HAL open science

Algorithmes d'approximation pour l'ordonnancement multi-objectif. Application aux systèmes parallèles et embarqués

Erik Saule

▶ To cite this version:

Erik Saule. Algorithmes d'approximation pour l'ordonnancement multi-objectif. Application aux systèmes parallèles et embarqués. Informatique [cs]. Institut National Polytechnique de Grenoble - INPG, 2008. Français. NNT : . tel-00345988

HAL Id: tel-00345988

https://theses.hal.science/tel-00345988

Submitted on 10 Dec 2008

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

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

INSTITUT POLYTECHNIQUE DE GRENOBLE

Numéro attribué par la bibliothèque
|__/__/__/__/__/__/__/__/__/__/

THESE

pour obtenir le grade de

DOCTEUR DE L'Institut polytechnique de Grenoble

Spécialité : Informatique

préparée au Laboratoire d'Informatique de Grenoble
dans le cadre de ***l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique***

présentée et soutenue publiquement

par

Erik SAULE

le 20 Novembre 2008

Titre :

Algorithmes d'approximation pour l'ordonnancement multi-objectif. Application aux systèmes parallèles et embarqués

JURY

M. VANGELIS PASCHOS,	Président du jury
M. ALESSANDRO AGNETIS,	Rapporteur
M. YVES ROBERT,	Rapporteur
M. DENIS TRYSTRAM,	Directeur de thèse
M. ODED MALER,	Examinateur
M. PATRICK MARTINEAU,	Examinateur

Remerciements

Ma thèse est maintenant achevée. J'ai passé trois longues années dans cette formidable aventure. Je n'aurais jamais eu la force de faire ce travail tout seul, autant socialement que scientifiquement.

J'ai été catapulté à Grenoble sur les conseils de Catherine Roucairol. Je ne serais pas la sans elle. Merci.

Merci aux membres de mon jury. À Vangelis Paschos de m'avoir fait l'honneur de présider le jury et pour ses remarques judicieuses. À mes examinateurs, Oded Maler et Patrick Martineau, pour toutes ces questions qui m'ont fait réfléchir sur la pertinence de mes travaux. Merci à Yves Robert pour avoir demandé à être rapporteur de ma thèse, pour ses corrections et les nombreuses discussions que nous avons eu. Great thanks to Alessandro Agnetis for accepting to review my thesis, coming from Italy to listen to a defense in french and for the several interesting issues he raised when we discussed.

Une thèse est aussi l'histoire du grand mandarin et du petit mandarin. Un grand merci à Chef pour m'avoir fait partagé ses connaissances, sa vision de la science, pour m'avoir forcé à réfléchir et pour son soutien permanent. Merci également pour ces discussions sur la société, l'histoire, l'art... En bref, merci Denis, tu m'as appris énormément de choses.

Merci aussi à tous les gens du havre de science ID. Merci à ceux avec qui j'ai travaillé, mais aussi à ceux avec qui je n'ai pas travaillé et qui sont tout aussi importants. Une mention spéciale pour le groupe de travail du midi sur les ensembles à 32 éléments. Certains rentrant dans plusieurs catégories, je tiens à les remercier nominativement (ce qui veut dire que je vais en oublier) : Pierre-François, Jean-Marc, Brice (je suis cycliste grâce à lui), Marin (on l'aura un jour ce donut), Fred, Greg, Florent, Yonel.

Merci à mes parents et à mes frères pour leur soutien. Merci à mes anciens colocataires, Sabrina, Nash, Julien, Gaelle, Dide et Fred pour m'avoir écouté, distrait, supporté pendant trois ans. Finalement, merci à Delphine de m'avoir supporté et soutenu sans relâche alors que je n'ai pas été facile tous les jours. Je ne serais pas arrivé au bout de ce travail tout seul. Merci à vous !

Table des matières

Table des matières **vii**

1 Introduction **1**
1.1 Contexte et motivation 1
1.2 Guide de lecture et contributions 3

2 Préliminaires **7**
2.1 Notations générales et définitions 7
2.2 Résultats classiques en ordonnancement à un objectif 9
2.2.1 Généralités 9
2.2.2 Du problème $P \mid\mid C_{max}$ 10
2.2.3 Du makespan dans d'autres modèles 12
2.2.4 Du makespan avec contraintes de précédence 13
2.2.5 De la somme des temps de terminaison 13
2.3 Concepts de l'optimisation multi-objectif 14
2.4 Résolution de problèmes multi-objectifs dans la littérature . . 17
2.5 Complexité 20
2.6 De l'approximation des problèmes multi-objectifs 22
2.6.1 Définition 22
2.6.2 Une Approximation du Zénith de MaxAndSum 24
2.6.3 Approximation de l'ensemble de Pareto de BiSumCi . 26
2.7 Bilan 29

3 Makespan et contrainte de mémoire **31**
3.1 Description du problème 31
3.2 Le problème formel 32
3.3 L'algorithme d'approximation *Symmetric Bi-Objective* 33
3.3.1 Principe 33
3.3.2 L'algorithme 34
3.4 Des rapports d'approximation impossibles 35
3.4.1 De la non existence de solution 36

3.4.2 Extension de l'idée à m processeurs 36
3.4.3 D'autres ordonnancements inexistants 37
3.5 Deux généralisations du problème 39
3.5.1 Un algorithme fondé sur *List Scheduling* 39
3.5.2 Une extension à trois objectifs sur des tâches indépendantes 42
3.6 Variante *Online* 43
3.7 Conclusion 44

4 La sûreté de fonctionnement 45
4.1 Motivation 45
4.2 Les propriétés des fautes 46
4.3 Les moyens de la sûreté de fonctionnement 46

5 Makespan et tolérance aux pannes transitoires 49
5.1 Les systèmes embarqués critiques 49
5.2 Modélisation du problème 50
5.2.1 Notations 50
5.2.2 Réplication de tâches pour la fiabilité 52
5.2.3 Calculer la fiabilité d'une allocation spatiale π 53
5.2.4 Inapproximabilité de la solution zénith 55
5.2.5 Des hypothèses du modèle 56
5.3 Résultats connexes 57
5.3.1 Optimisation du makespan sur machine hétérogène . . 57
5.3.2 Optimisation de la fiabilité 58
5.3.3 Optimisation simultanée des deux objectifs 58
5.4 Décomposition du problème en deux phases 59
5.4.1 Principe 59
5.4.2 Phase 1 : Génération d'allocation spatiale 60
5.4.3 Phase 2 : ordonnancement pré-alloué 61
5.4.4 Discussion 65
5.5 Étude expérimentale 66
5.5.1 Buts des expériences 66
5.5.2 Construction du benchmark 66
5.5.3 Protocole 68
5.5.4 Résultats et analyse 68
5.6 Conclusion 73

6 Makespan et tolérance aux pannes permanentes 75
6.1 Le calcul sur *cluster* 75
6.2 Modèle 77

6.3 Analyse 77
6.4 Tâches indépendantes 80
6.4.1 Tâches UET 80
6.4.2 Vers des temps de calcul arbitraires 81
6.4.3 Une $\langle \bar{2}, 1 \rangle$-approximation 82
6.5 Intégration à des heuristiques existantes 86
6.5.1 Généralisation d'algorithme d'ordonnancement : le cas de HEFT 86
6.5.2 Vers plus de compromis 87
6.5.3 Extension à d'autres heuristiques 90
6.6 Conclusion 91

7 Ordonnancement multi-utilisateurs 93
7.1 Introduction 93
7.1.1 Description du problème 93
7.1.2 Différentes approches 94
7.1.3 Contributions 95
7.2 Modèle et première analyse 96
7.2.1 Définitions et notations 96
7.2.2 Analyse des travaux connexes 96
7.2.3 Discussion sur la fonction à optimiser 98
7.3 Complexité et inapproximabilité 99
7.4 $MUSP(k : C_{max})$ 101
7.4.1 Approximation de la solution zénith 101
7.4.2 Distance à l'ensemble de Pareto 102
7.5 $MUSP(k : \sum C_i)$ 105
7.5.1 Analyse préliminaire pour $m = 1$ 105
7.5.2 Extension à m processeurs 106
7.6 Le cas mixte 107
7.7 Conclusion 108

8 Conclusion 109
8.1 Bilan 109
8.2 Pourquoi utiliser l'optimisation multi-objectif ? 110
8.3 Comment traiter un problème multi-objectif ? 111
8.4 Perspectives 112

Table des figures

2.1 Quelques ordonnancement possibles 16
2.2 Quelques solutions d'une instance d'exemple 18
2.3 Approximation d'ensemble de Pareto 23
2.4 Approximation de l'ensemble de Pareto par une approche de contrainte 30

3.1 Les deux ordonnancements Pareto-optimaux d'une instance simple 36
3.2 Les solutions Pareto-optimales de la seconde instance. 38
3.3 Rapports d'approximation inexistants 38

5.1 Un graphe application. 52
5.2 Illustration de la réplication pour la fiabilité sur le graphe de la Figure 5.1 53
5.3 Illustration de la réplication pour l'efficacité sur le graphe de la Figure 5.1 54
5.4 Comparaison des allocations aléatoires avec et sans descente locale 69
5.5 Impacte du nombre de générations aléatoire sur le makespan . 70
5.6 Impacte de l'initialisation des allocations spatialles avec HEFT 71
5.7 L'instance r150 avec différentes valeurs du CCR 72

6.1 Illustration de la différence entre HEFT (qui alloue i au processeur 1) et RHEFT (qui alloue i au processeur 2). 87
6.2 Limitation du nombre de processeurs en prenant les plus petits indices de fiabilité d'abord. 88
6.3 Limitation du nombre de processeurs en prenant les plus rapides d'abord. 89
6.4 Limitation du nombre de processeurs en prenant les plus petits produit $\lambda_j \tau_j$ d'abord. 90

6.5 Impact de la variable de compromis sur 100 processeurs (le cas $\alpha = 0$ est RHEFT et le cas $\alpha = 1$ est HEFT). 91

Chapitre 1

Introduction

1.1 Contexte et motivation

Aujourd'hui, l'évolution de l'informatique est telle que la majorité des systèmes sont parallèles ou distribués. Depuis les accessoires électroniques (lecteurs multimédias, téléphones portables, ...) jusqu'aux grands centres de calcul scientifique, les systèmes que nous utilisons sont composés de plusieurs unités d'exécution. Les architectures parallèles sont si présentes que les machines à flot d'exécution unique sont presque une exception. Le problème de l'exploitation efficace de ces ressources se pose alors.

Historiquement, les systèmes parallèles ont été étudiés pour les applications de calcul scientifique. Le problème principal est de répartir judicieusement les calculs à effectuer sur les différentes ressources d'exécution afin d'optimiser le temps de total de calcul d'une application. La théorie de l'ordonnancement est la branche de la recherche opérationnelle qui considère ce type de problème d'optimisation. Un problème typique d'ordonnancement consiste à choisir pour chaque tâche à traiter, la ressource d'exécution où la tâche va s'exécuter et à quelle date. Les problèmes d'ordonnancement sont très nombreux selon d'éventuels paramètres et contraintes qui affectent les tâches (comme des temps de calcul différents, des relations de précédence ou d'exclusion entre les tâches, ...), selon la plate-forme d'exécution (homogène, hétérogène, dédiée, ...) et selon l'indice de performance que l'on souhaite optimiser (par exemple, le temps total d'exécution, le temps moyen que passe une tâche dans le système, ...).

Malheureusement, la plupart de ces problèmes d'ordonnancement sont $\mathcal{NP}$-complets. Il n'est donc généralement pas possible obtenir en temps raisonnable un ordonnancement optimal (sauf si $\mathcal{P} = \mathcal{NP}$). On se concentre alors sur l'écriture de méthodes qui fournissent une solution dont la valeur

de l'indice de performance est proche de la valeur optimale. On s'intéresse usuellement à montrer que le rapport de dégradation peut être borné par une constante. On parle alors d'algorithmes d'approximation à facteur constant. L'approximation en ordonnancement a permis à de nombreuses reprises une utilisation efficace des plates-formes parallèles en fondant théoriquement des solutions logicielles comme par exemple l'analyse du modèle de programmation BSP [Gol99], les algorithmes de compilation pour processeurs VLIW, les algorithmes pour les ordonnanceurs par lots dans les *clusters* [SWW95], les modèles de tâches divisibles des grilles légères [MYCR05], et plus récemment la répartition de charge par vol de travail [ABP01].

Cependant, les systèmes parallèles modernes sont très différents de ce qu'ils étaient il y a vingt ans. Les plates-formes de calcul scientifique sont maintenant composées de beaucoup plus d'unités de calculs, la mémoire est répartie et les réseaux d'inter-connection sont hétérogènes. En 1993, 90% des machines du Top500 disposaient de moins de 100 processeurs. Aujourd'hui, 50% en ont plus de 2000. L'augmentation du nombre de processeurs a largement augmenté la puissance des machines, mais elle a également largement accru l'éventualité de l'apparition d'une panne dans ces machines. Il n'est pas concevable d'écrire une application uniquement pour optimiser les temps de calcul car inévitablement, une machine tombera en panne pendant ce calcul. Ainsi, la performance d'une application ne se mesure pas uniquement sur le temps de calcul, mais est une mesure multi-dimensionelle : le temps de calcul est une dimension et la probabilité que le calcul s'exécute correctement en est une autre. Dans d'autres systèmes parallèles modernes, la performance peut être une quantité multi-dimensionelle. Dans le cas des systèmes embarqués ou des ordinateurs portables, exécuter une application rapidement ne peut pas se faire au détriment de la durée de vie des batteries alimentant l'appareil. Dans les applications ludiques interactives, il faut concilier latence, fréquence d'affichage et qualité d'image.

L'optimisation multi-objectif est le domaine qui étudie l'optimisation des quantités multi-dimensionelles. Le problème est que nous ne disposons pas d'une relation d'ordre totale entre les différentes solutions des problèmes d'optimisation. C'est un domaine en pleine expansion. Tous les ans de nouvelles analyses apparaissent pour traiter différents problèmes multi-objectifs : la sûreté de fonctionnement en calcul parallèle [AGK04, DÖ02], la consommation énergétique des machines à vitesse variable [AF06, Bun06], la qualité de service en ordonnancement [LSV06], le coût des requêtes web [PY00] ou dans les réseaux, en dimensionnement [Laf01] ou pour l'optimisation des tables de routage [AGM06b, AGM06a].

C'est donc dans ce contexte de l'informatique parallèle et distribué mo-

derne que s'ancre cette thèse sur l'ordonnancement multi-objectif. Les modèles et les problèmes théoriques sont nombreux. Ainsi, traiter quelques problèmes *ad hoc* n'est pas le but ultime que nous recherchons. Notre ambition est de considérer l'ordonnancement multi-objectif d'un point de vue méthodologique afin de présenter un ensemble d'outils d'analyse et de résolution.

1.2 Guide de lecture et contributions

Le Chapitre 2 présente les définitions classiques en ordonnancement et en optimisation multi-objectif, ainsi que les notations qui seront utilisées dans tout le manuscrit. Les techniques classiques d'optimisation sont discutées et deux méthodes de résolutions sont détaillées : l'approximation de la solution zénith et l'approximation de l'ensemble de Pareto. Ce chapitre est largement inspiré d'un chapitre de livre en cours de publication sur l'ordonnancement multi-objectif coécrit avec Pierre-François Dutot, Krzysztof Rzadca et Denis Trystram.

Quatre problèmes seront considérés dans ce manuscrit et leur présentation sera organisée par plate-forme cible. Les deux premiers sont liés aux systèmes embarqués et les deux derniers aux grappes et grilles de calcul.

Le Chapitre 3 traite un problème répartition de tâches sur un système embarqué ou des contraintes de mémoire pour le stockage du code applicatif s'appliquent. Le premier objectif est le makespan et le second est la consommation maximale en mémoire. Dans le cas de tâches indépendantes, nous fournissons un algorithme (de paramètre de compromis Δ) qui est une $(1 + \Delta + \epsilon, 1 + \frac{1}{\Delta} + \epsilon)$-approximation de la solution zénith avec $\Delta, \epsilon > 0$ ainsi que différentes bornes de non approximabilité de la solution zénith. Pour le problème avec précédences, nous fournissons un algorithme de $(2 + \frac{1}{\Delta-2} - \frac{\Delta-1}{m(\Delta-2)}, \Delta)$-approximation avec $\Delta > 2$. Ce travail est un travail commun avec Pierre-François Dutot et Grégory Mounié et a fait l'objet d'une publication dans la conférence IPDPS en 2008.

Les deux problèmes suivants traitent de la sûreté de fonctionnement. C'est pourquoi le Chapitre 4 fournit une vue d'ensemble des problèmes de tolérance aux fautes depuis l'origine et la caractérisation des fautes jusqu'aux moyens que l'on peut mettre en oeuvre pour améliorer la sûreté de fonctionnement.

Le Chapitre 5 traite du problème d'optimisation du makespan et de la fiabilité dans les systèmes embarqués critiques en environnement réactif. La fiabilité est augmentée à l'aide de réplication active. Le premier problème qui apparaît est celui du calcul de la fiabilité. Nous proposons de la traiter à l'aide d'un nouveau schéma de réplication. Nous montrons que le zénith du problème d'ordonnancement qui en ressort n'est pas approchable à fac-

teur constant. Le problème d'ordonnancement étant difficile, nous traitons le problème à l'aide d'une méthodologie heuristique en deux phases. Dans la première, la fiabilité est augmentée jusqu'à un seuil donné en allouant des copies des tâches aux processeurs. La deuxième phase consiste uniquement à fournir pour chaque copie une date de début d'exécution. Cette méthode est finalement évaluée expérimentalement. Ce travail à été réalisé avec Alain Girault et Denis Trystram et a fait l'objet d'une publication à paraître dans JPDC.

Le Chapitre 6 traite de l'efficacité et de la sûreté de fonctionnement dans les grilles et les *clusters*. La fiabilité est augmentée grâce à un placement judicieux des tâches sur les processeurs. Les grilles et les *clusters* étant hétérogènes, il est difficile d'obtenir des algorithmes d'approximation pour l'ordonnancement d'applications quelconques. C'est pourquoi nous considérons d'abord uniquement des tâches indépendantes. Nous fournissons un algorithme de $\langle \bar{1}, 1 \rangle$-approximation pour les tâches UET et un autre algorithme de $\langle \bar{2}, 1 \rangle$-approximation dans le cas général. Ces résultats fondent théoriquement la modification d'une heuristique pour les graphes d'application. Cette heuristique est validée expérimentalement. Ce travail a fait l'objet de deux publications. La première a été écrite avec Jack Dongarra, Emmanuel Jeannot et Zhiao Shi et publiée dans la conférence SPAA en 2007. La deuxième a été écrite avec Denis Trystram et Emmanuel Jeannot et publiée dans la conférence Euro-Par en 2008.

Le Chapitre 7 considère l'ordonnancement de tâches appartenant à de nombreux utilisateurs sur une plate-forme de calcul parallèle. L'innovation par rapport aux techniques classiques de *batch scheduling* est de faire l'optimisation en fonction des besoins des utilisateurs et non pas en fonction de la plate-forme. Ainsi, chaque utilisateur choisit une fonction objectif parmi la liste suivante : makespan, somme des temps de terminaison, somme des temps de terminaison pondéré et maximum *flow time*. Nous considérons successivement les cas où tous les utilisateurs sont intéressés par la même métrique. Pour les cas du makespan et de la somme des temps de terminaison, nous obtenons des algorithmes d'approximation du zénith et nous montrons qu'il n'existe pas de meilleure approximation du zénith. Pour le cas du *flow time*, nous montrons qu'il n'existe pas d'approximation du zénith à facteur constant. Nous proposons un algorithme garanti pour le cas où les utilisateurs choisissent soit le makespan, soit la somme des temps de terminaison. Ce travail à été réalisé avec Denis Trystram et a fait l'objet d'un article accepté à IPDPS 2009.

Le Chapitre 8 conclut ce document en fournissant un bilan sur les problèmes abordés ainsi qu'un point de vue synthétique sur les méthodes de résolution en optimisation multi-objectif.

Durant cette thèse, deux autres travaux ont été menés. Ils ne sont pas inclus dans ce document car ils sortent du cadre thématique abordé. Le premier travail a été mené avec Jean-François Méhaut et Brice Videau sur une bibliothèque de calcul parallèle pour les machines multi-coeurs pour les petits jeux de données. Il s'agit d'exploiter efficacement ces architectures de façon à pouvoir traiter en parallèle de petits volumes de calcul afin d'améliorer les performances d'applications courantes. Une publication à ce sujet est parue dans la conférence RenPar en 2008. Le second travail concerne l'écriture d'un algorithme d'ordonnancement pour un processeur de type VLIW à *bypass* incomplet. En terme d'ordonnancement, il s'agit d'ordonnancer un graphe de tâches UET sur une machine de calcul hiérarchique. Ce travail a fait l'objet d'une soumission à *Journal Of Scheduling* et a été effectué en collaboration avec Florent Blachot, Guillaume Huard, Johnatan Pecero et Denis Trystram.

Chapitre 2

Préliminaires

Tout le début de ce chapitre concerne des notions classiques en ordonnancement et peut être trouvé dans des livres tels que [BESW93, Leu04a].

2.1 Notations générales et définitions

De façon générale, l'ordonnancement consiste à allouer des tâches sur des machines de façon à optimiser une certaine fonction objectif. Cette vue des problèmes d'ordonnancement est synthétisé par la célèbre notation à trois champs $\alpha \mid \beta \mid \gamma$ proposée par Graham *et al.* [GLLK79]. Le champ α sert à décrire le type de machines utilisé. Le champ β sert à décrire les contraintes qu'une solution valide doit respecter. Le champ γ exprime la fonction objectif qu'il faut optimiser. Nous laissons volontairement de côté les problèmes de *general shop* pour nous concentrer sur les problèmes de machines parallèles. Dans ce domaine on parle souvent de processeurs et pas de machines.

Soit $M = \{1, \ldots, m\}$ un ensemble de m processeurs qui doivent exécuter n tâches de $T = \{1, \ldots, n\}$. De façon générale, j sera l'indice des processeurs alors que i sera l'indice des tâches.

Trois modèles de processeurs seront considérés : processeurs identiques, processeurs hétérogènes uniformes et processeurs hétérogènes non uniformes. Ces modèles sont respectivement notés P, Q et R dans la littérature. Ils se distinguent par les temps de traitement des tâches. Dans le modèle P (processeurs identiques), la tâche i est calculée par n'importe quel processeur avec la même durée p_i. Dans le modèle Q (processeurs hétérogènes uniformes), les processeurs se distinguent par le temps de traitement d'une opération τ_j. La tâche i est alors composée de o_i opérations élémentaires traitées en $p_{ij} = o_i \tau_j$ unités de temps. Dans le modèle R (processeurs hétérogène non uniformes), il n'y a pas de relations *a priori* sur les temps de calcul des tâches par les

processeurs. Le temps de calcul de i sur le processeur j est alors p_{ij}. Les problèmes où le nombre de processeurs est constant se notent en ajoutant le nombre de processeurs après le nom du modèle (ainsi, $Q2$ est le modèle où il y a deux processeurs reliés par un facteur de puissance, et Pm est le modèle où il y a un nombre arbitraire mais fixé de processeurs). Si un problème d'ordonnancement est mono-processeur, il est alors noté 1 dans la notation à trois champs.

Un ordonnancement de l'ensemble T sur M peut être défini de plusieurs façons. De manière générale, on définit un ordonnancement comme deux applications π, qui alloue une tâche i à un processeur $\pi(i)$, et σ, qui alloue une tâche i à une date de début d'exécution $\sigma(i)$. Pour qu'un ordonnancement soit valide, il doit au moins respecter les contraintes suivantes : chaque tâche est exécutée exactement une fois sans interruption, un processeur n'exécute qu'une seule tâche à la fois. On notera $C_i = \sigma(i)+p_{i\pi(i)}$ la date de terminaison de la tâche i.

Il est fréquent que des contraintes supplémentaires soient ajoutées au problème d'ordonnancement de base. La plus classique étant d'ajouter des contraintes de précédences données par un graphe $G = (T, E)$ (noté *prec* dans la notation à trois champs). Un arc $(i_1, i_2) \in E$ reliant i_1 à i_2 indique que la tâche i_1 doit être terminée avant que la tâche i_2 ne commence. Il existe des modèles avec des délais de communication ou des temps de communication qui représentent le temps nécessaire pour que le résultat d'une tâche soit communiquée à une autre allouée à un autre processeur. Ce modèle sera explicité quand il sera utilisé. Nous noterons $\Gamma^+(i)$ l'ensemble des successeurs de la tâche i et $\Gamma^-(i)$ l'ensemble de ses prédécesseurs.

D'autres contraintes apparaissent classiquement comme la présence de dates de disponibilités (*release date*) r_i avant laquelle la tâche i ne peut pas commencer, ou de dates d'échéances (*deadline*) d_i avant laquelle la tâche i doit être terminée.

Finalement, les problèmes d'ordonnancement sont associés à une fonction à optimiser. Les critères les plus classiques sont des fonctions des dates de terminaisons. La date de terminaison d'un ordonnancement (ou *makespan*) est $C_{max} = \max C_i$. La somme des temps de terminaison $\sum C_i$ est souvent utilisée parce qu'elle est directement reliée au temps moyen de terminaison (les deux sont même parfois confondus dans la littérature). Le flot d'une tâche (ou *flowtime*) est le temps passé par la tâche dans le système $F_i = C_i - r_i$. Les fonctions objectif sur le flot maximum F_{max} et la somme des flots $\sum F_i$ sont alors définies. Il est également classique de considérer des pondérations des tâches lors de la définition de la fonction objectif. On définit alors la somme des temps de terminaison pondérés (*sum of weighted completion times*) $\sum \omega_i C_i$ et la somme des flots pondérés $\sum \omega_i F_i$. Tous ces objectifs

sont à minimiser. De façon général, si f est une fonction à minimiser, nous noterons la plus petite valeur atteignable de f par $f^* = min_{\pi,\sigma} f(\pi, \sigma)$.

Remarquons que nous n'utilisons ici qu'un modèle de tâches séquentielles. Il existe d'autres modèles où les tâches sont multiprocesseurs où on distingue : les tâches rigides (ou *rigid tasks*) sont des tâches parallèles qui occupent un nombre fixe de processeurs. Les tâches modelables (ou *moldable tasks*) s'exécutent en parallèle sur un nombre de processeurs constant choisi par l'ordonnanceur. Les tâches malléables (ou *malleable tasks*) s'exécutent en parallèle sur un nombre de processeurs qui peut varier au cours du temps. Dans certains modèles, il est possible d'arrêter l'exécution d'une tâche pour la reprendre plus tard. Ces problèmes, usuellement plus simples que les modèles classiques, sont appelés problèmes d'ordonnancement avec préemption.

2.2 Résultats classiques en ordonnancement à un objectif

Dans cette section, nous donnons un résumé de quelques résultats et techniques classiques en ordonnancement mono-objectif. Nous discuterons de la complexité des problèmes aussi bien que de leur approximabilité. Des informations sur la théorie de la complexité peuvent être trouvées dans [GJ79]. Toutes les bases de la théorie de l'approximabilité peuvent être trouvées dans les livres [Hoc97] et [ACG+03].

2.2.1 Généralités

Tout d'abord, intéressons nous à la complexité des problèmes d'ordonnancement. Il existe des réductions classiques sur chacun des champs de la notation à trois champs. Les généralisations de problèmes sont toujours plus difficiles. Il est évident que le modèle R est plus difficile que le modèle Q. Avoir un nombre arbitraire de processeurs est toujours plus difficile qu'un nombre fixé de processeurs. Ajouter des contraintes de précédences arbitraires, des dates de disponibilités ou d'échéances complexifie usuellement les problèmes (la taille de l'instance change d'un polynôme et le nouveau problème est un cas plus général). Les objectifs avec pondération comme $\sum \omega_i C_i$ sont toujours plus difficiles à optimiser que leur contrepartie non pondérée. Un ouvrage de référence sur la complexité des problèmes d'ordonnancement a été écrit par Brucker [Bru95].

Optimiser le makespan est un problème généralement trivial sur une seule machine ; il suffit d'ordonnancer toutes les tâches les unes à la suite des

autres. L'ajout de dates de mise à disposition ou bien de dates d'échéances ne rendent pas le problème plus difficile. Leur ajout simultané rend le problème $\mathcal{NP}$-difficile au sens fort. En effet, ces deux contraintes peuvent forcer le placement de certaines tâches de façon à partitionner le temps en de nombreux intervalles. 3-PARTITION se réduit alors à $1 \mid r_i, d_i \mid C_{max}$ [GJ79].

Avec 2 processeurs identiques, ordonnancer des tâches indépendantes est un problème $\mathcal{NP}$-difficile au sens ordinaire. En effet, $P2 \mid\mid C_{max}$ est strictement équivalent à 2-PARTITION. Avec un nombre quelconque fixé de processeurs, le problème reste $\mathcal{NP}$-difficile au sens ordinaire. En revanche, si le nombre de processeurs fait parti de l'instance, alors le problème devient $\mathcal{NP}$-difficile au sens fort par réduction directe de 3-PARTITION.

Ces informations de complexité ne nous renseignent que sur la difficulté des problèmes de décision : il est difficile de trouver la valeur exacte du meilleur makespan dans les modèles qui nous intéressent. Cependant, la théorie de la complexité ne nous apporte que peu d'informations sur la difficulté de trouver un ordonnancement de makespan raisonnable. La théorie de l'approximabilité, en revanche, nous permet de construire de tels résultats. Un algorithme est une ρ-approximation d'un objectif f (sans perte de généralité, nous supposons que les objectifs sont à minimiser), si la solution S qu'il retourne est telle que $f(S) \leqslant \rho f^*$. On dit également que l'algorithme à un rapport (ou facteur) d'approximation de ρ pour la fonction f. Plus le rapport est proche de 1 et plus l'approximation est bonne. On dit qu'un problème appartient à $\mathcal{APX}$ s'il existe un algorithme polynômial ρ-approché avec ρ constant. Si ρ peut être aussi proche de 1 que l'on souhaite (ρ est de la forme $1+\epsilon, \epsilon > 0$), le problème appartient à $\mathcal{PTAS}$ (ou *Polynomial Time Approximation Scheme*) et la famille d'algorithmes qui les génère est une $\mathcal{PTAS}$. Si l'algorithme est en plus polynômial en $\frac{1}{\epsilon}$, la famille d'algorithmes est une $\mathcal{FPTAS}$ (ou *Fully Polynomail Time Approximation Scheme*) et le problème appartient à la classe du même nom. Certains résultats liant la complexité et l'approximabilité existent. Le plus important est que si un problème de décision est $\mathcal{NP}$-complet au sens fort alors sa version d'optimisation n'est pas dans $\mathcal{FPTAS}$ (si $\mathcal{P} \neq \mathcal{NP}$).

Il existe d'autres façons d'analyser l'approximation en bornant l'erreur absolue commise par l'algorithme ou encore en bornant son erreur relative. Ces pistes ne seront pas explorées dans ce manuscrit.

2.2.2 Du problème $P \mid\mid C_{max}$

Commençons par le modèle processeurs identiques avec des tâches indépendantes. Ce cas peut être traité avec l'algorithme *List Scheduling* proposé par Graham [Gra66]. Cet algorithme glouton considère les tâches dans un

ordre particulier, nous supposerons sans perte de généralité que les tâches sont triées suivant cet ordre. A son tour, la tâche i est ordonnancée au plus tôt sur le processeur qui exécute le moins de calcul (c'est à dire la somme des temps de calcul des tâches allouées sur ce processeur). Cet algorithme génère la solution LS qui est une 2-approximation pour le makespan. Nous rappelons la démonstration du rapport d'approximation en mettant en évidence une propriété des temps de terminaison des tâches dans LS.

PROPOSITION 2.1. *Si les tâches sont triées suivant l'ordre de l'algorithme, nous avons* $\forall i, C_i(LS) \leqslant \frac{\sum_{i' \leqslant i} p_{i'}}{m} + (1 - \frac{1}{m})p_i$

Démonstration. Lorsque la tâche i commence son exécution, tous les processeurs exécutent des tâches pendant au moins $\sigma(i)$ unités de temps. En effet, la tâche est exécutée par *List Scheduling* sur le processeur le moins chargé. Remarquons que $\sigma(i) \leqslant \sum_{i'<i} \frac{p_{i'}}{m}$ et cette valeur est atteinte si tous les processeurs sont exactement chargés de la même façon quand le tour de i arrive. La tâche i termine son exécution à la date $C_i = \sigma(i) + p_i \leqslant \frac{\sum_{i'<i} p_{i'}}{m} + p_i = \frac{\sum_{i' \leqslant i} p_{i'}}{m} + (1 - \frac{1}{m})p_i$. □

THÉORÈME 2.2. $C_{max}(LS) \leqslant (2 - \frac{1}{m})C^*_{max}$

Démonstration. La proposition précédente est vérifié pour toutes les tâches et en particulier pour la tâche i qui vérifie $C_i(LS) = C_{max}(LS) \leqslant \frac{\sum_{i' \leqslant i} p_{i'}}{m} + (1 - \frac{1}{m})p_i$. Remarquons que p_i est une borne inférieure du makespan optimal car dans un ordonnancement optimal, la tâche i est exécutée. $\frac{\sum_{i' \leqslant i} p_{i'}}{m}$ est également une borne inférieure du makespan optimal, car un ordonnancement doit exécuter toutes les tâches i', ce qui dans un ordonnancement préemptif (une relaxation de notre problème) se fait en exactement $\frac{\sum_{i' \leqslant i} p_{i'}}{m}$ unités de temps. Nous obtenons $C_{max}(LS) = C_i(LS) \leqslant (2 - \frac{1}{m})C^*_{max}$. □

Ce théorème montre qu'il y a au plus un facteur $2 - \frac{1}{m}$ entre la solution de *List Scheduling* et la solution optimale. Un contre exemple montre que la borne est atteinte pour certaines instances. L'instance est composée de m processeurs et de $m(m-1)+1$ tâches. Les $m(m-1)$ premières ont un temps de calcul $p_i = \frac{1}{m}(\forall i \in \{1, \ldots, m(m-1)\})$ et la dernière a pour temps de calcul $p_{m(m-1)+1} = 1$. L'ordonnancement optimal a un makespan $C^*_{max} = 1$. Tandis que, si *List Scheduling* est exécuté avec un ordre où la tâche $m(m-1)+1$ est considérée en dernier alors le makespan de *List Scheduling* est $C_{max}(LS) = 1 + \frac{m(m-1)}{m}$. Le rapport entre les deux est de $2 - \frac{1}{m}$.

La preuve de la borne de *List Scheduling* et le contre exemple qui montre que la borne est atteinte utilisent tous les deux une grande tâche pour terminer l'ordonnancement, il devient alors logique de considérer un cas particulier

de *List Scheduling*, celui où les tâches sont ordonnées par p_i décroissant. Cet algorithme est nommé LPT (pour *Largest Processing Time*) à un rapport de performance de $\frac{4}{3} - \frac{1}{3m}$ pour le problème $P \mid\mid C_{max}$ [Gra69]. Il est intéressant de remarquer que la preuve de ce rapport de performance utilise une analyse de cas pour les situations où il y a moins de $3m$ tâches et ainsi considère des bornes sur le makespan optimal meilleures que celles de l'analyse générale de Graham.

Une variante du problème a été considérée dans laquelle les processeurs ne sont pas tous disponibles depuis le temps 0 mais deviennent disponible au cours du temps. L'algorithme LPT est une $\frac{3}{2}$-approximation de ce problème. Un algorithme MLPT (pour *Modified LPT*) restaure le rapport d'approximation de $\frac{4}{3}$ [Lee91]. Sur ce problème, un algorithme $\frac{5}{4}$-approché a été proposé par Kellerer [Kel98] par une technique plus sophistiquée.

Pour finir avec le problème $P \mid\mid C_{max}$, un algorithme a été proposé par Hochbaum et Shmoys [HS87] qui permet d'avoir un rapport d'approximation de $(1+\epsilon)$ pour tout ϵ positif. La contre partie de cet algorithme est qu'il contient un terme $m^{\frac{1}{\epsilon}}$. C'est donc une $\mathcal{PTAS}$ pour le problème $P \mid\mid C_{max}$ et une $\mathcal{FPTAS}$ pour le problème $Pm \mid\mid C_{max}$.

2.2.3 Du makespan dans d'autres modèles

Dans le cas du modèle Q, deux algorithmes sont important à noter. Le premier est LPT-EFT pour *Largest Processing Time - Earliest Finish Time* qui consiste à ordonnancer de façon gloutonne les tâches dans l'ordre de temps de calcul décroissant sur le processeur qui les terminent au plus tôt. Remarquons que commencer au plus tôt ou finir au plus tôt sont rigoureusement équivalents dans le modèle P mais pas dans le modèle Q. LPT-EFT est donc le pendant direct de LPT pour le modèle Q et a un rapport de performance de 2 [GIS77]. Pour le modèle Q, la $\mathcal{PTAS}$ proposée par Hochbaum et Shmoys pour le modèle P a été étendue [HS88].

Le cas du modèle R est par contre très différent. Lenstra, Shmoys et Tardos fournissent deux résultats sur l'approximabilité du problème. Le premier est qu'il n'existe pas d'algorithme d'approximation polynômial ayant un rapport inférieur ou égal à $\frac{3}{2}$ si $\mathcal{P} \neq \mathcal{NP}$ [LST90]. Le deuxième est un algorithme 2-approché qui utilise l'approximation duale. Ce résultat est détaillé ci dessous :

Considérons que l'on connaît le makespan optimal G. Le problème est traité avec des outils de programmation linéaire. La variable x_{ij} vaut 1 si la tâche i est ordonnancée sur le processeur j et 0 sinon. $M(i) = \{j, p_{ij} \leqslant G\}$ est l'ensemble des processeurs qui peuvent ordonnancer la tâche i en moins

de G unités de temps. Symétriquement, $T(j) = \{i, p_{ij} \leqslant G\}$ est l'ensemble des tâches que le processeur j peut exécuter en moins de G unités de temps. On peut maintenant définir les contraintes linéaires suivantes

$$\begin{array}{rcl} \forall i, j, & x_{ij} & \geqslant 0 \\ \forall j, & \sum_{i \in T(j)} x_{ij} p_{ij} & \leqslant G \\ \forall i, & \sum_{j \in M(i)} x_{ij} & = 1 \end{array}$$

Il est possible à l'aide d'un outil de programmation linéaire de trouver une solution de base vérifiant ces contraintes. Cette solution de base n'est pas entière dans le cas général, mais elle a la propriété d'avoir un makespan inférieur à G. Une étude de ces contraintes montre que si G est un makespan réalisable, il est possible d'allouer chaque tâche i à un processeur différent de $M(i)$ (sinon, la solution retournée est invalide). Ainsi chaque processeur termine avant la date $2G$.

La méthode suppose que l'on connaisse la valeur du makespan optimal. Cette hypothèse n'est pas restrictive car il est possible de trouver la plus petite valeur de G qui rend une solution valide. Une telle valeur est inférieure ou égal à C^*_{max}. Cela prouve que l'algorithme est une 2-approximation. Les détails de l'algorithme peuvent être trouvé dans [Hal97]

2.2.4 Du makespan avec contraintes de précédence

L'optimisation du makespan avec contraintes de précédence a également été considérée. Il a été montré qu'il n'existe pas d'algorithme de rapport inférieur à $\frac{4}{3}$[LS95]. Dans le modèle de processeurs identiques P, l'algorithme List Scheduling peut être étendu avec contraintes de précédence sans dégrader son facteur d'approximation de $2 - \frac{1}{m}$ [Gra69]. Ce résultat est détaillé dans la Section 3.5.1 sur un problème proche. Dans le modèle Q, on ne connaît pas d'algorithme à facteur d'approximation constant même dans des cas restreint comme plusieurs chaînes indépendantes avec un nombre fixé de processeurs. Il est intéressant de noter que des algorithmes avec un rapport d'approximation dans $O(\log(m))$ ont été proposés (le plus récemment par Chekuri et Bender [CB01]). Aucun algorithme d'approximation intéressant n'est connu pour le modèle R.

2.2.5 De la somme des temps de terminaison

Sur une machine, la somme des temps de terminaison est optimisée de façon optimale par la règle SPT pour *Shortest Processing Time* qui ordonnance

les tâches en ordre non-décroissant de temps de calcul. Cet algorithme est encore optimal sur un nombre quelconque de processeurs identiques [Leu04b].

Lorsque les temps de terminaison sont pondérés, le problème d'optimisation est différent. Sur une machine la règle classique de Smith, qui consiste à trier les tâches par $\frac{p_i}{\omega_i}$ croissant, donne l'algorithme optimal WSPT (pour *Weighted Shortest Processing Time*) [Leu04b]. Sur un nombre arbitraire de machines, le problème est $\mathcal{NP}$-difficile et WSPT est une $(4 - \frac{1}{m})$-approximation [HSSW97].

2.3 Concepts de l'optimisation multi-objectif

Dans cette section, nous définissons formellement les concepts de l'optimisation multi-objectif. Il est impossible de fournir une définition simple de tous ces problèmes car la variété de problèmes qu'il faudrait couvrir rendrait la définition bien trop générale pour être utile. Remarquons comme précédemment que sans perte de généralité, nous pouvons supposer que toutes les fonctions objectif sont à minimiser.

DÉFINITION 2.3. ***Problème d'ordonnancement multi-objectif***
Soit T un ensemble de tâches à ordonnancer sur un ensemble M de processeurs.

Le problème d'ordonnancement multi-objectif *consiste à déterminer deux applications π et σ qui respectent des contraintes et qui minimisent les fonctions :*

$$f_1(\pi, \sigma), \ldots, f_k(\pi, \sigma)$$

Nous illustrerons les définitions de cette section sur une instance particulière du problème MaxAndSum [SW97]. Dans ce problème, n tâches séquentielles et indépendantes doivent être ordonnancées sur m processeurs identiques. Le problème d'optimisation est de minimiser simultanément le makespan C_{max} et la somme des temps de terminaisons $\sum C_i$. Dans la notation à trois champs généralisée aux problèmes multi-objectifs, il se note $P \mid\mid (C_{max}, \sum C_i)$.

EXEMPLE 2.4. *L'instance est composée de trois processeurs identiques et de neuf tâches. Les temps de calcul p_i sont donnés par la Table 2.1. Le problème est de fournir deux applications π et σ telles que chaque processeur exécute au plus une tâche à la fois :*

$$\forall i, i' \in T, \ \textit{si } \pi(i) = \pi(i') \textit{ alors } C_i \leqslant \sigma(i') \textit{ ou } C_{i'} \leqslant \sigma(i)$$

Le problème d'optimisation consiste à minimiser les deux fonctions objectif :

- $f_1(\pi, \sigma) = \max C_i = C_{max}$
- $f_2(\pi, \sigma) = \sum C_i$

TAB. 2.1 – Les temps de calcul des neuf tâches de notre instance

Numéro de la tâche	1	2	3	4	5	6	7	8	9
Temps de calcul	1	2	4	8	16	32	64	128	128

À chaque ordonnancement correspond un k-uplet de valeurs des fonctions objectif. Dans tout le manuscrit, nous remplacerons implicitement les ordonnancements par les k-uplets leur correspondant. Inversement, s'il existe des ordonnancements pour un k-uplet donné, nous utiliserons indifféremment le k-uplet ou l'un des ordonnancements correspondant. Ainsi, les ordonnancements présentés dans la Figure 2.1 peuvent être construits par LPT, SPT et LPT-SPT ou par les valeurs des fonctions objectif, respectivement $(128, 1025)$, $(164, 453)$ et $(128, 503)$.

En optimisation classique, la notion de meilleure solution est donnée par la relation d'ordre totale induite par la fonction objectif. Ceci n'est évidement pas utilisable directement en optimisation multi-objectif. La notion de **Pareto-dominance** [Voo03] est la relation d'ordre partielle qui formalise la notion de meilleure solution. La solution S Pareto-domine la solution S' si S est au moins aussi efficace que S' sur tous les objectifs et meilleure sur au moins l'un d'entre eux. Plus formellement :

DÉFINITION 2.5. ***Pareto-dominance***
Soient $S = (\pi, \sigma)$ et $S' = (\pi', \sigma')$ deux solutions d'un problème d'optimisation multi-objectif.

$$S \text{ Pareto-domine } S' \Leftrightarrow \forall l \in \{1, \ldots, k\}, f_l(\pi, \sigma) \leqslant f_l(\pi', \sigma') \\ \text{et } \exists l \in \{1, \ldots, k\} f_l(\pi, \sigma) < f_l(\pi', \sigma')$$

EXEMPLE 2.6. *Dans l'exemple précédent, l'ordonnancement LPT-SPT (Figure 2.1(c)) Pareto-domine l'ordonnancement LPT (Figure 2.1(a)).*

La Pareto-dominance est uniquement un ordre partiel. En effet, deux solutions sont dites **Pareto-indépendantes** lorsque aucune des deux ne Pareto-domine l'autre. La solution S^* est dite **Pareto-optimale** si elle n'est Pareto-dominée par aucune autre solution.

EXEMPLE 2.7. *Dans notre exemple, l'ordonnancement SPT (Figure 2.1(b)) et l'ordonnancement SPT-LPT (Figure 2.1(c)) sont Pareto-indépendants.*

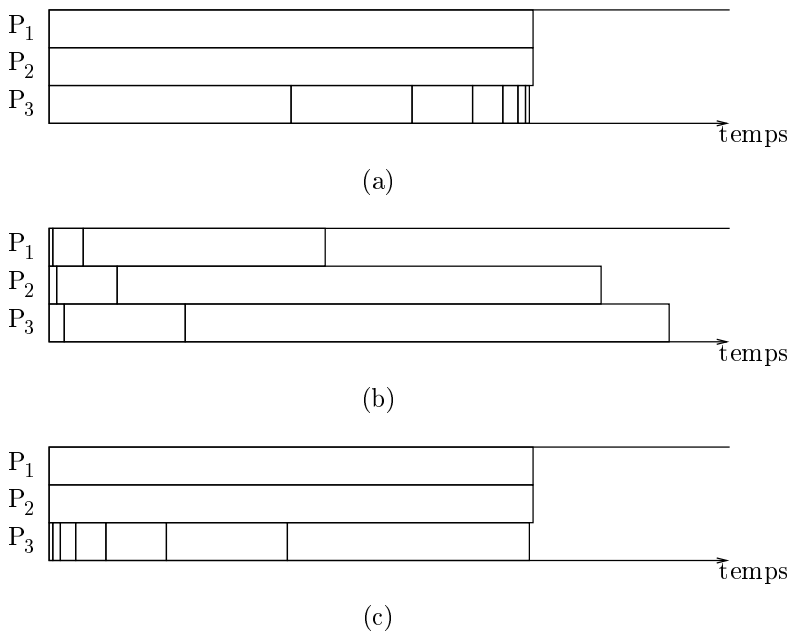


FIG. 2.1 – Quelques ordonnancement possibles obtenus par différents algorithmes présentés sous la forme de diagrammes de Gantt.(a) LPT (Plus grandes tâches d'abord), (b) SPT (Plus petites tâches d'abord) et (c) une version modifiée de LPT où les tâches sont allouées aux processeurs suivant LPT et réordonnées par processeur suivant SPT.

DÉFINITION 2.8. ***Pareto-optimalité***
Soit $S = (\pi, \sigma)$ une solution d'un problème d'optimisation multi-objectif. La solution S est Pareto-optimale si et seulement si :

$$\forall S' = (\pi', \sigma') \neq S, \exists l,\ f_l(\pi, \sigma) > f_l(\pi', \sigma') \Rightarrow \exists l',\ f'_l(\pi, \sigma) < f'_l(\pi', \sigma')$$

EXEMPLE 2.9. *L'ordonnancement LPT-SPT de valeur objectif* $(128, 503)$ *(Figure 2.1(c)) est Pareto-optimal car :*

- *Aucun ordonnancement ne peut avoir un meilleur makespan car la tâche la plus longue fait* 128 *unités de temps.*
- *Pour obtenir ce makespan, toutes les tâches de* 128 *unités de temps doivent être allouées sur des processeurs différents et aucune autre tâche ne doit être allouée sur ces processeurs. Toutes les autres tâches doivent être allouée sur le processeur restant.*
- *Les tâches sont allouées sur chaque processeur en ordre croissant de temps de calcul. Cette propriété implique l'optimalité pour $\sum C_i$ sur un unique processeur.*

Remarquons que toutes les solutions Pareto-optimales sont Pareto-indépendantes.

DÉFINITION 2.10. ***Ensemble de Pareto*** P
*L'***ensemble de Pareto** P *d'un problème est l'ensemble des solutions Pareto-optimales de cette instance.*

$$S \in P \Leftrightarrow S \text{ est Pareto-optimale}$$

L'ensemble de Pareto est parfois appelé courbe de Pareto (où *Pareto curve*). Cependant, en ordonnancement l'ensemble de Pareto est souvent discret et l'appellation "courbe" est impropre et ajoute de la confusion.

EXEMPLE 2.11. *L'ensemble de Pareto de notre instance d'illustration est représenté sur la Figure 2.2 par des diamants. Il est composé de huit points. Remarquons que tous les points Pareto-optimaux ne sont pas sur l'enveloppe convexe des solutions possibles. Pour un point particulier, nous marquons en gris l'ensemble des solutions qu'il domine (le rectangle en haut à droite) et l'ensemble des solutions qui le domine (le rectangle en bas à gauche). Il n'y a pas de solutions dans le rectangle qui le domine, cette solution est donc Pareto-optimale.*

À partir de toutes les solutions Pareto-optimales, nous déterminons les valeurs optimales de chaque fonction objectif indépendamment f_i^*. De ces valeurs, nous construisons la **solution zénith** qui est optimale sur tous les objectifs à la fois $z = (f_1^*, \ldots, f_k^*)$. Cette solution Pareto-domine toutes les solutions Pareto-optimales et est en général non réalisable. Cependant, de nombreuses analyses prennent z comme référence pour mesurer la qualité des solutions produites par un algorithme d'optimisation.

2.4 Résolution de problèmes multi-objectifs dans la littérature

Les analyses d'algorithmes les plus simples qui ont été proposées sont fondées sur des simulations ou des expériences dans le but d'estimer l'efficacité d'algorithmes mono-objectifs sur plusieurs objectifs à la fois (voir par exemple [CJ04]). L'intérêt principal de telles études est d'obtenir de l'information sur le comportement moyen des heuristiques sur un échantillon d'instances. Des études de ce type n'ont de sens que si l'échantillon est représentatif pour un système particulier. La composition d'un tel échantillon est en soi un problème difficile qui est partiellement traité par [Fei].

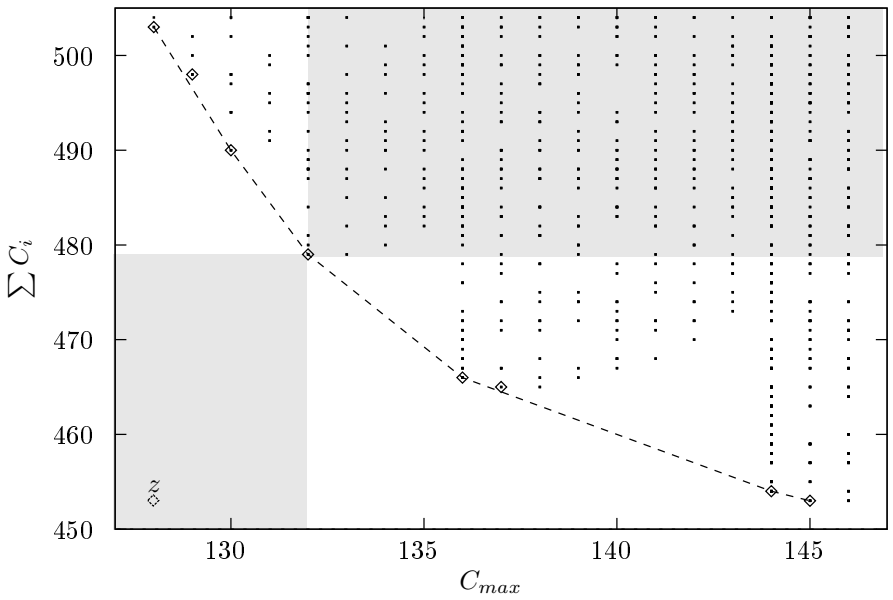


FIG. 2.2 – Quelques solutions réalisables de l'instance d'exemple utilisée dans cette section. Toutes les solutions réalisables de makespan inférieures à 146 et de $\sum C_i$ inférieures à 503 sont représentés par des points. Les huit solutions de l'ensemble de Pareto sont notifiées par des diamants. L'enveloppe convexe des solutions réalisables est représenté par un trait achuré. Remarquons que deux solutions Pareto-optimales ne sont pas sur l'enveloppe convexe. La solution zénith est dans le coin en bas à gauche et n'est pas réalisable.

Une approche plus élaborée consiste à agréger artificiellement les fonctions objectif en une seule fonction à optimiser, puis à réutiliser les techniques classiques d'optimisation mono-objectif pour optimiser la fonction agrégée. La première agrégation qui vient a l'esprit est d'optimiser la somme des objectifs ou encore leur maximum (voir [BS03, AGK04] pour de tels exemples). Cependant, si les domaines de définition des objectifs sont trop différents, il faut redimensionner les fonctions pour compenser le déséquilibre des valeurs. Le nombre de fonctions d'agrégats possibles est infini. De nombreuses fonctions ont un sens particulier et nécessitent d'être finement paramétrées pour convenir au problème considéré. Beaucoup de propriétés peuvent justifier l'utilisation de technique d'agrégation [Det00]. La plus classique étant la propriété de Pareto-optimalité de la solution qui optimise une agrégation. Cette propriété est valide pour toutes les combinaisons linéaires de fonctions objectif. Lorsque l'on représente l'espace des solutions comme un espace k-dimensionnel, certaines fonctions d'agrégation sont proches des normes Euclidiennes. Il y a cependant une limitation intrinsèque à l'agrégation par des fonctions linéaires :

l'ensemble de Pareto n'est pas toujours convexe, et donc certaines solutions Pareto-optimales pourraient ne pas être atteignables par cette technique. Par exemple sur la Figure 2.2, seulement six solutions des huit solutions sont sur l'enveloppe convexe (les solutions $(129, 498)$ et $(137, 465)$ n'y sont pas). Dans le pire des cas sur un problème bi-objectif, seulement les deux solutions extrêmes peuvent être obtenues, les autres solutions Pareto-optimales étant au dessus de l'enveloppe convexe. Notons cependant que souvent l'ensemble de Pareto est proche de l'enveloppe convexe et donc cette technique reste utile comme première approche. Dans la notation à trois champs la combinaison linéaire d'objectifs est noté par $\sum_l \alpha_l f_l$.

Une autre façon naturelle de transformer un problème multi-objectif en un problème mono-objectif consiste à traiter chaque objectif séparément, en hiérarchisant les fonctions objectif de la plus importante à la moins importante. Avec cette approche, un ordre total sur les solutions est fourni par l'ordre lexicographique des k-uplets les représentant. Une utilisation de cette technique peut être trouvé dans [Ho04]. La fonction objectif résultante est usuellement noté $Lex(f_1, \ldots, f_k)$.

Une autre possibilité est de transformer quelques objectifs (classiquement, tous sauf un) en contrainte. Une telle approche a été utilisé dans [Ste86]. La plupart du temps, le problème est résolu à l'aide de techniques de type metaheuristique ou programmation linéaire en nombres entiers dans lesquelles ajouter des contraintes est facile. Le principal défaut de cette technique est qu'elle ne simplifie pas vraiment le problème. Cependant, cette approche est naturelle dans bien des domaines. Par exemple, dans les systèmes temps-réels ou juste-à-temps, les contraintes temporelles émergent de l'application. Cette approche est aussi appelé technique par ϵ-contrainte (ou ϵ-*constraint*) et est notée par $\epsilon(f_u/f_1, \ldots, f_{u-1}, f_{u+1}, \ldots, f_k)$ dans la notation à trois champs.

Finalement, des approches directes visant une approximation de tous les objectifs à la fois ont été proposées. Cela revient à approcher la solution zénith du problème. Un exemple célèbre est celui de l'algorithme d'approximation du problème MaxAndSum proposé dans [SW97]. Dans la plupart des cas, un paramètre de compromis permet à l'utilisateur de choisir le rapport d'approximation. Ainsi, le responsable du système peut directement contrôler les dégradations sur chaque objectif et obtenir une solution qui convient au besoin de son application. Une autre approche directe revient à rechercher une approximation des solutions qui respecte des contraintes sur une partie des objectifs et qui optimise les autres objectifs. Par exemple, Shmoys et Tardös ont étudié le problème d'ordonnancer des tâches sur des processeurs hétérogènes non uniformes dans lequel allouer une tâche sur un processeur entraine un coût. Ils ont proposé un algorithme qui pour un makespan ω donné, fournit un ordonnancement de coût inférieur au coût du meilleur ordonnancement

de longueur inférieure à ω et qui dont le makespan est inférieur à 2ω [ST93]. Ces deux techniques sont plus longuement décrites en Section 2.6.2 et en Section 2.6.3.

Remarquons que tous les exemples précédents concernent l'optimisation de deux objectifs. Optimiser k (fixé) objectifs ne semble pas plus difficile (évidement, l'analyse sera plus technique). Cependant, ajouter un nouvel objectif peut empêcher d'obtenir une approximation de la solution zénith. Nous fournissons des références vers quelques travaux considérant des problèmes à trois objectifs [TB07, PY00, Laf01]. Un tel problème sera considéré au Chapitre 3.

2.5 Complexité

Lorsque l'on considère une classe de problème d'un aspect théorique, il est logique de se poser la question de leur complexité. Dans cette section, nous établissons un bref bilan des questions reliées à la théorie de la complexité par rapport aux problèmes multi-objectifs.

Remarquons d'abord que les problèmes qui nous intéressent sont des problèmes d'*optimisation* et non des problèmes de décision. Cependant, de la même façon que lorsque l'on considère des problèmes mono-objectifs, la version de décision d'un problème d'optimisation nous fournit de l'information sur la difficulté du problème. Il est évident que si le problème de décision associé à la version mono-objectif d'un problème multi-objectif est $\mathcal{NP}$-complet alors le problème de décision associé au problème d'optimisation multi-objectif est $\mathcal{NP}$-complet également. Cependant, un problème de décision multi-objectif peut être $\mathcal{NP}$-complet alors que les versions mono-objectifs sont toutes dans $\mathcal{P}$. Par exemple, calculer la longueur du plus court chemin dans un graphe est connu pour être polynômial. Cependant, si les arêtes du graphe sont bivaluées et que l'on associe deux distances à chaque chemin, alors décider s'il existe un chemin respectant une contrainte sur chaque distance est un problème $\mathcal{NP}$-complet (dans [GJ79] sous la référence ND30).

En pratique, le problème central en optimisation multi-objectif est de trouver un compromis entre toutes les solutions réalisables. Une réponse raisonnable consiste à fournir toutes les solutions Pareto-optimales. Trois questions différentes apparaissent alors. La première concerne le nombre de solutions Pareto-optimales. Ce nombre peut être exponentiel dans la taille de l'instance. Par exemple, dans le problème de plus court chemin bivalué, il peut y avoir autant de solutions Pareto-optimales que de chemins dans le graphe. Les problèmes dont la sortie est le nombre de solutions Pareto-

optimales sont appelés **problèmes de comptage** [TBE07]. La deuxième question est de fournir tout l'ensemble de Pareto d'une instance. Ces problèmes sont appelés **problèmes d'énumération** [TBE07]. Finalement, la dernière question concerne l'approximation de l'ensemble de Pareto par un nombre limité de solutions. Nous discutons maintenant ces trois points.

La classe de complexité $\#\mathcal{P}$ contient les problèmes de comptage qui peuvent être résolus en temps fini pour lesquels vérifier si une solution est valide peut être fait en temps polynômial [TB07]. Insistons sur le fait que si un problème est dans $\#\mathcal{P}$, il n'est pas forcément solvable en un temps borné par une fonction raisonnable de la taille de l'instance. En un sens, la classe $\#\mathcal{P}$ est aux problèmes de comptage ce que $\mathcal{NP}$ est aux problèmes de décision. Une sous-classe à temps de calcul borné par un polynôme de la taille de l'instance a été défini et appelé $\mathcal{FP}$.

Les problèmes de comptage ne sont intéressants que pour l'information qu'ils nous apportent sur la difficulté des problèmes d'énumération. La classe des problèmes d'optimisation dont la version de décision appartient à $\mathcal{NP}$, $\mathcal{NPO}$ est généralisé à $\mathcal{ENP}$ pour les problèmes d'énumération [TB07]. Un problème appartient à $\mathcal{ENP}$ s'il existe un algorithme qui vérifie qu'un ensemble de solutions est l'ensemble de Pareto de l'instance en temps borné par une fonction polynômiale en la taille de l'instance et dans le nombre de solutions Pareto-optimales. Un problème dans $\mathcal{ENP}$ qui peut être résolu en temps polynômial appartient à $\mathcal{EP}$.

Remarquons que de telles classes ne sont pas utiles en pratique. Le nombre de solutions Pareto-optimales n'est pas le critère qui sert à classer les problèmes. Par exemple, le problème de plus court chemin bivalué appartient à $\mathcal{EP}$, la classe la plus simple. Pourtant, l'énumération pourra prendre un temps exponentiel dans la taille de l'instance puisqu'il existe potentiellement un nombre exponentiel de solutions Pareto-optimales.

Plus de détails sur les classes de complexité pour les problèmes de comptage et d'énumération sont disponibles dans [TBE07].

Les classes sur l'approximabilité des problèmes ne sont pas bien formalisées dans le contexte de l'optimisation multi-objectif. Il est souvent possible de montrer que certaines solutions de compromis ne sont pas approximables à moins de certains facteurs. Il existe même des résultats d'inapproximabilités qui ne dépendent pas de conjecture de complexité (comme l'hypothèse $\mathcal{P} \neq \mathcal{NP}$) mais qui sont déduits de la non existence de solutions qui atteignent une valeur objectif donnée. A notre connaissance, personne n'a étudié les extensions des classes $\mathcal{APX}$ ou $\mathcal{PTAS}$ pour les problèmes multi-objectifs.

2.6 De l'approximation des problèmes multi-objectifs

2.6.1 Définition

Il existe deux principaux types de résultats en approximation multi-objectif. Le premier consiste à calculer une solution qui est une approximation d'une solution particulière du problème. Le deuxième consiste à calculer un ensemble de solutions approchant un autre ensemble de solutions.

Nous commençons par définir formellement les concepts clés de l'approximation multi-objectif en étendant le concept classique de ρ-approximation de l'optimisation mono-objectif.

DÉFINITION 2.12. *Soient S et S' deux solutions. S est une $\rho = (\rho_1, \ldots, \rho_k)$-approximation $(\forall l, \rho_l \geqslant 1)$ de S' si et seulement si $\forall l \in \{1, \ldots, k\}, f_l(S) \leqslant \rho_l f_l(S')$.*

On généralise la notion de ρ-approximation d'une solution aux algorithmes permettant de les obtenir. Un algorithme est une ρ-approximation de S si la solution qu'il génère est une ρ-approximation de S.

Cette définition est valide pour toutes les solutions. Calculer certaines solutions Pareto-optimales est parfois un problème difficile. Par exemple, on peut vouloir calculer une approximation de la solution optimisant $Lex(f_1, \ldots, f_k)$ car trouver cette dernière est difficile. Notons que la solution qui est approchée n'est pas obligatoirement réalisable. Ainsi, chercher à obtenir l'approximation de la solution zénith d'un problème a bien du sens. Un tel résultat est particulièrement puissant puisque tous les objectifs sont alors approchés indépendamment. On omet souvent de parler de la solution zénith lorsque l'on énonce ces résultats et une ρ-approximation du zénith d'un problème devient souvent une ρ-approximation du problème.

Cependant, l'approximation de la solution zénith n'est pas forcément suffisante pour les besoins applicatifs. On peut vouloir choisir une solution parmi les solutions Pareto-optimales. Cependant, l'ensemble de Pareto peut avoir une cardinalité exponentielle dans la taille de l'instance rendant l'énumération de cet ensemble impossible en temps raisonnable. On s'intéresse alors à approcher l'ensemble de Pareto par un nombre polynômial de solutions. Papadimitriou et Yannakakis [PY00] fournissent une définition de l'approximation de l'ensemble de Pareto et prouvent que si le problème voit la taille des objectifs bornée par un polynôme alors il existe une approximation de cardinalité raisonnable. La définition suivante est une version plus lisible mais strictement équivalente de la définition qu'ils proposent.

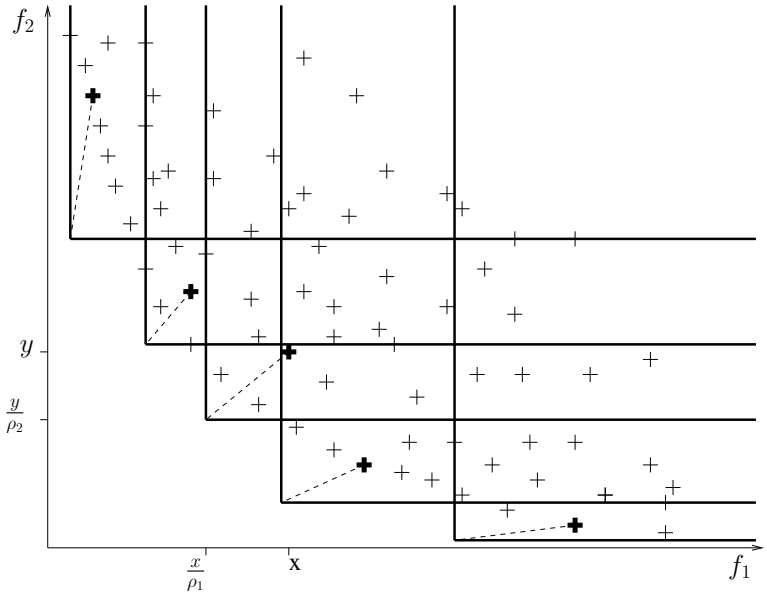


FIG. 2.3 – Les solutions représentées en gras forment une (ρ_1, ρ_2)-approximation de l'ensemble de Pareto.

DÉFINITION 2.13. *L'ensemble de solutions P est une $\rho = (\rho_1, \ldots, \rho_k)$-approximation $(\forall l, \rho_l \geqslant 1)$ de l'ensemble de solutions P' si et seulement si $\forall S' \in P', \exists S \in P, S$ est une ρ-approximation de S'.*

La Figure 2.3 illustre le concept sur un problème à deux objectifs. Les solutions sont représentées dans l'espace $(f_1; f_2)$. Les solutions en gras forment l'approximation de l'ensemble de Pareto. Chaque solution dans l'ensemble gras (ρ_1, ρ_2)-approche toutes les solutions du quart positif associé. Toutes les solutions du problème sont incluses dans au moins un des quarts positifs. Ainsi l'ensemble des croix en caractères gras est une (ρ_1, ρ_2)-approximation de l'ensemble de Pareto.

Le théorème suivant concerne l'existence d'un ensemble qui approche l'ensemble de Pareto des problèmes multi-objectifs. Il a été proposé par Papadimitriou et Yannakakis dans [PY00].

THÉORÈME 2.14. *Soit un problème d'optimisation multi-objectif tel que pour toute instance I, les valeurs des fonctions objectif sont comprises dans $[\frac{1}{2^{p(|I|)}}; 2^{p(|I|)}]$. Soit P l'ensemble de Pareto de I. Pour tout $\epsilon > 0$, il existe une $(1+\epsilon, \ldots, 1+\epsilon)$-approximation de l'ensemble de Pareto P^ϵ tel que $\mid P^\epsilon \mid$ est polynômial en $\mid I \mid$ et en $\frac{1}{\epsilon}$.*

Démonstration. Considérons l'espace k-dimensionnel des valeurs des fonctions objectifs. Les valeurs atteignables de chaque fonction objectif est partitionné en intervalles de la forme $[(1+\epsilon)^i; (1+\epsilon)^{i+1}], i \in \mathbb{Z}$. Il y a de l'ordre de $p(\mid I \mid)/\epsilon$ intervalles par objectif. L'espace k-dimensionnel est découpé en hyperrectangles suivant la partition des valeurs de chaque fonction. Le nombre d'hyperrectangles est alors dans $O(p(\mid I \mid)^k/\epsilon^k)$. Un ensemble de solutions composé d'une solution par hyperrectangles (si il y en a une) est alors une $(1+\epsilon)$-approximation de l'ensemble de Pareto. □

Il est intéressant de noter que la cardinalité de l'ensemble n'est pas nécessairement polynômiale dans le nombre d'objectifs. Remarquons également qu'une ρ-approximation de la solution zénith est une ρ approximation de l'ensemble de Pareto. Pour désigner l'approximation du zénith, on parle parfois d'approximation de l'ensemble de Pareto en un seul point.

2.6.2 Une Approximation du Zénith de MaxAndSum

Ce problème à été utilisé pour illustrer les définitions générales en Section 2.3. Rappelons que le problème consiste à ordonnancer n tâches indépendantes sur m processeurs de façon à optimiser simultanément le makespan C_{max} et la somme des temps de terminaison $\sum C_i$. Rappelons également que *List Scheduling* est une 2-approximation du makespan et que l'ordre particulier SPT est optimal pour $\sum C_i$. Le théorème suivant est donc immédiat.

THÉORÈME 2.15. *SPT est une* $(2,1)$*-approximation de* MaxAndSum.

Ce résultat ne tient pas dans des cas plus complexes comme par exemple des problèmes avec des tâches parallèles ou encore avec des contraintes de dates de disponibilité. De plus, il n'est pas évident de construire à partir de SPT, un algorithme ayant un meilleur rapport d'approximation sur le makespan. Dans le reste de cette section, nous présentons une technique qui a été proposé par Stein et Wein [SW97]. L'idée principale derrière cette technique est que la somme des temps de terminaison est optimisée en ayant une bonne allocation des tâches au début de l'ordonnancement et que c'est la fin de l'ordonnancement qui est importante pour obtenir un bon makespan.

Considérons que nous avons deux ordonnancements S_{cmax}, un ordonnancement ρ_{cmax}-approché pour le makespan et S_{minsum}, un ordonnancement ρ_{minsum}-approché pour la somme des temps de terminaison. La technique de Stein et Wein mélange ces deux ordonnancements sans trop dégrader les deux rapports.

Soit $l = \alpha C_{max}^{S_{cmax}}$ avec $\alpha > 0$. Les tâches sont partitionées en deux ensembles $T = T_{minsum} \cup T_{cmax}$ où $i \in T_{minsum}$ si $C_i(S_{minsum}) \leqslant l$ et $i \in T_{cmax}$

sinon. Un ordonnancement S est construit où les tâches de T_{minsum} sont ordonnancées entre les temps 0 et l suivant S_{minsum} (cette étape est la *truncation*). Ensuite, les tâches de T_{cmax} sont allouées aux processeurs où elles étaient ordonnancées à partir du temps l (cette étape est la *composition*).

PROPOSITION 2.16. *Le makespan de S est borné : $C_{max}(S) \leqslant (1+\alpha)\rho_{cmax}C^*_{max}$.*

Démonstration. Aucune tâche n'est en cours d'exécution au temps l : chaque tâche est soit terminée avant l ou commence après (par construction). Les tâches, qui sont après l, sont ordonnancées suivant S_{cmax}. Elles finissent donc avant $l + C_{max}(S_{cmax})$. Rappelons que $l = \alpha C_{max}(S_{cmax})$ et que S_{cmax} est une ρ_{cmax}-approximation du makespan optimal. D'où $C_{max}(S) \leqslant (1+\alpha)\rho_{cmax}C^*_{max}$. □

La borne sur le makespan de S permet de borner la dégradation en somme des temps de terminaison induite par les tâches de T_{cmax}.

PROPOSITION 2.17. *La somme des temps de terminaison de S est bornée : $\sum C_i(S) \leqslant \frac{1+\alpha}{\alpha}\rho_{minsum}\sum C^*_i$.*

Démonstration. La preuve est obtenue en bornant le rapport $\frac{C_i(S)}{C_i^{S_{minsum}}}$ pour chaque tâche i. Soit $i \in T_{minsum}$ et le rapport est exactement 1. Soit $i \in T_{cmax}$ et le rapport $\frac{C_i(S)}{C_i(S_{minsum})} \leqslant \frac{(1+\alpha)C_{max}(S_{cmax})}{\alpha C_{max}(S_{cmax})} = \frac{1+\alpha}{\alpha}$. En effet, i termine avant $l + C_{max}(S_{cmax})$ dans S mais après l dans S_{minsum}. En se rappelant que S_{minsum} est une ρ_{minsum}-approximation de la somme des temps de terminaison, on en déduit : $\sum C_i(S) \leqslant \sum \frac{1+\alpha}{\alpha}C_i(S_{minsum}) \leqslant \frac{1+\alpha}{\alpha}\rho_{minsum}\sum C^*_i$. □

Ces deux propriétés conduisent à l'approximation du zénith de MaxAndSum par S :

COROLLAIRE 2.18. *S est une $((1+\alpha)\rho_{cmax}, \frac{1+\alpha}{\alpha}\rho_{minsum})$-approximation de la solution zénith de* MaxAndSum.

Considérons que S_{cmax} et S_{minsum} sont des solutions optimales pour le makespan et la somme des temps de terminaison. Alors, S est une $(1+\alpha, \frac{1+\alpha}{\alpha})$-approximation de la solution zénith. Ceci prouve qu'il existe une telle solution valide pour ce problème. En sélectionnant le point de découpe l plus finement, il est possible d'améliorer le résultat en obtenant une $(1+\rho, \frac{e^\rho}{e^\rho-1})$-approximation du zénith ($\forall\rho \in [0;1]$) [ARSY99]. Plus récemment, Rasala *et al.* ont montré que des variantes avec des dates de disponibilité n'admettent pas d'algorithme ayant un meilleur rapport d'approximation car un tel ordonnancement peut ne pas exister [RSTU02].

La technique de Stein et Wein est générique et a été étendue à d'autres fonctions objectif comme la somme des temps de terminaison pondérés ou le retard maximal [RSTU02]. Elle peut aussi être étendue à des contraintes de précédences. Cette technique repose sur la connaissance de deux ordonnancements qui optimisent chacun un des objectifs. L'idée peut être étendue en ayant deux politiques d'ordonnancement différentes où l'une des deux utilise dynamiquement l'autre pour remplir certains intervalles de temps. *Recursive doubling* [SWW95] exploite cette idée et a été adapté pour des tâches modelables [DEMT04].

Cependant, la technique de Stein et Wein ne peut directement être étendue que si les deux opérations élémentaires de *truncation* et de *composition* génèrent des ordonnancements valides. Ce qui n'est pas le cas pour des problèmes avec dates d'échéances où la *composition* pourrait retarder une date après son échéance.

Mélanger des solutions n'est pas la seule façon d'obtenir une approximation de la solution zénith. Sur un problème tout à fait différent, Skutella [Sku98] résout un problème à l'aide d'un programme linéaire paramétré par une valeur α qui produit une famille de solutions de rapport d'approximation $(1+\alpha, 1+\frac{1}{\alpha})$.

2.6.3 Approximation de l'ensemble de Pareto de BiSumCi

Dans le problème BiSumCi [ABG05], n tâches sont à ordonnancer sur un unique processeur. Le problème d'optimisation est composé de deux fonctions de type somme des temps de terminaison pondérés : $\sum \omega_i C_i$ et $\sum x_i C_i$. Ce problème se note : $1 \mid\mid (\sum \omega_i C_i, \sum x_i C_i)$. Nous présentons dans cette section une technique d'approximation de l'ensemble de Pareto utilisant une approche par contraintes. Un objectif est transformé en une contrainte de la même façon que dans une technique d'approximation duale. Cela est équivalent à l'approximation de la solution optimale du problème ϵ-contraint.

Notation

Nous nous intéressons aux algorithmes qui prennent en paramètre une borne G sur un objectif et construisent une ρ_2 approximation sur l'autre objectif. La solution peut dépasser la borne G mais ne doit pas la dépasser plus qu'un facteur ρ_1. Dans la littérature, ces algorithmes sont souvent appelés (ρ_1, ρ_2)-approximations. Cependant cette notation porte à confusion car, d'une part on risque de confondre l'algorithme avec une approximation de la solution zénith, et d'autre part la notation est symétrique alors que sa

signification ne l'est pas (la borne est appliquée sur le premier objectif et pas sur le deuxième).

Dans un but de clarifier la notion d'approximation dans ce contexte, nous avons proposé de dire d'un algorithme avec la propriété précédente que c'est une $\langle \bar{\rho}_1, \rho_2 \rangle$-approximation [JST08]. Formellement sur notre problème,

DÉFINITION 2.19. *A partir d'un seuil G, un algorithme $\langle \bar{\rho}_1, \rho_2 \rangle$-approché retourne une solution tel que $\sum \omega_i C_i \leqslant \rho_1 G$ et $\sum x_i C_i \leqslant \rho_2 \sum x_i C_i^*(G)$ où $\sum x_i C_i^*(G)$ est la valeur optimale pour $\sum x_i C_i$ dans les ordonnancements où $\sum \omega_i C_i \leqslant G$. L'algorithme ne retourne pas de solution seulement dans des cas où il n'existe pas d'ordonnancement tel que $\sum \omega_i C_i \leqslant G$.*

Une $\langle \bar{2}, 2 \rangle$-approximation

Le point important de l'algorithme *LPorder* (proposé dans [ABG05]) réside dans l'analyse de la solution construite par un programme linéaire qui construit une solution non réalisable de somme des temps de terminaison pondérés bornée $\sum w_i C_i \leqslant G$. Soit y_i la variable positive qui donne la date de terminaison de la tâche i. Il est facile d'écrire avec ces variables la fonction objectif : $\min \sum x_i y_i$ ainsi que la contrainte liée au second objectif : $\sum w_i y_i \leqslant G$.

La description des ordonnancements valides est plus compliquée. La contrainte choisie assure que la somme des $\sum p_i C_i$ dans l'ordonnancement est supérieure à la valeur optimale pour ce critère. La règle de Smith nous dit que la solution optimale pour $\sum p_i C_i$ est obtenue en triant les tâches par $\frac{p_i}{p_i} = 1$ croissant. Donc, tous les ordonnancements sans temps d'inactivité sont optimaux. Dans un tel ordonnancement, nous avons $\sum_{i=1}^{n} p_i C_i = \sum_{i=1}^{n} p_i (\sum_{i'=1}^{i} p_{i'}) = \sum_{i=1}^{n} \sum_{i'=1}^{i} p_i p_{i'}$. Cette expression vient de l'ordonnancement particulier où les tâches sont ordonnancées dans l'ordre $1, \ldots, n$. Cette valeur optimale pour $\sum p_i C_i$ indique que quel que soit l'ordonnancement valide que l'on considère, la contrainte $\sum_i p_i y_i \geqslant \sum_{i=1}^{n} \sum_{i'=1}^{i} p_i p_{i'}$ doit être vérifiée. Cette contrainte est également valide pour tout sous-ensemble A de tâches. Elle peut être écrite de la manière suivante : $\sum_{i \in A} p_i y_i \geqslant \frac{1}{2}(\sum_{i \in A} p_i^2 + (\sum_{i \in A} p_i)^2), \forall A \subseteq \{1, \ldots, n\}$.

La valeur objectif de la solution optimale du programme linéaire est une borne inférieure de $\sum x_i C_i^*(G)$, la valeur minimale de $\sum x_i C_i$ dans les ordonnancements tels que $\sum \omega_i C_i \leqslant G$. Le programme linéaire résultant dispose d'un nombre exponentiel de contraintes (à cause des contraintes sur tous les sous-ensembles de tâches). Cependant, selon les auteurs de [ABG05], il peut être résolu en temps polynômial. La solution obtenue n'est pas réalisable, dans le sens où il peut ne pas y avoir d'ordonnancement qui obtienne ces temps de terminaison. Dans la solution S générée par *LPorder*, les tâches

sont ordonnancées en ordre non-décroissant de y_i. On peut sans perte de généralité réordonner les tâches dans l'ordre de S. Ainsi $C_i = \sum_{i'=1}^{i} p_{i'}$.

THÉORÈME 2.20. *LPorder est une* $\langle \bar{2}, 2 \rangle$*-approximation de* BiSumCi.

Démonstration. Pour chaque tâche i, nous nous intéressons au produit : $y_i \sum_{i'=1}^{i} p_{i'}$. Les tâches sont triées par y_i croissant. D'où : $y_i \sum_{i'=1}^{i} p_{i'} \geqslant \sum_{i'=1}^{j} p_{i'} y_{i'}$. La contrainte sur le bon sous-ensemble de tâches nous amène à $y_i \sum_{i'=1}^{i} p_{i'} \geqslant \frac{1}{2}(\sum_{i'=1}^{i} p_{i'})^2$. Cette expression se réduit à $y_i \geqslant \frac{1}{2} \sum_{i'=1}^{i} p_{i'}$. En se rappelant que $C_i = \sum_{i'=1}^{i} p_{i'}$, on obtient finalement, $\forall i, C_i \leqslant 2y_i$. Nous en déduisons pour le premier objectif : $\sum \omega_i C_i \leqslant 2 \sum \omega_i y_i \leqslant 2G$ et pour le second : $\sum x_i C_i \leqslant 2 \sum x_i y_i \leqslant 2 \sum x_i C_i^*(G)$. □

Mécanisme

L'algorithme présenté en Algorithme 2.1 construit une approximation de l'ensemble de Pareto en appliquant l'algorithme $\langle \bar{2}, 2 \rangle$-approché sur les valeurs d'une suite géométrique de $\sum \omega_i C_i$.

L'ensemble des valeurs efficaces de $\sum \omega_i C_i$ est partitionné en tranches de tailles exponentiellement croissantes. La $i$ème tranche contient les solutions dont les $\sum \omega_i C_i$ sont dans $[(1 + \frac{\epsilon}{2})^{i-1} \sum \omega_i C_i^{min}; (1 + \frac{\epsilon}{2})^i \sum \omega_i C_i^{min}]$ où $\sum \omega_i C_i^{min}$ est une borne inférieure au $\sum \omega_i C_i$ optimal. Similairement, on définit $\sum \omega_i C_i^{max}$ comme une borne supérieure aux valeurs de $\sum \omega_i C_i$ raisonnables. Cette valeur sera explicitée plus tard. Pour chaque tranche i, l'algorithme calcule une solution S_i $LPorder$. Nous montrons dans le Théorème 2.21 que S_i approche toutes les solutions de la tranche i avec un facteur $(2 + \epsilon, 2)$.

Algorithme 2.1 Approximation de l'ensemble de Pareto

Input :ϵ un réel positif
Output :P un ensemble de solutions
Begin
  $P = \emptyset$
  For all $i \in \left\{1, 2, \ldots, \left\lceil \log_{1+\frac{\epsilon}{2}} \left(\frac{\sum \omega_i C_i^{max}}{\sum \omega_i C_i^{min}} \right) \right\rceil \right\}$
    $G_i = (1 + \frac{\epsilon}{2})^i \sum \omega_i C_i^{min}$
    $S_i = LPorder(G_i)$
    $P = P \cup S_i$
  Return(P)
End

Les valeurs de $\sum \omega_i C_i$ efficaces sont bornées par les deux expressions suivantes. La borne inférieure $\sum \omega_i C_i^{min} = \sum \omega_i$ est obtenue en considérant l'ordonnancement où toutes les tâches finissent au temps 1, cet ordonnancement est irréalisable et est une borne très large de $\sum \omega_i C_i^*$. La borne supérieure $\sum \omega_i C_i^{max} = \sum \omega_i \sum p_i$ est obtenue en considérant que toutes les tâches finissent au temps $\sum p_i$; tout ordonnancement compact est meilleur que celui ci. En utilisant des tranches de tailles exponentiellement croissantes, il faut $\left\lceil \log_{1+\frac{\epsilon}{2}} \left(\frac{\sum \omega_i C_i^{max}}{\sum \omega_i C_i^{min}} \right) \right\rceil$ étapes à l'algorithme. Remarquons que la cardinalité de l'ensemble généré est polynômiale : l'algorithme génère moins de $\left\lceil \log_{1+\frac{\epsilon}{2}} \frac{\sum \omega_i C_i^{max}}{\sum \omega_i C_i^{min}} \right\rceil \leqslant \left\lceil \log_{1+\frac{\epsilon}{2}} \sum p_i \right\rceil$ solutions, ce qui est polynômial en $\frac{1}{\epsilon}$ et dans la taille de l'instance.

Les bornes $\sum \omega_i C_i^{min}$ et $\sum \omega_i C_i^{max}$ sont données ici en formes closes que pour confirmer la polynômialité du nombre d'itérations de l'algorithme. En pratique, il est possible de fixer les valeurs de $\sum \omega_i C_i^{min}$ et $\sum \omega_i C_i^{max}$ à l'aide des valeurs de $\sum \omega_i C_i$ des ordonnancements optimaux pour $\sum \omega_i C_i$ et $\sum x_i C_i$.

Nous montrons maintenant que l'ensemble de solutions P générées par l'algorithme présenté en Algorithme 2.1 approche l'ensemble de Pareto avec un facteur $(2+\epsilon, 2)$.

THÉORÈME 2.21. *L'algorithme 2.1 est une* $(2+\epsilon, 2)$*-approximation de l'ensemble de Pareto du problème* BiSumCi.

Démonstration. Soit S_* un ordonnancement Pareto-optimal. Alors, il existe $k \in \mathbb{N}$ tel que $(1+\frac{\epsilon}{2})^k \sum \omega_i C_i^{min} \leqslant \sum \omega_i C_i(S_*) \leqslant (1+\frac{\epsilon}{2})^{k+1} \sum \omega_i C_i^{min}$. Nous montrons que S_{k+1} est une $(2+\epsilon, 2)$-approximation de S_*. Ceci est illustré par la Figure 2.4.

- $\sum x_i C_i$. $\sum x_i C_i(S_{k+1}) \leqslant 2 \sum x_i C_i^* ((1+\frac{\epsilon}{2})^{k+1} \sum \omega_i C_i^{min})$ (grâce au Théorème 2.20). S_* est Pareto-optimal, d'où $\sum x_i C_i(S_*) = \sum x_i C_i^* (\sum \omega_i C_i(S_*))$. Or, $\sum \omega_i C_i(S_*) \leqslant (1+\frac{\epsilon}{2})^{k+1} \sum \omega_i C_i^{min}$. Comme $\sum x_i C_i^*$ est une fonction décroissante, nous avons : $\sum x_i C_i(S_{k+1}) \leqslant 2 \sum x_i C_i(S_*)$.
- $\sum \omega_i C_i$. $\sum \omega_i C_i(S_{k+1}) \leqslant 2(1+\frac{\epsilon}{2})^{k+1} \sum \omega_i C_i^{min} = (2+\epsilon)(1+\frac{\epsilon}{2})^k \sum \omega_i C_i^{min}$ (grâce au Théorème 2.20) et $\sum \omega_i C_i(S_*) \geqslant (1+\frac{\epsilon}{2})^k \sum \omega_i C_i^{min}$. D'où, $\sum \omega_i C_i(S_{k+1}) \leqslant (2+\epsilon) \sum \omega_i C_i(S_*)$.

□

2.7 Bilan

Dans ce chapitre, nous avons rappelé les principales définitions et notations utiles pour la résolution des problèmes d'ordonnancement. Quelques

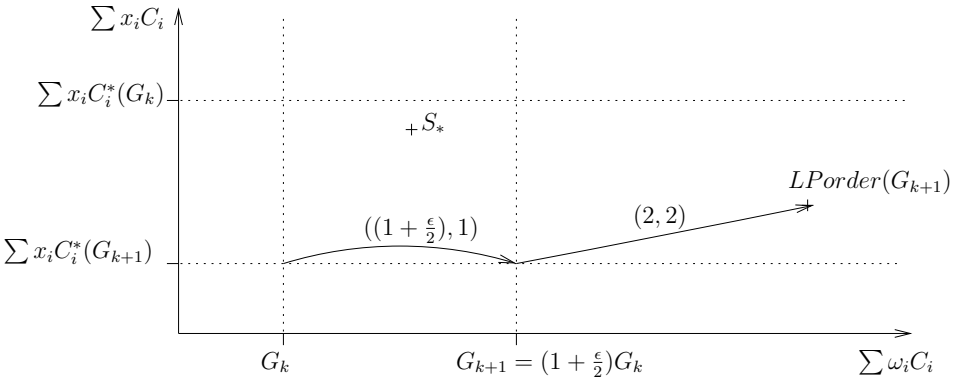


FIG. 2.4 – La solution Pareto-optimale S_* est dominée par $(G_k, \sum x_i C_i^*(G_{k+1}))$. Par construction, $(G_k, \sum x_i C_i^*(G_{k+1}))$ est $(1+\frac{\epsilon}{2}, 1)$-approché par $(G_{k+1}, \sum x_i C_i^*(G_{k+1}))$ qui à son tour est $(2,2)$-approchée par $LPorder(G_{i+1})$.

résultats classiques et importants ont été détaillés pour l'optimisation mono-objectif. Nous avons introduit l'optimisation multi-objectif ainsi que les deux notions clés : l'approximation du zénith et l'approximation de l'ensemble de Pareto.

Chapitre 3

Makespan et contrainte de mémoire

3.1 Description du problème

La capacité de stockage est un besoin peu souvent considéré. La plupart du temps, il est bien plus facile d'augmenter cette capacité que d'obtenir des ordonnancements optimisés pour la mémoire. En effet, le coût financier additionnel est globalement linéaire en l'espace que l'on souhaite rajouter. Cependant, augmenter la capacité mémoire des systèmes n'est pas toujours possible. Par exemple dans les *Multi-Processor-Systems-on-Chip* (où MPsoC), chaque processeur dispose d'une mémoire limitée pour stocker le code des applications. Dans ce contexte, la réplication de code pour permettre des optimisations dynamiquement fait que les contraintes de mémoire deviennent un point clé à optimiser [CKC07]. Il existe des contextes différents où il est important d'optimiser simultanément les temps de calcul et l'occupation en mémoire comme les applications de simulations physiques [CBB+05] ou encore l'allocation des fichiers dans un serveur web [Tse07]. Dans tous les cas, les techniques d'ordonnancement qui s'appliquent sont similaires.

Ce chapitre traite le problème d'optimiser simultanément le temps total d'exécution et la consommation mémoire dans lequel le temps de calcul d'une tâche n'est pas relié à la quantité de mémoire qu'elle utilise. Le problème d'optimiser le makespan avec une contrainte stricte sur la mémoire est un problème connexe intéressant. Cependant, il n'est pas possible de le traiter par la théorie de l'approximation (voir Section 7.2).

Ce chapitre propose deux familles d'algorithmes paramétriques pour calculer une approximation du problème. La première fournit une $(1+\Delta+\epsilon, 1+$

$\frac{1}{\Delta}+\epsilon$)-approximation avec $\Delta > 0$, le paramètre de l'algorithme. Cette famille d'algorithmes ne fonctionne que pour des tâches indépendantes.

La seconde famille produit une $(2+\frac{1}{\Delta-2}-\frac{\Delta-1}{m(\Delta-2)},\Delta)$-approximation avec $\Delta > 2$ pour le problème avec précédences. On peut en dériver une famille d'algorithmes optimisant trois objectifs sur des tâches indépendantes. La somme des temps de complétion est alors approximée avec un rapport de performance de $(2+\frac{1}{\Delta-2})$.

Nous fournissons également une analyse du meilleur algorithme d'approximation que l'on peut obtenir. Nous prouvons qu'aucun algorithme ne peut obtenir un ratio meilleur que $(\frac{3}{2},\frac{3}{2})$. Nous montrons également que des rapports meilleurs que $(1+\frac{i}{km}, 1+(m-1)(1-\frac{i}{k})), \forall m,k \geqslant 2, i \in \{0,\ldots,k\}$ ne sont pas atteignables.

3.2 Le problème formel

Nous commençons cette étude sur un problème d'ordonnancement de tâches indépendantes. Les notations utilisées sont standards et identiques à celles du chapitre précédent à l'exception de s_i, le nombre d'unités de mémoire utilisé par la tâche i. Pour un ordonnancement π, $M_{max}(\pi) = max_{j\in M}\sum_{\pi(i)=j} s_i$ est la quantité maximale de mémoire utilisée par un processeur. Le paramètre π sera omis lorsqu'il n'y a pas d'ambiguïté, remarquons que pour des tâches indépendantes, π suffit à décrire l'ordonnancement.

Remarquons que tant que l'on considère uniquement des tâches indépendantes, les valeurs M_{max} et C_{max} sont strictement équivalentes et peuvent être échangées sans perte de généralité. Ainsi, nous pouvons voir la consommation en mémoire comme une seconde ligne de temps. Cependant, nous pensons que conserver deux notations différentes aidera le lecteur à ne pas confondre les deux objectifs. Tous les résultats sur des tâches indépendantes seront donc symétriques.

Le problème que nous considérons est la minimisation simultanée de C_{max} et M_{max}. Dans la notation à trois champs, il serait noté : $P \mid p_i, s_i \mid (C_{max}, M_{max})$. Rappelons que le problème $P \mid p_i \mid C_{max}$ est $\mathcal{NP}$-complet au sens fort [GJ79]. Les valeurs optimales du makespan et de la consommation mémoire sont respectivement C^*_{max} et M^*_{max}.

Dans la Section 3.5, nous considérerons une variante du problème avec contraintes de précédence (rappelons que les successeurs d'une tâche i sont notés $\Gamma^+(i)$). Nous introduirons alors $\sigma(i)$ la date de début d'exécution de la tâche i. Les contraintes usuelle sur π et σ sont valides. Avec des contraintes de précédence, le problème se note alors : $P \mid p_i, s_i, prec \mid (C_{max}, M_{max})$.

Nous discutons ici de la raison qui nous pousse à considérer le problème

d'optimisation bi-objectif alors que le problème applicatif est de trouver un ordonnancement qui minimise C_{max} sous contrainte $M_{max} \leqslant M$. La raison principale est que le problème $P \mid p_i, s_i, M_{max} \leqslant M \mid C_{max}$ n'est pas approximable à facteur constant. En effet, décider si un ordonnancement tel que $M_{max} \leqslant M$ existe est un problème $\mathcal{NP}$-complet au sens fort directement équivalent à $P \mid p_i \mid C_{max}$ [GJ79]. Un algorithme d'approximation devrait être capable de toujours trouver un ordonnancement valide ou d'assurer qu'un tel ordonnancement n'existe pas. Il est donc impossible (si $\mathcal{P} \neq \mathcal{NP}$) de trouver un algorithme d'approximation pour ce problème qui s'exécute en temps polynômial.

3.3 L'algorithme d'approximation *Symmetric Bi-Objective*

3.3.1 Principe

L'idée de l'algorithme que nous proposons est inspirée de [SW97] (voir la Section 2.6.2). Il s'agit de combiner les solutions de deux algorithmes, chacun optimisant un des deux objectifs. Chaque algorithme ordonnance toutes les tâches et potentiellement certaines tâches ne seront pas ordonnancées sur le même processeur dans les deux solutions. Il faut alors faire un choix pour chaque tâche afin de déterminer sur lequel des deux processeurs la tâche s'exécutera. Ce choix sera effectué en fixant un seuil sur le rapport entre le temps d'exécution et la consommation mémoire. Intuitivement, si une tâche nécessite beaucoup de mémoire mais s'exécute rapidement, il faut l'ordonnancer pour optimiser la consommation mémoire. Inversement, une tâche ayant un long temps de calcul mais une faible consommation mémoire devrait être ordonnancée principalement pour optimiser le temps de calcul.

Étant donné que la consommation mémoire et le makespan sont des objectifs similaires quand ils sont découplés, nous pouvons utiliser le même algorithme pour générer les deux ordonnancements de base. Il faut naturellement remplacer les temps de calcul p_i par la taille du code s_i lorsque l'on calcul l'ordonnancement optimisant la consommation mémoire. Il existe de nombreux algorithmes d'approximation pour ce problème. Les principaux ont été présentés en Section 2.2.2. Rappelons que Kellerer a proposé une $\frac{5}{4}$-approximation [Kel98] et qu'une $\mathcal{PTAS}$ a été exhibée [HS87].

3.3.2 L'algorithme

L'algorithme décrit ci-dessous est formellement écrit en pseudo code dans l'Algorithme 3.1 et est appelé SBO_Δ (pour *Symmetric Bi-Objective*) dans le reste du document. Δ est un paramètre réel strictement positif. Soient π_1 un ordonnancement généré avec une ρ_1-approximation du makespan et π_2 un ordonnancement généré avec une ρ_2-approximation de la consommation mémoire. Pour simplifier l'écriture, nous notons C le makespan produit par le premier algorithme (*i.e.* $C_{max}(\pi_1)$) et M la consommation mémoire du second ordonnancement (*i.e.* $M_{max}(\pi_2)$). Une tâche i est ordonnancée suivant π_1 si son temps de calcul est faible par rapport à sa consommation en mémoire, exprimées relativement aux valeurs garanties C et M.

Algorithme 3.1 SBO_Δ

Input : m : un entier
$\{p_1, \ldots, p_n\}$: n entiers
$\{s_1, \ldots, s_n\}$: n entiers
Begin
Soit π_1 un ordonnancement ρ_1-approché pour C_{max}
Soit $C = C_{max}(\pi_1)$
Soit π_2 un ordonnancement ρ_2-approché pour M_{max}
Soit $M = M_{max}(\pi_2)$
For $i = 1$ **to** n
  if $\frac{p_i}{C} < \Delta \frac{s_i}{M}$
    then $\pi_\Delta(i) = \pi_2(i)$
    else $\pi_\Delta(i) = \pi_1(i)$
End For
Return π_Δ
End

PROPOSITION 3.1. *L'ordonnancement π_Δ généré par SBO_Δ est $(1 + \Delta)\rho_1$-approché pour le makespan.*

Démonstration. Soit T_1 l'ensemble des tâches allouées suivant π_1 et T_2 l'ensemble des tâches allouées suivant π_2. Le temps total d'exécution sur chaque processeur est la somme des temps d'exécution des tâches allouées à ce processeur dans chacun des deux ensembles. Pour l'ensemble T_1, la somme des temps de calcul est aisément bornée par C. Pour l'ensemble T_2, nous avons d'un côté le fait que pour chaque tâche de T_2 le rapport entre son temps d'exécution et sa consommation mémoire est inférieur à $\Delta \frac{C}{M}$ et d'un autre

coté que la consommation mémoire de chaque processeur est dans l'ordonnancement π_2 bornée par M. En combinant ces deux faits, nous obtenons pour chaque processeur j :

$$\begin{aligned}\sum_{i \in T_2, \pi_2(i)=j} p_i &< \textstyle\sum_{i \in T_2, \pi_2(i)=j} \Delta C \frac{s_i}{M} \\ &= \Delta C \left(\textstyle\sum_{i \in T_2, \pi_2(i)=j} \frac{s_i}{M} \right) \leqslant \Delta C_{max}(\pi_1)\end{aligned}$$

Pour chaque processeur j, nous avons donc $\sum_{i,j=\pi_\Delta(i)} p_i \leqslant C_{max}(\pi_1) + \Delta C_{max}(\pi_1)$. Le fait que π_1 soit un ordonnancement ρ_1-approché termine la preuve. □

PROPOSITION 3.2. *L'ordonnancement π_Δ généré par SBO_Δ est $(1 + 1/\Delta)\rho_2$-approché pour la consommation mémoire.*

La preuve est similaire à celle de la Proposition 3.1 et est omise.

THÉORÈME 3.3. *L'algorithme SBO_Δ est une $((1 + \Delta)\rho_1, (1 + \frac{1}{\Delta})\rho_2)$-approximation de la solution zénith de $P \mid p_i, s_i \mid C_{max}, M_{max}$.*

Il existe une $\mathcal{PTAS}$ pour $P \mid\mid C_{max}$, ce qui implique que pour tout $\epsilon > 0$, il existe un algorithme polynômial de $(1 + \Delta + \epsilon, 1 + \frac{1}{\Delta} + \epsilon)$-approximation du zénith. De plus, théoriquement, on pourrait utiliser un algorithme exact pour chacun des objectifs. Cela mène au corollaire suivant :

COROLLAIRE 3.4. *Pour tout $\Delta > 0$, il existe une solution de makespan inférieur à $(1 + \Delta)C^*_{max}$ et de consommation mémoire inférieure à $(1 + \frac{1}{\Delta})M^*_{max}$.*

Remarquons que l'équilibre entre les deux rapports est atteint pour $\Delta = 1$ où le rapport d'approximation est de $(2, 2)$.

3.4 Des rapports d'approximation impossibles

Dans cette section, nous nous concentrons sur les propriétés d'inapproximabilité de l'ensemble de Pareto en une seule solution. Nous détaillons d'abord le principe sur un résultat simple en Section 3.4.1 que nous généralisons avec un paramètre dans la Section 3.4.2. Puis nous fournissons un autre résultat qui raffine certaines inapproximabilités en Section 3.4.3.

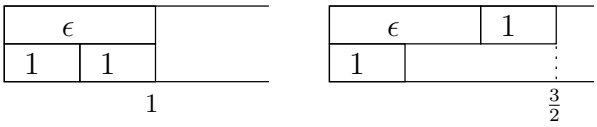


FIG. 3.1 – Les deux ordonnancements Pareto-optimaux pour une première instance (La longueur des tâches est proportionnelle au temps de calcul et la consommation mémoire est inscrite sur la tâche).

3.4.1 De la non existence de solution

Considérons une instance du problème composée de 2 processeurs et de 3 tâches avec : $p_1 = 1, p_2 = p_3 = \frac{1}{2}$ et $s_1 = \epsilon, s_2 = s_3 = 1$. Il n'y a que 3 ordonnancements possibles (lorsque l'on retire les symétries et les ordonnancements avec temps d'inactivité). Dans le premier, la tâche 1 est ordonnancée en parallèle des tâches 2 et 3. Dans le deuxième, les tâches 1 et 2 sont ordonnancées sur le même processeur. Dans le troisième, toutes les tâches sont ordonnancées sur le même processeur. Ces trois solutions ont les trois valeurs objectifs suivantes : $(1, 2), (\frac{3}{2}, 1 + \epsilon)$ et $(2, 2 + \epsilon)$. La dernière solution est Pareto-dominée par les deux autres. La Figure 3.1 représente les deux ordonnancements Pareto-optimaux sous la forme de diagramme de Gantt (le temps est représenté par la longueur des rectangles) où la consommation mémoire d'une tâche est notée sur la tâche.

Dans cette instance, nous avons $C^*_{max} = 1$ et $M^*_{max} = 1 + \epsilon$. Supposons qu'il existe un algorithme de rapport inférieur à $(1, 2)$, par exemple de rapport $(1, \frac{7}{4})$-approché de la solution zénith. Cet algorithme générerait une solution avec $C_{max} \leq 1$ et $M_{max} \leq \frac{7}{4}(1+\epsilon)$. Cependant, une telle solution n'existe pas pour cette instance. Il n'est donc pas possible qu'un tel algorithme existe. Pour nos calculs, ϵ peut être aussi petit que l'on veut. Il n'existe pas d'algorithme d'approximation avec un rapport de performance meilleur que $(1, 2)$. Ce résultat est symétrique.

LEMME 3.5. *Aucun algorithme d'approximation de la solution zénith n'a de rapport d'approximation meilleur que* $(1, 2)$ *ou* $(2, 1)$.

3.4.2 Extension de l'idée à m processeurs

Pour aller plus loin, ce résultat peut être exprimé pour plus de 2 processeurs et de 3 tâches. Avec m processeurs, nous construisons une instance similaire contenant $km + m - 1$ tâches. Comme précédemment, il y a deux types de tâches. Les $m - 1$ premières tâches sont identiques et respectent $p_1 = \cdots = p_{m-1} = 1$ et $s_1 = \cdots = s_{m-1} = \epsilon$. Alors que les km autres tâches

sont telles que $p_m = \cdots = p_{km+m-1} = \frac{1}{km}$ et $s_m = \cdots = s_{km+m-1} = 1$. Le makespan optimal est 1 et la consommation mémoire optimale est $k + \epsilon$.

Une analyse de cas similaire à celle effectuée précédemment montre qu'il y a $k + 1$ ordonnancements Pareto-optimaux. La solution Pareto-optimale $i \in \{0, 1, \ldots, k\}$ ordonnance i tâches du second type et une tâche du premier type sur chacun des $m-1$ premiers processeurs et ordonnance les $km-i(m-1)$ tâches restantes du second type sur le dernier processeur.

Le makespan de la solution i est de $1 + \frac{i}{km}$ et sa consommation mémoire est de $k + (k - i)(m - 1)$ sauf pour $i = k$. La solution k a une consommation mémoire de $k + \epsilon$. Le même argument que précédemment nous dit qu'il n'existe pas d'algorithme avec un rapport d'approximation meilleur que $(1 + \frac{i}{km}, 1 + (m - 1)(1 - \frac{i}{k}))$.

LEMME 3.6. $\forall m, k \geqslant 2, i \in \{0, \ldots, k\}$, *il n'existe pas d'algorithme qui approche la solution zénith avec un facteur inférieur à* $(1 + \frac{i}{km}, 1 + (m - 1)(1 - \frac{i}{k}))$.

Remarquons que la fraction $\frac{i}{k}$ peut atteindre toutes les valeurs rationnelles entre 0 et 1. Le résultat d'inapproximabilité est ainsi continu. De plus, nous pouvons produire les résultats symétriques en inversant les temps de calcul et les consommations mémoires.

Les résultats obtenus pour des petites valeurs de m sont représentés en Figure 3.3 avec les résultats de la section suivante.

3.4.3 D'autres ordonnancements inexistants

Nous considérons ici une instance avec une structure différente bien que l'instance soit toujours composée de 2 processeurs et de 3 tâches. Les temps de calcul et les tailles du code de chaque tâches sont : $p_1 = 1, p_2 = \epsilon, p_3 = 1 - \epsilon$ et $s_1 = \epsilon, s_2 = 1, s_3 = 1 - \epsilon$. Comme dans la Section 3.4.1, il n'y a que peu d'ordonnancements possibles. Les trois ordonnancements potentiellement Pareto-optimaux sont obtenus en groupant exactement deux tâches sur le même processeur. Exécuter les trois tâches sur le même processeur fournit une solution dominée par chacun des autres ordonnancements. Les valeurs objectifs des trois ordonnancements sont : $(1, 2 - \epsilon)$ lorsque la tâche 1 est seule, $(1 + \epsilon, 1 + \epsilon)$ lorsque la tâche 3 est seule, et $(2 - \epsilon, 1)$ lorsque la tâche 2 est seule. Étant donné que C^*_{max} et M^*_{max} valent tout les deux 1, ces valeurs sont également les rapports d'approximation minimaux. Remarquons que le point $(1 + \epsilon, 1 + \epsilon)$ n'est Pareto-optimal que lorsque ϵ est inférieur à $\frac{1}{2}$. Les ordonnancements Pareto-optimaux correspondant sont représentés dans la Figure 3.2. Pour des valeurs de ϵ proches de $\frac{1}{2}$, cette instance mène au lemme suivant :

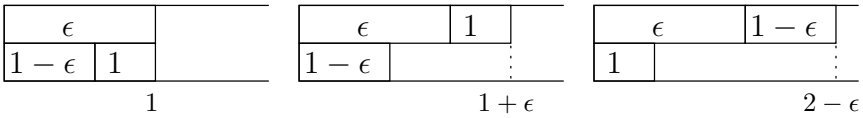


FIG. 3.2 – Les solutions Pareto-optimales de la seconde instance.

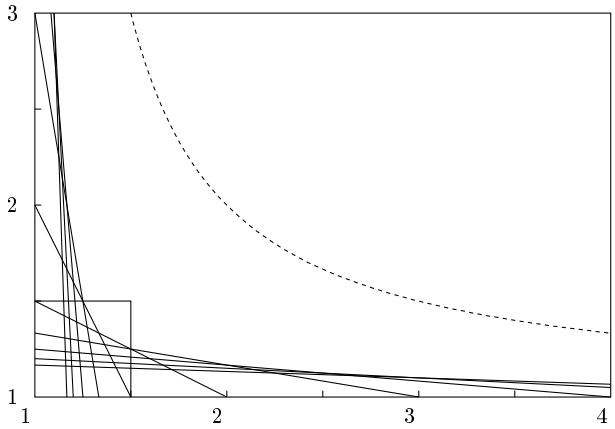


FIG. 3.3 – Le domaine de rapport d'approximation non atteignable pour la minimisation du makespan et de la consommation mémoire. La ligne en pointillés représente les rapports d'approximation atteignables par l'algorithme présenté en Section 3.3.

LEMME 3.7. *Aucun algorithme n'approche la solution zénith avec un rapport d'approximation inférieur à* $(\frac{3}{2}, \frac{3}{2})$.

Ce résultat ne peut pas être étendu vers plus de processeurs pour obtenir de nouveaux résultats comme nous l'avons fait dans la section précédente. En effet, toutes les tâches sont différentes et découper une tâche ferait apparaître de bien meilleures solutions (qui amène à des rapports d'approximation impossibles moins intéressants).

Les résultats de ce lemme et de celui d'avant sont représentés sur la Figure 3.3. Le carré de coin supérieur droit $(\frac{3}{2}, \frac{3}{2})$ vient du Lemme 3.7. Les autres rapports viennent du Lemme 3.6 pour $m \leqslant 6$. Remarquons que les rapports d'approximation impossible que nous avons exhibés sont des ouverts et qu'il est donc peut être possible d'avoir un algorithme d'approximation dont le rapport est sur la ligne. Pour compléter l'illustration, nous avons inclus les rapports de performances de l'algorithme SBO_{Δ} en pointillés.

3.5 Deux généralisations du problème

Alors que le problème de tâches indépendantes ressemble plus à des modèles d'exécution de type grille de calcul, l'ordonnancement de DAG de tâches est un modèle plus proche des problématiques d'applications parallèles sur systèmes embarqués. L'algorithme SBO_Δ présenté en Section 3.3 ne s'étend pas en présence de contraintes de précédence. Nous développons alors un nouvel algorithme pour traiter les graphes de tâches. L'algorithme est détaillé et analysé en Section 3.5.1. Puis nous présentons en Section 3.5.2 une extension de cet algorithme qui permet l'optimisation d'un troisième critère, la somme des temps de terminaison, dans le cas de tâches indépendantes.

3.5.1 Un algorithme fondé sur *List Scheduling*

Comme nous l'avons rappelé dans le chapitre précédent, l'algorithme *List Scheduling* est connu pour être une 2-approximation du makespan sur des machines identiques en présence de contraintes de précédence [Gra69]. Son principe revient à assurer que si un processeur est inactif alors il n'y a pas de tâche disponible à ce moment précis. La preuve de la 2-approximation pour un ensemble de tâches indépendantes a été présentée en Section 2.2.2 et repose sur deux bornes inférieures du makespan : le temps de calcul la plus longue tâche et le temps de calcul s'il n'y avait pas de temps d'inactivité dans la solution optimale. Avec des contraintes de précédence, le temps de la plus longue tâche est avantageusement remplacé par le temps de calcul de la chaîne la plus longue, aussi appelé chemin critique (ou *Critical Path*). L'algorithme que nous proposons est inspiré de *List Scheduling*.

L'algorithme RLS_Δ (pour *Restricted List Scheduling*), présenté formellement en Algorithme 3.2, calcule une borne inférieure LB sur la valeur optimale de M_{max} à partir des bornes inférieures classiques. Ensuite, une dégradation par rapport à cette borne est autorisée sur M_{max} : aucun processeur ne doit utiliser plus de ΔLB unités de mémoire. L'algorithme ordonnance itérativement la tâche qui peut commencer le plus tôt sans dépasser la borne sur la mémoire ΔLB. Si deux tâches peuvent être ordonnancées, l'algorithme choisit celle qui est le plus tôt dans un ordre total arbitrairement choisi.

En ce qui concerne l'analyse de l'algorithme, nous nous intéressons d'abord au nombre de processeurs qui ont été marqués, c'est-à-dire au nombre de processeurs qui n'ont pas été choisis par l'algorithme car leur consommation en mémoire étaient trop importante.

LEMME 3.8. *A la fin de l'algorithme, il y a moins de* $\frac{m}{\Delta-1}$ *processeurs marqués.*

Algorithme 3.2 RLS_Δ

Input : m : un entier
$\{p_1, \ldots, p_n\}$: n entiers
$\{s_1, \ldots, s_n\}$: n entiers
Begin
Soit $LB = \max(\max_i s_i, \sum_i \frac{s_i}{m})$
$load[1] = \cdots = load[m] = 0$
$memsize[1] = \cdots = memsize[m] = 0$
While Il reste au moins une tâche à ordonnancer
  For all tâche disponible i
    Soit $proc[i] = j$ le processeur minimisant $load[j]$
      tel que $memsize[j] + s_i \leqslant \Delta LB$
    Soit $ready[i] = \max(\max_{i' \in \Gamma^-(i)} \sigma(i') + p_{i'}, load[proc[i]])$
    /*pour l'analyse : */
    Marquer les processeurs j' tel que $load[j'] < load[proc[i]]$
  End For
  Soit i^* la tâche disponible qui minimise $ready[i]$
  $\pi(i^*) = proc[i^*]$
  $\sigma(i^*) = ready[i^*]$
  $load[proc[i^*]] = \sigma(i^*) + p_{i^*}$
  $memsize[proc[i^*]] += s_{i^*}$
End Until
Return (π, σ)
End

Démonstration. Pour chaque processeur marqué j, il existe une tâche i telle que $memsize[j] > \Delta LB - s_i \geq (\Delta - 1)LB$ (la dernière inégalité vient de $s_i \leq LB$). Supposons que plus de $\frac{m}{\Delta-1}$ processeurs sont marqués. Nous avons donc, $\sum_{j \in marked} memsize[j] > \frac{m}{\Delta-1}(\Delta - 1)LB = mLB$. Cependant, $\sum_{j \in marked} memsize[j] \leq \sum_i s_i \leq mLB$. Nous arrivons à la contradiction : $mLB \geq \sum_{j \in marked} memsize[j] > mLB$. D'où, il y a moins de $\frac{m}{\Delta-1}$ processeurs marqués. □

Ce lemme implique directement que l'algorithme RLS_Δ ne peut pas admettre des valeurs de Δ inférieures à 2. En effet, pour ces petites valeurs, il pourrait exister une tâche qui ne peut être placée sur aucun processeur à cause d'une importante consommation en mémoire (qui marquerait tous les processeurs). Ainsi la valeur de Δ que nous choisissons doit être supérieure ou

égale à 2. Par construction, cela amène un ordonnancement de consommation mémoire inférieure à ΔLB.

COROLLAIRE 3.9. *RLS_Δ est une Δ-approximation sur la consommation mémoire M_{max} pour $\Delta \geqslant 2$.*

Étant donné que moins de $\frac{m}{\Delta-1}$ processeurs sont inutilisés pour cause de consommation mémoire, plus de $m\frac{\Delta-2}{\Delta-1}$ processeurs peuvent être librement utilisés pour optimiser C_{max}. Cela est suffisant pour permettre une garantie sur le makespan.

LEMME 3.10. *RLS_Δ est une $(2+\frac{1}{\Delta-2}-\frac{\Delta-1}{m(\Delta-2)})$-approximation sur C_{max} pour $\Delta > 2$.*

Démonstration. Nous considérons une partition $CP \cup W$ du temps entre 0 et C_{max}. Une unité de temps t appartient à CP si au moins un processeur est inoccupé au temps t et toutes les tâches i ordonnancées après ne sont pas disponibles au temps t. Formellement, $\max_{i' \in \Gamma^-(i)} \sigma(i') + p_{i'} > t$. Remarquons que la cardinalité de CP est inférieure au temps de calcul de toutes les chaînes du DAG. D'où $\mid CP \mid \leqslant C^*_{max}$.

Toute la charge de calcul $\sum p_i$ est ordonnancée dans W à l'exception de la partie qui est exécutée dans CP. Cette partie est plus grande de $\mid CP \mid$ puisqu'au moins un processeur est actif dans CP (et la borne est atteinte si seulement un processeur est actif durant tout ce temps). Rappelons que le Lemme 3.8 nous indique qu'au moins $m\frac{\Delta-2}{\Delta-1}$ processeurs ne sont jamais trop chargés en mémoire. Nous en déduisons que la longueur de W est plus petite que $\frac{\sum p_i - |CP|}{m\frac{\Delta-2}{\Delta-1}}$.

Comme les temps de l'ordonnancement sont partitionnés en W et CP, nous pouvons exprimer la longueur de l'ordonnancement comme la somme de la longueur des deux parties :

$$\begin{aligned} C_{max} = \mid W \mid + \mid CP \mid \leqslant \quad & (1+\tfrac{1}{\Delta-2})\tfrac{\sum p_i}{m} \\ & +(1-\tfrac{\Delta-1}{m(\Delta-2)}) \mid CP \mid \end{aligned}$$

Rappelons que $\frac{\sum p_i}{m}$ et $\mid CP \mid$ sont des bornes inférieures du makespan optimal C^*_{max}. Nous obtenons à partir de l'équation précédente :

$$C_{max} \leqslant (2+\frac{1}{\Delta-2}-\frac{\Delta-1}{m(\Delta-2)})C^*_{max}$$

□

Les deux lemmes précédents nous amènent au facteur d'approximation de RLS_Δ :

THÉORÈME 3.11. *RLS_Δ est une $(2 + \frac{1}{\Delta-2} - \frac{\Delta-1}{m(\Delta-2)}, \Delta)$-approximation de la solution zénith de complexité $O(n^2m)$, pour $\Delta > 2$.*

Notons que tant que n est plus grand que m, cet algorithme est polynômial dans la taille de l'instance. Si cette hypothèse n'est pas vrai, il n'est pas utile d'utiliser plus de n processeurs. Pour simplifier la comparaison avec les résultats de la Section 3.3, nous pouvons remplacer Δ par $2 + \Delta'$. Ainsi le rapport d'approximation s'écrit $(2 + \frac{1}{\Delta'} - \frac{\Delta'+1}{m\Delta'}, 2 + \Delta')$.

Remarquons que notre analyse ne fonctionne pas pour $\Delta = 2$. Cependant, nous pouvons traiter ce cas de manière séparée en remarquant qu'au moins un processeur restera non marqué. Le rapport d'approximation sera alors de m pour le makespan.

3.5.2 Une extension à trois objectifs sur des tâches indépendantes

Nous nous intéressons dans cette section à ordonnancer des tâches indépendantes en optimisant les fonctions objectives suivantes : makespan, consommation mémoire et somme des temps de terminaison. Une fois encore, l'algorithme présenté en section Section 3.3 ne s'adapte pas facilement à l'optimisation de la somme des temps de terminaison. Cependant l'algorithme RLS_Δ présenté dans la section précédente nous permet de considérer les tâches dans l'ordre SPT. Rappelons que *List Scheduling* est optimal pour $\sum C_i$ si l'ordre SPT est utilisé.

Nous considérons d'abord la dégradation que peut subir l'objectif $\sum C_i$ lorsque fraction des processeurs est indisponible.

LEMME 3.12. *Soient π_1, π_2 deux ordonnancements SPT sur m et ρm processeurs du même ensemble de tâches ($0 < \rho \leqslant 1$) alors $\sum C_i(\pi_2, \sigma_2) \leqslant (\frac{1}{\rho} + 1) \sum C_i(\pi_1, \sigma_1)$.*

Démonstration. Nous prouvons le lemme en montrant que $\forall j, C_j(\pi_2, \sigma_2) \leqslant (\frac{1}{\rho} + 1)C_j(\pi_1, \sigma_1)$.

Sans perte de généralité, nous pouvons renuméroter les tâches suivant l'ordre SPT. Dans des ordonnancements SPT partiel où la tâche i vient d'être ordonnancée, elle est la dernière à terminer. Ainsi, $\frac{1}{m} \sum_{k=1,\ldots,i} p_k \leqslant C_i(\pi_1, \sigma_1)$.

Sur ρm processeurs, lorsque i commence, seulement les $i - 1$ premières tâches ont été ordonnancées. D'où $\sigma_2(j) \leqslant \frac{1}{\rho m} \sum_{k=1}^{i-1} p_k$. Cela mène à $C_j(\pi_2, \sigma_2) \leqslant \frac{1}{\rho m} \sum_{k=1}^{i-1} p_k + p_i \leqslant \frac{1}{\rho} C_i(\pi_1, \sigma_1) + p_i$. Comme le temps de terminaison $C_i(\pi_1, \sigma_1)$

de la tâche i est supérieur à son temps d'exécution p_i, nous avons $C_i(\pi_2, \sigma_2) \leqslant (\frac{1}{\rho} + 1)C_i(\pi_1, \sigma_1)$ □

Dans RLS_Δ, $m\frac{\Delta-2}{\Delta-1}$ processeurs sont toujours disponibles. Ainsi, nous pouvons appliquer le lemme précédent avec cette valeur particulière pour ρ. De plus, SPT est optimal pour $\sum C_i$. Donc, avoir des processeurs supplémentaires ne peut pas dégrader l'objectif $\sum C_i$. De tous ces résultats, nous obtenons le rapport d'approximation de RLS_Δ si l'ordre SPT est utilisé.

THÉORÈME 3.13. *Utiliser l'ordre SPT dans RLS_Δ génère une $(2+\frac{1}{\Delta-2}-\frac{\Delta-1}{m(\Delta-2)}, \Delta, 2+\frac{1}{\Delta-2})$-approximation de la solution zénith pour les objectifs $(C_{max}, M_{max}, \sum C_i)$*

3.6 Variante *Online*

Un article de Bilò *et al.* [BFM06] traite du problème d'optimiser deux objectifs de type makespan pour ordonnancer des tâches indépendantes. La différence principale avec notre travail est qu'ils considèrent le problème *online* : les tâches sont soumises au système les unes après les autres et l'ordonnanceur ne peut pas revenir sur ses décisions et il ne connaît pas les tâches qu'il devra ordonnancer à l'avenir.

Pour ce problème, ils proposent un algorithme dérivé de *List Scheduling* qui place les tâches sur le processeur qui minimise une dimension de temps parmi les x processeurs qui minimise la deuxième dimension de temps où x est un paramètre de l'algorithme. Ils montrent que l'algorithme A_x est une $(\frac{2m-x}{m-x+1}, \frac{m+x-1}{k})$-approximation du zénith. L'algorithme *online* est naturellement utilisable dans le cadre *offline*. Il mène quasiment au même rapport d'approximation que RLS_Δ à l'avantage de A_x lorsque le nombre de processeurs est faible. Cependant, la différence vient uniquement de l'analyse. Il est en effet possible de voir l'algorithme RLS_Δ comme une version dynamique de A_x où x commence à m et décroît jusqu'à une constante donnée. L'analyse fournit dans [BFM06] peut alors être utilisée pour cette nouvelle écriture de RLS_Δ. Dans le cadre offline, nous préférons donc l'algorithme RLS_Δ qui correspond mieux aux applications avec contrainte de mémoire. Dans ces systèmes, la valeur du paramètre Δ apparaît naturellement. Il est même possible de calculer à postériori le rapport d'approximation obtenu. De plus, l'algorithme RLS_Δ ressemble beaucoup plus aux algorithmes qui sont usuellement proposés lorsque ce style de problème apparaît. Cette analyse permet de fournir un argument théorique validant les approches gloutonnes.

Les auteurs de [BFM06] s'intéressent également à borner le meilleur rapport d'approximation atteignable. Les techniques sont différentes de celles

présentées en Section 3.4 car ils utilisent le fait que l'ordre de placement des tâches est imposé et inconnu. Ils montrent que $\forall m > 3, \forall x \in \mathbb{N}, 2 \leqslant x \leqslant \lceil \frac{m}{2} \rceil$, il n'existe pas d'algorithme d'approximation du zénith avec un rapport meilleur que $(2 + \frac{1}{\lceil \frac{m}{x-1} \rceil}, \frac{m}{x})$. Ils montrent également que pour 2 ou 3 processeurs les algorithmes qu'ils proposent ont des rapports d'approximation non améliorables.

3.7 Conclusion

Outre les différents résultats que nous montrons dans ce chapitre, il est important de noter ici le coeur de notre approche. Il s'agit de transformer un problème d'optimisation contraint non approximable en problème multi-objectif. Cela étant fait, il devient possible d'analyser le problème pour extraire les cas qui posent réellement problème.

Sur notre problème d'optimisation du makespan sous contrainte de mémoire, les instances où la marge de manoeuvre sur la mémoire est inférieure à un facteur 2 sont réellement problématiques. En effet, à cette échelle une très légère modification de la contrainte de mémoire a un effet important sur le makespan que l'on obtient.

Chapitre 4

La sûreté de fonctionnement

Dans ce court chapitre, nous décrivons le contexte général de la sûreté de fonctionnement qui est commun aux systèmes parallèles à grande échelle et embarqués.

4.1 Motivation

Les machines que nous utilisons au quotidien ne sont pas parfaites. Il arrive qu'elles fonctionnent mal. Différentes causes à ces mauvais fonctionnements existent. L'usure d'un processeur dûe à son utilisation ou dûe à la chaleur, une pièce mécanique d'un disque dur qui casse, des blocs défectueux dans la mémoire vive, les rayons cosmiques sont autant de causes de dysfonctionnement d'une machine de calcul. Tant que l'on traitait des calculs courts sur un nombre faible de machines, ces problèmes étaient insignifiants et, ont été ignorés.

Aujourd'hui, l'utilisation simultanée de nombreuses ressources de calcul pendant une durée importante provoque l'apparition de pannes qui ne peuvent plus être négligées. Par exemple les machines BlueGene d'IBM disposent d'un grand nombre de processeurs, de l'ordre de 65000, et sont soumises à des pannes fréquentes : de l'ordre d'une panne tous les 10 jours était prévu à la conception du système [Aea02], de l'ordre d'une panne par jour était attendu [OSM+04] et des dizaines de pannes par jour arrivent en pratique [LZS+06] (par exemple, le temps moyen entre deux pannes liées au réseau est de 2 heures).

Les concepts de base en sûreté de fonctionnement [Lap04] sont la faute (ou bien *fault*), l'erreur (*error*), et la défaillance (*failure*). La faute est un dysfonctionnement dans un composant matériel ou logiciel qui peut se traduire par une erreur. Une défaillance est une divergence d'un système par rapport

à son comportement attendu. La défaillance d'un composant peut être une faute d'un système plus large. Ceci est un cycle classique en sûreté de fonctionnement. C'est pourquoi les systèmes qui résistent à la défaillance d'un processeur sont dit tolérants aux fautes (ou *fault-tolerant*) et non tolérants aux défaillances.

4.2 Les propriétés des fautes

Différents types de fautes sont usuellement distingués en fonction de leurs origines. Elles peuvent être intentionnelles ou accidentelles, logicielles ou matérielles, modifier le temps de traitement d'une opération, fournir un faux résultat (fautes byzantines) ou ne pas en fournir. Dans ce manuscrit, nous nous intéressons aux fautes accidentelles qui ne modifient pas le temps de traitement et ne fournissent pas de résultats en cas de fautes.

Les fautes sont également distinguées par le temps durant lequelle elles existent. On parle de faute permanente (ou *permanent fault*) si le composant ne fonctionnera plus jamais ou faute transitoire (ou *transient fault*) si la faute n'est effective que pendant un temps fini. On associe alors aux fautes transitoires la durée pendant laquelle la faute est effective. Cette durée peut être déterministe ou stochastique. Remarquons qu'il est classique de considérer des fautes transitoires de durées quasi nulle.

Il est également important de décrire les dates auxquelles les fautes arrivent dans le système. Bien qu'il existe des cas où les fautes arrivent de façon déterministe, il est classique de considérer des arrivées de fautes stochastiques. Le concept de taux moyen de pannes apparaît alors (souvent appelé MTBF pour *Mean Time Between Failures*). Lorsque les fautes sont indépendantes ou encore si le taux de panne est constant, il est classique de considérer que les fautes sont déterminées par un processus de Poisson. Dans d'autre cas, la loi de Weibull fournit des modèles plus complets de distribution de fautes. La description de plusieurs types de distribution de fautes et une discussion de leur applicabilité fait l'objet d'un chapitre de [BP96].

4.3 Les moyens de la sûreté de fonctionnement

Pour se protéger des fautes, plusieurs approches ont été proposées. Pour lutter contre des fautes byzantines, de nombreuses techniques *ad hoc* ont été proposées. Dans le domaine du transfert ou du stockage de données, la théorie des codes fournit des algorithmes de récupération [DRTV07]. Pour lutter contre des fautes byzantines, des algorithmes tolérants des fautes ont

été conçus. C'est le cas par exemple du tri de Leighton [LM99] qui tri un ensemble de donnée même lorsque la fonction de comparaison fournit parfois un résultat erroné.

Le cas des fautes qui ralentissent les exécutions peut être traité de plusieurs façons. Il est par exemple possible de les traiter comme un problème d'ordonnancement avec incertitude ou perturbations sur les données. Les analyses de robustesse et de sensibilité d'algorithmes d'ordonnancement peuvent alors répondre à ces problèmes [BMS05, Mah04]. On trouve également des modèles où les tâches ont une probabilité de s'exécuter correctement. Si la tâche a échouée, il faut alors la recommencer. On s'autorise alors à exécuter plusieurs copies de la tâche simultanément [LR07, CDF+08].
Pour les cas qui nous intéressent dans cette thèse, une technique connue est l'utilisation de point de reprise (ou *checkpoint*) [CL85]. L'idée est de sauvegarder périodiquement l'état de la machine sur un support stable (*i.e.*, réputé sans faute ; typiquement, un disque dur RAID externe). Lorsqu'une faute est détectée, le calcul est arrêté et repris depuis la dernière sauvegarde. Cette technique pose plusieurs problèmes. Il est nécessaire d'utiliser des algorithmes compliqués à mettre en oeuvre pour synchroniser les machines pour faire une sauvegarde afin d'obtenir une image cohérente du calcul [Jaf06]. De plus, le coût de cette sauvegarde augmente avec la taille des systèmes. Par exemple, sur les machines BlueGene, une sauvegarde peut prendre de l'ordre d'une demi heure [LZS+06].

Un autre moyen de se prémunir des fautes éventuelles est de dupliquer que certains modules. Plusieurs façons de le faire apparaissent alors selon que les multiples instances d'un module sont effectivement exécutées ou non. On parle alors de réplication active ou passive. Dans les techniques de réplication passives, il est classique de trouver la *primary-backup approach* dans laquelle un doublon de chaque module est présent et reste inactif tant qu'aucune faute n'a été détectée. Les techniques de réplication active prévoient et exécutent plusieurs copies de chaque module. Un mécanisme de détection de fautes permet de choisir le résultat de quelle copie d'un module il faut conserver [Kal04].

La question de l'estimation de sûreté de fonctionnement lors de l'utilisation de techniques de duplication se pose alors. Comment savoir si les moyens mis en oeuvre rendent le système tolérant aux fautes ?

Le premier indice utilisé est le nombre minimum de fautes supportées par le système. Il correspond au nombre de duplications du module le moins répliqué. Cet indice est peu satisfaisant parce qu'il n'utilise aucune connaissance sur les fautes. De plus, il représente le pire cas où toutes les fautes sont appliquées au même module. Ainsi, il ne donne pas d'information précise sur le niveau de sûreté du système Le deuxième indice est la fiabilité du système.

Il s'agit de la probabilité que l'application s'exécute correctement.

La fiabilité est exactement la quantité qui intéresse le décideur, c'est pourquoi nous nous focaliserons dessus. Cependant, l'optimisation de cette quantité dépend fortement des modèles que l'on considère. Souvent, le simple fait de calculer la fiabilité d'un système est un problème difficile. Optimiser cette quantité l'est encore plus.

La sûreté de fonctionnement est un problème important du point de vue des applications et difficile du point de vue théorique. Les modèles sont très nombreux et chaque légère variation du problème amène son lot de particularités. Ainsi nous trouvons des modèles de fautes transitoires séparées par des temps exponentiels où la fiabilité est optimiser par de la duplication active [Kal04]. Nous trouvons aussi des modèles où les fautes sont permanentes, de probabilités de pannes constantes dans des problèmes d'ordonnancement en régime permanent [BRSR08]. Ou encore des modèles de fautes permanentes où la probabilité de faute est abstraite par une fonction objectif d'efficacité de l'ordonnancement en présence de fautes [FHKS08].

En bref, aucun modèle ne fait l'unanimité car chaque plate-forme de calcul nécessite de développer un modèle de fiabilité *ad hoc*. Les problèmes qui en découlent sont suffisamment différents pour que les techniques correspondant à un modèle ne soient pas applicable à un autre problème.

Les deux chapitres suivants considèrent les problèmes d'optimisation simultanée de la fiabilité et du makespan. Le Chapitre 5 considère des systèmes embarqués critiques où les défaillances sont transitoires. La fiabilité est améliorée à l'aide de duplication active. Le Chapitre 6 traite de l'optimisation de la fiabilité sur des *clusters* où les défaillances sont permanentes, non pas par la duplication mais par l'allocation des tâches sur les processeurs les plus fiables.

Chapitre 5

Makespan et tolérance aux pannes transitoires

Dans ce chapitre, nous considérons le problème de l'optimisation simultanée de la performance et de la sûreté de fonctionnement dans les systèmes embarqués.

5.1 Les systèmes embarqués critiques

Nous considérons dans ce chapitre la fiabilité dans des systèmes critiques, distribués et réactifs. De tels systèmes sont déployés par exemple dans les systèmes de freinage des voitures, dans l'avionique ou encore dans le domaine aérospatial. Ces systèmes, étant critiques, doivent être fiables et répondre en un temps faible car ils sont utilisés dans des environnements réactifs. Nous étudions le problème de l'optimisation du makespan et de la fiabilité d'une application dans de tels systèmes. Une application sera représentée classiquement par un graphe de tâches.

Ces systèmes sont généralement composés de processeurs de types différents, ils sont évidement hétérogènes, mais il n'est pas réaliste de supposer qu'il existe un facteur de puissance reliant les performances des processeurs pour toutes les opérations possibles (Nous sommes donc dans le modèle R, unrelated). Les fautes sur les processeurs sont supposées transitoires et les défaillances des tâches sont supposées statistiquement indépendantes. Nous verrons que cette dernière hypothèse est obligatoire pour que le calcul de la fiabilité d'un ordonnancement soit possible en temps polynomial. De plus, elle implique que les fautes soient de durées suffisamment courtes pour n'affecter qu'une seule tâche, sinon, les fautes sur les tâches ne seraient pas statistiquement indépendantes. Du point de vue des systèmes embarqués critiques, les

défaillance transitoires sont plus communes que les défaillances permanentes. De plus, les processeurs sont généralement à silence sur défaillance (ou *fail-silent*) [Pea88], c'est-à-dire qu'en cas de défaillance, ils ne fournissent aucun résultat plutôt que de fournir un résultat erroné (matériellement, ce comportement est obtenu en faisant exécuter le même code à deux unités de calcul et en comparant les résultats générés). Ainsi l'indépendance des fautes sur des processeurs différents est une hypothèse pertinente. Les fautes apparaissent suivant une loi de Poisson. Cette hypothèse est valide sur de tels systèmes car les calculs exécutés sont très courts. Ainsi, le taux de panne d'un processeur peut être supposé constant. Ce modèle est directement emprunté à [SW92] et est largement utilisé dans la littérature [AGK04, KM97, PPI07].

Les processeurs sont reliés par un réseau de communication qui n'est pas nécessairement homogène, mais qui est supposé pleinement fiable. Cette hypothèse est facilement obtenue en pratique en répliquant les liens de communications peu chers un nombre de fois suffisant pour qu'ils soient bien plus fiables que les processeurs et en implémentant un protocole de communication capable de traiter leur fautes de façon transparente comme dans [GKS06].

Les contraintes de réactivité du système font que les approches à base de points de reprise ou de réplication passive ne sont pas envisageables. Il faut pouvoir garantir les temps de calcul au pire cas et mais aussi le bon fonctionnement de l'application. Le principe de réplication active est donc utilisé pour améliorer la fiabilité du système.

Ce problème a été partiellement étudié. Plusieurs algorithmes ont été proposés pour minimiser le makespan et certain utilisent la réplication de tâches pour limiter l'influence des communications. Cependant ces réplications n'améliorent généralement pas vraiment la fiabilité. De plus, en présence de réplication, la fiabilité est difficile à calculer. Ainsi, nous ferons quelques hypothèses sur les duplications. Quelques algorithmes ont été proposés pour optimiser les deux objectifs, mais ils ne fournissent usuellement qu'une seule solution ; c'est-à-dire un seul point dans l'espace des objectifs [AGK04, DÖ05, SW92].

5.2 Modélisation du problème

5.2.1 Notations

Nous considérons classiquement un graphe d'application à ordonnancer sur un réseau de processeurs hétérogènes (modèle R). À chaque arc $(i, i') \in E$ est associé une quantité de données $data_{ii'}$ à transférer de la tâche i à la tâche i'. Le réseau de communication est complètement connecté et le lien entre j et

j' dispose d'une bande passante de $BW_{jj'}$. Ainsi le temps de communication inter-processeur entre la tâche i sur j et la tâche i' sur j' est $comm_{iji'j'} = data_{ii'}/BW_{jj'}$. Les temps de communications intra-processeur sont supposés très courts et négligeables. De plus, les communications peuvent avoir lieu en même temps que des calculs locaux (Cette hypothèse est réaliste car la plupart des processeurs disposent d'un co-processeur dédié à la communication).

Les définitions d'un ordonnancement fournies au Chapitre 2 sont valides pour des ordonnancements sans réplication. Nous étendons ici la définition des fonctions π et σ pour les ordonnancements avec réplication active.

Ainsi π fournit l'ensemble des processeurs où les tâches sont exécutées. On peut le définir comme $\pi : T \rightarrow I(M)$ ou $I(M)$ est l'ensemble des parties de M. Il peut aussi être vu comme une fonction de deux variables $\pi : T \times M \rightarrow \{0, 1\}$. Les deux notations seront utilisées en fonction de l'expressivité nécessaire. Nous écrirons également $\pi \subseteq \pi'$ si et seulement si $\forall i \in T, \pi(i) \subseteq \pi'(i)$, ou de façon équivalente, $\pi \subseteq \pi'$ si et seulement si $\forall (i, j) \in T \times M, \pi(i, j) \leqslant \pi'(i, j)$.

Nous modifions également la définition de σ pour qu'elle contienne l'information de la date de démarrage de chaque copie de chaque tâche, $\sigma : T \times M \rightarrow \mathbb{R}^+$. Pour un ordonnancement sans réplication, π est tel que : $\forall i \in T, \sum_{j \in M} \pi(i, j) = 1$. Pour un ordonnancement avec réplication, on appelle $r_i = \sum_{j \in M} \pi(i, j) = r_i$, le facteur de réplication de la tâche i.

Les défaillances sont statistiquement indépendantes et arrivent sur le processeur j suivant un processus de Poisson de paramètre λ_j appelé le taux de défaillance par unité de temps de j. Ainsi, la probabilité que j soit opérationnel durant un intervalle de temps de largeur ℓ est $e^{-\lambda_j . \ell}$. Inversement, la probabilité que j subisse une défaillance durant un intervalle de temps de longueur ℓ est $1 - e^{-\lambda_q . \ell}$. Les processeurs à silence sur défaillance modernes peuvent avoir des taux de défaillance de l'ordre de 10^{-6} par heure.

Un ordonnancement est opérationnel si et seulement si toutes les tâches sont opérationnelles. Une tâche i ordonnancée sur le processeur j est opérationnelle si et seulement si j ne subit pas de défaillance durant l'exécution de i. Ainsi, la probabilité que i soit opérationnelle sur j est

$$\mathcal{P}(i, j) = e^{-\lambda_j p_{ij}} \tag{5.1}$$

Finalement, la fiabilité d'un ordonnancement est la probabilité qu'il soit opérationnel. Nous noterons par UR la non fiabilité d'un ordonnancement, égale à 1 moins sa fiabilité.

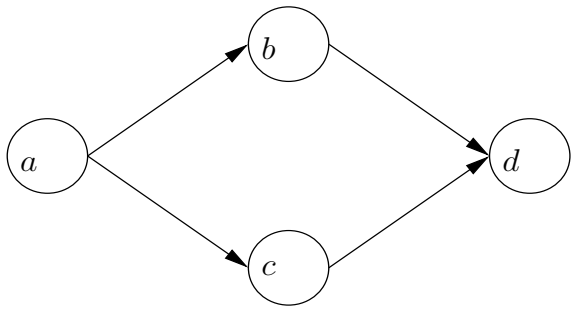


FIG. 5.1 – Un graphe application.

5.2.2 Réplication de tâches pour la fiabilité

L'idée est d'améliorer la fiabilité grâce à la réplication active de tâches. Cette technique est aussi connue sous le nom de *state machine approach* [Sch90]. Le principe est d'ordonnancer plusieurs copies de chaque tâche sur différents processeurs afin que les copies puissent être exécutées en parallèle.

Ajouter des copies améliore la fiabilité d'un ordonnancement mais en général, cela augmente aussi son makespan. En ce sens, les deux objectifs sont antagonistes.

Pour pouvoir garantir le makespan d'un ordonnancement et ne retarder aucune exécution, nous forçons une copie (i, j) à attendre la terminaison de toutes les copies de ses prédécesseurs avant de commencer sa propre exécution. Nous appelons cela la réplication pour la fiabilité (ou *replication for reliability*). Elle est différente de la réplication classique considérée en ordonnancement, où une copie n'attend que la terminaison de la première copie de tous ses prédécesseurs. Cette différence est illustrée par les Figures 5.2 et 5.3. La Figure 5.1 présente le DAG qu'il faut ordonnancer sur trois processeurs. La Figure 5.3 utilise la réplication à but d'efficacité tandis que la Figure 5.2 utilise la réplication pour la fiabilité. Dans ces figures, chaque communication inter-processeur est représentée par une flèche dont la projection sur l'axe du temps est proportionnelle au délai de communication.

Formellement, dans le cas de la réplication pour la fiabilité(Figure 5.2), les contraintes de précédence s'écrivent : $\forall (i, i') \in E, \forall j' \in \pi(i'), \sigma(i', j') \geq \max_{j \in \pi(i)}(\sigma(i, j) + p_{ij} + comm_{iji'j'})$. Dans le cas de la réplication pour l'efficacité, le max serait remplacé par un min.

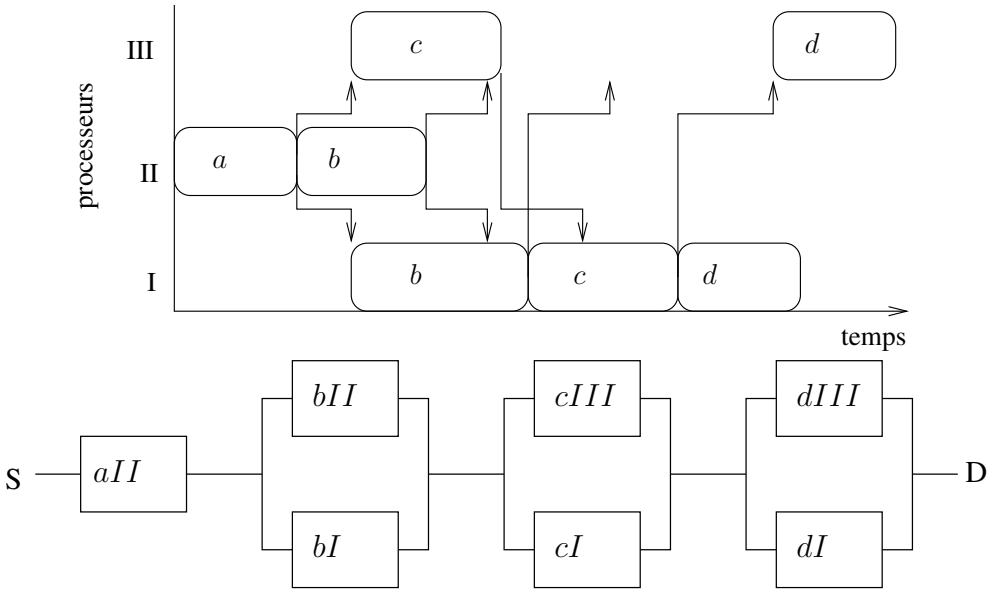


FIG. 5.2 – Illustration de la réplication pour la fiabilité sur le graphe de la Figure 5.1

5.2.3 Calculer la fiabilité d'une allocation spatiale π

Une allocation spatiale peut être représentée par un *Reliability Block Diagram* (RBD) (voir par exemple [LL62, SS98]). Formellement, un RBD est un graphe orienté (N, E), tel que chaque noeud de N est un bloc représentant un élément de l'allocation (*i.e.*, une copie d'une tâche placée sur un processeur), et chaque arc de E est un lien de causalité entre deux blocs. N possède deux noeuds particuliers, la source S et la destination D (S n'a pas de prédécesseur et D n'a pas de successeur). Un RBD est dit opérationnel si et seulement si il existe au moins un chemin opérationnel reliant S à D. Un chemin est opérationnel si et seulement si tous les blocs qui le composent sont opérationnels. La probabilité qu'un bloc soit opérationnel est sa fiabilité (donnée par l'Équation (5.1)). Par construction, la probabilité qu'un RBD soit opérationnel est égale à la fiabilité de l'allocation qu'il représente.

Quand une allocation spatiale ne contient pas de réplication (*i.e.*, un ordonnancement sans réplication), son RBD est série. En effet, il est composé d'un unique chemin de S à D où le $i$eme bloc représente l'exécution de la tâche i. Pour que l'ordonnancement soit opérationnel, il faut que toutes les tâches s'exécutent sans fautes. Ainsi le calcul de la fiabilité d'un tel RBD est linéaire dans le nombre de tâches.

Quand une allocation spatiale contient des copies, son RBD n'a pas de

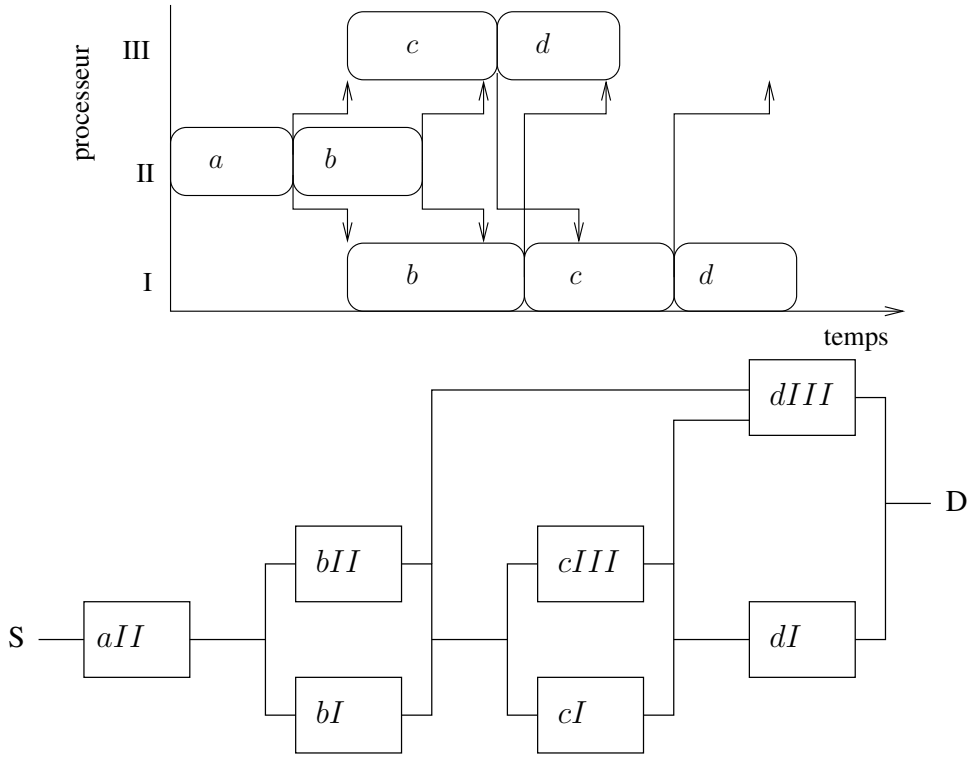


FIG. 5.3 – Illustration de la réplication pour l'efficacité sur le graphe de la Figure 5.1

forme particulière prédéfinie. Par exemple, la Figure 5.3 donne le RBD correspondant à l'ordonnancement utilisant la réplication pour l'efficacité. Soit $dIII$, soit dI doit être opérationnel (où dI désigne la copie de d sur le processeur I). D'un coté, $dIII$ ne peut pas être opérationnel si ces prédécesseurs ne sont pas opérationnels. Au moment où $dIII$ est exécuté, bI et cI ne sont pas terminés. $dIII$ ne peut recevoir ses données que depuis bII et $cIII$. D'un autre coté, dI peut être opérationnel si soit cI ou $cIII$ est opérationnel et si soit bI ou bII est opérationnel. Calculer la fiabilité d'un tel RBD ne peut se faire qu'en temps exponentiel dans la taille du RBD (sauf si $\mathcal{P} = \mathcal{NP}$). Les méthodes classiques reposent sur encodage efficace du RBD (voir, *e.g.*, AltaRica [GLR07] ou l'outil Sharpe [HSZT00]).

L'utilisation de la réplication pour la fiabilité amène à différentes propriétés. En effet, si seulement une copie de la tâche i fonctionne la tâche est toujours opérationnelle. De plus, toutes les copies de toutes les tâches qui dépendent du résultat de i peuvent toujours être exécutées sans problème. Ainsi l'ordonnancement est opérationnel si au moins une copie de chaque

tâche est opérationnelle. En conséquence, le RBD est toujours série parallèle, *i.e.*, une chaîne de macro blocs parallèles. La Figure 5.2 montre le RBD de l'ordonnancement qui utilise la réplication pour la fiabilité. La probabilité qu'un tel ordonnancement soit opérationnelle peut être calculée en temps linéaire dans la taille du RBD [LL62, SS98].

Remarquons que la fiabilité d'un ordonnancement ne dépend pas de l'allocation temporelle mais uniquement de l'allocation spatiale. Nous notons par $\mathcal{P}(\pi)$ la fiabilité de l'allocation spatiale π calculée par l'expression suivante (5.2) :

$$\begin{aligned} \mathcal{P}(\pi) &= \prod_{i\in T}\left(1-\prod_{j\in\pi(i)}\left(1-\mathcal{P}(i,j)\right)\right) \\ &= \prod_{i\in T}\left(1-\prod_{j\in\pi(i)}\left(1-e^{-p_{ij}\lambda_j}\right)\right) \\ &= \prod_{i\in T}\mathcal{P}(\pi_i) \end{aligned} \tag{5.2}$$

Par commodité, nous définissons la fiabilité de toutes les copies d'une tâche $\mathcal{P}(\pi_i) = 1-\prod_{j\in\pi(i)}\left(1-e^{-\lambda_j p_{ij}}\right)$. Ainsi nous avons, $\mathcal{P}(\pi) = \prod_{i\in T}\mathcal{P}(\pi_i)$. Cette expression de la fiabilité dérive directement des trois hypothèses : défaillances transitoire et silencieuse, indépendance statistique des défaillances et réplication pour la fiabilité. Finalement, nous notons la non fiabilité d'un ordonnancement par $UR(\pi,\sigma) = 1-\mathcal{P}(\pi)$.

5.2.4 Inapproximabilité de la solution zénith

THÉORÈME 5.1. *Le zénith du problème de minimisation simultanée de C_{max} et de UR n'est pas approximable à facteur constant.*

Démonstration. Considérons l'instance composée de deux processeurs 1 et 2 une tâche 1 tel que les temps de calcul de 1 soient : $p_{11} = 1$ et $p_{12} = k$, où k est un entier positif. Le taux de panne du processeur 1 est λ_1 tandis que celui du processeur 2 est $\lambda_2 = \frac{\lambda_1}{k^2}$.

Considérons les trois solutions possibles de cette instance : S_1 où la tâche est ordonnancée sur le premier processeur, S_2 ou elle est ordonnancée sur le second processeur et S_3 où une copie de la tâche est ordonnancée sur chaque processeur. Remarquons que S_1 est optimale pour le makespan avec $C_{max}(S_1) = 1$ et $UR(S_1) = 1-e^{-\lambda_1 p_{11}}$. S_2 est proche de la solution optimale pour UR, avec $C_{max}(S_2) = k$ et $UR(S_2) = 1-e^{-\lambda_1 p_{11}/k}$. S_3 est optimale pour UR, avec $C_{max}(S_3) = k$ et $UR(S_3) = 1-e^{-\lambda_1 p_{11}} - e^{-\lambda_1 p_{11}/k} + e^{-\frac{k+1}{k}\lambda_1 p_{11}}$.

Les facteurs $C_{max}(S_2)/C_{max}(S_1)$ et $UR(S_1)/UR(S_2)$ tendent vers l'infini avec k. Les facteurs impliquant S_3 et S_1 tendent vers l'infini avec k également. Ainsi, aucune des trois solutions du problème ne peut approcher toutes les solutions (et ainsi le zénith) à facteur constant. □

A cause de ce résultat, nous proposons de calculer un ensemble de solution de compromis parmi lesquels l'utilisateur choisira celle qui correspond le mieux à ses besoins applicatifs.

5.2.5 Des hypothèses du modèle

Dans cette section, nous discutons de la pertinence des hypothèses.

De la connaissance des temps de calcul. Nous supposons que nous connaissons les temps de calcul des tâches. En réalité, nous disposons d'une borne supérieur p_{ij} au temps réel d'exécution de la tâche i sur le processeur j. Cette borne au pire sur les temps de calcul est appelé WCET(pour *Worst Case Execution Time*). Par le passé, cette hypothèse a été largement critiquée car l'analyse des WCET semblait être un problème difficile. Cependant, il a été largement étudié récemment (voir [PB00, Lis06]). Connaître les caractéristiques des exécutions n'est pas une hypothèse critique car des analyses de WCET ont été appliquées avec succès sur des processeurs existants y compris des processeurs avec prédiction de branchement [CP00] ou encore avec des caches et de pipelines [TFW00]. En particulier, ce style d'analyse a été appliqué aux processeurs des systèmes embarqués les plus critiques, l'Airbus A380 dont le logiciel fonctionne sur un processeur Motorola MPC755 [FHL+01, SPH+05].

De l'efficacité de la réplication pour la fiabilité. Dans des approches de tolérances aux fautes, une tâche n'a pas besoin d'attendre la terminaison de toutes les copies de ses prédécesseurs. Il faut uniquement attendre la terminaison de la première copie. Ici, nous visons la fiabilité et non la tolérance aux fautes. Ainsi, nous devons nous assurer que le Reliability Block-Diagram qui représente l'ordonnancement est série-parallèle, sinon, le calcul de la fiabilité ne serait pas réalisable. C'est pour cette raison que nous adoptons la réplication pour la fiabilité. Nous pouvons nous demander si cela induit un surcoût important sur le système. Girault et Kalla [GKar] montrent sur un modèle de réplication proche que l'impact sur le makespan final est relativement faible. C'est pourquoi nous pensons que la réplication pour la fiabilité est utilisable et réaliste.

Des liens de communication pleinement fiable. Si les liens de communications sont soumis aux défaillances alors le RBD ne sera pas série-parallèle. Ainsi calculer la fiabilité devient un problème compliqué. De plus, avoir des défaillances sur les liens de communication est bien plus difficile car il faudrait traiter des problèmes supplémentaires comme celui du routage. Finalement, les défaillances sur les liens de communications causeraient des

fautes non statistiquement indépendantes sur les successeurs d'une tâche. [Kal04] traite ce problème en ajoutant pour chaque tâche i une tâche de synchronisation qui dépend de toutes les copies de i et dont dépend tous les successeurs de i. Cela dégrade les performances et la fiabilité de l'application mais rend le problème tractable.

5.3 Résultats connexes

5.3.1 Optimisation du makespan sur machine hétérogène

L'ordonnancement hors ligne pour l'optimisation du makespan sur des ressources homogènes a été largement étudié. Bien que la plupart des variantes soient $\mathcal{NP}$-difficiles, il est souvent possible d'obtenir des analyses théoriques pour construire des algorithmes d'approximation. En calcul parallèle, l'ordonnancement sur des ressources hétérogènes est un problème plus récent et plus difficile. Les résultats existants sont composés d'heuristiques fines mais sans propriétés théoriques intéressantes. L'hypothèse de tâches indépendantes est l'une des seules qui amène à un algorithme à performances garanties (voir Section 2.2.3).

Pour ordonnancer des graphes de tâches, les méthodes les plus classiques sont des extensions des algorithmes de liste. HEFT (pour *Heterogeneous Earliest First Task*) [Top02] trie les tâches par ordre décroissant de longueur moyenne du chemin critique restant. C'est un algorithme glouton : chaque tâche est considérée à son tour et est ordonnancée sur le processeur qui la termine au plus tôt. HEFT est souvent utilisé comme heuristique de référence quand il s'agit de faire des expériences. Cet algorithme a été amélioré dans [ZS03] en utilisant un autre ordre des tâches.

L'heuristique min-min et ses variantes [IK77] considèrent pour chaque tâche le processeur qui la termine le plus tôt. Min-min alloue la tâche de plus petit temps de terminaison pour optimiser l'utilisation des processeurs (dans ce sens, il s'agit d'une approche orthogonale à HEFT). La variante max-min alloue la tâche de plus grand temps de terminaison. L'idée ici est de prendre en compte le plus tôt possible les tâches qui sont très pénalisantes pour le makespan global. Le nombre de variantes proposées reflète bien le côté heuristique de cette méthode. Bien que ces algorithmes aient été construits pour des tâches indépendantes, ils peuvent facilement être adaptés pour des graphes de tâches.

D'autre approches sont basées sur le regroupement (ou *clustering*), dont le principe est de forcer les tâches qui communiquent beaucoup entre elles à

être ordonnancées sur le même processeur. Les tâches du même groupe sont exécutées sur la même ressource. Finalement, les dates de début des groupes sont déterminées à l'aide d'un algorithme classique d'ordonnancement. [CJ01] est un exemple de technique de regroupement pour les systèmes hétérogènes.

5.3.2 Optimisation de la fiabilité

Rappelons d'abord que la fiabilité est augmentée en répliquant les tâches. La fiabilité maximale est alors obtenue en exécutant une copie de chaque tâche sur tous les processeurs. Lorsque les liens de communications sont fiables, le calcul de la fiabilité du RBD induit peut être effectué en temps linéaire. En présence de défaillances sur les liens de communications, le calcul de la fiabilité du RBD est $\mathcal{NP}$-difficile. Ainsi il n'existe pas d'algorithme polynomial (sauf si $\mathcal{P} = \mathcal{NP}$) pour la calculer. Cependant, il est possible de construire des algorithmes exponentiels mais efficace en encodant le RBD à l'aide de BDD (pour *Binary Decision Diagram*) [HSZT00, GLR07].

Une autre façon de calculer la fiabilité d'un RBD est d'ajouter des tâches de synchronisation pour rendre le graphe série-parallèle. Bien que le calcul de la fiabilité devienne polynomial, il y a un surcoût lié à ces tâches de synchronisation (détails dans [Kal04]).

[SJ99] optimise la fiabilité en utilisant des techniques de regroupement pour minimiser les temps de communications. Ainsi, les communications les plus coûteuses sont allouées aux liens les plus fiables tandis que les groupes de tâches les plus longs à traiter sont alloués aux processeurs les plus fiables.

Nos hypothèses sur les fautes rendent facile le calcul de l'optimisation fiabilité.

5.3.3 Optimisation simultanée des deux objectifs

Dans [AGK04], les auteurs proposent une heuristique pour un problème similaire au notre où les liens de communications ne sont pas fiables. Les auteurs calculent une borne supérieure de la fiabilité à l'aide d'une méthode basée sur les coupes minimales du RBD. L'heuristique proposée optimise gloutonnement une combinaison linéaire des deux objectifs, normalisée à l'aide de seuils fournis par l'utilisateur.

[DÖ02] propose un problème d'ordonnancement bi-objectif pour le cas de réseau de processeurs non entièrement connecté. La fiabilité exacte est $\mathcal{NP}$-difficile à calculer, ce qui rend le problème plus difficile. Un algorithme de liste optimisant le makespan DLS (pour *Dynamic Level Scheduling*) [SL93] est adapté en RDLS (pour *Reliable DLS*) pour prendre en compte la fiabilité.

DLS est un algorithme glouton pour l'ordonnancement sur ressources hétérogènes qui alloue la paire (tâche, processeur) qui a le plus grand *Dynamic Level* (il prend en compte l'allocation existante dans le calcul du chemin critique). RDLS ajoute un terme au *Dynamic Level* pour prendre en compte la fiabilité des processeurs. Cependant, aucune tâche n'est répliquée et l'impact sur la fiabilité est limité.

5.4 Décomposition du problème en deux phases

L'analyse du problème nous amène aux faits suivants. Le zénith du problème n'est pas approximable à facteur constant (Théorème 5.1). Nous nous intéressons alors à l'approximation de l'ensemble de Pareto. Cependant le simple problème d'optimisation du makespan est traité uniquement avec des heuristiques car on ne connaît pas d'algorithme d'approximation (Section 5.3.1). Nous allons donc construire une heuristique dans la veine des algorithmes de $\langle \bar{\alpha}, \beta \rangle$-approximation, qui ne sera pas garantie. Cependant, nous portons une attention particulière à l'utilisation des propriétés des solutions optimales du problème.

5.4.1 Principe

Pour suivre la construction des ensembles approchés de Pareto, nous posons un seuil sur un objectif qui sera fixé successivement à différentes valeurs. La fiabilité d'une solution $S = (\pi, \sigma)$ dépend uniquement de π. Il est ainsi plus simple de poser un seuil sur *UR*. Nous commençons par calculer une allocation spatiale π qui satisfasse le seuil fixé *UR*. Puis nous choisirons une allocation temporelle σ pour optimiser le makespan en respectant π. Cependant, il y a de nombreuses allocations spatiales qui satisfont le seuil, mais potentiellement peu qui mène à de bons makespans. Distinguer ces quelques allocations spatiales n'est pas facile *a priori* ; ce point sera détaillé en Section 5.4.4. Ainsi, nous allons générer les allocations spatiales aléatoirement en Section 5.4.2. Une descente locale sera appliquée ensuite pour post-optimiser ces allocations

Une fois les tâches associées aux processeurs, nous nous occupons uniquement d'optimiser le makespan en construisant l'allocation temporelle σ. À π fixé, c'est exactement le problème "classique" d'ordonnancement pré-alloué. Nous le traitons en Section 5.4.3.

5.4.2 Phase 1 : Génération d'allocation spatiale

Propriétés et algorithme

Étudions tout d'abord un ensemble de propriétés sur les allocations spatiales qui seront utiles pour la construction d'un algorithme.

La propriété suivante définit une relation d'ordre partiel entre les allocations spatiales.

PROPRIÉTÉ 5.2. *Soient π et π' deux allocations spatiales, $\pi \subset \pi' \Rightarrow UR_\pi > UR_{\pi'}$.*

Démonstration. Pour chaque $(i', j') \in \pi' \backslash \pi$, nous avons $0 \leqslant \mathcal{P}(i', j') \leqslant 1$, ou de façon équivalente :

$$0 \leqslant 1 - \mathcal{P}(i', j') \leqslant 1 \tag{5.3}$$

Selon l'Équation (5.2), $\mathcal{P}(\pi) = \prod_{i \in T}(1 - \prod_{j \in \pi(i)}(1 - \mathcal{P}(i, j)))$. Grâce à l'Équation (5.3), $0 \leqslant \prod_{(i',j') \in \pi' \backslash \pi}(1 - \mathcal{P}(i', j')) \leqslant 1$. Ainsi, multiplier ce terme par le terme $\prod_{j \in \pi(i)}(1 - \mathcal{P}(i, j))$ amène à un terme plus petit :

$$\prod_{(i',j') \in \pi' \backslash \pi} (1 - \mathcal{P}(i', j')) \times \prod_{j \in \pi(i)} (1 - \mathcal{P}(i, j)) \leqslant \prod_{j \in \pi(i)} (1 - \mathcal{P}(i, j))$$

En conséquence, $\forall i \in T$, le terme $1 - \prod_{j' \in \pi'(i')}(1 - \mathcal{P}(i', j'))$ est plus grand que le terme $1 - \prod_{j \in \pi(i)}(1 - \mathcal{P}(i, j))$. D'où, $\mathcal{P}(\pi') \geq \mathcal{P}(\pi)$. □

Cette propriété peut être utilisée par un algorithme glouton pour améliorer la fiabilité d'une allocation spatiale en ajoutant des copies de tâches. Les deux propriétés suivantes aident à choisir la tâche dont il faut rajouter une copie. La Propriété 5.3 énonce que la fiabilité d'une allocation spatiale ne peut pas être meilleure que la fiabilité de la tâche la moins fiable. La Propriété 5.4 énonce qu'il existe une tâche dont la fiabilité est supérieure à la racine $n$ieme de la fiabilité de l'allocation.

PROPRIÉTÉ 5.3. *Soit π une allocation spatiale, telle que $UR(\pi) < UR_0$, alors $\forall i \in T, \mathcal{P}(\pi_i) > 1 - UR_0$.*

PROPRIÉTÉ 5.4. *Soit π une allocation spatiale telle que $UR(\pi) < UR_0$, alors $\exists i \in T, \mathcal{P}(\pi_i) > \sqrt[n]{1 - UR_0}$.*

Démonstration. Par contradiction. Supposons que, pour toutes les tâches i, nous avons $\mathcal{P}(\pi_i) \leqslant \sqrt[n]{1 - UR_0}$. La fiabilité de l'allocation spatiale est $\mathcal{P}(\pi) = \prod_{i \in T} \mathcal{P}(\pi_i)$. Les $\mathcal{P}(\pi_i)$ sont des probabilités, d'où $0 \leqslant \mathcal{P}(\pi_i) \leqslant 1$. Ainsi, $\mathcal{P}(\pi) \leqslant (\max_{i \in T} \mathcal{P}(\pi_i))^n$. L'hypothèse mène à la borne suivante sur la non fiabilité de l'ordonnancement : $UR(\pi) = 1 - \mathcal{P}(\pi) \geqslant 1 - \sqrt[n]{1 - UR_0}^n \geqslant UR_0$. Ce qui mène a une contradiction. □

Nous présentons maintenant l'algorithme `IRSAG` (pour *Iterative Randomized Spatial Allocation Generator*) qui construit une allocation spatiale de non fiabilité inférieure au seuil UR_0 (voir Figure 5.1). Le principe est d'ajouter des copies de tâches à une allocation déjà existante. L'algorithme commence par remplir les pré-requis de la Propriété 5.3 ; c'est-à-dire que pour chaque tâche il ajoute des copies tant que la non fiabilité de la tâche est inférieure au seuil. Si après cela, aucune tâche ne satisfait la Propriété 5.4, une tâche est aléatoirement choisie et des copies sont ajoutées aléatoirement jusqu'à ce que la propriété soit vérifiée. Finalement, des copies de tâches sont ajoutées jusqu'à ce que le seuil de non fiabilité soit atteint.

Remarquons que `IRSAG` peut aussi fonctionner avec une allocation initiale π_0 et pas seulement à partir de l'ensemble vide.

`IRSAG` utilise une distribution uniforme pour choisir les copies (i, j) qu'il faut ajouter. La dernière partie de l'algorithme assure que les copies sont réparties équitablement entre les tâches. Il semble que cela soit une bonne propriété compte tenu de la structure de la fonction de fiabilité (Équation (5.2)).

Amélioration

`IRSAG` renvoie une allocation qui a une non fiabilité plus petite que UR_0. Mais `IRSAG` ne garantie pas que cette allocation est la meilleure satisfaisant cette propriété. L'allocation retournée n'est même pas forcément minimale par inclusion. A cause du modèle de réplication pour la fiabilité, retirer des copies de tâches ne peut qu'améliorer C_{max}. C'est pourquoi nous proposons la procédure de descente locale `maopt` (pour *Minimal Allocation Optimization*). Elle retire itérativement des copies tant que la non fiabilité reste inférieure à UR_0. Les copies sont triées en ordre décroissant de temps de calcul afin de retirer en priorité les copies les plus longues.

5.4.3 Phase 2 : ordonnancement pré-alloué

Comme nous utilisons la réplication pour la fiabilité (Figure 5.2), nous ne pouvons pas réutiliser les résultats classiques d'ordonnancement avec duplication, car tous ces travaux adressent le modèle classique de duplication (voir la Figure 5.3) et la discussion associée).

Trouver la meilleur allocation temporelle σ à allocation spatiale π fixée est équivalent à ordonnancer un DAG de tâches où les tâches sont associées à un processeur donné. En effet toutes les copies sont équivalentes, dans le sens où il n'y a plus de distinction entre les copies d'une même tâche et les copies de tâches différentes. Ainsi, dans cette section, le terme de tâche fait référence à une copie d'une tâche sur un processeur. Ce problème est connu sous le nom

Algorithme 5.1 IRSAG : Spatial allocation generation

Input : une instance, un seuil UR_0 et une allocation π_0
Output : une allocation π
Begin
$\pi := \pi_0$
$prop3 := false$
Forall $t \in T$
  While $\mathcal{P}(\pi_t) < 1 - UR_0$
    $\pi := \pi \cup (t, alea(1, m))$
    If $\mathcal{P}(\pi_t) > \sqrt[n]{1 - UR_0}$ **Then**
      $prop3 := true$
    End If
  EndWhile
End Forall
If $prop3 = false$ **Then**
  $t_0 := alea(1, n)$
  While $\mathcal{P}(\pi_{t_0}) < \sqrt[n]{1 - UR_0}$
    $\pi := \pi \cup (t_0, alea(1, m))$
  End While
End If
While $(UR(\pi) > UR_0)$
  If $\pi = T \times Q$ **Then Return** NIL
$\pi := \pi \cup (alea(1, n), alea(1, m))$
End While
Return π
End

de problème d'ordonnancement avec pré-allocation et a été montré fortement $\mathcal{NP}$-complet [RSBJ95]. Aucun algorithme d'approximation de ce problème n'est connu.

La Définition 5.5 mène à une propriété de dominance faible pour les ordonnancements et devrait nous aider à résoudre le problème. Nous introduisons une notion de priorité entre les tâches pour caractériser une allocation temporelle sur un processeur. Quand deux tâches i et i' sont disponibles, si i a une plus grande priorité que i', alors i est ordonnancée avant i'. i est une **tâche communicante** si un des successeurs de i est ordonnancé sur un processeur différent de celui sur lequel i est ordonnancé.

DÉFINITION 5.5. *Soit σ un ordonnancement. Pour chaque processeur j, $\{tc_1^j, \ldots, tc_{k_j}^j\}$ représente l'ensemble des tâches communicantes de j dans*

l'ordre de σ. L'ordonnancement est dit communication friendly si pour tout processeur j et pour chaque paire de tâches communicantes $tc_i^j, tc_{i'}^j$ telle que $i < i'$, les prédécesseurs non communicant de tc_i^j ont une plus grande priorité que les prédécesseurs non communicants de $tc_{i'}^j$.

PROPRIÉTÉ 5.6. *Soit σ un ordonnancement valide qui n'est pas communication friendly, alors il existe un ordonnancement communication friendly de makespan inférieur ou égal.*

Démonstration. Pour chaque processeur, nous déduisons depuis σ l'ordre total des tâches communicantes. En inversant itérativement les tâches consécutives pour correspondre à la Définition 5.5, le makespan ne peut que diminuer car les tâches communicantes sont exécutées plus tôt (ou à la même date) que dans l'ordonnancement original. □

La preuve nous donne un algorithme pour améliorer les allocations temporelles déjà existantes. Nous avons cependant besoin d'en avoir une pour appliquer l'algorithme. Nous pouvons en générer à l'aide d'algorithme de liste.

Dans la littérature, les algorithmes de liste sont largement utilisés en ordonnancement. Le principe est de ne pas avoir de processeurs inactifs si une tâche est disponible [Gra69]. Il arrive fréquemment que plusieurs tâches soient disponible simultanément, une fonction de priorité est alors utilisée pour résoudre ce problème. Le *B-Level* (pour *Bottom Level*) d'une tâche est une borne inférieure classique du temps minimum de terminaison d'une application à partir de la date d'ordonnancement de cette tâche. On peut voir cette quantité comme le temps de calcul de l'application sur un nombre infini de processeurs. Le *B-Level* d'une feuille du DAG (*i.e.*, une tâche sans successeur) est son temps d'exécution, tandis que le *B-Level* d'une tâche intermédiaire est la somme de son temps de calcul avec le plus grand *B-Level* de ses successeurs. Notre problème étant pré-alloué, nous pouvons prendre en compte les temps de communication lors du calcul du *B-Level* ; cela reste une borne inférieure du makespan optimal.

L'algorithme `CommBlevelList` se concentre sur les tâches communicantes en fixant la priorité des tâches non communicantes à la priorité de son successeur le plus prioritaire. La priorité des tâches communicantes est fixée à leur *B-Level* (voir Algorithme 5.2). L'algorithme contient une boucle sur le temps pour calculer les dates de démarrage des tâches. Cette boucle est présente pour des raisons de clarté : un algorithme plus efficace peut être écrit à l'aide de tas pour remplacer les itérations sur le temps.

Nous nous intéressons maintenant au rapport d'approximation de `CommBlevelList`. La proposition suivante indique qu'aucune approximation cons-

Algorithme 5.2 CommBlevelList

Input : une instance et une allocation π
Output : une allocation temporelle σ
Begin
Soit $blevel[i]$ la longueur du plus long chemin entre i et la fin du graphe.
Forall $i \in T$ en ordre inverse topologique
  If i est une tâche communicante ou une feuille **Then**
    $prio[i] := blevel[i]$
  Else
    $prio[i] := \max_{i' | \exists (i,i') \in E} prio[i']$
  End If
End Forall
Forall unité de temps x de 0 à ∞
  Forall $j \in M$
    If j is idle at x
      Sélectionner i disponible sur j au temps x minimisant $prio[i]$
      Ordonnancer i dans σ de x à $x + p_i$ sur j
    End If
  End Forall
End Forall
Return σ
End

tante ne peut être obtenue par un algorithme de liste LS.

PROPOSITION 5.7. *Soit LS un algorithme de liste. Il existe asymptotiquement des instances du problème d'ordonnancement pré-alloué telles que* $C_{max}^{LS} \geqslant (m-1)C_{max}^{*}$.

Démonstration. Cette proposition est prouvée par la construction d'une instance qui atteint asymptotiquement la borne $(m-1)$. Le résultat est valide pour n'importe quel algorithme de liste. Ainsi, l'ordonnancement ne devrait pas être fonction de la liste de priorité de l'algorithme. Seulement une tâche devrait être disponible par processeur à tout moment.

L'instance sur m processeurs est la suivante. Le processeur 1 a une unique tâche t_1^1 de temps de calcul $p_{11} = \varepsilon$. Tous les autres processeurs j $(2 \leqslant j \leqslant m)$ ont deux tâches t_j^1 et t_j^2 de temps de calcul $p_{j1} = \varepsilon$ et $p_{j2} = 1$. Chaque tâche t_j^i $(j \neq 1)$ est un successeur de t_{j-1}^1. Le délai de communication entre t_{j-1}^1 et t_j^1 est ε, tandis que le délai entre t_{j-1}^1 et t_j^2 est $\frac{\varepsilon}{2}$.

Le makespan optimal de cette instance est obtenu en ordonnançant toutes

les tâches t_j^1 au plus tôt et en exécutant ensuite toutes les tâches t_j^2 en parallèle, sauf pour le processeur m où la tâche t_m^2 est ordonnancée avant t_m^1. Ainsi, le makespan optimal est $C_{max}^* = (2m - \frac{3}{2})\varepsilon + 1$.

Cependant, conformément au principe des algorithmes de liste, dans la solution de LS, les tâches t_j^2 sont exécutées avant les tâches t_j^1 car les temps de communications sont plus courts avant t_j^1. Cela retarde la mise à disposition des tâches sur le processeur suivant. Dans ce cas, toutes les tâches sont exécutées séquentiellement. Le makespan résultat est : $C_{max}^{LS} = m\epsilon + \frac{(m-1)\epsilon}{2} + m - 1$.

En conséquence, le rapport $\frac{C_{max}^{LS}}{C_{max}^*}$ tends vers $(m-1)$ quand ε tend vers 0. □

En d'autre terme, cette proposition énonce que pour obtenir un rapport d'approximation constant, un algorithme doit être capable d'attendre les tâches importantes au lieu d'exécuter une tâche qui ne l'est pas. Le problème principal dans la conception d'algorithme d'approximation pour ce problème est le manque de "bonne" borne inférieure sur le makespan optimal. En effet, les bornes inférieures classiques telles que le chemin critique ou la charge maximale de travail d'un processeur ne sont pas à facteur constant du makespan optimal. Dans ce problème, nous ne connaissons pas de critère permettant de distinguer les tâches importantes des tâches qui ne le sont pas.

5.4.4 Discussion

Nous avons proposé un algorithme en deux phases où la première génère une allocation et la seconde phase génère une allocation temporelle. Cette décomposition est pertinente car la fiabilité d'un ordonnancement dépend uniquement de l'allocation spatiale et pas de l'ordre local des tâches. Dans notre méthode, l'allocation spatiale est générée aléatoirement à l'aide de propriétés qui semblent pertinentes. On peut se demander si il est possible d'obtenir plus à l'aide d'un algorithme déterministe plus sophistiqué.

La première phase ne donne pas directement le makespan de l'ordonnancement. Elle assure simplement que la fiabilité est supérieure à un seuil donné. Il est très difficile d'optimiser le makespan dans cette phase car il ne sera pas connue tant que la seconde phase ne sera pas terminée. Ainsi, la première phase devrait générer une allocation spatiale qui sera ordonnancée efficacement en phase 2. Cependant, le problème d'ordonnancement traité en phase 2 est difficile et comme nous l'avons montré dans la section précédente, les algorithmes classiques comme *List Scheduling* ne peuvent garantir une approximation constante. C'est pourquoi des bornes inférieures utiles sur le makespan ne peuvent pas être facilement déterminées. Les bornes inférieures que nous connaissons ne sont pas suffisantes car nous pouvons obtenir

des makespans très différents à partir de deux instances qui ont les mêmes bornes inférieures. Tant que nous ne saurons pas quelles fonctions optimiser en phase 1, la génération aléatoire d'allocation spatiale semble être la seule politique raisonnable.

5.5 Étude expérimentale

Dans cette section, nous présentons des résultats expérimentaux autour de la méthode proposée dans la section précédente. Il n'y a pas d'intérêt à comparer notre méthode avec les méthodes qui n'utilisent pas de duplication de tâches car elles n'améliorent pas la fiabilité de plus qu'un ordre [DÖ05, DÖ02]. Ainsi, nous ne comparons pas notre algorithme avec d'autres algorithmes bi-objectifs.

5.5.1 Buts des expériences

Valider l'efficacité de la méthode est difficile car extraire l'ensemble de Pareto d'une instance requiert un temps exponentiel. Nous essayons de valider notre méthode en étudiant les quelques points suivants :

- Pour être compétitif, l'algorithme pour le problème d'ordonnancement pré-alloué doit être efficace. S'il ne l'est pas, la méthode ne fournira pas de résultats intéressants, même si la première phase fournissait la meilleure allocation spatiale possible.
- Il est intéressant d'étudier l'impact de la qualité de l'allocation spatiale sur le makespan. En particulier, on devrait considérer le changement du nombre d'itérations de l'algorithme aléatoire et l'utilisation de la descente locale.
- Les deux premières questions sont cruciales pour améliorer et valider la méthode. Cependant, la question principale est : peut-on obtenir un ensemble intéressant de solutions de compromis avec une telle méthode ? Est-ce que notre méthode est compétitive avec les méthodes d'ordonnancement optimisant uniquement le makespan ?

5.5.2 Construction du benchmark

Nous construisons une instance du problème d'ordonnancement hétérogène en assemblant d'un coté, un graphe d'application avec des temps de calcul et de communication et d'un autre coté un réseau de processeurs de différentes vitesses et des liens de communications de différentes bandes passantes.

Génération du graphe d'application

Nous considérons une instance du problème d'ordonnancement statique avec contraintes de précédences $P \mid prec, p_i, c_{ij} \mid C_{max}$ [Leu04a]. Une instance classique de ce problème est décrite par m processeurs, un graphe d'application $G = (V, E)$, des temps de calculs p_v pour chaque tâche $v \in V$ et des tailles de données $Comm_{ij}$ entre les tâches, pour tout $(i, j) \in E$.

Nous utilisons un benchmark existant [KA98] qui contient 602 instances de structures variées. Ce benchmark a déjà été utilisé pour tester différents algorithmes d'ordonnancement [DLMM04, KB03].

Génération du réseau de processeurs

Un ensemble de processeurs est caractérisé par le nombre de processeurs m, leurs vitesses $speed_j$, les taux de défaillance par unité de temps λ_j, ainsi que la bande passante du lien entre deux processeurs $BW_{jj'}$.

Selon [QJS02, DÖ05], dans les systèmes réels, le rapport entre les vitesses de calcul et entre deux processeurs et les bandes passantes entre deux différent liens sont uniformément distribués dans $[1, 10]$. Les systèmes réels ont des taux de défaillance uniformément réparti dans $[10^{-6}/h; 10^{-5}/h]$. Nous générons aléatoirement 60 réseaux de processeurs avec des nombres de processeurs entre 5 et 10.

Construction des instances

À partir d'une application et d'un réseau de processeurs, nous construisons une instance pour le problème d'ordonnancement hétérogène. Une tâche i ordonnancée sur le processeur j prend $p_i/speed_j$ unité de temps à calculer.

Nous avons extrait un ensemble de 275 grands graphes du benchmark. Bien qu'elles étaient traitables séparément en un temps raisonnable (de l'ordre de 5 minutes par instances), traiter les 16200 instances (275×60) aurait pris un temps prohibitif. Nous avons conservé les 327 autres graphes. Ainsi nos expériences sont effectuées sur 19620 instances (327×60).

Les instances que nous avons générées ne sont certainement pas uniformément distribuées sur l'ensemble des instances et ne sont certainement pas "représentatives" d'un champ d'application. Ainsi, les conclusions que nous en tirons ne sont valides que pour ce benchmark particulier. Cependant, cela nous permet de comprendre le comportement général de l'algorithme.

5.5.3 Protocole

Notre méthode n'est pas monolithique, plusieurs options et paramètres peuvent varier. D'abord, notre algorithme est aléatoire, nous pouvons changer le nombre de tirages. Nous avons considéré les nombres d'itérations suivant : 1, $\log nm$, and $\sqrt{nm}$. Aussi, la descente locale `maopt` peut être utilisée ou non. Finalement, la première phase de l'algorithme peut être initialisée avec différentes allocations : nous considérons l'allocation vide et l'allocation de HEFT. Rappelons que la méthode prend une fiabilité minimale en paramètre.

Les expériences sont conformes au protocole suivant. D'abord HEFT [Top02] est exécuté sur chaque instance. Soient C_{max}^{HEFT} et UR^{HEFT} les valeurs objectif obtenues par l'ordonnancement produit par HEFT. Notre algorithme est ensuite exécuté avec différents seuils sur UR. Chaque seuil de UR est obtenu en multipliant UR^{HEFT} par une valeur r, où r prend les valeurs suivantes : 1, 0.9, 0.8, 0.7, 0.1, 0.01, 0.001, 0.0001, ou 0.00001. Par exemple, si $UR^{HEFT} = 0.1$ et $r = 0.1$, alors notre seuil UR est 0.01 et inversement, la fiabilité minimale que nous acceptons est 0.99.

Nous avons aussi fait ces expériences en "condition réelle". C'est-à-dire que comme dans l'algorithme d'approximation de l'ensemble de Pareto, nous avons fixé des seuils suivant une suite géométrique croissante entre UR_{min} et UR_{max}. Seulement les solutions Pareto-indépendantes sont conservées. Cette méthode correspond à l'application en environnement de production de notre méthode ainsi, nos simulations peuvent confirmer son utilisation en pratique.

5.5.4 Résultats et analyse

Les résultats sont donnés pour chaque r comme la moyenne géométrique sur toutes les instances des rapports entre le makespan de notre algorithme et le makespan de HEFT. Nous utilisons des moyennes géométriques car ce sont celles qui ont du sens lorsque l'on considère des rapports [Jai91].

Pour vérifier l'efficacité de notre algorithme d'ordonnancement pré-alloué, nous comparons le makespan obtenu par HEFT et le makespan de notre algorithme basé sur le *B-Level* exécuté sur l'allocation de HEFT. Le rapport moyen entre les performances de notre algorithme et celles de HEFT est 1.01009, le rapport minimal est 0.885272 et le rapport maximal est 2.29752. Ces résultats montrent que bien qu'il n'y ait pas de garantie de performance sur notre algorithme, les performances de l'algorithme de seconde phase fournit de bonne performance en pratique.

Nous testons l'impact d'une mauvaise allocation spatiale en comparant les résultats sans descente locale (courbe `iterlog`) et avec la descente locale `maopt` (courbe `maoptiterlog`). L'algorithme d'allocation aléatoire a été ap-

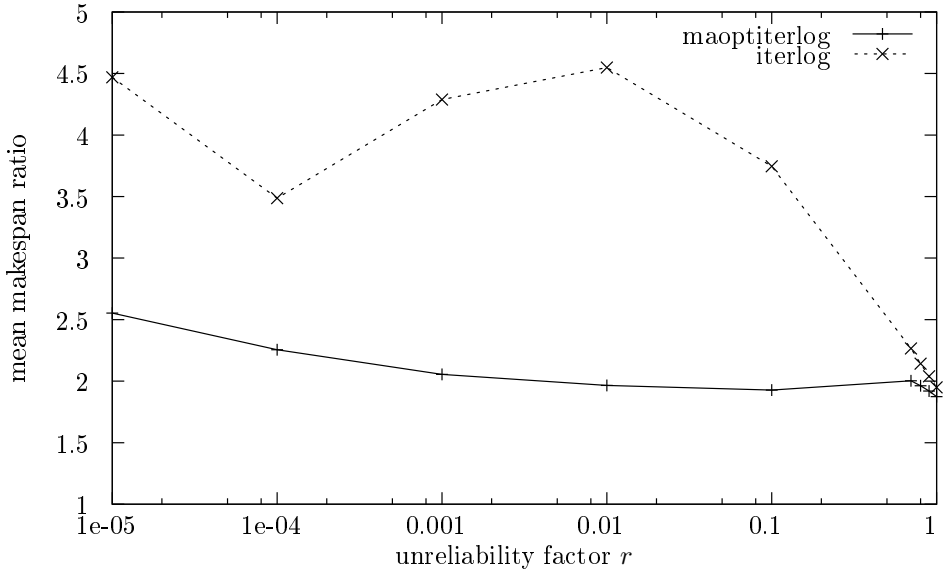


FIG. 5.4 – Comparaison des allocations aléatoires avec et sans descente locale

pelé $\log nm$ fois. Les résultats sont présentés en Figure 5.4. Elle montre pour chaque valeur de r, le rapport moyen entre le makespan obtenu et celui de HEFT. Sur la courbe `maoptiterlog`, le point $r = 0.1$ de rapport moyen sur le makespan de 2 peut être interprété comme : sur ce benchmark, pour gagner deux ordres sur la fiabilité, il faut en moyenne dégrader le makespan d'un facteur 2 ; Les résultats montrent que l'algorithme de descente locale `maopt` améliore réellement le makespan car la courbe `maoptiterlog` est largement plus basse que la courbe *iterlog*. Obtenir une bonne allocation spatiale pour optimiser le makespan est réellement important.

Des résultats étranges apparaissent pour $r = 10^{-4}$ et $r = 10^{-3}$ sans la descente locale `maopt`. Ils sont dus à un effet de seuil dans les Propriétés 5.4 et 5.3. En d'autres termes, la Propriété 5.4 est quasiment inutile pour les ordonnancements valide pour $r \geqslant 0.01$. Seulement une copie de chaque tâche est nécessaire pour la propriété. Cependant, il faudra répliquer la plupart des tâches pour obtenir la fiabilité requise. Lorsque $0.0001 \leqslant r \leqslant 0.01$, le nombre moyen de copies qu'il faut avoir pour être en accord avec la Propriété 5.4 est plus proche du nombre de copie qu'il faut pour obtenir la fiabilité requise.

La Figure 5.5 montre les résultats obtenus en changeant le nombre d'itérations de l'algorithme. Trois nombres d'itérations ont été considérés : $\sqrt{nm}$, $\log(nm)$, et 1, qui correspondent aux courbes `maoptitersqrt`, `maoptiterlog`, et `maoptiter1`. Le plus grand nombre d'itérations testé a été $\sqrt{nm}$ car les temps de calcul requis pour nm itérations devenaient prohibitifs (quelques

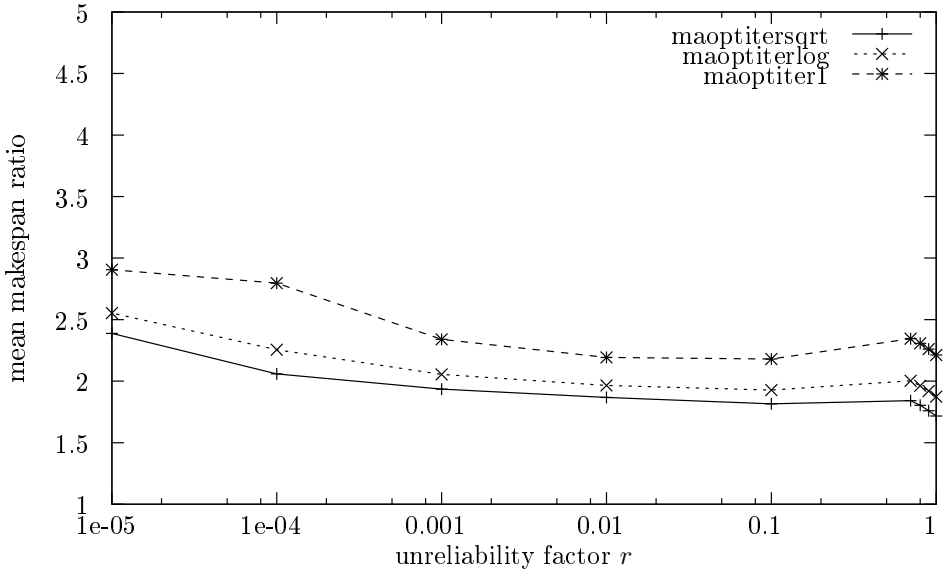


FIG. 5.5 – Impacte du nombre de générations aléatoire sur le makespan

minutes pour de petites instances contre quelques secondes avec $\sqrt{nm}$ itérations). De plus, il semble difficile de faire mieux que ce que l'on a obtenu avec ces paramètres car la différence entre les courbes `maoptiterlog` et `maoptitersqrt` est bien plus faible qu'entre `maoptiter1` et `maoptiterlog`. Nous pouvons conjecturer (bien qu'il faudrait le confirmer expérimentalement) qu'augmenter le nombre d'itérations n'augmenterait pas significativement les performances. La valeur moyenne obtenue pour $r = 0.1$ est plutôt étrange : le makespan moyen pour $r = 0.1$ est meilleur que le makespan moyen pour $r = 0.7$. Ce comportement se voit mieux lorsqu'il n'y a qu'une seule itération de l'algorithme aléatoire. Cela montre que la descente locale `maopt` devient plus efficace quand r augmente, *i.e.*, quand il y a plus de réplication.

Bien que la seconde phase de l'algorithme soit plutôt efficace, notre méthode ne fournit globalement pas de solutions de makespan proche de celui de HEFT. Ainsi, la première phase de l'algorithme ne renvoie pas d'allocation spatiale efficace pour le makespan. Tous ces résultats montrent que les allocations spatiales complètement aléatoires ne sont pas bonnes pour le makespan. Initialiser notre allocation aléatoire avec une allocation existante efficace pour le makespan devrait améliorer les makespans obtenus pour les facteurs de fiabilité proches de 1. La Figure 5.6 confirme que l'initialisation des allocations spatiales avec celle de HEFT améliore le makespan pour les petites fiabilités. Cependant, cela semble inutile si l'on souhaite gagner un

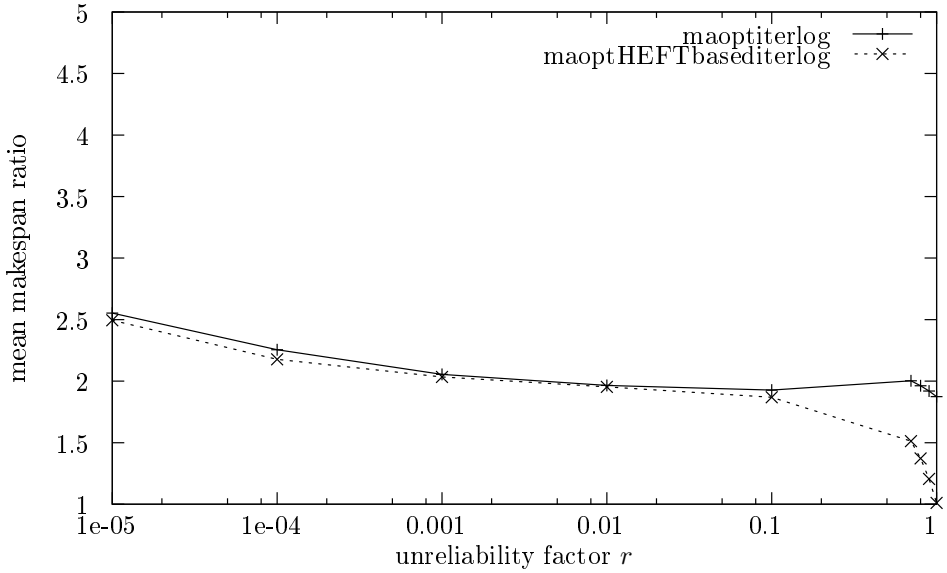
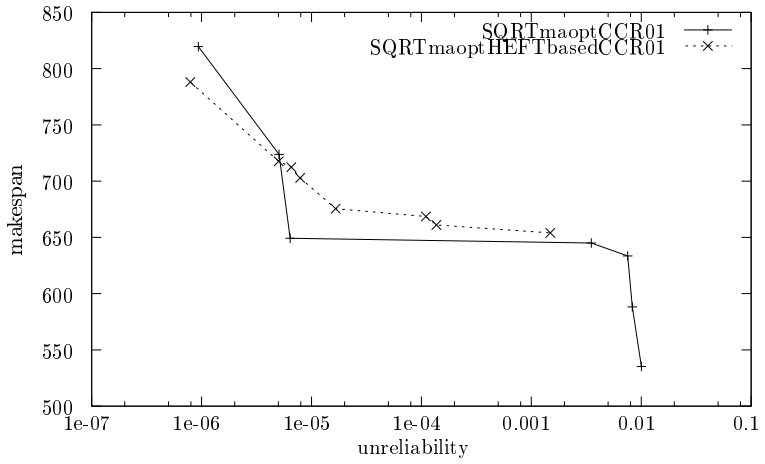


FIG. 5.6 – Impacte de l'initialisation des allocations spatialles avec HEFT

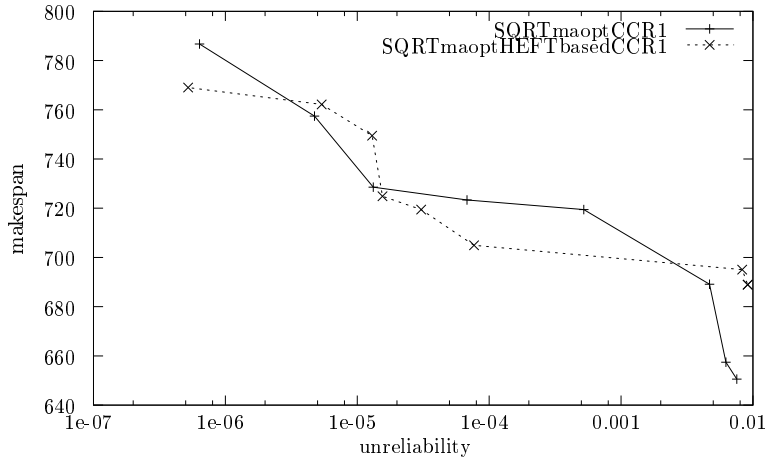
ordre (ou plus) en fiabilité. Sur la Figure 5.6, la courbe `maoptiterlog` est la même qu'en Figure 5.5, tandis que la courbe `maoptHEFTbasediterlog` correspond à notre méthode initialisée avec l'allocation spatiale de HEFT.

Considérons une instance particulière de notre benchmark, la Figure 5.7 montre pour trois valeurs différentes du CCR (pour *Communication to Computation Ratio*), l'ensemble des solutions de compromis calculés par notre méthode (c'est-à-dire notre ensemble approché de Pareto). Les résultats sont fournis avec et sans l'initialisation de la génération aléatoire avec l'allocation de HEFT. Les résultats dépendent clairement du CCR. D'un coté, l'utilisation de l'allocation de HEFT amène à des résultats moins bons lorsque le CCR est faible (Figure 5.7(a)), tandis que de l'autre coté, quand le CCR est élevé, l'ensemble obtenu lorsque l'on initialise notre allocation avec HFET domine complètement les allocations aléatoires (Figure 5.7(b)). Cependant, les Figures 5.7(a) et (b) nous mettent en garde contre l'utilisation systématique de HEFT car nous pourrions ne pas trouver des solutions intéressantes.

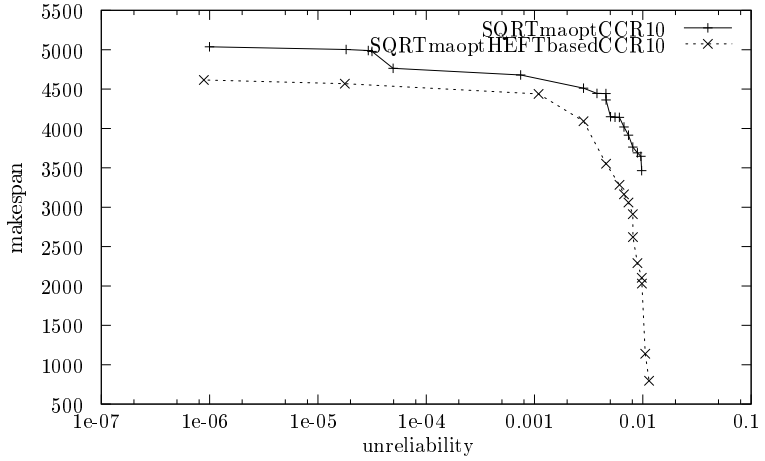
Nous pouvons remarquer que lorsque le CCR est élevé, l'amélioration de la fiabilité a un impact sur le makespan bien plus important que lorsque le CCR est faible. Cela est dû au schéma de réplication pour la fiabilité qui induit beaucoup de communications.



(a) CCR=0.1



(b) CCR=1



(c) CCR=10

FIG. 5.7 – L'instance r150 avec différentes valeurs du CCR

5.6 Conclusion

Nous avons présenté une nouvelle méthode d'ordonnancement de graphe de tâches sur des architectures hétérogènes à mémoire distribuée prenant en compte deux objectifs : le makespan et la fiabilité. Le modèle de fautes suppose que les processeurs sont *fail-silent*, que les liens de communications sont fiables, que les occurrences des fautes suivent une loi de Poisson de paramètre constant. Pour améliorer la fiabilité de plusieurs ordres, nous utilisons la réplication active de tâches. Un point important est d'arriver à calculer efficacement la fiabilité du système. Nous introduisons pour cela la notion de duplication pour la fiabilité. Il est ainsi possible de calculer la fiabilité d'un ordonnancement en temps polynomial, au coût d'une dégradation du makespan.

Nous prouvons qu'il n'existe pas d'algorithme d'approximation à facteur constant de la solution zénith du problème. Nous nous intéressons alors à l'approximation de l'ensemble de Pareto du problème, en utilisant le problème ϵ-contraint sur la fiabilité. Cependant, le problème bi-objectif est très difficile (nous ne connaissons même pas d'algorithme à performance garantie pour le makespan). Ainsi, nous envisageons une méthode heuristique fondée sur des propriétés des solutions optimales.

La méthode que nous proposons s'exécute en deux phases : d'abord, elle détermine l'allocation spatiale des tâches, puis la date à laquelle chaque copie de chaque tâche commence est choisie. Notre modèle de fiabilité et de réplication impliquent que la probabilité d'échec de notre ordonnancement dépend uniquement de l'allocation spatiale. Dans la première phase, nous générons des allocations spatiales aléatoires de fiabilités supérieures à un seuil donné en répliquant des tâches bien choisies. Dans la deuxième phase, une allocation temporelle est générée pour optimiser le makespan, à l'aide d'algorithme pour le problème d'allocation pré-alloué.

La méthode est validée expérimentalement à l'aide de nombreuses simulations. Différentes variantes de la méthode ont été considérées. Il est montré que bien que la méthode puisse être améliorée, elle répond à la demande originale : fournir un ensemble de solutions de compromis entre la fiabilité et le makespan afin qu'un décideur externe puisse choisir la solution qui correspond au besoin de son application.

Chapitre 6

Makespan et tolérance aux pannes permanentes

Dans ce chapitre, nous considérons le problème de l'optimisation simultanée de la performance et de la sûreté de fonctionnement dans les systèmes de calcul parallèles.

6.1 Le calcul sur *cluster*

Le calcul parallèle sur des machines hétérogènes devient de plus en plus populaire. L'augmentation rapide du nombre de machines provoque l'apparition de problèmes d'efficacité de ces plate-formes et des solutions algorithmiques doivent être développées en même temps que des solutions architecturales et interface middleware. Un des problèmes majeurs est l'ordonnancement. À partir d'un graphe d'application (composé de tâches reliées par une relation de précédence), ordonnancer les tâches sur les processeurs pour minimiser le makespan est un problème $\mathcal{NP}$-difficile pour lequel on ne connaît pas d'algorithme d'approximation à facteur constant. Cependant, dans ce cadre, de nombreuse heuristiques non garanties ont été proposées [THW99, CJ01, Bou01] et implémentées.

L'augmentation de la taille des systèmes de calcul parallèles nous oblige à considérer les problèmes de sûreté de fonctionnement. Nous considérons ici des fautes de type *crash-fault* où les machines peuvent tomber en panne et ne fonctionneront plus dans la suite du calcul. C'est par exemple le cas, si le disque dur contenant les données du calcul ou le système d'exploitation ne fonctionne plus. Comme discuté au Chapitre 4, il y a deux façons d'améliorer la sûreté de fonctionnement. Augmenter la fiabilité d'une application en utilisant de la duplication entraine un surcoût non négligeable en temps de

calcul et n'est donc pas satisfaisante.

Les approches à base de *checkpointing* sauvegardent l'application périodiquement et redémarrent l'application en cas de fautes [BHK+05, AFBD06]. Cependant, l'application peut être retardée par le mécanisme de reprise sur faute qui oblige l'utilisateur à redémarrer l'application sur un sous ensemble des processeurs et implique de recommencer des calculs et des communications. Il est donc important de réduire le risque de faute sans même utiliser de duplication. De plus, même si l'on utilise des mécanismes de *checkpointing*, il est important de garantir que la probabilité d'échec de l'application est la plus petite possible. Malheureusement, l'augmentation de la fiabilité implique une augmentation du temps de calcul, bien qu'elle soit moindre que dans les approches avec duplication. Nous recherchons des algorithmes qui minimisent le makespan tout en maximisant la fiabilité.

Dans la littérature, ce problème à déjà été étudié à travers diverses heuristiques [DÖ02, DÖ05, HB06]. Mais aucun de ces travaux ne considère le problème d'optimisation d'un point de vue théorique. Dans [DÖ02], Dogan and Ozgüner proposent un algorithme bi-objectif sans garantie appelé RDLS. Dans [DÖ05], les mêmes auteurs améliorent les performances en utilisant des algorithmes génétiques. Dans [HB06], Hakem and Butelle proposent BSA, un autre algorithme bi-objectif sans garantie qui fournit de meilleurs résultats que RDLS. Cependant, tous ces résultats concernent le cas général où le graphe de tâches n'est pas structuré. Nous manquons d'une analyse fondamentale de ce problème. Parmi les questions non considérées, nous pouvons citer :

- Est ce que maximiser la fiabilité est un problème $\mathcal{NP}$-difficile ?
- Est-il possible de trouver des algorithmes efficaces pour des cas particulier de graphes de précédence ?
- Est-il possible de construire des algorithmes d'approximations, ou encore des $\mathcal{PTAS}$?
- Comment aider l'utilisateur à trouver un bon compromis entre fiabilité et makespan ?

Un problème majeur lorsque l'on traite de la sûreté de fonctionnement est de modéliser les temps d'arrivées des fautes. La distribution exponentielle semble rallier l'avis général. Il peut sembler étrange que cette distribution corresponde à une réalité existante. Cependant, on peut penser qu'une fois les machines démarrées, les temps de calcul sont courts et donc que la probabilité d'apparition d'une faute est constante. Finalement, cette distribution est souvent un choix de référence en calcul stochastique où les calculs sont souvent plus compliqués avec d'autres distributions. Commencer notre étude sur cette distribution de fautes est donc raisonnable.

Il y a trois différences avec le modèle du Chapitre 5. Premièrement, les

processeurs sont reliés par un facteur de puissance (ici, le modèle de processeur est Q (uniforme) alors que le modèle R (unrelated) était utilisé dans le chapitre précédent). Deuxièmement, on ne cherche pas à améliorer la fiabilité en utilisant la duplication. Finalement, nous considérons des fautes permanentes au lieu de fautes transitoires. Ainsi les temps d'inactivité contribuent à la probabilité d'erreur du système.

6.2 Modèle

Le problème d'optimisation est d'ordonnancer un graphe de tâches sur des processeurs hétérogènes uniformes [CB01] (problème $Q \mid prec \mid C_{max}$). En plus des définitions et notations usuelles données au Chapitre 2 (la tâche i est composée de o_i opérations et chacune d'entre elle est exécutée en τ_j unités de temps par le processeur j), nous définissons les quantités suivantes. A chaque arc (i, i') du graphe est associé l_i, le temps de communication du résultat de la tâche i si les deux tâches ne sont pas exécutées sur le même processeur. Nous introduisons aussi λ_j le taux de défaillance (ou *failure rate*) du processeur j. Ainsi si la tâche i est ordonnancée sur le processeur j, la probabilité qu'elle s'exécute correctement est $e^{-o_i \tau_j \lambda_j}$.

La fiabilité d'un ordonnancement est la probabilité que la machine finisse correctement son exécution. Cette quantité est la probabilité que les processeurs soient fonctionnels durant l'exécution de leurs tâches soit $\mathbb{P}_{\text{succ}} = e^{-\sum_{j=1}^{m} C_{max}^{j} \lambda_j}$, où $C_{max}^{j} = \max_{i \mid \pi(i)=j}\{C_i\}$ est la date de terminaison de la dernière tâche exécutée sur le processeur j. Finalement, la probabilité d'échec de l'ordonnancement est $\mathbb{P}_{\text{fail}} = 1 - \mathbb{P}_{\text{succ}}$.

Nous considérons le problème d'optimisation du makespan et de $\mathbb{P}_{\text{succ}}$ simultanément. Remarquons qu'optimiser $\mathbb{P}_{\text{succ}}$ ou $\mathbb{P}_{\text{fail}}$ sont équivalent du point de vue de l'optimalité.

6.3 Analyse

Les deux objectifs que nous cherchons à optimiser sont antagonistes. Plus précisément, nous montrons en Proposition 6.1, que la fiabilité optimale est obtenue en ordonnançant toutes les tâches sur le processeur j qui minimise $\lambda_j \tau_j$; *i.e.*, celui pour lequel le produit *{taux de défaillance}* par *{temps de traitement d'une unité de calcul}* est minimal. Cette quantité est en fait, le taux de défaillance par opération élémentaire. En général, un tel ordonnancement peut être arbitrairement loin de l'ordonnancement optimal pour le makespan.

PROPOSITION 6.1. *Soit S un ordonnancement de toutes les tâches sur un processeur j comme $\lambda_j \tau_j$ est minimal. Soit $\mathbb{P}^*_{succ}$ la fiabilité de l'ordonnancement S. Alors tout ordonnancement S' de fiabilité $\mathbb{P}_{succ}(S')$ est tel que $\mathbb{P}_{succ}(S') \leqslant \mathbb{P}^*_{succ}$.*

Démonstration. Nous supposons sans perte de généralité que $j = 1$ (*i.e.*, $\forall j', \tau_1\lambda_1 \leqslant \tau_{j'}\lambda_{j'}$). Nous avons : $\mathbb{P}^*_{\text{succ}} = e^{-C^1_{max}(S)\lambda_1}$. Ainsi, $\mathbb{P}_{\text{succ}}(S') = e^{-\sum_{j'=1}^{m} C^{j'}_{max}(S')\lambda_{j'}}$. Soit T' l'ensemble de tâches qui ne sont pas exécutées sur le processeur 1 dans S'. On a $C^1_{max}(S') \geqslant C^1_{max}(S) - \tau_1 \sum_{i \in T'} o_i$ (Les tâches de $T \backslash T'$ sont toujours exécutées sur le processeur 1). On considère la partition suivante de ces tâches $T' = T'_2 \cup T'_3 \cup \ldots \cup T'_m$, où $T'_{j'}$ est le sous ensemble des tâches de T' exécutées sur le processeur j' par l'ordonnancement S'. Alors, $\forall 1 \leqslant j' \leqslant m,\ C^{j'}_{max}(S') \geqslant \tau_{j'} \sum_{i \in T'_{j'}} o_i$. Nous comparons les exposants de $\mathbb{P}^*_{\text{succ}}$ et $\mathbb{P}_{\text{succ}}(S')$. Nous avons :

$$
\begin{aligned}
& \sum_{j'=1}^{m} C^{j'}_{max}(S')\lambda_{j'} - C^1_{max}(S)\lambda_1 \\
\geqslant\ & C^1_{max}(S)\lambda_1 - \tau_1\lambda_1 \sum_{i \in T'} o_i + \sum_{j'=2}^{m} \left(\tau_{j'}\lambda_{j'} \sum_{i \in T'_{j'}} o_i \right) - C^1_{max}(S)\lambda_1 \\
=\ & \sum_{j'=2}^{m} \left(\tau_{j'}\lambda_{j'} \sum_{i \in T'_{j'}} o_i \right) - \tau_1\lambda_1 \sum_{i \in T'} o_i \\
=\ & \sum_{j'=2}^{m} \left(\tau_{j'}\lambda_{j'} \sum_{i \in T'_{j'}} o_i \right) - \tau_1\lambda_1 \sum_{j'=2}^{m} \left(\sum_{i \in T'_{j'}} o_i \right) \\
& \qquad \text{(Les ensembles } T'_{j'} \text{ sont disjoints)} \\
=\ & \sum_{j'=2}^{m} \left((\tau_{j'}\lambda_{j'} - \tau_1\lambda_1) \sum_{i \in T'_{j'}} o_i \right) \\
\geqslant\ & 1 \\
& \qquad \text{(car } \forall j' :\ \tau_1\lambda_1 \leqslant \tau_{j'}\lambda_{j'})
\end{aligned}
$$

D'où,

$$\frac{\mathbb{P}_{\text{succ}}}{\mathbb{P}'_{\text{succ}}} = e^{\sum_{j'=1}^{m} C^{j'}_{max}(S')\lambda_{j'} - C^1_{max}(S)\lambda_1} \geqslant 1$$

□

Les deux objectifs ne peuvent pas être optimisés simultanément. Nous nous intéressons alors à deux solutions particulières : la solution Pareto-optimale de fiabilité maximale et la solution zénith.

La Proposition 6.1 montre que la solution Pareto-optimale de fiabilité maximale n'utilise que les processeurs qui minimisent le produit $\tau_j \lambda_j$. Si un seul processeur minimise ce produit, alors le problème est trivial, il suffit d'ordonnancer toutes les tâches séquentiellement sur ce processeur. Cependant, si plusieurs processeurs minimisent ce produit, alors le problème est aussi difficile que $Q \mid prec \mid C_{max}$.

Le Théorème 6.2 montre qu'il n'est pas possible d'obtenir une approximation à facteur constant de la solution zénith.

THÉORÈME 6.2. *Le problème de minimiser C_{max} et de maximiser $\mathbb{P}_{succ}$ n'est pas approximable à facteur constant.*

Démonstration. Nous considérons l'instance du problème avec 2 processeurs tel que $\tau_2 = \tau_1/k$ et $\lambda_2 = k^2\lambda_1$ ($k \in \mathbb{R}^{+*}$) et une seule tâche 1. Seulement deux solutions sont réalisables, S_1 où la tâche est exécutée sur le processeur 1 et S_2 où la tâche est exécutée sur le processeur 2. Remarquons que S_2 est optimale pour C_{max} et que S_1 est optimale pour la fiabilité.

Sur le makespan, nous avons $C_{max}(S_1) = o_1\tau_1$ et $C_{max}(S_2) = o_1\tau_1/k$. Ce qui mène à $C_{max}(S_1)/C_{max}(S_2) = k$. Ce rapport tend vers l'infini avec k. Ainsi, S_1 n'est pas une approximation constante du makespan.

Sur la fiabilité, nous avons $\mathbb{P}_{\text{succ}}(S_1) = e^{-o_1\tau_1\lambda_1}$ et $\mathbb{P}_{\text{succ}}(S_2) = e^{-o_1\tau_1\lambda_1 k}$. Ce qui mène à $\mathbb{P}_{\text{succ}}(S_1)/\mathbb{P}_{\text{succ}}(S_2) = e^{o_1\tau_1\lambda_1(k-1)}$. Ce ratio tend vers l'infini avec k. Ainsi, S_2 n'est pas une approximation constante pour la fiabilité.

Aucune des solutions n'est une approximation constante des deux critères. □

Ce dernier résultat fait que nous nous intéressons aux algorithmes de $\langle \bar{\alpha}, \beta \rangle$-approximation tel que présenté au Chapitre 2. Nous allons prendre comme référence la solution Pareto-optimale qui maximise la fiabilité et de makespan inférieur à G. Cependant, une approximation de la probabilité de réussite n'a pas forcément de sens. Par exemple, pour $\beta = 5$, une probabilité de réussite de 1 est approchée par 0.2 et une probabilité d'échec de 0.3 est approchée par 1.5... Dans ce sens, un rapport d'approximation serait peu intéressant. Une façon de renforcer cette borne est donnée par la Proposition 6.3.

PROPOSITION 6.3. *Soit S une solution du problème et $\beta > 1$ un réel. Nous avons, $\mathbb{P}_{succ}(S) = {\mathbb{P}^*_{succ}}^{\beta} \Rightarrow \mathbb{P}_{fail}(S) \leqslant \beta \mathbb{P}^*_{fail}$.*

Démonstration. La preuve est basée sur l'inégalité de Bernoulli où, $\forall x \in [0,1], \forall n \in [1,+\infty[, (1-x)^n \geqslant 1-nx$.

$$\begin{aligned}\mathbb{P}_{\text{fail}} &= 1-\mathbb{P}_{\text{succ}} = 1-{\mathbb{P}^*_{\text{succ}}}^{\beta} = 1-(1-\mathbb{P}^*_{\text{fail}})^{\beta} \\ &\leqslant 1-(1-\beta\cdot\mathbb{P}^*_{\text{fail}}) = \beta\cdot\mathbb{P}^*_{\text{fail}}\end{aligned}$$

□

Dans le reste du chapitre, nous chercherons des solutions où la fiabilité est bornée par une puissance de la fiabilité optimale. Cela a d'ailleurs du sens. En pratique, ce qui intéresse la communauté de sûreté de fonctionnement est l'ordre de grandeur de la fiabilité et pas la valeur exacte.

L'objectif que nous considérerons sera donc la minimisation de $rel(S) = \sum_j C^j_{max}(S)\lambda_j$. L'approximation au sens classique de rel est équivalente à borner une puissance de la fiabilité optimale.

6.4 Tâches indépendantes

Les principes de la construction d'un algorithme de $\langle\bar{\alpha},\beta\rangle$-approximation sont d'abord traités sur le cas simple où toutes les tâches sont indépendantes et ont le même temps de calcul. Ils sont ensuite étendus au cas où les tâches ont des temps de calcul arbitraires. Cette analyse servira de base au cas avec contrainte de précédences.

6.4.1 Tâches UET

Remarquons d'abord qu'il est facile de savoir s'il existe des ordonnancement de makespan inférieur à G. En effet, il est facile de trouver l'allocation de plus petit makespan pour le problème de tâches UET indépendantes comme montré dans [LR05], p. 161. Dans cette variante du problème les tâches ne sont pas distinguées, il suffit donc de donner le nombre de tâches n_j sur le processeur j. Ce nombre se calcul en utilisant l'algorithme classique ETF.

On s'intéresse au problème de minimiser rel sous contrainte que le makespan soit inférieur à G. Nous proposons l'Algorithme 6.1 pour résoudre ce problème. Il fournit une solution optimale au problème ; ceci est énoncé par le Théorème 6.4. C'est un algorithme glouton qui alloue le plus de tâches possible sur le processeur de meilleur produit $\lambda_i\tau_i$ sans dépasser un makespan de G.

THÉORÈME 6.4. *L'Algorithme 6.1 est une $\langle\bar{1},1\rangle$-approximation du problème de tâches UET indépendantes.*

Algorithme 6.1 Ordonnancement optimal pour les tâches indépendantes UET

Input : Un makespan $G \geqslant C^*_{max}$
Begin
Trier les processeurs par valeurs croissantes de $\tau_j \lambda_j$
$alloue \leftarrow 0$
for j=1 :m
  if $alloue < n$
    $n_j \leftarrow \min\left(n - alloue, \left\lfloor \frac{G}{\tau_j} \right\rfloor\right)$
  else
    $n_j \leftarrow 0$
  $alloue \leftarrow X + n_j$
End

Démonstration. Il est facile de vérifier que G est réalisable à l'aide de l'algorithme optimal pour le makespan. *alloue* est le nombre de tâches qui ont déjà été ordonnancées. Tous les processeurs sont considérés et autant de tâches que possible sont allouées à chaque processeur. À la fin de l'algorithme toutes les tâches ont été allouées en moins de G unité de temps.

Il nous reste à montrer que *rel* est le plus petit atteignable en G unités de temps. Remarquons d'abord que l'algorithme remplit les processeurs en ordre croissant du produit $\tau_j \lambda_j$. Supposons la solution ne soit pas optimale. Alors l'allocation optimale $S' = \{n'_1, \ldots, n'_m\}$ est telle que $n'_j < n_j$ et $n'_{j'} > n_{j'}$ avec $j < j'$. Le déplacement d'une tâche de j' vers j dans S' provoque une amélioration de la fiabilité de S'. Ce qui contredit son optimalité. □

Cet algorithme de $\langle \bar{1}, 1 \rangle$-approximation permet de construire une $(1 + \epsilon, 1)$-approximation de l'ensemble de Pareto à l'aide de la technique décrite au Chapitre 2. Il suffit de vérifier que l'on ait des valeurs de makespan encadrant l'ensemble de Pareto dont la taille est polynomiale dans la taille du problème. Une borne inférieure est $\frac{n}{\sum_j \frac{1}{\tau_j}}$. Une borne supérieure est $n \max_j \tau_j$.

6.4.2 Vers des temps de calcul arbitraires

Nous étendons maintenant le problème à des temps de calcul arbitraires. Nous commençons par étudier une propriété d'optimalité du problème. La Proposition 6.1 montre que l'ordonnancement le plus fiable est obtenu en allouant toutes les tâches sur le processeur qui minimise $\tau_j \lambda j$. Nous pouvons dériver un résultat pour les ordonnancements fractionnaires de makespan inférieur à G.

THÉORÈME 6.5. *L'ordonnancement fractionnaire S^* pour lequel si le processeur j_0 exécute une tâche alors tous les processeurs j tel que $\tau_j\lambda_j < \tau_{j_0}\lambda_{j_0}$ exécutent $\frac{G}{\tau_{j_0}}$ opérations est optimal.*

Démonstration. Sans perte de généralité, on peut supposer que les processeurs sont triés en ordre non décroissant de $\tau\lambda$. Supposons que l'ordonnancement S^* existe. Soit S une solution optimale de meilleure fiabilité que S^*. S est tel que $\exists j' \leqslant j_0, C^{j'}_{max}(S) < C^{j'}_{max}(S^*)$ et $\exists j'' > j', C^{j''}_{max}(S) > C^{j''}_{max}(S^*)$. Le déplacement d'une opération de j'' à j' améliore la fiabilité de S. Ainsi S n'est pas optimale. □

Ce théorème indique que, parmi les ordonnancements dont le makespan ne dépasse pas G, l'ordonnancement le plus fiable est obtenu en ordonnançant les tâches sur un sous ensemble des processeurs ; Ce sous ensemble est tel que les valeurs de $\lambda\tau$ sont les plus petites possibles. Remarquons que la solution S^* est fractionnaire et non entière. Cependant, ce théorème montre que l'on peut aider à obtenir une solution plus fiable en n'utilisant que les processeurs de plus faible produit $\lambda_j\tau_j$.

On peut ainsi laisser à l'utilisateur le choix du nombre de processeurs à utiliser pour exécuter son calcul. Seulement les k processeurs de meilleur produit $\tau_j\lambda_j$ seront utilisés. L'idée est que lorsque l'on retire des processeurs, le makespan va être dégradé, mais grâce au Théorème 6.5, cela augmentera la fiabilité. Par exemple, si $k = 1$, alors toutes les tâches seront allouées au processeur le plus fiable et l'optimal en fiabilité sera atteint. Symétriquement, avec $k = m$, un algorithme d'optimisation du makespan fournit une mauvaise fiabilité mais un bon makespan. Remarquons également qu'en présence de précédences, le retrait de certains processeurs devrait limiter les temps d'inactivités qui contribuent à la probabilité d'échec.

6.4.3 Une $\langle\bar{2}, 1\rangle$-approximation

La recherche de $\langle\bar{\alpha}, \beta\rangle$-approximation est plus difficile car le problème d'optimisation est $\mathcal{NP}$-difficile. Dans [ST93], Shmoys et Tardos étudient le problème d'optimiser le makespan et la somme des coûts d'un ordonnancement sur des machines hétérogènes (modèle R). Dans leur problème, le coût est induit par l'exécution d'une tâche sur un processeur et ce coût est arbitraire, il n'est pas relié au temps de calcul de cette tâche sur ce processeur. Ils proposent un algorithme qui prend deux paramètres, un makespan G et un coût C et qui retourne un ordonnancement de makespan inférieur à $2G$ et de coût inférieur à C. Ce problème peut être utilisé pour résoudre notre problème. Cependant, leur algorithme est difficile a implémenter, il repose sur la programmation linéaire et sa complexité est élevée : $O(mn^2 \log n)$.

Nous présentons maintenant un algorithme de $\langle \bar{2}, 1 \rangle$-approximation appelé CMLT (pour *Constrained Min Lambda Tau*) qui a une meilleur complexité et qui est bien plus facile à implémenter que celui présenté dans [ST93]. Cet algorithme utilise une partie des idées de la 2-approximation du problème $R \mid\mid C_{max}$ présentée au Chapitre 2.

Soit $M(i) = \{j \mid p_{ij} \leqslant G\}$, l'ensemble de processeurs qui peuvent exécuter la tâche i en moins de G unité de temps. Il est évident que dans une solution de makespan inférieur à G, la tâche i n'est pas allouée à un processeur $j \notin M(i)$. La proposition suivante énonce que si la tâche i comporte moins d'opérations que la tâche i' alors toutes les machines qui peuvent exécuter i' en moins de G unités de temps peuvent exécuter i dans le même temps. La preuve suit directement la définition de $M(i)$ et est omise.

PROPOSITION 6.6. *$\forall i, i' \in T$ tels que $o_i \leqslant o_{i'}$, $M(i') \subseteq M(i)$*

L'algorithme CMLT (pour *ConstrainedMinLambdaTau* ordonnance les tâches de la façon suivante. Pour chaque tâche i considérée en ordre LPT, ordonnancer i sur le processeur $j \in M(i)$ qui minimise $\lambda_j \tau_j$ et qui respecte $C^j_{max} \leqslant G$ (ainsi, i terminera avant $2G$). Si un tel processeur n'existe pas, l'algorithme retourne qu'il n'y a pas d'ordonnancement de makespan inférieur à G. Trier les tâches en ordre décroissant de nombre d'opérations impliquent que de plus en plus de processeurs sont considérés.

Bien que le principe de l'algorithme soit simple, plusieurs propriétés doivent être vérifiées pour assurer qu'il est toujours possible d'ordonnancer les tâches de cette façon. Rappelons que la valeur optimale en fiabilité dans les ordonnancements de makespan inférieur à G est noté $rel^{*,G-}$.

LEMME 6.7. *CMLT retourne un ordonnancement dont le makespan est inférieur à $2G$ ou alors assure qu'il n'existe pas d'ordonnancement de makespan inférieur à G.*

Démonstration. Remarquons d'abord que si l'algorithme renvoie un ordonnancement alors son makespan est inférieur à $2G$: la tâche i est ordonnancée sur le processeur j qui appartient à $M(i)$ et avant que la tâche ne soit allouée, nous avons $C^j_{max} \leqslant G$. Il reste a montrer que si l'algorithme ne renvoie pas de solution, alors il n'existe pas d'ordonnancement de makespan inférieur à G.

Sans perte de généralité, nous pouvons supposer que les tâches sont triées dans un ordre décroissant de leur nombre d'opérations.

Pour que la tâche i ne puisse être ordonnancée sur aucun processeur de $M(i)$, il faut que tous les processeurs de $M(i)$ exécutent des tâches pendant plus de G unité de temps : $\forall j \in M(i), C^j_{max} > G$.

De plus, grâce à la Proposition 6.6, chaque tâche $i' \leqslant i$ n'aurait pas pu être ordonnancée sur un processeur n'appartenant pas à $M(i)$ (quelques processeurs de $M(i)$ pourraient ne pas appartenir à $M(i')$, mais l'inverse n'est pas possible). Ainsi dans un ordonnancement de makespan inférieur à G, toutes les tâches $i' \leqslant i$ doivent être ordonnancées sur $M(i)$.

Il y a plus d'opérations dans l'ensemble de tâches $\{i' \leqslant i\}$ que les processeurs $M(i)$ ne peuvent exécuter en G unités de temps et aucun autre processeur ne peut les exécuter. Ainsi, il n'y a donc aucun ordonnancement de makespan inférieur à G. □

LEMME 6.8. *Les ordonnancements retournés par CMLT sont tel que* $rel \leqslant rel^{*,G-}$

Démonstration. Nous construisons d'abord un ordonnancement non réalisable S^* dont la fiabilité est une borne inférieure de $rel^{*,G-}$. Ensuite, nous montrons que $rel(CMLT) \leqslant rel(S^*)$.

La solution que nous construisons est similaire à la solution évoquée dans le Théorème 6.5. Les tâches sont itérativement ordonnancées par ordre décroissant de nombre d'opérations. La tâche i est allouée au processeur de $M(i)$ qui minimise $\lambda_j \tau_j$. Si i termine après G, les opérations qui débordent de G sont ordonnancées sur le prochain processeur appartenant à $M(i)$ dans l'ordre des $\tau_j \lambda_j$ croissant. Des arguments similaires à ceux du Théorème 6.5 montre que $rel(S^*) \leqslant rel^{*,G-}$.

L'ordonnancement généré par CMLT est similaire à S^*. La seule différence est que des opérations sont ordonnancées après G. Dans S^*, ces opérations ont été ordonnancées sur un processeur moins fiable. Ainsi, l'ordonnancement de CMLT a une meilleure fiabilité que S^*.

Finalement, nous avons $rel(CMLT) \leqslant rel(S^*) \leqslant rel^{*,G-}$ qui termine la preuve. □

LEMME 6.9. *La complexité en temps de CMLT est en* $O(n \log n + m \log m)$.

Démonstration. En fait, l'algorithme devrait être implémenté à l'aide d'un tas de la façon présentée en Algorithme 6.2. Le coût des opérations de tri des tâches appartient à $O(n \log n)$ et le coût des opérations de tri des processeurs appartient à $O(m \log m)$. Ajouter (et retirer) un processeur au (du) tas coûte $O(\log m)$. Ces opérations sont faites m fois. Tous les accès au tas coûtent $O(m \log m)$. Ordonnancer chaque tâche se fait en temps constant et il y a n tâches à ordonnancer. Les coût d'ordonnancement sont donc de $O(n)$. □

THÉORÈME 6.10. *CMLT est un algorithme de* $\langle \bar{2}, 1 \rangle$*-approximation de complexité appartenant à* $O(n \log n + m \log m)$.

Algorithme 6.2 CMLT

Begin
Trier les tâches en ordre décroissant de o_i
Trier les processeurs en ordre croissant de τ_j
Soit H un tas vide
$j = 1$
For $i = 1$ **to** n
  While $j \in M(i)$
    Ajouter j au tas H avec la clé $\lambda_j \tau_j$
    $j = j + 1$
  End While
  If H.empty()
    Return *no solution*
  Ordonnancer i sur $j' = H$.min()
  $C_{max}^{j'} = C_{max}^{j'} + o_i \tau_{j'}$
  If $C_{max} j' > G$
    Retirer j' de H
End For
End

Pour que ce théorème mène à une approximation de l'ensemble de Pareto du problème, il faut que le nombre de points générés soit polynomial dans la taille de l'instance. Il faut pour cela borner les makespans des solutions Pareto-optimales. Ils sont bornés inférieurement par $C_{max}^{min} = \frac{\sum_i p_i}{\sum_j \frac{1}{\tau_j}}$, le makespan obtenu en considérant que l'on a qu'une seule machine qui dispose de la puissance de calcul agrégé de toutes les autres. La borne supérieure est $C_{max}^{max} = \sum_i p_i \max_j \tau_j$, le makespan obtenu si toutes les tâches sont ordonnancées sur le processeur le plus lent. Remarquons que C_{max}^{max} peut être atteint si le processeur le plus lent est aussi le plus fiable. Le rapport entre les deux quantités étant de taille polynomiale, l'algorithme CMLT amène bien à une $(2+\epsilon, 1)$-approximation de l'ensemble de Pareto.

De plus, dans la perspective de la construction d'un ensemble approché de Pareto, nous pouvons remarquer qu'une partie des calculs de CMLT peuvent être évités. En effet, il n'est utile de trier les tâches et les processeurs qu'une seule fois. Ainsi la complexité de la construction d'un ensemble $(2+\epsilon, 1)$-approché avec CMLT appartient à $O(n \log n + \left\lceil \log_{1+\epsilon/2}(\frac{C_{max}^{max}}{C_{max}^{min}}) \right\rceil (n + m \log m))$.

Nous avons brièvement rappelé certains travaux de Shmoys et Tardos

qui pourraient être utilisés dans notre contexte [ST93]. Nous pouvons en dériver un algorithme de $\langle \bar{2}, 1 \rangle$-approximation de complexité appartenant à $O(mn^2 \log n)$. Cette complexité est bien plus important que cette de CMLT qui appartient à $O(n \log n + m \log m)$. Pour la construction d'un ensemble de Pareto approché, l'algorithme dérivé des travaux de [ST93] aurait une complexité de $O(\left\lceil \log_{1+\epsilon/2}(\frac{C_{max}^{max}}{C_{max}^{min}}) \right\rceil (mn^2 \log n))$. Il n'est pas facilement améliorable pour conserver une partie des calculs de l'algorithme d'un appel à l'autre. Ainsi, CMLT est significativement meilleur que l'algorithme présenté dans [ST93]. Rappelons que ce dernier à été présenté dans le cadre plus général de processeur hétérogène (modèle R).

6.5 Intégration à des heuristiques existantes

Dans cette section, nous étudions le cas général d'un graphe de tâches quelconque. Le produit $\tau_j \lambda_j$ est une quantité importante du problème. Nous proposons une façon simple d'étendre les heuristiques non garantie d'optimisation du makespan en algorithme bi-objectif en intégrant ce produit dans la décision gloutonne des algorithmes.

6.5.1 Généralisation d'algorithme d'ordonnancement : le cas de HEFT

Dans [THW99], Topcuoglu *et al.* ont proposé l'heuristique HEFT (pour *Heterogeneous Earliest Finish Time*) qui ordonnance un graphe de tâches sur un ensemble de processeurs hétérogène. HEFT fonctionne de la manière suivante. À chaque étape, il considère toutes les tâches prêtes et simule leur allocation à tous les processeurs. HEFT choisit la tâche qui termine le plus tôt et l'alloue sur le processeur qui la termine le plus tôt.

Nous proposons de transformer HEFT en RHEFT (pour *Reliable HEFT*) en intégrant le produit $\tau_j \lambda_j$ dans le processus de décision. Nous notons ${t_{\text{end}}}_j^i$ la date de terminaison de la tâche i si elle était allouée au processeur j (${t_{\text{end}}}_j^i$ peut être calculé en ajoutant $o_i \tau_j$ au maximum entre la date de mise à disposition de i sur j et la date de disponibilité de j). Pour toutes les tâches prêtes i et tous les processeurs j, au lieu de choisir la tâche qui minimise ${t_{\text{end}}}_j^i$, nous choisissons la tâche qui minimise ${t_{\text{end}}}_j^i \lambda_j$ et l'allouons à j.

Pour illustrer la différence entre HEFT et RHEFT, considérons l'exemple de la Figure 6.1. Dans cet exemple, nous avons deux processeurs avec une vitesse de calcul τ_j respectivement de 1 et de 2 et un taux de panne λ_j respectivement de 2 et 1. Supposons que la tâche i constituée de 2 opérations

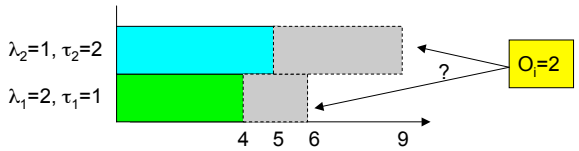


FIG. 6.1 – Illustration de la différence entre HEFT (qui alloue i au processeur 1) et RHEFT (qui alloue i au processeur 2).

puisse être ordonnancée et que le processeur 1 est disponible au temps 4 alors que le processeur 2 est disponible au temps 5. Alors nous avons :

- ${t_{\text{end}}}_1^i = 6$, ${t_{\text{end}}}_1^i \times \lambda_1 = 12$
- ${t_{\text{end}}}_2^i = 9$, ${t_{\text{end}}}_2^i \times \lambda_2 = 9$

HEFT allouerait la tâche i au processeur 1 alors que RHEFT l'allouerait au processeur 2.

Il est important de noter que si une tâche i' de 2 opérations élémentaires est soumise après que i ait été allouée au processeur 2, alors RHEFT allouera i' au processeur 1 car : ${t_{\text{end}}}_1^{i'}\lambda_1 = 6 \times 2 = 12$ et ${t_{\text{end}}}_2^{i'}\lambda_2 = 13 \times 1 = 13$.

6.5.2 Vers plus de compromis

RHEFT favorise les processeurs qui ont une grande fiabilité et une vitesse raisonnable. Comme dans les approches d'approximation de l'ensemble de Pareto, l'utilisateur voudra certainement choisir le compromis entre temps d'exécution et fiabilité qui correspond à son application. Nous pouvons utiliser l'approche présentée en Section 6.4.2 en allouant les tâches qu'à une partie des processeurs.

Nous avons généré aléatoirement un ensemble de 100 machines avec des taux de panne λ_j choisis uniformément dans $[10^{-2}, 10^{-3}]$ et des vitesses τ_j choisies uniformément dans $[10^{-5}, 10^{-7}]$. Ces valeurs ne sont pas très réalistes mais elles fournissent des résultats faciles à lire. Les expériences menées avec d'autres valeurs fournissent des résultats similaires et sont omises. Le graphe de tâches utilisé est la multiplication de Strassen avec 4800 tâches (correspondant à une multiplication de matrice de 1000 blocs par 1000 blocs). Nous avons expérimentalement testé plusieurs façon de trier les processeurs pour ne sélectionner que k processeurs sur les m disponibles. Les trois ordres suivant ont été considérés : λ_j croissant en Figure 6.2, τ_j croissant en Figure 6.3 et $\tau_j\lambda_j$ croissant en Figure 6.4. Dans chaque figure, pour chaque nombre de processeurs utilisé entre 1 et 100 (sur l'axe des abscisses), nous donnons le makespan et la fiabilité.

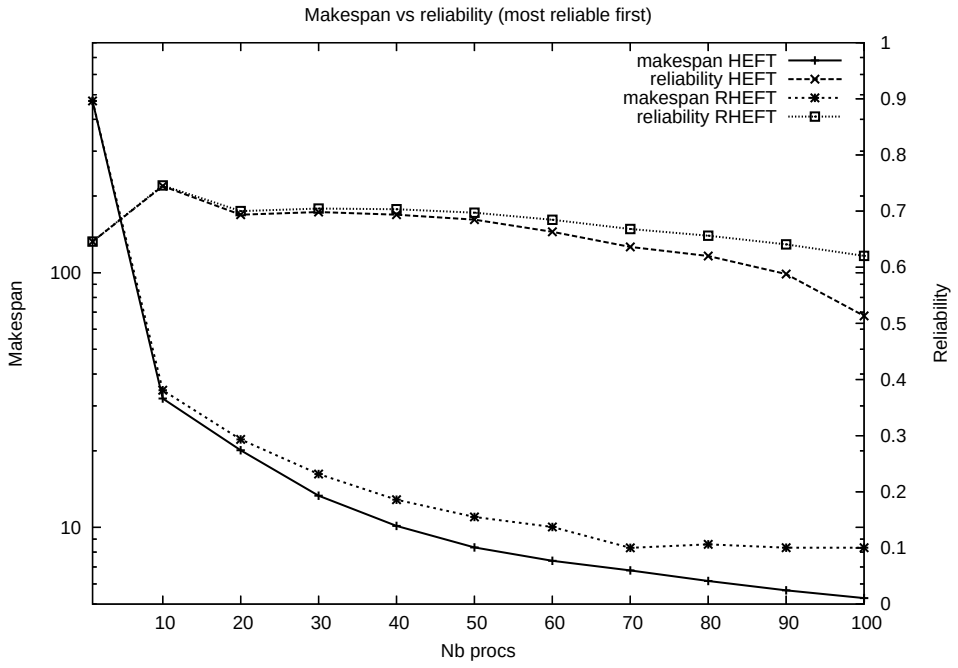


FIG. 6.2 – Limitation du nombre de processeurs en prenant les plus petits indices de fiabilité d'abord.

Conformément à l'intuition, prendre les processeurs de bon indice de fiabilité ne fournit pas forcément des ordonnancements fiables. En effet, λ_j représente le taux de défaillance par unité de temps et pas le taux de défaillance par unité de calcul effectué qui lui est donné par le produit $\lambda_j \tau_j$.

La comparaison entre le tri par vitesse ou par taux de défaillance par unité de temps n'est par contre pas si évidente. Globalement, le tri par $\lambda_j \tau_j$ fournit une meilleur fiabilité mais un makespan légèrement moins bon que le tri par vitesse. En particulier sur moins de 20 processeurs, l'opération améliore grandement la fiabilité sans vraiment dégrader le makespan. Ainsi, restreindre l'ensemble des processeurs peut permettre d'optimiser la fiabilité en fixant un seuil à remplir sur le makespan.

Si un utilisateur veut obtenir un résultat intermédiaire entre HEFT et RHEFT, nous pouvons modifier RHEFT pour intégrer une variable de compromis α de façon à ce que quand $\alpha = 1$, nous utilisons HEFT, que quand

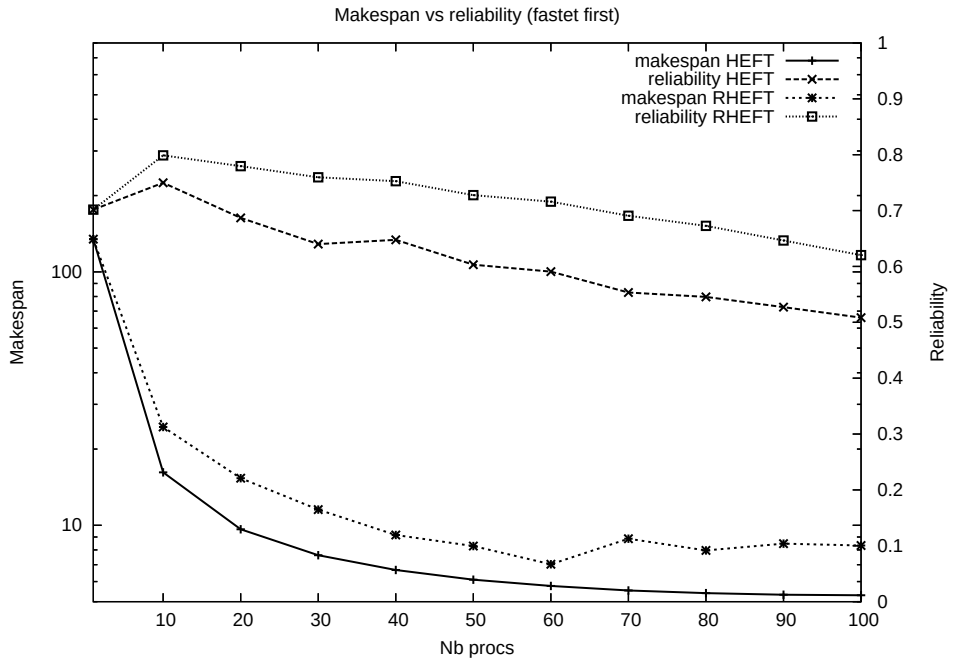


FIG. 6.3 – Limitation du nombre de processeurs en prenant les plus rapides d'abord.

$\alpha = 0$ nous utilisons RHEFT, et que pour toute valeur de α entre 0 et 1, l'heuristique construise une solution intermédiaire en prenant la combinaison linéaire des deux critères de décision. Dans la Figure 6.5, nous montrons les variations du makespan et de la fiabilité quand α varie entre 0 et 1. Nous avons utilisé $\alpha' = e^{1-1/\alpha}$ pour $\alpha \neq 0$ de façon à visualiser l'influence de petites valeurs de la variable de compromis. Les deux plateaux sont dus aux valeurs extrême du paramètre. Dans un cas, le paramètre a une valeur trop grande et influence très peu HEFT tandis que lorsque sa valeur est trop faible, le terme de RHEFT devient prédominant.

Nous pouvons ainsi vérifier que ce paramètre permet de choisir des comportements intermédiaires en HEFT et RHEFT.

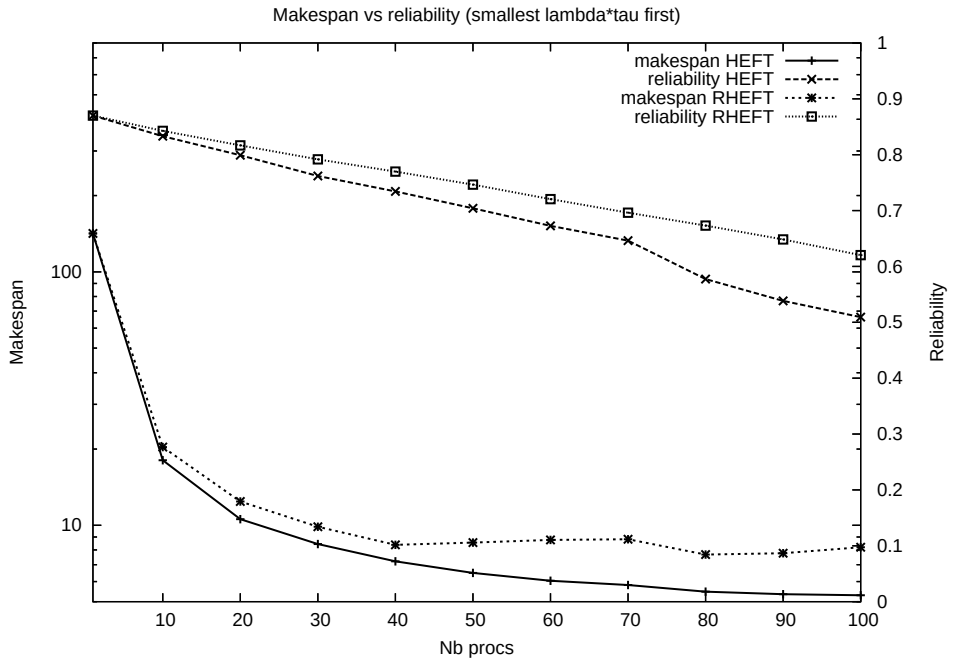


FIG. 6.4 – Limitation du nombre de processeurs en prenant les plus petits produit $\lambda_j \tau_j$ d'abord.

6.5.3 Extension à d'autres heuristiques

Il est facile d'étendre d'autres heuristiques d'ordonnancement pour les systèmes hétérogènes pour prendre en compte la fiabilité. Cela semble direct pour des heuristiques comme BIL [OH96], PCT [MS98], GDL [SL93] ou CPOP [THW99]. Par exemple, dans CPOP il faudrait modifier la notion de chemin critique pour prendre en compte la fiabilité des processeurs sur lesquels les tâches sont allouées.

L'utilisation de moins de processeurs tout comme l'utilisation de variable de compromis peuvent s'appliquer.

Il faut néanmoins rester vigilant lorsque l'on ajoute un terme dans l'expression qui sert a choisir l'allocation dans l'algorithme. Tout d'abords il est important de noter qu'il ne permet pas d'optimiser une combinaison linéaire des deux objectifs. De plus, le produit ${t_{\text{end}}}_j^i$ représente la contribution de toutes les tâches allouées sur le processeur j à la probabilité de panne et

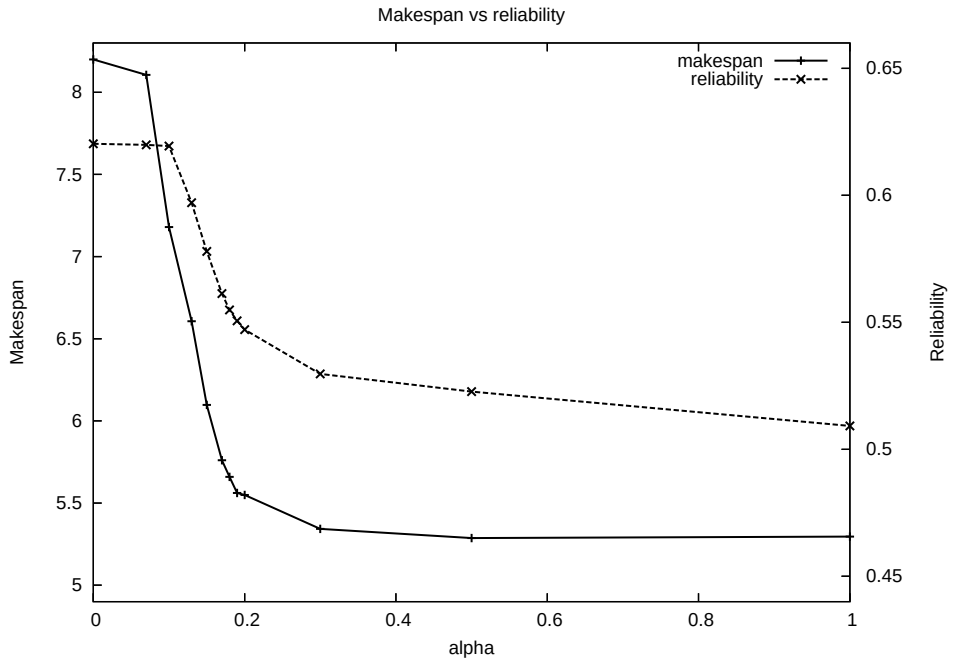


FIG. 6.5 – Impact de la variable de compromis sur 100 processeurs (le cas $\alpha = 0$ est RHEFT et le cas $\alpha = 1$ est HEFT).

pas uniquement l'allocation de la tâche i sur ce processeur. L'algorithme ne distingue pas l'allocation d'une tâche à un processeur très utilisé (qui modifie légèrement la probabilité de d'échec) de l'allocation d'une tache à un processeur non utilisé qui rajoute tout le temps libre du processeur à la probabilité de panne. Ainsi l'algorithme a tendance à utiliser les processeurs peu fiable après les processeurs fiable, au lieu de faire le contraire. Cette approche aurait plus de sens dans des modèles de pannes transitoires.

6.6 Conclusion

Nous avons considéré le problème de l'optimisation de la fiabilité sans réplication pour les *clusters* et grilles dans un modèle de crash fault. Une analyse du problème à partir du sous problème de tâches indépendantes a été proposée. Un algorithme optimal pour des tâches UET à été fourni et un

algorithme de $\langle \bar{2}, 1 \rangle$-approximation pour le cas général de tâches indépendantes a été proposé. Cette analyse a permis de généraliser des algorithmes classiques non garanties d'optimisation du makespan pour optimiser également la fiabilité.

Chapitre 7

Ordonnancement multi-utilisateurs

7.1 Introduction

7.1.1 Description du problème

Les systèmes de calculs hautes performances tels que les *clusters* ou les grilles de calcul ne sont généralement pas dédiés à un unique utilisateur. Ils sont généralement partagés entre de nombreux utilisateurs qui exécutent de nombreuses tâches de calcul. Ces utilisateurs sont en concurrence pour l'exécution de leurs tâches sur la ressource de calcul commune. Dans ces systèmes, le problème d'allocation de ressources est traité par un ordonnanceur dont le principe est plutôt simple : les tâches sont soumises par les utilisateurs dans une file d'attente puis sont traitées par lots (ou *batches*). A notre connaissance, l'ordonnanceur le plus élaboré est MAUI [JSC01]. Il utilise des paramètres de file pour prendre en compte les différentes caractéristiques des tâches. MAUI peut être utilisé de telle sorte que chaque utilisateur soit associé à une file qui lui est dédiée. Régler finement les paramètres de la file peut permettre à l'utilisateur d'exprimer ses besoins. Cependant, les propriétés théoriques de cet ordonnanceur n'ont pas été étudiées du point de vue des utilisateurs. Remarquons que la plupart des ordonnanceurs existant sont centralisés, ce qui signifie qu'un unique programme est responsable de l'ordonnancement de toutes les tâches.

Dans ce chapitre, nous étudions le problème d'ordonnancer des tâches appartenant à différents utilisateurs pour optimiser simultanément les besoins de chacun d'entre eux. Les utilisateurs n'ont aucune décision à prendre, ils se contentent de soumettre leurs tâches à l'ordonnanceur et de choisir une fonction objectif parmi quelques objectifs classiques. Notre but est de propo-

ser et d'analyser des politiques de partage de ressources prenant en compte les objectifs des utilisateurs.

7.1.2 Différentes approches

Plusieurs tentatives ont été proposées pour optimiser simultanément plusieurs fonctions objectif. Nous distinguons principalement deux approches, la théorie des jeux et l'optimisation multi-objectif.

La théorie des jeux est un cadre efficace pour étudier les systèmes dans lesquels de nombreux agents (utilisateurs dans notre contexte) interagissent [OR94]. Dans les systèmes de calcul, nous distinguons deux types de problèmes. D'un coté, un arbitre centralisé construit une solution qui dépend des décisions des utilisateurs. Dans ces cas, le problème est souvent considéré d'un point de vue économique où des prix sont associés à des tranches de temps. Un tel problème peut être résolu avec des techniques d'enchère [WWWMM01] ou des protocoles de découpe [Str80]. D'un autre coté, les utilisateurs construisent une solution de façon décentralisée. Par exemple, Pascual *et al.* traite le problème d'équilibrage de charge de plusieurs organisations virtuelles et montrent que la coopération entre les organisations peut toujours être utilisée pour améliorer les performances globales [PRT07]. Legrand et Touati étudient un système maître-esclave simple où des utilisateurs soumettent leurs tâches divisible de façon non coopérative [LT07].

En optimisation multi-objectif classique, la plupart des résultats concernent un nombre restreint d'objectifs (deux objectifs dans la plupart des cas) et reposent sur des algorithmes sophistiqués qui sont difficile à adapter dans un cadre plus large. Dans le cadre de l'ordonnancement multi-utilisateurs, deux travaux peuvent servir de point de départ [BS03, AMPP04]. Les deux traitent du problème d'ordonnancer les tâches de deux utilisateurs sur une seule machine. Nous analysons en détails le contenu de ces deux travaux dans la Section 7.2.

Les problèmes multi-utilisateurs peuvent également être considérés par l'optimisation de fonctions objectif peu classiques. Les objectifs des utilisateurs sont agrégés par une fonction d'équité. Un nombre infini de fonctions d'équité (elles sont appelées α-*fairness*) peuvent être définies et toutes ont leurs avantages. Ainsi, aucune ne peut être facilement écartée [MW98]. Cependant, les techniques d'agrégation n'ont de sens que lorsque les objectifs des utilisateurs sont similaires.

7.1.3 Contributions

Dans ce chapitre nous considérons le problème d'ordonnancer des tâches indépendantes séquentielles appartenant à k utilisateurs différents sur m processeurs dans lequel chaque utilisateur choisit une fonction objectif parmi une liste de fonctions objectif classiques : makespan, somme des temps de terminaison (pondérés ou non) et *maximum flow time*. Notre contribution est de fournir des algorithmes qui ordonnancent les tâches d'utilisateurs ayant des objectifs différents avec la meilleur garantie théorique atteignable. Plus précisément :

Premièrement, nous montrons que si tous les utilisateurs veulent optimiser le makespan alors la solution zénith du problème n'est pas approximable avec un facteur meilleur que $(1, 2, \ldots, k)$. Si tous les utilisateurs veulent optimiser la somme des temps de terminaison alors le zénith du problème n'est pas approximable avec un facteur meilleur que $(k, \ldots, k)$. Si un utilisateur veut optimiser le *maximum flow time* alors le zénith du problème n'est pas approximable à facteur constant.

Ensuite, nous proposons un algorithme (appelé MULTICMAX) pour le cas où tous les utilisateurs veulent optimiser le makespan et nous prouvons que c'est une $(\rho, 2\rho, \ldots, k\rho)$-approximation du zénith où ρ est le rapport d'approximation pour le cas à un seul utilisateur. Pour le cas où tous les utilisateurs sont intéressés par la somme des temps de terminaison, nous proposons un algorithme (appelé MULTISUM) qui fournit une $(k, \ldots, k)$-approximation du zénith. Les deux algorithmes sont optimaux dans le sens où il n'existe pas d'algorithme qui obtienne un meilleur rapport d'approximation. Notons que l'algorithme pour la somme des temps de terminaison peut être étendu pour la somme des temps de terminaison pondérés.

Ces rapports peuvent sembler décevant car ils dépendent du nombre d'utilisateurs. Cependant, ils sont élevés car ils prennent en référence la solution zénith qui n'est pas valide. Si une solution est une mauvaise approximation du zénith, cela n'implique pas qu'elle ne soit pas Pareto-optimale. Nous montrons qu'une légère adaptation de MULTICMAX est une approximation constante d'une solution Pareto-optimale.

Finalement, nous traitons le cas mixte où k' utilisateurs veulent optimiser la somme des temps de terminaison et k'' utilisateurs veulent optimiser le makespan. Nous proposons un algorithme (appelé MULTIMIXED) qui produit une solution qui approche le zénith à un facteur $(k, \ldots, k)$ pour les sommes des temps de terminaison et à un facteur $(\frac{k}{k''}\rho, \frac{2k}{k''}\rho, \ldots, k\rho)$ pour les makespans.

7.2 Modèle et première analyse

7.2.1 Définitions et notations

Considérons k utilisateurs indépendants (indicés par u) en compétition pour les ressources d'une plate-forme de calcul parallèle composés de m processeurs identiques. Chaque utilisateur u possède $n^{(u)}$ tâches (nommées $t_i^{(u)}, 1 \leqslant i \leqslant n^{(u)}$). Les tâches sont soumises à un ordonnanceur centralisé.

Soit $n = \sum_{u=1}^{k} n^{(u)}$ le nombre total de tâches. La tâches $t_i^{(u)}$ appartient à l'utilisateur u et a un temps de calcul $p_i^{(u)}$. Dans le cas de dates de disponibilité, la tâche $t_i^{(u)}$ est disponible après le temps $r_i^{(u)}$. Comme il n'y a pas de raison de distinguer les processeurs entre eux, un ordonnancement est défini par la date de départ de chaque tâche : $\sigma(t_i^{(u)})$. La date de terminaison de la tâche $t_i^{(u)}$ est $C_i^{(u)} = \sigma(t_i^{(u)}) + p_i^{(u)}$. Les contraintes usuelles d'ordonnancement s'appliquent.

Le makespan des tâches de l'utilisateur u est $C_{max}^{(u)} = \max_{t_i^{(u)}} C_i^{(u)}$. La somme des temps de terminaison des tâches de l'utilisateur u est$\sum C_i^{(u)}$ (la variante pondéré étant $\sum w_i^{(u)} C_i^{(u)}$). Le *maximum flow time* de l'utilisateur u est $F_{max}^{(u)} = \max_{t_i^{(u)}} C_i^{(u)} - r_i^{(u)}$.

Le problème d'optimisation multi-objectif où chaque objectif est choisi par un utilisateur est $MUSP$ (pour *Multi-Users Scheduling Problem*). Cependant, nous allons considérer les sous problèmes ou les choix des utilisateurs sont donnés dans le problème. Par exemple, nous noterons $MUSP(k' : \sum C_i \ ; \ k'' : C_{max})$ le sous problème où k' utilisateurs veulent optimiser la somme des temps de terminaison et $k'' = k - k'$ utilisateurs veulent optimiser le makespan.

7.2.2 Analyse des travaux connexes

Nous rappelons dans cette section les résultats de deux articles récents traitant du même problème avec des hypothèse simplificatrices. Ils peuvent servir de base à la résolution du problème général. Les deux travaux (Baker et Smith [BS03] et Agnetis *et al.* [AMPP04]) considèrent deux utilisateurs qui se partagent un unique processeur.

Le premier papier [BS03] traite de l'optimisation d'une combinaison linéaire de deux des trois des objectifs suivants : makespan, somme des temps de terminaison (pondérés ou non) et retard maximal. Il montre que toutes les paires d'objectifs peuvent être traitées en temps polynomial à l'exception de

la combinaison du retard maximal et de la somme des temps de terminaison pondérés. De plus, il montre que sur une seule machine, les tâches d'un utilisateur qui veut optimiser le makespan peuvent être fusionnées en une seule tâche.

Ces résultats peuvent être étendus à plusieurs utilisateurs (plus que 2) car la fonction objectif peut toujours être vue comme la combinaison linéaire de deux utilisateurs. En effet, deux utilisateurs voulant optimiser le makespan de leurs tâches peuvent être vus par l'ordonnanceur comme un seul utilisateur qui optimise la somme des temps de terminaison pondérés de deux tâches (chaque tâche étant la fusion des tâches d'un utilisateur). De la même façon les utilisateurs intéressés par le makespan de leurs tâches peuvent être agrégés dans un utilisateur optimisant la somme des temps de terminaison pondérés.

Cependant, optimiser une combinaison linéaire des objectifs des utilisateurs n'est pas une bonne idée. Considérons l'instance impliquant deux utilisateurs dans laquelle le premier veut optimiser le makespan de plusieurs tâches qui sont agrégées en une seule de temps de calcul $p^{(1)}$, tandis que le second veut minimiser la somme des temps de terminaison de $Kp^{(1)}$ tâches unitaires (K entier). L'optimisation de la somme des deux objectifs par l'ordonnanceur implique que toutes les tâches du second utilisateur sont ordonnancées avant celles du premier. Ce comportement n'est pas équitable car les tâches du premier utilisateur peuvent être retardées d'un temps arbitraire lorsque K tend vers l'infini. De plus, le premier utilisateur a intérêt à mentir. Il aurait obtenu de bien meilleures performances s'il avait déclaré qu'il était intéressé par la somme des temps de terminaison au lieu du makespan... En théorie des jeux, on dit que cet ordonnanceur ne garantie pas la véracité (ou *truthfulness* [APTT03].

Un second argument contre les approches à base de combinaisons linéaires des objectifs est que l'ensemble de Pareto d'un problème n'est pas forcément convexe ou bien que toutes les solutions Pareto-optimales pourraient être alignées. Dans ces cas, seulement quelques solutions intéressantes peuvent être atteintes par ces techniques. Le cas où deux utilisateurs sont intéressés par la somme des temps de terminaison dans lequel le premier utilisateur n'a que des tâches de taille 1 et le second n'a que des tâches de taille 2 est un cas où toutes les solutions Pareto-optimales sont alignées.

Le second résultat établi par Agnetis *et al.* [AMPP04] se concentre sur des problèmes de complexité. Il distingue deux questions :

Premièrement, les auteurs s'intéressent au problème d'optimisation contraint où un des objectifs est fixé à une valeur et l'autre objectif est optimisé. C'est exactement les problèmes ϵ-contraint décrit dans [TB07]. Ils montrent que lorsque les deux utilisateurs s'intéressent au makespan, le problème peut

être résolu en temps polynomial. Si les deux utilisateurs sont intéressés par la somme des temps de terminaison, le problème devient $\mathcal{NP}$-complet dans le sens ordinaire et ils fournissent un algorithme pseudo-polynomial pour le résoudre. Lorsque les objectifs sont mixtes, si un utilisateur désire optimiser la somme des temps de terminaison pondérés, le problème est $\mathcal{NP}$-complet dans le sens ordinaire. Les autres cas sont polynomiaux.

Deuxièmement, ils s'intéressent au problème d'énumération de l'ensemble de Pareto. Le point le plus important est la cardinalité de l'ensemble de Pareto. Lorsque un utilisateur est intéressé par le makespan, si un autre est intéressé par le makespan ou la somme des temps de terminaison, alors la cardinalité de l'ensemble est polynomial. Tandis que si le second utilisateur veut optimiser la somme des temps de terminaison pondérés, la question est ouverte. Si les deux utilisateurs veulent optimiser la somme des temps de terminaison, alors la cardinalité de l'ensemble peut être exponentielle.

Tous ces résultats sont intéressants d'un point de vue théorique. Cependant, ils ne peuvent pas être directement utilisés pour traiter le problème multi-utilisateurs multi-processeur.

7.2.3 Discussion sur la fonction à optimiser

L'optimisation multi-objectif est difficile et quasiment aucun problème à plus de trois fonctions objectif n'ont été résolu par la théorie de l'optimisation. Nous pourrions essayer d'agréger les fonctions objectif de tous les utilisateurs en une seule fonction à optimiser.

Dans la littérature, lorsque l'on ordonnance des tâches importantes de tailles différentes, le problème de l'équité apparaît. Il peut être traité par l'utilisation de fonction objectif *stretch* qui est définie comme le temps passé dans le système par une tâche normalisé par son temps de calcul. L'optimisation du stretch a été étudié pour des tâches indépendantes sans préemption [BMR02]. En un certain sens, le stretch d'une tâche est le facteur de dégradation de son temps de traitement qu'elle obtient pour ne pas être la seule tâche du système. Le stretch est également utilisé pour les applications de type *bag-of-tasks* [LSV06] qui ressemblent à notre contexte. Dans le modèle *bag-of-tasks*, des applications sont composées de beaucoup de petites tâches (potentiellement une infinité), alors que dans MUSP, des utilisateurs soumettent des tâches.

On en déduit l'idée d'optimiser une fonction qui ressemble au *stretch* dérivée de la fonction objectif de chaque utilisateur. Si $f^{(u)}$ est la fonction objectif de l'utilisateur u, la dégradation de u dans l'ordonnancement S est

$d^{(u)}(S) = \frac{f^{(u)}(S)}{f^{(u)*}}$ où $f^{(u)*}$ est la valeur minimale de $f^{(u)}$. Si l'on prends $f^{(u)}$ comme le makespan de u, on retrouve "le *stretch* de l'utilisateur u".

L'optimisation du *stretch* des applications nécessite une fonction d'agrégation pour assurer l'équité entre les applications. Trois fonctions d'agrégat sont usuellement considérées, les normes L_∞, L_1, L_2 : minimiser le plus grand *stretch*, la somme des *stretch* et le produits des *stretch*. Ainsi optimiser la somme (ou le maximum) des dégradations semble être l'agrégation logique.

Une telle fonction objectif est $\mathcal{NP}$-difficile à optimiser et des algorithmes d'approximation devraient être proposés. Cependant, un pré requis à la construction d'algorithme d'approximation fait défaut : aucune borne inférieure non triviale sur ces fonction n'est connue.

Nous proposons alors de tenter d'optimiser l'agrégation des dégradations à l'aide de l'approximation multi-objectif. Chaque *stretch* d'application serait un objectif. Considérons le problème de minimiser le k-uplet $(f^{(1)}, \ldots, f^{(k)})$. En utilisant la technique de l'approximation du zénith (voir Chapitre 2), le vecteur d'approximation que l'on obtient est une borne supérieur des dégradations. De plus, cette technique admet généralement des paramètres qui permettent de calibrer le rapport d'approximation [RSTU02]. Ainsi, il serait possible de fournir une approximation (au sens large et non au sens de la théorie de l'approximation) de l'agrégation des dégradations.

7.3 Complexité et inapproximabilité

Nous étudions d'abord la complexité de tous les sous problèmes. Ensuite nous nous intéressons à borner les meilleures performance atteignables par un algorithme d'approximation de la solution zénith.

Optimiser le makespan d'un seul utilisateur sur un nombre arbitraire de processeurs est un problème $\mathcal{NP}$-complet au sens fort [GJ79]. Dès qu'un utilisateur veut optimiser le makespan de ses tâches, le problème devient $\mathcal{NP}$-complet au sens fort. L'optimisation du *maximum flow time* est plus difficile que l'optimisation du makespan. Pour les cas où les utilisateurs sont tous intéressés par la somme des temps de terminaison, rappelons que le problème sur une machine et deux utilisateurs est déjà $\mathcal{NP}$-complet au sens ordinaire [AMPP04]. Toutes les variantes du problèmes sur un nombre arbitraire de processeurs sont $\mathcal{NP}$-complètes.

Comme tous les sous problèmes sont $\mathcal{NP}$-complets, il est vraisemblable qu'aucun algorithme polynomial n'existe. Nous nous intéressons alors aux algorithmes d'approximation de la solution zénith de ce problème. Remarquons d'abord que la valeur optimale pour un objectif particulier est obtenue en or-

donnançant optimalement les tâches d'un utilisateur comme si il était seul à utiliser la machine parallèle. Ainsi tous les résultats classiques d'inapproximabilité mono-objectif s'appliquent. Nous examinons maintenant des instances des trois sous problèmes successivement où tous les utilisateurs choisissent la même fonction objectif et fournissons des résultats d'inapproximabilité. Ils sont résumés dans les propositions suivantes :

PROPOSITION 7.1. *Le zénith de $MUSP(k : C_{max})$ ne peut pas être approché avec un facteur meilleur que $(1, 2, \ldots, k)$ sur un seul processeur.*

Démonstration. Dans ce cas, tous les utilisateurs veulent optimiser le makespan. Nous considérons l'instance suivante. Chaque utilisateur a uniquement une tâche et le système est composé d'un seul processeur. Toutes les tâches sont de même longueur 1. Trivialement, le makespan optimal pour chaque utilisateur est 1. Cependant dans tout ordonnancement efficace, un seul utilisateur aura un makespan de 1, un seul aura un makespan de 2, *etc.*. Il est donc impossible d'obtenir un algorithme qui approche le zénith avec un facteur meilleur que $(1, 2, \ldots, k)$. Remarquons de plus que n'importe quelle permutation de ce k-uplet peut ainsi être obtenue.

Cependant, le vecteur $(1, 2, \ldots, k)$ ne peut pas être atteint dans toutes les instances. Considérons, par exemple, l'instance où les $k - 1$ derniers utilisateurs ont une tâche de longueur 1 alors que le premier a une tâche de longueur k. Si le premier utilisateur obtient un rapport de 1 alors tous les autres obtiennent des rapports supérieurs à k.

En conclusion, le meilleur k-uplet atteignable est valide pour une permutation des utilisateurs qui dépend de l'instance. Ainsi, si l'ordonnanceur cherche à garantir un rapport de performance de $(1, 2, \ldots, k)$, il est impossible de garantir quel utilisateur aura quel rapport d'approximation sans connaître à priori les tâches des autres utilisateurs. □

PROPOSITION 7.2. *Le zénith du problème $MUSP(k : \sum C_i)$ ne peut pas être approché avec un rapport meilleur que $(k, \ldots, k)$ sur un seul processeur.*

Démonstration. Nous considérons l'instance où les k utilisateurs sont intéressés par la somme des temps de terminaison. Chaque utilisateur possède $n^{(u)} = x$ tâches unitaires $(\forall u, \forall i, p_i^{(u)} = 1)$ et le système est composé d'un unique processeur. L'ordonnancement optimal pour un utilisateur est immédiat et vaut $\sum C_i^{(u)*} = \sum_{i=1}^{x} i = \frac{x(x+1)}{2}$.

Quel que soit l'ordre des tâches dans un ordonnancement, la somme des temps de terminaison de toutes les tâches est constante : $\sum C_i^* = \sum_{i=1}^{kx} i = \frac{kx(kx+1)}{2}$. Remarquons que la somme des temps de terminaison de toutes les tâches est la somme des fonctions objectif de tous les utilisateurs : $\sum C_i^* = \sum_u \sum C_i^{(u)}$.

Tous les utilisateurs sont équivalents. Ainsi aucun utilisateur ne devrait obtenir un rapport de performance meilleur que les autres. Nous considérons que l'utilisateur u obtient un k-ième de la somme des temps de terminaison de toutes les tâches, $\sum C_i^{(u)} = \frac{\sum C_i^*}{k} = \frac{kx^2+x}{2}$. Nous pouvons maintenant calculer le rapport d'approximation minimal qui assure l'équité entre les utilisateurs : $\frac{\sum C_i^{(u)}}{\sum C_i^{(u)*}} = \frac{(kx^2+x)/2}{(x^2+x)/2} = k - \frac{k-1}{x+1}$. Ce rapport tend vers k quand x tend vers l'infini. □

PROPOSITION 7.3. *Le zénith du problème $MUSP(k : F_{max})$ n'est pas approximable à facteur constant sur un processeur.*

Démonstration. Nous considérons l'instance avec seulement 2 utilisateurs qui sont tous les deux intéressés par le *maximum flow-time*. Chacun des deux possède x tâches et $\forall i, u, r_i^{(u)} = i - 1, p_i^{(u)} = 1$. L'ordonnancement optimal pour chaque utilisateur indépendamment est donné par la règle FIFO (*First In, First Out*). Ainsi pour tout utilisateur u, $F_{max}^{(u)*} = 1$.

Supposons qu'il existe un ordonnancement tel que $\frac{F_{max}^{(1)}}{F_{max}^{(1)*}} \leqslant \rho$ où ρ est un entier positif plus petit que x. Dans un tel ordonnancement, au plus ρ tâches de l'utilisateur 2 sont ordonnancées avant la dernière tâche de l'utilisateur 1. Il reste $x - \rho$ tâches de l'utilisateur 2 à ordonnancer après les tâches de l'utilisateur 1. Pour chacune de ces tâches, nous avons $F_i^{(2)} \geqslant x$. D'où, $\frac{F_{max}^{(2)}}{F_{max}^{(2)*}} \geqslant x$. Ainsi les deux objectifs ne peuvent être approchés à facteur constant en même temps.

Remarquons que ce résultat est toujours valide si seulement un utilisateur est intéressé par F_{max} et quel que soit l'objectif de l'autre utilisateur. □

L'objectif *maximum flow-time* est plus difficile à approcher que les autres et devrait être traiter d'une manière différente. Dans le reste du chapitre, nous restreignons notre étude au makespan et à la somme des temps de terminaison (pondérés ou non).

7.4 $MUSP(k : C_{max})$

7.4.1 Approximation de la solution zénith

Nous avons montré dans la section précédente que $MUSP(k : C_{max})$ est $\mathcal{NP}$-complet au sens fort ainsi que aucun algorithme polynomial ne peut garantir une solution approchant le zénith à un facteur inférieur à $(1, 2, \ldots, k)$ pour toute permutation des objectifs. Rappelons qu'il existe des algorithmes d'approximation efficaces pour le cas à un utilisateur : une $\mathcal{PTAS}$ proposée

par Hochbaum et Shmoys [HS87] ou encore une $\frac{5}{4}$-approximation proposée par Kellerer [Kel98] (plus de détails en Section 2.2.2).

Considérons l'algorithme suivant appelé MULTICMAX. Pour chaque utilisateur u, calculer une solution $S^{(u)}$ à l'aide d'un algorithme de ρ-approximation uniquement sur les tâches de l'utilisateur u. Ensuite, trions les utilisateurs par valeurs non décroissantes de $C_{max}^{(u)}(S^{(u)})$. Finalement, construire la solution S qui ordonnance les tâches de u de la même façon que dans $S^{(u)}$ entre les temps $\sum_{u'<u} C_{max}^{(u')}(S^{(u')})$ et $\sum_{u'\leqslant u} C_{max}^{(u')}(S^{(u')})$.

Le théorème suivant énonce la garantie de performance de cet algorithme.

THÉORÈME 7.4. MULTICMAX *est une* $(\rho, 2\rho, \ldots, k\rho)$*-approximation de la solution zénith de* $MUSP(k : C_{max})$.

Démonstration. Il y a deux propriétés à vérifier. Premièrement, l'ordonnancement est valide. En effet, les tâches des utilisateurs u sont ordonnancées dans des intervalles de temps disjoints de longueur $C_{max}^{(u)}(S^{(u)})$ suivant $S^{(u)}$, un ordonnancement valide. Deuxièmement, le rapport de performance est de $(\rho, 2\rho, \ldots, k\rho)$. En effet, $C_{max}^{(u)} = \sum_{u'\leqslant u} C_{max}^{(u')}(S^{(u')})$ et les utilisateurs sont ordonnés par valeurs croissantes de $C_{max}^{(u)}(S^{(u)})$. D'où, $C_{max}^{(u)} \leqslant uC_{max}^{(u)}(S^{(u)})$. De plus, $S^{(u)}$ a été généré avec un algorithme ρ-approché. D'où, $C_{max}^{(u)} \leqslant u\rho C_{max}^{(u)*}$. □

Un corollaire direct vient de l'utilisation d'un algorithme exact pour le calcul des $S^{(u)}$ ($\rho = 1$), le rapport obtenu serait de $(1, 2, \ldots, k)$. Cela montre que la borne d'inapproximabilité donné en Proposition 7.1 est atteinte. Remarquons qu'en utilisant une $\mathcal{PTAS}$, nous obtenons une $(1+\epsilon, 2+2\epsilon, \ldots, k+k\epsilon)$-approximation de la solution zénith.

Le problème de la permutation des objectifs abordé en Section 7.3 apparaît également dans le Théorème 7.4 : le théorème est valide pour une permutation donnée des utilisateurs. Remarquons que cette permutation n'est pas connue *a priori*.

7.4.2 Distance à l'ensemble de Pareto

Le rapport d'approximation de MULTICMAX dépend de k et ce rapport est atteint lors de l'utilisation d'un algorithme exact pour le calcul de $S^{(u)}$. On pourrait croire qu'une telle solution est inefficace et donc peu pertinente. Cependant le rapport d'approximation représente la distance entre la solution générée et la solution zénith du problème qui n'est (en général) pas réalisable. Dans cette section, nous présentons une classe de solutions *List*, qui sont générées par MULTICMAX pour $\rho = 2$, et qui est formée de solutions qui sont proches de l'ensemble de Pareto. Nous montrons ce résultat

pour une classe restreinte d'instances dans laquelle chaque utilisateur soumet un nombre raisonnable de tâches. Formellement $\forall u, \sum p_i^{(u)} > \frac{mC_{max}^{(u)*}}{2}$. Notre point dans cette section est de montrer qu'il est possible d'obtenir une approximation constante d'une solution Pareto-optimale tout en conservant l'approximation de la solution zénith.

List Scheduling est un principe qui ordonnance les tâches au plus tôt dans un ordre donné [Gra66]. C'est un algorithme de 2-approximation de $P \mid\mid C_{max}$ dont nous avons déjà parlé dans les chapitres précédents. Plus précisément, si les tâches sont triées dans l'ordre de la liste, la propriété suivante est valide : $\forall i, C_i \leqslant \frac{\sum_{i' \leqslant i} p_{i'}}{m} + (1 - \frac{1}{m})p_i$ (Plus d'informations en Section 2.2.2). Nous appelons *List* les ordonnancements produits par *List Scheduling* lorsque les tâches sont dans un ordre tel que toutes les tâches de u sont après les tâches de $u' < u$ où les utilisateurs sont triés par valeurs non décroissantes de $C_{max}^{(u)*}$. On peut facilement vérifier que chaque solution de *List* peut être dégradée pour obtenir une solution de MULTICMAX avec $\rho = 2$. Ainsi, toute solution de *List* est une $(\rho, 2\rho, \ldots, k\rho)$-approximation du zénith de $MUSP(k : C_{max})$.

Le calcul de la distance entre les solutions de *List* et l'ensemble de Pareto requiert une solution dans ce dernier. Soit *Lex* un ensemble de solutions qui sont optimales pour l'ordre lexicographique des utilisateurs. Formellement *Lex* est défini par : $\forall S \in Lex, C_{max}^{(1)}(S) = C_{max}^{(1)*}; \forall u > 1, C_{max}^{(u)}(S) = \min_{S' \mid \forall u' < u, C_{max}^{(u')}(S') = C_{max}^{(u')}(S)} C_{max}^{(u)}(S')$. Dans la suite, nous noterons $Lex(u)$ l'ensemble des solutions *Lex* restreintes aux u premiers utilisateurs *i.e.*, les tâches des autres utilisateurs ne sont pas considérées. Remarquons que toutes les solutions dans *Lex* ont les mêmes valeurs objectifs. Cela a donc du sens de définir $C_{max}^{(u)}(Lex)$. Nous définissons également $Idle(Lex(u)) = mC_{max}(Lex(u)) - \sum_{u' \leqslant u} \sum_i p_i^{(u')}$, le temps de calcul inoccupé dans les ordonnancements de $Lex(u)$.

PROPRIÉTÉ 7.5. $Idle(Lex(u)) < mC_{max}^{(u)*}$

Démonstration. Soit $t_i^{(u')}$ la dernière tâche qui termine dans une solution $S \in Lex(u)$. $p_i^{(u')} \leqslant C_{max}^{(u)*}$ car $p_i^{(u')} \leqslant C_{max}^{(u')*} \leqslant C_{max}^{(u)*}$ (pour $u' \leqslant u$) .

Par contradiction, si $Idle(Lex(u)) \geqslant mC_{max}^{(u)*}$, alors il existe un processeur qui est inoccupé pendant plus de $C_{max}^{(u)*}$ unités de temps. Ce processeur pourrait ordonnancer $t_i^{(u')}$ plus tôt. D'où, l'ordonnancement S n'appartient pas à $Lex(u)$. □

Si l'hypothèse précédente $\forall u, \sum p_i^{(u)} > \frac{mC_{max}^{(u)*}}{2}$ n'est pas vérifiée, il est facile de construire une instance où chaque utilisateur a une unique tâche qui devrait être ordonnancée au temps 0. N'importe quelle solution de *List* ne

l'ordonnance pas aussi tôt. Nous utilisons l'hypothèse pour prouver le lemme suivant :

LEMME 7.6. *$\forall u > 2$, si $C_{max}^{(u-1)}(Lex) < C_{max}^{(u-2)}(Lex)$ alors $C_{max}^{(u)}(Lex) > C_{max}^{(u-2)}(Lex)$*

Démonstration. Si $C_{max}^{(u-1)}(Lex) < C_{max}^{(u-2)}(Lex)$ alors $Idle(Lex(u-1)) = Idle(Lex(u-2)) - \sum p_i^{(u-1)}$.

En se rappelant que $Idle(Lex(u-2)) < mC_{max}^{(u-2)*}$ (de la Propriété 7.5), on obtient $Idle(Lex(u-1)) < mC_{max}^{(u-2)*} - \sum p_i^{(u-1)}$. Comme $\sum p_i^{(u-1)} > \frac{mC_{max}^{(u-1)*}}{2}$ (de l'hypothèse) et $C_{max}^{(u)*} \geqslant C_{max}^{(u-1)*} \geqslant C_{max}^{(u-2)*}$ (du tri des utilisateurs), on obtient $Idle(Lex(u-1)) < \frac{mC_{max}^{(u)*}}{2}$. D'où, toutes les tâches de l'utilisateur u ne peuvent pas rentrer dans les temps d'inactivité présent avant $C_{max}^{(u-2)}(Lex)$. Au moins une des tâches de u doit finir après cette valeur. □

COROLLAIRE 7.7. $\forall u > 2, C_{max}^{(u)}(Lex) > C_{max}^{(u-2)}(Lex)$

La preuve du corollaire vient directement du lemme précédent.

Le prochain théorème montre que n'importe quelle solution de *List* est une 3-approximation de *Lex*.

THÉORÈME 7.8. $\forall u, C_{max}^{(u)}(List) \leqslant (3 - \frac{1}{m})C_{max}^{(u)}(Lex)$

Démonstration. Soit $p_{max}^{(u)} = \max_i p_i^{(u)}$. Nous déduisons que l'analyse de *List Scheduling* [Gra66] : $C_{max}^{(u)}(List) \leqslant \frac{\sum_{u' \leqslant u} \sum_i p_i^{(u')}}{m} + (1 - \frac{1}{m})p_{max}^{(u)}$ que l'on peut écrire de la façon suivante :

$$C_{max}^{(u)}(List) \leqslant \left(\frac{\sum_{u' \leqslant u-2} \sum_i p_i^{(u')}}{m} + \frac{\sum_i p_i^{(u)}}{m} \right) + \left(\frac{\sum_i p_i^{(u-1)}}{m} \right) + (1 - \frac{1}{m})p_{max}^{(u)}$$

Le premier terme de l'expression est inférieur à $C_{max}^{(u)}(Lex)$ (c'est une conséquence du Corollaire 7.7). Le deuxième terme est inférieur à $C_{max}^{(u)*}$ car c'est une borne inférieure de $C_{max}^{(u-1)*}$ et que les utilisateurs sont triés en ordre croissant de leur makespan optimal. $p_{max}^{(u)}$ est une borne inférieure de $C_{max}^{(u)*}$.

Finalement, nous obtenons l'expression : $C_{max}^{(u)}(List) \leqslant C_{max}^{(u)}(Lex) + (2 - \frac{1}{m})C_{max}^{(u)*}$. Nous déduisons le théorème de $C_{max}^{(u)*} \leqslant C_{max}^{(u)}(Lex)$. □

Dans l'analyse précédente, il est nécessaire de connaître le makespan optimal de chaque utilisateur pour construire une solution de *List*. Cependant, si on connaît uniquement une ρ-approximation du makespan de chaque utilisateur, alors le rapport à l'ensemble de Pareto reste constant. Il faut alors se comparer à la solution *Lex* où les utilisateurs sont triés suivant les ρ-approximations des makespans de leur tâches. Le raisonnement est alors le même. Un facteur ρ apparaît alors dans certaines expressions mais le rapport final demeure constant.

7.5 $MUSP(k : \sum C_i)$

Nous nous intéressons maintenant au cas où tous les utilisateurs veulent optimiser la somme des temps de terminaison de leurs tâches. Rappelons que pour un seul utilisateur, ce problème peut être résolu en temps polynomial en ordonnançant les tâches en ordre non décroissant de temps de calcul (il s'agit de l'algorithme SPT, voir la Section 2.2.5). Cependant, pour plus d'un utilisateur, ce problème est $\mathcal{NP}$-complet. De plus, aucun algorithme ne peut garantir une solution qui approche le zénith à moins de $(k, \ldots, k)$.

Pour résoudre ce problème, nous étudions d'abord le cas mono-processeur pour lequel nous proposons et analysons un algorithme. Puis nous étendons cette analyse à un nombre arbitraire de processeurs.

7.5.1 Analyse préliminaire pour $m = 1$

De la même façon que dans la Section 7.4, nous calculons l'ordonnancement final à partir d'ordonnancement des tâches de chaque utilisateur. Soit $S^{(1)}, \ldots, S^{(k)}$ des ordonnancements des tâches de chaque utilisateur. Nous construisons un ordonnancement S à l'aide de l'algorithme glouton AGGREG suivant : Ordonnancer les tâches en ordre non décroissant de $C_i^{(u)}(S^{(u)})$.

Cet algorithme simple assure que les temps de terminaison ne sont pas dégradés de plus qu'un facteur k. Cela est énoncé par la proposition suivante :

PROPOSITION 7.9. *L'ordonnancement S est tel que $\forall u, \forall i \leqslant n^{(u)}, C_i^{(u)}(S) \leqslant kC_i^{(u)}(S^{(u)})$.*

Démonstration. Considérons une tâche particulière $t_i^{(u)}$. Remarquons d'abord que $C_i^{(u)}(S) \leqslant C_{i'}^{(u')}(S)$ implique $C_i^{(u)}(S^{(u)}) \leqslant C_{i'}^{(u')}(S^{(u')})$. D'où pour chaque utilisateur u', l'ensemble $J^{(u')}$ de tâches ordonnancées avant $C_i^{(u)}(S)$ dans S est tel que $\sum_{t_{i'}^{(u')} \in J^{(u')}} p_{i'}^{(u')} \leqslant C_i^{(u)}(S^{(u)})$. Il y a un ensemble par utilisateur. $t_i^{(u)}$ se termine quand toutes les tâches de $\cup_{u'} J^{(u')}$ sont terminées. D'où, $C_i^{(u)}(S) = \sum_{u'} \sum_{t_{i'}^{(u')} \in J^{(u')}} p_{i'}^{(u')} \leqslant kC_i^{(u)}(S^{(u)})$ ce qui prouve la proposition □

L'algorithme précédent considère chaque utilisateur avec la même priorité. Nous pouvons construire un autre algorithme qui ne donne pas la même priorité à tous les utilisateurs. Soit $\lambda_1, \ldots, \lambda_k$ des réels positifs tel que $\sum_u \lambda_u = 1$. Nous construisons l'ordonnancement S^λ à l'aide de l'algorithme glouton appelé AGGREG$^\lambda$: Ordonnancer les tâches en ordre non décroissant de $\frac{C_i^{(u)}(S^{(u)})}{\lambda_u}$. Remarquons que si $\forall u, \lambda_u = \frac{1}{k}$ alors $S = S^\lambda$.

Les temps de terminaison dans S^λ des tâches appartenant à l'utilisateur u ne sont pas dégradés de plus qu'un facteur $\frac{1}{\lambda_u}$. Ce fait est énoncé par la proposition suivante. Sa preuve est similaire à la preuve de la Proposition 7.9 et est omise.

PROPOSITION 7.10. *L'ordonnancement S^λ est tel que :*
$\forall u, \forall i \leqslant n^{(u)}, C_i^{(u)}(S) \leqslant \frac{C_i^{(u)}(S^{(u)})}{\lambda_u}$.

À l'aide AGGREG, il est possible de fusionner k ordonnancements et d'assurer que le temps de terminaison de chaque tâche n'est pas dégradé de plus qu'un facteur k. En particulier, si un ordonnancement est une ρ-approximation de la somme des temps de terminaison (pondérés ou non) alors en le fusionnant avec $k-1$ autres ordonnancements, nous obtenons une $(k\rho)$-approximation pour cet utilisateur.

LEMME 7.11. AGGREG $(SPT(1), \ldots, SPT(k))$ *est une $(k, \ldots, k)$-approximation de la solution zénith de $MUSP(k : \sum C_i)$ sur un processeur.*

La preuve de ce lemme vient directement de l'optimalité de SPT pour un utilisateur et de la Proposition 7.9.

7.5.2 Extension à m processeurs

Nous décrivons maintenant l'algorithme MULTISUM. Soit $S^{(1)}, \ldots, S^{(k)}$ k ordonnancements SPT sur m processeurs. Chaque ordonnancement $S^{(u)}$ peut être vu comme m ordonnancements indépendants notés $S_1^{(u)}, \ldots, S_m^{(u)}$. Pour chaque processeur j, les ordonnancements $S_j^{(u)}$ de chaque utilisateur u sont fusionnés à l'aide de AGGREG dans S_j. Ensuite, un ordonnancement global sur m processeurs est construit en exécutant S_j sur le processeur j $(1 \leqslant j \leqslant m)$.

THÉORÈME 7.12. MULTISUM *est une $(k, \ldots, k)$-approximation du zénith de $MUSP(k : \sum C_i)$.*

La preuve du Théorème 7.12 vient directement du Lemme 7.11 et de l'indépendance des processeurs.

L'optimisation de la somme des temps de terminaison pondérés est $\mathcal{NP}$-difficile même pour un seul utilisateur. Cependant, il existe des algorithmes d'approximation, par exemple celui de Hall *et al.* qui obtient un rapport de performance de $(4 - \frac{1}{m})$ [HSSW97]. En utilisant une ρ-approximation pour la somme des temps de terminaison pondérés à la place de SPT amène à l'algorithme MULTIWEIGHTEDSUM.

COROLLAIRE 7.13. MULTIWEIGHTEDSUM *est une* $(\rho k, \ldots, \rho k)$*-approximation du zénith de* $MUSP(k : \sum w_i C_i)$.

En utilisant l'algorithme de Hall *et al.*, nous obtenons une $((4-\frac{1}{m})k, \ldots, (4-\frac{1}{m})k)$-approximation.

7.6 Le cas mixte

Dans cette section, nous nous intéressons au cas mixte où k' utilisateurs veulent optimiser la somme des temps de terminaison alors que k'' utilisateurs s'intéressent au makespan de leurs tâches : $MUSP(k' : \sum C_i \ ; \ k'' : C_{max})$ avec $k = k' + k''$.

Remarquons que la technique présentée dans la section précédente pour la somme des temps de terminaison fonctionne également pour le makespan. Chaque utilisateur obtiendra un rapport de performance de k. Cependant, nous pouvons espérer améliorer ce résultat. En effet, si aucun utilisateur n'est intéressé par la somme des temps de terminaison, nous pouvons obtenir le rapport $(1, \ldots, k)$. Tandis que si un utilisateur est intéressé par la somme des temps de terminaison, le rapport de tous les utilisateurs intéressés par le makespan se dégrade à k.

L'idée dans cette section est d'incorporer les techniques présentées dans les Sections 7.4 et 7.5 dans un unique algorithme d'ordonnancement appelé MULTIMIXED.

Considérons la sous instance composée des k'' utilisateurs intéressés par le makespan. Nous remplaçons ces utilisateurs par un seul utilisateur $cmax$. Soit $S^{(cmax)}$ l'ordonnancement généré par MULTICMAX (à l'aide d'un algorithme de ρ-approximation) sur cette sous instance. Pour chaque utilisateur u intéressé par la somme des temps de terminaison, soit $S^{(u)}$ l'ordonnancement SPT des ses tâches sur m processeurs. Soit $S_j^{(u)}$ le sous ordonnancement de $S^{(u)}$ sur le $j$ième processeur ($S_j^{(cmax)}$ est défini de la même manière). Pour chaque processeur j, on fusionne les ordonnancements $S_j^{(u)}$ $(\forall u \leqslant k')$ et $S_j^{(cmax)}$ dans S_j à l'aide de AGGREG$^\lambda$ avec $\lambda_u = \frac{1}{k}, \forall u \leqslant k'$ et $\lambda_{cmax} = \frac{k''}{k}$. Nous construisons l'ordonnancement global S en exécutant S_j sur le processeur j.

La garantie théorique de MULTIMIXED est énoncée dans le théorème suivant.

THÉORÈME 7.14. MULTIMIXED *est une* $(k, \ldots, k, \frac{k}{k''}\rho, \frac{2k}{k''}\rho, \ldots, k\rho)$*-approximation du zénith de* $MUSP(k' : \sum C_i \ ; \ k'' : C_{max})$.

Démonstration. Tous les utilisateurs intéressés par la somme des temps de terminaison obtient un rapport de k. La preuve de cette affirmation est similaire à celle du Théorème 7.12 et est laissée au lecteur.

Considérons maintenant un utilisateur u voulant optimiser le makespan. Sans perte de généralité, nous avons $C_{max}^{(u)}(S^{(cmax)}) \leqslant u\rho C_{max}^{(u)*}$ (grâce au Théorème 7.4) ce qui signifie que $\forall i \leqslant n^{(u)}, C_i^{(u)}(S^{(cmax)}) \leqslant u\rho C_{max}^{(u)*}$.

La Proposition 7.10 nous indique que les dates de terminaison des tâches des utilisateurs intéressés par C_{max} ne sont pas dégradées d'un facteur supérieur à $\frac{k}{k''}$. D'où, $\forall i \leqslant n^{(u)}, C_i^{(u)}(S^{(cmax)}) \leqslant \frac{k}{k''} u\rho C_{max}^{(u)*}$ qui termine la preuve. □

Nous discutions en Section 7.2.3 de considérer l'optimisation de fonctions de dégradation. Remarquons que la garantie de performance de l'algorithme MULTIMIXED permet assez facilement d'optimiser la somme, ou le maximum des dégradations.

7.7 Conclusion

Dans ce chapitre, nous avons considéré le problème d'ordonnancement à plusieurs utilisateurs. La version du problème dans laquelle les utilisateurs choisissent un objectif parmi une liste est nouvelle et les cas multiprocesseurs sont considérés pour la première fois dans ce chapitre. La contribution principale sur ce problème est de montrer que le problème est traitable : à l'aide d'outils d'optimisation multi-objectif, il est possible de produire des ordonnancements efficace pour tous les utilisateurs.

D'un point de vue plus général, deux points sont importants. Premièrement, les algorithmes de ce chapitre permettent d'optimiser un nombre arbitraire d'objectifs simultanément. Certes, de tels algorithmes ont déjà été exhibés par le passé. Cependant, ils restent rares.

Deuxièmement, l'approximation de la solution zénith laisse parfois perplexe. Nous avons proposé une analyse permettant de confirmer l'efficacité d'une approximation de la solution zénith en montrant que c'est une meilleur approximation d'une solution Pareto-optimale.

Chapitre 8

Conclusion

8.1 Bilan

Nous avons considéré durant cette thèse quatre problèmes d'ordonnancement multi-objectifs. Les quatre problèmes ont été traités à l'aide des outils de l'optimisation multi-objectif. Nous présentons un bilan de cette thèse à travers deux points de vue différents. Le premier est centré sur les problèmes tandis que le deuxième est centré sur le domaine de l'optimisation multi-objectif.

Le premier problème considère l'optimisation simultanée de deux objectifs de type makespan lors de l'ordonnancement de tâches indépendantes ou d'un graphe de tâches. Des algorithmes d'approximation du zénith ont été proposés et des bornes inférieures sur les meilleurs rapports d'approximation ont été exhibées. Ces dernières montrent qu'il est difficile d'obtenir de bien meilleurs résultats par la technique de l'approximation du zénith.

Le deuxième problème traite de l'optimisation du makespan et de la fiabilité dans les systèmes critiques, réactifs et embarqués. La difficulté du problème implique que peu de résultats théoriques pouvaient être attendus. Nous avons montré que le zénith du problème n'est pas approximable. Ainsi, nous avons proposé une méthode heuristique d'approximation de l'ensemble de Pareto.

Le troisième problème traite également de l'optimisation du makespan et de la fiabilité, mais cette fois ci dans les *clusters* et grilles. Les différences de modèles rendent le problème plus facilement traitable que le problème précédent. Sur ce problème encore, nous avons montré que le zénith du problème n'est pas approximable à facteur constant. Sur le sous-problème où toutes les tâches sont indépendantes, nous avons proposé un algorithme efficace d'ap-

proximation de l'ensemble de Pareto. Ces résultats ont servi à adapter des heuristiques existantes pour le problème général où un graphe de tâches doit être ordonnancé.

Le dernier problème traite de l'optimisation de l'ordonnancement des tâches de nombreux utilisateurs en concurrence pour une ressource de calcul parallèle, conformément aux besoins des utilisateurs. Nous montrons que pour les fonctions à optimiser de types *flow-time*, aucune approximation constante du zénith n'est possible. Nous fournissons des algorithmes d'approximation atteignant les bornes d'approximabilité pour les cas où tous les utilisateurs s'intéressent soit au makespan soit à la somme des temps de terminaison. Nous fournissons également un algorithme permettant de traiter le cas mixte. Les rapports d'approximation étant linéairement dépendant du nombre d'utilisateurs, nous proposons, dans un cas particulier, l'analyse d'une optimisation locale qui permet de montrer que la solution est en plus une approximation à facteur constant (indépendant du nombre d'utilisateurs) d'une solution Pareto-optimale.

Rappelons que cette thèse n'a pas uniquement pour but de résoudre des problèmes multi-objectifs, mais aussi d'apporter une contribution d'ordre méthodologique à l'optimisation multi-objectif. Les deux sections suivantes fournissent une réponse à deux questions fondamentales.

8.2 Pourquoi utiliser l'optimisation multi-objectif ?

L'optimisation multi-objectif est un domaine difficile. Cependant, plusieurs raisons de l'utiliser apparaissent.

Le problème applicatif que l'on considère est multi-objectif. Cette raison est la plus simple de toutes. Nous l'avons illustré dans les Chapitres 5 et 6 sur des problèmes de sûreté de fonctionnement : l'optimisation des performances d'une application et de sa fiabilité sont globalement contradictoires. D'autres exemples existent dans la littérature, comme par exemple l'efficacité et la consommation énergétique.

Pour traiter des problèmes non approximables. Les contraintes d'un problème peuvent rendre difficile la simple construction d'une solution et ainsi rendre le problème non approximable. Transformer une contrainte en objectif est un processus classique en optimisation ; c'est ce que l'on fait lorsque l'on passe d'un problème de décision à un problème d'optimisation. C'est également la base des techniques de Lagrangien. Cependant, considérer une contrainte comme un objectif alors que le problème est déjà un problème d'optimisation est (à notre connaissance) une approche nouvelle. Cela peut

permettre de fournir un critère sur la difficulté de création d'une instance ou encore d'exprimer la qualité de la solution en fonction du niveau de serrage de la contrainte. Nous avons appliqué avec succès cette méthode sur le problème du Chapitre 3.

Comme une méthode d'optimisation d'objectif difficile à approcher. Certains objectifs sont difficiles à optimiser. Cependant si l'on peut exprimer un tel objectif comme une fonction de plusieurs variables monotones pour l'ordre de Pareto, alors on peut optimiser l'objectif difficile en considérant le problème d'optimisation multi-objectif où chaque variable de la fonction est un objectif à optimiser. Cette idée a été utilisée dans le Chapitre 7 lorsque nous montrons que les rapports d'approximation au zénith sont des bornes supérieures de la dégradation de chaque utilisateur.

8.3 Comment traiter un problème multi-objectif ?

Finalement, la résolution des problèmes multi-objectifs est le point central de cette thèse. Nous proposons maintenant un guide pour traiter ces problèmes.

La première question à résoudre concerne chaque objectif indépendamment. Il est évident que l'on n'obtiendra pas de meilleurs résultats sur le problème multi-objectif que sur les différents problèmes mono-objectifs associés. Les problèmes mono-objectifs sont ils polynomiaux ou $\mathcal{NP}$-complets ? Disposons nous d'algorithmes d'approximation pour les résoudre ? Quelles sont les propriétés des bonnes solutions pour chaque objectif ?

Il est important de rappeler que le problème de décision associé au problème multi-objectif peut être $\mathcal{NP}$-complet même si tous les problèmes mono-objectifs sont polynomiaux.

La structure de l'ensemble de Pareto est primordiale. Peut on borner le nombre de solutions Pareto-optimales ? L'ensemble de Pareto est il convexe ? Toutes les solutions Pareto-optimales sont elles sur l'enveloppe convexe de l'ensemble des solutions ? Peut-on borner les valeurs objectifs des solutions Pareto-optimales ?

La solution zénith n'est généralement pas réalisable. Si elle l'est systématiquement, alors le problème considéré n'est pas réellement multi-objectif. Il est souvent facile de borner inférieurement le rapport d'approximation de n'importe quel algorithme d'approximation du zénith à l'aide d'instances bien choisies.

Pour résoudre le problème multi-objectif, plusieurs approches se distinguent :

- L'énumération de l'ensemble de Pareto est un choix envisageable si cet ensemble est de taille raisonnable et si trouver les solutions Pareto-

optimales est un problème facile.
- Les approches d'agrégation par combinaison linéaire sont pertinentes lorsque toutes les solutions Pareto-optimales sont sur l'enveloppe convexe et qu'il est possible d'obtenir la solution optimale pour cette métrique.
- L'approximation de la solution zénith permet d'avoir une garantie absolue sur la qualité de la solution générée. Cette approche n'est possible que si l'ensemble de Pareto est pour toutes les instances à distance bornée de la solution zénith.
- L'approximation de l'ensemble de Pareto par l'utilisation d'algorithme de $\langle \bar{\alpha}, \beta \rangle$-approximation fournie une approximation en un nombre polynomial de solutions si l'ensemble de Pareto n'est pas trop dispersé dans l'espace des solutions. Remarquons que si cet ensemble est dispersé, alors le représenter à l'aide d'un nombre exponentiel de solutions semble raisonnable.

8.4 Perspectives

La théorie de l'optimisation multi-objectif est aujourd'hui mieux connue mais de nombreux points restent à aborder.

Un point central est d'ordre théorique et concerne les liens avec la théorie de la complexité. Les seules classes de complexité spécifiques à l'optimisation multi-objectif qui existent sont des classes de problèmes de comptage et d'énumération. Elles sont assez peu utiles en pratique car elles traitent de méthodes exactes. Des équivalents aux classes $\mathcal{APX}$, $\mathcal{PTAS}$ et $\mathcal{FPTAS}$ restent à proposer en optimisation multi-objectif.

Du point de vue de la méthode, deux pistes peuvent être suivies. D'abord, beaucoup de méthodes d'approximation du zénith disposent d'un paramètre de compromis permettant de trouver de nombreuses solutions différentes. Exprimer l'approximation paramétrique du zénith en termes d'approximation de l'ensemble de Pareto lorsque cela est possible augmenterai la compréhension de ces techniques.

Ensuite, il reste à trouver des méthodes pour traiter les problèmes à beaucoup d'objectifs. Obtenir une approximation de l'ensemble de Pareto n'est pas suffisant car la cardinalité de l'approximation est exponentielle dans le nombre d'objectifs. Nous avons traité le problème dans le Chapitre 7 en assurant l'équité dans des dégradations des utilisateurs. L'extension de cette technique à d'autres problèmes devrait être étudiée. Plus généralement, des méthodes pour réduire la cardinalité des approximations d'ensemble de Pareto devrait être développées.

Finalement, certains concepts utilisés en optimisation multi-objectif viennent

de la théorie des jeux, comme par exemple la véracité garantie ou l'équité. L'optimisation multi-objectif n'est pas adapté aux systèmes où la prise de décision est distribuée, mais les concepts qui y sont étudiés pourraient certainement s'intégrer dans les protocoles distribués.

Bibliographie

[ABG05] E. Angel, E. Bampis, and L. Gourvès. Approximation results for a bicriteria job scheduling problem on a single machine without preemption. *Information Processing Letters*, 94(1) :19–27, April 2005.

[ABP01] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2) :115–144, 2001.

[ACG+03] G. Ausiello, P. Crecenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation.* Springer, 2003.

[Aea02] N. R. Adiga and et al. An overview of the blue gene/l supercomputer. In *Proceedings of the ACMjIEEE Conference on Super computing*, pages 1–22, 2002.

[AF06] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. In Springer LNCS, editor, *Proceedings of the 23rd International Symposium on Theoretical Aspects of Computer Science*, volume 3884, pages 621–633, 2006.

[AFBD06] T. Angskun, G. Fagg, J. Bosilca, G.and Pjesivac-Grbovic, and J. Dongarra. Scalable Fault Tolerant Protocol for Parallel Runtime Environments. In *Euro PVM/MPI*, Bonn, Germany, 2006.

[AGK04] I. Assayad, A. Girault, and H. Kalla. A bi-criteria scheduling heuristic for distributed embedded systems under reliability and real-time constraints. In *2004 International Conference on Dependable Systems and Networks (DSN'04)*, pages 347–356, June 2004.

[AGM06a] I. Abraham, C. Gavoille, and D. Malkhi. On space-stretch trade-offs : Lowerbound. In *Proc. of SPAA'06*, pages 207–216, 2006.

[AGM06b] I. Abraham, C. Gavoille, and D. Malkhi. On space-stretch trade-offs : Upper bounds. In *Proc. of SPAA'06*, pages 217–224, 2006.

[AMPP04] A. Agnetis, P. B. Mirchandani, D. Pacciarelli, and A. Pacifici. Scheduling problems with two competing agents. *Operations Research*, 52(2) :229–242, April 2004.

[APTT03] A. Archer, C. Papadimitriou, K. Talwar, and E. Tardos. An approximate truthful mechanism for combinatorial auctions with single parameter agent. In *Proc. of the 14th Annual ACM Symposium on Discrete Algorithms (SODA)*, 2003.

[ARSY99] J. Aslam, A. Rasala, C. Stein, and N. Young. Improved bicriteria existence theorems for scheduling. In *SODA '99 : Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 846–847, 1999.

[BESW93] J. Blazewicz, K. Ecker, G. Schmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems.* Springer-Verlag, 1993.

[BFM06] V. Bilò, M. Flammini, and L. Moscardelli. Pareto approximations for the bicriteria scheduling problem. *JPDC*, 66(3) :393–402, 2006.

[BHK+05] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. Mpich-v : a multiprotocol fault tolerant mpi. *International Journal of High Performance Computing and Applications*, 2005.

[BMR02] M. A. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 762–771, 2002.

[BMS05] J.-C. Billaut, A. Moukrim, and E. Sanlaville, editors. *Flexibilité et robustesse en ordonnancement.* Hermes Lavoisier, 2005.

[Bou01] V. Boudet. Heterogeneous task scheduling : a survey. Technical Report RR2001-07, ENS Lyon, February 2001.

[BP96] R. E. Barlow and F. Proschan. *Mathematical Theory of Reliability.* SIAM, classics edition, 1996.

[BRSR08] A. Benoit, V. Rehn-Sonigo, and Y. Robert. Optimizing latency and reliability of pipeline workflow applications. In *Proc. of Heterogeneity in Computing Workshop*, 2008.

[Bru95] P. Brucker. *Scheduling Algorithms.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[BS03] K. Baker and J.C. Smith. A multiple-criterion model for machine scheduling. *Journal of Scheduling*, 6 :7–16, 2003.

[Bun06] D. Bunde. Power-aware scheduling for makespan and flow. In *Proceedings of the 18th Symposium of Parallelim in Algorithms and Architecture*, pages 190–196, 2006.

[CB01] C. Chekuri and M. A. Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. *Journal of Algorithms*, 41 :212–224, 2001.

[CBB+05] S. Campana, D. Barberis, F. Brochu, A. De Salvo, F. Donno, L. Goossens, S. Gonzalez de la Hoz, T. Lari, D. Liko, J. Lozano, G. Negri, L. Perini, G. Poulard, S. Resconi, D. Rebatto, and L. Vaccarossa. Analysis of the atlas rome production experience on the lhc computing grid. In *E-SCIENCE '05 : Proceedings of the First International Conference on e-Science and Grid Computing*, pages 82–89, Washington, DC, USA, 2005. IEEE Computer Society.

[CCLL95] P. Chretienne, E.G. Coffman, J.K Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications.* Wiley, 1995.

[CDF+08] C. Y. Crutchfield, Z. Dzunic, J. T. Fineman, D. R. Karger, and J. H. Scott. Improved approximations for multiprocessor scheduling under uncertainty. In *SPAA '08 : Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 246–255, New York, NY, USA, 2008. ACM.

[CJ01] B. Cirou and E. Jeannot. Triplet : a Clustering Scheduling Algorithm for Heterogeneous Systems. In *IEEE ICPP International Workshop on Metacomputing Systems and Applications (MSA'2001)*, Valencia, Spain, September 2001.

[CJ04] Y. Caniou and E. Jeannot. Experimental study of multicriteria scheduling heuristics for GridRPC systems. In *Proc. of Euro-Par*, pages 1048–1055, 2004.

[CKC07] P. Choudhury, R. Kumar, and P. P. Chakrabarti. Hybrid scheduling of dynamic task graphs with selective duplication for multiprocessors under memory and time constraints. *IEEE Transactions on Parallel and Distributed Systems*, (preprint), 2007.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[CP00] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3) :249–274, 2000.

[DEMT04] P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bicriteria algorithm for scheduling jobs on cluster platforms. In *Symposium on Parallel Algorithms and Architectures*, pages 125–132. ACM Press, 2004.

[Det00] M. Detyniecki. *Mathematical aggregation operators and their application to video querying.* PhD thesis, Université Paris VI, 2000. `http://www.lip6.fr/reports/lip6.2001.002.html`.

[DLMM04] T. Davidovic, L. Liberti, N. Maculan, and N. Mladenovic. Mathematical programming-based approach to scheduling of communicating tasks. Technical report, LIX, December 2004.

[DÖ02] A. Dogan and F. Özguner. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3) :308–324, March 2002.

[DÖ05] A. Dogan and F. Özgüner. Biobjective scheduling algorithms for execution time-reliability trade-off in heterogeneous computing systems. *The Computer Journal*, 48(3) :300–314, 2005.

[DRTV07] J.-G. Dumas, J.-L. Roch, E. Tannier, and S. Varrette. *Théorie des Codes - Compression Cryptage Correction.* Dunod Sciences-Sup, Paris, France, February 2007.

[Fei] D. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation.* draft. `http://www.cs.huji.ac.il/~feit/wlmod/`.

[FHKS08] B. Fechner, U. Hönig, J. Keller, and W. Schiffmann. Fault-tolerant static scheduling for grids. In *Elec Proc. of Dependable Parallel, Distributed and Network-Centric Systems*, 2008.

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Re-

liable and precise WCET determination for a real-life processor. In *International Workshop on Embedded Software, EMSOFT'01*, volume 2211 of *LNCS*, Tahoe City (CA), USA, October 2001. Springer-Verlag.

[GIS77] T. Gonzalez, O.H. Ibarra, and S. Sahni. Bounds for LPT schedules on uniform processors. *SIAM Journal of Computing*, 6 :155–166, 1977.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman, San Francisco, 1979.

[GKar] A. Girault and H. Kalla. A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate. *IEEE Transactions on Dependable and Secure Computing*, To appear. INRIA research report 6319. http ://hal.inria.fr/inria-00177117.

[GKS06] A. Girault, H. Kalla, and Y. Sorel. Transient processor/bus fault tolerance for embedded systems. In *IFIP Working Conference on Distributed and Parallel Embedded Systems, DIPES'06*, pages 135–144, Braga, Portugal, October 2006. Springer-Verlag.

[GLLK79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. *ann. Discrete Math.*, 5 :287–326, 1979.

[GLR07] J. Gauthier, X. Leduc, and A. Rauzy. Assessment of large automatically generated fault trees by means of binary decision diagrams. *J. of Risk and Reliability*, 221(2) :95–105, 2007.

[Gol99] A. Goldman. *Impact des modèles d'exécution pour l'ordonnancement en calcul parallèle.* PhD thesis, INPG, 1999.

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

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

[Hal97] L.A. Hall. Approximation algorithms for scheduling. In Hochbaum [Hoc97], chapter 1, pages 1–45.

[HB06] M. Hakem and F. Butelle. A Bi-objective Algortithm for Scheduling Parallel Applications on Heterogeneous Systems

Subject to Failures. In *Renpar 17*, Canet en Roussillon, October 2006.

[Ho04] K. Ho. Dual criteria optimization problems for imprecise computation tasks. In Leung [Leu04a], chapter 35.

[Hoc97] D. S. Hochbaum, editor. PWS publishing company, 20 park plazma. Boston, MA 02116, 97.

[HS87] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems : Practical and theoretical results. *Journal of ACM*, 34 :144–162, 1987.

[HS88] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors : Using the dual approximation approach. *SIAM Journal on Computing*, 17(3) :539 – 551, 1988.

[HSSW97] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time : Off-line and on-line approximation algorithms. In *SODA : ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[HSZT00] C. Hirel, R. Sahner, X. Zang, and K.S. Trivedi. Reliability and performability modeling using Sharpe. In *International Conference on Computer Performance Evaluation : Modelling Techniques and Tools, TOOLS'00*, volume 1786 of *LNCS*, pages 345–349. Springer-Verlag, March 2000.

[IK77] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the Association for Compution Machinery*, 24(2) :280–289, April 1977.

[Jaf06] S. Jafar. *Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité*. PhD thesis, INPG, June 2006.

[Jai91] R. Jain. Ratio games. In *The Art Of Computer Systems Performance Analysis*, chapter 11, pages 165–174. Wiley professional computing, New York, USA, 1991.

[JSC01] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the maui scheduler. In D. G. Feitelson and L. Rudolph, editors, *Proc. of the 7th International workshop JSSPP*, number 2221 in LNCS. Springer, 2001.

[JST08] E. Jeannot, E. Saule, and D. Trystram. Bi-objective approximation scheme for makespan and reliability optimization on

uniform parallel machines. In *Euro-Par 2008*. LNCS, August 2008.

[KA98] Y-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *IPPS/SPDP*, pages 531–537, 1998.

[Kal04] H. Kalla. *Génération automatique de distributions/ordonnancements temps réel, fiables et tolérants aux fautes*. PhD thesis, INPG, Grenoble, 2004.

[KB03] V. Kianzad and S. Bhattacharyya. A comparison of clustering and scheduling techniques for embedded multiprocessor systems. Technical report, Institute for Advanced Computer Studies, December 2003.

[Kel98] H. Kellerer. Algorithms for multiprocessor scheduling with machine release times. *IIE Transactions*, 30(11) :991–999, November 1998.

[KM97] S. Kartik and C.S.R. Murthy. Task allocation algorithms for maximising reliability of distributed computing systems. *IEEE Transaction on Computers*, 46(6) :719–724, June 1997.

[Laf01] C. Laforest. A tricriteria approximation algorithm for steiner tree in graphs. Technical Report 69-2001, LaMI, 2001.

[Lap04] J.C. Laprie. Sûreté de fonctionnement des systèmes : concepts de base et terminologie. Technical Report 04520, LAAS, October 2004.

[Lee91] C.-Y. Lee. Parallel machines scheduling with nonsimultaneous machine available time. *Discrete Applied Mathematics*, 30(1) :53–61, January 1991.

[Leu04a] J. Y-T. Leung, editor. *Handbook of Scheduling*. CRC Press, Boca Raton, FL, USA, 2004.

[Leu04b] J. Y-T. Leung. Some basic scheduling algorithms. In *Handbook of Scheduling* [Leu04a], chapter 3.

[Lis06] B. Lisper. Trends in timing analysis. In *IFIP Working Conference on Distributed and Parallel Embedded Systems, DIPES'06*, pages 85–94, Braga, Portugal, October 2006. Springer.

[LL62] D. Lloyd and M. Lipow. *Reliability : Management, Methods, and Mathematics*, chapter 9. Prentice-Hall, 1962.

[LM99] F. T. Leighton and Y. Ma. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM J. Comput.*, 29(1) :258–273, 1999.

[LR05] A. Legrand and Y. Robert. *Algorithmique Parallèle*. Dunod, 2005.

[LR07] G. Lin and R. Rajaraman. Approximation algorithms for multiprocessor scheduling under uncertainty. In *SPAA '07 : Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 25–34, New York, NY, USA, 2007. ACM.

[LS95] J.K. Lenstra and D.B. Shmoys. *Scheduling Theory and its Applications*, chapter 1 - Computing Near-Optimal Schedules, pages 1–14. In Chretienne et al. [CCLL95], 1995.

[LST90] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46 :259–271, 1990.

[LSV06] A. Legrand, A. Su, and F. Vivien. Minimizing the stretch when scheduling flows of biological requests. In *Symposium on Parallelism in Algorithms and Architectures SPAA'2006*. ACM Press, 2006.

[LT07] A. Legrand and C. Touati. Non-cooperative scheduling of multiple bag-of-task applications. In *Proc of InfoCom 2007*, pages 427–435, 2007.

[LZS+06] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *DSN '06 : Proceedings of the International Conference on Dependable Systems and Networks*, pages 425–434, Washington, DC, USA, 2006. IEEE Computer Society.

[Mah04] A. Mahjoub. *Etude de la Robustesse et de la Flexibilité pour des problèmes d'ordonnancement et de localisation*. PhD thesis, INPG, Grenoble, July 2004.

[MS98] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Heterogeneous Computing Workshop*, pages 57–69, 1998.

[MW98] J. Mo and J. Warland. Fair end-to-end window-based congestion control. In *Proc. of SPIE '98 : International Symposium on Voice, Video and Data Communications*, 1998.

[MYCR05] L. Marchal, Y. Yang, H. Casanova, and Y. Robert. A realistic network/application model for scheduling divisible loads on large-scale platforms. In *Proc of IPDPS'05*, April 2005.

[OH96] H. Oh and S. Ha. A static scheduling heuristic for heterogeneous processors. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par, Vol. II*, volume 1124 of *Lecture Notes in Computer Science*, pages 573–577. Springer, 1996.

[OR94] M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. The MIT Press, 1994.

[OSM+04] A.J. Oliner, R.K. Sahoo, J.E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for bluegene/l systems. In *IPDPS'04*, 2004.

[PB00] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2/3) :115–128, 2000.

[Pea88] D. Powell and et al. The Delta-4 approach to dependability in open distributed systems. In *International Symposium on Fault-Tolerant Computing, FTCS-18*, pages 246–251, Tokyo, Japan, June 1988. IEEE.

[PPI07] P. Pop, K. Poulsen, and V. Izosimov. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *CODES-ISSS'07*, Salzburg, Austria, October 2007. ACM.

[PRT07] F. Pascual, K. Rzadca, and D. Trystram. Cooperation in multi-organization scheduling. In *Proc. of EuroPar 2007*, number 4641 in LNCS, pages 224–233, August 2007.

[PY00] C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In FOCS, editor, *41st Annual Symposium on Foundations of Computer Science*, pages 86–92, 2000.

[QJS02] X. Qin, H. Jiang, and D. R. Swanson. An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems. In *International Conference on Parallel Processing, ICPP'02*, pages 360–386, Vancouver, Canada, August 2002.

[RSBJ95] V. Rayward-Smith, F.N. Burton, and G.J. Janacek. Scheduling parallel program assuming preallocation. In Chretienne et al. [CCLL95], chapter 7, pages 146–165.

[RSTU02] A. Rasala, C. Stein, E. Torng, and P. Uthaisombut. Existence theorems, lower bounds and algorithms for scheduling to meet two objectives. In *SODA '02 : Proceedings of the*

thirteenth annual ACM-SIAM symposium on Discrete algorithms, pages 723–731, 2002.

[Sch90] F.B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys*, 22(4) :299–319, December 1990.

[SJ99] S. Srinivasan and N. K. Jha. Safety and reliability driven task allocation in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(3) :238–251, March 1999.

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

[SL93] G.C. Sih and E.A Lee. A compile-time scheduling heuristic for interconnection-constraint heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4 :175–187, February 1993.

[SPH+05] J. Souyris, E.L. Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *International Workshop on Worst-case Execution Time, WCET'05*, pages 21–24, Mallorca, Spain, July 2005.

[SS98] D.P. Siewiorek and R.S. Swarz. *Reliable Computer Systems, Design and Evaluation.* A.K. Peters, third edition, 1998.

[ST93] D. B. Shmoys and E. Tardos. Scheduling unrelated machines with costs. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms*, pages 448–454, 1993.

[Ste86] R. Steueur. *Multiple Criteria optimization : theory, computation and application.* John Wiley, 1986.

[Str80] W. Stromquist. How to cut a cake fairly. *The American Mathematical Monthly*, 87(8) :640–644, 1980.

[SW92] S. Shatz and J.P. Wang. Task allocation for maximizing reliability of distribued computer systems. *IEEE Transactions on Computers*, 41(9) :1156–1169, September 1992.

[SW97] C. Stein and J. Wein. On the existence of schedules that are near-optimal for both makespan and total weighted completion time. *Operational research letters*, 21(3) :115–122, October 1997.

[SWW95] D. Shmoys, J. Wein, and D. Williamson. Scheduling parallel machine on-line. *SIAM Journal on Computing*, 24(6) :1313–1331, 1995.

[TB07] V. T'kindt and J.-C. Billaut. *Multicriteria Scheduling*. Springer, 2007.

[TBE07] V. T'kindt, K. Bouibede-Hocine, and C. Esswein. Counting and enumeration complexity with application to multicriteria scheduling. *Annals of Operations Research*, April 2007.

[TFW00] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separate cache and path analyses. *Real-Time Systems*, 18(2/3) :157–179, May 2000.

[THW99] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. *8th IEEE Heterogeneous Computing Workshop (HCW'99)*, pages 3–14, April 1999.

[Top02] H. Topcuoglu. Performance-effective and low-complexity task scheduling for heterogeneous scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 13(3) :260–274, March 2002.

[Tse07] S. Tse. Online bicriteria load balancing for distributed file servers. In *Proc. Of CHINACOM'07*, pages 218–222, August 2007.

[Voo03] M. Voorneveld. Characterization of Pareto dominance. *Operations Research Letters*, 31(1) :7–11, January 2003.

[WWWMM01] M. P. Wellman, W. E. Walsh, P. R. Wurman, and J. K. MacKie-Mason. Auction protocols for decentralized scheduling. *Games and Economic Behavior*, 35(1-2) :273–303, April 2001.

[ZS03] H. Zhao and R. Sakellariou. An experimental investigation into the rand function of the heterogeneous earliest finish time scheduling algorithm. *In Proc. of EuroPar 03. LNCS 2790*, pages 189–194, 2003.

résumé :
L'informatique moderne n'est plus uniquement composée de machines personnelles et de super calculateurs. De nombreux supports de calcul sont maintenant disponibles et chacun pose des contraintes particulières amenant à de nombreux objectifs. Ainsi, la notion de performance d'une application est devenue multi-dimensionnelle. Par exemple, ordonnancer optimalement (en temps) une application sur une grille de calcul est inutile si elle ne fournit pas de résultat parce qu'une machine tombe en panne. Fournir une solution à ces problèmes est un défi algorithmique actuel.
Dans ce manuscrit, nous étudions l'ordonnancement multi-objectif à l'aide des outils de la théorie de l'approximation. Nous traitons ainsi quatre problèmes. Les deux premiers sont inspirés des systèmes embarqués, tandis que les deux derniers sont inspirés des problématiques que l'on retrouve sur les grilles et les *clusters*.
Le premier problème étudié est l'optimisation des performances d'une application sur une machine disposant de peu de mémoire de stockage. Nous montrons que l'utilisation de l'optimisation multi-objectif permet de fournir une solution et des informations sur le problème que la théorie mono-objectif de l'approximation ne pouvait pas obtenir.
Les deux problèmes suivants concernent l'optimisation des performances d'une application lorsque les machines ne sont pas entièrement fiables. Les différents modèles de défaillances amènent à des problèmes d'optimisation radicalement différents. C'est pourquoi le deuxième problème traite de la sûreté de fonctionnement des systèmes embarqués alors que le troisième considère la fiabilité des grilles et *clusters*.
Le dernier problème concerne l'utilisation simultanée d'une plate-forme de calcul parallèle par de nombreux utilisateurs. Nous montrons comment l'utilisation de l'optimisation multi-objectif peut permettre de prendre en compte les besoins utilisateurs au sein du processus d'optimisation.

abstract :
Modern computing systems are no longer composed of personal computers and super computers. Many computing platforms are now available and each one has its own constraints, leading to many different objectives. Thus, the notion of performances becomes multi-dimensional. For instance, scheduling optimally (for time) and application on a computational grid is useless if no results are given when a computer crashes. Solving these problems is a present algorithmical challenge.
In this manuscript, we study multi-objective scheduling through the approximation theory. We tackled four problems. The first two ones come from embedded systems while the last two ones are derived from cluster and grid computing.
The first studied problem is the optimization of performances of an application running on a machine with low storage capacity. We show that using multi-objective optimization leads to solutions and informations for this problem that could not have been obtained by the classical approximation theory.
The two following problems tackled the optimization of the performances of an application when computers are not entirely reliable. Different models can be studied leading to radically different optimization problems. That is why the second problem considers safety in embedded systems while the third one deals with the reliability in computational grids and clusters.
The last problem is about scheduling tasks of several users on a parallel computing platform. We show how multi-objective optimization can be used to take the users' needs into account during the scheduling process.