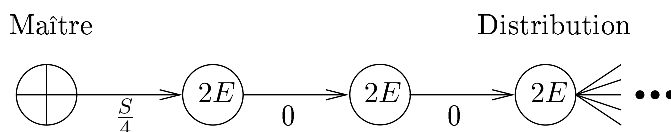

FIG. 7.4 – Arbre avec communications de durée zéro.

Plus généralement, on peut émuler la présence de deux cartes de communications (une entrante et une sortante) en remplaçant chaque noeud par une succession de trois noeuds avec des liens très rapides Cette transformation avec des liens de temps nul est présentée dans la figure 7.5.

Cependant, l'utilisation de liens de temps nul est discutable. En effet le temps de transmission des liens étant nul, plusieurs communications peuvent être effectuées les unes à la suite des autres dans le même instant. Du point

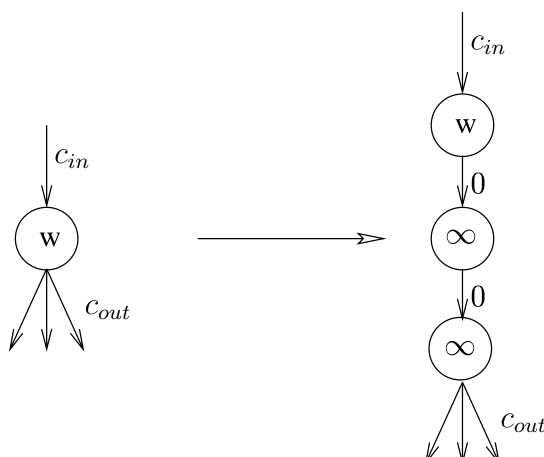

FIG. 7.5 – Transformation d'un noeud.

de vue macroscopique le comportement est celui d'un lien sur lequel le parallélisme de communications est autorisé (et le temps de transfert nul).

C'est pour éviter ce genre de problème que dans la preuve principale les liens vers les processeurs Q_i ont un temps non nul (la première écriture avait été faite avec des temps nuls). On peut ici à nouveau modifier légèrement l'arbre pour obtenir un arbre sans liens de temps nul. Il suffit en fait de choisir pour ces liens un temps petit que l'on retranche au lien précédent et au lien suivant. Le nouvel arbre obtenu est présenté dans la figure 7.6. Dans cette figure les liens z_i sont égaux à $x_i - \frac{S}{32}$ pour tout i.

Dans la figure 7.7 est représenté le diagramme de l'exécution sur l'arbre modifié de la figure 7.6. La preuve de la NP-complétude est très similaire aux précédentes. Il faut montrer ici que les communications entrant et sortant d'un noeud ne sont jamais en conflit.

En fait la preuve se fait uniquement sur le deuxième noeud après le maître, puisque l'on considère que le premier reçoit et émet directement les tâches, et que le distributeur ne va chercher une tâche qu'avant d'envoyer vers ses fils. Les communications entrantes sur ce noeud ont lieu pendant les instants $i\frac{S}{4} - \frac{S}{32}$ à $i\frac{S}{4}$ pour tout i entre 1 et $4n$. Les communications sortantes ont elles lieu (pour tout j entre 1 et n) aux instants suivants :

- $j\frac{S}{4}$ à $j\frac{S}{4} + \frac{S}{32}$ pour la tâche $4j-3$
- $j\frac{S}{4} + x_{k_j} > j\frac{S}{4} + \frac{9S}{32}$ à $j\frac{S}{4} + x_{k_j} + \frac{S}{32} < j\frac{S}{4} + \frac{11S}{32}$ pour la tâche $4j-2$
- $j\frac{S}{4} + x_{k_j} + x_{k_{j+1}} = j\frac{S}{4} + (x_{k_j} + x_{k_{j+1}} + x_{k_{j+2}}) - x_{k_{j+2}} > j\frac{S}{4} + \frac{7S}{8} - \frac{10S}{32} = j\frac{S}{4} + \frac{18S}{32}$ à $j\frac{S}{4} + x_{k_j} + x_{k_{j+1}} + \frac{S}{32} = j\frac{S}{4} + (x_{k_j} + x_{k_{j+1}} + x_{k_{j+2}}) - x_{k_{j+2}} + \frac{S}{32} < j\frac{S}{4} + \frac{7S}{8} - \frac{9S}{32} + \frac{S}{32} = j\frac{S}{4} + \frac{20S}{32}$ pour la tâche $4j-1$

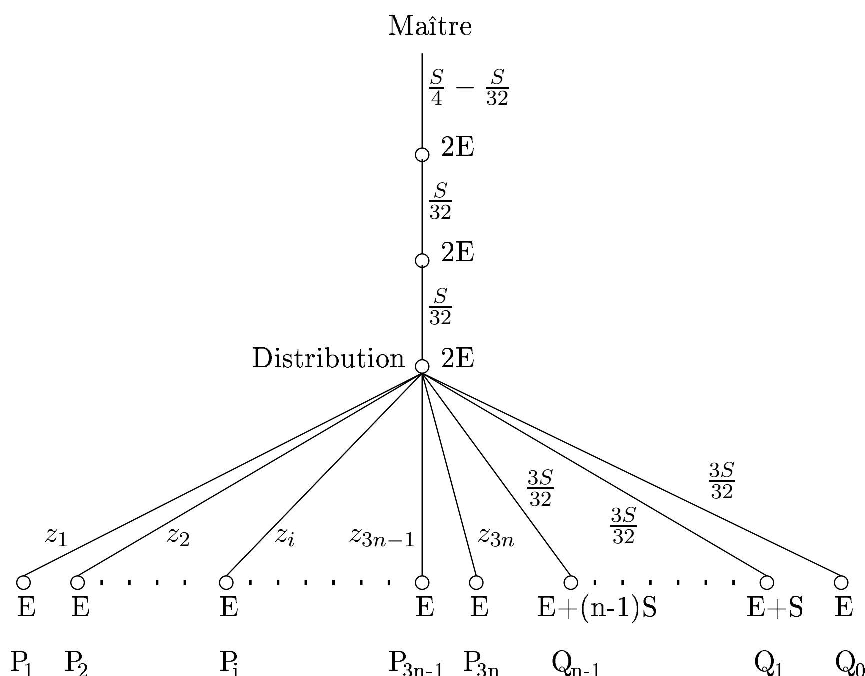

FIG. 7.6 – Arbre modifié, sans lien de communication de temps 0.

– $j\frac{S}{4}+x_{k_j}+x_{k_{j+1}}+x_{k_{j+2}} = j\frac{S}{4}+\frac{7S}{8}$ à $j\frac{S}{4}+x_{k_j}+x_{k_{j+1}}+x_{k_{j+2}}+\frac{S}{32} = j\frac{S}{4}+\frac{29S}{32}$ pour la tâche $4j$

On peut projeter ces temps dans l'intervalle $[0; S/4]$, en prenant pour tout réel r le projeté $p = r - \lambda\frac{S}{4}$ où λ est un entier. Les projections de ces temps de communication sont entre $\frac{7S}{32}$ et $\frac{S}{4}$ pour les réceptions et entre 0 et $\frac{5S}{32}$ pour les émissions. Il n'y a donc pas de recouvrement au niveau de ce noeud.

7.4 Conclusion

Nous avons montré dans les chapitres précédents que le problème de l'ordonnancement sur des chaînes et des pieuvres était polynomial. Avec cette preuve de NP-complétude sur les arbres à un niveau, la frontière entre les problèmes simples et les problèmes complexes est définitivement fixée. Ce qui rend ici le problème difficile est l'existence d'un noeud ayant plusieurs fils en dessous du noeud maître. Les seuls arbres n'ayant pas d'autres noeuds que le maître avec une arité plus grande que 2 sont les pieuvres. Il n'y a donc pas d'autres sous-classes d'arbre à considérer.

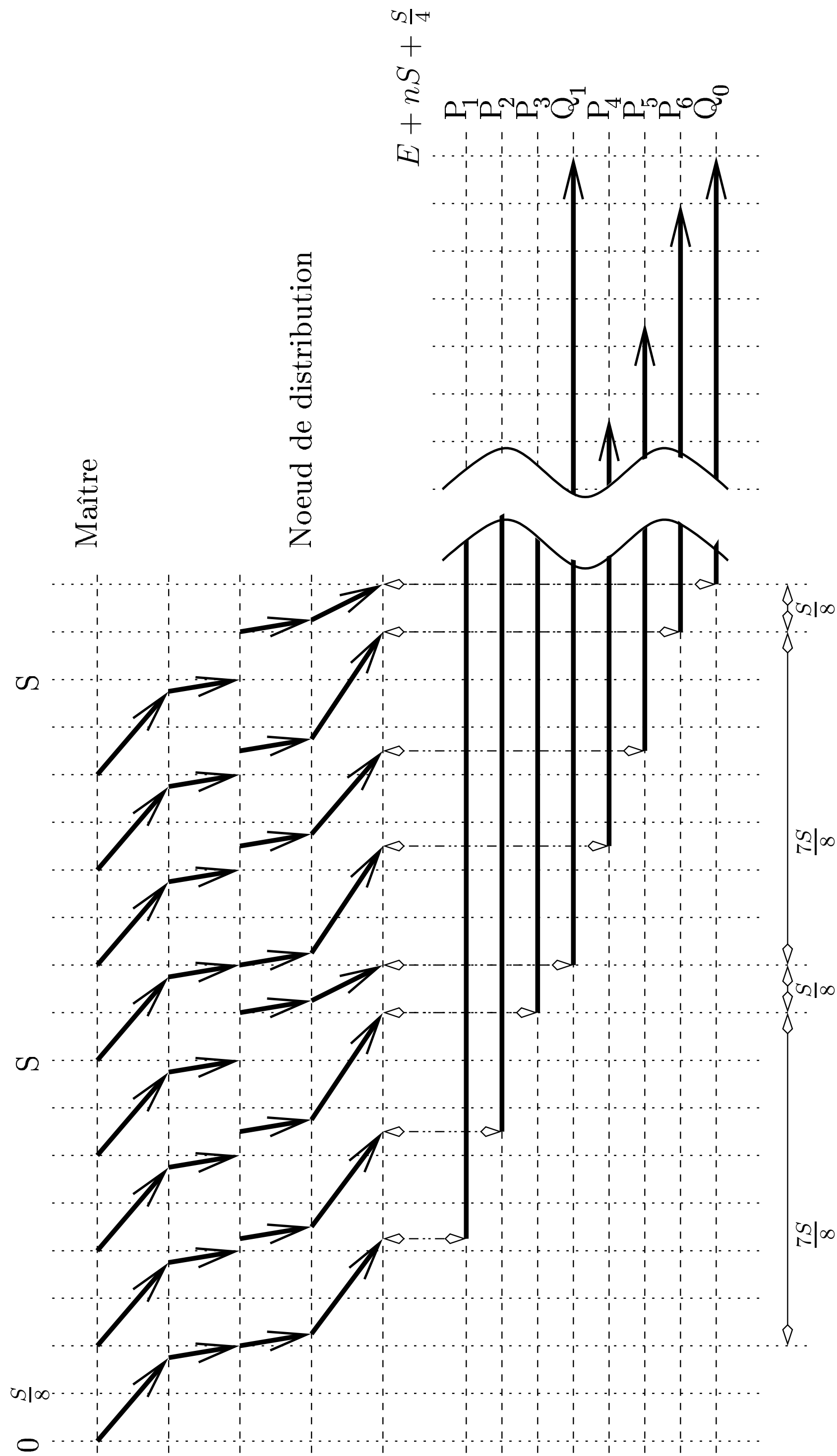

FIG. 7.7 – Ordonnancement de 8 tâches sur l'arbre modifié.

Par contre le problème avec l'arité maximale fixée reste un problème ouvert. Il peut être intéressant en pratique de considérer par exemple le problème pour les arbres binomiaux. Une autre continuation théorique serait de montrer des résultats d'inapproximabilité. Du coté pratique, on peut s'intéresser aux heuristiques d'approximation garanties. Le résultat de complexité sur les arbres pérennise les travaux sur ces heuristiques qui existent déjà [BCF+02].

Conclusion et perspectives

L'étude menée dans le cadre de cette thèse montre bien que les modèles ne peuvent être universels, et doivent donc être manipulés à bon escient. Avant de modéliser le comportement d'une grappe ou d'une application pour tenter d'optimiser le temps de calcul, il faut donc bien réfléchir aux comportements des applications, aux souhaits des utilisateurs ainsi qu'aux capacités réelles. C'est au prix d'une relation étroite entre les administrateurs responsables de l'ordonnancement des travaux sur les machines, les développeurs qui écrivent les applications parallèles et les utilisateurs qui attendent les résultats que peut se faire une gestion efficace des ressources de calcul.

Nous avons présenté ici deux modèles pertinents, le modèle des tâches malléables et le modèles des tâches divisibles. Ces deux modèles sont opposés sur bien des aspects. Les tâches malléables masquent les communications et supposent que le réseau est homogène et complet. À l'inverse, les tâches divisibles mettent les communications au centre du problème, et peuvent être déployées sur un réseau quelconque. Ces deux visions reposent pourtant sur un point de vue macroscopique, où les tâches élémentaires ne sont plus prises en compte, mais noyées par leur nombre. En fait les deux modèles sont duaux et permettent à eux deux de représenter une large palette de situations.

La recherche d'un modèle unique qui puisse recouvrir toutes les situations est une quête difficile, qui semble faire place aujourd'hui à une spécialisation des modèles pour des situations bien définies. Mais les problèmes de grille ne sont pas toujours aussi spécialisés. Il peut y avoir sur une même grille plusieurs profils d'utilisateurs ayant des souhaits très différents. Imposer à tous ces utilisateurs un modèle unique est beaucoup trop restrictif, il n'est pas raisonnable de vouloir convaincre un utilisateur ayant une application rodée de la recoder entièrement pour qu'elle puisse être intégrée dans un autre modèle. On peut donc imaginer que l'avenir réside dans des approches mixtes qui font cohabiter plusieurs modèles.

Un ordonnancement de tâches malléables peut par exemple prendre en compte des tâches multiprocesseurs dont l'allocation est fixée à l'avance, ou une réservation d'une partie de la grille pour des calculs urgents. On peut éga-

lement faire cohabiter le modèle des tâches malléables avec celui des tâches divisibles, en remplissant les temps morts d'un ordonnancement de tâches malléables par un ordonnancement de tâches divisibles, la machine étant alors vue par les tâches divisibles comme un ensemble de processeurs disponibles pendant quelques fenêtres temporelles. Il se pose alors des questions de déploiements bien plus compliqués que ceux qui ont été abordés ici. Un des plus grands défis est alors d'essayer de satisfaire des demandes très différentes en nature, puisque chaque utilisateur peut avoir son propre objectif.

Bibliographie

[AGR03] M. Adler, Y. Gong, and A.L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *15th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 1–10, 2003.

[BCF+02] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium*, 2002. Rapport technique disponible à l'adresse `http://www.ens-lyon.fr/~yrobert`.

[BCR80] R. Baker, E.G. Coffman, and R.L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4) :846–855, 1980.

[BDMT99] E. Blayo, L. Debreu, G. Mounié, and D. Trystram. Dynamic load balancing for ocean circulation model with adaptive meshing. In *EuroPar'99*, volume 1685 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[BGK99a] E. Bampis, R. Giroudeau, and J.C. König. An approximation algorithm for precedence constrained scheduling problem with hierarchical communications. Technical Report 35, LaMI, 1999. accepté à TCS.

[BGK99b] E. Bampis, R. Giroudeau, and J.C. König. On the hardness of approximating the precedence constrained multiprocessor scheduling problem with hierarchical communications. Technical Report 34, LaMI, 1999. accepté à RAIRO-RO.

[BGT97] E. Bampis, F. Guinand, and D. Trystram. Some models for scheduling parallel programs with communication delays. *Discrete Applied Mathematics*, 72 :5–24, 1997.

[BLR02a] O. Beaumont, A. Legrand, and Y. Robert. A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs. In *ISCIS XVII, Seventeenth International Symposium on*

Computer and Information Sciences, pages 115–119. CRC Press, 2002.

[BLR02b] O. Beaumont, A. Legrand, and Y. Robert. Static scheduling strategies for heterogeneous systems. Technical Report 2002-29, École Normale Supérieure de Lyon, July 2002. http://www.ens-lyon.fr/~yrobert.

[BLR03] O. Beaumont, A. Legrand, and Y. Robert. Optimal algorithms for scheduling divisible workloads on heterogeneous systems. In *Heterogeneous Computing Workshop*. IEEE computer society, 2003.

[CD73] E.G. Coffman and P.J. Denning. *Operating systems theory*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Ciment] Ciment project. http://ciment.ujf-grenoble.fr.

[CK02] P. Crescenzi and V. Kann. A compendium of NP optimization problems. Technical report, http://www.nada.kth.se/~viggo/problemlist/compendium.html, 2002.

[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.

[CPS+96] S. Chakrabarti, C.A. Phillips, A.S. Schulz, D.B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. *Lecture Notes in Computer Science*, (1099) :646–657, 1996.

[CR88] Y.C. Cheng and T.G. Robertazzi. *Distributed computation with communication delays*, volume 24, pages 700–712. 1988.

[CR90] Y.C. Cheng and T.G. Robertazzi. *Distributed computation for a tree network with communication delays*, volume 26, pages 511–516. 1990.

[CRL00] Saravut Charcranoon, Thomas G. Robertazzi, and Serge Luryi. Optimizing computing costs using divisible load analysis. *IEEE Transactions on computers*, 49(9) :987–991, September 2000.

[CS02] Y. Collette and P. Siarry. *Optimisation multiobjectif*. Eyrolles, oct 2002.

[CSG99] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture - A Hardware,Sofware approach*. Morgan Kaufmann Pub., 1999.

[CT93] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles*. collection IIA. InterEditions, 1993.

[DBBD95] P. Dell'Olmo L. Bianco, J. Blazewicz and M. Drozdowski. Scheduling multiprocessor tasks on a dynamic configuration of dedicated processors. *Annals of Operations Research*, 58 :493–517, 1995.

[DBS95] P. Dell'Olmo L. Bianco and M.G. Speranza. Scheduling independent tasks with multiple modes. *Discrete Applied Mathematics*, 62 :35–50, 1995.

[Dec] Decrypthon. `http://www.infobiogen.fr/services/decrypthon/`.

[Dec00] T. Decker. *Ein universelles Lastverteilungssystem und seine Anwendung bei der Isolierung reeller Nullstellen*. PhD thesis, Universität Paderborn, 2000.

[DEMT04] P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *Symposium on Parallel Algorithm and Architectures*, 2004.

[DK99] T. Decker and W. Krandick. Parallel real root isolation using the Descartes method. In Prith Banerjee and Victor K., editors, *Proceedings of the 6th High Performance Conference (HiPC99)*, volume 1745 of *Lecture Notes in Computer Science*, pages 261–268. Springer-Verlag, 1999.

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

[DMT04] P.-F. Dutot, G. Mounié, and D. Trystram. *Handbook of Scheduling*, chapter Scheduling Parallel Tasks - Approximation Algorithms. Number 26. CRC press, 2004.

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

[Dro04a] M. Drozdowski. Divisible load vortal, 2004. `http://www.cs.put.poznan.pl/mdrozdowski/divisible/`.

[Dro04b] M. Drozdowski. *Handbook of Scheduling*, chapter Scheduling Parallel Tasks - Algorithms and Complexity. CRC Press, 2004. chapter 25.

[DT] P.-F. Dutot and D. Trystram. A best-compromise bicriteria scheduling algorithm for malleable tasks. Soumis à Algorithmica.

[DT01] P.-F. Dutot and D. Trystram. Scheduling on hierarchical clusters using malleable tasks. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 199–208. ACM Press, 2001.

[Dut02a] P.-F. Dutot. Ordonnancement de chaînes de tâches malléables. In *14èmes Rencontres Francophones du Parallèlisme*, pages 35–41, april 2002.

[Dut02b] P.-F. Dutot. Ordonnancement de tâches identiques sur réseau hétérogène. In *École thématique sur la globalisation de ressources informatiques et des données*, pages 375–384. INRIA, December 2002.

[Dut04] P.-F. Dutot. Complexity of master-slave tasking on heterogeneous trees. *European Journal on Operationnal Research*, 2004. À paraître.

[Fold] Folding@home. `http://www.stanford.edu/group/pandegroup/folding/`.

[FR95] D. G. Feitelson and L. Rudolph. Parallel job scheduling : Issues and approaches. *Lecture Notes in Computer Science*, 0(949) :1–18, 1995.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machiness. In *Proceedings of the 10th ACM Symposium on the Theory of Computing*, pages 114–118, May 1978.

[Gir00] R. 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-completeness.* W.H. Freeman, New York, 1979.

[Gra69] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2) :416–429, March 1969.

[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.

[HSSW97] L. A. Hall, A. S. Schulz, D.B. Shmoys, and J. Wein. Scheduling to minimize average completion time : Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22(3) :513–544, 1997.

[HSW96] L. Hall, D. Shmoys, and J. Wein. Scheduling to minimize average completion time : Off-line and on-line algorithms. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 142–151, 1996.

[JP02] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. *Algorithmica*, 32(3) :507, 2002.

[KdSF+00] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *International Parallel and Distributed Processing Symposium*, 2000.

[Leg03] A. Legrand. *Algorithmique parallèle hétérogène et techniques d'ordonnancement : approches statiques et dynamiques.* PhD thesis, École Normale Supérieure de Lyon, 2003.

[Li02] K. Li. Scheduling divisible tasks on heterogeneous linear arrays with applications to layered networks. In *Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications*, 2002.

[LM00] R. Lepère and G. Mounié. Ordonnancement de tâches malléables. une alternative efficace pour la programmation d'applications parallèles. *Calculateur Parallèle*, 2000. À paraître.

[LMM02] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems : A survey. *European Journal of Operational Research*, 141(2) :241–252, 2002.

[LR03] A. Legrand and Y. Robert. *Algorithmique Parallèle – Cours et exercices corrigés.* Dunod, 2003.

[LTW01] R. Lepere, D. Trystram, and G. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. *submitted to Internat. J. of Foundations of Computer Science*, 2001.

[Lud95] W. T. Ludwig. *Algorithms for scheduling malleable and non-malleable parallel tasks.* PhD thesis, University of Wisconsin - Madison, Department of Computer Sciences, 1995.

[MJT04] Y. Slimani M. Jemni and D. Trystram, editors. *Informatique répartie : architecture, parallélisme et système.* Hermès Publications, 2004.

[Mou00] G. Mounié. *Ordonnancement efficace d'applications parallèles : les tâches malléables monotones.* PhD thesis, Institut National Polytechnique de Grenoble, Juin 2000.

[MRT99] G. Mounié, C. Rapine, and D. Trystram. Efficient approximation algorithm for scheduling malleable tasks. In *Proc. of 11th ACM Symposium of Parallel Algorithms and Architecture*, pages 23–32, 1999.

[MRT01] G. Mounié, C. Rapine, and D. Trystram. A $\frac{3}{2}$-approximation algorithm for scheduling independent malleable tasks. Soumis en 2001.

[MT02] G. Mounié and D. Trystram. Ordonnancement de tâches malléables. Tutoriel a l'Ecole thématique sur la globalisation des ressources informatiques et des données, Aussois, France, décembre 2002.

[Mers] Mersenne prime search. `http://www.mersenne.org`.

[PSTW97] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the 29th annual ACM symposium on Theory of Computing*, pages 140–149, 1997.

[Que93] M. Queyranne. Structure of a simple scheduling polyhedron. *Mathematical Programming*, 58(2) :263–285, 1993.

[Rob04] T. Robertazzi. Divisible load scheduling, 2004. `http://www.ee.sunysb.edu/~tom/dlt.html`.

[SETI] SETI. `http://setiathome.ssl.berkeley.edu`.

[SLW+98] U. Schwiegelshohn, W. Ludwig, J. Wolf, J. Turek, and P . Yu. Smart SMART bounds for weighted response time scheduling. *SIAM Journal on Computing*, 28, 1998.

[Smi56] W.E. Smith. Various optimizers for single stage production. *Naval Research Logistics Quarterly*, 3(1) :59–66, 1956.

[SRL98] Jeeho Sohn, Thomas G. Robertazzi, and Serge Luryi. Optimizing computing costs using divisible load analysis. *IEEE Transactions on parallel and distributed systems*, 9(3) :225–234, March 1998.

[TB02] V. T'kindt and J.-C. Billaut. *Multicriteria Scheduling – Theory, Models and Algorithms.* Springer Verlag, 2002.

[Top500] The top500 organization website. `http://www.top500.org`.

[TWY92] J. Turek, J.L. Wolf, and P.S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, San Diego, California, June 29–July 1, 1992. SIGACT/SIGARCH.

[Ull75] J.D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Science*, 10 :384–393, 1975.

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

[YC03] Y. Yang and H. Casanova. Umr : A multi-round algorithm for scheduling divisible workloads. In *International Parallel and Distributed Processing Symposium*, 2003.

Liste de mes publications

Chapitre de livre

Pierre-François Dutot, Grégory Mounié et Denis Trystram. *Handbook of Scheduling*, chapter 26 : Scheduling Parallel Tasks - Approximation Algorithms. CRC press, 2004.

Journal international

Pierre-François Dutot. Complexity of master-slave tasking on heterogeneous trees. *European Journal on Operationnal Research*, 2003. À paraître.

Conférences internationales avec comité de relecture et actes

Pierre-François Dutot. Master-slave tasking on heterogeneous processors. In *International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2003.

Pierre-François Dutot et Denis Trystram. Scheduling on hierarchical clusters using malleable tasks. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 199–208. ACM Press, 2001.

Pierre-François Dutot, Lionel Eyraud, Grégory Mounié et Denis Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *Symposium on Parallel Algorithm and Architectures*, 2004.

Conférences nationales avec comité de relecture et actes

Pierre-François Dutot. Ordonnancement de chaînes de tâches malléables. In *Rencontres Francophones du Parallèlisme*, pages 35–41, 2002.

Pierre-François Dutot. Ordonnancement de tâches identiques sur réseau hétérogène. In *École thématique sur la globalisation de ressources informatiques et des données*, pages 375–384. INRIA, December 2002.

Papiers invités

Pierre-François Dutot. Scheduling divisible tasks on heterogeneous processors. in the Scheduling in Computer and Manufacturing Systems workshop in Dagstuhl, Jun 2002.

Pierre-François Dutot et Denis Trystram. Scheduling on hierarchical clusters using malleable tasks. in workshop New trends in scheduling for parallel and distributed systems, Oct 2001. CIRM.

Pierre-François Dutot, Lionel Eyraud, Grégory Mounié et Denis Trystram. Models for scheduling on large scale platforms : which policy for which application ? In *International Parallel and Distributed Processing Symposium*, 2004.

Papiers soumis

Pierre-François Dutot et Denis Trystram. A best-compromise bicriteria scheduling algorithm for malleable tasks. Soumis à Algorithmica.