Impact des modèles d'exécution pour l'ordonnancement en calcul parallèle

Alfredo Goldman

▶ To cite this version:

Alfredo Goldman. Impact des modèles d'exécution pour l'ordonnancement en calcul parallèle. Autre [cs.OH]. Institut National Polytechnique de Grenoble - INPG, 1999. Français. NNT : . tel-00004839

HAL Id: tel-00004839
https://theses.hal.science/tel-00004839

Submitted on 18 Feb 2004

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.

Thse

prsente par

Alfredo Goldman

pour obtenir le titre de Docteur

de l'Institut National Polytechnique de Grenoble

(Arrt ministriel du 30 mars 1992)

Spcialit : Informatique

Impact des modèles d'exécution pour l'ordonnancement en calcul parallèle

Date de soutenance : 17 novembre 1999

Composition du jury :

Maurizio BONUCCELLI, *Rapporteur*
Maciej DROZDOWSKI, *Examinateur*
Afonso FERREIRA, *Rapporteur*
Borut ROBIČ, *Examinateur*
Joseph SIFAKIS, *Président*
Denis TRYSTRAM, *Directeur*

Thse prpare au sein du Laboratoire de Modélisation et Calcul - LMC et soutenue au Laboratoire Informatique et Distribution - ID

(Institut d'Informatique et Mathmatiques Appliques de Grenoble - IMAG)

À mes meninas :
Júlia et Paula.

Remerciements

Quand j'ai commencé à réfléchir sur mes remerciements, j'ai remarqué que beaucoup des gens m'ont aidé directement et indirectement. Pour organiser un peu la liste, je divise les remerciements en trois catégories : avant, pendant la thèse le coté scientifique et pendant la thèse le coté humain.

Trois personnes ont été très importantes pour que l'idée de faire une thèse aboutisse. D'abord mon père qui depuis que j'ai conscience d'être moi même (je ne me rappelle pas de grand chose avant ça), a crée et a alimenté ma curiosité scientifique (je me rappelle, que d'abord je voulais être astronaute. Mais ensuite, je pris à l'âge de huit ans, la décision de devenir chercheur).

Ensuite je tiens à remercier mon premier directeur de recherche au Brésil, le Professeur Siang Song, qui m'a montré que pour devenir chercheur en plus de l'inspiration, la transpiration est aussi essentielle. Pour terminer (attention ce sont le remerciements pour l'idée de faire une thèse, il y en a encore), je remercie aussi le Professeur Afonso Ferreira, qui m'a montré aussi que nous avons besoin de beaucoup lire avant d'écrire. Tous les trois m'ont bien soutenu dans le projet de faire un doctorat dans le pays du fromage (eh, oui même si je n'aime pas trop le fromage).

La personne la plus importante grace à qui ma thèse est là aujourd'hui est sans doute mon chef bien aimé le Professeur Denis Trystram, il m'a guidé durant quatres années sur la bonne route. Sa direction et son amitié ont été essentielles à l'aboutissement de ce mémoire.

Je tiens à remercier aussi les personnes avec qui j'ai eu la chance de coopérer, en plus de mon yoguique chef, j'ai travaillé avec Grégory Mounié, Joseph Peters et Christophe Rapine. Ils m'ont montré les avantages de la recherche en équipe. Greg avec sa bonne humeur éternelle, Joe avec sa grande rigueur mathématique et Chris avec sa boite crânienne pleine d'idées.

Il ne faut pas oublier que pour rendre le document dans ce format finale (en français), j'ai eu besoin de plusieurs réviseurs qui m'ont aidé à écrire

la langue de Baudelaire. Je remercie Christophe (qui arrivait à corriger le français et la mathématique en même temps), Gilles (un correcteur très efficace, surtout si le mot **hormis** ne fait pas partie de mon vocabulaire), Grégory (toujours disponible pour les corrections de dernières minutes) et Renaud (qui a pu corriger des chapitres dans l'intervalle entre deux blitz).

Finalement je remercie mes rapporteurs le Professeur Afonso Ferreira et le Professeur Maurizio Bonuccelli pour leurs lectures attentives et leurs suggestions. Je remercie également les autres membres du jury le Professeur Robič Borut, le Professeur Maciej Drozdowski, ainsi qui le président Professeur Joseph Sifakis, pour avoir partagé avec moi l'aboutissement de mon travail de thèse.

Bien sur, il a eu un coté humain très important sans lequel je n'aurais pas tenu le coup. Tout d'abord ma femme Paula qui a tout laissé au Brésil pour pouvoir participer à mon aventure française. Sans elle je ne serait pas capable de concevoir mon oeuvre majeure, Júlia (qu'est ce qu'elle est jolie !!). Eh oui, elle est arrivée le 26 octobre et a éclipsé un peu la sensation de devenir docteur (sans elle, ceux qui n'ont pas écouté auraient écouté, ceux qui ont écouté auraient écouté encore plus : Chui Docteur !!). Le soutien continu de mon père, ma mère et ma soeur ont été aussi très important. Ils m'ont transmit leur appui à distance, ainsi que parfois sur place.

Les collègues du labo ont aussi fait en sorte que mes heures de travail soient plus agréables (est que ce quelqu'un a dit X-Blast ?). Les cris "séminaire", la recherche hurlante de Gerson et la musique (bretonne ??) seront gravés pour toujours dans ma mémoire.

Les amis et amies extérieures au labo ont aussi permis que mon séjour soit plus agréable. D'abord, je voudrais citer les brésilliens (merci pour la Penduick Celso !!) entre autres (je ne peux pas citer tout le monde !!) Gustavo et Kelly, Fabiano et Alessandra, Elson et Stela, Carlos et Ciane, Gerson et Luciana, Paulo et Marilia, Luis et Carla, ... Ensuite des amis qui nous ont bien accueillis à Grenoble : Aurélie et Claude le Faou, Grégory et Frédérique, Géraldine, Lina, Laurent, Nadia, ...

Pour finir je voudrais aussi remercier les institutions brésiliennes qui m'ont sponsorisé pendant ces quatre années : l'Université de São Paulo et le CNPq.

Dès maintenant, je m'excuse auprès de ceux qui n'ont pas été cités par leur nom mais je n'ai eu que quelques heures pour écrire tout ça. Finalement, Il y a tant de monde qui m'ont aidé !!

MERCI !!

Table des matières

1 Introduction **11**

2 Concepts d'ordonnancement **19**
2.1 Introduction . . . 19
2.2 Graphe de précédence . . . 20
2.3 Modélisation de machines parallèles . . . 25
2.3.1 Description des caractéristiques des machines . . . 26
2.4 Description des modèles d'exécution . . . 27
2.4.1 Un modèle pour la complexité : PRAM . . . 28
2.4.2 Modèles avec délai de communication . . . 29
2.4.3 Modèle LogP . . . 33
2.4.4 Modèles plus précis . . . 35
2.4.5 Modèle BSP . . . 37
2.4.6 Tâches malléables . . . 39
2.5 Ordonnancement . . . 40
2.5.1 Évaluation pratique des applications parallèles . . . 41
2.6 Résultats d'ordonnancement . . . 43
2.6.1 Algorithmes de liste . . . 43
2.6.2 PRAM . . . 43
2.6.3 Modèles avec délai de communication . . . 43
2.6.4 LogP . . . 46
2.6.5 BSP . . . 46
2.7 Conclusion . . . 47

3 Modèle à grain fin **49**
3.1 Problème du sac à dos . . . 49
3.2 Solutions parallèles . . . 50
3.3 Approche par programmation dynamique . . . 52

3.3.1 La grille irrégulière 52
3.4 Ordonnancement de la grille irrégulière 55
3.4.1 Un premier ordonnancement 55
3.4.2 Un algorithme amélioré 60
3.5 Conclusion 64

4 Ordonnancement sous BSP 65
4.1 Ordonnancement de chaînes indépendantes 65
4.1.1 Ordonnancement sous le modèle délai 67
4.1.2 Préliminaires sur l'ordonnancement sous BSP 67
4.2 Deux cas préliminaires dans BSP 68
4.2.1 SIC sur deux processeurs dans BSP 69
4.2.2 SIC avec un nombre non borné de processeurs dans BSP 70
4.3 Nombre fixé de processeurs 71
4.3.1 Équilibrage total de la charge 71
4.3.2 Un algorithme 74
4.3.3 Algorithme avec nombre fixé de super-étapes 75
4.4 Influence de la latence 79
4.5 Quelques expérimentations 80
4.5.1 Remarques sur les expérimentations 82
4.6 Conclusion 83

5 Ordonnancement avec duplication 85
5.1 La duplication 85
5.1.1 Nombre de processeurs illimité 86
5.1.2 Nombre de processeurs borné 87
5.2 Duplication dans le modèle SCT 88
5.2.1 Notations 88
5.3 Algorithme EDTF 92
5.3.1 Chemin de duplication 92
5.3.2 La date réalisable 95
5.3.3 Analyse de l'algorithme 98
5.4 Conclusion 100

6 Ordonnancement des communications 101
6.1 Le problème de l'échange de messages 101
6.1.1 Description du problème 103
6.1.2 Le modèle 104

6.1.3 Notations et bornes inférieures 105
6.2 Travaux précédents 106
6.2.1 Modèles plus restreints 106
6.2.2 Problèmes similaires 107
6.2.3 Résultats précédents pour le MEP 107
6.3 Notre contribution 110
6.3.1 Algorithme avec commutation de messages 111
6.4 Algorithme à motif fixé 113
6.4.1 Algorithmes avec connaissance globale 116
6.4.2 Étude qualitative 120
6.4.3 Un algorithme combiné 121
6.5 Résultats expérimentaux 122
6.5.1 Implantation 122
6.5.2 Simulation 124
6.6 Conclusion 131

7 Conclusion **133**

A Linéarité par morceau **137**

B Solving Knapsack on hypercube **139**
B.1 Full description of Knap 139
B.2 Routing scheme 140
B.3 Properties on the average idle time 142

C Scheduling independent chains on BSP **145**
C.1 SIC on two processors in BSP 145
C.2 Complexity of SIC 146
C.3 Guaranty on the algorithm 4.3 147
C.4 Fixed number of supersteps algorithm 148
C.5 Bound improvement 150
C.6 Influence of the latency 151

D Scheduling with duplication **153**
D.1 Number of favorite predecessors 153
D.2 Feasible starting time 153
D.3 Algorithm analysis 155
D.4 Algorithm complexity 158

E Scheduling with communications **161**
E.1 Complexity 161
E.2 Even number of processors HL algorithm 164
E.3 Matching algorithms 165
E.3.1 Max-min matching 165
E.3.2 Max-weight matching 166

Chapitre 1

Introduction

L'évolution de l'informatique dans les cinquante dernières années a été remarquable. Restreinte à ses débuts aux applications scientifiques ou militaires, elle fait aujourd'hui partie de la vie quotidienne. Une évolution si rapide que plusieurs ordinateurs qui appartenaient au domaine de la science fiction il y a vingt ans sont aujourd'hui disponibles pour le grand public. Deux facteurs principaux ont favorisé ce développement : l'augmentation des performances et la chute importante du prix du matériel informatique.

Même avec l'évolution incessante dans le domaine du matériel, il a toujours existé un fossé entre la technologie disponible et les besoins de performances. Chaque progression de la technologie ouvre l'horizon à de nouveaux besoins ; et de nouvelles exigences. La demande pour de nouveaux progrès est constante. En effet, les applications deviennent de plus en plus gourmandes en temps de calcul et en espace mémoire. Plusieurs applications éprouvent un besoin croissant en ressources de calcul, notamment les applications temps réel et les programmes tels que la prédiction météorologique, où les besoins de précision, de fiabilité et de plage d'estimation sont toujours croissants.

De tout temps, le parallélisme a été une possibilité de répondre à cette demande de performances. Ceci est d'autant plus vrai aujourd'hui avec la présence massive des ordinateurs reliés entre eux. Le parallélisme est présent dans les ordinateurs parallèles spécialisés, dans les réseaux d'ordinateurs, dans les télécommunications et même dans les ordinateurs personnels haut-de-gamme qui peuvent avoir jusqu'à huit microprocesseurs. Dorénavant, nous allons utiliser le terme machine parallèle pour faire référence aux machines où le parallélisme est présent, ainsi que le terme processeur pour dénommer leurs unités de calcul.

Le recours au parallélisme est motivé principalement par le besoin de plus de rapidité - un travail avance plus vite à plusieurs qu'à un seul. Le concept de base repose sur l'idée intuitive qu'avec m ordinateurs un programme s'exécutera m fois plus vite que sur un seul.

Cependant, la programmation sur une machine parallèle demande beaucoup plus d'efforts que l'écriture d'un programme séquentiel. La programmation dépend fortement de la machine cible et de son support au parallélisme. Par exemple, les processeurs d'une machine parallèle peuvent avoir des propriétés distinctes, telles que le mode d'échange des données (mémoire partagée ou distribuée via un réseau d'interconnexion), de même que la cadence de son fonctionnement. Il est clair que pour obtenir une implantation efficace une bonne connaissance de la machine cible et de son support au parallélisme est essentielle.

L'élaboration des applications parallèles

Dans l'informatique séquentielle, l'écriture d'une application consiste à choisir un algorithme qui résout un problème et de faire son analyse. Pour concevoir une application parallèle, nous devons d'abord choisir un algorithme qui résout le problème envisagé. Cet algorithme est alors partitionné en tâches, lesquelles correspondent à des portions du code qui seront exécutées en séquentiel. Ensuite, le programme parallèle est conçu en attribuant à chaque tâche un processeur et une date d'exécution. La performance de l'application est mesurée alors par le temps d'exécution parallèle.

Si une tâche a besoin des données produites par une autre tâche, il existe une relation de dépendance. Les tâches et leurs relations de dépendance sont représentées sous la forme d'un graphe, où les sommets correspondent aux tâches et les arcs aux relations de précédence entre elles. Ce graphe exprime le degré de parallélisme de l'algorithme et est nommé *graphe de précédence* ou de tâches. L'opération qui attribue aux tâches des dates d'exécution et des processeurs est dénommée ordonnancement, elle sera présentée en détail au chapitre 2.

Il est clair qu'il faut distinguer la phase d'extraction du parallélisme d'un algorithme de la phase d'exploitation de ce parallélisme. Dans un premier temps un graphe de précédence est construit, et ensuite un schéma d'ordonnancement est appliqué. Or, pour certains problèmes, le graphe de précédence est construit au cours de l'exécution. Dans ce cas, les deux étapes ne peuvent pas être dissociées dans le temps.

Les deux phases peuvent se faire d'une façon implicite, ou explicite. Lors de l'extraction du parallélisme, la construction du graphe de précédence peut être faite par des techniques de compilation (HPF [86]) de manière implicite. Le graphe de précédence peut aussi être construit de façon explicite comme en Ath1 [38] et Cilk [10] où le passage des arguments ou le retour des résultats induisent les précédences. L'ordonnancement peut aussi se développer implicitement comme dans Ath1 et Cilk où des techniques d'ordonnancement dynamique avec équilibrage de charge (par exemple vol de travail, en anglais *work stealing*) sont utilisées. Pyrros [104] possède un langage de description du graphe de précédence ainsi que des fonctions pour son ordonnancement. Rapid [37] est un environnement reposant sur les mmes principes que Pyrros. Dans les deux derniers cas l'ordonnancement est explicite. Pourtant le spectre des applications disponibles est encore très restreint.

Nous nous intéressons dans cette thèse aux problèmes où le graphe de précédence est fourni de façon explicite avant le début de l'exécution (ordonnancement statique). Dans ces problèmes d'ordonnancement notre objectif est de minimiser le temps d'exécution total. L'approche choisie est le développement d'algorithmes spécialisés, c'est-à-dire d'algorithmes dans lesquelles l'ordonnancement peut être décrit de façon explicite.

Idées directrices

Lors d'une exécution séquentielle (ordonnancement sur un seul processeur) le choix d'une allocation des tâches qui minimise le temps total d'exécution est facile. Toute attribution de dates, sans temps d'inactivité entre les tâches, qui respecte les contraintes de précédence aura le même temps d'exécution et sera valide. Par contre, lorsque plusieurs processeurs sont disponibles, le problème de l'ordonnancement devient plus ardu. Dans ce cas, certains facteurs jouent un rôle primordial. Parmi eux, nous pouvons citer l'architecture de la machine parallèle, son support d'exécution et les particularités liées au graphe [54].

Pour l'application, les caractéristiques principales sont la structure de son graphe de précédence et sa granularité[1]. La structure détermine les possibilités de parallélisme. La granularité fournie une estimation du temps de calcul d'une tâche par rapport à la quantité des données à échanger. Si ce rapport est important le graphe est dit à gros grain, sinon il est dit à grain fin.

[1] Défini comme le rapport entre le coût des tâches et leurs coûts de communication.

Selon les caractéristiques de la machine et son support d'exécution, plusieurs facteurs influencent l'ordonnancement. Parmi les plus importants nous avons le mode d'échange des données entre les processeurs, le synchronisme et l' uniformité des processeurs. Suivant l'architecture de la machine, l'échange des données peut s'effectuer soit par une mémoire partagée, soit par un réseau d'interconnexion. Le synchronisme peut être en mode strict, s'il existe au niveau du matériel, ou être lié au modèle de programmation. Le modèle *BSP* [99] est fondé sur un fonctionnement par étapes, une phase de calcul étant suivie d'une phase de communication et synchronisation. Les processeurs sont dits uniformes s'ils ont le même rapport de vitesse pour toutes les tâches. Au cours de cette thèse, la durée d'exécution de chaque tâche est la même sur tous les processeurs, ceux-ci sont alors dits identiques.

Les autres caractéristiques du support d'exécution que nous pouvons aussi prendre en compte, mais qui ne sont liées ni à la machine, ni au graphe, sont la possibilité de préempter ou de dupliquer les tâches.

La qualité d'un ordonnancement, voir sa validité, dépendent de nombreux paramètres, tant au niveau de la structure de l'application que de la machine cible et son support d'exécution. L'étude de l'influence de ces paramètres est capitale dans la recherche d'implantations parallèles efficaces. Il est à noter que la recherche d'un ordonnancement optimal, c'est-à-dire le plus efficace, est en général un problème $\mathcal{NP}$-difficile. Dans ce cas nous recherchons des heuristiques.

Nous nous intéresserons au cours de cette thèse à l'étude des ordonnancements, en particulier, en ce qui concerne les effets des paramètres de grain, de communication et de synchronisme. L'approche choisie est l'étude approfondie de quelques problèmes, chacun d'entre eux mettant en valeur un aspect particulier important pour l'ordonnancement.

Au cours de cette thèse notre intérêt est l'ordonnancement, les modèles de machines étant son support. Nous ne proposons pas de nouveaux modèles adaptés à tel ou tel algorithme et/ou machine. Selon le problème, un modèle cohérent est choisi, et un ordonnancement est proposé.

Une des influences les plus importantes est celle de la communication. Si la granularité est fine un modèle où le *makespan*[2] est calculé par rapport au nombre d'étapes de calcul/communication est bien adapté. Si les communications impliquent tous les processeurs, il est intéressant de grouper les échanges de messages entre processeurs.

[2]Temps total d'un ordonnancement.

Nous étudierons en premier lieu dans cette thèse l'ordonnancement d'un graphe de précédence régulier à grain très fin. Pour ce problème, afin d'avoir une exécution efficace, un modèle synchrone où le coût des communications est implicite s'impose. Puis, nous introduirons les coûts de communication, une contrainte qui s'avère plus réaliste pour la plupart des machines parallèles courantes, en gardant le synchronisme. Dans ce cas nous étudierons un problème autre que à grain fin : l'ordonnancement de chaînes. Ensuite, après avoir constaté l'importance des coûts de communication, nous proposerons deux techniques pour alléger ce coût, la duplication et l'ordonnancement de communications.

Organisation du document

Dans ce mémoire nous étudions les problèmes liés à l'ordonnancement statique des graphes orientés acycliques. Toutes les notations et les propriétés de base sont présentées dans le chapitre 2. Nous y détaillons le modèle de graphe de tâches et nous introduisons les modèles de machines. Pour chaque problème d'ordonnancement développé, nous consacrons un chapitre et une annexe technique en anglais. L'objectif de cette organisation est de permettre une bonne lisibilité, sans surcharge technique, du document. Dans chaque chapitre sont présentés, le problème, ses caractéristiques, un état de l'art et les solutions proposées. Les détails techniques et les preuves complètes se trouvent en annexes.

La résolution du problème du sac à dos est considérée dans le chapitre 3. La parallélisation efficace de ce problème a été largement étudiée dans la littérature pour les modèles systolique et *PRAM*[3]. Nous proposons une solution dans un modèle moins restreint que le systolique et plus proche des machines réelles que le modèle *PRAM*, avec une architecture à mémoire distribuée. Notre approche est aussi basée sur un algorithme de programmation dynamique du problème du sac à dos. Le graphe de précédence correspondant se caractérise par un grain très fin. Dans un modèle d'exécution synchrone négligeant le coût des accès mémoire aux données partagées, des algorithmes d'ordonnancement efficaces peuvent être mis en œuvre. Le temps d'exécution séquentiel de l'algorithme qui résout le problème du sac à dos est $O(mc)$ où m est le nombre d'objets différents et c est la capacité du sac à dos. Nous proposons d'abord un algorithme qui résout le problème dans un hypercube

[3]Modèles synchrones à grain fin, cf section 2.3.1

avec p processeurs en temps $O(\frac{mc}{p}\frac{w_{\max}}{w_{\min}})$ où $w_{\max}$ et $w_{\min}$ sont, respectivement, les poids des objets le plus lourd et le plus léger. Ensuite nous présentons un algorithme efficace qui résout le problème du sac à dos en temps $O(\frac{mc}{p})$ dans un hypercube avec p processeurs. Ces algorithmes sont dérivés à partir des ordonnancements de la grille irregulière dans l'hypercube. Ce travail a été mené avec Denis Trystram.

Dans le chapitre 4, nous examinons l'ordonnancement de chaînes dans le modèle *BSP*. Dans ce modèle, les échanges de données sont effectuées avant les phases de synchronisation, mais leurs coûts sont pris en compte. Contrairement au modèle asynchrone l'ordonnancement de chaînes sur *BSP* est $\mathcal{NP}$-difficile, notre approche a été la recherche d'algorithmes d'approximation. Nous avons été capables de proposer des heuristiques quasi optimales. Dans ce but, nous avons utilisé deux techniques, le regroupement des communications asynchrones dans des communications du type *BSP*, et la recherche d'un compromis entre le temps d'inactivité et le nombre de communications du type *BSP*. Nous avons complété l'analyse théorique par un grand nombre de simulations. Cette étude a été réalisée avec Grégory Mounié et Denis Trystram.

La réduction de l'impact de la communication en introduisant la duplication a été bien étudiée dans le cas de ressources illimitées (nombre de processeurs non borné). Dans le cas d'un nombre fixé de processeurs, le problème n'a été traité que pour un cas simple (taille des tâches et des communications unitaires). Nous étudions le cas de petits temps de communication. Nous proposons un algorithme de liste, et nous démontrons qu'il a une garantie de performance de 2. Cette garantie est très proche de la garantie des algorithmes de liste lorsque le temps de communication inter processeurs n'est pas considéré, qui est $2 - \frac{1}{m}$ pour m processeurs. L'idée de base de l'algorithme est la construction d'un chemin de duplication pour chaque tâche prête[4]. À partir de ces chemins, l'algorithme possède des informations pour décider lors de l'allocation d'une tâche lesquels de ces prédécesseurs doivent être dupliqués. Cette étude a été faite avec Christophe Rapine. Les résultats sont présentés dans le chapitre 5.

L'ordonnancement de communications est traité dans le chapitre 6. Il consiste à optimiser les phases de communication en ordonnançant les messages sur le réseau d'interconnexion afin de réduire les problèmes de congestion et d'optimiser les schémas de diffusions des données. Cependant, trouver

[4]Une tâche est considéré prête quand tous ses prédécesseurs ont été alloués

la solution optimale est un problème NP-difficile, par conséquent plusieurs heuristiques sont proposées. Nous analysons chacune d'entre elles et nous proposons des critères qui déterminent l'heuristique la plus adaptée selon les communications à réaliser et les caractéristiques de la machine. Nous proposons à la fin du chapitre des algorithmes combinés, c'est-à-dire composés de deux heuristiques différentes. Nous achevons le chapitre avec des expérimentations et des simulations. Celles-ci confirment les bonnes performances des algorithmes combinés. Ce travail a été mené avec Joseph Peters et Denis Trystram.

Chapitre 2

Concepts d'ordonnancement

Résumé

La mise au point d'algorithmes d'ordonnancement dépend fortement des caractéristiques de l'application et de la machine cible. Pour décrire ces caractéristiques l'utilisation de *modèles* s'impose. Nous présenterons un modèle d'application basé sur la notion de graphe de précédence. Les modèles nécessaires à la programmation sont étudiés ensuite : modèles de machine et modèles d'exécution. L'objectif de ce chapitre est de faire un tour d'horizon des modèles pour le parallélisme et l'ordonnancement statique.

2.1 Introduction

L'étude de l'ordonnancement des tâches d'un programme sur une machine parallèle réelle étant très complexe, l'introduction des modèles s'impose. La prise en compte de toutes les caractéristiques de la machine s'avère assez difficile. En outre, le degré d'influence pour certaines caractéristiques est plus important que pour d'autres. Les modèles considèrent quelques unes d'entre elles, sans prendre en compte d'autres caractéristiques qui influencent moins les performances. Ils servent à estimer le comportement de la machine lors d'une véritable exécution. Malheureusement, il n'existe pas de modèle universel pour les machine parallèles similaire au modèle d'architecture de *Von Neuman* pour les ordinateurs séquentiels.

Un bon modèle doit avoir deux qualités : être réaliste et simple. Le réalisme est un critère antagoniste à la simplicité. Plus un modèle est simple, donc abstrait, moins il est proche du comportement d'une machine réelle. À

l'opposé, un modèle réaliste s'avère souvent trop complexe à utiliser. Nous allons classer les modèles utilisés selon trois catégories:

- Les modèles d'application, qui servent à représenter le parallélisme potentiel d'un algorithme.
- Les modèles de machine, qui décrivent les principales caractéristiques de la machine parallèle comme les flots d'instructions et les communications entre processeurs.
- Les modèles d'exécution, qui détaillent les formes de programmation ainsi que ses paradigmes.

Ce n'est pas une classification standard de la littérature spécialisée, cependant elle est assez logique. Les trois concepts de l'informatique parallèle : algorithme, machine et support d'exécution restent bien délimités. Notons de plus que la frontière entre types de modèles n'est pas stricte.

Le développement d'une application parallèle sur une machine cible peut être vue comme la succession des étapes suivantes :

1. Trouver un modèle de machine qui représente la machine cible,
2. Représenter le parallélisme potentiel de l'algorithme et choisir un modèle d'exécution cohérent avec le modèle de machine,
3. Rechercher un ordonnancement.

Les étapes 2 et 3 sont les principales. Dans l'étape 2, le choix du modèle dépend fortement de la machine cible et du parallélisme présent dans l'algorithme. De même le choix du niveau de parallélisme relève du modèle d'exécution. Après le choix des modèles, la recherche d'un ordonnancement convenable peut commencer.

Dans ce chapitre nous présentons les différentes représentations d'une application, puis les modèles de machines parallèles et les modèles d'exécution. Nous finissons avec une description détaillée de l'ordonnancement en présentant quelques résultats connus.

2.2 Graphe de précédence

La représentation d'un algorithme est faite principalement de deux façons analogues, chacune utilisant un graphe orienté comme structure de base. Nous pouvons représenter un programme par un graphe de flots de données (*data-flow graph*), par un graphe de précédence (*precedence task graph*) ou plus simplement par un graphe de dépendance.

Dans la représentation d'un programme par un graphe de précédence, l'algorithme est divisé en unités de base dénommées tâches (qui sont des instructions ou groupes d'instructions). Les tâches sont représentées par les sommets du graphe. Les dépendances entre les tâches sont explicitées par des arcs. L'existence d'un arc (u, v) d'une tâche u à une tâche v dans le graphe signifie que la tâche v ne peut pas être exécutée sans les données produites par la tâche u. Nous associons à chaque arc un poids proportionnel à la quantité de données à transmettre, et à chaque sommet un poids correspondant au temps de calcul de la tâche. Dorénavant, nous ferons référence à ces valeurs comme poids de l'arc ou du sommet. Lorsque les arcs ne sont pas associés à des poids, c'est-à-dire seule les dépendances entre les tâches est donné nous avons le représentation par graphe de dépendance.

Le graphe de flots de données est construit à partir de l'évolution des données. Les relations de précédence sont induites par la circulation des données. Typiquement, les sommets correspondent à l'évaluation d'une instruction et les précédences aux accès en lecture ou écriture des opérandes [25].

Connaissant une représentation d'un programme par un graphe de flots de données, sa transformation en graphe de précédence est immédiate car le premier fourni plus d'informations que le dernier. Dans ce mémoire, nous nous consacrerons à l'étude des graphes de précédence.

A priori, les tâches peuvent être quelconques, et le graphe de précédence inconnu au moment de l'exécution d'un programme. Ce qu'est le cas de l'ordonnancement dynamique. Nous nous intéressons plutôt à l'ordonnancement statique. Dans nos problèmes, le graphe de précédence est connu à l'avance. Dorénavant, lors de l'utilisation d'un graphe de tâches, il sera entièrement connu.

Pour un graphe de précédence G donné, nous adoptons la terminologie suivante :

Successeurs : l'ensemble des successeurs d'un sommet v est constitué de tous les noeuds u tels qu'il existe un chemin orienté de v à u dans G.

Successeurs directs : l'ensemble des successeurs directs d'un sommet v est constitué de tous les noeuds u tels qu'il existe un arc de v à u dans G.

Prédécesseurs : l'ensemble des prédécesseurs d'un sommet v est constitué de tous les noeuds u tels qu'il existe un chemin orienté de u à v dans G.

Prédécesseurs directs : l'ensemble des prédécesseurs d'un sommet v est constitué de tous les noeuds u tels qu'il existe un arc de u à v dans G.

Largeur : le cardinal du plus grand ensemble de sommets du graphe tel qu'il n'existe pas deux sommets appartenant au même chemin orienté.

Chemin critique : le plus long chemin orienté du graphe, prenant en compte les temps de calcul.

Granularité : rapport entre le poids des sommets et des arcs. La granularité ρ d'un graphe orienté G est le rapport entre le plus petit poids d'un sommet et le plus grand poids d'un arc de G. Si $\rho \leq 1$ alors le graphe est dit à grain fin, sinon il est dit à gros grain. Intuitivement, les tâches d'un graphe à gros grain calculent plus qu'elles ne communiquent.

Dans ce mémoire nous utilisons toujours la notion de grain de calcul, une autre approche est l'inverse, c'est-à-dire considérer le grain de communication. Il existe d'autres définitions de la granularité tels que la granularité globale (rapport entre le travail total et la sommes des communications) et la granularité locale qui considère le grain de chaque tâche. Gerasoulis et Yang [42] présentent plusieurs définitions de la granularité.

Pour illustrer ces terminologies, dans le graphe orienté G de la figure 2.1 nous avons : les successeurs du sommet v_5 sont $\{v_8, v_9, v_{10}, v_{11}\}$, les successeurs de v_2 sont $\{v_6, v_{11}\}$. Les prédécesseurs directs de v_9 sont $\{v_5, v_7\}$. La largeur du graphe G est 4. Si toutes les tâches du graphe G ont le même poids, le graphe a trois chemins critiques : (v_1, v_3, v_7, v_9), (v_1, v_4, v_7, v_9) et (v_1, v_5, v_8, v_{10}).

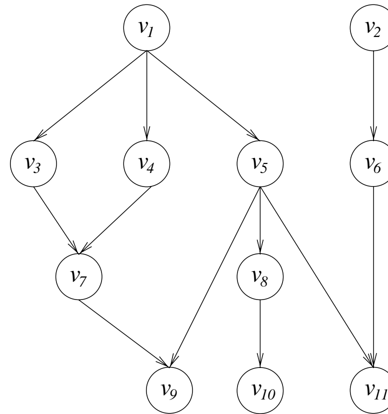

FIG. 2.1 – Graphe de précédence G.

Dans le but de montrer la construction d'un graphe de tâches à partir d'un algorithme, nous présentons le problème de multiplication de matrices carrées de dimensions paires. Si les dimensions sont impaires, il suffit d'ajouter une ligne ou une colonne remplie avec des zéros, ou une ligne et une colonne remplies avec des zéros. Cet ajout peut s'avérer très utile lors de l'application recursive de l'algorithme. La première étape consiste à choisir un algorithme pour résoudre le problème. L'algorithme que nous avons choisi est l'algorithme de Strassen[1]. Étant données deux matrices $n \times n$, A et B, où n est un entier pair, nous voulons calculer $C = A \times B$. Avec la partition des matrices en blocs nous avons :

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Le symbole $\times$ dénote la multiplication de matrices. Chaque quadrant de la matrice C est calculé selon l'algorithme 2.1.

Algorithm 2.1 Algorithme de Strassen.

`Entrées:` Matrices A et B partitionées en quadrants.

$T_1 = A_{11} + A_{12}$; $T_2 = A_{21} - A_{11}$;

$T_3 = A_{12} - A_{22}$; $T_4 = A_{21} + A_{22}$; $T_5 = A_{11} + A_{22}$;

$U_1 = B_{11} + B_{22}$; $U_2 = B_{11} + B_{12}$;

$U_3 = B_{21} - B_{11}$; $U_4 = B_{12} - B_{22}$; $U_5 = B_{21} + B_{22}$;

$P_1 = T_1 \times U_1$; $P_2 = T_4 \times B_{11}$; $P_3 = A_{11} \times U_4$;

$P_4 = A_{22} \times U_3$; $P_5 = T_1 \times B_{22}$; $P_6 = T_2 \times U_2$; $P_7 = T_3 \times U_5$;

$C_{11} = P_1 + P_4 - P_5 + P_7$; $C_{12} = P_3 + P_5$;

$C_{21} = P_2 + P_4$; $C_{22} = P_1 + P_3 - P_2 + P_6$;

Il est clair qu'il existe plusieurs façons de représenter un algorithme à l'aide d'un graphe de précédence. Dans les cas extrêmes, le graphe de précédence d'un programme peut n'avoir qu'une seule tâche. Dans ce cas, la tâche correspond au programme lui même. À l'opposé, chaque instruction élémentaire d'un algorithme peut correspondre à une tâche. Le choix de la représentation est un problème difficile, souvent fait à la main.

[1] Cet algorithme effectue un nombre plus petit de multiplications.

La figure 2.2 illustre un graphe de précédence pour l'algorithme de Strassen. Dans cette figure, chaque tâche reçoit une étiquette correspondant à ses données de sortie. Pour simplifier, les poids des arcs et des sommets ne sont pas représentés.

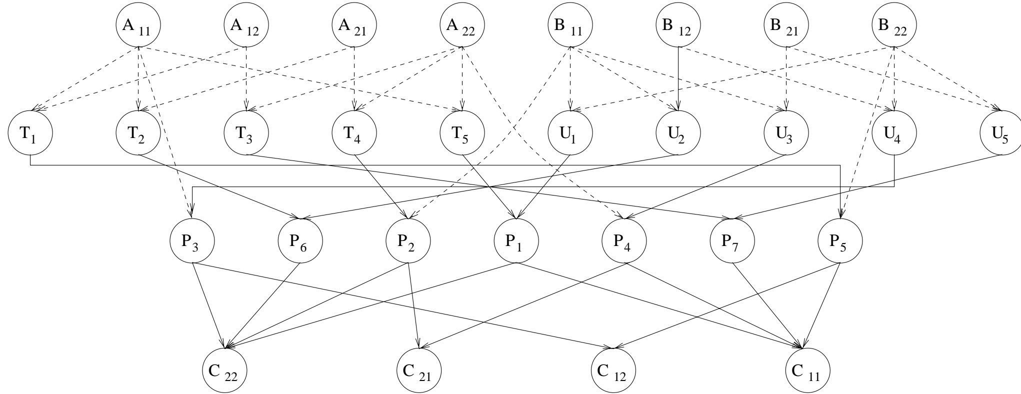

FIG. 2.2 – Un graphe de précédence pour l'algorithme de Strassen.

Chaque tâche U_i, T_j correspond à la somme/soustraction de matrices. Ces tâches ont $n^2/4$ opérations arithmétiques, donc une autre possibilité était de partitionner chacune d'elles en $n^2/4$ tâches d'une seule instruction. Ceci est aussi vrai pour les tâches C_{ij}. Les tâches P_i correspondent à des multiplications de matrices. Chacune de ces tâches aurait pu être partitionnée, par exemple, de façon recursive jusqu'à ce que les tâches du graphe ne soient constituées que par des opérations sur des scalaires.

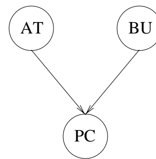

FIG. 2.3 – Un autre graphe de précédence pour l'algorithme de Strassen.

Un autre graphe de précédence possible pour l'algorithme 2.1 est présenté dans la figure 2.3. Dans ce graphe les tâches AT (BU) calculent les valeurs T_i (U_i). La tâche PC calcule les valeurs P_i et finalement la matrice C.

Nous pouvons aussi dériver un graphe de précédence à partir de la description haut niveau d'un algorithme. Un graphe de précédence pour l'algo-

rithme 2.2 ci-dessous est illustré par la figure 2.4 (les étiquettes données aux tâches sont évidentes). Ce graphe de tâches est une chaîne, les tâches f_1, f_2 et f_3 doivent s'exécuter dans cet ordre.

Algorithm 2.2 Algorithme séquentiel.

$b = f_1(a);$

$c = f_2(b);$

$d = f_3(c);$

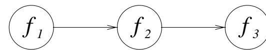

FIG. 2.4 – Graphe de précédence du type chaîne.

À partir d'un graphe de précédence nous disposons de plusieurs informations sur le parallélisme réalisable. La largeur du graphe donne le nombre de processeurs nécessaires à son exécution, car lors de l'allocation des sommets aux processeurs le nombre maximal de tâches différentes qui peuvent être exécutées simultanément est égal à la largeur du graphe. Cependant il existe des résultats classiques où le nombre de processeurs utilisés est aussi grand que le nombre de tâches [84]. Le chemin critique fournit une borne inférieure du temps d'exécution. Le grain du graphe donne une estimation du rapport entre temps de calcul et temps de communication.

Les graphes de précédence proposés pour l'algorithme de Strassen sont à grain fin, car la quantité de données à communiquer est grande. Si dans l'algorithme 2.2 les variables sont des matrices et les fonctions f_i sont assez complexes (inversion de matrices, valeurs propres, etc) le graphe de précédence de la figure 2.4 devient à gros grain.

Pour estimer plus finement le temps d'exécution, nous avons besoin d'introduire des modèles de machine et d'exécution.

2.3 Modélisation de machines parallèles

Les modèles sont employés pour classer les machines elles mêmes, ainsi que pour fournir un support pour la programmation. Ces modèles sont dénommés respectivement modèles de machine et modèles d'exécution. Les

modèles de machine caractérisent l'architecture du matériel, alors que le modèle d'exécution sert à expliciter le comportement de la machine et à préciser des caractéristiques telles que la gestion des communications.

2.3.1 Description des caractéristiques des machines

Pendant longtemps la classification la plus utilisée des ordinateurs parallèles a été celle proposé par Flynn [32]. Les machines peuvent être classées selon deux critères : le flot d'instructions et le flot de données. Quand les flots d'instructions et de données sont uniques, la machine est dite SISD (*single instruction single data*), ce qui correspond aux machines séquentielles. Les deux autres types de machines courantes sont les machines SIMD (*single instruction multiple data*) qui ont un flot d'instructions unique, et plusieurs flots de données, et les machines MIMD (*multiple instruction multiple data*) où les flots d'instructions et les flots de données sont multiples.

Aujourd'hui la plupart des machines est MIMD, donc la classification de Flynn devient obsolète au profit d'autres caractéristiques. Il est courant de diviser la classe MIMD en deux groupes : les multiordinateurs et les multiprocesseurs [94]. Il est à noter que les latences pour l'échange des données sont plus importantes dans les multiordinateurs que dans les multiprocesseurs.

Un autre point qui différencie les architectures d'ordinateurs parallèles est l'architecture physique de la mémoire :

UMA : (de l'anglais *Uniform Memory Access*) il existe un espace d'adressage commun accessible à tous les processeurs. Le temps d'accès à cette mémoire est le même pour tous les processeurs.

NUMA : (de l'anglais *Non-Uniform Memory Access*) chaque processeur possède sa propre mémoire locale. Dans ce cas, la machine et son système d'exploitation fournissent des primitives de communication entre les processeurs. Pour un processeur, il existe plusieurs formes d'accès à la mémoire d'un autre processeur : soit par un espace d'adressage unique comme dans le CRAY T3E, soit par l'échange de messages comme dans le IBM SP2. Dans une machine à mémoire NUMA le coût d'un accès à une mémoire non locale est beaucoup plus important que le coût d'un accès local. Dans ce cas la notion de localité des données devient primordiale.

Le fait d'avoir une mémoire avec accès uniforme simplifie le travail de parallélisation. Cependant, les contraintes technologiques font que le nombre

de processeurs dans les machines UMA est limité (de l'ordre d'une cinquantaine). Pour disposer de beaucoup plus de processeurs, les architectures NUMA s'imposent. Dans le modèle de machines NUMA les processeurs sont reliés par un réseaux d'interconnexion, la manière que cette interconnexion est réalisée définit la topologie du réseau.

Le modèle SIMD présuppose un fonctionnement synchrone des processeurs (généralement, il existe une unité centrale de contrôle responsable pour le flot d'instructions), chaque processeur exécute des cycles composés de calculs et de communications entre processeurs voisins. À chaque cycle, les processeurs exécutent les mêmes instructions, mais avec des données différentes. Dans le modèle MIMD les processeurs peuvent fonctionner de façon asynchrone, il n'y a pas de restriction sur le programme exécuté par chaque processeur.

Le modèle systolique [69] est encore plus restreint que le modèle SIMD, les processeurs sont reliés par un réseaux d'interconnexion, seuls les processeurs à la frontière peuvent communiquer avec le monde extérieur, et ils ont une mémoire locale assez réduite. Le fonctionnement est synchrone, et les processeurs répètent les pas : réception, calcul, envoi. Dans un réseau systolique, les données sont fournies à rythme régulier, et après la traversé du réseau par les données les résultats sont livrés aussi à rythme régulier.

Les modèles de machines servent à classer les machines existantes, mais ils ne sont pas suffisants lors du développement d'une application. Si nous voulons connaître le traitement des conflits lors de l'accès à une donnée partagée, ou les formes de communication entre les processeurs, nous avons besoin des modèles d'exécution.

2.4 Description des modèles d'exécution

Il est clair que le modèle d'exécution doit être adapté au modèle de machine. Un modèle d'exécution qui ne prend pas en compte le temps d'accès à une donnée distante s'avère peu pratique pour une machine MIMD de type NUMA. Nous allons diviser la présentation des modèles selon leur origine. D'abord nous présentons les modèles théoriques, c'est-à-dire, ceux qui n'ont pas été inspirés de machines existantes. Ensuite les modèles basés sur les caractéristiques de machines réelles sont présentés. Dans ce chapitre nous allons étudier de manière assez brève plusieurs de ces modèles afin de fournir la connaissance de base du sujet.

Les modèles d'exécution servent de base à la programmation d'une machine parallèle. Ils donnent la sémantique d'exécution. Un des principaux buts des modèles d'exécution est de servir à la prédiction du temps d'exécution d'un programme parallèle. De la même façon que dans le cas de machines SIMD, le temps d'exécution peut être mesuré en cycles (calcul plus communication). Nous nous intéresserons plutôt aux modèles pour les machines MIMD.

Nous allons restreindre notre étude à des modèles où tous les processeurs sont identiques, c'est-à-dire qu'ils ont la même vitesse et peuvent exécuter le même ensemble d'instructions.

2.4.1 Un modèle pour la complexité : PRAM

Le modèle PRAM (de l'anglais *Parallel Random Access Machine*) est le premier modèle à avoir été proposé pour l'informatique parallèle [33]. Encore aujourd'hui il sert de référence. Il est très populaire pour l'évaluation et la comparaison d'algorithmes parallèles [65]. Dans ce modèle les processeurs identiques fonctionnent en cadence par cycle d'une instruction, et ont accès à une mémoire globale commune. Le nombre de processeurs ainsi que la taille de la mémoire sont illimités. Ce modèle se révèle irréaliste en pratique, car le coût de maintien d'une mémoire globale dépend du nombre de processeurs. Un modèle plus restreint, la p-PRAM, où le nombre de processeurs est égal à p, a aussi été proposé dans la littérature. Ce modèle sert à approcher le comportement des ordinateurs SMP (de l'anglais *Symmetric MultiProcessing*), lesquels partagent les ressources mémoire et entrée/sortie. Les ordinateurs multi processeurs du type PC de dernière génération, où les cartes mères peuvent accueillir jusqu'à huit processeurs qui accèdent la mémoire de façon uniforme, sont des machines SMP.

Dans le but de définir les règles d'accès à la mémoire dans le modèle PRAM, plusieurs versions ont été proposées. Les types sont : le EREW (*Exclusive Read Exclusive Write*), où chaque cellule est accédée par au plus un processeur à chaque cycle, le CREW (*Concurrent Read Exclusive Write*) et le CRCW (*Concurrent Read Concurrent Write*) où l'accès aux cellules peut se faire par plusieurs processeurs pour la lecture, et pour la lecture et écriture, respectivement. Dans le dernier cas, des règles de résolution de conflits sont définies. Les plus courantes, en ordre croissant de complexité sont : arbitraire, prioritaire et combinaison de valeurs (par un maximum, une somme, etc).

Akl et Guenther [2] ont proposé le modèle BSR basée sur la CRCW

PRAM. Ils ont remarqué que l'ajout d'une instruction de diffusion en temps constant n'était pas irréaliste. L'instruction de diffusion présentée autorise à chaque instant l'accès simultané en écriture de toutes les cellules mémoire pour chaque processeur. Les règles de résolution de conflit sont similaires à celles pour la PRAM, avec l'ajout de la notion de données valides.

Plusieurs modifications du modèle PRAM original où les contraintes d'accès à la mémoire distante sont considérées ont été proposées dans la littérature. Il en existe deux classes : ceux qui tiennent compte de la topologie du réseau tel que le modèle XRAM [22], et ceux qui l'ignorent. Dans ce dernier cas le modèle le plus connu est la LPRAM [1] (*Local-memory PRAM*).

Dans les modèles du type PRAM les problèmes de communication sont masqués. La communication est incluse implicitement dans le modèle. En pratique c'est un point important dont il faut tenir en compte.

2.4.2 Modèles avec délai de communication

Dans les modèles avec délai de communication il existe un support pour la communication entre les processeurs. Ce support consiste en l'envoi et la réception de messages. Dorénavant nous dénotons par modèles délai, les modèles que considèrent uniquement la taille des tâches et le délai de communication entre tâches successives allouées à des processeurs différents, lequel peut être aussi considéré comme étant zéro.

Les modèles délai et des techniques d'ordonnancement associées ont été proposés simultanément. Pour ne pas anticiper la présentation de l'ordonnancement, nous allons donner une description informelle. Un ordonnancement d'un graphe sur une machine consiste à attribuer à chaque tâche du graphe un processeur et une date de début d'exécution.

Lors de la transmission, ou de la réception, d'un message dans une machine réelle le processeur est occupé pendant une période de temps (avec des copies mémoire, allocation de tampons, etc). Les modèles de cette section ne prennent pas en compte cette période sur le temps d'exécution des processeurs. Le recouvrement total des communications par du calcul est autorisé, c'est-à-dire que les processeurs peuvent calculer pendant les communications. Les modèles délai ne prennent également pas en compte la congestion du réseaux. Le surcoût de communication sur le temps de calcul et la congestion ont été considérés dans le modèle LogP (voir plus loin dans la section 2.4.3).

Nous présentons d'abord le modèle avec bande passante illimitée, c'est-à-dire, sans surcoût de communication. Ensuite nous présentons les modèles

avec bande passante limitée, ce qui introduit des délais de communication. Une des façons d'alléger ce surcoût peut se faire à travers la duplication de tâches. Dans ce cas, nous pouvons au lieu de communiquer à partir des prédécesseurs, dupliquer quelques-uns d'entre eux. Les paramètres que nous utiliserons pour les modèles suivants sont le nombre de processeurs et la possibilité de dupliquer des tâches.

Nous présentons des exemples d'exécutions du graphe G de la figure 2.1 (page 22) sous différents modèles d'exécution. La représentation utilisée est le diagramme de Gantt (diagramme espace temps classique, où l'espace correspond à l'occupation des processeurs), où les tâches, les communications et les temps d'attente sont placés selon leurs dates, processeurs et durées d'exécution. Le poids des sommets du graphe G sont identiques, le poids de ses arcs sera explicité pour chaque modèle. Nous allons présenter des schémas d'exécution sur trois processeurs identiques.

Modèle UET

L'approche théorique de base est simplement d'ignorer le temps de communication entre processeurs. La bande passante est considérée illimitée. Le modèle UET (*unit execution time*) a été proposé par Papadimitriou et Ullman [83]. Le temps d'exécution des tâches est unitaire, les attentes dues aux communications ne sont pas considérées. Il est clair que dans ce cas la duplication de tâches s'avère inutile.

Ce modèle est similaire au modèle PRAM. Dans le diagramme de la figure 2.5 nous présentons un schéma d'exécution optimal.

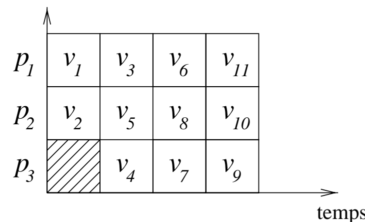

FIG. 2.5 – Exécution dans le modèle UET.

Les zones rayées correspondent au temps d'inactivité, cette notation sera utilisée dorénavant.

Modèle UET-UCT

L'extension naturelle du modèle UET considère de manière simplifiée les communications. Lorsque les temps d'exécution des tâches ainsi que les temps de communication sont unitaires, nous avons le modèle UET-UCT (*unit execution time - unit communication time*). Ce modèle a été proposé par Rayward-Smith [90]. Les schémas d'exécution avec et sans duplication sont représentés dans la figure 2.6.

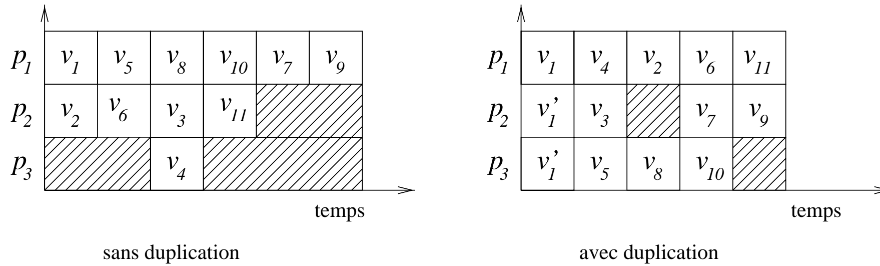

FIG. 2.6 – Exécutions dans le modèle UET-UCT sans et avec duplication.

Dans l'exemple de la figure 2.6, en permettant la duplication, le temps d'exécution a été diminué d'une unité. Dans la figure 2.6 et dorénavant, lorsqu'une tâche est exécutée plusieurs fois, une de ses allocations est dénommé par l'index de la tâche. Les autres exécutions d'une tâche v_i sont désignées par v_i'.

Modèle UET-LCT

Le modèle proposé par Papadimitriou et Yannakakis [84] considère toujours des tâches de durée unitaire, mais le coût de communication est donné par $\gamma > 1$. Ce modèle est dénommé UET-LCT (*unit execution time - large communication time*).

Ce modèle convient aux réseaux d'ordinateurs où les processeurs sont rapides et le réseau représente le point d'étranglement du système. Dans la figure 2.7, le temps de communication entre tâches exécutées sur processeurs distincts est $\gamma = 2,5$, c'est-à-dire deux fois et demi le temps d'exécution d'une tâche.

Jusqu'ici, les coûts des tâches ainsi que les coûts des communications ont été constants. Il existe aussi la possibilité d'avoir des coûts variables.

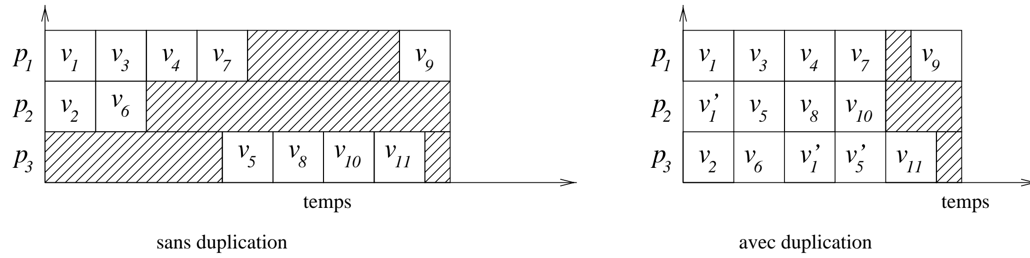

FIG. 2.7 – Exécutions dans le modèle UET-LCT sans et avec duplication.

Modèle SCT

Il existe d'autres approches tel que le modèle SCT (*small communication time*) proposé par Colin et Chrétienne [21]. Dans ce cas, les temps d'exécution sont plus grands que les temps de communication. La figure 2.8 montre des schémas d'exécution du graphe G, avec et sans duplication. Le temps de communication est la moitié de la durée d'une tâche.

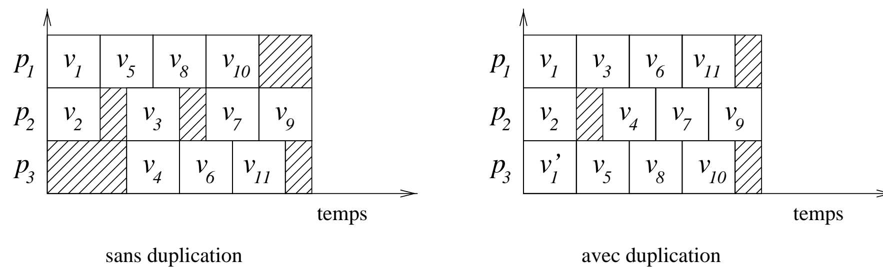

FIG. 2.8 – Exécutions dans le modèle SCT sans et avec duplication.

L'intérêt de l'introduction de plusieurs restrictions sur les modèles délai réside dans la possibilité de pouvoir donner des garanties de performances plus fines pour les problèmes d'ordonnancement.

Modèles plus généraux

Un modèle général où les durées d'exécution des tâches dépendent de leur poids et le coût de communication du poids des arcs a été proposé par Hwang et al. [62]. Ils ont considéré également la topologie de la machine. Les auteurs ont proposé une fonction élémentaire Γ qui associe à chaque paire

de processeurs le coût de communication d'un mot. Il perd en simplicité ce qu'il gagne en réalisme.

Dans les autres modèles avec délai de communication le réseau d'interconnexion n'est pas considéré. La position d'un processeur dans le réseau n'affecte pas le coût de communication.

Dans cette thèse nous allons nous restreindre aux modèles UET, UET-UCT, UET-LCT et SCT. Il est à noter que les modèles avec délai de communication ne considèrent pas le coût de gestion des communications. Ils ne considèrent pas non plus les effets de la congestion. Ces coûts seront considérés dans le modèle LogP qui est inspiré de l'architecture des machines parallèles réelles.

2.4.3 Modèle LogP

Le modèle LogP [24] a le mérite d'avoir été conçu conjointement par des spécialistes en architectures, en environnements d'exécution et en algorithmique. Ce modèle suppose un nombre fini P de processeurs à mémoire locale. La topologie du réseau n'est pas prise en compte. Les synchronisations sont faites par échange de messages. Le temps de communication considère le coût d'échanges de message pour chaque processeur.

Dans le modèle LogP les coûts de communication sont déterminés à travers les paramètres L, o et g. Lors de l'envoi d'un message le processeur expéditeur ne peut pas calculer pendant un période de temps, ce surcoût est dénoté par o (de l'anglais *overhead*). La réception d'un message coûte aussi un temps de calcul o du processeur récepteur. Il existe aussi un intervalle de temps minimal entre l'envoi de deux messages par le même processeur, cet intervalle est dénoté par g (de l'anglais *gap*). Cet intervalle de temps doit aussi être respecté lors de la réception des messages. La latence L est le maximum entre le temps d'envoi d'un message (achèvement de l'opération d'envoi) et le temps de réception de ce message (début de l'opération de réception), sur des conditions de communication normales. Pour éviter la congestion du réseau, au plus $\lceil \frac{L}{g} \rceil$ messages peuvent transiter simultanément.

Dans la figure 2.9 nous illustrons les paramètres du modèle LogP, les "tâches noires" sont dues aux surcoûts de transmission et de réception. Un carré gris représente la latence. Entre deux communications consécutives il existe un intervalle de taille au moins g.

La définition première de LogP a été donnée pour de petits messages.

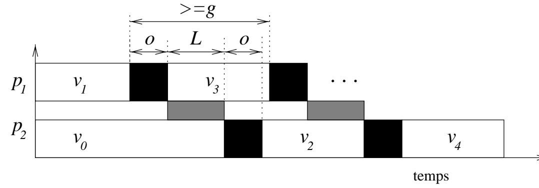

FIG. 2.9 – Exemple des paramètres LogP.

Avec de gros messages, la latence peut devenir négative, le premier mot du message peut arriver avant le départ de son dernier mot. Quelques variations plus générales du modèle LogP ont été proposées [4, 29]. Les modèles avec délai de communication peuvent être vus comme un cas particulier du modèle LogP avec $o = g = 0$.

Les valeurs des paramètres du modèle LogP pour plusieurs machines sont présentés dans la table 2.1. Elles ont été mesurées par plusieurs auteurs [4, 24, 27, 30, 76]. La taille en octets du message transmis est représentée par x. Nous observons que pour la plupart des machines la valeur de g n'est pas très significative (de l'ordre de deux fois o). Nous constatons aussi que le surcoût d'initialisation est assez important.

	L	o	g	P
CM-5	$6\mu s$	$2.2\mu s$	$4\mu s$	512
Parsytec Xplorer	$-21 - 0.82x\mu s$	$70 + x\mu s$	$115 + 1.43x\mu s$	8
ParaStation	$50 - 0.1x\mu s$	$3 + 0.112x\mu s$	$3 + 0.119x\mu s$	4
IBM SP-1	1000 cycles	8000 cycles	-	16
IBM SP-2	$13 - 0.005x\mu s$	$8 + 0.008x\mu s$	$10 + 0.01x\mu s$	32
Meiko CS-2	$8.6\mu s$	$1.7\mu s$	$14.2 + 0.03x\mu s$	64

TAB. 2.1 – Paramètres du modèle LogP sur plusieurs machines.

Les études sur le thème de la duplication de tâches sont pour l'instant restreintes aux modèles avec délai de communication (voir section 2.6). Il n'existe pas des résultats sur l'influence de la duplication sur le coût de communication ni sur le modèle LogP, ni sur les modèles présentés dans les sections suivantes. Dorénavant, dans les exemples nous montrons des schémas d'exécution sans duplication.

La figure 2.10 exhibe deux ordonnancements sous le modèle LogP. Dans le premier (à gauche) $o = 0,125$ et $L = 0,25$ du temps d'exécution d'une

tâche, le paramètre g est au plus 1. Dans le deuxième exemple (à droite) nous utilisons les mêmes valeurs pour o et L, cependant $g = 1,5$. Donc, la communication de v_5 à v_{11} est retardée.

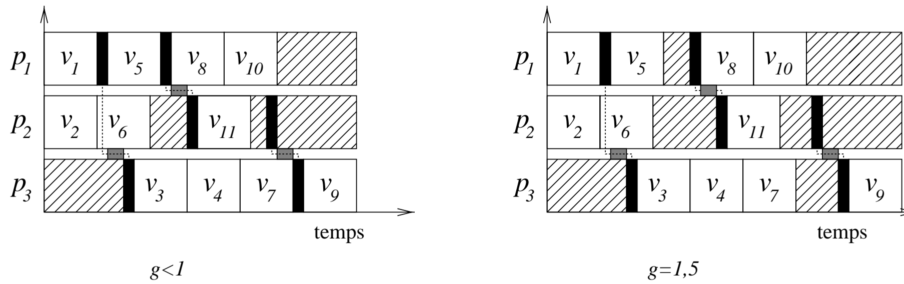

FIG. 2.10 – Exécutions dans le modèle LogP.

Tous les modèles d'exécution présentés jusqu'à présent (sauf le modèle proposé par Hwang et al. [62]) ne considèrent pas la topologie du réseau. Toutefois, une communication entre voisins coûte souvent "moins cher" qu'une communication distante. Il y a deux arguments pour justifier ces modèles. D'une part les modes de communication modernes sont peu sensibles à la notion de localité. D'autre part, ces modèles ont privilégié la simplicité au réalisme. Nous allons conclure la présentation des modèles d'exécution en montrant les façons de considérer d'une manière plus fine le coût des communications.

2.4.4 Modèles plus précis

Une approche consiste à considérer la topologie du réseau ainsi que les contraintes de communication d'une machine réelle. Une bonne approximation du temps de communication entre processeurs voisins est fournie par le modèle linéaire (approximation linéaire du premier ordre). La transmission d'un message dans un réseau sans congestion nécessite une phase d'initialisation (en anglais *start-up*) correspondant au temps d'initialisation des registres mémoires plus le temps nécessaire aux protocoles (empaquetage des données). Ensuite la bande passante est considérée. Dans ce modèle le coût de transmission d'un message de taille S est $t_c = \beta + S\gamma$, où β est le temps d'initialisation, et γ est l'inverse de la bande passante [26]. Dans l'annexe A nous proposons une petite discussion à propos de l'acuité de ce modèle.

La communication entre processeurs non adjacents peut se faire, selon l'architecture de la machine, par des techniques telles que la commutation des messages (*store-and-forward*), la commutation de circuits (*circuit-switching*) et la commutation par paquets (*wormhole*). Un (ou plusieurs) chemins sont choisis entre l'émetteur et le récepteur. Nous présenterons l'estimation du temps de communication lorsqu'un seul chemin est utilisé. Une bonne présentation de ces techniques peut être trouvée dans [26].

La commutation de messages consiste en plusieurs étapes de communication voisin à voisin, à chaque transmission le message est stocké sur les processeurs intermédiaires. Pour une communication à distance d, le temps nécessaire est $d(\beta + S\gamma)$. Lorsque la communication par circuits ou la commutation par paquets est utilisée, un facteur additif relatif à la distance d entre les processeurs impliqués est considéré. Dans ce cas le coût de transmission devient : $t_c = \beta + S\gamma + d\delta$, où δ est le temps induit par la commutation d'un routeur sur chaque processeur intermédiaire.

Par rapport au modèle LogP, ce modèle considère la topologie du réseau, cependant ni les surcoûts dûs aux envois et réceptions de messages sur le temps de calcul des processeurs ni la congestion ne sont pas pris en compte.

Il existe d'autres modèles relatifs aux contraintes de communication, si par exemple, les liens sont bidirectionnels, ou si le nombre de liens qu'un processeur peut utiliser simultanément est limité. Mais une description aussi détaillée dépasse le cadre de cette thèse.

Ces modèles ont été étudiés plutôt pour les algorithmes de communication globale dans une topologie spécifique [34, 58]. Plusieurs résultats sur des communications comme la diffusion, l'échange total, la distribution, le rassemblement et la multidistribution ont été traités il y a une dizaine d'années. Généralement les études se concentrent sur une topologie cible, tels que l'hypercube, l'anneau, la grille ou le tore entre autres. Avec l'évolution des machines, ces modèles ne sont plus pertinents pour représenter les mécanismes matériels, même s'ils peuvent encore être utiles du point de vue algorithmique.

Dans les modèles suivants, contrairement à une approche qui provient des caractéristiques de la machine, nous allons analyser différentes possibilités pour la programmation.

2.4.5 Modèle BSP

Plus qu'un modèle d'exécution, BSP [59, 78, 99] (*Bulk Synchronous Parallel*) est un modèle de programmation. Son objectif est de fournir un cadre permettant de concevoir facilement des algorithmes portables et efficaces. Le modèle BSP n'est pas basé sur des modèles de machines existantes, mais il convient aux machines MIMD.

L'idée principale du modèle BSP est la séparation du calcul de la communication. Ses concepts de base sont la super-étape (de l'anglais *super-step*) et la synchronisation. L'application est divisée en super-étapes. Tous les processeurs commencent une super-étape au même instant. Entre deux super-étapes, il existe une étape de synchronisation. Les données communiquées lors d'une super-étape seront disponibles aux processeurs destinataires au début de la super-étape suivante. Les trois paramètres utilisés afin de décrire le modèle sont p, l et g. Nous utilisons les mêmes notations que ces utilisés dans [78], malgré l'utilisation de g auparavant pour le description du modèle LogP. Ce choix est motivé par la standardisation de ces notations pour les deux modèles.

p le nombre de processeurs de la machine ;

l le coût d'une synchronisation globale ;

g le temps de transport d'un mot par le réseau. Autrement dit, $\frac{1}{g}$ est la bande passante.

Le modèle original proposé par Valiant [99] introduit un paramètre qui représente la périodicité d'une synchronisation. Dans son modèle, une vérification globale est effectuée après chaque période de L unités de temps. Elle sert à déterminer si la super-étape a été achevée sur tous les processeurs. Dans la version du modèle présentée par McColl [78] il n'y a pas de référence à la périodicité. Cette approche a été probablement choisie parce que dans les machines courantes la périodicité peut être aussi petite que le coût de synchronisation. C'est cette approche que nous avons adoptée dans ce document.

Pour pouvoir estimer le temps d'une application BSP nous introduisons les terminologies suivantes :

- p_i - processeurs de la machine $(0 \leq i < p)$;
- w_i^s - coût des calculs exécutés par le processeur p_i au cours de la super-étape s ;
- h_i^s - désigne le maximum du nombre de mots reçus (ou envoyés) par le

processeur p_i au cours de la super-étape s.

Une machine dans le modèle BSP est capable d'accomplir une $\lceil \frac{l}{g} \rceil$-relation dans chaque super-étape, cette contrainte est similaire à celle proposée dans le modèle LogP. Une h-relation est une opération d'échanges de données point à point entre les processeurs, où chaque processeur peut envoyer et recevoir au plus h mots. L'estimation de coût d'une h-relation faite par McColl [78] était donnée par hg. Cette estimation sert de borne inférieure, mais pour les machines réelles, l'estimation de Eisenbiegler, Loẅe et Zimmermann [28] de $2hg$ est plus précis. Une discussion à propos de l'estimation de coût d'une h-relation est présentée dans [59], en effet les différentes estimations ne changent le coût que d'un petit facteur constant.

Le coût de communication d'une super-étape s peut être estimé par $2h^s g$, où $h^s = \max_{i=0}^{p-1} h_i^s$. Cette estimation est bonne si le réseau n'est pas congestionné. Le coût d'une super-étape s est borné par $c = max_{0 \leq i < p} w_i^s + 2h^s g + l$ [28]. L'estimation du temps total d'un programme BSP est obtenu en sommant les temps de ses super-étapes.

Avec un regard plus attentif sur la borne de coût d'une super-étape, nous observons que le coût d'initialisation lors d'un envoi de message (voir la section suivante) n'est pas considéré. Cependant ce coût est inclut dans le coût de synchronisation.

Les valeurs des paramètres du modèle BSP pour plusieurs machines sont présentés dans la table 2.2[2]. Dans la deuxième colonne nous présentons la vitesse du processeur en Mflops (de l'anglais *FLoating point OPerations per Second*). g est donné en flop par mot et l en flop. Nous constatons que le coût d'une synchronisation est très important pour toutes les machines de la table.

	(Mflops)	g (flop/mot)	l (flop)
400Mhz PII, 100Mb ether. (8 procs.)	88	30.9	18347
Cray T3D, 150Mhz (256 procs.)	12	2.4	387
Cray T3E, 300Mhz(20 procs.)	47	1.63	880
IBM SP2, 66.7Mhz (8 procs.)	26	11.4	5412
Parsytec (8 procs.)	19.3	25.4	29080

TAB. 2.2 – Paramètres du modèle BSP sur plusieurs machines.

Les démarches pour obtenir une application efficace dans le modèle BSP

[2]Source : `www.BSP-Worldwide.org/implmnts/oxtool/params.html`

sont donc : équilibrer la charge de calcul entre les processeurs au cours de chaque super-étape, équilibrer les communications au cours d'une super-étape (éviter les congestions) et finalement minimiser le nombre de super-étapes.

Pour les schémas d'exécution dans le modèle BSP nous avons choisi une représentation légèrement différente de la représentation traditionnelle (figure 2.11 à gauche). Cette représentation explicite les communications entre processeurs. Cependant l'un des avantages majeurs du modèle BSP est d'effectuer des communications de façon implicite entre super-étapes. Donc, nous avons choisi de représenter les communications entre processeurs comme une barrière grise (correspondant à la h-relation) qui se trouve juste avant la synchronisation (barrière noire dans la figure 2.11 à droite). De cette façon les communications restent implicites. Dans l'exemple, le temps pour achever une h-relation est 0,5 du temps d'exécution d'une tâche. Le temps de synchronisation est 0,25.

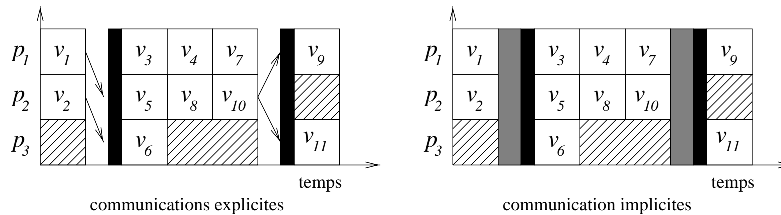

FIG. 2.11 – Schémas d'exécution dans le modèle BSP.

2.4.6 Tâches malléables

Les tâches malléables [98] (de l'anglais *malleable tasks*) est un modèle où les tâches du graphe de précédence peuvent elles mêmes s'exécuter en parallèle. Pour considérer le surcoût d'exécution d'une tâche sur plusieurs processeurs une fonction d'inefficacité $\mu(p, N)$ est introduite, où p est le nombre de processeurs utilisés lors de l'exécution parallèle d'une tâche et N est le nombre d'instructions de la tâche. Généralement, cette fonction augmente avec le nombre de processeurs et diminue avec la taille de la tâche.

La figure 2.12 illustre un exemple d'allocation de tâches malléables. À gauche les cinq tâches malléables sont alloués une par processeur. À droite est présentée une allocation où l'inefficacité est visible, pour chaque tâche

exécutée sur plus d'un processeur un surcoût est ajouté. Par exemple la tâche alloué à gauche sur P_6 est exécutée en 5 unités de temps sur 4 processeurs à droite, et donc l'inefficacité est $\mu(4, 18) = 20 - 18 = 2$.

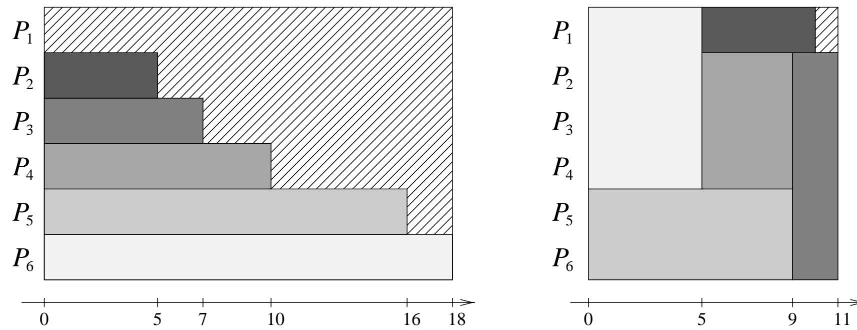

FIG. 2.12 – Schéma d'exécution de tâches malléables.

2.5 Ordonnancement

L'exécution d'un programme sur une machine parallèle sera vu, dans le cadre de cette thèse, comme l'ordonnancement d'un graphe de précédence selon un modèle d'exécution. Un ordonnancement consiste à attribuer à chaque tâche un, ou plusieurs, processeurs cibles et les dates d'exécution correspondantes ; cette opération s'appelle allocation d'une tâche. Un ordonnancement pour un graphe $G(V, E)$ est valide s'il vérifie que :

- Chaque tâche $v \in V$ est exécutée par au moins un processeur,
- A chaque instant t chaque processeur exécute au plus une tâche. Dans les modèles LogP et BSP, pour rester cohérent avec la notion de tâche, nous ajoutons des tâches de préparation à la communication (LogP) et de communication et de synchronisation (BSP),
- Si une tâche v débute son exécution à l'instant t sur un processeur p, toutes les données nécessaires à l'exécution de v sont déjà disponibles sur p à l'instant t.

Le temps total d'un ordonnancement est le maximum des dates de fin d'exécution de toutes les tâches. Cette quantité est souvent désignée par le terme *makespan*. Un ordonnancement optimal est un ordonnancement valide avec le plus petit *makespan*.

Notre intérêt est la recherche d'algorithmes qui fournissent des ordonnancements à partir de classes de graphes de précédence spécifiques, qui proviennent des algorithmes à exécuter en parallèle.

Nous conseillons au lecteur intéressé par une étude approfondie de l'ordonnancement de se rapporter aux ouvrages [9, 18, 89, 93].

2.5.1 Évaluation pratique des applications parallèles

Parmi les principaux critères d'évaluation d'une application parallèle nous pouvons citer l'efficacité et la garantie de performance. L'efficacité compare le temps de l'application parallèle avec le temps de la résolution séquentielle. La garantie de performance confronte dans les mêmes conditions l'application parallèle avec une application parallèle de plus petit makespan. Le choix du critère le mieux adapté va dépendre de l'utilisateur.

Efficacité

Pour comparer les algorithmes parallèles entre eux et avec les algorithmes séquentiels nous allons introduire la notion d'efficacité. Intuitivement, plus l'exécution d'un programme parallèle est efficace plus les ressources de calcul sont pleinement utilisées.

Étant donnée le graphe de précédence, le travail total t_s correspond à la somme du poids de toutes les tâches. Ce travail est le temps d'exécution séquentiel du graphe. Dans un ordonnancement sur m processeurs avec *makespan* w, nous utilisons la terminologie suivante : pour chaque processeur $p_i (0 \leq i < m)$ dans l'intervalle de temps $[0, w]$, soit

- $T(i)$, la somme du poids des tâches exécutées par p_i ;
- $T_c(i)$, le temps de calcul du processeur p_i utilisé pour réaliser et gérer les communications et les synchronisations ;
- $Id(i)$, le temps d'inactivité du processeur p_i.

Lors d'un ordonnancement aucune nouvelle tâche de calcul est créée, donc il est logique de s'attendre à ce que la somme de $T(i)$ pour tous les processeurs soit égale au travail du graphe de précédence t_s, c'est-à-dire $(t_s = \sum_{i=0}^{m-1} T(i))$. Cependant une même tâche peut être exécutée plusieurs fois indépendamment, dans ce cas, seul le temps d'une de ces exécutions est considéré comme temps de calcul, les autres temps sont comptés dans le temps d'inactivité.

Les temps totaux d'inactivité Id et de communication T_c correspondent respectivement à $\sum_{i=0}^{m-1} Id(i)$ et $\sum_{i=0}^{m-1} T_c(i)$. La somme de ces deux temps est le surcoût (de l'anglais *overhead*) d'une application parallèle. Un programme parallèle idéal aurait un temps d'exécution de $\frac{t_s}{m}$, dans ce cas le travail du graphe est réparti uniformément parmi les m processeurs sans surcoût. L'efficacité est définie par rapport au temps d'exécution séquentiel :

$$E = \frac{t_s}{t_s + T_c + Id}. \tag{2.1}$$

Dans l'expression 2.1 nous calculons la somme du temps passé dans chaque état (ce qui correspond à la somme des surfaces de calcul, de communication et d'inactivité dans les diagrammes de Gantt). Dans l'équation 2.2, le *makespan* est multiplié par le nombre de processeurs (ce qui correspond à la surface totale dans le diagramme de Gantt). Le deux expressions sont équivalentes.

$$E = \frac{t_s}{mw}. \tag{2.2}$$

Le surcoût de l'ordonnancement, $T_c + Id$, dépends d'un compromis entre les valeurs de T_c et de Id. Si nous voulons un ordonnancement où $T_c = 0$, la minimisation du temps d'inactivité revient généralement à un problème $\mathcal{NP}$-difficile (problème de complexité au moins aussi difficile que le problème de la partition [39]). En outre, dans le meilleur ordonnancement avec $T_c = 0$, la valeur de Id peut être importante. D'autre part, avec l'ajout de communications ($T_c > 0$) nous pouvons espérer réduire le temps d'inactivité. Donc, pour minimiser le surcoût il faut trouver un compromis entre T_c et Id.

Garantie de performance

Nous allons voir dans la section suivante que la plupart des problèmes de recherche d'ordonnancements optimaux appartient à classe $\mathcal{NP}-$difficile. Dans ce cas nous nous intéressons à des heuristiques. La qualité d'une heuristique est mesurée par son rapport de performance à l'optimal. De cette façon il est possible de fournir des garanties de performance.

Le rapport de performance d'un algorithme d'ordonnancement est donné par w/w_{opt}, où w est le *makespan* de l'ordonnancement calculé par l'algorithme et w_{opt} le *makespan* d'un ordonnancement optimal.

2.6 Résultats d'ordonnancement

Pour terminer ce chapitre nous présentons un tour d'horizon des résultats d'ordonnancement dans les principaux modèles étudiés.

2.6.1 Algorithmes de liste

Une grande partie des algorithmes d'ordonnancement est basée sur des algorithmes de liste, lesquels maintiennent une liste des tâches prêtes à ordonnancer selon leur priorité. Les grands avantages des algorithmes de liste sont la simplicité et l'existence de garanties de performance. Par exemple, si il n'y a pas de délai de communication entre processeurs les algorithmes de liste ont une garantie de performance $2 - \frac{1}{m}$ [55] pour un graphe quelconque. Un algorithme de liste répète les pas suivants jusqu'à l'allocation de toutes les tâches :

1. Les tâches prêtes sont placées dans une liste de priorité. Une tâche est prête lorsque tous ses prédécesseurs ont été déjà alloués ;
2. Un processeur approprié est choisi. En général, un processeur approprié est celui qui peut exécuter la tâche au plus tôt ;
3. La tâche en tête de la liste est alloué sur le processeur choisi.

2.6.2 PRAM

Le modèle PRAM s'est révélé un outil extrêmement bien adapté pour l'étude de la complexité parallèle. Ce modèle a permis, entre autre, la définition de classes de complexité NC^i, $1 \leq i \leq \infty$ [82]. Il existe un grand nombre de résultats d'ordonnancement célèbres pour le modèle PRAM [65]. Nous ne détaillerons pas cela ici car nous nous sommes intéressés plutôt aux modèles à mémoire distribuée.

2.6.3 Modèles avec délai de communication

Dans le but de simplifier la description des problèmes, nous introduisons la notation à trois champs proposée par Veltman et al. [100]. La notation $\alpha_1|\alpha_2|\alpha_3$ où α_1 décrit les ressources, α_2 donne le graphe à ordonnancer et α_3 le critère d'optimisation. Pour les ressources, nous pouvons expliciter le modèle et le nombre de processeurs. Si le nombre de processeurs est illimité,

le premier champ s'écrit P_∞, sinon il s'écrit P (nombre arbitraire de processeurs) ou P_m (nombre fixé de processeurs). Le modèle par défaut est le modèle délai. Dans la description du graphe on précise d'abord sa structure, par exemple *prec* pour les graphes de précédence généraux et *arbre* pour les arbres, si aucune précision n'est donnée les tâches sont indépendantes. Dans le champ α_2, p_j correspond au poids des tâches et c_{ik} au poids des arcs. Si tous les arcs ont le même poids, il est dénoté par C. Au cours de cette thèse, le critère d'optimisation que nous avons utilisé est la minimisation du *makespan*, symbolisé par $w_{\max}$.

Selon la disponibilité des ressources, les problèmes d'ordonnancement s'avèrent plus ou moins faciles. Intuitivement, il est plus facile de résoudre un problème avec un nombre illimité de processeurs. Un autre paramètre important est la possibilité de faire la duplication des tâches, dans ce cas nous pouvons espérer un allégement du surcoût dû aux communications. Nous organisons la présentation selon la disponibilité de ressources et la possibilité de dupliquer. Les modèles sont organisés en ordre croissant de difficulté, c'est-à-dire du moins au plus contraignant. Nous commençons avec un nombre de processeurs non borné en permettant la duplication. Après, nous interdisons la duplication. Ensuite, le nombre de processeurs est borné et la duplication permise. Enfin, le nombre de processeurs est borné et la duplication est interdite.

Duplication et nombre illimité de processeurs

Pour les problèmes $P_\infty|prec, p_j, c_{ik}, dup|w_{\max}$, c'est-à-dire, nombre de processeurs non borné et duplication autorisée, il existe deux résultats très importants. Lorsque la granularité est grosse, $P_\infty|prec, p_j, c_{ik}, SCT, dup|w_{\max}$, un algorithme polynômial qui trouve la solution optimal pour un graphe de précédence quelconque a été proposé par Colin et Chrétienne [21]. Papadimitriou et Yannakakis [84] on montré que le problème $P_\infty|prec, p_j = 1, c > 1, dup|w_{\max}$ est $\mathcal{NP}$-difficile. Ils ont aussi proposé un algorithme polynômial d'approximation avec rapport de performance 2 dans le cas général (p_j et c_{ik} quelconques).

Plusieurs autres algorithmes d'approximation ont été proposés dans la littérature. Park, Shirazi et Marquis [85] ont présenté un état de l'art avec un résumé des principaux algorithmes. Ils ont aussi proposé un algorithme d'approximation pour le problème $P_\infty|prec, p_j, c_{ik}, dup|w_{\max}$. Le *makespan* de l'ordonnancement construit est au plus $w_{opt}^\infty + C_{comm}$, où C_{comm} est la

somme des communications dans le chemin critique et w_{opt}^{∞} le *makespan* d'un ordonnancement optimal. Dans ce cas, au lieu de présenter un rapport de performance constant [84], le rapport dépend de la somme des communications dans le chemin critique.

Nombre illimité de processeurs sans duplication

Lorsque la duplication n'est plus autorisée ($P_\infty|prec, p_j, c_{ik}|w_{\max}$), le problème devient plus dur. Nous présentons quelques-uns des résultats connus. Dans les modèles très restreints, si le graphe de précédence est quelconque, le problème est $\mathcal{NP}-$difficile. $P_\infty|prec, p_j = 1, c = 1|w_{\max}$ (ordonnancement dans le modèle UET-UCT) est $\mathcal{NP}-$difficile [88]. Même si le graphe de précédence est un arbre (*out-tree*) du type harpon (une découpe des chaînes de longueur deux) le problème est $\mathcal{NP}$-difficile [18]. Il existe des ordonnancements pour des classes de graphes simples sous l'hypothèse SCT [18]. Un algorithme d'approximation avec garantie de performance $(1 + 1/\rho)$, dénommé DSC (de l'anglais *Dominant Sequence Clustering*), a été proposé par Gerasoulis et Yang [42], où ρ est une fonction qui donne la granularité du graphe, $\rho = \min p_j / \max c_{ki}$.

Duplication et nombre de processeurs limité

Les problèmes avec nombre de processeurs borné et duplication ont été très peu étudiés, à notre connaissance ce problème n'a été analysé que dans le cas UET-UCT par Hanen et Munier [57]. Les auteurs ont proposé un algorithme de liste pour $P|prec, p_j = 1, c = 1|w_{\max}$, avec critère de priorité quelconque et rapport de performance $2 - \frac{1}{m}$ où m est le nombre de processeurs, c'est-à-dire le même rapport de performance que dans le modèle sans coût de communication.

Nombre limité de processeurs sans duplication

Quand le nombre de processeurs est limité et la duplication n'est pas permise, la majorité des problèmes sont $\mathcal{NP}$-difficiles. Les problèmes sont au moins aussi difficiles que sans délais de communication où le problème de l'ordonnancement optimal d'un graphe quelconque est $\mathcal{NP}$-difficile ($P|prec, p_j = 1, c_{ik} = 0|w_{\max}$ est $\mathcal{NP}$-difficile [90]). Le problème de l'ordonnancement de tâches indépendantes sur deux processeurs $2|p_j|w_{\max}$ est aussi $\mathcal{NP}$-difficile.

Hoogeveen, Lenstra et Vetman [60] ont démontré que décider si le problème $P|biparti, p_j = 1, c = 1|w_{\max}$ a un ordonnancement avec *makespan* au plus 4 est un problème $\mathcal{NP}$-complet. Ce qui implique qu'il n'existe pas d'algorithme polynômial avec rapport de performance meilleur que $\frac{5}{4}$ à moins que $\mathcal{P} = \mathcal{NP}$.

Pour le problème $P|prec, p_j = 1, c_{ik} = 1|w_{\max}$ les algorithmes de liste ont un rapport de performance $3 - \frac{2}{m}$ [90] et cette borne est atteinte. Le meilleur algorithme d'approximation pour le modèle SCT a une garantie de performance de $\frac{7}{3} - \frac{4}{m}$ (ce résultat est une conséquence directe de [56]).

Dans le cas général, $P|prec|w_{\max}$, les algorithmes de liste ont pour performance $(2 - \frac{1}{m})w_{opt} + C_{comm}$ [71, 75], où C_{comm} est la somme des communications dans le chemin critique et w_{opt} le *makespan* d'un ordonnancement optimal sur m processeurs.

2.6.4 LogP

La plupart des travaux réalisés concernent des graphes spécifiques (notamment des arbres). Verriet [101] a étudié les ordonnancements d'un graphe *fork*, *join* et arbres pour un nombre non borné de processeurs. Loẅe, Middendorf et Zimmermann [79] ont présenté un algorithme polynômial pour l'ordonnancement linéaire d'arbres. Quelques résultats sur l'ordonnancement des arbres compacts (de l'anglais *flat trees*) on été présentés par Kort et Trystram [68]. Par contre, la recherche d'ordonnancements optimaux arbitraires LogP$|prec|w_{\max}$ est un problème $\mathcal{NP}-$difficile [101]. Une étude intéressante sur l'exécution de programmes PRAM sur des machines LogP a été réalisée par Löwe et Zimmermann [77]. Kalinowski, Kort et Trystram ont proposé une analyse d'une heuristique générale de liste à partir de ETF [62] sous LogP [63]. ETF (de l'anglais *Early Task First*) est un algorithme de liste où le critère de priorité est de choisir la tâche qui peut être ordonnancée au plus tôt.

2.6.5 BSP

L'ordonnancement sous le modèle BSP a été étudié en particulier par le groupe d'applications parallèles de l'université d'Oxford[3]. L'ordonnancement de graphes acycliques directs uniformes (de l'anglais *uniform directed*

[3]http://www.BSP-Worldwide.org/implmnts/oxtool.htm

acyclic graphs), aussi connus par *tightly-nested loops* a été analysé par Calinescu [14, 15]. Des ordonnancements pour des algorithmes classiques issus de méthodes connues en algèbre linéaire matricielle ont été considérées par McColl [78]. Il a présenté des algorithmes pour : la multiplication matrice-vecteur, la multiplication des matrices, la décomposition LU, la résolution d'un système linéaire triangulaire et la multiplication d'une matrice creuse par un vecteur.

Une extension du modèle BSP (dénommé BSPRAM) à été présentée par Tiskin [97]. Dans le même travail, un algorithme pour ordonnancer le graphe orienté acyclique *buterfly* a été proposé. Il a aussi considéré des algorithmes pour la multiplication des matrices denses et pour le tri. Pour l'instant il n'existe pas de résultats pour des graphes plus généraux.

2.7 Conclusion

Le premier constat de ce chapitre est qu'il existe un rapport étroit entre le réalisme des modèles et leur utilisation. Les modèles où les supports d'exécution sont considérés d'une façon simplifiée sont les plus étudiés dans le cadre de l'ordonnancement statique. Les algorithmes d'ordonnancement développés pour ces modèles sont habituellement utilisés en pratique sur machines réelles sans garantie de performance et pas toujours de manière efficace. Par contre, le travail sur des algorithmes d'ordonnancement pour les modèles plus précis (qui considèrent la topologie de la machine cible et d'autres caractéristiques de l'architecture) est resté restreint. Il nous semble que le compromis entre réalisme et simplicité a été trouvé pour les modèles à gros grain comme BSP et les tâches malléables qui prennent en considération plusieurs paramètres des machines réelles. Cependant, ces modèles n'ajoutent pas trop de détails qui compliqueraient trop les problèmes d'ordonnancement. Pour LogP, bien que les problèmes soient plus difficiles, en pratique, il est possible d'adapter les algorithmes simples comme les algorithmes de liste.

L'autre constat est l'influence de la duplication sur les modèles d'exécution avec délai de communication. Cependant il existe encore des problèmes ouverts quand le nombre de processeurs est limité. La duplication joue un rôle très important, particulièrement quand les délais de communication sont grands. Nous avons vu aussi que la recherche d'ordonnancements sous les modèles LogP et BSP en est à ses débuts, car ces modèles sont relativement nouveaux. La même chose est vraie pour les tâches malléables.

Une voie très intéressante pour le futur est l'étude de l'ordonnancement sur des modèles qui prennent en compte l'hétérogénéité des machines parallèles (de l'anglais *cluster computing*). Car une des tendances des architectures parallèles actuelles est l'interconnexion de plusieurs ordinateurs qui ne sont pas forcement du même type.

Chapitre 3

Modèle à grain fin

Résumé

Dans ce chapitre nous étudions un problème d'ordonnancement avec grain de calcul fin. Le graphe est composé de tâches simples qui exécutent un petit nombre d'opérations arithmétiques. Dans l'ordonnancement de ce type de graphe, le coût de communication peut s'avérer être un facteur très important car il peut devenir extrêmement pénalisant. Plusieurs modèles sont bien adaptés aux problèmes à grain fin. Entre autres nous pouvons citer le modèle systolique, le modèle PRAM et les modèles SIMD. Dans ces modèles, intrinsèquement synchrones, les communications sont incluses dans les pas de calcul élémentaires et ne sont donc pas prises en compte explicitement.

3.1 Problème du sac à dos

Nous analysons dans ce chapitre la résolution du problème du sac à dos en parallèle. Nous nous sommes intéressés à sa résolution exacte. Ce problème est bien connu, il est un classique en optimisation combinatoire. Nous considérons le problème du sac à dos où le poids des objets et la capacité du sac sont des entiers. Ce problème est $\mathcal{NP}$-complet même quand le nombre d'objets de chaque type est limité à 1 [39].

Une instance $Knap(A, c)$ du problème du sac à dos consiste en un ensemble A composé de m types d'objets, et de la capacité du sac à dos, dénoté par c. À chaque objet du type a_i de l'ensemble A correspond un poids entier positif w_i. Pour chaque objet du type a_i inséré dans le sac, un profit p_i est acquis. Le but est de remplir le sac à dos, sans dépasser sa capacité, de façon à

maximiser le profit total. Il n'existe pas de restriction sur le nombre d'objets d'un même type. Le problème peut être formalisé par :

$$\begin{array}{l} \text{trouver des entiers } x_i \geq 0, \\ \text{pour maximiser } \sum_{i=0}^{m-1} p_i x_i \\ \text{soumis à la contrainte } \sum_{i=0}^{m-1} w_i x_i \leq c. \end{array} \quad (3.1)$$

Lors de la présentation des solutions parallèles existantes nous expliciterons le problème traité selon les notations *Knap* ou *0/1-Knap*. Où *Knap* (*0/1-Knap*) désigne le problème du sac à dos entier avec un nombre illimité (limité à un) d'objets de chaque type.

En plus des paramètres du sac à dos, au cours de ce chapitre nous utilisons la terminologie suivante :

- p - nombre de processeurs disponible ;
- $w_{\min}$ - le poids de l'objet le plus léger, c'est-à-dire $\min_{i=0}^{m-1}\{w_i\}$;
- $w_{\max}$ - le poids de l'objet le plus lourd, c'est-à-dire $\max_{i=0}^{m-1}\{w_i\}$.

3.2 Solutions parallèles

Plusieurs approches en algorithmique parallèle ont été proposées pour la résolution exacte du problème du sac à dos entier. Les principales études sont fondées sur : l'énumération des solutions possibles, les techniques de liste [1], la réduction à d'autres problèmes connus et la programmation dynamique. Un résumé sur les diverses solutions proposées a été publié par Gerarsch et Wang [41].

La méthode naïve où toutes les solutions du *0/1-Knap* sont calculées a été implanté par Gerarsch et Wang [40]. L'inconvénient de cette méthode est le besoin, pour calculer une instance avec m objets, de 2^m processeurs. Une approche plus adaptée par séparation-évaluation (de l'anglais *branch-and-bound*) a été proposée par Kindervater et Trinekens [66].

Pour éviter conflit de notation avec les algorithmes de liste (section 2.6.1) la présentation des techniques de liste est restreinte à ce paragraphe. Les algorithmes de liste ont été proposés par plusieurs auteurs pour résoudre une variation du *0/1-Knap*, où la somme du poids des objets doit être égale à la capacité du sac à dos c. La technique consiste à diviser le problème original en problèmes de taille identique. Chaque problème est calculé par l'énumération

[1] Malgrée la similaritée de nom avec les algorithmes de liste, il n'y a pas de rapport entre eux.

de toutes combinaisons possibles ($2^{m/2}$). L'algorithme de liste séquentiel a été proposé par Horowitz et Sahni [61], dans une parallélisation proposée par Cosnard, Ferreira et Herbelin [23] les solutions possibles sont partitionées entre les processeurs disponibles. D'autres variations plus efficaces ont aussi été proposées, entre autres [31, 64]. Le principal revers de l'approche qui utilise les algorithmes de liste est que le temps d'exécution, et/ou le nombre de processeurs requis sont des fonctions exponentielles du nombre d'objets.

L'approche proposée par Teng [95] consiste à réduire *Knap* à des problèmes connus, telles que "*Circuit Value*" et "*Prefix Convolution*" . Cependant, les solutions proposées requièrent au moins c processeurs.

Il existe plusieurs études adaptées au modèle systolique. Chen et al. [16] ont proposé des algorithmes *pipeline* pour les problèmes *Knap* et 0/1-*Knap*. L'algorithme qui résout *Knap* est fondé sur un vecteur linéaire de files d'attente. L'algorithme pour 0/1-*Knap* emploie un vecteur linéaire avec plusieurs niveaux. Andonov et Quinton [5] ont proposé un algorithme efficace pour un chemin systolique avec mémoire limitée. Aleksandrov et Fidanova [3] ont utilisé un vecteur systolique pour résoudre des problèmes similaires au *Knap*. Goldman et Fidanova [43] ont proposé une solution pour le même problème avec un peu moins de processeurs.

L'approche la mieux adaptée à la résolution efficace du problème du sac à dos en parallèle avec un nombre de processeurs donné est la programmation dynamique [41], qui a complexité séquentielle $O(mc)$. Dans ce cas, deux algorithmes pour le problème *0/1-Knap* ont été proposés pour le modèle PRAM. Lee, Shragowitz et Sahni [72] ont utilisé le paradigme diviser pour régner. Ils ont présenté un algorithme pour p processeurs avec temps total $O(\frac{mc}{p} + c^2)$. Lin et Storer [74] ont proposé un algorithme de complexité $O(\frac{mc}{p})$.

Les deux études ont implantée les algorithmes sur une machine avec la topologie de l'hypercube. Dans un hypercube avec p processeurs le diamètre, distance entre les processeurs plus éloignés, est $\log_2 p$. La première étude a considérée la distance inter processeurs, donc la performance est restée inchangée. Ce ne pas le cas de la deuxième étude qui a obtenue ainsi des dégradation des performances de $O(\log p)$, c'est-à-dire une complexité $O(\frac{mc}{p} \log p)$.

Nous proposons dans ce chapitre deux algorithmes qui calculent la solution du problème *Knap* sur l'hypercube. Le premier résout le problème avec p processeurs en temps $O\left(\frac{mc}{p}\frac{w_{\max}}{w_{\min}}\right)$; observons que le rapport $\frac{w_{\max}}{w_{\min}}$ peut être beaucoup plus grand que 1. La complexité du deuxième algorithme est

$O\left(\frac{mc}{p}\right)$.

3.3 Approche par programmation dynamique

L'idée de la programmation dynamique consiste à présenter une fonction $f(k,g)$ pour k et g entiers, où k et g sont, respectivement, le plus grand index d'objet considéré et la capacité du sac à dos. Nous avons alors :

$$f(k,g) = \max\left\{\sum_{i=0}^{k} p_i x_i | \sum_{i=0}^{k} w_i x_i \leq g, x_i \text{ (entier)} \geq 0, i = 0, \ldots, k\right\}.$$

Les conditions limites de la fonction $f(k,g)$ sont obtenues avec zéro objet et capacité du sac à dos quelconque, et avec capacité du sac zéro et nombre d'objets quelconque, c'est-à-dire :

$$f(-1,g) = 0, \forall g \in [0,c];$$
$$f(k,0) = 0, \forall k \in [0,m].$$

La valeur $f(k,g), w_k \leq g \leq c, 0 \leq k < m$ peut être calculée récursivement, il suffit de choisir le maximum entre $f(k-1,g)$ et $p_k + f(k, g-w_k)$. Quand g est petit ($< w_k$), nous ne pouvons pas ajouter un objet de poids w_k dans le sac, donc la valeur de $f(k,g), 0 \leq g < w_k, 0 \leq k < m$ est $f(k-1,g)$. Dans le but de simplifier la formule, nous ajoutons la condition de bord $f(k,y) = -\infty, 0 \leq k < m; y < 0$ qui exprime que quand le sac à dos possède une capacité négative, le profit est $-\infty$. Nous avons alors, immédiatement :

$$f(k,g) = \max\{f(k-1,g), p_k + f(k, g-w_k)\} \text{ avec } \begin{cases} f(-1,g) = 0 \\ f(k,0) = 0 \\ f(k,y) = -\infty, y < 0. \end{cases}$$

Il est facile de constater que la complexité pour le calcul de $f(m-1,c)$ est $O(mc)$.

3.3.1 La grille irrégulière

Nous introduisons ici la grille irrégulière, laquelle correspond au graphe de précédence de la fonction de programmation dynamique du sac à dos $f(k,g)$.

Définition 3.1 *Une grille irrégulière en deux dimension, $NM(m,c)$, avec des sauts $(w_i)_{i=0}^{m-1}$ est le graphe orienté avec des sommets (i,j) (pour $0 \leq i \leq m-1$ et $0 \leq j \leq c$). Un sommet (i,j) est l'origine de au plus deux arcs : si $0 \leq i < m-1$ il y a un arc du sommet (i,j) au sommet $(i+1,j)$, si $0 \leq j \leq c - w_i$ il y a un arc du sommet (i,j) au sommet $(i, j+w_i)$.*

Les lignes de $NM(m,c)$ sont dénotées par des ensembles $L_i = \{(i,j), 0 \leq j \leq c\}$. La figure 3.1 présente une grille irrégulière avec quatre lignes et douze colonnes et correspond au graphe de précédence du $Knap(A, 11)$, avec des objets de taille deux, quatre, cinq et sept.

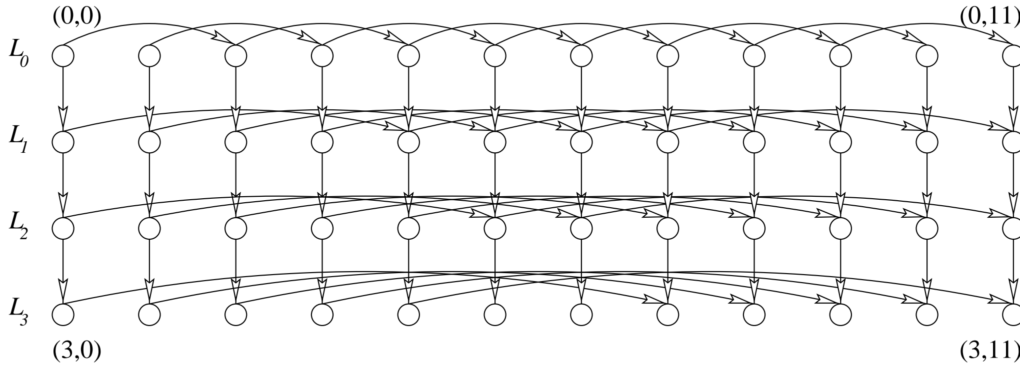

FIG. 3.1 – Grille irrégulière $NM(4,11)$ avec sauts $(w_i)_{i=0}^{3} = (2,4,5,7)$.

Le graphe de précédence pour calculer $f(m-1,c)$ est une grille irrégulière $NM(m,c)$ avec sauts $(w_i)_{i=0}^{m-1}$. Chaque tâche (i,j) calcule :

Si $i = 0$ et $j < w_0$	si $(m \geq 2)$ envoie 0 à $(i+1,j)$
	si $(j \leq c - w_0)$ envoie p_0 à $(i, j+w_0)$.
Si $i = 0$ et $j \geq w_0$	reçoit d de $(i, j-w_0)$
	si $(m \geq 2)$ envoie d à $(i+1,j)$
	si $(j \leq c - w_0)$ envoie $d + p_0$ à $(i, j+w_0)$.
Si $i > 0$ et $j < w_i$	reçoit d de $(i-1,j)$
	si $(i < m-1)$ envoie d à $(i+1,j)$
	si $(j \leq c - w_i)$ envoie $d + p_i$ à $(i, j+w_i)$.
Si $i > 0$ et $j \geq w_i$	reçoit d_1 de $(i-1,j)$ et d_2 de $(i, j-w_i)$
	calcul $d = \max\{d_1, d_2\}$
	si $(i < m-1)$ envoie d à $(i+1,j)$
	si $(j \leq c - w_i)$ envoie $d + p_i$ à $(i, j+w_i)$.

La grille irrégulière fournit la valeur du profit maximal, mais elle n'apporte pas la description complète du sac à dos. Dans le but d'avoir la composition du sac à dos nous proposons trois solutions. Deux d'entre elles envoient un vecteur de données. L'autre solution utilise la technique de parcours à l'envers. Une description détaillée est présentée dans la section B.1 de l'annexe B.

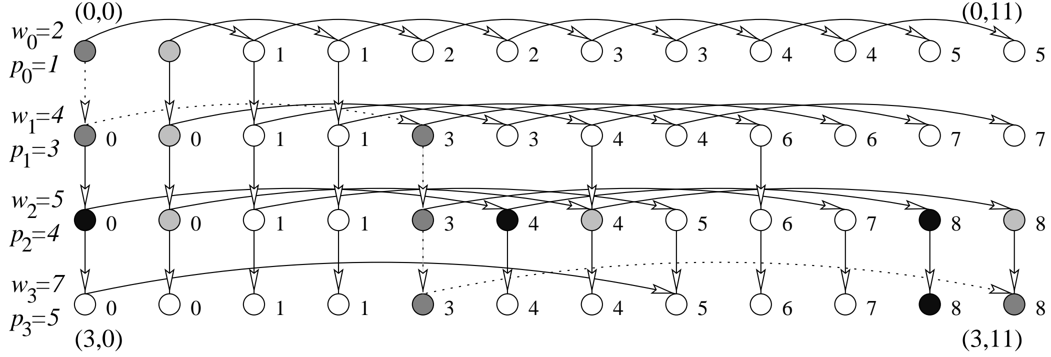

FIG. 3.2 – Graphe de précédence du problème du sac à dos avec objets de poids 2,4,5,7 et valeurs respectifs 1,3,4,5. La taille du sac est $c = 11$.

Dans l'exemple de la figure 3.2, nous voyons les valeurs de la variable d de chaque tâche. Nous avons enlevé de la figure les arcs qui n'ont pas été considérés lors du calcul du maximum. Le profit maximal est 8, trois des chemins qui mènent à ce profit ont eu leurs sommets colorés. Les compositions possibles avec profit 8 sont $(x_i)_0^3$: (0,1,0,1) et (0,0,2,0).

Le graphe de précédence du problème 0/1-*Knap* est aussi obtenu à partir d'une fonction $f_{01}(k, g)$:

$$f_{01}(k,g) = \max\{f_{01}(k-1,g), p_k + f_{01}(k-1, g-w_k)\} \quad \begin{cases} f(-1,g) = 0 \\ f(k,0) = 0 \\ f(k,y) = -\infty, y < 0. \end{cases}$$

La solution proposée par Lin et Storer [74] correspond à l'ordonnancement du graphe de précédence de cette fonction.

Pour résoudre le problème 0/1-*Knap* avec l'aide d'un grille irrégulière, il suffit d'ajouter une condition aux tâches. Une description rapide est : si lors du choix du maximum par la tâche (i, j), la valeur choisie est venue de la gauche (de la tâche $(i, j - w_i)$) alors envoi ∞ à la droite (à la tâche $(i, j + w_i)$).

3.4 Ordonnancement de la grille irrégulière

Dans cette section nous allons étudier l'ordonnancement de la grille irrégulière sur un hypercube. Le modèle utilisé est le SIMD, où nous sommes intéressés par le nombre d'étapes de l'algorithme. Dans chaque étape les processeurs peuvent calculer une opération mathématique élémentaire. À la fin de chaque étape, les processeurs voisins peuvent communiquer. Le modèle adopté est similaire au modèle utilisé dans les études précédentes [72, 74]. Mais en plus, nous considérons la topologie de la machine.

La borne inférieure de résolution d'une grille irrégulière est son chemin critique :

Proposition 3.1 *La longueur d'un chemin critique de la grille irrégulière* $NM(m,c)$ *avec des sauts* $(w_i)_0^{m-1}$ *est égale à* $m + \left\lfloor \frac{c}{w_{\min}} \right\rfloor - 1$.

3.4.1 Un premier ordonnancement

Étant donnée une grille irrégulière $NM(m,c)$ avec des sauts $(w_i)_0^{m-1}$, nous proposons un ordonnancement dans un hypercube avec au moins w_{max} noeuds.

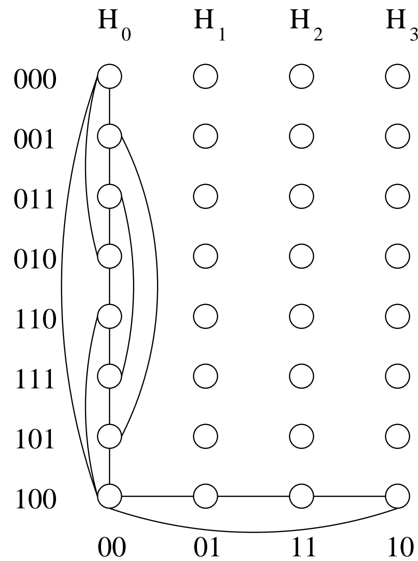

FIG. 3.3 – Exemple de décomposition de $H(5)$ en $H(3)\Box H(2)$. Les arêtes redondantes ont été omises pour simplifier la figure.

D'abord nous rappelons la définition de produit cartésien :

Définition 3.2 *Le produit cartésian de deux graphes* $G_1(V_1, E_1)$ *et* $G_2(V_2, E_2)$ *est le graphe* $G(V, E)$*, où* $V = \{v_1v_2, v_1 \in V_1$ *et* $v_2 \in V_2\}$*. Il existe des arêtes du sommet* $v_1v_2 \in V$ *aux sommets* $\{u_1v_2 | v_1$ *est adjacent à* u_1 *dans* $G_1\}$ *et* $\{v_1u_2 | v_2$ *est adjacent à* u_2 *dans* $G_2\}$*.*

Étant donnée un hypercube $H(n)$ avec $p = 2^n$ noeuds, nous considérons le produit cartésien $H(n) = H(\lceil \log_2 w_{\max} \rceil) \Box H(n')$, où $n' = n - \lceil \log_2 w_{\max} \rceil$. Pour la description de l'ordonnancement, nous dénommons les $2^{n'}$ hypercubes $H(\lceil \log_2 w_{\max} \rceil)$ par $H_0, \ldots, H_{2^{n'}-1}$, de façon à ce que deux indices consécutifs correspondent à des hypercubes adjacents. Un exemple est donné dans la figure 3.3.

Dans le but de fournir une numérotation uniforme aux hypercubes, nous allons associer un chemin *Hamiltonien* aux hypercubes $H(\lceil \log_2 w_{\max} \rceil)$ (par exemple le code de Gray réfléchi [73]). Dorénavant nous utiliserons le chemin *Hamiltonien* P_k associé à l'hypercube H_k. Les chemins P_k seront utilisés lors de l'ordonnancement. Nous dénommons les sommets de P_k par $P_k^j, 0 \leq j \leq 2^{\lceil \log_2 w_{\max} \rceil}$. Un exemple est présenté dans la figure 3.4.

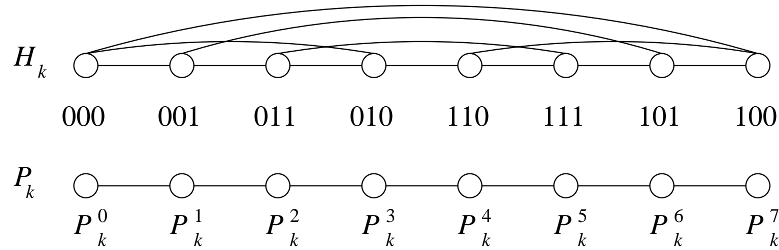

FIG. 3.4 – Représentation des sommets d'un hypercube $H(3)$ par un chemin *Hamiltonien.*

L'idée de l'ordonnancement est d'exécuter chaque ligne L_i de $NM(m, c)$ dans un chemin P_k. Chaque ligne L_i de la grille irrégulière est composée de w_i chaînes de taille au plus $\left\lfloor \frac{c}{w_i} \right\rfloor$. Donc, il existe plus de processeurs dans un chemin P_k qu'il n'existe de chaînes dans la ligne L_i.

Le nombre de chemins P_k dépend du nombre de processeurs de l'hypercube $H(n')$. Il existe

$$2^{n'} = \frac{p}{2^{\lceil \log_2 w_{\max} \rceil}}$$

chemins P_k. Pour simplifier la description de l'ordonnancement nous supposons que $\log_2 w_{\max}$ est entier, c'est-à-dire $2^{n'} = p/w_{\max}$. Cette restriction ne modifie pas les calculs de complexité.

L'ordonnancement

Dans l'ordonnancement, les tâches de la j-ème chaîne de la ligne L_i sont allouées au j-ème processeur du chemin $P_{i \bmod 2^{n'}}$. C'est-à-dire, les tâches $\{(x,y)|x = i, y \bmod w_i = j\}$ sont alloués au processeur $P^j_{i \bmod 2^{n'}}$.

La ligne $L_i, i \geq p/w_{\max}$, est allouée au même chemin $P_{i \bmod (p/w_{\max})}$ que la ligne $P_{i-(p/w_{\max})}$. Pour cette raison, la ligne $L_i, i \geq p/w_{\max}$ ne peut pas commencer son exécution avant l'achèvement de la ligne $L_{i-(p/w_{\max})}$. Dans le but de simplifier l'analyse, nous allons borner supérieurement ce temps d'attente pour les lignes $L_i, i \bmod (p/w_{\max})$ par t_w.

La tâche j de la ligne $L_i, i < p/w_{\max}$ est exécutée à l'instant $i(\log_2 w_{\max} + 1) + \left\lfloor \frac{j}{\min\{w_j|j \leq i\}} \right\rfloor$. Pour les autres lignes il faut considérer le temps d'attente. La tâche (i,j) est exécutée à l'instant :

$$i(\log_2 w_{\max} + 1) + \left\lfloor \frac{j}{\min\{w_j|j \leq i\}} \right\rfloor + \left\lfloor \frac{i}{p/w_{\max}} \right\rfloor t_w.$$

L'exactitude de l'algorithme découle des propriétés suivantes :

- Il n'y a pas de relation de précédence parmi les tâches de la même ligne allouées à différents processeurs du même chemin *Hamiltonien* P_k ;
- Chaque tâche allouée au chemin P_k possède au plus deux successeurs directs. Un d'entre eux est alloué au même processeur que lui, l'autre est alloué à un processeur du chemin $P_{(k+1) \bmod (p/w_{\max})}$;
- La distance maximale entre tâches avec relations de précédence allouées aux chemins P_k et $P_{(k+1) \bmod (p/w_{\max})}$ est $\log_2 w_{\max} + 1$. Un algorithme de routage qui garantie l'acheminement des résultats de précédence est explicité dans la section B.2 de l'annexe B.
- Dès qu'une chaîne commence son exécution, ses tâches sont exécutées aux étapes suivantes. Si une ligne commence son exécution avec w' tâches, à chaque cycle suivant au plus w' tâches de cette ligne sont exécutées.

Un exemple

Nous illustrons dans cette section l'exécution d'une grille irrégulière avec deux objets, $w_0 = 3$ et $w_1 = w_{\max} = 8$ (voir figure 3.5).

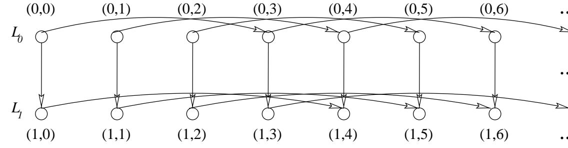

FIG. 3.5 – Grille irrégulière avec deux objets.

- au pas 0, les tâches (0,0), (0,1) et (0,2) sont exécutées sur P_0.
- au pas 1, les résultats des tâches (0,0), (0,1) et (0,2) sont acheminés et les tâches (0,3), (0,4) et (0,5) sont exécutées sur P_0.
- au pas 2, les résultats des tâches (0,0), (0,1), (0,2), (0,3), (0,4) et (0,5) sont acheminés et les tâches (0,6), (0,7) et (0,8) sont exécutées sur P_0.
- au pas 3, les résultats de tâches (0,3), (0,4), (0,5), (0,6), (0,7) et (0,8) sont acheminés. Les tâches (0,9), (0,10) et (0,11) sont exécutées sur P_0 et les tâches (1,0), (1,1), (1,2) sont exécutées sur P_1. À ce moment le *pipeline* est plein.
- et ainsi de suite jusqu'à l'exécution de toutes les tâches.

La figure 3.6 illustre le schéma d'exécution. Dans le diagrame de Gantt, il existe trois états possibles pour un processeur : inactif (carré hachuré), en cours d'exécution (d'une tâche), et acheminement (carré gris).

Analyse de la complexité

Nous analysons le temps total de l'ordonnancement. Nous divisons l'analyse en deux cas selon le temps d'attente t_w. Dans le premier cas, le temps d'attente est nul :

- Lorsque

$$p \geq \left\lfloor \frac{c}{w_{\min}} \right\rfloor \left\lceil \frac{w_{\max}}{\log_2 w_{\max} + 1} \right\rceil \tag{3.2}$$

à l'instant $i(\log_2 w_{\max} + 1)$, $\min\{w_j | j \leq i\}$ tâches de la ligne $L_i, i < p/w_{\max}$ commencent leur exécution. À l'instant $p/w_{\max}(\log_2 w_{\max} + 1)$ les tâches de la première ligne seront achevées ($p \geq \left\lfloor \frac{c}{w_0} \right\rfloor \left\lceil \frac{w_{\max}}{\log_2 w_{\max}+1} \right\rceil$). Donc, la ligne $L_{p/w_{\max}}$ peut débuter son exécution sans attente.

P_0	P_0^0	(0,0)	(0,3)	(0,6)	(0,9)	...
	P_0^1	(0,1)	(0,4)	(0,7)	(0,10)	...
	P_0^2	(0,2)	(0,5)	(0,8)	(0,11)	...
	P_0^3					
P_1	P_1^0				(1,0)	
	P_1^1				(1,1)	(1,5)
	P_1^2				(1,2)	(1,4)
	P_1^3					(1,3)

...

FIG. 3.6 – Grille irrégulière avec deux objets.

L'équation 3.2 assure qu'il n'y a pas d'attente pour le lignes L_i avec $i \bmod p/w_{\max} = 0$. La dernière ligne commence son exécution à l'instant $(m-1)(\log_2 w_{\max}+1)$. Son exécution s'achève à l'instant :

$$(m-1)(\log_2 w_{\max}+1) + \left\lfloor \frac{c}{w_{\min}} \right\rfloor. \tag{3.3}$$

– Quand

$$p < \left\lfloor \frac{c}{w_{\min}} \right\rfloor \left\lceil \frac{w_{\max}}{\log_2 w_{\max}+1} \right\rceil, \tag{3.4}$$

l'exécution des premières lignes ($L_i, i < p/w_{\max}$) se passe comme auparavant. Pour commencer l'exécution de la ligne $L_{p/w_{\max}}$ il faut attendre l'achèvement de la ligne L_0. Ce délai existe aussi pour les lignes L_i telles que $i > p/w_{\max}$.

Le temps d'attente pour chaque allocation dans le chemin P_0 est borné supérieurement par :

$$t_w = \left\lfloor \frac{c}{w_{\min}} \right\rfloor - \frac{p}{w_{\max}}(\log_2 w_{\max}+1).$$

En considérant le temps d'attente t_w, il suffit de prendre en compte le retard pour les lignes L_i, $i \bmod (p/w_{\max}) = 0$. Avant chaque période d'attente, le temps d'exécution est $(\log_2 w_{\max}+1)p/w_{\max}$. Il existe $\left\lfloor \frac{m w_{\max}}{p} \right\rfloor$ périodes, donc le temps pour achever la dernière ligne est bornée par

$$\left\lfloor \frac{c}{w_{\min}} \right\rfloor + \left\lceil \frac{m w_{\max}}{p} \right\rceil \left(\frac{p}{w_{\max}} (\log_2 w_{\max} + 1) + t_w \right),$$

c'est-à-dire,

$$\left\lfloor \frac{c}{w_{\min}} \right\rfloor \left(\left\lceil \frac{m w_{\max}}{p} \right\rceil + 1 \right). \tag{3.5}$$

Nous dénommons ces deux ordonnancements par algorithme $w_{\max}$ avec et sans temps d'attente. Les résultats sont résumés dans les propositions suivantes. Lorsque il n'y a pas de temps d'attente (équation 3.2) le temps de l'ordonnancement est donné par :

Proposition 3.2 *Le makespan de l'algorithme $w_{\max}$, sans temps d'attente, d'une grille irrégulière $NM(m, c)$ sur un hypercube avec au moins (équation 3.3) $\left\lfloor \frac{c}{w_{\min}} \right\rfloor \left\lceil \frac{w_{\max}}{\log_2 w_{\max} + 1} \right\rceil$ processeurs est $O\left(m \log w_{\max} + \frac{c}{w_{\min}}\right)$.*

Quand le nombre de processeurs disponible est plus petit (équation 3.4), il faut prendre en compte le temps d'attente (équation 3.5).

Proposition 3.3 *Le temps d'exécution d'une grille irrégulière par l'algorithme $w_{\max}$ avec temps d'attente sur l'hypercube en utilisant $p \geq w_{\max}$ processeurs est $O\left(\frac{mc}{p} \frac{w_{\max}}{w_{\min}}\right)$.*

Corollaire 3.1 *L'efficacité de l'algorithme $w_{\max}$ avec temps d'attente est $O\left(\frac{w_{\min}}{w_{\max}}\right)$.*

3.4.2 Un algorithme amélioré

Dans le premier algorithme il existe des processeurs qui sont inactifs, ce phénomène se produit lors de l'exécution de lignes avec moins de $w_{\max}$ processeurs.

L'idée pour améliorer la performance est d'utiliser des hypercubes de taille $H(\lceil \log_2 w_{\min} \rceil)$. Nous considérons le produit Cartésien de l'hypercube $H(n) = H(\lceil \log_2 w_{\min} \rceil) \,\square H(n')$ et nous utilisons les mêmes notions de chemin *Hamiltonien* P_k associé aux hypercubes du type $H(\lceil \log_2 w_{\min} \rceil)$. Nous supposons dorénavant que $\log_2 w_{\min}$ est entier.

L'allocation des tâches aux processeurs est telle que les tâches de la chaîne j de la ligne L_i sont allouées au processeur $P_{i \bmod (p/w_{\min})}^{(j \bmod w_{\min})}$.

Un problème qui apparaît dans cette nouvelle allocation est le flot de données entre des lignes consécutives. Si nous supposons que lors de l'exécution de la ligne L_i toutes les données nécessaires sont disponibles, nous aurons une exécution identique à celle de la table 3.1, où $x_i = \left\lfloor \frac{w_i}{w_{\min}} \right\rfloor$ et $y_i = w_i \bmod w_{\min}$.

<table>
<tr><td>temps</td><td>t_i</td><td>...</td><td>$t_i + x_i - 1$</td><td>$t_i + x_i$</td><td>$t_i + x_i + 1$</td><td>...</td><td>...</td></tr>
<tr><td>n. de tâches</td><td>$w_{\min}$</td><td>...</td><td>$w_{\min}$</td><td>y_i</td><td>$w_{\min}$</td><td>...</td><td>...</td></tr>
<tr><td></td><td colspan="4">w_i</td><td colspan="2">w_i</td><td>...</td></tr>
</table>

TAB. 3.1 – Schéma d'exécution.

Dans ce cas, le temps total de calcul pour achever l'exécution de la ligne L_i est $\left\lfloor \frac{c}{w_i} \right\rfloor \left\lceil \frac{w_i}{w_{\min}} \right\rceil$. Cependant, à un instant de l'exécution d'une ligne L_i sur un chemin P_k, il peut arriver que P_k reçoive w_a tâches alors qu'il attendait w_b tâches, avec $w_a < w_b$. La minimisation de ce type de situation va améliorer la performance de l'algorithme. Dans ce but, nous introduisons la définition suivante :

Définition 3.3 *Étant donnés deux poids w_i : $w_{\min}$ et $y_i = w_i \bmod w_{\min}$ la moyenne du temps d'inactivité est définie par :*

$$I(w_i) = \begin{cases} 0 & si\ y_i = 0, \\ \frac{w_{\min} - y_i}{\left\lceil \frac{w_i}{w_{\min}} \right\rceil} & si\ y_i > 0. \end{cases}$$

Cette fonction donne une estimation de l'occupation d'un chemin P_k par une ligne L_i. La valeur minimale de la fonction moyenne du temps d'inactivité ($I(w_i) = 0$) est obtenue avec des w_i tels que $w_i \bmod w_{\min} = 0$. Si les poids des objets du sac à dos sont triés en ordre croissant par rapport à la fonction $I()$ nous pouvons énoncer la proposition suivante :

Proposition 3.4 *Étant données deux lignes adjacentes L_i, L_{i+1} d'une grille irrégulière, il existe au plus une étape inactive dans l'exécution de la ligne L_{i+1}, due aux relations de précédence.*

La preuve se trouve dans l'annexe B, section B.3.

L'ordonnancement

Les lignes $L_i, i \geq p/w_{\max}$ sont allouées au même chemin $P_{i \bmod (p/w_{\max})}$ que la ligne $L_{i-(p/w_{\max})}$. Donc, la ligne $L_i, i \geq p/w_{\max}$ ne peut pas commencer son

exécution avant que l'exécution de la ligne $L_{i-(p/w_{\max})}$ s'achève. Nous bornons ce temps d'attente à nouveau par t_w.

Pour considérer les pas potentiellement inactifs (voir proposition 3.4) nous ajoutons 1 au temps de début d'exécution de chaque ligne. La tâche (i,j) allouée au processeur $P_{i \bmod (p/w_{\min})}^{(j \bmod w_i) \bmod w_{\min}}$ est exécutée à l'instant :

$$i(\log_2 w_{\min} + 2) + \left\lfloor \frac{j}{w_i} \right\rfloor \left\lceil \frac{w_i}{w_{\min}} \right\rceil + \left\lfloor \frac{i \bmod w_i}{w_{\min}} \right\rfloor + \left\lfloor \frac{i}{p/w_{\min}} \right\rfloor t_w. \tag{3.6}$$

Exemple

Nous présentons le schéma d'exécution d'une grille irrégulière sur 12 processeurs. Les objets sont de poids $w_0 = 4$, $w_1 = 9$ et $w_2 = 6$. Les états possibles d'un processeur sont dénotés de la même façon que pour la figure 3.6. Dans le schéma de la figure 3.7 il existe un pas inactif sur le chemin P_2 dû aux relations de précédence. Ce pas arrive juste après l'exécution des tâches (2,4) et (2,5), le chemin P_2 attend 4 résultats et il en reçoit seulement 3.

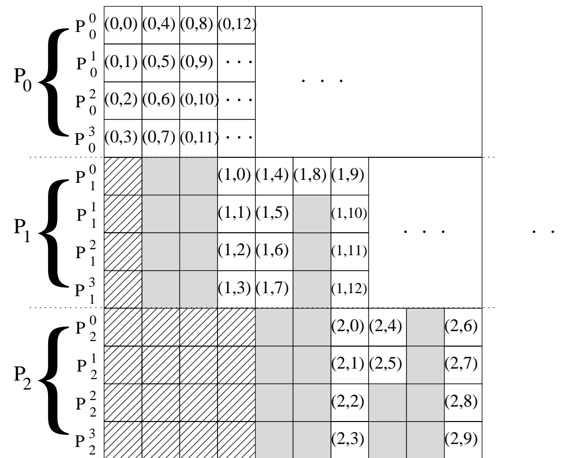

FIG. 3.7 – Schéma d'exécution d'une grille irrégulière avec des objets de tailles 4,9 et 6. La capacité du sac à dos est $c \geq 12$.

Analyse de la complexité

Dans le but de simplifier les équations nous dénotons

$$M = \max_i \left\lfloor \frac{c}{w_i} \right\rfloor \left\lceil \frac{w_i}{w_{\min}} \right\rceil , \tag{3.7}$$

De la même façon que pour l'analyse du premier algorithme, nous allons diviser l'analyse selon que le temps d'attente (t_w) est nul ou non.

– Si

$$p \geq M \left\lceil \frac{w_{\min}}{\log_2 w_{\min} + 1} \right\rceil \tag{3.8}$$

la ligne L_i commence son exécution à l'instant $i(\log_2 w_{\min} + 2)$, donc le *makespan* est :

$$(m-1)(\log_2 w_{\min} + 2) + \left\lfloor \frac{c}{w_{m-1}} \right\rfloor \left\lceil \frac{w_{m-1}}{w_{\min}} \right\rceil . \tag{3.9}$$

– Si

$$p < M \left\lceil \frac{w_{\min}}{\log_2 w_{\min} + 1} \right\rceil \tag{3.10}$$

il faut considérer le temps d'attente pour chaque ligne L_i, telle que $i \bmod (p/w_{\min}) = 0, i > 0$. Donc le temps total est le temps de l'équation 3.9 plus $\left\lceil \frac{m w_{\min}}{p} \right\rceil$ multiplié par le temps d'inactivité $t_w = (M - (\log_2 w_{\min} + 1)(p/w_{\min}))$. Ce temps est borné par :

$$M \left(\left\lceil \frac{m w_{\min}}{p} \right\rceil + 1 \right) . \tag{3.11}$$

Nous dénommons ces deux ordonnancements par algorithmes $w_{\min}$ avec et sans temps d'attente. Si nous observons que M (équation 3.7) est $O(\frac{c}{w_{\min}})$, les deux propositions suivantes sont immédiates. Quand il n'y a pas de temps d'attente (voir équation 3.8) :

Proposition 3.5 *Le makespan de l'ordonnancement $w_{\min}$ sans temps d'attente de la grille irrégulière dans un hypercube avec au moins $\frac{c}{\log w_{\min}}$ processeurs est donné par l'équation 3.9, c'est-à-dire en $O(m \log w_{\min} + \frac{c}{w_{\min}})$.*

S'il existe des temps d'attente le *makespan* est donné par l'équation 3.11.

Proposition 3.6 *Le makespan de l'ordonnancement $w_{\min}$ avec temps d'attente de la grille irrégulière dans un hypercube avec au moins $w_{\min}$ processeurs est $O\left(\frac{mc}{p}\right)$.*

En sachant que le temps d'exécution séquentiel de la grille irrégulière est mc, nous avons alors :

Corollaire 3.2 *L'efficacité de l'algorithme amélioré est asymptotiquement 1.*

L'extension de résultats présentés pour le graphe de précédence du problème 0/1-*Knap* est très simple. Il suffit de constater qu'il n'y a pas de précédence entre les tâches situées sur la même ligne.

3.5 Conclusion

Nous avons présenté un algorithme efficace pour la résolution du problème du sac à dos dans l'hypercube. Les principaux avantages de notre algorithme par rapport aux précédents sont la possibilité d'avoir plusieurs objets de chaque type et le fait qu'il a été conçu pour des machines existantes.

L'algorithme que nous avons proposé résout le problème *Knap* avec $\frac{c}{\log w_{\min}}$ processeurs en $O(m \log w_{\min} + \frac{c}{w_{\min}})$ étapes. Avec un nombre plus petit de processeurs, l'algorithme résout le problème en $O(\frac{mc}{p})$ cycles.

La solution efficace antérieure a été conçue pour le modèle PRAM, puis adaptée, sans prendre en compte les distances inter processeurs, à une machine SIMD dont la topologie était l'hypercube. Donc la dégradation des performances était de l'ordre du logarithme du nombre de processeurs. Nous évitons ce problème en développant un algorithme spécialisé pour l'hypercube, lequel peut être adapté aux autres topologies avec des techniques de plongement. Par exemple, le dernier algorithme présenté peut s'adapter à une grille torique, avec une dégradation de performance de $O(\log w_{\min})$, selon les techniques de [73], c'est-à-dire l'algorithme adaptée a complexité $O(\frac{mc}{p} \log w_{\min})$.

Il est aussi intéressant de remarquer que, les algorithmes présentés dans la littérature pour un nombre variable de processeurs [72, 74] ont été conçus pour résoudre le problème 0/1-*Knap*, où le nombre d'objets de chaque type est limité à un.

Chapitre 4

Ordonnancement sous BSP

Résumé

L'objet de ce chapitre est l'étude d'un ordonnancement particulier dans le cadre du modèle BSP. L'ordonnancement de chaînes indépendantes, facile sous le modèle avec délais de communication, s'avère un problème $\mathcal{NP}-$difficile sous BSP. Nous proposons plusieurs heuristiques qui considèrent le coût des communications. Nous avons complété l'analyse théorique du comportement au pire des cas par un grand nombre de simulations pour cerner le comportement en moyenne.

4.1 Ordonnancement de chaînes indépendantes

Le problème considéré dans ce chapitre est l'ordonnancement de k chaînes indépendantes $\{ch_1, \ldots, ch_k\}$ sur m processeurs identiques. Les chaînes sont des séquences de tâches de durée unitaire qui représentent par exemple l'exécution de procédures de bibliothèques indépendantes. La longueur, c'est-à-dire le nombre de tâches, de la chaîne ch_i est dénotée par n_i $(1 \leq i \leq k)$. Nous considérons dorénavant $k \geq 2$. Le nombre total de tâches est $n = \sum_{i=1}^{k} n_i$. Sans perte de généralité, nous supposons que les chaînes sont triées en ordre décroissant de longueurs, c'est-à-dire $n_1 \geq \ldots \geq n_k$.

Nous présentons tout d'abord le problème et une solution sous un modèle simple. Puis, nous proposons une solution sous BSP en étudiant successivement le cas sans communication et celui qui minimise l'équilibrage de la charge. Finalement, nous présentons des solutions intermédiaires.

Étant donnés les chaînes et le nombre de processeurs nous dénommons le problème de recherche d'un ordonnancement valide par SIC (de l'anglais *Scheduling Independent Chains*). La duplication est sans intérêt pour le problème SIC car chaque tâche possède au plus un successeur direct.

Définition 4.1 *Le makespan idéal d'un ordonnancement du problème SIC est défini par* $t^* = \max\left\{\left\lceil \frac{n}{m} \right\rceil, n_1\right\}$ *(le maximum entre le travail total également distribué et la longueur de la plus grande chaîne).*

Tous les ordonnancements valides ont un *makespan* égal à au moins t^*. Il est à noter que cette borne n'est pas toujours atteinte. Il existe des instances pour lesquelles afin d'atteindre la borne t^* il faut couper des chaînes (la figure 4.1 illustre un exemple). Cependant, les coupures introduisent des communications entre les différents morceaux de la même chaîne. Cette remarque induit la notation suivante :

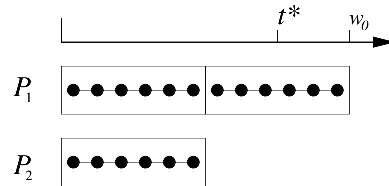

Fig. 4.1 – Meilleur ordonnancement sans communication de trois chaînes identiques sur deux processeurs.

Notation 4.1 *Le plus petit makespan d'un ordonnancement sans communication est dénoté* w_0.

Pour ne pas avoir besoin de dessiner les tâches dans tous les exemples, lorsqu'une chaîne n'est pas coupée, elle ne sera représentée que par son rectangle dans le diagramme de Gantt.

Dans ce chapitre nous allons voir d'abord que SIC est un problème facile sous le modèle délai. Nous examinons ensuite le problème sous le modèle BSP. Dans un premier temps nous allons voir deux cas particuliers, deux et un nombre illimité de processeurs. Nous constatons que le problème est $\mathcal{NP}-$complet même pour deux processeurs. Dans un deuxième temps nous analyserons le problème sur un nombre limité de processeurs. Nous étudierons sa complexité et nous proposerons des algorithmes d'approximation. Puis

nous étudierons l'imposition d'une taille minimal de super-étape. Pour achever le chapitre nous présenterons des simulations qui confirment le bon comportement des algorithmes proposés.

4.1.1 Ordonnancement sous le modèle délai

L'algorithme d'ordonnancement dans le modèle SCT est simple (voir algorithme 4.1). Nous dénommons cet algorithme "asynchrone" car les communications sont indépendantes les unes des autres. Le principe consiste à remplir les processeurs, l'un après l'autre, de l'instant 0 à l'instant t^*, c'est-à-dire de la gauche vers la droite dans les figures.

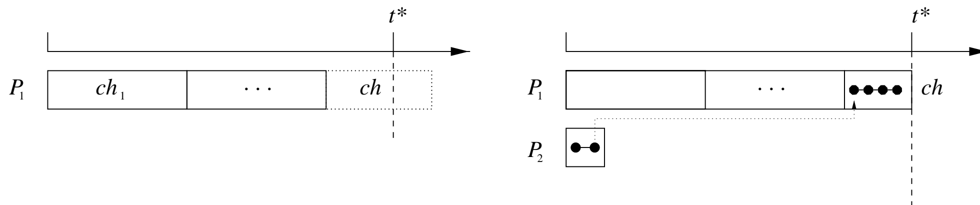

FIG. 4.2 – Allocation gloutonne.

Lorsqu'il n'y a plus de place pour allouer une chaîne entière sur un processeurs P_j, la chaîne est coupée et une communication est introduite entre ses deux morceaux. Pour respecter les contraintes de précédence, la première partie de la chaîne est exécutée sur P_{j+1} à gauche et la dernière partie sur P_j à droite (la figure 4.2 illustre le principe). L'algorithme est détaillé ci-dessous.

Le point principal qui fait que l'algorithme 4.1 est valide est que la communication peut toujours se faire entre les parties d'une chaîne coupée (les chaînes qui sont coupées ont moins de t^* tâches).

4.1.2 Préliminaires sur l'ordonnancement sous BSP

Dans tous les algorithmes qui seront présentés dans ce chapitre, chaque processeur envoie et reçoit au plus une communication. Donc, le coût des phases de communication est au plus $2hg$, où h est le nombre de mots de la plus grande communication (voir section 2.4.5). Nous supposons que les coûts de communication et de synchronisation (l) de chaque super-étape sont représentés par un coût de communication/synchronisation (noté par CS). Le coût de chaque CS est borné par C.

Algorithm 4.1 Algorithme asynchrone.

`Entrées:` $load(P_j)$ donne le nombre de tâches allouées au processeur P_j

$i \leftarrow 1$, $j \leftarrow 1$;
tant que $i \leq k$ **faire** {ils restent des chaînes à allouer}
  tant que $load(P_j) + n_i \leq t^*$ et $i \leq k$ **faire**
    allouer ch_i sur P_j; $i \leftarrow i + 1$;
  fin tant que
  si $load(P_j) \neq t^*$ **et** $i \leq k$ **alors**
    allouer les premières $n_i - (t^* - load(P_j))$ tâches de ch_i sur P_{j+1};
    allouer les dernières $t^* - load(P_j)$ tâches de ch_i sur P_j
    $i \leftarrow i + 1$;
  fin si
  $j \leftarrow j + 1$;
fin tant que

Nous nous intéressons aussi à l'espacement minimal entre deux super-étapes, dénoté par λ. Pour simplifier nos analyses la dernière super-étape d'un programme n'est pas prise en compte. Donc un programme avec s super-étapes a un temps de CS égal à $(s - 1)C$.

Nous pouvons dériver facilement d'un algorithme asynchrone (un algorithme dans le modèle délai) un algorithme BSP.

Propriété 4.1 *Pour transformer un algorithme asynchrone quelconque en un algorithme BSP il suffit de remplacer les communications (asynchrones) par des communications globales.*

4.2 Deux cas préliminaires dans BSP

Nous présentons d'abord deux cas extrêmes, l'ordonnancement sur deux processeurs et sur un nombre non borné de processeurs.

4.2.1 SIC sur deux processeurs dans BSP

Dans cette section nous montrons que la recherche de la solution optimal pour SIC sur deux processeurs dans BSP est $\mathcal{NP}$-complet et nous proposons un algorithme avec *makespan* borné par $t^* + C$, c'est-à-dire avec au plus deux super-étapes.

Lemme 4.1 *Le problème de déterminer si il existe une solution de SIC sur deux processeurs avec makespan t^* est $\mathcal{NP}$-complet.*

Preuve : par réduction au problème partition [39]. La preuve se trouve dans la section C.1 de l'annexe C.

L'heuristique que nous proposons est fondée sur l'algorithme asynchrone. Le premier processeur est remplit exactement jusqu'à l'instant t^*. Si une chaîne est coupée, une CS est introduite entre ses deux parties. D'où la proposition 4.1.

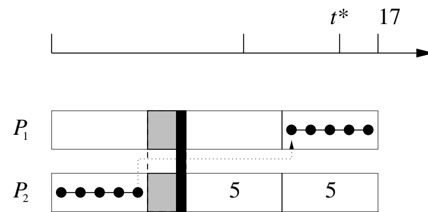

FIG. 4.3 – Algorithme BSP sur deux processeurs.

Proposition 4.1 *Le makespan de l'ordonnancement généré par l'algorithme 4.2 est au plus $t^* + C$.*

Pour terminer la présentation du cas sur deux processeurs, nous fournissons un exemple. Étant donnée une instance de SIC avec 4 chaînes de tailles respectives 10, 10, 5 et 5, et le temps d'une CS, $C = 2$. La figure 4.3 illustre l'allocation.

Le temps du *makespan* idéal est $t^* = 15$. D'abord le processeur P_1 est remplit jusqu'à l'instant t^*. Pour que la chaîne coupée puisse communiquer une CS est insérée et le *makespan* devient $17 = t^* + C$.

L'algorithme 4.2 ne génère pas toujours un ordonnancement optimal. Parfois un ordonnancement optimal existe (voir figure 4.4), cependant comme nous l'avons vue la détermination de l'existence d'un tel ordonnancement est un problème $\mathcal{NP}-$complet.

Algorithm 4.2 Algorithme BSP sur deux processeurs.

```
Entrées: load(P_1) donne le nombre de tâches allouées au processeur P_1
  i ← 1;
  tant que load(P_1) + n_i ≤ t* et i ≤ k faire
    allouer ch_i à P_1;
    i ← i + 1;
  fin tant que
  si load(P_1) ≠ t* et i ≤ k alors
    allouer les n_i − (t* − load(P_1)) premières tâches de ch_i à P_2;
    allouer les t* − load(P_1) dernières tâches de ch_i à P_1;
    Insérer une CS a l'instant n_i − (t* − load(P_1));
    i ← i + 1;
  fin si
  tant que i ≤ k faire
    allouer ch_i à P_2;
    i ← i + 1;
  fin tant que
```

4.2.2 SIC avec un nombre non borné de processeurs dans BSP

Quand le nombre de processeurs est au moins égal au nombre de chaînes k, la solution directe est d'allouer une chaîne par processeur. Le *makespan* est donné par la longueur de la plus grande chaîne n_1.

Une solution avec moins de processeurs peut exister, mais il n'est pas possible de diminuer le *makespan*. La recherche du nombre minimal de processeurs nécessaires est aussi un problème $\mathcal{NP}$-difficile. Elle est équivalente au problème de *bin-packing* unidimensionnel [39] avec des *bins* de longueur n_1.

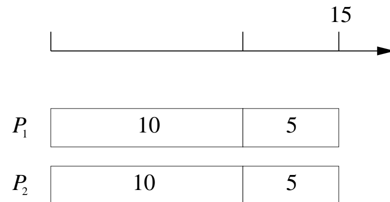

FIG. 4.4 – Exemple d'un ordonnancement qui atteint t^* sans communication.

4.3 Nombre fixé de processeurs

Dans cette section, nous montrons d'abord que le problème de minimiser les temps d'inactivité sans l'ajout de CS est un problème $\mathcal{NP}-$difficile. Ensuite nous étudions la minimisation du temps d'inactivité avec l'ajout de CS. Premièrement, nous fournissons un exemple où sont nécessaires $\lceil \frac{m+1}{2} \rceil$ super-étapes de façon à minimiser le temps d'inactivité (Id). Deuxièmement, un algorithme d'ordonnancement avec temps de calcul t^* et au plus $\lceil \frac{m}{2} \rceil + 1$ super-étapes est présenté.

Il est donc clair que la minimisation du temps d'inactivité peut occasionner un grand nombre de super-étapes. Nous recherchons alors un compromis entre Id et le nombre de CS. Nous terminons la section avec la présentation d'une famille d'algorithmes pour lesquels la garantie de performance dépend du nombre de CS.

4.3.1 Équilibrage total de la charge

Nous avons vu dans la section 2.5.1 que la minimisation du *makespan* est équivalente à la minimisation du surcoût $T_c + Id$. Une question naturelle est qu'arrive-t-il quand nous minimisons un de ces termes indépendamment de l'autre ? Nous commençons avec la minimisation du temps d'inactivité sans l'ajout de CS.

Analyse théorique

Nous nous intéressons à l'ordonnancement de chaînes avec *makespan* t^* sur $m \geq 3$ processeurs. Il est possible qu'il n'existe pas de solution de SIC sans communication qui atteigne la borne t^*. Mais, même si un tel ordonnancement existe, il est difficile de le construire.

Théorème 4.1 *La recherche d'un ordonnancement optimal pour le problème SIC sans l'ajout de communication avec un nombre arbitraire de processeurs est un problème $\mathcal{NP}-$difficile au sens fort (de l'anglais* strongly $\mathcal{NP}-$ hard*).*

Preuve : par réduction de la partition tridimensionnelle [39]. La preuve complète se trouve dans la section C.2 de l'annexe C.

Nombre maximum de super-étapes

Dans cette section, nous présentons une instance pour laquelle tout ordonnancement sous BSP nécessite $\left\lceil \frac{m-1}{2} \right\rceil$ CS pour minimiser le temps d'inactivité Id. La preuve est constructive, nous proposons des ordonnancements optimaux pour successivement 1, 2,... $\left\lceil \frac{m-1}{2} \right\rceil$ CS. L'instance de SIC est composée de $m+1$ chaînes de même taille S. Pour simplifier le développement nous supposons que $\frac{(m+1)S}{m}$ est entier.

Une super-étape

Avec une seule super-étape, c'est-à-dire sans l'introduction de CS, dans tous les ordonnancements valides, il existe un processeur avec au moins deux chaînes. Le meilleur *makespan* possible est $w_o = 2S$. Un des ordonnancements avec *makespan* $2S$ est obtenu en allouant deux chaînes à un processeur, et une chaîne à chacun des $m-1$ processeurs restants.

Deux super-étapes

Propriété 4.2 *Dans une super-étape plus longue que $\frac{S}{m}$, la meilleure stratégie est d'exécuter des tâches de deux chaînes différentes sur un processeur et une chaîne sur chaque autre processeur.*

Preuve : Si il existe une chaîne qui n'est pas exécutée au cours de cette super-étape, le plus petit *makespan* possible sera la taille de cette super-étape plus S, donc un *makespan* plus grand que t^*. Si toutes les chaînes sont exécutées, un meilleur *makespan* est possible. Puisqu'il existe $m+1$ chaînes et m processeurs, l'un d'entre eux aura deux chaînes.❑

Il est facile de remarquer que les chaînes qui partagent un processeurs pendant une super-étape doivent être seules dans l'autre super-étape. Par exemple dans la figure 4.5, les chaînes ch_1, ch_2 et ch_3, ch_4 sont, respectivement, ensemble dans les première et deuxième super-étapes.

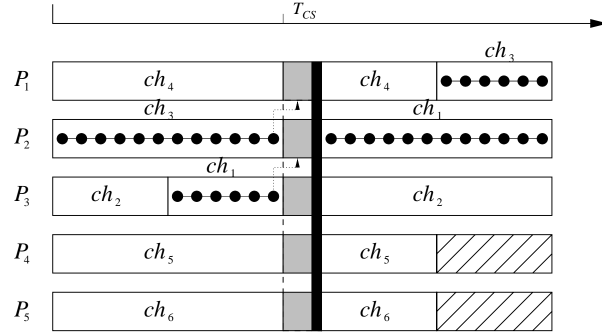

FIG. 4.5 – L'exemple de cas au pire avec deux super-étapes.

Propriété 4.3 *Dans une super-étape, lorsque deux chaînes sont exécutées sur le même processeur, le temps de calcul disponible est partagé équitablement entre les chaînes.*

Preuve : Étant données deux chaînes qui partagent un processeur au cours de la première super-étape, ch_1 et ch_2. Dénotons par x le nombre de tâches de ch_1 qui sont exécutées pendant cette super-étape, et par T_{CS} le temps de calcul de la première super-étape. L'objectif est de minimiser le *makespan*. $(S - x)$ tâches de ch_1 seront exécutées au cours de la deuxième étape. $S - (T_{CS} - x)$ tâches de ch_2 seront exécutées au cours de la deuxième étape. En d'autres termes, il faut minimiser $\max\{T_{CS} + (S - x), T_{CS} + (S - (T_{CS} - x))\}$. Ce qui est réalisé pour $x = \frac{T_{CS}}{2}$.

Dans un ordonnancement optimal les deux chaînes qui partagent un processeurs au cours de la deuxième étape ont un processeur exclusif au cours de la première étape. Alors T_{CS} tâches de chacune sont exécutées dans la première super-étape. Il reste donc $2(S - T_{CS})$ tâches, la moitié appartenant à chaque chaîne, à exécuter dans un processeur au cours de la deuxième super-étape. ❑

Pour déterminer le *makespan* optimal, il faut aussi choisir où insérer la CS. Dans ce cas, il faut choisir T_{CS} pour minimiser le *makespan*. Donc, nous voulons minimiser les temps d'achèvement des chaînes qui partagent un processeur. En autres termes, nous voulons minimiser $\max\{S + \frac{T_{CS}2}{2}, 2S - T_{CS}\}$, ce qui est acquis pour $T_{CS} = \frac{2S}{3}$, et donc conduit à un *makespan* de $\frac{4S}{3}$.

Le temps du meilleur ordonnancement avec deux super-étapes est alors $S + C + \frac{S}{3}$. Il est à noter que pour trois processeurs cette valeur correspond au travail total plus le temps d'une CS, C.

Plus de deux super-étapes

De la propriété 4.2, il existe i régions dans lesquelles deux chaînes peuvent partager un processeur dans un ordonnancement avec i super-étapes. D'une extension directe de la propriété 4.3, dans les algorithmes optimaux, les chaînes vont partager également les super-étapes.

Les instants pour le placement des CS sont calculés par induction comme une série T_{CS}^{j} $(1 \leq j \leq i)$. Le *makespan* qui minimise le temps d'inactivité a un temps de $S + iC + \frac{S}{2i+1}$ avec les super-étapes placées aux instants $2j\frac{S}{2i+1} + jC, 1 \leq j \leq i$.

Pour éliminer le temps d'inactivité, nous devons communiquer $z = \left\lceil \frac{m-1}{2} \right\rceil$ fois, et les CS sont distribuées à distance $\frac{2S}{m}$ les une des autres. Dans ce cas le *makespan* est $S + zC + \frac{S}{2z+1}$, c'est-à-dire $t^* + zC$. Ce résultat est valable pour m pair, pour m impair nous avons besoin de l'arrondir.

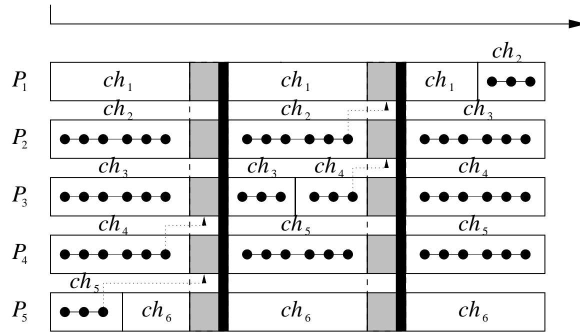

FIG. 4.6 – L'exemple de cas au pire avec trois super-étapes.

4.3.2 Un algorithme

Nous présentons un algorithme (détaillé dans l'algorithme 4.3) qui minimise le temps d'inactivité Id, cet algorithme peut avoir jusqu'à $\lceil \frac{m}{2} \rceil + 1$ super-étapes, c'est-à-dire $\lceil \frac{m}{2} \rceil$ CS car nous ne considérons pas la dernière super-étape. L'algorithme est fondé sur la propriété 4.1 appliquée à l'algorithme 4.1.

L'idée principale est de montrer qu'il est possible de grouper les communications asynchrones (voir figure 4.7), au moins deux par deux, dans la même CS.

Algorithm 4.3 Algorithme avec nombre de super-étapes limitée.

Appliquer l'algorithme asynchrone;

Mettre des étiquettes sur toutes les communications asynchrones (CA);

#cs $\leftarrow$ 0;

`tant que` il existe des CA étiquetées `faire`

  #cs $\leftarrow$ #cs+1;

  calculer synch[#cs] le plus petit temps de réception entre toutes les communications étiquetées;

  enlever les étiquettes de toutes les CA qui peuvent être groupées dans la même CS à l'instant synch[#cs];

`fin tant que`

`pour` $i \leftarrow 1$ `jusqu'à` #cs `faire`

  insérer une CS après un temps de calcul de synch[i] tâches;

`fin pour`

Une preuve que les ordonnancements produits par l'algorithme 4.3 ont au plus $\lceil \frac{m}{2} \rceil$ CS se trouve dans la section C.3 de l'annexe C.

4.3.3 Algorithme avec nombre fixé de super-étapes

L'algorithme présenté dans la section précédente peut avoir jusqu'à $\lceil \frac{m}{2} \rceil + 1$ super-étapes, alors qu'il fournit un équilibrage de charge le meilleur possible. Dans cette section, au lieu de minimiser seulement le temps d'inac-

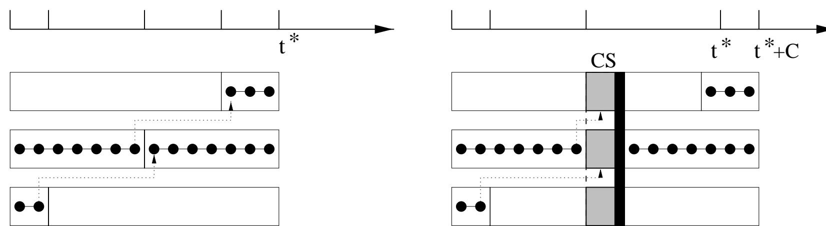

FIG. 4.7 – Exemple de regroupement de deux communications asynchrones dans la même CS.

tivité nous recherchons un compromis pour le surcoût $Id + T_c$. En effet, Id peut augmenter si T_c diminue.

Sans communication, c'est-à-dire $T_c = 0$, il existe plusieurs algorithmes d'approximation bien connus comme LPT [55] et *multi-fit* [19]. Nous présentons ici l'algorithme LPT qui a une garantie de performance de :$w_{LPT} \leq (\frac{4}{3} - \frac{3}{m})w_o$. Dans cet algorithme, les tâches (dans notre cas les chaînes) sont allouées en ordre décroissant de taille avec temps d'exécution au plus tôt. La figure 4.8 illustre un exemple.

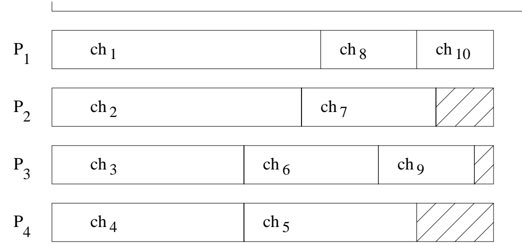

FIG. 4.8 – Principe d'allocation de l'algorithme LPT sur 4 processeurs.

Nous introduisons une notation pour le temps du meilleur algorithme avec un nombre donné de super-étapes.

Notation 4.2 *Le makespan optimal avec s super-étapes, c'est-à-dire le plus petit makespan possible avec au plus s super-étapes est noté w_o^s.*

Nous présentons un algorithme qui dépend du nombre de super-étapes s et d'une valeur α (entier compris entre n_1 et t^*). Nous analysons le comportement de cet algorithme lorsque le nombre de super-étapes s varie. Pour terminer, nous proposons des perfectionnements pour l'algorithme et nous choisissons la valeur de α qui garantie la meilleure performance.

Proposition 4.2 *Étant donnée une instance de SIC, un nombre de super-étapes s et un entier α tel que $n_1 \leq \alpha \leq t^*$, w_o^s est borné par :*

$$w_o^s \leq t^* + \frac{1}{2s-1}\alpha + (s-1)C.$$

Preuve : La preuve est constructive et l'algorithme 4.4 atteint la borne. La preuve complète se trouve dans la section C.4 de l'annexe C.

Algorithm 4.4 Algorithme avec un nombre donné des super-étapes (s).

```
Entrées: load(P_j) donne le nombre de tâches alloués au processeur P_j
  i ← 1; j ← 1; Δ_α = (2/(2s−1))α; max_load = t* + Δ_α/2;
  insérer jusqu'à s − 1 CS en intervalles réguliers de longueur Δ_α;
  tant que (i ≤ k) faire
    si (load(P_j) + n_i ≤ max_load) alors
      allouer ch_i à P_j;
    sinon
      allouer les max_load −(n_i + load(P_j)) premières tâches de ch_i à P_{j+1};
      allouer les tâches qui restent de ch_i à P_j;
      si il n'y a pas de CS entre les deux parties d'une chaîne ch_i coupée
      alors
        transférer les tâches de P_j à P_{j+1} de la gauche à la droite jusqu'à
        trouver une CS;
      fin si
      j ← j + 1;
    fin si
    i ← i + 1;
  fin tant que
```

Le surcoût de l'algorithme 4.4 pour s super-étapes et α donné ($n_1 \leq \alpha \leq t^*$) est $\frac{\Delta_\alpha}{2} + (s-1)C$, où $\Delta_\alpha = \frac{2}{2s-1}\alpha$.

Nous présentons dans la section C.4 de l'annexe C des propriétés sur le nombre nécessaire de CS, c'est-à-dire que avec l'ajout de super-étapes il n'y a pas de diminution du temps d'inactivité. Nous montrons aussi que le nombre optimal de super-étapes pour minimiser le surcoût de l'ordonnancement produit par l'algorithme 4.4 est donné par $s^*_{\max} = \left\lceil \sqrt{\frac{\alpha}{2C}} + \frac{1}{2} \right\rceil$ ou $s^*_{\max} = \left\lfloor \sqrt{\frac{\alpha}{2C}} + \frac{1}{2} \right\rfloor$.

Perfectionnements

La performance de l'algorithme 4.4 dépend de la distribution de longueur des chaînes. Pour l'instant ses garanties de performance sont une borne supérieure dans le cas au pire. D'abord nous proposons des perfectionnements dans le comportement moyen, et après nous analysons la bonne valeur de α.

Perfectionnement de l'algorithme 4.4

Dans le but d'avoir une meilleure performance moyenne, l'allocation de chaînes peut être un peu modifiée. Nous allons remplir les processeurs jusqu'à t^*. Si pendant l'allocation, une chaîne est coupée et s'il n'existe pas de CS entre ses deux moitiés, la moins coûteuse des procédures suivantes est choisi :

Pour les prochaines descriptions, pour une chaîne coupée ch on considère p_i le processeur où la dernière partie de ch est allouée et p_{i+1} le processeur où la première partie de ch est allouée. Nous dénommons β la super-étape où la communication a lieu.

(i) l'arrêt de l'exécution des tâches de la chaîne ch dans la super-étape sur le processeur p_{i+1}. Ces tâches sont transférées au processeur p_i. Dans ce cas le temps alloué auparavant pour ch sur p_{i+1} (remplit avec des lignes horizontales dans la figure 4.9) peut être utilisé pour l'allocation de la prochaine chaîne.

(ii) le retard des tâches de la chaîne ch à la super-étape β sur le processeur p_i jusqu'au début de la prochaine super-étape. C'est-à-dire l'exécution de la chaîne ch sur le processeur p_i commence après la super-étape β.

Il est clair que le délai introduit par cette procédure peut être aussi grand que la moitié de la partie de calcul d'une super-étape. Donc, cet algorithme a la même borne au pire que l'algorithme original. Toutefois, l'algorithme perfectionné a un comportement moyen meilleur.

Avec ce perfectionnement le nombre de super-étapes peut être réduit dans le cas moyen. Pour trouver le meilleur *makespan* avec les perfectionnements il faut faire une recherche du nombre de super-étapes entre 1 et $s^*_{\max}$.

Notation 4.3 *Étant donnée une instance de SIC, nous notons s^* le nombre de super-étapes qui fournit le plus petit makespan avec l'algorithme perfectionné.*

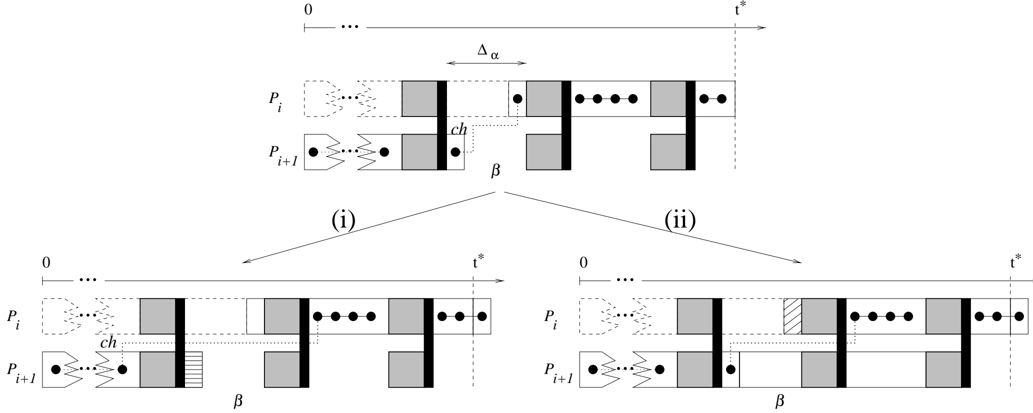

FIG. 4.9 – Les alternatives pour réarranger les chaînes sans CS entre ses moitiés.

Amélioration de la borne

L'idée est de prendre en compte les différences entre t^* et n_1, et entre n_1 et l'instant où la dernière CS a été insérée.

Proposition 4.3 *Avec le perfectionnement de l'algorithme la borne devient :*

$$t^* + \max\left(\frac{\Delta_\alpha}{2} - \frac{t^* - n_1}{2}, n_1 - (s-1)\Delta_\alpha, 0\right) + (s-1)C.$$

La preuve se trouve dans la section C.5 de l'annexe C. La valeur minimale de cette borne est atteinte pour $\alpha_{\min} = \frac{t^*+n_1}{2}$. Le *makespan* de l'algorithme perfectionné est alors :

$$t^* + \frac{1}{2s-1}\frac{t^* + n_1}{2} + (s-1)C.$$

4.4 Influence de la latence

Lorsque nous considérons un délai minimal entre deux CS consécutives λ, les ordonnancements produits par les algorithmes qui ont été présentés peuvent ne plus s'avérer valides. Pour considérer la latence, nous proposons l'algorithme suivant :

Dans l'algorithme 4.5, il existe deux opérations avec les super-étapes : l'avancement et le délai. Lorsqu'une de ces opérations est appliquée, une fusion de super-étapes peut aussi arriver. Si *ch* est une chaîne coupée et la CS

Algorithm 4.5 Algorithme où les CS sont espacés d'au moins λ

Appliquer un des algorithmes BSP donnés;

`pour` chaque CS (de la première à la dernière) `faire`

`si` CS se trouve à moins de $\frac{\lambda}{2}$ unités de temps de la CS précédente `alors`

transférer les tâches des chaînes coupées qui utilisent cette CS;

fusionner cette CS avec la précédente;

`fin si`

`si` CS est entre $\frac{\lambda}{2}$ et λ unités de temps de la précédente `alors`

Retarder la super-étape jusqu'à ce qu'elle soit à une distance λ de la super-étape précédente;

Fusionner avec cette super-étape avec toutes les super-étapes à moins de $\frac{\lambda}{2}$ unités de temps;

Retarder les tâches des chaînes coupées qui utilisaient ces super-étapes;

`fin si`

`fin pour`

utilisée est avancée, les tâches de la première partie de ch sont migrées vers la deuxième partie. Sinon, lorsque la CS est retardée, le début d'exécution de la deuxième partie de la chaîne est aussi retardée. La fusion des CS consiste simplement à remplacer plusieurs CS par une seule CS. La proposition suivante donne une borne supérieure à cette transformation.

Proposition 4.4 *Le coût supplémentaire pour considérer l'influence de la latence avec l'algorithme 4.5 est un terme additionnel de* $\frac{\lambda}{2}$.

La preuve de la proposition, ainsi que plus de détails sur l'influence de la latence se trouvent dans la section C.6 de l'annexe C.

4.5 Quelques expérimentations

Nous avons effectué des simulations pour comparer les algorithmes précédents. Les instances étaient composées de chaînes de longueurs choi-

sies de façon aléatoire selon une loi *binomiale* de moyenne 100 et d'écart type 10. Comme le comportement des algorithmes ne dépendait du nombre de processeurs, nous avons étudié le cas avec 50 processeurs. Le coût d'une CS a été fixé à 10% de la taille moyenne de chaînes.

Toutes les courbes obtenues par des simulations sont présentées et commentées dans [45]. Nous présentons les principaux résultats obtenus dans les paragraphe suivant.

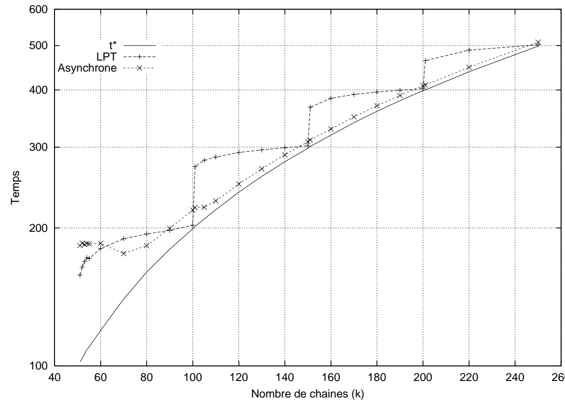

FIG. 4.10 – Performance moyenne des algorithmes LPT et Asynchrone sur 50 processeurs avec chaînes de taille quasi égal.

Dans les descriptions suivantes nous faisons usage pour les *makespans* construits les légendes t^*, LPT, Asynchrone - l'algorithme 4.3 et $C/s^*/\alpha$, où la dernière dénote les algorithmes de la section 4.4 avec nombre optimal de super-étapes et α est $n_1, (t^* + n_1)/2$ ou t^*. t^* correspond au *makespan* idéal.

Dans la courbe de la figure 4.10, nous voyons clairement les sauts de l'algorithme LPT et le bon comportement de l'algorithme Asynchrone quand le nombre de chaînes est grand. Lorsque le nombre de tâches par processeur est plus grand que $2n_1$ toutes les chaînes coupées peuvent utiliser la première et unique CS insérée par l'algorithme 4.3. Dans l'exemple ce nombre est un peu plus grand que le double du nombre de processeurs.

Dans la figure 4.11 nous pouvons vérifier que le nombre moyen de CS est loin de celui du cas au pire (25). Cela veux dire qu'en moyenne le nombre de communications asynchrones groupées par l'algorithme 4.3 est plus grand que deux.

La figure 4.12 illustre le surcoût engendré par les variations des algorithmes $C/s^*/\alpha$. Le *makespan* de l'algorithme $C/s^*/n_1$ ne s'approche pas de

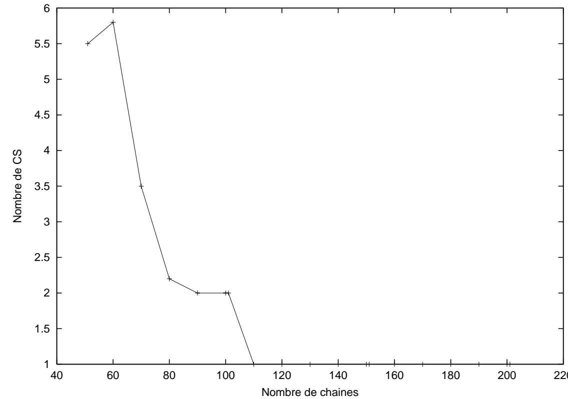

FIG. 4.11 – Nombre moyen de CS dans l'algorithme Asynchrone.

$t^* + C$ lorsque le nombre de chaînes augmente. Car l'effet d'avoir les CS très proches et au début est le réarrangement de beaucoup des chaînes coupées. D'autre part il est plus performant que l'algorithme $C/s^*/t^*$, où le délai introduit pour le réarrangementde chaque chaîne coupée peut être important $(\frac{t^*}{2s^*-1})$. Quand le nombre de chaînes est petit. L'algorithme $C/s^*/(t^*+n_1)/2$ combine les avantages des deux autres algorithmes. Il réarrange moins de chaînes que $C/s^*/n_1$ et le délai introduit pour chaque chaîne réarrangée est limitée par $\frac{t^*+n_1}{2(2s^*-1)}$.

Dans la figure 4.13 nous présentons les makespans construits pour des problèmes SIC avec 65 chaînes. Nous vérifions le comportement de l'algorithme $C/i/(t^* + n_1)/2$ lorsque le nombre de CS i varie. Le compromis entre le nombre de CS et le temps d'inactivité dans les processeurs est clair. Sans communication le makespan est 240, avec l'ajout de CS nous avons obtenu des makespans plus petits. À partir de deux CS, la diminution du temps d'inactivité par l'ajout d'une nouvelle CS est moins importante que son coût.

4.5.1 Remarques sur les expérimentations

Malgré le nombre élevé de super-étapes dans le cas au pire pour l'algorithme 4.3 le nombre de communications utilisées en pratique, même quand le nombre de chaînes est proche du nombre de processeurs, est beaucoup moins important. En dépit d'une garantie de performance meilleure pour l'algorithme C.1 avec $\alpha = n_1$, l'algorithme perfectionné avec $\alpha = \frac{t^*+n_1}{2}$ produit des ordonnancements avec *makespan* plus petit dans le cas moyen.

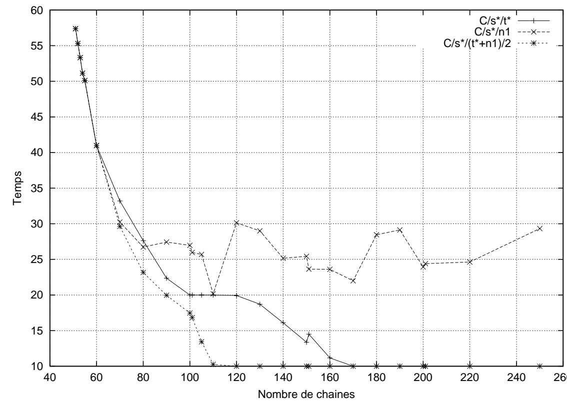

FIG. 4.12 – Performance moyenne du surcoût des algorithmes $C/s^*/t^*$, $C/s^*/n_1$, $C/s^*/(t^*+n_1)/2$ sur 50 processeurs avec chaines de quasi le même taille.

4.6 Conclusion

Nous avons étudié le problème du regroupement des communications entre paires de processeurs en phases de communication globale (qui impliquent tous les processeurs). Nous étudions dans chapitre 6 l'optimisation de phases de communication globale.

Le fait d'imposer le regroupement de communications rend $\mathcal{NP}-$difficile un problème qui était polynômial dans le modèle délai. Donc, nous pouvons nous attendre à ce que les problèmes d'ordonnancement sous BSP soient $\mathcal{NP}-$difficiles en général. Cependant, en dépit du fait que le problème de l'ordonnancement de chaînes sur BSP est difficile, nous avons été capables de proposer des heuristiques quasi optimales.

Les perspectives ouvertes sont, entre autres, l'étude des ordonnancements d'autres classes de graphe, ou même de graphes de précédence quelconques sous le modèle BSP. Une autre perspective intéressante est l'utilisation d'une généralisation du modèle BSP, le modèle BSPWB [91], dans lequel il existe des synchronisation partielles.

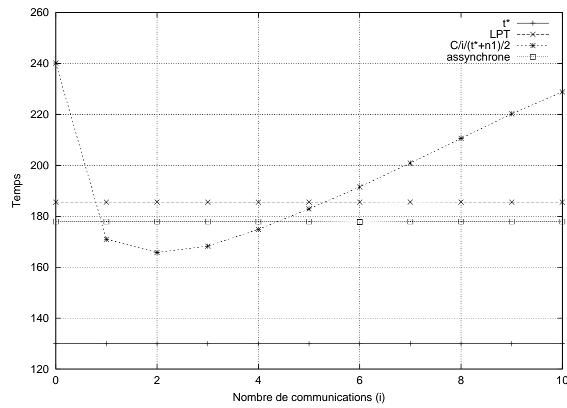

FIG. 4.13 – Influence du nombre de communications sur l'algorithme $C/i/(t^* + n_1)/2$.

Chapitre 5

Ordonnancement avec duplication

Résumé

Nous examinons dans ce chapitre les effets de la duplication de tâches sur le *makespan* de l'ordonnancement. Ce problème a été étudié auparavant pour un nombre de processeurs non borné. Quand le nombre de processeurs est limité, le problème a été analysé uniquement dans le cas où les tâches ainsi que les échanges de donnés sont unitaires. Nous proposons une extension du résultat au cas SCT.

5.1 La duplication

Les premières études sur l'ordonnancement dans le modèle délai n'ont pas pris en compte le coût des communications. Ces études ont considéré un nombre limité de ressources. Graham [55] a montré que les algorithmes de liste ont un rapport de performance par rapport à l'optimal de $2 - \frac{1}{m}$, où m est le nombre de processeurs. Lorsque les retards dûs à l'échange de messages ont été introduits, les problèmes sont devenus plus difficiles (voir la section 2.6 du chapitre 2). Dans ce cas, la performance des algorithmes de liste devient borné par $(2 - \frac{1}{m})w_{opt} + C_{comm}$ [71, 75], où C_{comm} est la somme des coûts de communication dans un chemin critique [1] et w_{opt} le *makespan* optimal sur m processeurs. La granularité ρ borne le rapport entre w_{opt} et C_{comm}, ce qui implique que le rapport de performance est $(2 - \frac{1}{m}) + \frac{1}{\rho}$. Donc,

[1] Le chemin critique avec le plus grand coût de communication.

le rapport de performance pour les algorithmes de liste dans le cas SCT est borné par $3 - \frac{1}{m}$ et cette borne est atteinte [75]. Hanen et Munier [56] ont proposé un algorithme d'approximation, qui n'est pas de liste, avec garantie de performance $\frac{7}{3} - \frac{4}{m}$ également pour le cas SCT.

Il existe deux façons de diminuer le surcoût dû à l'introduction des retards de communication : fournir un nombre de ressources illimitées et permettre la duplication. Nous allons examiner dans la section suivante l'analyse des problèmes avec nombre de processeurs non borné. Nous sommes intéressés par la performance des algorithmes de liste.

5.1.1 Nombre de processeurs illimité

L'ordonnancement avec duplication sur un nombre de processeurs non borné est un problème polynômial dans le cas SCT [21]. Dans le cas général le problème est $\mathcal{NP}-$difficile, mais il existe un algorithme polynômial avec garantie de performance 2 [84].

Papadimitriou et Yannakakis [84] ont d'abord montré que le problème $P_\infty|prec, p_j = 1, c_{ik} = c > 1, dup|w_{\max}$ est $\mathcal{NP}$-difficile avec une réduction au problème Clique [39]. Ensuite ils ont proposé une fonction d'estimation de la date de début d'exécution pour les tâches. Pour chaque tâche v, cette fonction $e(v)$ est définie par :

- Si v n'a pas de prédécesseur alors $e(v) = 0$;
- Sinon, soit $E = (u_1, \ldots, u_p)$ l'ensemble de tous les prédécesseurs de v triés en ordre décroissant par rapport à la fonction e. Étant défini k égal à $\min\{c+1, p\}$, $e(v)$ est défini par $e(u_k) + k$.

Ils ont proposé deux lemmes :

- **Lemme I.**[84] Il n'existe pas d'ordonnancement donc la tâche v est allouée avant la date $e(v)$.
- **Lemme II.** [84] Il existe un ordonnancement donc la tâche v est allouée avant la date $2e(v)$.

À partir de la preuve du lemme II les auteurs ont produit un algorithme avec rapport de performance 2. Finalement, ils ont proposé un algorithme plus général avec rapport de performance aussi égal à 2.

Colin et Chrétienne [21] ont proposé une solution polynômial pour le problème $P_\infty|prec, p_j, c_{ik}, dup|w_{\max}$ dans le modèle SCT, ce qui signifie que les temps de calcul des tâches sont plus grands que les temps d'échanges de données ($\min\{p_j\} \geq \max\{c_{ik}\}$). L'idée de base est le calcul des dates de disponibilité (de l'anglais *release dates*) pour chaque tâche. La date de

disponibilité b_v d'une tâche v est définie par les relations suivantes :
- Si v n'a pas de prédécesseur alors $b_v = 0$;
- Si v n'a qu'un prédécesseur u alors $b_v = b_u + p_u$;
- Sinon, soit U l'ensemble des prédécesseurs directs de v,

$$b_v = \min_{u \in U} \max\{b_u + p_u, \max_{k \in U - \{u\}} \{b_k + p_k + c_{kv}\}.$$

Un *arbre critique* est construit selon les dates de disponibilité. Un arc (u, v) fait partie de cet arbre si et seulement si $b_v < b_u + p_u + c_{uv}$. L'ordonnancement optimal est construit en allouant chaque chemin de l'arbre à un processeur. Les dates d'allocation des tâches correspondent aux dates de disponibilité. Cet algorithme est aussi valable si pour chaque tâche v, le poids du plus grand arc (u, v) est plus petit que le plus petit temps d'exécution parmi les prédécesseurs directs de v.

Le deux algorithmes précédents utilisent un grand nombre de processeurs. Dans le premier, à chaque tâche allouée, un nouveau processeur est utilisé. Dans le deuxième, le nombre de processeurs nécessaires est égal au nombre de feuilles de l'arbre critique.

Il existe aussi des algorithmes plus complexes. Park, Shirazi et Marquis [85] ont présenté un algorithme performant ainsi qu'une classification des algorithmes existantes. Le *makespan* de leur algorithme est compris entre le *makespan* optimal et le *makespan* optimal plus les communications dans le chemin critique.

5.1.2 Nombre de processeurs borné

À notre connaissance, le seul travail sur l'ordonnancement dans les modèles délai avec un nombre limité de processeurs a été proposé par Hanen et Munier [57]. Elles ont étudié le problème dans le cas UET-UCT. D'abord la notion de chemin de duplication est introduite. Lors de l'ordonnancement d'une tâche v ce chemin est construit avec des prédécesseurs de v qui doivent être dupliqués pour pouvoir exécuter v juste après la terminaison de son dernier prédécesseur. Un algorithme de liste qui utilise le chemin de duplication est alors proposé. Pour donner la garantie de performance de $2 - \frac{1}{m}$, les auteurs ont borné le temps d'inactivité de l'algorithme. Ils ont aussi prouvé que la borne est atteinte. C'est remarquable : la duplication efface la prise en compte des communications.

Nous proposons une extension du travail sur UET-UCT. Nous étudions la duplication sur un nombre limité de processeurs dans le modèle SCT. Notre étude est aussi fondée sur un chemin de duplication, mais comme le temps de communication varie, sa construction est plus subtile. La garantie de performance que nous proposons a un rapport de 2 à l'optimal.

5.2 Duplication dans le modèle SCT

Nous considérons l'ordonnancement d'un graphe $G(V, E)$ orienté et acyclique, l'ensemble $V = \{1, \ldots, n\}$ dénote les tâches et l'ensemble E les arcs. Le temps d'exécution d'une tâche $i \in V$ est dénommé p_i. À chaque arc d'une tâche j à une tâche i correspond un poids c_{ji}. Avec le modèle SCT nous avons $\frac{\min_{i \in V}\{p_i\}}{c} \geq 1$, où c est le poids de la plus grande communication, c'est-à-dire $c = \max_{j,i \in V}\{c_{ji}\}$. Nous voulons minimiser le *makespan* de l'ordonnancement sur m processeurs.

5.2.1 Notations

Chaque tâche $i \in V$ peut être allouée plusieurs fois dans un ordonnancement σ. Chacune de ses exécutions est dénommée *copie*. Nous dénotons la copie qui est ordonnancée le plus tôt comme étant la copie *originale*. Les références à une tâche $i \in V$ dans un ordonnancement σ sont toujours des références à la copie originale. Dans l'algorithme que nous proposons dans la section 5.3, aucune des copies n'est allouée avant la date d'allocation de la première copie ordonnancée. Donc, nous choisissons comme copie originale la première copie d'une tâche devant être allouée.

Nous ajoutons des précisions à la définition d'ordonnancement donné dans la section 2.5. D'abord nous proposons une table synthétique avec les notations utilisées :

Un ordonnancement valide σ assure que :

1. Chaque tâche $i \in V$ est exécutée *sans interruption* pendant un intervalle de temps p_i sur au moins un processeur ;
2. Si un arc (j, i) appartient à E, soit il y a une copie de j allouée au processeur π_i qui finit avant la date t_i, soit j est achevée avant la date $t_i - c_{ji}$.

Sans perte de généralité nous supposons qu'il n'y a pas d'arc transitif dans le graphe $G(V, E)$, c'est-à-dire $\forall j \in \Gamma_i, k \in \Gamma_i \rightarrow j \notin \Gamma_k$.

ordonnancement	σ
makespan de σ	ω
makespan d'un ordonnancement optimal sur m processeurs	w_{opt}
date et processeur d'allocation de $i \in V$	t_i et π_i
date de fin de $i \in V$	$f_i = t_i + p_i$
prédécesseurs directs de $i \in V$	Γ_i
le plus grand temps de communication	c

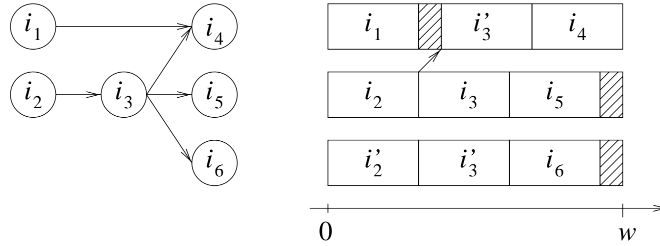

FIG. 5.1 – Exemple d'un ordonnancement valide avec duplication.

Dans l'exemple de la figure 5.1 toutes les tâches sont unitaires, les communications durent un quart du temps des tâches. L'exemple présenté illustre un ordonnancement optimal sur 3 processeurs. Lors de l'allocation des tâches i_4 et i_6, des chemins de duplication ont été utilisés.

Nous introduisons une série de définitions qui seront utiles à l'introduction du chemin de duplication. Ces définitions correspondent à un ordonnancement constructif, où tous les prédécesseurs de la tâche à laquelle nous faisons référence sont déjà alloués.

Définition 5.1 *La date de libre allocation $\tilde{d}_i$ d'une tâche $i \in V$ est :*

$$\tilde{d}_i = \max_{k \in \Gamma_i} \{f_k + c_{ki}\}.$$

Une tâche i peut être ordonnancée à partir de la date $\tilde{d}_i$ sur tout processeur libre. À partir de cette date il n'y a plus besoin de prendre en compte les relations de précédence. Les tâches ordonnancées après leur date libre sont dénommées *tâches libres*.

La figure 5.2 illustre un exemple, les prédécesseurs de la tâche i sont $\Gamma_i = \{j_1, j_2, j_3\}$. La figure montre aussi pour la tâche j_3 les temps de début, de fin, et le délai de communication avec la tâche i. Dorénavant dans les

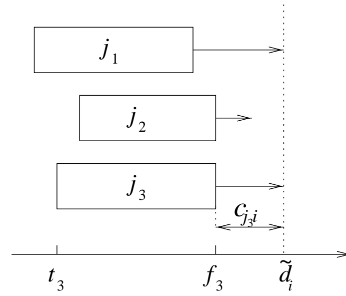

FIG. 5.2 – Date libre d'ordonnancement de la tâche i.

figures de ce chapitre nous allons présenté seulement les tâches impliquées dans le contexte.

Définition 5.2 *La date de disponibilité de la tâche $i \in V$, d_i, est la date d'exécution au plus tôt de la tâche i. Elle est calculée par rapport aux prédécesseurs directs de i. d_i est définie par :* 0, *si* $\Gamma_i = \emptyset$ *;*

$$\min_{j \in \Gamma_i} \max\{ \max_{k \in \Gamma_i | \pi_k = \pi_j} \{f_k\}, \max_{k \in \Gamma_i | \pi_k \neq \pi_j} \{f_k + c_{ki}\}\}, \ si \ \Gamma_i \neq \emptyset.$$

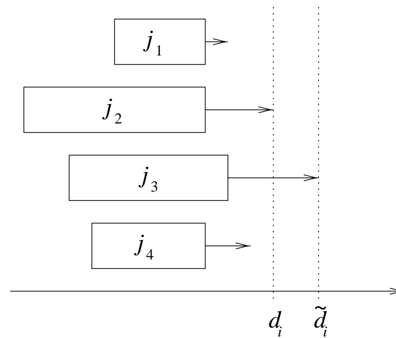

FIG. 5.3 – Dates de disponibilité et libre pour la tâche i.

Dans l'exemple de la figure 5.3 les prédécesseurs de la tâche i sont $\Gamma_i = \{j_1, j_2, j_3, j_4\}$. Si nous voulons ordonnancer la tâche i avant la date $\tilde{d}_i$, il faut allouer la tâche i sur le même processeur que la tâche j_3, ou sur le même que celui d'une de ses copies. La définition de la date de disponibilité nous fourni, d'une façon similaire à [21], une notion d'arc critique.

Définition 5.3 *Les prédécesseurs favoris d'une tâche $i \in V$ sont les tâches $j \in \Gamma_i$ tels que $d_i < f_j + c_{ji}$.*

Il est prouvé [21] que le graphe composé par des arcs critiques est une forêt où les arbres son orientés à partir d'une racine (*spanning outforest*). Avec une idée similaire nous démontrons que chaque tâche possède au plus un prédécesseur favori.

Propriété 5.1 *Une tâche $i \in V$ admet au plus un prédécesseur favori.*

Preuve : Si nous supposons qu'il existe deux ou plus prédécesseurs favoris pour la tâche i, nous tomberons en contradiction avec l'hypothèse SCT. La preuve complète est dans l'annexe D, section D.1.❑

Lorsqu'une tâche $i \in V$ n'a pas de prédécesseur favori, nous avons alors soit $\Gamma_i = \emptyset$, soit il existe deux prédécesseurs directs distincts $j_1, j_2 \in \Gamma_i$ tels que $f_{j_1} + c_{j_1 i} = f_{j_2} + c_{j_2 i}$ (voir l'exemple de la figure 5.2). Dans ce cas $d_i = \tilde{d}_i$.

Pour finir cette section nous introduisons la notion de délai d'une tâche par rapport à son prédécesseur favori. Lors de la construction du chemin de duplication proposé par Hanen et Munier [57] il n'existe pas de temps d'inactivité entre une tâche et son prédécesseur favori. En effet si ce délai n'est pas 0, il est forcément au moins 1, donc il n'y a plus besoin du chemin de duplication. Avec l'hypothèse SCT le chemin de duplication peut avoir des délais. Le délai d'une tâche à son prédécesseur favori est :

Définition 5.4 *Si j est le prédécesseur favori de i ($d_i < f_j + c_{ji}$), Δ_i est la différence $t_i - f_j$. Si Δ_i est positif nous disons que la tâche i a été retardée.*

La figure 5.4 illustre un exemple où la tâche i est retardée.

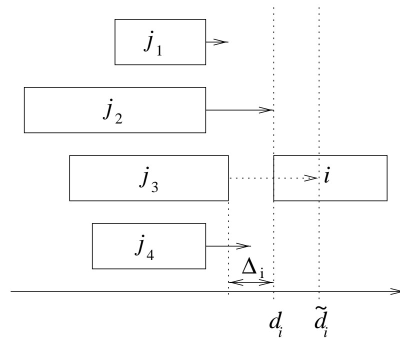

FIG. 5.4 – Exemple de tâche retardée.

5.3 Algorithme EDTF

Nous proposons un algorithme de liste fondé sur la date réalisable d'une tâche. Une version préliminaire est présentée dans l'algorithme 5.1. La version complète est exhibée dans l'algorithme 5.2. La description du calcul de la date réalisable lorsque la duplication est permise sera donnée par la suite.

L'ordonnancement est construit d'une façon itérative, à chaque pas l'ensemble de tâches prêtes à ordonnancer F est considéré, une tâche est alors choisie et allouée.

Algorithm 5.1 Algorithme EDTF

$\sigma = \emptyset$, $F = \{i | \Gamma_i = \emptyset\}$;
`tant que` $F \neq \emptyset$ `faire`
  `pour` chaque tâche $i \in F$ `faire`
    calculer la date réalisable τ_i dans l'ordonnancement σ;
  `fin pour`
  allouer dans σ la tâche avec la plus petit date réalisable;
  mettre à jour l'ensemble des tâches prêtes à ordonnancer F;
`fin tant que`

Dans l'algorithme 5.1 lors de l'allocation d'une nouvelle tâche i sur l'ordonnancement σ il peut s'avérer nécessaire d'allouer des copies de ses prédécesseurs de façon à respecter la date réalisable τ_i. Dans ce cas, les tâches à dupliquer seront déterminées lors du calcul de τ_i.

5.3.1 Chemin de duplication

Pour le calcul de la date réalisable d'une tâche i, nous avons besoin de déterminer les prédécesseurs de i qui doivent être dupliqués. Nous allons tout d'abord caractériser le chemin de duplication dans le cas d'un nombre illimité de processeurs. Ensuite nous montrerons l'utilisation de ce chemin avec un nombre limité de processeurs. Il existe des cas où nous n'allons dupliquer qu'une partie du chemin de duplication.

Nous présentons l'idée de base et ensuite la construction détaillée. Si la tâche i n'a pas de prédécesseur favori nous pouvons allouer la tâche i à la date $\tilde{d}_i$ à un processeur virtuel π (nous supposons que π est inactif). Il n'y a pas

besoin de dupliquer des tâches. Sinon, pour pouvoir exécuter la tâche i avant sa date libre $\tilde{d}_i$ sur un processeur virtuel π, il faut dupliquer son prédécesseur favori j sur π. Si j n'a pas de prédécesseur favori, sa copie peut être allouée sur π à l'instant t_j. Sinon, il faut également dupliquer le prédécesseur favori de j, et ainsi de suite. Un autre critère qui peut stopper la construction du chemin de duplication est la somme des délais. Si la somme des délais dans le chemin de duplication est plus grande que c, il vaut mieux faire la communication au lieu de dupliquer.

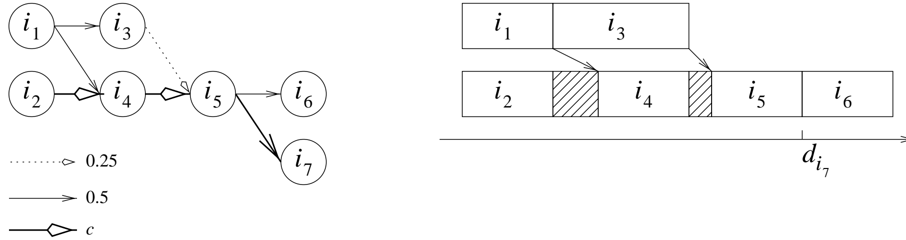

FIG. 5.5 – Exemple de construction d'un ordonnancement avec duplication.

Dans l'exemple de la figure 5.5, la tâche i_3 a un temps d'exécution de 1,5. Le temps d'exécution de toutes les autres tâches est unitaire. La valeur de la communication est donnée selon le type de l'arc. D'après l'hypothèse SCT, nous avons $c \leq 1$. La construction du chemin de duplication pour la tâche i_7 dépend de la valeur de c. Si $c \geq 0.75$ le chemin de duplication est construit jusqu'à la tâche i_2. Sinon le chemin de duplication est construit jusqu'à la tâche i_4.

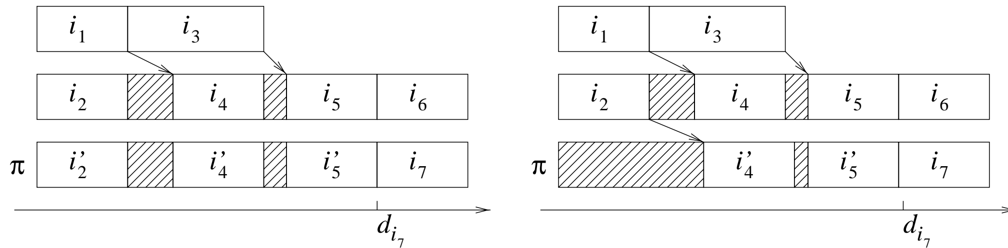

FIG. 5.6 – Construction d'un ordonnancement avec duplication sur un processeur virtuel π.

La figure 5.6 illustre les chemins de duplication sur un processeur virtuel π, à gauche $c = 1$ et à droite $c = 0.6$. Dans ces deux exemples nous obser-

vons qu'il existe des temps d'inactivité dans le chemin de duplication. Ces temps sont dûs aux autres relations de précédence qui ne sont pas dues à des prédécesseurs favoris. Par exemple, dans la figure 5.6, à droite de la tâche i'_5, la copie de la tâche i_5 ne peut pas être allouée avant car elle dépend d'une donnée produite par la tâche i_3.

Lors de la construction d'un chemin de duplication il faut considérer ces délais. En effet une copie ne peut jamais être ordonnancée avant la tâche originale correspondante.

Le chemin de duplication $\mathcal{D}_i$ de la tâche i est construit récursivement, à chaque étape le prédécesseur favori de la dernière tâche ajoutée est additionné à $\mathcal{D}_i$. Deux critères peuvent stopper la construction récursive, la somme des délais en $\mathcal{D}_i$ devient plus grande que c, ou une tâche libre.

Le chemin de duplication $\mathcal{D}_i = (x_0, x_1, \ldots)$ est défini récursivement. La tâche i correspond à la tâche x_0 virtuellement allouée à l'instant d_i. Si nous supposons que le chemin a été construit jusqu'à la tâche x_k, alors la prochaine tâche du chemin x_{k+1} est définie comme le prédécesseur favori de x_k. Δ dénote la somme des délais dans le chemin ($\Delta = \sum_{i=0}^{k-1} \Delta_k$). Deux situations peuvent stopper la construction :

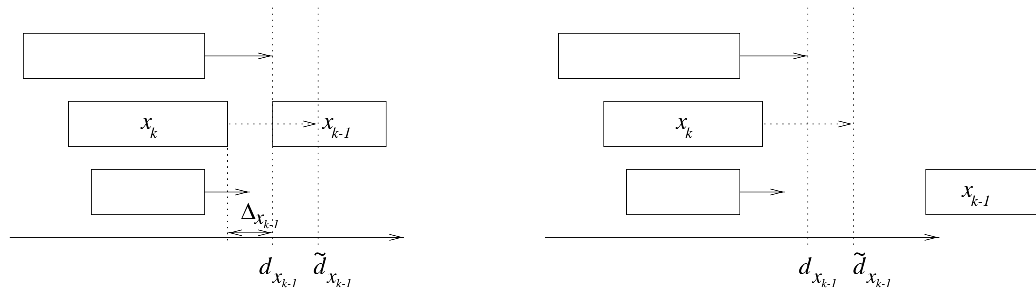

FIG. 5.7 – Les deux possibilités lors de la construction du chemin de duplication. Dans les figures nous observons la tâche x_{k-1} et ses prédécesseurs. À gauche x_{k-1} a un prédécesseur favori. À droite x_{k-1} est une tâche libre.

- $\Delta \geq c$, c'est-à-dire la somme des délais du chemin est plus grande que le plus grand coût de communication. La construction est arrêtée et la tâche x_k devient l'origine du chemin. Cette condition n'était pas satisfaite par la tâche x_{k-1} donc x_{k-1} est une tâche retardée. La partie gauche de la figure 5.7 illustre un exemple.
- x_k est une tâche libre. Pour rester consistant avec le cas antérieur nous

supposons que x_k est une tâche retardée avec un prédécesseur favori fantôme x_{k+1} (sans coût de calcul) allouée à l'instant $t_{x_k} - c$. La tâche x_{k+1} devient l'origine du chemin. La figure 5.7 coté droit illustre un exemple de tâche libre.

Nous allons centrer notre attention sur les tâches retardées du chemin dupliqué.

Définition 5.5 *Nous notons les tâches retardées du chemin de duplication $\mathcal{D}_i$ par $(b_0, \ldots, b_r)$, et par $(a_o, \ldots, a_r)$ leur prédécesseurs favoris, qui appartiennent aussi au chemin $\mathcal{D}_i$. Par convention b_0 est la tâche i virtuellement ordonnancée à l'instant d_i avec retard $\Delta_{b_0} = d_i - f_j$, et $a_0 = j$.*

5.3.2 La date réalisable

Pour estimer la date réalisable d'une tâche i, nous calculons les temps possibles d'allocation sur tous les processeurs. Puis nous choisissons le temps minimal. Si la tâche i ne possède pas de prédécesseur favori, pour estimer la date réalisable sur un processeur π nous choisissons le maximum entre la date de disponibilité d_i et l'instant auquel le processeur π devient inactif.

Sinon, pour estimer la date réalisable, nous construisons le chemin de duplication $\mathcal{D}_i$ et nous vérifions pour chaque processeur les tâches qui doivent être dupliquées. L'idée de base est de permettre la duplication seulement à partir d'une tâche a_l du chemin, et dupliquer le chemin jusqu'à la tâche j.

Nous voulons calculer la date réalisable de la tâche i sur un processeur π, nous dénotons par $\sigma_{\pi,i}$ ce temps et par α la dernière tâche allouée sur le processeur π. α est une tâche originale car l'unique raison pour dupliquer une tâche est de permettre l'exécution d'un de ses successeurs originaux sur le même processeur.

Il existe quatre possibilités :

cas 0 : i ne possède pas de prédécesseur ;

cas 1 : i ne possède pas de prédécesseur favori ;

cas 2 : i possède un prédécesseur favori j et est ordonnancé sur π_j juste après j, ou i est ordonnancée par un schéma de duplication ;

cas 3 : i possède un prédécesseur favori et n'est pas ordonnancé par un schéma de duplication.

Dans le cas 0, il est clair que $\sigma_{\pi,i} = f_\alpha$.

Dans le cas 1, $\sigma_{\pi,i} = \max\{d_i, f_\alpha\}$.

Dans le cas 2 et 3, si le prédécesseur favori de la tâche i est α alors $\sigma_{\pi,i} = d_i$ (cas 2). Sinon, nous avons j le prédécesseur favori de i avec $j \neq \alpha$. Selon la définition 5.5, nous dénotons par $k = k(\pi)$ l'indice du plus grand chemin de duplication qui vérifie :

$$f_\alpha < t_{b_k} + c. \tag{5.1}$$

Si k n'existe pas alors nous avons $\sigma_{\pi,i} = f_\alpha$. En effet, si nous considérons la tâche b_0 du chemin (celle qui représente la tâche i ordonnancée à l'instant d_i) nous avons $f_\alpha \geq d_i + c$. C'est-à-dire la tâche α finit son exécution après la date libre de i ($\tilde{d}_i = f_j + c_{ji}$). Donc il n'y a pas besoin de dupliquer pour ordonnancer i à l'instant $\sigma_{\pi,i}$, Ce que correspond au cas 3.

Dans le cas 2. Il existe une valeur de k qui vérifie l'équation 5.1. Nous définissons $\sigma_{\pi,i}$ comme l'instant de début de la tâche i lorsque sont chemin de duplication est répliqué de b_k jusqu'à j. Ce sous-chemin sera dénoté $\mathcal{D}_i^k$. Observons que si $k = 0$, le sous-chemin est vide. Dans le but de simplifier les descriptions nous assumons que le délai de communication de a_k est exactement c. Lors de la duplication, les tâches dupliquées commencent au plus tôt au même instant que ses correspondants originaux.

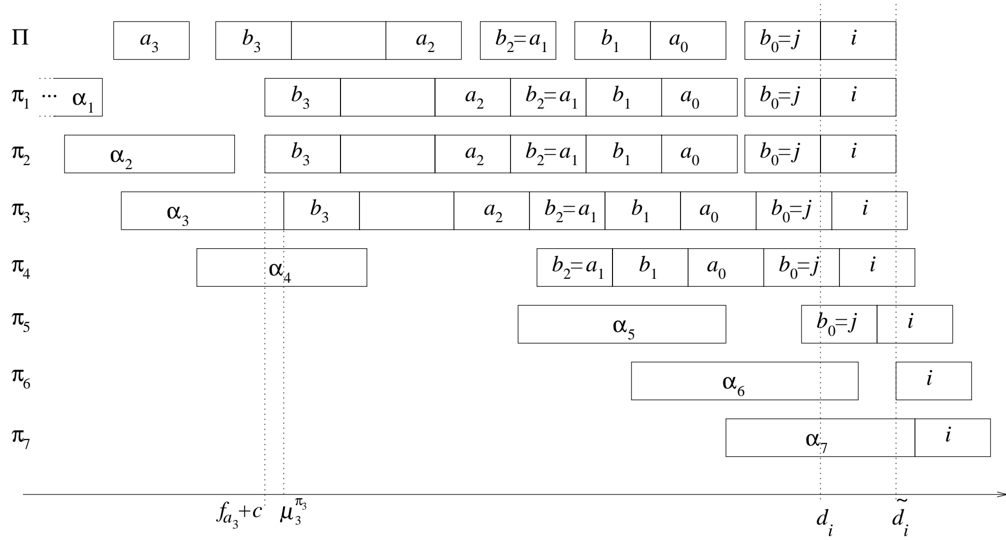

FIG. 5.8 – Ordonnancement réalisable de la tâche i sur plusieurs processeurs.

La figure 5.8 illustre les possibilités lors de l'allocation de la tâche i sur

les processeurs de π_1 à π_7. Dans cet exemple, la tâche α_i est la dernière tâche exécutée par le processeur π_i. Le chemin de duplication $\mathcal{D}_i$ est représenté dans le processeur virtuel Π. Les tâches du chemin sont dénommées selon la définition 5.5. Le délai de communication à partir des tâches c_i est assumé comme étant c.

Sur le processeur π_1 la date de la fin de la tâche α_1 est plus petite que la date de début de la tâche à l'origine du chemin a_3. Sur le processeurs π_2 la date de la fin de la tâche α_2 est plus petite que $t_{b_3} + c$. Dans les deux cas précédents le chemin est dupliqué à partir de la tâche b_3 ($f_{\alpha_1} < f_{\alpha_2} < t_{b_3} + c$). Nous pouvons observer que les copies des tâches b_0 ont été retardées pour respecter les autres relations de précédence. Nous allons voir grâce aux propriétés 5.2 et 5.4 que dans ce cas $\sigma_{\pi_1,i} = \sigma_{\pi_2,i} = d_i$.

Sur le processeur π_3 le chemin est dupliqué à partir de la tâche b_3, comme pour les processeurs π_1 et π_2 ($f_{\alpha_3} < t_{b_3} + c$). Cependant $f_{\alpha_3} > f_{a_3} + c$ donc le chemin est retardé par la fin de la tâche α_3. Sur les processeurs π_4 et π_5 le chemin est dupliqué respectivement à partir de b_1 et $b_0 = j$. Sur le processeur π_6 le chemin dupliqué est vide. Finalement sur le processeur π_7, $f_{\alpha_7} > t_{b_0} + c$, donc il n'y a pas de duplication (ce qui correspond au cas 3).

Les propriétés suivantes seront utiles pour démontrer la garantie de performance de l'algorithme. Les preuves complètes se trouvent dans la section D.2 de l'annexe D.

Propriété 5.2 *Si il existe un délai sur le processeur π entre les tâches du sous-chemin dupliqué $\mathcal{D}_i^k$ alors $\sigma_{\pi,i} = d_i$.*

Idée de la preuve : S'il existe un délai, il est provoqué par une communication qui provient d'un prédécesseur non favori. Ce même prédécesseur retarde aussi la tâche originale, donc à partir de cette copie, les tâches du sous-chemin commencent au même temps que ces originaux.❑

Nous introduisons $\mu_k^\pi = \max\{0, f_\alpha - (f_{a_k} + c)\}$. Cette valeur correspond au délai de début du sous chemin $\mathcal{D}_i^k$, c'est-à-dire le sous chemin débuta à l'instant $f_{a_k} + c + \mu_k^\pi$ (ce délai est visible sur le processeur π_3 de la figure 5.8).

Propriété 5.3 *Lors d'une allocation par duplication, l'équation 5.2 donne une borne du temps pour $\sigma_{\pi,i}$,*

$$\sigma_{\pi,i} \leq d_i + c + \mu_k^\pi - \sum_{l=0}^{k} \Delta_{b_l}. \tag{5.2}$$

Un résultat immédiat à partir de la propriété précédente est : $\sigma_{\pi,i} \leq d_i + c$.

Propriété 5.4 *Si a_r dénote l'origine du chemin de duplication $\mathcal{D}_i$ de i et si π est inactif après l'instant $f_{a_r} + c$ alors $\sigma_{\pi,i} = d_i$*

La date réalisable σ_i d'une tâche i est le minimum des dates réalisables entre tous les processeurs, c'est-à-dire :

$$\sigma_i = \min_{\pi}\{\sigma_{\pi,i}\}.$$

Lemme 5.1 *Étant donné π tel que $\sigma_{\pi,i} > d_i$. Nous avons pour tout π':*

$$\sigma_{\pi,i} \leq \sigma_{\pi',i} \Rightarrow k(\pi) \geq k(\pi').$$

Une idée graphique de ce lemme est illustrée dans la figure 5.8, plus le chemin est dupliqué tard, plus la tâche i commence tard.

5.3.3 Analyse de l'algorithme

Pour faire l'analyse de l'algorithme nous bornons son temps total d'inactivité en fonction du *makespan* optimal avec un nombre illimité des processeurs. Une technique similaire a été utilisée dans [57]. Nous introduisons les notations suivantes pour un ordonnancement optimal sur un nombre illimité de processeurs : t_i^∞, w^∞ qui sont respectivement le temps d'allocation de la tâche i et le *makespan*.

Lemme 5.2 *Si $\mathcal{I}[0,t)$ dénote le temps d'inactivité total entre $[0,t)$ dans l'ordonnancement σ (ce temps inclut aussi les copies non originales d'une tâche), alors pour toute tâche i nous avons :*

$$\mathcal{I}[0,t_i) \leq mt_i^\infty.$$

Preuve : Pour démontrer le lemme nous prouvons le résultat équivalent : $\mathcal{W}_\pi[0,t_i) \geq t_i - t_i^\infty$ pour tous les processeurs, où $\mathcal{W}_\pi[0,t_i)$ dénote le temps total des tâches originales exécutées par le processeur π pendant l'intervalle $[0,t)$. Nous divisons la preuve en plusieurs cas, en fonction des cas pour la recherche de la date réalisable d'un tâche (voir section 5.3.2). La preuve complète se trouve dans l'annexe D section D.3.

Théorème 5.1 *La garantie de performance de l'algorithme EDTF est 2.*

Preuve : Soit α une des tâches qui finit à l'instant w. D'après le lemme précédent $\mathcal{I}[0,w) \leq mt_i^\infty + (m-1)p_\alpha \leq mw^\infty - p_\alpha$. Nous avons $w_{opt} \geq \sum_{k=1}^{|V|} p_k/m$, donc par la conservation de la surface de calcul :

$$mw \leq \sum_{k=1}^{|V|} p_k/m + \mathcal{I}[0,w) \leq mw_{opt} + mw^\infty \leq 2mw_{opt}.$$

Algorithm 5.2 Algorithme EDTF.

$\sigma = \emptyset$, $U = F = \{i \mid \Gamma_i^- = \emptyset\}$;

`tant que` $F \neq \emptyset$ `faire`

  `pour` chaque tâche $i \in U$ `faire`

    calculer son chemin de duplication avec la date de début des tâches retardées;

    calculer sa date réalisable $\tau_{\pi,i}$ pour tous les processeurs;

  `fin pour`

  $F = F \cup U$;

  choisir la tâche i' avec $\tau_{\min,i'} = \min_{i \in F,\pi} \tau_{\pi,i}$ (la date réalisable minimal);

  allouer i' sur le processeurs π_0 tel que $\tau_{\pi_0,i'} = \tau_{\min,i'}$;

  $F = F - \{i'\}$;

  mettre à jour $\tau_{\pi,i}$ pour toutes les tâches i dans F, pour tous les processeurs π;

  U = ensemble des tâches qui sont devenues prêtes (successeurs de i');

`fin tant que`

Proposition 5.1 *L'algorithme EDTF peut être implanté en $O(n^2 \log n)$, où n est le nombre de tâches du graphe de précédence.*

Preuve : Pour estimer le temps d'exécution de l'algorithme, nous estimons les temps de ses quatre principales étapes, la boucle `pour`, le choix, l'allocation et la mise à jour. Cette dernière étape possède la plus grande complexité $O(n \log n)$. Donc, la complexité totale est $O(n^2 \log n)$. La preuve complète se trouve dans l'annexe D, section D.4.

5.4 Conclusion

Nous avons étudié l'influence de la duplication dans la réduction du surcoût dû à l'introduction du coût de communication. Il existe des résultats pour un nombre illimité de processeurs et pour le cas UET UCT avec un nombre borné de processeurs. Dans le cas UET UCT le surcoût de communication pour les algorithmes de liste a été éliminé, c'est-à-dire les algorithmes de liste ont un rapport de performance $2 - \frac{1}{m}$ lorsque la duplication est permise.

Nous avons proposé une extension du résultat antérieur pour le cas SCT. Nous avons fourni un algorithme de liste avec rapport de performance 2 pour ce cas. Les techniques de duplication ont permis la minimisation presque totale du surcoût de communication.

Chapitre 6

Ordonnancement des communications

Résumé

Dans ce chapitre nous étudions l'échange de messages entre processeurs. Nous envisageons l'échange de messages de tailles différentes, dans une machine parallèle où le coût de communication est linéaire. Nous étendons les résultats de complexité existants pour montrer que ce problème est aussi $\mathcal{NP}-$complet. Nous présentons plusieurs algorithmes d'approximation ainsi que leur analyse de performance. Nous examinons les points forts de chacun d'entre eux. Nous proposons alors des algorithmes combinés, lesquels produisent les plus petits *makespans*. Nous achevons le chapitre avec des expérimentations et des simulations. Celles-ci confirment le bon comportement des algorithmes combinés.

6.1 Le problème de l'échange de messages

Nous nous intéressons à la minimisation du temps nécessaire pour effectuer l'échange de messages dans un ensemble de processeurs reliés par un réseau d'interconnexion. La motivation de ce travail provient d'applications réelles, où il existe des périodes de calcul et d'échanges de donnés alternées. Le comportement de ces applications peut être modélisé par le modèle BSP (présenté dans la section 2.4.5). Une première version de ce travail a été présenté dans [46].

Nous analysons le problème de l'échange de messages de la même manière

qu'un ordonnancement de tâches. Le graphe de précédence est constitué des processeurs qui correspondent aux sommets. Les arcs indiquent la source, la destination et la taille des messages. Au lieu d'attribuer des dates de début pour les tâches (sommets), nous allons ordonnancer les arcs.

Ce problème d'ordonnancement sur un modèle qui ne considère ni les effets de gestion de communication ni la congestion est facile. Il suffit de débuter toutes les communications simultanément à l'instant 0, le *makespan* correspond à la communication la plus longue. Cependant, cet ordonnancement ne fournira pas une bonne estimation de temps pour les machines réelles actuelles, où la gestion de communication et les effets de la congestion ne sont pas négligeables. Notre positionnement n'est pas opposé au modèle délai, bien au contraire. Toutefois, pour les ordonnancements de communications il existe le besoin d'une estimation plus précise, car ils peuvent être exécutés un très grand nombre de fois pendant un algorithme. Il faut considérer donc plus finement la gestion des communications.

Le choix du modèle de communication de messages entre les processeurs a été inspiré par les propriétés suivantes : la taille du message influence le temps nécessaire pour la communication ; un processeur peut communiquer avec un nombre limité de processeurs de façon simultanée ; la distance entre les processeurs n'est pas un paramètre dominant et la plupart des machines actuelles sont asynchrones. Avec ces restrictions nous pouvons choisir deux des modèles vues dans la section 2.4, le modèle LogP et le modèle où les temps de transmission se conforment au modèle linéaire. Nous avons choisi le modèle linéaire car nous ordonnançons seulement les communications, nous ne considérons pas les tâches éventuellement ordonnancées sur les processeurs.

Nous limitons le nombre de communications simultanées de chaque processeur à 1, c'est-à-dire chaque processeur peut envoyer ou recevoir au plus un message au même instant. Cette limitation est bien connue dans la littérature par 1-port *full-duplex*. Il existe des ordinateurs où la capacité de transmission/réception n'est pas limité à 1. Mais, l'hypothèse 1-port *full-duplex* s'avère intéressante car elle sert aussi à limiter la congestion du réseau. Avec cette hypothèse le nombre de messages qui circulent à un instant donné dans une machine est au plus égal à son nombre de processeurs.

6.1.1 Description du problème

Le problème de l'échange de messages, MEP, est décrit par un graphe orienté d'ordre n, $dG(V, E)$. Les sommets $V = \{p_0, \ldots, p_{n-1}\}$ représentent les processeurs, et les arcs les messages à transmettre. Chaque arc $e = (p_i, p_j)$ possède un poids entier $w(e)$, qui correspond à la taille du message à transmettre du sommet p_i au sommet p_j. S'il existe plusieurs messages d'un processeurs p_i à un processeur p_j, il existe un seul arc de p_i à p_j avec un poids $w(e)$ égal à la somme de tous messages. Le but est de trouver un ordonnancement valide qui minimise le temps d'échange de tous les messages du graphe.

Au cours de ce chapitre nous allons représenter les données d'entrée du MEP sous deux autres formes, celle d'un graphe biparti et à l'aide d'une matrice.

Représentation par un graphe biparti

Étant donné un graphe orienté $dG(V, E)$ nous construisons le graphe biparti $bG(V' = V_1 \bigcup V_2, E')$. À chaque sommet $p_i \in V$ correspond deux sommets, $p_{i_1} \in V_1$ et $p_{i_2} \in V_2$. À chaque arc $e = (p_i, p_j) \in E$ est associé une arête $e' = \{p_{i_1}, p_{j_2}\} \in E'$ de poids $w(e') = w(e)$. La figure 6.1 illustre les deux représentations du problème.

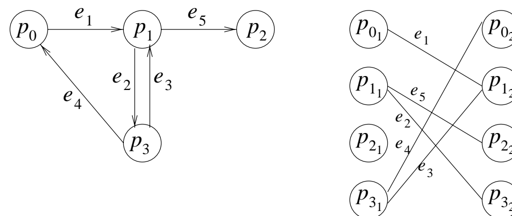

FIG. 6.1 – Représentations par un graphe orienté et par un graphe biparti.

Représentation par une matrice

Un graphe du MEP $dG(V, E)$ peut aussi être représenté par une matrice carrée $A = (m_{i,j})$ de dimension n. Les éléments de A sont les poids des arcs de $dG(V, E)$. S'il existe un message de p_i à p_j alors $m_{i,j} = w((p_i, p_j))$. Sinon

$$\begin{pmatrix} 0 & e_1 & 0 & 0 \\ 0 & 0 & e_5 & e_2 \\ 0 & 0 & 0 & 0 \\ e_4 & e_3 & 0 & 0 \end{pmatrix}$$

FIG. 6.2 – Représentation du problème MEP par une matrice.

$m_{i,j} = 0$. La figure 6.2 montre la représentation du graphe de la figure 6.1 par une matrice.

Ces trois représentations du MEP étant équivalentes, nous choisissons dans la suite de ce chapitre celle la mieux adaptée au contexte.

6.1.2 Le modèle

Dans cette section nous détaillons le modèle pour l'ordonnancement des messages. Nous divisons la présentation en trois sous modèles :

Modèle de communication : 1-port *full-duplex*. Les communications sont faites les une après les autres, c'est-à-dire que pour un processeur, à partir du moment où une transmission ou une réception commence, la prochaine opération du même type ne peut débuter qu'au terme de la précédente. Nous avons utilisé ce modèle pour les analyses. Lors des implantations des algorithmes, nous avons employé des envois non bloquants et des réceptions bloquantes [81].

Modèle de transmission : linéaire, c'est-à-dire que pour transmettre un message de taille L d'un processeur p_i à un processeur p_j, le temps nécessaire est $\beta + L\gamma$.

Modèle de synchronisme : il n'y a pas de restriction sur la cadence des processeurs. Cependant, tous les processeurs débutent l'algorithme au même instant. Les algorithmes sont composés de phases de communication. Dans chaque phase un motif qui respecte les contraintes de communication est exécuté. Un processeur peut commencer une nouvelle phase sans attendre la fin de la même phase sur les autres processeurs. Le *makespan* d'un algorithme est la différence entre l'instant initial et le moment où la dernière communication s'achève.

Deux autres paramètres influencent l'ordonnancement : la préemption des messages et la possibilité de commuter des messages (de l'anglais *message forwarding*).

Division de messages : correspond à la préemption dans un ordonnancement de tâches. Lorsque la division de messages est permise, les messages peuvent être divisés et envoyés en plusieurs transmissions (pas forcément consécutives).

Commutation de messages : lors de sa transmission, un message peut être envoyé directement à sa destination ou peut transiter par des processeurs intermédiaires. Le transit se caractérise par le stockage complet du message dans un processeur intermédiaire. Si le message transite par d processeurs son coût de transmission est $d(\beta + L\gamma)$. L'intérêt de la commutation de messages est d'utiliser un seul temps d'initialisation pour la transmission de plusieurs messages. Quand la commutation de messages (qui vient de l'anglais *message forwarding*) n'est pas autorisée, les messages doivent être envoyés directement à leurs destinataires. Dans ce cas, chaque message n'est transmis qu'une fois.

Il n'existe pas, a priori, de restriction sur l'architecture liée à ces deux paramètres. Nous utilisons ces paramètres dans le but de classer la complexité du problème ainsi que les heuristiques proposées.

6.1.3 Notations et bornes inférieures

Dans les études de problèmes d'ordonnancement de cette thèse nous avons envisagés l'efficacité ou la minimisation du surcoût dû au parallélisme : $T_c + Id$. Pour l'étude de l'ordonnancement de communications, nous fournissons deux types de bornes inférieures, une sur le nombre d'initialisations (β) et l'autre sur la bande passante (γ).

Étant donné un graphe $dG(V, E)$, le degré entrant (de l'anglais *in-degree*) $\Delta_r(p_i)$ d'un sommet $p_i \in V$ est le nombre d'arcs incidents à p_i. Le degré sortant (de l'anglais *out-degree*) $\Delta_s(p_i)$ d'un sommet $p_i \in V$ est le nombre d'arcs sortants de p_i. Nous introduisons $\Delta_r = \max_{p_i \in V}\{\Delta_r(p_i)\}$ et $\Delta_s = \max_{p_i \in V}\{\Delta_s(p_i)\}$.

Nous présentons plusieurs propriétés immédiates.

Propriété 6.1 *Le nombre de pas d'initialisation nécessaire pour ordonnancer $dG(V, E)$ sans la commutation de messages est au moins* $\max\{\Delta_s, \Delta_r\}$.

Lorsque la commutation de messages est permise, un processeur peut envoyer plus d'un message par le même lien avec un seul temps d'initialisation. La preuve de la propriété suivante est inspirée des algorithmes d'échange total pour l'hypercube où les messages sont regroupés [26].

Propriété 6.2 *Le nombre de pas d'initialisation disjoints pour achever un échange personnalisé entre n processeurs avec commutation de messages est* $log_2 n$.

Nous proposons des limites évidents pour la bande passante, elles sont relatives au temps dont un processeur a besoin pour envoyer ou recevoir ses messages. Chaque sommet p_i doit envoyer les messages $S_{p_i} = \{e \in E | e = (p_i, p_j), p_j \in V\}$ ce qui prend au moins le temps $t_s = \max_{p_i \in V} \sum_{e \in S_{p_i}} w(e)\gamma$. Nous avons des résultats similaires du coté de la réception, $R_{p_i} = \{e \in E | e = (p_j, p_i), p_j \in V\}$ et $t_r = \max_{p_j \in V} \sum_{e \in R_{p_j}} w(e)\gamma$. Donc, nous avons la propriété suivante :

Propriété 6.3 *Le temps pour achever les communications est au moins* $\max\{t_s, t_r\}$.

Nous pouvons mettre ensemble les propriétés pour le nombre de pas d'initialisation et pour la bande passante. Pour cela, il faut supposer qu'un modèle est synchrone ou que les valeurs maximales sont dues au même processeur.

Propriété 6.4 *Si la commutation de messages n'est pas autorisée, le makespan d'un ordonnancement de communications est au moins* $\max\{\Delta_r, \Delta_s\}\beta + \max\{t_s, t_r\}$.

6.2 Travaux précédents

Plusieurs auteurs ont mené des études sur des problèmes similaires à l'échange de messages. Nous allons diviser cette section en trois parties : une avec des études dans des modèles plus restreints, une avec des problèmes similaires et finalement une avec des résultats connus sur le MEP.

6.2.1 Modèles plus restreints

Certaines recherches ont considéré le problème de l'échange de messages dans des réseaux avec une topologie donnée. Parmi d'autres nous faisons référence aux études pour la grille [36, 87, 96] et pour l'hypercube [8, 11].

Gonzalez [52, 53] a examiné le problème dans un réseau complet avec et sans la commutation de messages. La plupart des solutions de la littérature considèrent des messages de même taille.

6.2.2 Problèmes similaires

L'étude de l'affectation des créneaux de temps dans un satellite (de l'anglais *time slot assignement for satellite systems*) ressemble au MEP. Le nombre de canaux différents (*transponders*) du satellite limite le nombre de messages qui peuvent transiter simultanément. Ce problème peut être vu comme le MEP dans un réseau complet où chaque processeur peut envoyer un message destiné à lui-même et les messages peuvent être divisés. L'algorithme est synchrone et ne prend pas en compte le coût d'initialisation. Une solution optimale a été proposée par Bongiovanni, Coppersmith et Wong [12]. D'autres auteurs ont étudié des variations de ce problème [7, 13].

Un autre problème semblable est l'ordonnancement dans des réseaux de fibre optique étoile passifs [92] (de l'anglais *TDM/WDM scheduling in passive star fiber optics network*). Une étoile passive avec N noeuds admet C longueurs d'ondes distinctes (généralement $C < N$). Chaque noeud possède un transmetteur et un récepteur. Chaque transmetteur peut être réglé, en temps Δ, à une longueur d'onde. Chaque récepteur est affecté à une longueur d'onde fixée. La circulation est donnée par une matrice $N \times N$, $D = (d_{i,j})$, où $d_{i,j}$ correspond au nombre de mots à envoyer du noeud i au noeud j. L'objectif du problème est de trouver un ordonnancement valide dans lequel chaque transmetteur utilise une seule longueur d'onde à chaque instant. La recherche d'un ordonnancement avec *makespan* minimal sans préemption est $\mathcal{NP}-$difficile pour $\Delta = 0$ et $C > 2$ [72] et pour $\Delta \geq 0$ et $C \geq 2$ [92]. Dans ce problème le coût d'initialisation ainsi que le nombre maximal de messages qui circulent au même instant sont pris en compte. Cependant le fonctionnement est synchrone.

6.2.3 Résultats précédents pour le MEP

Messages indivisibles

Une des premières études a été réalisée par Coffman, Garey, Johnson et Lapaugh [20]. Dans un graphe non orienté les sommets correspondent aux processeurs et les arêtes aux fichiers et leur temps de transfert. L'objectif

est la minimisation du temps nécessaire au transfert de tous les fichiers. Ils ont examiné l'ordonnancement de transfert de fichiers. Ils n'ont autorisé ni la commutation ni la division de messages. Dans leur modèle, le nombre de liens d'un processeur utilisés par des communications simultanées est un paramètre du problème. À partir de la propriété suivante, nous dérivons deux corollaires :

Propriété 6.5 *Étant donné un problème MEP, nous pouvons trouver un problème d'ordonnancement de transfert de fichiers équivalent.*

Preuve : la preuve est simple, à partir de la représentation du MEP par un graphe biparti. Car, lorsque la division de messages n'est pas autorisée, le coût d'initialisation peut être considéré dans le coût de la taille du message.❑

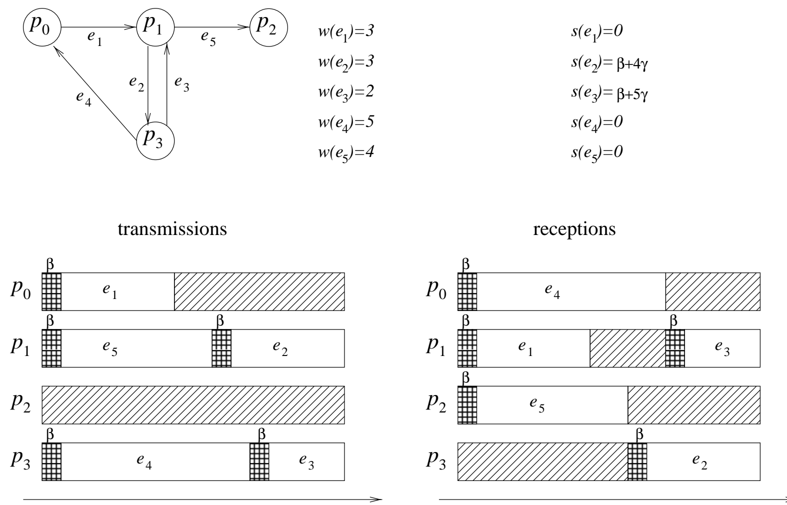

FIG. 6.3 – Ordonnancement avec des messages indivisibles.

La figure 6.3 illustre un exemple d'ordonnancement et son schéma d'exécution correspondant. Les poids des arêtes sont des entiers. Pour obtenir un problème de transfert de fichiers équivalent, il faut considérer comme coût d'arête $w(e_i) + \beta$. Une différence par rapport à l'ordonnancement de tâches est que chaque communication apparaît sur deux processeurs distincts, un pour la transmission et l'autre pour la réception. Dans la figure, la fonction s fournie le début de la communication. Le coté gauche du schéma peut être vu comme la transmission et le droit comme la réception.

Avec la propriété 6.5 nous pouvons énoncer deux corollaires qui sont des conséquences directes des théorèmes de [20].

Corollaire 6.1 *Un ordonnancement optimal pour le MEP, quand tous les arcs ont le même poids, peut être trouvé en temps polynômial.*

Corollaire 6.2 *La recherche de l'ordonnancement optimal pour le MEP est un problème $\mathcal{NP}-$difficile.*

Messages divisibles

Choi et Hakimi [17] ont analysé le problème de l'échange de messages dans un réseau totalement connecté synchrone. Ils n'ont pas étudié la commutation de messages, cependant ils ont autorisé la division de messages. Pour présenter un ordonnancement quand la division de messages est autorisée, il faut fournir en plus des dates de début de chaque communication la durée des temps de transmission. À chaque instant de l'ordonnancement, les communications actives forment un sous graphe où la capacité des portes n'est pas dépassée.

La figure 6.4 illustre un exemple d'ordonnancement, Nous avons utilisé la représentation du MEP par un graphe orienté.

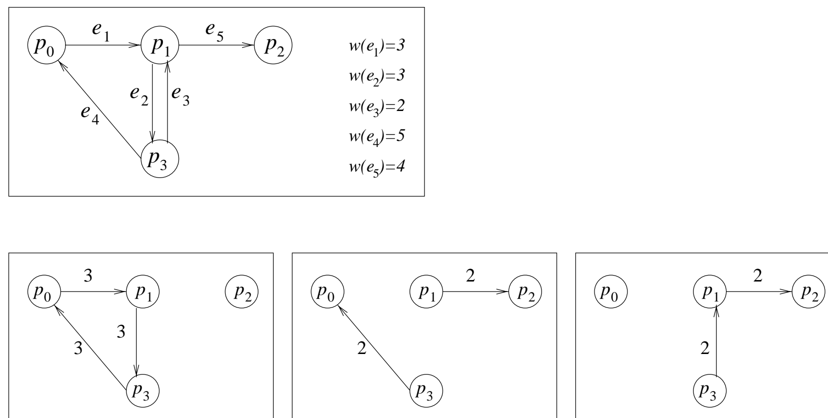

FIG. 6.4 – Ordonnancement avec messages divisibles.

L'ordonnancement de la figure 6.4 dépend du modèle. S'il est synchrone un schéma d'exécution est donné dans la figure 6.5 à gauche. Dans ce cas, chaque envoi d'un même message, consécutif ou non, implique un coût d'initialisation. Lorsque le synchronisme n'est pas imposé, le schéma d'exécution est illustré dans la figure 6.5 à droite. Dans ce cas, le coût d'initialisation n'existe que pour les envois non consécutifs d'un message.

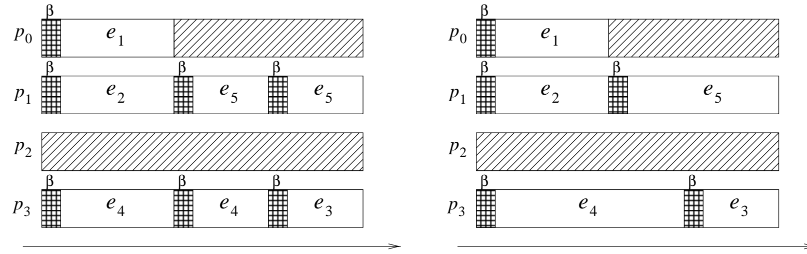

FIG. 6.5 – Schémas d'exécution de messages divisibles.

Choi et Hakimi [17] ont démontré les théorèmes suivants :

Théorème 6.1 *Quand le coût d'initialisation est nul ($\beta = 0$) un ordonnancement optimal du MEP peut être trouvé en temps polynômial.*

Théorème 6.2 *Lorsque $\beta > 0$, le problème de décision de l'existence d'un ordonnancement pour le MEP synchrone avec makespan au plus M est un problème $\mathcal{NP}-$complet.*

h-relation

Un autre problème qui ressemble au MEP est la h-relation (voir section 2.4.5). Plusieurs auteurs ont proposé des algorithmes pour la h-relation.

Wang et Ranka [102, 103] ont implanté deux méthodes. L'une d'entre elles utilise un motif fixé fondé sur des permutations linéaires avec des échanges entre paires de processeurs. L'autre approche est un ordonnancement aléatoire capable d'éviter la congestion. Bader, Helman et JáJá [6] ont proposé un algorithme en deux phases avec des performances meilleures que Ranka et Wang. Leur méthode consiste en deux phases d'échange total personnalisé avec des blocs de taille égale. Nous avons comparé les algorithmes que nous allons présenter avec les algorithmes de [6], les résultats se trouvent dans [50].

6.3 Notre contribution

Nous nous sommes intéressés au comportement du MEP dans des machines réelles, lesquelles sont aujourd'hui, pour la plupart, asynchrones. Nous proposons d'abord une généralisation du théorème 6.2 lorsque le fonctionnement n'est pas synchrone.

Théorème 6.3 *Quand le temps d'initialisation n'est pas négligeable, $\beta > 0$, décider dans un modèle asynchrone s'il existe un ordonnancement d'un MEP $dG(V', E')$ avec makespan au plus M est un problème $\mathcal{NP}-$complet.*

La preuve est une réduction du problème 3-partition [39], elle se trouve en détail dans la section E.1 de l'annexe E, car elle est longue et très technique.

Nous présentons des algorithmes d'approximation pour ce problème. Un compromis similaire à celui de l'ordonnancement de tâches apparaît. Nous voulons trouver un bon compromis entre le nombre d'initialisations et l'usage de la bande passante. Nous proposons des algorithmes avec le nombre minimal de pas d'initialisation ainsi que des algorithmes où l'usage de la bande passante est le meilleur. Nous fournissons aussi des algorithmes où il existe un compromis entre les deux. Dans tous les algorithmes existent des phases de communication, qui ne sont pas forcément synchrones. Ces algorithmes ont déjà été proposés dans la littérature pour résoudre des problèmes différents. Nous proposons ensuite de nouveaux algorithmes combinés.

Dans ce chapitre notre but est de fournir une estimation assez précise des heuristiques. Il nous faut donc prendre en compte des aspects du support d'exécution. Pour des supports fondés sur la technologie de l'échange de messages, lors de la transmission d'un message, il faut que le récepteur connaisse la taille, ou au moins une estimations de la taille du message. Ce besoin est lié aux allocations des tampons mémoire. Donc, pour les algorithmes que nous examinons, il faut prévoir un pré-traitement dans lequel les processeurs vont recevoir les tailles des messages qui leurs sont destinés.

Lors de la présentation des algorithmes nous avons : n le nombre de processeurs, $\{p_0, \ldots, p_{n-1}\}$ les processeurs et $m_{i,j}$ le message du processeur p_i au processeur p_j. Nous dénotons aussi le plus grand et le plus petit message respectivement par M et m.

6.3.1 Algorithme avec commutation de messages

Ce premier algorithme a été présenté d'abord pour l'échange de messages dans un hypercube, nous le dénommons donc par HL (de l'anglais *hypercube-like*). Il atteint la borne inférieure du nombre de pas d'initialisation. Nous présentons l'algorithme pour un nombre de processeurs puissance de 2 ($n = 2^p$). Nous introduisons à partir de la représentation binaire de p_i, $b_i = (b_{i_1}, \ldots, b_{i_p})$, $p_i^{\bar{j}}$ le processeur dont la représentation binaire est $(b_{i_1}, \ldots, \bar{b}_{i_j}, \ldots, b_{i_p})$.

Algorithm 6.1 Algorithme HL.

`pour` $t = 1$ `jusqu'à` p `faire`

  `pour tout` i $(0 \leq i < n)$ `faire`

    échanger tous les messages entre p_i et $p_i^{\bar{t}}$;

  `fin pour`

`fin pour`

Analyse

Pour perfectionner l'algorithme, les échanges de messages peuvent être réalisés d'une façon limitée. Au lieu d'échanger tous les messages, seuls ceux qui doivent être commutés sont envoyés. Ainsi à chaque étape il n'existe qu'une copie de chaque message $m_{i,j}$. Dans le but d'implanter le mécanisme de contrôle, il suffit d'observer que pour chaque message, il existe un chemin unique à sa destination.

L'algorithme a $\log_2 n$ phases, dans la phase t $(t = 1, \ldots, \log_2 n)$, chaque processeur échange $\frac{n}{2^i}$ de ses messages, et $\frac{n}{2} - \frac{n}{2^i}$ des messages reçus auparavant avec un nouveau voisin. À chaque phase t le temps nécessaire pour compléter l'échange est borné par $\beta + \frac{n}{2}M\gamma$. Donc, le temps total de l'algorithme est au plus $\log_2 n.\beta + \log_2 n.\frac{n}{2}M\gamma$.

Nous illustrons dans la figure 6.6 les schémas d'exécution du problème MEP donné par la matrice 6.1. Cette matrice a été générée à partir d'une distribution normale avec moyenne 20000 et écart type 10000. Nous présentons les schémas d'exécution de plusieurs algorithmes sur une machine sans contention avec 8 processeurs. Le temps d'initialisation est 0.5 millisecondes et l'inverse de la bande passante est 0.00035 millisecondes. Dans toutes les schémas d'exécution l'échelle de l'axe x est donné en millisecondes.

$$\begin{pmatrix} 0 & 27304 & 18848 & 23943 & 7795 & 31195 & 18757 & 42838 \\ 27399 & 0 & 8229 & 9859 & 24571 & 44168 & 38956 & 38309 \\ 10430 & 11806 & 0 & 15320 & 15126 & 20844 & 27330 & 22510 \\ 27251 & 20086 & 7153 & 0 & 24918 & 30176 & 18139 & 14030 \\ 24814 & 3246 & 3323 & 30984 & 0 & 22312 & 0 & 0 \\ 0 & 45242 & 15672 & 19776 & 9556 & 0 & 17499 & 19896 \\ 25376 & 23198 & 34487 & 20776 & 4792 & 15060 & 0 & 14228 \\ 26357 & 25295 & 18615 & 12747 & 27613 & 25310 & 28632 & 0 \end{pmatrix} \quad (6.1)$$

Dans les schémas d'exécution, les transmissions des processeurs sont illustrées, chacune comprend le temps d'initialisation plus l'usage de la bande passante. Pour aider la visualisation des phases de communication, les rectangles sont dessinés différemment selon la phase. Dans les schémas de la figure 6.6 nous observons facilement les trois ($\log_2 8$) phases de communication. La différence entre l'exécution synchrone (à droite) et asynchrone (à gauche) est visible lors du début de chaque phase d'échange de messages.

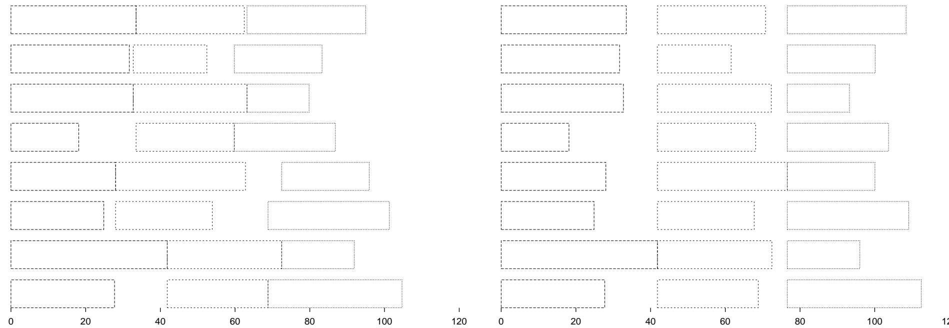

FIG. 6.6 – Schémas d'exécution pour l'algorithme HL.

Nous avons aussi proposé une généralisation de l'algorithme du HL pour un nombre pair de processeurs. L'algorithme et son analyse complète sont dans la section E.2 de l'annexe E.

Le plus grand désavantage de l'algorithme HL est la retransmission des messages. Chaque message $m_{i,j}$ possède une destination unique. Seulement $(n-1) - log_2 n$ messages de chaque processeur ne sont pas commutés. Par exemple, il existe n messages qui sont commutés $\log_2 n - 1$ fois, ce qui donne un surcoût de $\log_2 n - 1$ retransmissions. Toutes les retransmissions de messages engendrent un surcoût. Ce surcoût peut être ignoré dans le cas de grand temps d'initialisation avec des messages petits. D'un autre coté, avec des messages grands le surcoût sera le paramètre prédominant. Si la commutation de messages n'est pas autorisée, un algorithme qui résout le problème possède plus de $n-2$ phases.

6.4 Algorithme à motif fixé

Nous présentons un algorithme de base qui n'utilise ni la commutation ni la division de messages. Cet algorithme, comme le précédent, ne profite pas

des informations sur la taille des messages.

Algorithm 6.2 Algorithme à motif fixé.

```
pour t = 1 jusqu'à n − 1 faire
  pour tout i (0 ≤ i < n) faire
    p_i envoie le message m_{i,i+t mod n} à p_{i+t mod n};
    p_i reçoit le message m_{i−t mod n,i} de p_{i−t mod n};
  fin pour
fin pour
```

Dans l'algorithme 6.2, chaque processeur exécute $n-1$ phases. Le temps nécessaire, dans un mode synchrone, pour achever chaque phase t est $\beta + \max_i\{m_{i,i+t \bmod n}\}\gamma$. Donc le temps total d'exécution est donné par $(n-1)\beta + \sum_{t=1}^{n-1} \max_i\{m_{i,i+t \bmod n}\}\gamma \leq (n-1)\beta + (n-1)M\gamma$. L'analyse du cas asynchrone est plus compliqué, il faut considérer les temps d'inactivité (c'est-à-dire sans communication) dûs à la différence de taille entre les messages.

Nous examinons un exemple où l'algorithme à motif fixé construit un ordonnancement avec de grandes périodes d'inactivité dans le cas synchrone ainsi que dans le mode asynchrone. Dans l'exemple il existe deux tailles de messages, une petite m et une grande M. La matrice donnée ci-dessous possède 7 messages grands et 49 messages petits.

$$\begin{pmatrix} 0 & M & m & m & m & m & m & m \\ m & 0 & m & m & m & m & m & m \\ M & m & 0 & m & m & m & m & m \\ m & m & m & 0 & m & m & m & M \\ m & m & m & m & 0 & m & M & m \\ m & m & m & m & M & 0 & m & m \\ m & m & m & M & m & m & 0 & m \\ m & m & M & m & m & m & m & 0 \end{pmatrix}$$

L'ordonnancement généré par l'algorithme à motif fixé dans le cas synchrone transmet un grand message à chaque phase. Il a un temps d'exécution $(n-1)\beta + (n-1)M\gamma$. Son schéma d'exécution est illustré dans la figure 6.7.

Lorsque le fonctionnement synchrone n'est plus imposé, l'algorithme à motif fixé fourni un algorithme avec *makespan* plus petit. Cependant, la

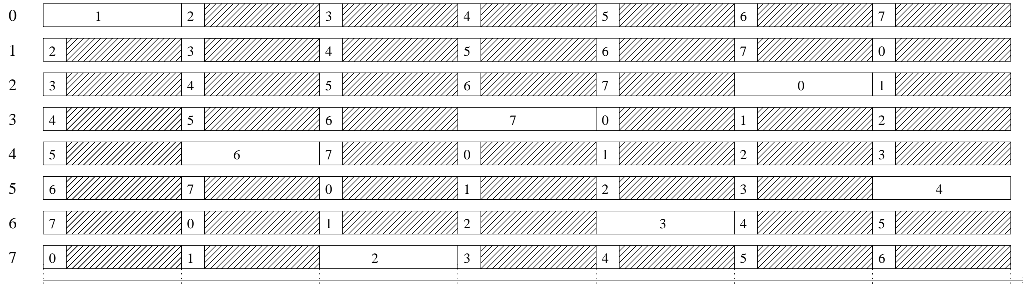

FIG. 6.7 – Ordonnancement à motif fixé dans le modèle synchrone. À l'intérieur des rectangles le processeurs destin est indiqué.

bande passante est encore largement sous utilisée. Le cas au pire correspond au cas où il y a deux chaînes de grands messages retardés. Dans ce cas le *makespan* devient $(n-1)\beta + (\frac{n-2}{2}(M+m)+M)\gamma$ pour n pair et $(n-1)\beta + (M+S)\frac{n-1}{2}\gamma$ pour n impair. La figure 6.8 montre un ordonnancement où les chaînes de communications retardées $(1,2,3,4)$ et $(6,7,0)$ sont visibles.

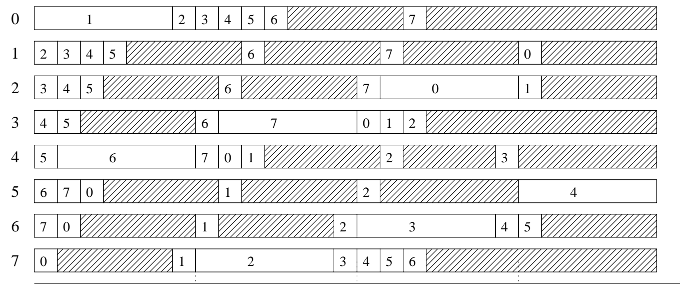

FIG. 6.8 – Ordonnancement à motif fixé dans le modèle asynchrone.

D'un autre coté il existe un ordonnancement qui atteint la borne inférieure dans la bande passante $((n-2)m+M)\gamma$. L'ordonnancement qui atteint cette borne envoie tous les messages grands dans une même phase. Nous présentons dans la section suivante un algorithme avec de meilleures garanties de performance. L'algorithme à motif fixé produit un ordonnancement optimal lorsque tous les messages ont des tailles égales (voir corollaire 6.1).

Nous illustrons dans la figure 6.9 les schémas d'exécution du problème MEP donnés par la matrice 6.1 pour l'algorithme à motif fixé. Le schéma asynchrone est à gauche et le synchrone à droite.

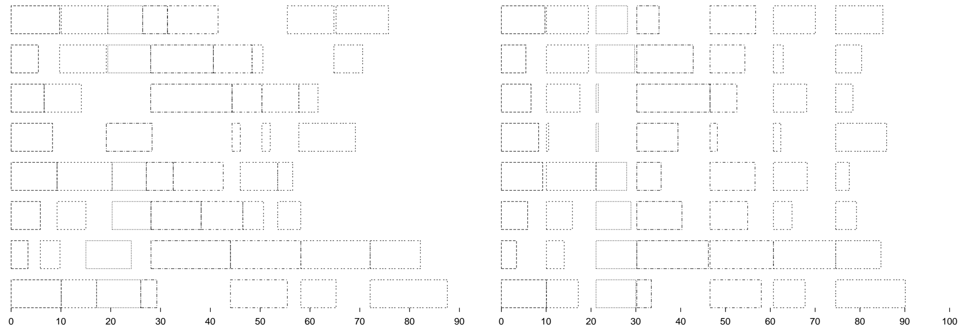

FIG. 6.9 – Schémas d'exécution pour l'algorithme à motif fixé.

6.4.1 Algorithmes avec connaissance globale

Nous présentons trois algorithmes qui profitent de la connaissance globale du problème. L'idée de fond est d'utiliser cette information pour équilibrer les messages transmis dans chaque phase. Les deux premiers algorithmes ne font pas de division de messages. Le principe des algorithmes est :

Algorithm 6.3 Principe de l'algorithme avec connaissance globale.

faire un échange des tailles de messages ;
trouver un ordonnancement ;
transmettre les messages selon l'ordonnancement ;

L'algorithme 6.3 consiste en trois étapes. La première étape fournie la connaissance globale du problème à tous les processeurs. Dans un langage parallèle fondé sur des primitives d'envoi et réception, l'obtention de la connaissance globale peut être vue comme une partie du pré-traitement. Au lieu de n'échanger que les tailles des messages à échanger, c'est-à-dire chaque processeur j reçoit les tailles de messages $m_{i,j}$, toutes les tailles de messages sont échangées. Soit s, le nombre de mots nécessaires pour stocker la taille d'un message. Le coût de pré-traitement standard est de $(n-1)\beta+(n-1)s\gamma$ (pour l'algorithme à motif fixé) ou de $\log_2 n.\beta+\log_2 n.\frac{n}{2}s\gamma$ (pour l'algorithme HL). Donc, le surcoût dû à l'échange de $n-1$ tailles de messages par processeur est $(n-1)(n-2)s\gamma$ ou $\lceil\log_2 n\rceil\frac{n}{2}(n-2)s\gamma$, respectivement avec l'algorithme HL ou l'algorithme à motif fixé.

Les étapes deux et trois peuvent être superposées. Pendant que le calcul de la deuxième étape est en cours, la communication de la troisième étape peut

commencer. Cette superposition n'affecte pas le synchronisme car durant une grande partie du temps de communication les processeurs sont libres pour le calcul.

Dans les descriptions des algorithmes, nous déterminons une séquence de sous graphes où chaque sommet est l'origine d'au plus 1 arc et est la destination d'au plus 1 arc aussi. À partir de la représentation du problème par un graphe biparti, ces sous graphes correspondent à des couplages. Dans la représentation par matrice, ils correspondent à des matrices de permutation avec poids. Nous avons choisi la représentation par matrice pour les implantations.

Pour chaque algorithme nous déterminons une couplage M_1 du graphe $bG(V.E)$. Les poids des arêtes en M_1 est soustrait de $bG(V,E)$, opération qui enlève aussi des arêtes avec poids 0. Après, nous cherchons un deuxième couplage dans le graphe résultant, et ainsi de suite. L'algorithme s'arrête lorsque toutes les communications sont réalisées par la séquence de couplages.

La technique pour rechercher des couplages précise les différences entre les algorithmes. Toutes ces techniques sont bien connues en algèbre linéaire [70]. Les algorithmes complets sont donnés dans la section E.3 de l'annexe E.

stratégie Max-Min : À chaque étape nous recherchons un couplage maximal par rapport au poids de la plus petite arête.

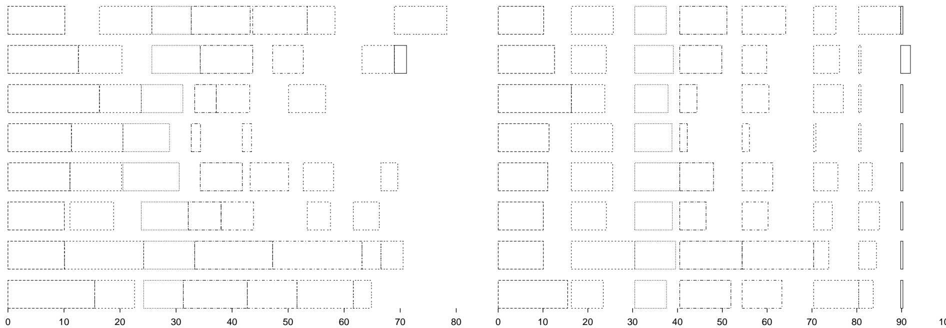

FIG. 6.10 – Schémas d'exécution pour l'algorithme Max-Min.

stratégie Max-Somme : À chaque étape nous recherchons un couplage maximal par rapport à la somme des poids des arêtes.

Nous illustrons dans les figures 6.10 et 6.11 les schémas d'exécution du problème MEP donné par la matrice 6.1 pour les algorithmes Max-Min et Max-Somme.

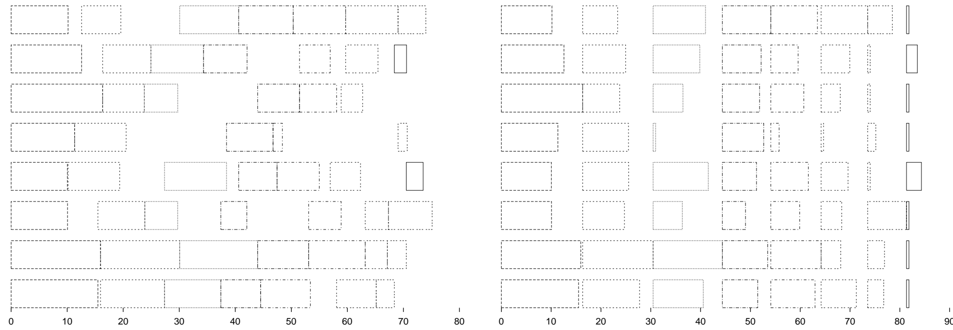

FIG. 6.11 – Schémas d'exécution pour l'algorithme Max-Somme.

Lorsqu'elles sont utilisées, ces deux stratégies permettent un échange de messages de tailles plus uniformes à chaque phase de communication. La complexité des algorithmes Max-Min et Max-Somme est $O(n^3)$ [70].

stratégie uniforme : À chaque étape nous recherchons un couplage où les poids de toutes les arêtes sont identiques. Une étape de pré-traitement locale est nécessaire. Étant donné le graphe $dG(V, E)$, nous utilisons la représentation par matrice $A = (m_{i,j})$. À partir du problème original, nous ajoutons des messages fantômes, de façon à obtenir une matrice quasi doublement stochastique Q (de l'anglais *quasi-doubly stochastic*). Dans cette matrice toutes les lignes et colonnes ont des sommes égales.

Dans la matrice A, r_i dénote la somme des tailles de la ligne i et c_j la somme des tailles de la colonne j. Soit $\sigma = \max_{i=1}^{n}\{r_i, c_i\}$. Nous utilisons l'algorithme 6.4 pour obtenir Q.

À la fin de l'algorithme la somme des tailles de messages de chaque lignes ou colonnes de Q sera σ. Il est intéressant d'observer que la borne inférieure de temps de communication de la matrice Q est la même que pour la matrice originale A.

À partir de la représentation par graphe biparti $bQ(M, E)$ de Q, nous recherchons des couplages comme dans la stratégie Max-Min. Le trafic généré par des messages fantômes ne doit pas être effectué lors de la phase de communication.

Le nombre des phases de communication de cet algorithme est presque toujours plus grand que $n-1$ (en général $O(n^2)$). Cet algorithme génère un ordonnancement optimal quand le temps d'initialisation n'est pas considéré (voir théorème 6.1).

Algorithm 6.4 Pré-traitement pour l'algorithme uniforme.

$Q \leftarrow A$
pour $i = 1$ jusqu'à n faire
  pour $j = 1$ jusqu'à n faire
    si $i \neq j$ alors
      $q_{i,j} \leftarrow q_{i,j} + \min\{\sigma - r_i, \sigma - c_j\}$
    fin si
  fin pour
fin pour
pour $i = 1$ jusqu'à n faire
  $q_{i,i} \leftarrow q_{i,i} + \min\{\sigma - r_i, \sigma - c_i\}$
fin pour

Pour tous les algorithmes avec connaissance globale, lorsque les messages sont petits, les deux premières phases peuvent prendre autant de temps que la phase de communication.

Nous illustrons dans la figure 6.12 les schémas d'exécution du problème MEP donné par la matrice 6.1 pour l'algorithme uniforme. Il est clair que les dernières phases de communication sont constituées d'échanges de petits messages. Dans la figure 6.12 les phases à partir de l'instant $70ms$ ont la même ordre de temps que le temps d'initialisation β.

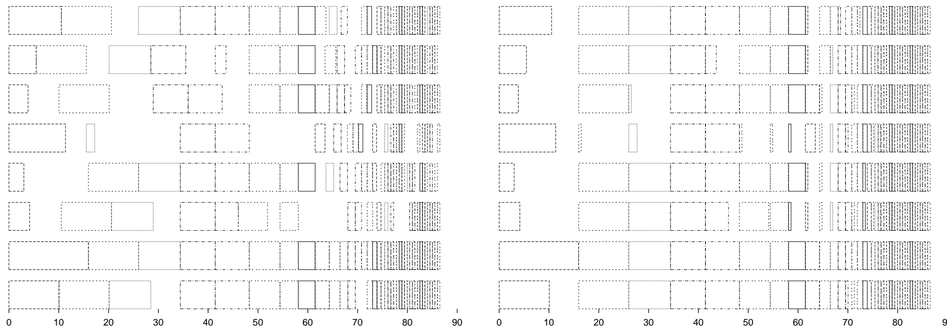

FIG. 6.12 – Schémas d'exécution pour l'algorithme uniforme.

Dorénavant nous faisons référence aux algorithmes vus comme : HL, fixé, Max-Min, Max-Somme et uniforme.

6.4.2 Étude qualitative

Les algorithmes qui ont été proposés possèdent des caractéristiques distinctes. Selon le MEP, l'un d'entre eux peut s'avérer plus adapté, construisant donc un ordonnancement meilleur par rapport aux autres. Le choix de l'algorithme qui construit l'ordonnancement avec le plus petit *makespan* dépend des paramètres du modèle linéaire (β et α) et de la moyenne et de la variance des messages (que nous allons dénoter par A et v). Pour le calcul de la moyenne, lorsqu'il n'y a pas de message d'un processeur à un autre, un message de taille 0 est considéré.

Deux constatations sont immédiates, l'algorithme HL minimise le nombre de pas d'initialisation et l'algorithme uniforme minimise la perte de bande passante. Le *makespan* de l'ordonnancement créé par l'algorithme HL est $O(\log_2 n(\beta + A\frac{n}{2}\gamma))$, et par l'algorithme uniforme est $O(n^2\beta + nA\gamma)$.

Quand le temps d'initialisation domine le temps de transmission, c'est-à-dire $\beta > A\frac{n}{2}\gamma$, le *makespan* de l'ordonnancement construit par l'algorithme HL est $O(\log_2 n.\beta)$. Pour les autres algorithmes le temps est $O(n\beta)$, sauf pour l'algorithme uniforme qui a un temps $O(n^2\beta)$. D'un autre coté, si $\beta < \frac{A\gamma}{n}$ alors il est intéressant de minimiser la perte de bande passante, donc l'algorithme uniforme construit les meilleurs ordonnancements avec *makespan* $O(nA\gamma)$.

Dans les cas intermédiaires il existe un compromis entre les valeurs de β, γ et A. Le choix de l'algorithme dépend de la variance de la taille des messages. Si la variance est petite par rapport au temps nécessaire pour fournir la connaissance globale, alors l'utilisation d'un algorithme plus raffiné n'apporte pas d'amélioration. Nous dénommons ce temps par t_g. Le calcul du temps t_g relève de l'algorithme utilisé ainsi que du support à la communication. Par exemple, si une bibliothèque de passage de messages est utilisée, il faut de toute façon prévoir une phase d'échange des tailles de messages, par conséquent la valeur de t_g est presque négligeable.

Nous avons examiné que pour l'algorithme à motif fixé, dans le cas au pire, le *makespan* peut être borné par un facteur additif de $(m + M)\frac{n-2}{2}\gamma$, lequel est plus grand que $\sqrt{v}(n-2)\gamma$. Donc si $\sqrt{v}(n-2)\gamma \leq t_g$, l'algorithme à motif fixé construit un ordonnancement avec temps total $O(\sqrt{v}(n-2)\gamma \leq t_g + n\beta)$. Sinon le surcoût t_g peut être compensé par un meilleur ordre d'échange des messages. Nous n'avons pas considéré les temps de calcul pour les couplages,

car ce temps peut être recouvert par les temps de communication.

Il n'est pas facile d'analyser la différence entre le temps des algorithmes Max-Somme et Max-Min, car elle dépend aussi de la distribution des messages. Nous avons choisi d'examiner ces différences avec des expérimentations et des simulations.

6.4.3 Un algorithme combiné

Lors de nos expériences et simulations nous avons remarqué que l'algorithme uniforme est le meilleur au cours des premières phases de communication, quand un grand nombre de messages est échangé. Néanmoins, les phases finales sont presque toujours des échanges de petits messages. Dans le but d'améliorer les performances nous utilisons l'algorithme uniforme pour les premières phases et ensuite nous utilisons un autre algorithme. Nous avons trois choix : l'algorithme HL, l'algorithme Max-Min et l'algorithme Max-Somme. Nous ne choisissons pas l'algorithme à motif fixé car la connaissance globale des tailles de message est disponible.

Algorithm 6.5 Principe des algorithmes combinés

faire l'échange des tailles de message;

appliquer les k premières phases de l'algorithme uniforme;

appliquer un algorithme avec un nombre donné de phases pour les messages restants;

Dans l'algorithme 6.5 il faut répondre à deux questions principales : la valeur de k et le choix de l'algorithme. Nous présentons l'analyse de l'algorithme combiné avec les algorithmes Max-Min et HL. Nous ne présentons pas l'analyse pour l'algorithme Max-Somme car il possède des caractéristiques similaires à l'algorithme Max-Min.

Nous dénotons par $A = (m_{i,j})$ la matrice de communication originale et par $Q = (q_{i,j})$ le pré-traitement correspondant à l'algorithme uniforme. Pour chaque phase p, $1 \leq p \leq k$, de l'algorithme uniforme nous notons par s_p la taille des messages échanges à la phase p, et $S_p = (s^p_{i,j})$ le couplage de la phase p. Il est immédiat que S_p est le produit du numéro s_p par une matrice de permutation. Les k phases de l'algorithme uniforme ont pour *makespan* $k\beta + \sum_{p=1}^{k} s_p\gamma$. Après ces phases, la matrice de communication devient $A' = (m'_{i,j})$, où $m'_{i,j} = \max\{0, m_{i,k} - \sum_{p=1}^{k} s^p_{i,j}\}$.

Si l'algorithme HL est alors appliqué, nous aurons au plus $\lceil \log_2 \rceil$ phases et la communication sera bornée par $\max\{m'_{i,j}\} \log_2 n.\frac{n}{2}\gamma$. Sinon, lorsque l'algorithme Max-Min est appliqué, nous aurons au plus $n-1$ phases additionnelles et la communication sera bornée par $\max\{m'_{i,j}\}n\gamma$. Les *makespans* fournis par les algorithmes combinés sont bornés par :

$$(k + \lceil \log n \rceil)\beta + \left(\sum_{p=1}^{k} s_p + \max(m'_{i,j}) \log_2 n.\frac{n}{2} \right) \gamma$$

pour l'algorithme combiné HL, et

$$(k + n - 1)\beta + \left(\sum_{p=1}^{k} s_p + \max(m'_{i,j})n \right) \gamma.$$

pour l'algorithme combiné Max-Min. Dans les deux équations il existe un facteur qui augmente avec k et l'autre qui diminue avec k. Selon l'algorithme combiné choisi il existe une valeur entière positive qui minimise le *makespan*. Nous dénotons par k^* la valeur de k qui fourni le plus petit makespan pour un algorithme donné.

Le choix de l'algorithme dépend principalement du temps d'initialisation et de la bande passante. Évidemment si β est très important, l'algorithme combiné HL avec $k^* = 0$ sera le meilleur. D'un autre coté si β est négligeable l'algorithme combiné Max-Min avec k^* grand, c'est-à-dire peu de phases de l'algorithme Max-Min, aura les meilleures performances. La détermination analytique de k^* est impossible, donc nous allons étudier les performances des algorithmes combinés à travers de simulations.

6.5 Résultats expérimentaux

La partie expérimentale de ce chapitre se divise en deux. Une partie avec des implantations dans une machine parallèle et une autre avec des simulations.

6.5.1 Implantation

Nous avons implanté les algorithmes avec $n - 1$ phases de communication dans un ordinateur parallèle, le IBM SP2. Notre but principal a été la

validation du modèle utilisé. L'implantation a été faite avec la libraire MPI F fourni par IBM. Nous avons comparé les algorithmes avec l'algorithme standard `MPI_Alltoallv` fourni avec la librairie.

Les instances du MEP choisies pour les mesures de temps ont été générées à partir de matrices carrées. Les tailles de messages ont été tirées au hasard. Les probabilités ont été fournies selon une distribution normale avec moyenne et écart type donnés.

Nous présentons les résultats obtenus avec 30 instances distinctes du MEP. Pour l'estimation du temps de chaque instance, nous avons fait la moyenne de 20 mesures. La variance de temps de mesures a été plus petit que 0.01%. Nous allons présenter les résultats obtenus pour des instances générées à partir d'une moyenne de 800k octets et écart type de 400k octets.

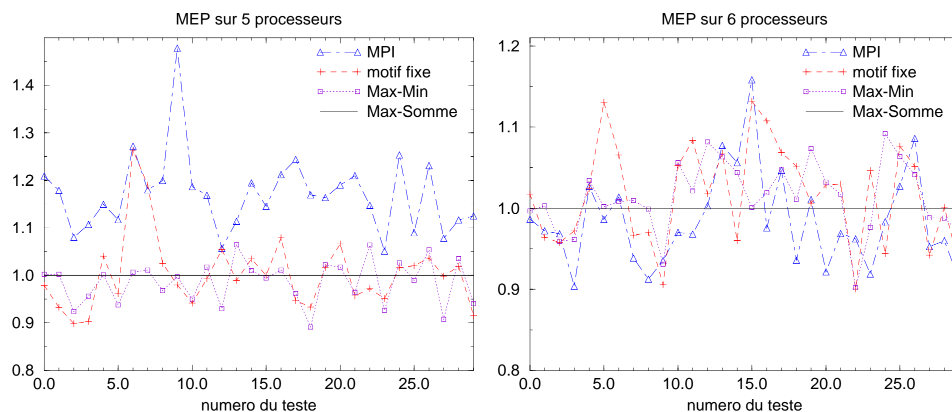

FIG. 6.13 – Rapport de performance relative à l'algorithme Max-Somme sur 5 et 6 processeurs.

Les figures 6.13 et 6.14 sont présentées par rapport à la performance de l'algorithme Max-Somme, c'est-à-dire pour chaque MEP, nous divisons les temps des algorithmes par le temps de l'algorithme Max-Somme. De cette facçn nous pouvons voir facilement la performance des algorithmes. Dans la figure 6.13 à gauche, les performances sur 5 processeurs sont présentées. L'algorithme le moins performant est celui fourni avec la librairie MPI F. À droite, dans l'exécution sur 6 processeurs, les performances des algorithmes sont similaires.

Lorsque le nombre de processeurs est plus grand (voir figure 6.14), nous observons les meilleures performances pour l'algorithme Max-Somme.

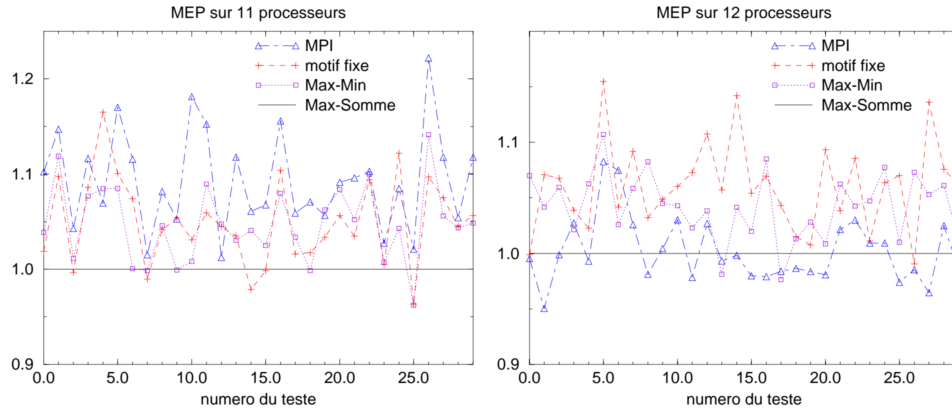

FIG. 6.14 – Rapport de performance relative à l'algorithme Max-Somme sur 11 et 12 processeurs.

Une idée générale de la performance des algorithmes est donnée dans la table 6.1. Chaque moyenne présentée correspond au calcul de la moyenne pour les 30 instances différentes avec un nombre donné de processeurs. Nous fournissons aussi l'écart type entre parenthèses. Les temps sont donnés en millisecondes. Le meilleur temps obtenu pour un nombre donné de processeurs est indiqué en gras. L'analyse des résultats de la table 6.1 nous montre que malgré les variations de performances selon l'instance, le calcul de la moyenne s'avère représentatif car l'écart type n'est pas très important. Nous observons aussi le bon comportement de l'algorithme Max-Somme qui a toujours de très bonnes performances.

Les temps obtenus avec l'algorithme Max-Somme sont de l'ordre de 5% meilleurs que les temps de la fonction `MPI_Alltoallv`. Cette différence est plus visible avec un nombre impair des processeurs. Nous trouvons ce résultat très motivant car nos algorithmes ont été implantés avec des primitives `MPI_send` and `MPI_recv`, tandis que la fonction `MPI_Alltoallv` a dû être implanté avec des primitives de plus bas niveau [1].

6.5.2 Simulation

Nous allons étudier à travers des simulations les performances des algorithmes combinés. Dans un premier temps, nous confirmons que la distribu-

[1] Le code source de `MPI_Alltoallv` n'est pas disponible.

Nombre de process.	MPI	motif fixé	Max-Min	Max-Somme
3	159 (30)	**126** (24)	127 (24)	128 (24)
4	**195** (27)	204 (24)	206 (36)	204 (29)
5	324 (34)	278 (33)	**274** (32)	278 (32)
6	**345** (42)	356 (39)	355 (33)	351 (33)
7	474 (37)	442 (34)	437 (30)	**426** (31)
8	494 (35)	511 (32)	501 (33)	**490** (37)
9	615 (32)	593 (26)	582 (32)	**561** (33)
10	643 (38)	674 (34)	**653** (42)	659 (39)
11	769 (44)	738 (30)	737 (40)	**704** (35)
12	783 (42)	829 (45)	815 (39)	**782** (36)
13	915 (47)	914 (41)	903 (44)	**860** (44)
14	**937** (41)	990 (53)	996 (41)	942 (53)

TAB. 6.1 – Résumé des résultats expérimentaux.

tion des messages ne joue pas un rôle très important dans le makespan des algorithmes. Ensuite nous comparons les algorithmes combinés entre eux et avec les algorithmes simples. Les comparaisons sont faites avec des instances qui caractérisent des particularités du MEP.

cas	β	$1/\gamma$	moyenne	écart type	n. de proc.	machine
1	1.3 ms	3.5e-05	100000	100000	12	IBM-SP2
2	0.8 ms	2.8e-04	100000	200000	8	réseau de Pcs
3	8.6 μs	3.3e-6	64000	32000	16	Cray T3D

TAB. 6.2 – Cas de l'analyse de la variance. $1/\gamma$ est donné en mots/seconde.

Dans la table 6.3 sont illustrés les valeurs de la moyenne et de l'écart type des algorithmes. Pour les algorithmes combinés, nous présentons la moyenne des meilleurs résultats. Des paramètres de trois machines réelles sont montrés. Pour chaque cas de la table 6.2 nous calculons la moyenne et l'écart type de l'exécution de 20 MEPs. Dans la table 6.3 les temps sont fournis en millisecondes, l'écart type est présenté entre parenthèses.

La variation provoquée par la distribution de messages dans un exemple de MEP donné s'avère n'être pas très importante. À partir de cette constatation nous pouvons espérer que la dépendance du temps d'exécution pour des exemples différentes, générés de la même courbe normale, ne sera pas très

cas	fixé	Max-Min	Max-Somme	Uniforme
1	90.8 (6.2)	91.9 (7.6)	87.4 (5.7)	152.2 (7.4)
2	584 (111)	564 (99)	533 (97)	512 (97)
3	4.85 (0.17)	4.67 (0.19)	4.54 (0.23)	4.8 (0.2)
	HL	Comb. Max-Min	Comb. Max-Somme	Comb. HL
1	132.6 (13.5)	87.9 (5.1)	85.5 (6.1)	96.9 (7.0)
2	739 (132)	500 (95)	499 (95)	506 (97)
3	8.22 (0.36)	4.40 (0.20)	4.39 (0.19)	4.42 (0.20)

TAB. 6.3 – Moyenne (en millisecondes) et variances pour plusieurs cas.

importante. Nous analysons donc les temps des algorithmes combinés dans le cas d'un temps d'initialisation important, le cas de grands messages et dans deux cas intermédiaires. Les courbes illustrent les temps en millisecondes par rapport au nombre de phases de l'algorithme uniforme (k).

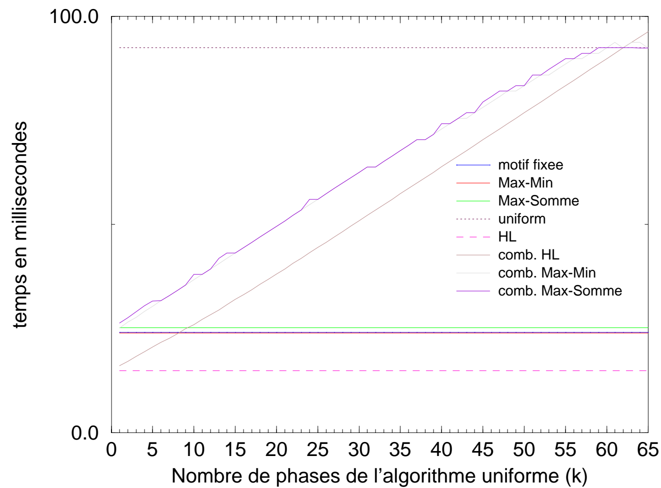

FIG. 6.15 – Performance des algorithmes quand le temps d'initialisation est important.

Nous commençons avec la simulation du MEP sur un IBM SP2 avec 14 processeurs. Les valeurs estimées pour de petits messages sont $\beta = 1.3ms$ et $\gamma = 0.035\mu s$. Nous étudions le cas où la moyenne des messages est $A = 5k$

octets et l'écart type est $\sqrt{v} = 2k$ octets. Avec ces valeurs, l'algorithme HL fourni les meilleurs *makespans*. La figure 6.15 illustre les performances des algorithmes selon la variation de k. Les performances des algorithmes qui ne sont pas combinés sont représentées par des lignes horizontales. Les performances des algorithmes combinés sont montrées pour les valeurs de k plus grandes que zéro, c'est-à-dire quand au moins une phase de l'algorithme uniforme est exécutée.

Nous vérifions dans la figure 6.15 que l'algorithme HL est le plus rapide. Nous observons aussi que le nombre de phases optimal pour les algorithmes combinés est k^* égale à zéro. Nous présentons dans la table ci-dessous les temps d'exécution pour les algorithmes uniformes et combinés. Pour les algorithmes combinés nous présentons aussi la valeur de k^*. Toutes les valeurs sont exprimées en millisecondes.

HL	uniforme	Max-Somme	Max-Min
14.9	92.4	25.2	23.9
motif fixé	comb. HL	comb. Max-Min	comb. Max-Somme
24.1	14.9 (k^* =0)	23.9 (k^* =0)	25.2 (k^* =0)

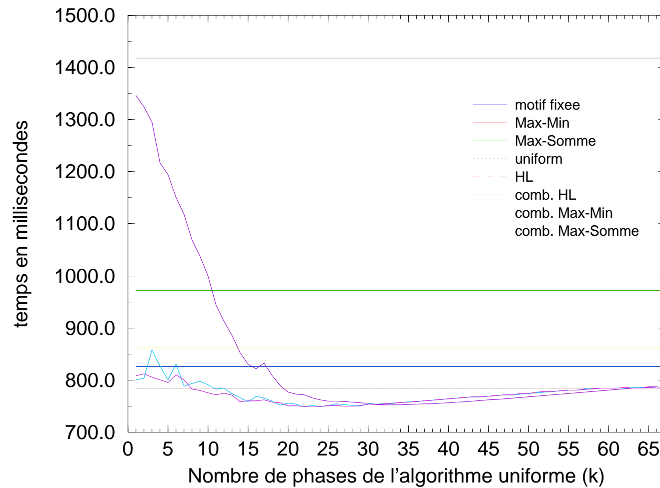

FIG. 6.16 – Performance des algorithmes quand les messages sont grands.

Dans la figure 6.16 nous présentons un MEP où le temps d'initialisation devient beaucoup moins important. Les paramètres de machine sont les mêmes que dans le cas précédent, cependant les messages sont plus grands. La

moyenne des messages et $A = 1M$ octets, l'écart type est égal à la moyenne $\sqrt{v} = A$. L'algorithme uniforme construit les plus petits ordonnancements dans ce cas. Les valeurs des *makespans* sont données dans la table ci-dessous. Nous vérifions que les algorithmes combinés ont les meilleurs performances.

HL 1417	uniforme 784	Max-Somme 863	Max-Min 826
motif fixé 972	comb. HL 752 (k^* =33)	comb. Max-Min 748 (k^* =22)	comb. Max-Somme 749 (k^* =24)

Dans les deux prochains exemples, nous avons choisi la taille moyenne des messages de façon à ce que le temps d'initialisation soit compris entre $\frac{A\gamma}{n}$ et $A\frac{n}{2}\gamma$. Dans ces cas, nos espoirs se confirment, le Max-Somme et le Max-Min algorithmes ont de meilleurs performances parmi les algorithmes homogènes.

Dans la première courbe (voir figure 6.17), la moyenne de taille des messages est $A = 400k$ octets et l'écart type $\sqrt{v} = 500k$ octets. Nous avons simulé ce cas avec 10 processeurs. La table de performances est la suivante :

HL 412	uniforme 287	Max-Somme 274	Max-Min 268
motif fixé 314	comb. HL 250 (k^* =20)	comb. Max-Min 242 (k^* =14)	comb. Max-Somme 241 (k^* =11)

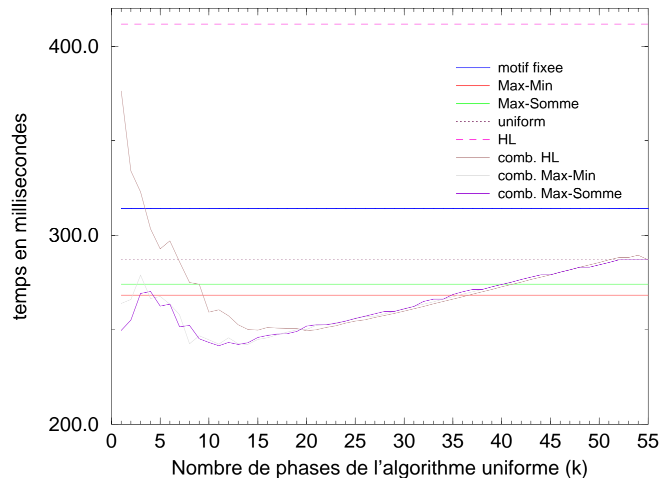

FIG. 6.17 – Performance des algorithmes dans un cas intermédiaire.

Dans la deuxième figure 6.18 nous présentons une simulation avec les

mêmes paramètres que nous avons utilisés lors des implantations. La moyenne de taille des messages est $A = 800k$ octets et l'écart type est $\sqrt{v} = 400k$ octets.

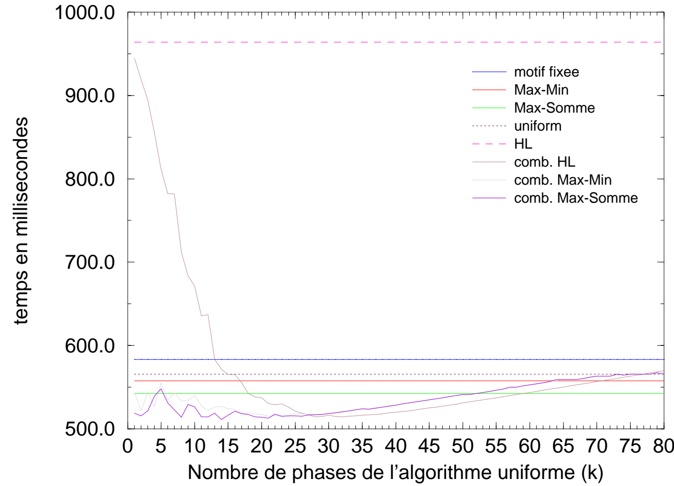

FIG. 6.18 – Performance des algorithmes dans un cas intermédiaire.

Les différences entre les valeurs obtenues avec l'implantation et avec la simulation sont dues à des effets de congestion. La machine où les implantations ont été réalisées n'est pas un réseau complet car il n'existe pas de liens exclusifs entre chaque pair de processeurs. Les processeurs sont reliés par des commutateurs selon un réseau multi-étage oméga [26]. Les meilleurs résultats de la simulations sont dans table ci-dessous.

HL	uniforme	Max-Somme	Max-Min
964	566	542	558
fixé	comb. HL	comb. Max-Min	comb. Max-Somme
583	514 (k^* =32)	515 (k^* =23)	511 (k^* =14)

D'après les simulations, nous avons pu conclure que les algorithmes combinés ont les meilleurs performances, sauf dans le cas de temps d'initialisation important. Nous avons observé aussi que l'algorithme combiné Max-Somme a été un petit peu plus performant.

Si un algorithme combiné est utilisé, alors la choix de quand stopper l'algorithme uniforme peut être dynamique car les calculs peuvent être superposés avec la communication. Dans le but de proposer un algorithme combiné

général nous introduisons l'intervalle d'horizon I (en anglais *look ahead*), car le temps de l'algorithme combiné n'est pas une fonction uniforme. La valeur de I est une fonction du volume de communication et des paramètres de la machine. Nous proposons l'algorithme combiné 6.6.

Algorithm 6.6 Algorithme combiné Max-Somme.

```
si (β > A(n/2)γ) alors
  appliquer l'algorithme HL;
sinon
  répéter
    appliquer une phase de l'algorithme uniforme;
    estimer t_MS le temps de l'algorithme Max-Somme;
    pour i = 1 jusqu'à I faire
      estimer t_ui le temps de i phases de l'algorithme uniforme suivies par
      l'algorithme Max-Somme;
    fin pour
  jusqu'à (t_MS < t_ui, 1 ≤ i ≤ I)
  si La matrice de communication n'est pas achevée alors
    appliquer l'algorithme Max-Somme;
  fin si
fin si
```

Les estimations des valeurs de t_{u_i} peuvent être faites par simulation. Pour chaque estimation de t_{u_i}, i phases de l'algorithme uniforme sont appliquées à la matrice de communications courante, puis l'algorithme Max-Somme est appliqué. Évidement pour chaque interaction de la boucle `répéter` une seule valeur nouvelle de t_{u_i} doit être estimée. Le surcoût de calcul par rapport à l'algorithme uniforme simple est une estimation de Max-Somme $O(n^3)$ par interaction. Le nombre total d'interactions est $O(n^2)$ ce qui implique que la complexité de l'algorithme 6.6 est $O(n^5)$.

6.6 Conclusion

Dans ce chapitre nous avons examiné le problème de l'échange de messages entre processeurs. L'intérêt provient des algorithmes où il existe des phases de calcul et d'échange de donnés intercalés, ainsi que des algorithmes dans le modèle BSP. Nous avons d'abord montré que trouver la solution optimale est un problème difficile, ensuite nous avons proposé plusieurs algorithmes pour résoudre le MEP.

Nous avons présenté cinq algorithmes polynômiaux, ces heuristiques ne sont pas nouvelles, cependant quelques unes d'entre elles ont été proposées pour des problèmes autres que le MEP. Nous avons fait une analyse de l'algorithme à utiliser selon les paramètres de la machine et du problème. Nous avons aussi proposé des algorithmes combinés pour lesquels nous obtenons les meilleurs performances. Dans ce cas aussi, la meilleure solution a été le compromis.

Une possibilité intéressante pour des recherches futures est l'étude de heuristiques à partir d'une connaissance locale du problème. Cette voie devient envisageable, surtout si la phase de pré-traitement pour échanger les tailles de messages n'est pas nécessaire.

Chapitre 7

Conclusion

Dans cette thèse nous avons étudié quatre problèmes d'ordonnancement différents. Les résultats obtenus pour ces quatre problèmes illustrent des techniques distinctes. Nous avons examiné la résolution parallèle synchrone à grain fin du problème du sac à dos, l'ordonnancement de chaînes sous un modèle synchrone à gros grain (BSP), l'ordonnancement avec duplication et l'ordonnancement de communications irrégulières. Nous pensons que pour chaque problème étudié nous avons proposé des solutions satisfaisantes.

Malgré l'intérêt théorique de la solution efficace trouvée pour la résolution du sac à dos, son intérêt pratique reste limité, car l'ordonnancement proposé ne sert qu'à des problèmes de programmation dynamique avec un graphe de précédence spécifique. Dans cette étude nous avons aussi examiné très finement l'ordonnancement. Avec l'objectif de garantir une efficacité optimale, nous avons examiné en détails le routage. L'ordonnancement proposé est aussi restreint aux machines où la communication peut se faire d'une façon implicite.

La remarque précédente n'est pas valable pour les autres études. Dans ces problèmes, les communications ont été considérées d'une façon explicite. D'un coté cette démarche est réaliste pour les machines parallèles courantes, cependant il faut prendre en compte le surcoût ajouté. Nous avons constaté que lorsque ce surcoût est considéré, les problèmes traités sont devenus une recherche de compromis. Pour l'ordonnancement de chaînes nous avons recherché un compromis entre le nombre de super-étapes et l'équilibrage de charge. Pour l'ordonnancement avec duplication, il existait un compromis, implicite, entre la communication et la duplication de tâches. Finalement, pour l'ordonnancement de communications, nous avons eu les meilleurs

résultats avec un algorithme qui réalisait un compromis entre le nombre de pas d'initialisation et l'utilisation efficace de la bande passante.

L'ordonnancement de chaînes sous BSP nous a éclairé sur les particularités liées au modèle. Dans un premier temps nous avons constaté que le regroupement de communications peut transformer un problème d'ordonnancement qui était facile en un problème difficile. Dans un deuxième temps, nous avons vérifié que malgré la complexité ajoutée il nous a été possible de proposer des heuristiques assez fines. Les principales contributions de ce travail ont été une technique directe pour transformer les algorithmes asynchrones dans des algorithmes BSP et l'introduction des techniques de regroupement de communications. Ces deux techniques peuvent être appliquées pour d'autres types de graphes. Un autre constat à partir de ce travail est l'intérêt du modèle BSP en pratique. Nous avons analysé que même si le surcoût introduit pour les synchronisations est grand, il existe de bons compromis entre l'équilibrage de charge et le nombre de super-étapes.

L'ordonnancement avec duplication sur un nombre limité de processeurs a aussi montré son intérêt. En dépit du grand intérêt de la duplication de tâches sur l'allégement du surcoût dû à l'introduction du coût de communication, ce problème a été très peu étudié quand le nombre de processeurs est limité. Le chemin de duplication utilisé pour l'algorithme EDTF peut être aussi appliqué pour des classes de graphes données pour fournir des résultats plus serrés. Cependant, le problème de la recherche d'un rapport de garantie constant pour le modèle à grands temps de communication reste ouvert.

Avec l'étude de l'ordonnancement de communications nous avons remarqué que les techniques d'ordonnancement sont sous-utilisées. Ce problème, très important dans l'algorithmique parallèle ne possédait que des approximations grossières. Par exemple, la bibliothèque *MPI Lam* utilise l'algorithme à motif fixé pour l'échange de messages. L'impact direct des algorithmes combinés proposés est une amélioration de performance importante pour les phases de communication. Le principal avantage de ces nouveaux algorithmes combinés est la compatibilité totale avec les algorithmes d'échange de messages précédents. Pour avoir un gain de performance, il suffit d'effectuer les phases d'échange de messages avec les algorithmes combinés.

En plus des perspectives de travaux futurs vues à la fin de chacun des quatre derniers chapitres, nous pouvons aussi envisager les directions pour la recherche future pour l'ordonnancement efficace de tâches sur systèmes parallèles actuels.

Perspectives

L'usage de plus en plus fréquent des concepts de parallélisme met en évidence les besoins de connaissance et de maîtrise du sujet. Aujourd'hui les outils qui utilisent le parallélisme implicite ont leurs limitations. Ils ne sont pas nombreux et il n'y a pas de garantie que ces outils et les techniques employées seront adaptés aux machines et réseaux futurs. Mais surtout, ces outils qui d'un coté facilitent la vie du programmeur, sont basés sur des modèles généraux d'exploration du parallélisme au niveau de la description ainsi qu'au niveau de l'architecture de la machine. Seule l'intervention directe du programmeur peut permettre la performance optimale. Nous pouvons faire l'analogie de cette intervention avec l'utilisation des langages assembleur pour accélérer les performances des algorithmes.

L'étude spécialisée a aussi d'autres avantages. Elle permet une connaissance abstraite du problème traité, laquelle ne dépend pas des modèles. Ainsi les problèmes de portabilité actuels ou futurs sont plus abordables.

Nous pensons que le développement d'algorithmes d'ordonnancement spécialisés a encore de beaux jours devant lui, particulièrement avec la croissance du poids des communications dans le parallélisme et avec l'hétérogénéité des systèmes. Car, contrairement à ses débuts, l'informatique parallèle se tourne vers les réseaux d'ordinateurs au détriment des machines spécialisées. Pour alléger le surcoût de communication, des techniques étudiées telles que le regroupement et l'ordonnancement de communications ainsi que la duplication de tâches peuvent être utilisées. Lorsque le problème de l'hétérogénéité est introduit, il faut prendre en compte aussi la vitesse de calcul de chaque processeur et l'irrégularité du support de communication. Il est clair que dans ce cas les problèmes d'ordonnancement deviennent plus difficiles et pourraient même n'être traités que dynamiquement par des algorithmes généralistes. Fort de l'expérience acquise dans le domaine de l'algorithmique parallèle homogène, nous pensons bénéficier d'outils et techniques pour aborder ces nouveaux problèmes.

Annexe A

Linéarité par morceau

Dans cette annexe nous présentons les latences d'envoie des messages par rapport à leur taille dans l'IBM SP2. Nous avons constaté durant nos premières expérimentations avec MPI F que la latence des communications peut être approchée par le modèle linéaire. Cependant, il est clair que selon la taille des messages le temps d'initialisation ainsi que la bande passante changent.

Nous avons implanté un algorithme du type *ping-pong* pour estimer la latence pour l'envoi d'une message entre deux processeurs. Dans cet algorithme nous mesurons plusieurs fois le temps nécessaire à un aller-retour d'un même message. Nous illustrons dans la figure A.1 les résultats obtenus pour la machine IBM SP2. Chaque point dans la courbe représente la moyenne de 100 tests.

Taille du message	β	γ
$0 \leq S < 4kb$	$95\mu s$	$0,085\mu s$/octet
$4kb \leq S < 16kb$	$180\mu s$	$0.086\mu s$/octet
$16kb \leq S < 32kb$	$830\mu s$	$0.046\mu s$/octet
$32kb \leq S < 1Mb$	$1300\mu s$	$0.035\mu s$/octet

TAB. A.1 – Paramètres du modèle linéaire pour le IBM SP2

Nous dérivons à partir des données utilisées pour construire la courbe A.1 les valeurs de la table A.1 pour le temps d'initialisation et pour la bande passante. Dans la courbe A.1 nous avons les valeurs du temps de latence selon la taille du message S. Nous avons quatre régions avec des inclinaisons

différentes. Si nous supposons que la latence est une fonction $\beta + S\gamma$, chaque région peut être approchée par une fonction linéaire.

Dans le but d'avoir une approximation plus fine, nous avons aussi mesuré les temps minimaux pour la même expérience. Dans ce cas, il apparaît un autre phénomène. Il existe des sauts de discontinuité à chaque intervalle de $16k$ octets. Cette anomalie a été aussi observée par Arruabarrena et al [80]. Cependant, sans informations additionnelles du constructeur nous n'étions pas capables d'expliquer ce comportement.

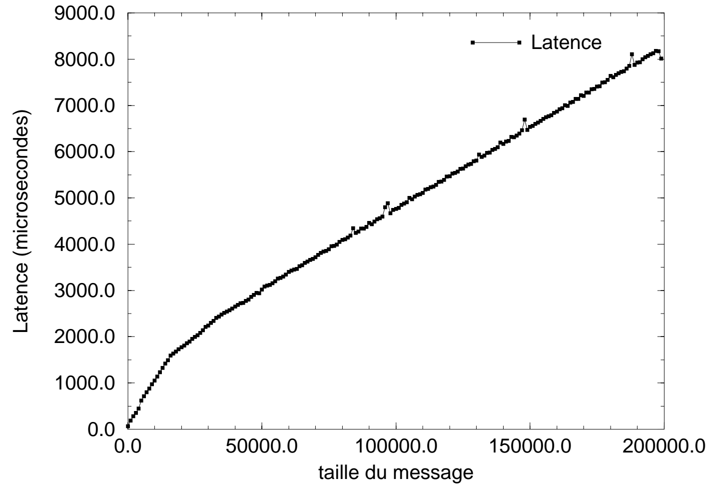

FIG. A.1 – Courbe de temps moyen pour la transmission d'un message.

Fondée sur les mesures de la latence moyenne, nous proposons une extension naturelle du modèle linéaire. Le modèle k-linéaire possède k régions où la latence d'envoi est considérée. Dans chaque région les valeurs de β et γ peuvent être différentes. La table A.2 illustre la description du modèle.

$$\begin{array}{cc} \beta_1 + S\gamma_1, & \text{if } S \leq S_1, \\ \beta_2 + S\gamma_2, & \text{if } S_1 < S \leq S_2, \\ \beta_i + S\gamma_i, & \text{if } S_{i-1} < S \leq S_i, \\ \beta_K + S\gamma_K, & \text{if } S_{K-1} < S. \end{array}$$

TAB. A.2 – Modèle K-linéaire.

Annexe B

Solving Knapsack on hypercube

This chapter contains some technical material from [51]. A preliminary version of this work was published in [49].

B.1 Full description of Knap

Together with the value of the final profit, we have also to obtain the full description of the knapsack. For that purpose, we propose three solutions. Two solutions consist in sending a data array along with each transmitted data, and the third one uses backtracking. To leave uniform the solution description, it is supposed that a task that receives data from just one other task calculates the maximum of $-\infty$, corresponding to no data received, and the received data.

bit array In this solution, a task (i, j) sends along with the value to be transmitted a bit array, and an index. Tasks $(0, j), j < w_0$, send index 0 with their data. If a task (i, j) receives its maximum from its upper task $(i-1, j)$, it sets to 0 the position index of the array, otherwise it sets it to 1. Then, the task increments the index, and send it along with the bit array to the next tasks.
This array of bits has at most $\left\lfloor \frac{c}{w_{\min}} \right\rfloor + m$ elements, and in this array there are $m-1$ zeros. We can easily construct the knapsack instance by visiting this array from the beginning, counting the number of ones before each zero, this number corresponds to the number of objects a_i, where i corresponds to the number of zeros already visited.

normal array Each task updates an array of m elements which contains a

partial solution. Tasks $(0, j), j < w_0$ send an array of zeros with their data. If a task (i, j) chooses the maximum from the left task $(i, j - w_i)$ it updates the number of objects of kind i.

backtracking The third solution consists in the backtracking technique. In order to implement it, each processor has to keep the value of its computed tasks. After calculating the final profit, the path which gives the final profit is found, by visiting the irregular mesh from the task where the final profit was computed, in reverse order. At most $\left\lfloor \frac{c}{w_{\min}} \right\rfloor + m$ tasks are visited in order to construct the path, and hence the instance.

B.2 Routing scheme

For the routing between adjacent hypercubes we use the following principle: Whenever a hypercube has finished to process a set of tasks (from independent chains), it sends the results to the next hypercube (according to the labeling H_k). Each result has to be transmitted to the processor that will execute the tasks depending on this result.

Generally at each step, a hypercube H_k receives w' results to be routed. As this hypercube executes w_i chains, each result have to be delivered to one of its w_i processors. The idea for avoiding conflicts is to route by dimensions using a pipeline technique. The first w' results are first routed in dimension 0 of the hypercube. In the next step, the second w' results are routed in dimension 0, while the first w' results continue to be routed in dimension 1, and so on.

After receiving w' results y times, the y-th set of results is routed on dimension 0, and the $(y - x)$-th set of results (if $x \leq \log_2 w_{\max}$) is routed on dimension x. This assures the routing of the results to satisfy the precedence relations within at most $\log_2 w_{\max} + 1$ steps.

Routing example

We detail in this section a routing example on the algorithm execution for an irregular mesh with $w_0 = 3$ and $w_1 = w_{\max} = 8$. First, we give below the notation that will be used in the partial execution example. Each processor is represented by a circle. On the right of each processor we have represented 3 buffers (equal to $\log_2 w_{\max}$), for transmissions on dimensions 0, 1 and 2.

Each of these buffers can store at most $\log_2 w_{\max}$ data. This means that at most $\log_2 w_{\max}$ data are sent on a link at each step.

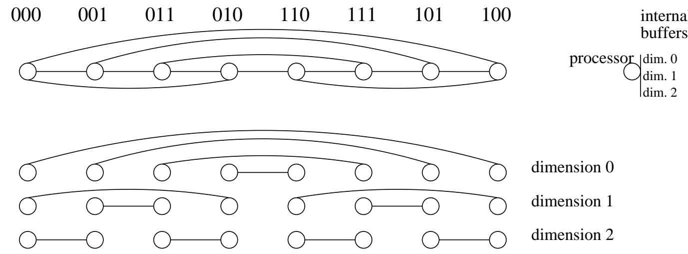

FIG. B.1 – Basic notation: the hypercube and the links on each dimension.

In figure B.2, we draw the first rounds of an execution example, the pair associated to a processor represents the task being processed.

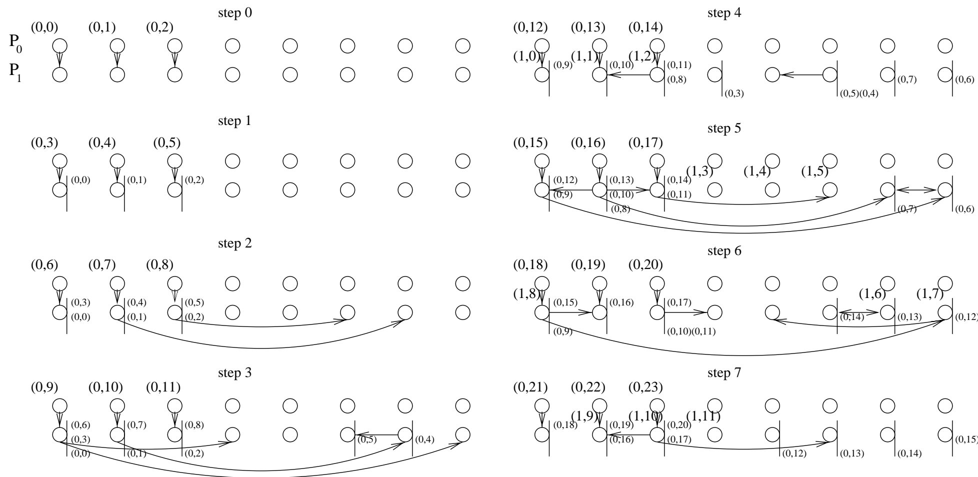

FIG. B.2 – Partial execution, with $w_0 = 3$ and $w_1 = w_{\max} = 8$.

The execution on the first path (P_0) is straightforward, at each step, 3 tasks are executed and sent to the next path (P_1). Let us detail the execution on the second path:

- Step 0 is straightforward.
- On step one, the results of the first w_0 tasks that are available can be transmitted on dimension 0, in such a way that in the next step these tasks will be closer to the processor where they are needed due to

the precedence relations. In this step there is no routing transmission because the results are already on the right processors.

- On step two, there can exist routing on dimensions 0 and 1. As before, the tasks on the second buffer position do not need to be transmitted. The tasks on the first buffer position are transmitted on dimension 0. Observe that the result of the task $(0, 4)$ is at distance 3 from destination (fifth processor on P_1), this explains why this result will be transmitted in the next two steps.
- On step three, as there are results on all buffer positions, they could be transmitted on all the three dimensions. There are effective transmissions on dimension 0 (tasks $(0, 6)$ and $(0, 7)$) and on dimension 1 (tasks $(0, 3)$ and $(0, 4)$).
- On step four, the routing scheme guarantees that the first results are on the right processor. These processors are ready to process the tasks from the second row of the irregular mesh.
- In general, a task is processed, and sent to the next hypercube, in the first buffer position, on the next step. Then it is transmitted (if necessary) and moved to the second buffer position two. Then, again to the third buffer position. In the next step the results can be transmitted (on dimension 2), and will be in the right processor, the one that executes the successor of the task result.

B.3 Properties on the average idle time

The attention will be focused now in the relationship between two adjacent paths. First of all, the routing can be the same as the previous one, as at each step one path sends at most $w' \leq w_{\min}$ results to the next path. The proposed order is a non decreasing order on the average idle time of the objects. Using this hypothesis, and given two adjacent rows (L_i, L_{i+1}) of an irregular mesh, we can state the following property:

Property B.1 *The execution of row L_i may cause an idle step on the execution of row L_{i+1} only just after the execution $y_i > 0$ tasks.*

Proof: If $y_i = 0$, row L_i does not cause an idle step on the execution of row L_{i+1}. Otherwise, on the execution of row L_i, the flow of $w_{\min}$ tasks by step is only interrupted after executing $y_i < w_{\min}$ tasks.❏

When the objects are sorted in a non decreasing order on the Average idle time $I(w_i)$ we can state the following proposition:

Proposition B.1 *Given two adjacent rows (L_i, L_{i+1}) of an irregular mesh there is at most one idle step on the execution of row L_{i+1} in a hypercube due to the data precedence.*

Proof: By property B.1, the existence of an idle step, implies that $y_i > 0$. Suppose that an idle step occurs on time step x, so we have the following formula:

$$\left\lfloor \frac{x}{\left\lceil \frac{w_i}{w_{\min}} \right\rceil} \right\rfloor (w_{\min} - y_i) > \left\lfloor \frac{x}{\left\lceil \frac{w_{i+1}}{w_{\min}} \right\rceil} \right\rfloor (w_{\min} - y_{i+1}),$$

where on the left side we count the number of times that a processor was idle processing row L_i, and on the right side we do the same for row L_{i+1}. But, by property B.1, the idle steps may occur only when x is a multiple of $\left\lceil \frac{w_i}{w_{\min}} \right\rceil$, so restricting the analysis to these cases

$$xI(w_i) > \left\lfloor \frac{x}{\left\lceil \frac{w_{i+1}}{w_{\min}} \right\rceil} \right\rfloor (w_{\min} - y_{i+1}). \tag{B.1}$$

Let us suppose that there are more than one idle steps, due to the execution of row L_i. If we consider the first idle step on the execution of row L_{i+1} on equation B.1, we have to find x which satisfies:

$$xI(w_i) > \left\lfloor \frac{x}{\left\lceil \frac{w_{i+1}}{w_{\min}} \right\rceil} \right\rfloor (w_{\min} - y_{i+1}) + w_{\min},$$

$$xI(w_i) > \left(\frac{x}{\left\lceil \frac{w_{i+1}}{w_{\min}} \right\rceil} - 1 \right) (w_{\min} - y_{i+1}) + w_{\min},$$

$$xI(w_i) > xI(w_{i+1}) + y_{i+1},$$

but as $y_{i+1} \geq 0$ and $I(w_i) < I(w_{i+1})$, such x must be negative. So there is at most one idle step.❏

Annexe C

Scheduling independent chains on BSP

This chapter contains some technical material from [45].

C.1 SIC on two processors in BSP

Lemma C.1 *Determining the existence of a schedule of UET independent chains on two identical processors within the time t^* is an NP-complete problem.*

Proof: In the presence of communications, any schedule will have a makespan greater than t^*, so we look for a schedule with only one superstep.

The problem is in NP, since a non-deterministic algorithm only needs to guess a subset CH of the set of chains, and to check in polynomial time that the sum of the chain lengths in CH is equal to t^*. Recall that the following *Partition* problem is NP-Complete [39]:

- Instance : a set X of elements of integer length n_i, an integer B such that $\sum n_i = B$.
- Question : does a subset of total length $\frac{B}{2}$ exist?

It is straightforward to reduce this problem into the SIC problem on two identical processors. Given a partition problem, consider the problem of scheduling k independent chains of length $n_i, 1 \leq i \leq k$. If $t^* \geq \sum_{i=2}^{k} n_i$, the problem is trivial, as the first chain is bigger than the sum of the others ($n_1 \leq \sum_{i=2}^{k} n_i$). Otherwise, when $t^* = \frac{B}{2}$, a schedule with makespan t^* exists if and only if there exists a set of tasks whose total length is $\frac{B}{2}$. ❑

C.2 Complexity of SIC

Proposition C.1 *Finding the minimum makespan of the SIC problem without communication for an arbitrary number of processors is NP-hard in the strong sense.*

Proof: We use a reduction from the numerical 3-dimensional matching problem [39] which is recalled below.

N3DM problem:

- Instance : Given three sets, $X = \{x_1, \ldots, x_N\}$, $Y = \{y_1, \ldots, y_N\}$ and $Z = \{z_1, \ldots, z_N\}$. Each element $x \in X$ (respectively $y \in Y$ and $z \in Z$) has a positive integer weight $s(x)$ (respectively $s(y)$ and $s(z)$). Let B be a positive integer, such that $\sum_{i=1}^{N} s(x_i) + s(y_i) + s(z_i) = NB$.
- Question : Find N disjoint 3-partitions of $X \cup Y \cup Z$ such that in each partition $\{x_i, y_j, z_k\}$ the sum of the weights is equal to B.

This problem is known to be NP-hard in the strong sense [39]. In the presence of CS (i.e. more than one superstep), any schedule will have a makespan larger than t^*, so we look for a schedule without CS.

It is easy to show that SIC is in NP, since a non-deterministic algorithm needs only to guess an m-partition $(S_1, \ldots, S_m)$ of the set of chains, and to check in polynomial time that the sum of the chain lengths of each subset $S_j, 1 \leq j \leq m$ is at most t^*.

To do the reduction, we solve a N3DM problem by solving the schedule of $3N$ chains on N processors. Given an instance of N3DM, the length of each chain is defined as follows:

$n_{x_i} = 8B + s(x_i), 1 \leq i \leq N,$

$n_{y_i} = 4B + s(y_i), 1 \leq i \leq N,$

$n_{z_i} = 2B + s(z_i), 1 \leq i \leq N.$

The total number of tasks is $N(8B+4B+2B)+\sum_{i=1}^{N} s(x_i)+s(y_i)+s(z_i) = 15NB$.

The ideal makespan is $15B$. We will show that if there exists a schedule of length at most $15B$, then, it corresponds to a solution for the N3DM problem.

There is exactly one chain of each type (x, y and z) on each processor: each processor owns one chain of length n_{x_i} (it is not possible to allocate more than one such chain per processor, without mapping more than $15B$ tasks on a processor). Taking into account the chains of length n_{x_i}, the number of remaining available time slots is less than $7B$ (see figure C.1). Thus, there is at most one chain of length n_{x_i} per processor. Obviously, the same argument holds also for the chains of type y and z.

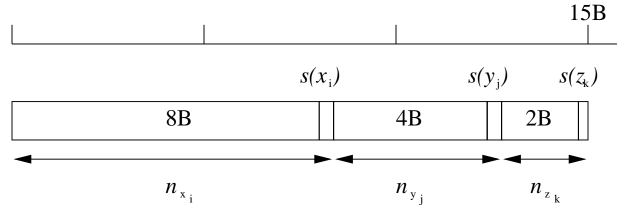

FIG. C.1 – Principle of the allocation on one processor.

Thus, each processor has exactly three chains, one of each type x_i, y_j, z_k, and $n_{x_i} + n_{y_j} + n_{z_k} = 15B$, that is $s(x_i) + s(y_j) + s(z_k) = B$. In any optimal schedule without communication, there is exactly one element from each set on each processor, and the sum of these elements is always equal to B. Thus, the solution of SIC is a solution for the N3DM problem. ❑

C.3 Guaranty on the algorithm 4.3

The main steps of the analysis are detailed as follows. Again, the details of the proofs may be found in [44].

The chains are partitioned into three disjoint sets according to their length which will be scheduled one after the other:

- Set A of chains of length t^*,
- Set B of chains of length between $\lceil \frac{t^*}{2} \rceil$ and t^*,
- Set C of chains of length at most $\lceil \frac{t^*}{2} \rceil$.

Claim C.1 *All the chains of set A can be scheduled without CS. All the chains of set C can be scheduled with a single CS at time $\lfloor \frac{t^*}{2} \rfloor$.*

It is more difficult to schedule chains on set B. The idea for reducing the number of supersteps is to gather some asynchronous communications into a single CS using the argument described in the following lemma. When the chains of set B are allocated in decreasing order of length we can state the following lemma:

Lemma C.2 *The asynchronous communications of consecutive split chains in set B can be partitioned into subsets of size at least two (only the last subset can have one communication), in such a way that the communications on each subset use the same CS.*

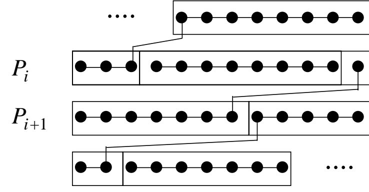

FIG. C.2 – Allocation of chains on set B.

Proof idea: The proof of the previous lemma is technical but not difficult. Two cases can occur (see figure C.2), namely processors with two or three chains of set B. On processor P_{i+1} there are two different chains which can use the same CS (see figure 4.7). On processor P_i there are three different chains, in this case, there are again two sub-cases. Let ch_j be the chain completely allocated on P_i. Either chain ch_{j-1}, which starts executing on P_i, can use the same CS as P_{i+1}. Or, it can not share this CS, but can share another CS with chain ch_{j-2}.❑

Set A does not need a CS, if set C needs a CS, set B will need at most $\lceil \frac{m-2}{2} \rceil$ CS (as set C has at least a split chain), otherwise set B will need at most $\lceil \frac{m-1}{2} \rceil$ CS. So the total number of CS is bounded by $\lceil \frac{m}{2} \rceil$.

C.4 Fixed number of supersteps algorithm

Proposition C.2 *Given a SIC instance to be scheduled within s supersteps and an integer α such that $n_1 \leq \alpha \leq t^*$, w_o^s is bounded from above by*

$$w_o^s \leq t^* + \frac{1}{2s-1}\alpha + (s-1)C.$$

Proof: This proof is constructive. The algorithm 4.4 detailed below is able to schedule any set of chains within s supersteps in time equal to the given bound. In this algorithm, we first compute where the CS are introduced, then we do the chain placement. So, we have to verify the validity of the chain allocation.

The $(s-1)$ CS are introduced from the beginning of the schedule in successive regular intervals of length $\Delta_\alpha = \frac{2}{2s-1}\alpha$. Let ϵ be the size of the last superstep, note that $\epsilon \geq \Delta_\alpha$.

The idea of this algorithm is to add some idle time (up to $\frac{\Delta_\alpha}{2}$ tasks by processor) to obtain a valid schedule. The chains are scheduled one after the other as in the asynchronous algorithm. A chain is split when a processor reaches the limit of $t^* + \frac{\Delta_\alpha}{2}$ tasks.

If a chain does not have a CS between its split parts, this chain is delayed (as shown in figure C.3). For the split chains smaller than $(s-1)\Delta_\alpha$, it is easy to verify that there always exists a CS between its two parts. For the other chains, if there is a split chain *ch*, it can be delayed in order to use one of the placed CS. When delaying of a chain, the tasks on the first processor where the chain is allocated are transfered, until a CS is found, to the second processor. As *ch* has at most n_1 tasks, and n_1 is smaller than $(s-1)\Delta_\alpha + \frac{\Delta_\alpha}{2}$, this delay is at most $\frac{\Delta_\alpha}{2}$. Of course, this process is not cumulative since it does not affect the CS locations.

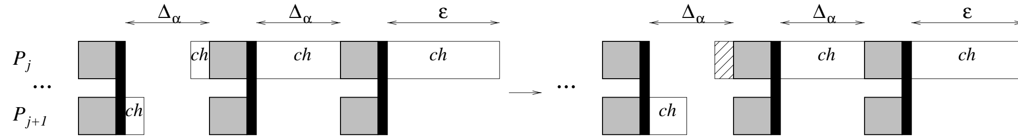

FIG. C.3 – Allocating chains longer than $(s-1)\Delta_\alpha$. If there is no CS between the two parts of a split chain on a interval, the tasks belonging to this interval on P_j are transfered to P_{j+1} and the rest of chain starts after CS.

By the proposed allocation, the scheduled time is $t^* + \frac{\Delta_\alpha}{2} + (s-1)C$. Each processor, except maybe the last ones which can have less than t^* tasks, have at least t^* tasks to process, and a chain that is split and allocated on two different processors uses one of the introduced CS in order to communicate. So, the schedule generated by algorithm 4.4 is feasible. ❑

We study now, for an instance of SIC, the number of supersteps which guarantees a minimal idle time schedule, and then number of supersteps which minimizes the overhead of the schedule produced by algorithm 4.4 $\frac{\Delta_\alpha}{2} + (s-1)C$.

Claim C.2 *When the chains are small ($n_1 < \lceil \frac{t^*}{2} \rceil$), $w_o^s \leq t^* + C$.*

Proof: Again, the proof is constructive. First, one CS is introduced at time $\lceil \frac{t^*}{2} \rceil$. Then, the chains are allocated one after the others as in the asynchronous algorithm. As the chains are small, all the split chains can use the CS in order to communicate.❑

We can state the following corollary of proposition C.2.

Corollary C.1 *Given a SIC instance and* $s_o = \max\{s | n_1 > \frac{s-1}{s} t^*\}$. *For all* s, $s > s_o$, $w_o^s \leq t^* + jC$.

Proof: Introducing the CS in successive intervals of length $\frac{t^*}{s}$ (which corresponds to Δ_α for $\alpha = t^*$), the condition on s_o guarantees that the longest chain is smaller than $t^* - \frac{t^*}{s}$ (that is $(s-1)\Delta$). So, if the chains are placed as in proposition C.2, there will always be a CS between the parts of a split chain. ❑

On proposition C.2, the bound on overhead is given by the sum of two terms, one proportional to the number of CS, and the other inversely proportional to the number of CS. To find a good compromise between the load balance and the number of supersteps, we present now the following result.

Proposition C.3 *The optimal number of supersteps to minimize the overhead for the schedule produced by algorithm 4.4 is given by* $s^*_{\max} = \left\lceil \sqrt{\frac{\alpha}{2C}} + \frac{1}{2} \right\rceil$ *or* $s^*_{\max} = \left\lfloor \sqrt{\frac{\alpha}{2C}} + \frac{1}{2} \right\rfloor$.

Proof: The length of the schedule is the sum of t^* plus the sum of two positive expressions: a decreasing function on s, $\frac{\Delta_\alpha}{2}$ and an increasing one, $(s-1)C$. So, the makespan is trivially minimized when the derivate on s is equal to zero. But, as s has to be integer, the minimum makespan is obtained by $s^*_{\max}$ the floor, or the ceil of $\sqrt{\frac{\alpha}{2C}} + \frac{1}{2}$. ❑

So, the best compromise algorithm is easy to derive:

Algorithm C.1 Best compromise algorithm.

Choose α such that $n_1 \leq \alpha \leq t^*$

Compute $s^*_{\max}$

Apply algorithm 4.4 with $s^*_{\max}$ supersteps

C.5 Bound improvement

Proposition C.4 *Using the algorithm improvement, with a fixed number of supersteps* s*, the bound becomes:*

$$t^* + \max\left(\frac{\Delta_\alpha}{2} - \frac{t^* - n_1}{2}, n_1 - (s-1)\Delta_\alpha, 0\right) + (s-1)C$$

.

Proof: In addition to $t^* + (s-1)C$, there is the maximum of two terms: the first term comes from the difference between t^* and n_1. This difference is a lower bound of the elapsed time between the two parts of a split chain. As $t^* - n_1$ increases, the time to add to t^* in order to schedule the chains in the worst case decreases. This time is proportional to $\Delta_\alpha - (t^* - n_1)$, but as we choose the less costly rearrangement, this time is divided by 2. This term increases with α

The second term comes from the date of the last CS, as it is the last time a chain can be split. If the first part of a chain is greater than this time, it needs to be truncated in order to be scheduled. This term decreases with α. ❑

C.6 Influence of the latency

The aim of this section is to show that it is possible to take the constraint of the latency into account with a small additional overhead proportional to λ. The main idea is to delay, or advance, close CS until they are spaced by a time interval greater than λ.

If a CS is delayed, an additional idle time may be introduced on the receiver side. The effect is to increase its completion time. If a CS is advanced, some idle time may be introduced on the sender side and the equivalent number of tasks can be migrated to the receiver side increasing the completion time (cf. figure C.4).

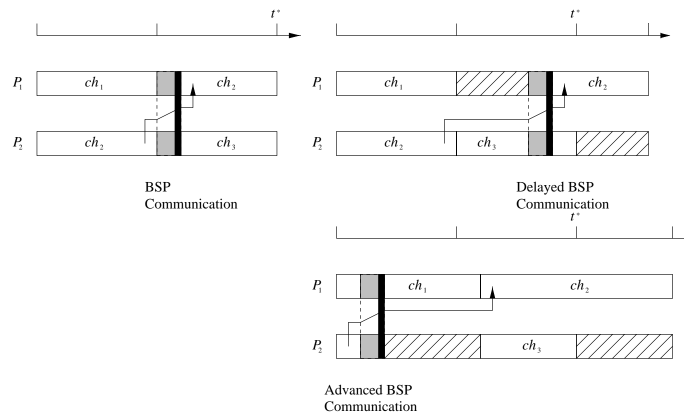

FIG. C.4 – Moving CS forward or backward

On figure C.4, on the initial scheduling, three chains are placed. Chain ch_2 starts on P_2 and finishes on P_1. If the CS is delayed, the end of ch_2 is delayed too. On the other hand, if the CS is advanced, the end of the first part of ch_2 must be moved to P_1 and increases its completion time. So, any change on a CS date will only increase the receivers completion time. The procedure is detailed on algorithm 4.5. This procedure is available for any BSP algorithm and transform it to a new (feasible) BSP algorithm with the latency constraint, but in general, we can not guarantee its performance.

Proposition C.5 *The additional cost from one of the proposed BSP algorithms, taking into account the latency influence as stated in algorithm 4.5 is a* $+\frac{\lambda}{2}$ *term.*

Proof: All CS introduced by the algorithm are displaced (forward or backward) of at most of $\frac{\lambda}{2}$ units. Each processor increases its completion time only if the communication it receives is displaced. As each processor receives only one communication, each processor finishes at most $\frac{\lambda}{2}$ time units after the initial BSP algorithm execution time. As each processor is the origin of only one communication, each merged CS still corresponds to a 1-relation. ❑

Annexe D

Scheduling with duplication

This chapter contains some technical material from [48].

D.1 Number of favorite predecessors

Property D.1 *A task i admits at most one favorite predecessor j.*

Proof: Indeed if we suppose the existence of another favorite predecessor j' for task i, we then have $d_i < f_{j'} + c_{j'i}$. It imposes that the communication delay between j' and i does not occur, and hence that d_i is reached on $\pi = \pi_{j'}$. The same argument clearly holds for task j: if not allocated to $\pi_{j'}$, we would have $d_i \geq f_j + c_{ij}$ meaning that j is not a favorite predecessor. Hence without loss of generality we can assume that j is executed before j' on a same processor π. It involves that $f_{j'} \geq f_j + p_j \geq f_j + c_{ji}$, due to the small communication time hypothesis. But clearly $d_i < f_j + c_{ji} \leq f_{j'}$, which leads to a contradiction.❑

D.2 Feasible starting time

Property D.2 *If a delay occurs on processor π between some of the duplicated tasks of $\mathcal{D}_i^k$, then $\tau_{\pi,i} = d_i$.*

Proof: Indeed if we suppose that a duplicated copy $\tilde{\beta}$ of task β in the path is not scheduled just after the completion of its favorite duplicated predecessor, clearly $t_{\tilde{\beta}} = t_\beta$. Hence the following copies up to j will be scheduled on π at the same time than their original tasks. ❑

We introduce Φ_k as the sum of the computation times along $\mathcal{D}_i^k$ (as pointed out previously, task i does not belong to $\mathcal{D}_i^k$). We also define μ_k^π as $\max\{0, f_\alpha - (f_{a_k} + c)\}$, i.e. the duplicated path $\mathcal{D}_i^k$ starts executing at time $f_{a_k} + c + \mu_k^\pi$, cf figure D.1. The choice of k implies that $f_\alpha < t_{b_k} + c$, as $f_{a_k} + \Delta_{b_k} = t_{b_k}$ we have $\mu_k^\pi < \Delta_{b_k}$.

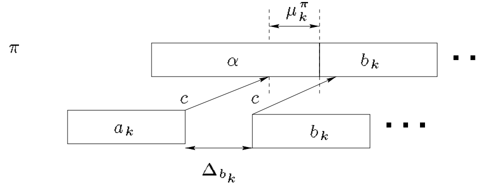

FIG. D.1 – The choice of $k(\pi)$, with the graphical meaning of μ_k^π.

If no delay occurs in the duplicated path $\mathcal{D}_i^k$, we can then write that $\tau_{\pi,i} = f_{a_k} + c + \mu_k^\pi + \Phi_k$, as $d_i = f_{a_k} + \Phi_k + \sum_{l=0}^{k} \Delta_{b_l}$,

$$\text{hence} \quad \tau_{\pi,i} \leq d_i + c + \mu_k^\pi - \sum_{l=0}^{k} \Delta_{b_l}. \tag{D.1}$$

In particular we have then $\tau_{\pi,i} \leq d_i + c$, since $\mu_k^\pi < \Delta_{b_k} \leq \sum_{l=0}^{k} \Delta_{b_l}$. We can also deduce the following property:

Property D.3 *If a_r denotes the root of the duplicative path of i, and if π is idle after time $f_{a_r} + c$, then $\tau_{\pi,i} = d_i$.*

Proof: Notice that in this case $\tau_{\pi,i}$ is obtained duplicating all the path from b_r, the successor of the root. Recall the possible two situations for the root of the duplicative path. In the case that a_r is a dummy task, or equivalently that b_r is scheduled after its free date, any task of the path can be clearly duplicated on π at the same date that its original copy, since π is idle after time $f_{a_r} + c = t_{b_r}$. Otherwise we are in the situation where the total delay time Δ of the duplicative path is greater than c. Suppose for sake of contradiction that $\tau_{\pi,i} > d_i$. Property D.2 involves that no delay occur, and hence the inequality D.1 can be applied. It results that $\tau_{\pi,i} \leq d_i + c - \Delta \leq d_i$. Contradiction. ❏

We then define the "earliest" feasible starting time τ_i of i as $\min_\pi\{\tau_{\pi,i}\}$. Before analyzing the performance of the algorithm we establish an helpful lemma:

Lemma D.1 *Given π such that $\tau_{\pi,i} > d_i$. Then for any π', we have:*

$$\tau_{\pi,i} \leq \tau_{\pi',i} \Rightarrow k(\pi) \geq k(\pi').$$

Proof: We prove the lemma showing the reverse implication. With straightforward notations suppose that $k' > k$. We have $\tau = f_{a_k} + c + \mu_k^\pi + \Phi_k$, and $\tau' = f_{a_{k'}} + c + \mu_{k'}^\pi + \Phi_{k'}$. But certainly $f_{a_{k'}} + \Phi_{k'} \leq f_{a_k} + \Phi_k - \Delta_{b_{k'}}$. It involves that $\tau' \leq \tau - \mu_k^\pi + \mu_{k'}^\pi - \Delta_{b_{k'}} < \tau$, since $\mu_{k'}^\pi < \Delta_{b_{k'}}$. ❑

D.3 Algorithm analysis

Lemma D.2 *Let $\mathcal{I}[0,t)$ be the total idle time during $[0,t)$, including the non-original copies of tasks. Then for any task i we have*

$$\mathcal{I}[0,t_i) \;\leq\; mt_i^\infty.$$

Proof: The proof is done by induction on the schedule construction. We denote by $\mathcal{W}_\pi[0,t)$ the total amount of original tasks performed on π during $[0,t)$. An equivalent formulation of lemma D.2 is to establish that for any task i we have $\mathcal{W}_\pi[0,t_i) \;\geq t_i - t_i^\infty$ for all the processors. With the notations introduced previously consider the step of the algorithm where a schedule decision is taken for the task i.

To proof the Lemma we study the same four possibilities as in Section 5.3.2. When scheduling task i, the four possibilities are:

Case 0 i has no predecessor; case 1 i has no favorite predecessor; case 2 i has a favorite predecessor and is scheduled either right after its favorite predecessor, or by a duplication scheme; and case 3 i has a favorite predecessor and is not scheduled by a duplication scheme.

Now we detail all the possible cases:

Case 0. If $\Gamma_i^- = \emptyset$, clearly all processors are busy in $[0,t_i($ and the lemma holds. Hence suppose in the following that $\Gamma_i^- \neq \emptyset$.

In order to prove the next cases, we start by proving the following lemma:

Lemma D.3 *For any real $t \in [d_i - c, d_i]$, we have $\mathcal{W}_\pi[0,t(\geq t - t_i^\infty$.*

Proof: consider any real $t \in [d_i - c, d_i]$. Let j be the predecessor of i maximizing the value $f_j + c_{ji}$. Said differently task j verifies $\tilde{d}_i = f_j + c_{ji}$. Notice that since $\rho \leq 1$ we surely have t greater than t_j. Suppose first that d_i coincides with f_j. By induction on j we get $\mathcal{W}_\pi[0,t_j) \geq t_j - t_j^\infty$. On the

other hand we have both $t_i^\infty \geq t_j^\infty + p_j$ and $d_i = t_j + p_j$. It involves that $\mathcal{W}_\pi[0, t_j) \geq d_i - t_i^\infty$. Since $t \geq t_j$ we write that $\mathcal{W}_\pi[0, t) \geq \mathcal{W}_\pi[0, t_j)$, and hence $\mathcal{W}_\pi[0, t) \geq d_i - t_i^\infty \geq t - t_i^\infty$.

Consider conversely that $d_i > f_j$. It implies the existence of another predecessor j' of i verifying $f_{j'} + c_{j'i} = d_i$. In σ^∞ we have either $t_i^\infty \geq f_j^\infty + c_{ji}$ or $t_i^\infty \geq f_{j'}^\infty + c_{j'i}$. Assume for instance, w.l.o.g, that the last inequality holds. By induction on j' we obtain in the same way $\mathcal{W}_\pi[0, t_{j'}) \geq d_i - t_i^\infty$. Since $t_{j'}$ is necessarily smaller than t, we can deduce that $\mathcal{W}_\pi[0, t) \geq \mathcal{W}_\pi[0, t_{j'}) \geq d_i - t_i^\infty \geq t - t_i^\infty$. □

We consider now the different allocation scenarios for task i, depending on the existence of a favorite predecessor or not. If such a predecessor exists, we make a supplementary distinction considering if a duplication scheme is used or not to schedule i. By duplication scheme we mean the allocation of i to a processor π_i such that $k(\pi_i)$ is defined. We detail below the different situations:

Case 1. Assume that i has no favorite predecessor. We have then $d_i = \tilde{d}_i$. Since i is absolutely free at time d_i, the greedy allocation of the algorithm ensures that all processors are busy in the time interval $[d_i, t_i)$. Using the lemma D.3, the result is immediate.

Case 2. Assume that i actually has a favorite predecessor, say j, if i is scheduled just after j, so $t_i = d_i$ and the result follows from lemma D.3. If i is scheduled by a duplication scheme. It implies in particular that $t_i \in [d_i, d_i + c]$ (inequality D.1). Due to lemma D.3 we can restrain to the case where $t_i > d_i$, which involves by the property D.2 that no delay occur in its duplicated path.

Let α be the last processed task on processor π. First we state a lemma for the sub-case $t_\alpha \leq d_i$, and then we study the sub-case where $t_\alpha > d_i$. Observe that in the latter sub-case, by the allocation algorithm, we have $t_\alpha \leq t_i$. We state now the following lemma:

Lemma D.4 *Let π be a processor and α be its last processed task in the current schedule. If $t_\alpha \leq d_i$, then $\mathcal{W}_\pi[0, t_i) \geq t_i - t_i^\infty$.*

Proof: consider a processor π and its last allocated task α, which is necessarily an original task. If $t_\alpha \geq t_i - c$, then $f_\alpha \geq t_i$, and hence the task is processed during the whole interval $[t_\alpha, t_i]$. But as $t_\alpha \leq d_i$ (lemma hypothesis), t_α is then in the interval $[d_i - c, d_i]$, and we can apply lemma D.3 which leads to the result. Hence we focus in the following in the case where $t_\alpha < t_i - c$.

We have two situations to consider, depending on the duplication scheme of i. Let $k = k(\pi_i)$ and a_k be the task chosen by the algorithm as the initial communication task for the duplication. If we note β the last task processed on π_i, it may happen that this task is being processed, or not, at time $f_{a_k} + c$.

Case 2a. Assume that task β is being processed at time $f_{a_k} + c$ on π_i and denote by λ its remaining processing time in $[t_{b_k}, +\infty)$. Notice that the algorithm ensures that $\lambda \leq c$ by the choice of k. Since no delay occurs in the duplicated chain, we have $t_i = t_{b_k} + \lambda + \Phi_k$. But clearly $t_i^\infty \geq t_{b_k}^\infty + \Phi_k$ always holds. It implies that:

$$t_i - t_i^\infty \ \leq \ (t_{b_k} - t_{b_k}^\infty) + \lambda. \qquad (2a)$$

Hence by using the inductive hypothesis on t_{b_k}, we just have to establish that $\mathcal{W}_\pi[t_{b_k}, t_i) \ \geq \lambda$. Since β is an original task, this inequality certainly holds for π_i. Hence assume $\pi \neq \pi_i$. Using lemma D.1, we know that $k(\pi) \leq k$. Hence:

- either task α completes after time $t_{b_k} + c$,
- or $k(\pi) = k$ and $t_{b_k} + c > f_\alpha$.

In the first case, since $t_\alpha < t_i - c$, at least c units of β are processed in the time interval $[t_{b_k}, t_i)$, leading to the result. Hence consider the second situation and let η be the remaining processing time of α in $[t_{b_k}, +\infty)$. To ensures that $\tau_{\pi,i} \geq \tau_{\pi_i,i}$, we surely have $\eta \geq \lambda$. It involves that $\mathcal{W}_\pi[t_{b_k}, t_i) \ \geq \eta \geq \lambda$.

Case 2b. Conversely assume that task β is completed at time $f_{a_k} + c$. As a corollary of property D.3 remark that a_k can not be the root of the duplicative chain. Hence we can consider tasks a_{k+1} and b_{k+1}. Since no delay occurs between b_{k+1} and a_k, we have $t_i = t_{b_{k+1}} + c + \Phi_{k+1}$. As $t_i^\infty \geq t_{k+1}^\infty + \Phi_{k+1}$, it follows that:

$$t_i - t_i^\infty \ \leq \ (t_{b_{k+1}} - \ t_{b_{k+1}}^\infty) + c. \qquad (2b)$$

As in the previous case, but using the induction hypothesis on b_{k+1}, we just need to establish that $\mathcal{W}_\pi[t_{b_{k+1}}, t_i) \ \geq c$. Since a_{k+1} was not chosen to initiate the duplication, a task was being processed (or is not started yet) on π_i at time $t_{b_{k+1}} + c$, which proves the inequality on π_i. If $\pi \neq \pi_i$, using again the lemma D.1 we have $k(\pi) \leq k$. In particular to ensure that $k(\pi) < k + 1$ we certainly have $t_{b_{k+1}} + c \leq f_\alpha$. As $t_\alpha \leq t_i - c$, it involves that α is computed at least c units of times in the interval $[t_{b_{k+1}}, t_i]$. Hence we

have $\mathcal{W}_\pi[t_{b_{k+1}}, t_i) \ \geq c$. □

To conclude the case 2, consider any processor π with its last computed task α. If we have $t_\alpha \leq d_i$, lemma D.4 shows immediately that $\mathcal{W}_\pi[0, t_i) \geq t_i - t_i^\infty$. Otherwise due to the greedy allocation α is certainly the only task (including copies) on π starting after the date d_i, and we have $t_\alpha \leq t_i$. Consider the partial scheduling S' obtained from removing of π the task α and its associated duplicated path if any. The last task of π in S' is then an original task starting before time d_i. Let t'_i be the feasible starting time of i in S', with necessarily $t'_i \geq t_\alpha > d_i$. Lemma D.4 can then be applied, showing that in S' we have $\mathcal{W}_\pi[0, t'_i) \geq t'_i - t_i^\infty$. But since $t_\alpha \leq t'_i$, task α is being processed during the whole period $[t'_i, t_i]$. It results that $\mathcal{W}_\pi[0, t_i) \geq (t'_i - t_i^\infty) + (t_i - t'_i) \geq t_i - t_i^\infty$, which concludes the case 2.

Case 3. Assume finally that i has a favorite predecessor j and is not scheduled using a duplication scheme. In this case an original task complete after time $d_i + c$ on any processor, and i is allocated in a greedy way to the first finishing processor. It is hence clear that no idle time can occur in the interval $[d_i + c, t_i]$. As previously consider a processor π and the schedule S' where any task completing after date $d_i + c$ is removed, with its associated duplicated path if any, from π. We denote by α the original task of smallest starting time removed from π. In S' a duplication scheme certainly can schedule the task i at a date $t'_i \leq d_i + c$. The case 2 shows that we have $\mathcal{W}_\pi[0, t'_i) \geq t'_i - t_i^\infty$. However the greedy allocation of the tasks ensures that $t'_i \geq t_\alpha$. Since between t_α and t_i no idle period occur on π we have $\mathcal{W}_\pi[t'_i, t_i) = t_i - t'_i$. It involves finally that $\mathcal{W}_\pi[0, t_i) \geq (t'_i - t_i^\infty) + (t_i - t'_i) = t_i - t_i^\infty$, which achieves the proof. □

D.4 Algorithm complexity

Proposition D.1 *The EDTF algorithm can be implemented in $\mathcal{O}(n^2 \log n)$, where n is the number of tasks in the precedence graph.*

Proof: The dominant complexity part of the algorithm is off course the computation and the updating of the feasible starting times at each step. Let introduce U the set of tasks that becomes ready at the current step. The algorithm can be implemented as in algorithm 5.2.

We can compute the duplicative path of a task in $\mathcal{O}(n)$ storing the list of the starting time of the delayed predecessors. Given π and task i the computation of $\tau_{\pi,i}$ is in $\mathcal{O}(\log n)$ by dichotomic search in the list of its delayed predecessors. For each task we insert in $\mathcal{O}(m \log m)$ its different feasible starting times in a quick search structure such as a balanced tree. Roughly speaking step 1 gives a total complexity of $\mathcal{O}(n^2 + mn \log m)$. The updating of τ_i needs to recompute $\tau_{\pi_0,i}$ for all $i \in F$, and the search for the min value. The computation of $\tau_{\pi_0,i}$ is performed in $\mathcal{O}(\log n)$ while the update of the min value with the tree structure needs $\mathcal{O}(\log m)$. Step 4 leads to complexity $\mathcal{O}(n \log n)$ for each schedule step and hence its total complexity dominates the algorithm.$\square$

Annexe E

Scheduling with communications

This chapter contains some technical material from [47].

E.1 Complexity

The MEP problem is also NP-complete in an asynchronous model:

Theorem E.1 *When $\beta > 0$, the problem of deciding if there is a schedule for $dG(V', E')$ with completion time at most M is NP-complete for the asynchronous model.*

Proof: The proof uses a reduction from the 3-partition problem.

3-partition problem: Given $A = \{a_1, \ldots, a_{3k}\}$, with $B/4 < a_i < B/2, 1 \leq i \leq 3k$, where $B = 1/k \sum_{i=1}^{3k} a_i$, does there exist a partition $A = A_1 \bigcup A_2 \bigcup \ldots A_k$ such that $\sum_{a \in A_i} a = B, 1 \leq i \leq k$?

Let $k = 2n + 1$, and construct the MED of Figure E.1. In Figure E.1, the triangles represent several out-going arcs, with very small weight. We will consider only the start-up time for these arcs. Let $W = 3k + 2(n+1) = 8n + 5$ (the out-degree of vertex v). The number of arcs in a triangle is either $W - 4i$ (if the other arc leaving the vertex is f_i or f'_i), or $4i+1$ (if the other arc leaving the vertex is d_i or d'_i).

Let $M = \sum_{i=1}^{3k} a_i + 2(n+1)$, let $C_i = i(B+1), 1 \leq i \leq n$ and choose $\beta > \max\{(W - B)\gamma, (nB + n + 1)\gamma\}$. In order to simplify the description, each arc label is directly associated with its weight. The weight on each arc labeled a_i is $a_i, i \leq i \leq 3k$. The other arc weights are as follows, for $1 \leq i \leq n$.

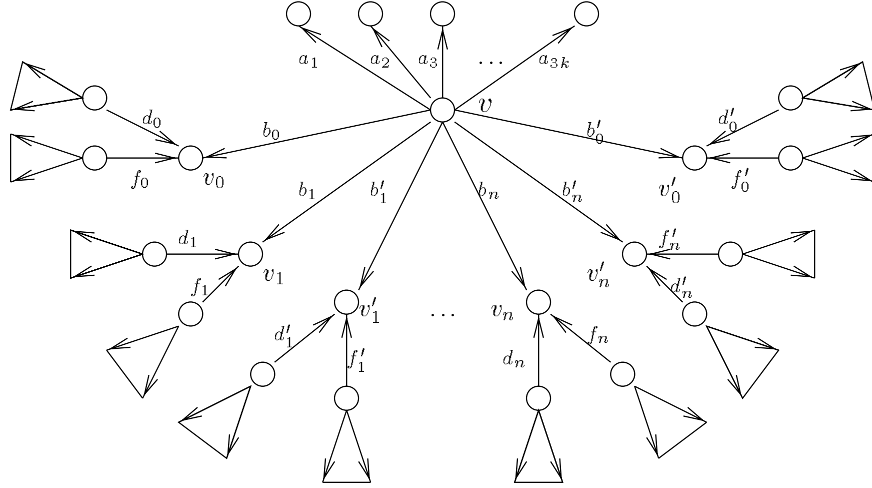

FIG. E.1 – Tree used in Theorem E.1

$b_i = b_{i'} = 1,$
$d_i = d_{i'} = (W - 4i - 2)\frac{\beta}{\gamma} + M - C_i - 1,$
$f_i = f_{i'} = (4i - 1)\frac{\beta}{\gamma} + C_i.$

We also have $b_0 = b'_0 = 1$, $d_0 = d'_0 = d_1$ and $f_0 = f'_0 = f_1$.

If there is a schedule for the graph in Figure E.1 within time $W\beta + M\gamma$ then the following conditions are satisfied:

– The message transmissions corresponding to the arcs leaving vertex v cannot be preempted. The out-degree of vertex v is W and the sum of the weights of its out arcs is M. If a transmission is preempted the makespan would be bigger than $W\beta + M\gamma$.
  The arcs f_i and f'_i, $0 \leq i \leq n$, leave vertices with out-degree $W - 4i + 1$. The minimum time required to schedule all of the arcs leaving one of these vertices is $\beta + (4i - 1)\beta + C_i\gamma + (W - 4i)\beta = W\beta + C_i\gamma$. If $x > 0$ preemptions were done, the time to schedule these vertices would be at least $(W + x)\beta + C_i\gamma$. Since $\beta > (W - B)\gamma$, $(W + x)\beta + C_i\gamma > W\beta + (W - B + C_i)\gamma > W\beta + M\gamma$, so x must be 0. A similar argument can be applied to the arcs d_i and d'_i and the triangles containing $4i + 1$ arcs. Their schedule time is $\beta + (W - 4i - 2)\beta + (M - C_i - 1)\gamma + (4i + 1)\beta = W\beta + (M - C_i - 1)\gamma$. If $x > 0$ preemptions were done, the makespan would be at least $(W + x)\beta + (M - C_i - 1)\gamma$. Since $\beta > (n(B + 1) + 1)\gamma$, $(W + x)\beta + (M - C_i - 1)\gamma > W\beta + (C_n + 1 + M - C_i - 1) > W\beta + M\gamma$,

so x must be 0.

In summary, the schedule time $W\beta + M\gamma$ is impossible if any transmission is preempted.

- The arcs b_0, b'_0, b_1 and b'_1 have to be scheduled at different times because they all begin at vertex v. The schedules for vertices v_0, v'_0, v_1 and v'_1 have to be consistent with the times chosen for b_0, b'_0, b_1 and b'_1. The scheduling time for vertex v_0 is $\beta + (M - C_1 - 1)\gamma + (W - 6)\beta$ (due to arc d_0) $+\beta + \gamma$ (due to arc b_0) $+\beta + C_1\gamma + 3\beta$ (due to arc f_0), that is $W\beta + M\gamma$. So, vertex v_0 receives messages during the whole schedule. The same is true for the vertices v'_0, v_1 and v'_1.

  It follows that the only possible schedule for these vertices is the one shown in Figure E.2 where each line corresponds to one of the vertices. Without loss of generality we assume that b_0 is scheduled in the interval $[0, \beta+\gamma]$, and b'_0 is scheduled in the interval $[(W-1)\beta+(M-1)\gamma, W\beta+M\gamma]$.

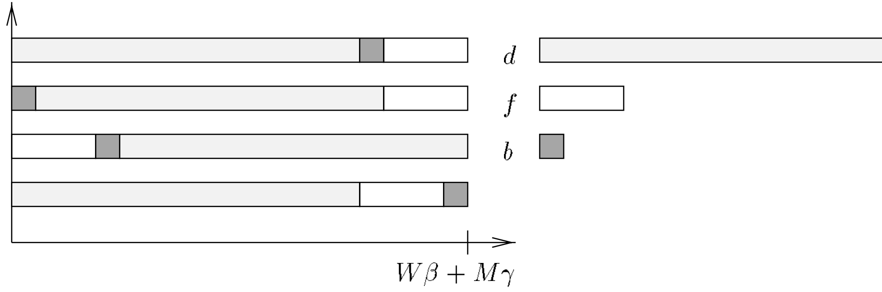

FIG. E.2 – The possible schedule for the arcs entering vertices v_0, v'_0, v_1 and v'_1.

- Arcs $b_i, 1 \leq i \leq n$, cannot be scheduled at the same time as b_0 and b'_0, so each b_i must be scheduled either after d_i and before f_i or after f_i and before d_i. The same is true for the arcs b'_i. More precisely, the arcs b_i and b'_i have to be scheduled in intervals $I_i = [4i\beta + C_i\gamma, (4i+1)\beta + (C_i+1)\gamma]$ and $I_{2n+1-i} = [(W-4i-1)\beta + (M-C_i-1)\gamma, (W-4i)\beta + (M-C_i)\gamma], 1 \leq i \leq n$.

  One can easily verify that between intervals I_i and $I_{i+1}, 1 \leq i < 2n$, there is a gap of size $3\beta + B$.
- No arc a_i can be scheduled before b_0 or after b'_0. At most three of these arcs can be scheduled in a gap of size $3\beta + B$ (since $a_i > B/4$). As there are $2n + 1$ such gaps, there are exactly three arcs a_i scheduled in each

one.

So, if there is a feasible schedule, there are three arcs scheduled in each gap (between I_i and I_{i+1}). Since the size of each gap is exactly $3\beta + B$, the weights of the arcs scheduled in each gap determine a 3-partition of the set A. ❏

E.2 Even number of processors HL algorithm

One alternative when the number of processors is not a power of 2 is to use an algorithm based on Knödel graphs [67] of dimension $\lfloor \log_2 n \rfloor$. We recall below the definition of Knödel graphs with even number of vertices.

Definition E.1 *Given a set* $\{p_0, \ldots, p_{n-1}\}$ *of* n *nodes labeled as* $p_i = (\lfloor \frac{2i}{n} \rfloor + 1, i \bmod \frac{n}{2})$*, the Knödel graph of degree* $\lfloor \log_2 n \rfloor$ *has edges between every node* $(1, j), 0 \leq j \leq \frac{n}{2} - 1$ *and each node* $(2, j + 2^k - 1 \bmod \frac{n}{2})$*, for* $k = 0, \ldots, \lfloor \log_2 n \rfloor - 1$*. The edge which connects a node* $(1, j)$ *to a node* $(2, j + 2^k - 1 \bmod \frac{n}{2})$ *is said to be in dimension* k *(cf figure E.3).*

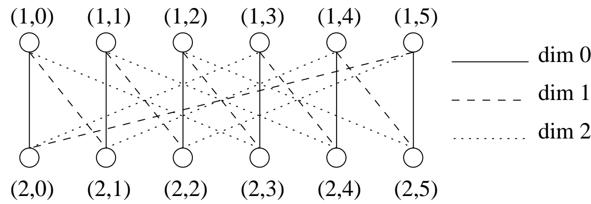

FIG. E.3 – Knödel graph with 12 nodes.

Using the notion of Knödel graphs, the following algorithm from [35], for an even number of processors has $\lceil \log n \rceil$ steps:

```
for t = 1 to ⌊log_2 n⌋ do
   do in parallel for all p_i (0 ≤ i < n)
       p_i exchanges messages with the neighbor in dimension t − 1
if n is not a power of 2 then
   do in parallel for all p_i (0 ≤ i < n)
      p_i  exchanges messages with the neighbor in dimension  0
```

On the algorithm when n is not a power of two, there exists more than one path from a node to some of the others. In such a case, in the algorithm

implementation, the message will be transmitted on the path to be used latter. The analysis is similar to the hypercube-like algorithm analysis, there are $\lceil \log_2 n \rceil$ phases and in each phase at most $\frac{n}{2}$ messages are sent by processor.

E.3 Matching algorithms

In this section we present the bipartite matching algorithms [70], the max-min matching and the max-weight matching. Both algorithms are described from a bipartite graph $G(V = S, T, A)$, where S and T are the vertex, and A the edges set.

E.3.1 Max-min matching

In this strategy, we look for a sequence of maximal matchings in which the minimum weight is maximized. The algorithm for finding such matchings is polynomial. The matching is found in a constructive way, the algorithm starts with the empty matching, the next matching is obtained by mean of an augmenting path on the previous one. So, at step l of the algorithm we have a matching with l edges, where the minimum weight is maximized. Given a problem instance, this strategy finds a solution that solves the problem, except on some particular instances, in at most $n - 1$ phases.

1. (Start) The bipartite graph $G(S, T, A)$ and a weight w_{ij} for each arc $(i, j) \in A$ are given. Set $X = \emptyset$, $W = +\infty$, and $\pi_j = -\infty$ for each node $j \in T$. No nodes are labeled.
2. (Labeling)
   (a) Give the label "$\emptyset$" to each exposed node in S.
   (b) If there are no unscanned labels, go to step 4. If there are unscaned labels, but each unscanned label is on a node $i \in T$ for which $\pi + i < W$, then set $W = \max\{\pi_i | \pi_i < W\}$.
   (c) Find a node i with an unscanned label, either $i \in S$ or else $i \in T$ and $\pi_i \geq W$. If $i \in S$, go to step 2.d; if $i \in T$ go to step 2.e.
   (d) Scan the label on node $i (i \in S)$ as follows. For each arc $(i, j) \notin X$ incident to i, if $\pi_j < w_{ij}$ and $\pi_j < W$, then give node j the label "i" (replacing any existing label) and set $\pi_j = w_{ij}$. Return to step 2.b.

   (e) Scan the label on node $i (i \in T)$ as follows. If node i is exposed, go to step 3. Otherwise, identify the unique arc $(i, j) \in X$ incident to node i and given node j the label "i". Return to step 2.b.

3. (Augmentation) An augmenting path has been found, terminating at node i (identified in step 2.e). The nodes preceding node i in the path are identified by backtracing from label to label. Augment X by adding to X all arcs in the augmenting path that are not in X, and removing from X those which are. Remove all labels from nodes. Set $\pi_j = -infty$, for each node $j \in T$. Return to step 2.b

4. (Hungarian Labeling) No augmenting path exists, and the matching X is a max-min matching of maximum cardinality.

E.3.2 Max-weight matching

In this strategy, we look for matchings in which the sum of the weights is maximal. The algorithm for finding such matchings is polynomial. Again, the augmenting path technique is used.

The max-weight strategy is equivalent to the assignment problem: Given an $n \times n$ matrix, find a subset of elements in the matrix, exactly one element in each column and one in each row, such that the sum of the chosen elements is minimal

1. (Start) The bipartite graph $G(S, T, A)$ and a weight w_{ij} for each arc $(i, j) \in A$ are given. Set $X = \emptyset$. Set $u_i = \max\{w_{ij}\}$ for each node $i \in S$. Set $v_j = 0$ and $\pi_j = -\infty$ for each node $j \in T$. No nodes are labeled.

2. (Labeling)

   (a) Give the label "$\emptyset$" to each exposed node in S.

   (b) If there are no unscanned labels, or if there are unscaned labels, but each unscanned label is on a node $i \in T$ for which $\pi > 0$, then go to step 4.

   (c) Find a node i with an unscanned label, either $i \in S$ or else $i \in T$ and $\pi_i = 0$. If $i \in S$, go to step 2.d; if $i \in T$ go to step 2.e.

   (d) Scan the label on node $i (i \in S)$ as follows. For each arc $(i, j) \notin X$ incident to i, if $u_i + v_j - w_{ij} < \pi_j$, then give node j the label "i" (replacing any existing label) and set $\pi_j = u_i + v_j - w_{ij}$. Return to step 2.b.

(e) Scan the label on node $i (i \in T)$ as follows. If node i is exposed, go to step 3. Otherwise, identify the unique arc $(i, j) \in X$ incident to node i and given node j the label "i". Return to step 2.b.

3. (Augmentation) An augmenting path has been found, terminating at node i (identified in step 2.e). The nodes preceding node i in the path are identified by backtracing from label to label. Augment X by adding to X all arcs in the augmenting path that are not in X, and removing from X those which are. Remove all labels from nodes. Set $\pi_j = +infty$, for each node $j \in T$. Return to step 2.b

4. (Change in dual variables) Find

$$\delta_1 = \min\{u_i | i \in S\}, \delta_2 = \min\{\pi_j | \pi_k > 0, j \in T\}, \delta = \min\{\delta_1, \delta_2\}.$$

Subtract δ from u_i, for each labeled node $i \in S$. Add δ to v_j for each node $j \in T$ with $\pi_j = 0$. Subtract δ from π_j for each labeled node $j \in T$ with $\pi + j > 0$. If $\delta < \delta_1$ go to step 2.b. Otherwise X is a maximum weight matching.

Bibliographie

[1] A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, March 1990.

[2] S.G. Akl and G.R. Guenther. Broadcasting with selective reduction. In *Proceedings of the 11th IFIP Congress*, pages 515–520, San Francisco, California, August 1989.

[3] V. Aleksandrov and S. Fidanova. On the expected execution time for a class of non uniform recurrence equations mapped onto 1d array. *Parallel Algorithms and Applications*, 1:303 – 314, 1994.

[4] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, July 1997.

[5] R. Andonov and P. Quinton. Efficient linear systolic array for knapsack problem. Technical Report 639, IRISA - France, 1992.

[6] D.A. Bader, D.R. Helman, and J. JáJá. Practical parallel algorithms for personalized communication and integer sorting. Technical report, Institute for Advanced Computer Studies, and Department of Electrical Engineering, University of Maryland, November 1995.

[7] P. Barcaccia, M. Bonuccelli, and M. Ianni. Minimun length scheduling of precedence constrained messages in distributed systems. In *EUROPAR'96, LNCS 1123*, number LNCS 1123, pages 594–601, Lyon, France, 1996.

[8] M. Barnett, L. Shuler, R. van de Geijn, S. Gupta, D.G. Payne, and J. Watts. Interprocessor collective communication library. In *Proceedings of the Scalable High Performance Computing Conference*, pages 357–364, Knoxville, Tennessee, 1994.

[9] J. Błażewicz, K. Ecker, G. Shmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems.* Springer-Verlag, 1993.

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

[11] S.H. Bokhari. Multiphase complete exchange on a circuit switched hypercube. In *Proceedings of the 1991 International Conference on Parallel Procesing*, volume I, pages 525–529, Boca Raton, FL, 1991.

[12] G. Bongiovanni, D. Coppersmith, and C. Wong. An optimum time slot assignment algorithm for an SS/TDMA system with variable number of transponders. *IEEE Transactions on Communications*, 29:721–726, 1981.

[13] M.A. Bonuccelli. A polynomial time optimal algorithm for satellite-switched time-division multiple access satellite communications with general switching modes. *SIAM Journal on Discrete Mathematics*, 4:28–35, February 1991.

[14] R. Calinescu. Bulk synchronous parallel scheduling of uniform dags. In Luc Bouge et al., editors, *Euro-Par'96. Parallel Processing*, volume 2 of *Lecture Notes in Computer Science 1124*, pages 555–562. Springer-Verlag, 1996.

[15] R. Calinescu. A BSP approach to the scheduling of tightly-nested loops. In *Proc. 11th International Parallel Processing Symposium (IPPS'97), 1-5 April 1997, Geneva, Switzerland*, pages 549–553. IEEE Computer Society Press, 1997.

[16] G.H. Chen, M.S. Chern, and J.H. Jang. Pipeline architectures for dynamic programming algorithms. *Parallel Computing*, 13(1):111 – 117, 1990.

[17] H. Choi and S.L. Hakimi. Data transfer in networks. *Algorithmica*, 3:223–245, 1988.

[18] P. Chrétienne, E.G. Jr Coffman, J.K. Lenstra, and Z. Liu. *Scheduling Theory and its Applications.* John Wiley & Sons Ltd, 1995.

[19] E.G. Coffman, M.R. Garey, and D.S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[20] E.G. Coffman, M.R. Garey, D.S. Johnson, and A.S. Lapaugh. Scheduling file transfers. *SIAM Jornal on Computing*, 14(3):744–780, August 1985.

[21] J-Y. Colin and P. Chrétienne. CPM scheduling with small interprocessor communication delays. *Operations Research*, 39(3), 1991.

[22] M. Cosnard and A. Ferreira. On the real power of loosely coupled parallel architectures. *Parallel Processing Letters*, 1(2):103–111, December 1991.

[23] M. Cosnard, A. Ferreira, and H. Herbelin. The two list algorithm for the knapsack problem on a FPS T20. *Parallel Computing*, 3(9):385 – 388, 1988/89.

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

[25] V.-D. Cung, P. Fraigniaud, T. Gautier, and D. Trystram. De l'algorithme au support. In D. Barth, J. Chassin de Kergommeaux, J.-L. Roch, and J. Roman, editors, *ICaRE'97: conception et mise en oeuvre d'applications parallles irrgulires de grande taille*, Aussois, France, December 1997. CNRS.

[26] J. de Rumeur. *Communications dans les réseaux de processeurs*. Masson, 1994.

[27] B. Di Martino and G. Iannello. Parallelization of non-simultaneous iterative methods for systems of linear equations. In *Parallel Processing: Compar 94 – Vapp VI*, volume 854 of *Lecture Notes in Computer Science*, pages 253–264, Linz, Austria, 1994. Springer.

[28] J. Eisenbiegler, W. Loewe, and W. Zimmermann. BSP, LogP, and oblivious programs. In *Europar'98*, number 1470 in lncs, pages 865–874, Southampton, England, September 1998. LNCS 1470, Springer-Verlag.

[29] J. Eisenbiegler, W. Lowe, and A. Wehrenpfennig. On the optimization by redundancy using an extended logP model. In *International Conference Advances in Parallel and Distributed Computing*, pages 149–155. IEEE Computer Society Press, 1997.

[30] J. Eisenbiegler, W. Löwe, and A. Wehrenpfennig. On the optimization by redundancy using an extended LogP model. In *International Confe-*

rence on Advances in Parallel and Distributed Computing (APDC'97), pages 149–155. IEEE Computer Society Press, 1997.

[31] A.G. Ferreira. A parallel time/hardware tradeoff $T.H = O(2^{n/2})$ for the knapsack problem. *IEEE Transactions on Computers*, 40(2):221–224, February 1991.

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

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

[34] P. Fraigniaud and E. Lazard. Methods and problems of communication in usual networks. *Discrete Applied Mathematics*, 53, 1994.

[35] P. Fraigniaud and J.G. Peters. Minimum linear gossip graphs and maximal linear (δ, k)-gossip graphs. Technical Report TR 94-06, School of Computing Science, Simon Fraser University, October 1994.

[36] P. Fraigniaud and J.G. Peters. Structured communication in cut-through routed torus networks. Technical Report TR 97-05, School of Computing Science, Simon Fraser Univ, 1997.

[37] C. Fu and T. Yang. Run-time compilation for parallel sparse matrix computations. In ACM, editor, *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing: Philadelphia, Pennsylvania, USA, May 25–28, 1996*, pages 237–244, New York, NY 10036, USA, 1996. ACM Press.

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

[39] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness.* W.H. Freeman and Company, San Francisco, 1979.

[40] T. Gerasch and P. Wang. Implementing dynamic programming on the connection machine. Technical Report Series TR-10-89, George Mason University, 1989.

[41] T. Gerasch and P. Wang. A survey of parallel algorithm for one-dimensional integer knapsack problems. *Infor*, 32(3):163–186, 1993.

[42] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic graphs. *IEEE Transaction on Parallel and Distributed Systems*, 4:686–701, 1993.

[43] A. Goldman and S. Fidanova. Parallel execution of irregular meshes into a systolic linear array. In *2nd International Conference on Parallel Processing and Applied Mathematics*, pages 267–274, Zakopane, Poland, September 1997.

[44] A. Goldman, G. Mounié, and D. Trystram. Near optimal algorithms for scheduling independent chains in BSP. In *HiPC'98*, Madras, India, December 1998. The 5th International Conference on High Performance Computing.

[45] A. Goldman, G. Mounié, and D. Trystram. Near optimal algorithms for scheduling independent chains in BSP. Submitted to Theoretical Computer Science, available at `http://www-apache.imag.fr/~goldman/publi/chaines.ps.gz`, 1999.

[46] A. Goldman, J. Peters, and D. Trystram. Exchange of messages of different sizes. In *Irregular'98, LNCS 1457*, pages 194–201, Berkeley, USA, September 1998.

[47] A. Goldman, J. Peters, and D. Trystram. Exchange of messages of different sizes. manuscrit, 1999.

[48] A. Goldman and C. Rapine. Scheduling with duplication on m processors with small communication delays. manuscript, 1999.

[49] A. Goldman and D. Trystram. An efficient parallel algorithm for solving the knapsack problem on hypercube. In *11th International Parallel Processing Symposium*, pages 608 – 615, Geneva, Switerland, April 1997.

[50] A. Goldman and D. Trystram. Algorithms for the message exchange problem. In *Parelec'98*, pages 153–159, Byalystok, Poland, September 1998.

[51] A. Goldman and D. Trystram. Efficient parallel algorithms for solving the knapsack problem on hypercube. Submitted to Journal of Parallel and Distributed Computing, 1999.

[52] T.F. Gonzales. Multi-message multicasting. In *Irregular'96, LNCS 1117*, volume LNCS 1117, pages 217–228, Santa Barbara, CA, 1996.

[53] T.F. Gonzales. Multimessage multicasting with forwarding. Technical Report, UCSD Department of Computer Science, TRCS-96-16, 1996.

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

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

[56] C. Hanen and A. Munier. An approximation algorithm for scheduling dependent tasks on m processors for small communication delays. In *Proceedings of IEEE Symposium on Emerging Technologies and Factory Automation*, volume 1, pages 167–189, Paris, France, October 1995. IEEE.

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

[58] S.M. Hedetniemi, T. Hedetniemi, and A.L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1986.

[59] J.M.D. Hill, W.F. McColl, and D.B. Skillircon. Questions and answers about BSP. Technical report PRG-TR-15-96, Oxford University Computing Laboratory, November 1996. to appear on Journal of Scientific Programming.

[60] J. Hoogeveen, J. Lenstra, and B. Veltman. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters*, 16:129–137, 1994.

[61] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of ACM*, (21):277 – 292, 1974.

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

[63] T. Kalinowski, I. Kort, and D. Trystram. Scheduling general task graphs under LogP model using a genetic algorithm. In *International Conference on Parallel Computing in Electrical Engineering (Parelec'98)*, Byalistok, Poland, September 1998.

[64] E. Karnin. A parallel algorithm for the knapsack problem. *IEEE Transactions on Computers*, 33(5):404 – 408, 1984.

[65] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 17, pages 870–941. North Holland, 1990.

[66] G.A.P. Kindervater and H.W.J.M. Trienekens. Experiments with parallel algorithms for combinatorial problems. *European Journal of Operational Research*, 33:65–81, 1995.

[67] W. Kndel. New gossips and telephones. *Discrete Mathematics*, 13:95, 1975.

[68] I. Kort and D. Trystram. Some results on scheduling flat trees in LogP model. *Journal of Information Systems and Operational Research (INFOR)*, 37(1), 1999.

[69] H.T. Kung and C.E. Leiserson. *Systolic Arrays, in: Introduction to VLSI Systems*, chapter 8.3. Addison-Wesley, 1980.

[70] E.L. Lawler. *Combinatorial optimization : networks and matroids.* Holt, Rinehart and Winston, 1976.

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

[72] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing*, 5(4):438–456, 1988.

[73] F. Leighton. *Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*. Morgan Kaufmann, 1992.

[74] J. Lin and J. Storer. Processor efficient hypercube algorithm for the knapsack problem. *Journal of Parallel and Distributed Computing*, 13(3):332 – 337, 1991.

[75] Z. Liu. A note on Graham's bound. *Information Processing Letters*, 36:1–5, October 1990.

[76] W. Löwe and W. Zimmermann. Programming data-parallel – executing process parallel. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications, 1995*, pages 50–64. IOS Press, 1995.

[77] W. Löwe and W. Zimmermann. Upper time bounds for executing PRAM-programs on the generalized LogP-machine. In M. Wolfe, editor, *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 41–40, Barcelona, Spain, July 1995. ACM.

[78] W.F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments, 1995*, volume 1000 of *LNCS*, pages 46–61. LNCS 1000, Springer-Verlag, 1995.

[79] M. Middendorf, W. Löwe, and W. Zimmermann. Scheduling inverse trees under the communication model of the LogP-machine. *Theoretical Computer Science*, 215(1–2):137–168, February 1999.

[80] J. Miguel, A. Arruabarrena, R. Beivide, and J.A. Gregorio. Assessing the performance of the new IBM SP2 communication subsystem. *IEEE parallel and distributed technology: systems and applications*, 4(4):12–22, Winter 1996.

[81] MPI Forum. MPI: A message-passing interface. Technical Report CSE-94-013, Oregon Graduate Institute of Science and Technology, April 1994.

[82] C.H. Papadimitriou. *Computational complexity*. Addison-Wesley, New York, 1994.

[83] C.H. Papadimitriou and J. Ullman. A communication time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, 1987.

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

[85] G. Park, B. Shirazi, and J. Marquis. DFRN: A new approach for duplication based scheduling for distributed memory multiprocessor systems. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS-97)*, pages 157–166, Los Alamitos, April 1–5 1997. IEEE Computer Society Press.

[86] G-R. Perrin and A. Darte. The data parallel programming model: foundations, HPF realization, and scientific applications. In *Lecture Notes in Computer Science, 1132*, volume 1132, New York, NY, USA, 1996. Springer-Verlag Inc.

[87] J.G. Peters and C.C. Spencer. Global communication on circuit-switched toroidal meshes. *Parallel Processing Letters*, 8(2):161–175, June 1998.

[88] C. Picouleau. New complexity results on scheduling with small communication times. *Discrete Applied Mathematics*, 60:331–342, 1995.

[89] M.L. Pinedo. *Scheduling: theory, algorithms and systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

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

[91] J.L. Roda, C. Rodriguez, F. Sande, and D.G. Morales. A new model for the analysis of asynchronous parallel algorithms. In *Lecture Notes in Computer Science, 1497 - 1998*, volume 1497, pages 387–394, 1998.

[92] G.N Rouskas and V. Sivaraman. On the design of optimal TDM schedules for broadcast wdm networks with arbitrary transceiver tuning latencies. In *Infocom'96*, pages 1217–1224, 1996.

[93] V. Sarkar. *Partitioning and scheduling parallel programs for multiprocessors*. Pitman, 1989.

[94] A.S. Tanenbaum. *Modern operating systems*. Prentice Hall, New Jersey, 1992.

[95] S. Teng. Adaptive parallel algorithms for integral knapsack problems. *Journal of Parallel and Distributed Computing*, 8:400–406, 1990.

[96] R. Thakur and A.N. Choudhary. All-to-all communication on meshes with wormhole routing. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing (IPPS'94)*, pages 561–565, Los Alamitos, CA, USA, April 1994. IEEE Computer Society Press.

[97] A. Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 1–2(196):109–130, 1998.

[98] 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.

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

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

[101] J. Verriet. Scheduling tree-structured programs in the LogP-model. Technical Report UU-CS-1997-18, Dept. of Computer Science, Utrecht University, 1997.

[102] J-C Wang and S. Ranka. Irregular personalized communication on distributed memory machines. Technical report, School of Computer and Information Sience, Syracuse University, April 1993.

[103] J-C Wang and S. Ranka. Static and runtime scheduling of unstructured communication. Technical report, School of Computer and Information Sience, Syracuse University, February 1993.

[104] T. Yang and A. Gerasoulis. Pyrros: Statis task scheduling and code generation for message passing multiprocessors. In *6th ACM International Conference on Supercomputing*, pages 428–437, Washington, D.C., July 1992.

Résumé : Le contexte général de ce travail est l'étude du comportement d'applications parallèles, représentées par un graphe de précédence. La programmation de telles applications dépend fortement des supports d'exécution. Nous présentons et discutons les principaux modèles d'exécution et leur influence sur les problèmes d'ordonnancement des tâches du programme parallèle. Nous étudions en détail quatre problèmes d'ordonnancement sur des modèles d'exécution où le coût de communication est pris en compte. Nous proposons une solution pour un problème à grain très fin, le problème du sac à dos, sur hypercube dans un modèle d'exécution synchrone où le coût de communication est implicite. Nous étudions l'ordonnancement de chaînes sur un modèle à gros grain de communication, le modèle BSP. Nous démontrons qu'ici la recherche d'un ordonnancement optimal est un problème $\mathcal{NP}-$difficile. Nous proposons des solutions avec un compromis entre le nombre de phases de communication/synchronisation et le temps d'inactivité dans chaque processeur. Les deux derniers problèmes étudiés concernent des techniques qui permettent de réduire l'impact du coût des communications inter processeurs. La première technique considère la duplication des tâches. Nous proposons un algorithme de liste avec garantie de performance 2 pour les problèmes à petit temps de communication sur un nombre limité de processeurs. Le deuxième méthode consiste à optimiser les phases de communication en ordonnançant les transmissions de messages. La recherche de la solution optimale étant $\mathcal{NP}-$difficile, nous proposons plusieurs heuristiques.
Mots cls : Graphe de précédence, Ordonnancement, Modèle d'exécution, Délai de communication.

Abstract: In this thesis we study the behavior of parallel applications represented by a precedence graph. The programming of such applications is strongly relied to the execution support. We present and discuss the main execution models and their influence on the task scheduling problems. We study more deeply four scheduling problems on execution models that consider the communication costs. We propose one solution for a fine grain problem, the knapsack problem, on the hypercube on synchronous execution model in which the communication costs are implicit. We study the scheduling of independent chains on BSP. We show that the problem of finding an optimal scheduling is $\mathcal{NP}-$hard. We also propose trade-off solutions on the number of communication/synchronization phases and on the processors idle time. The last two problems concern techniques to reduce the impact of the communication cost. The first one considers task duplication, we propose a list algorithm with performance guaranty 2 for the problems with small communication time on a limited number of processors. The second one consists of optimizing the communication phases by scheduling the messages transmission. As the search of the optimal solution is $\mathcal{NP}-$hard, we propose several heuristics, and we analyze them experimentally.
Keywords: Precedence graph, Scheduling, Execution model, Communication delay.