INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

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

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : « ***Informatique*** »

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

présentée et soutenue publiquement

par

Lionel Eyraud

le 16 Octobre 2006

Titre :

Théorie et pratique de l'ordonnancement d'applications sur les systèmes distribués

---

Directeur de thèse : Denis TRYSTRAM

JURY

M. PHILIPPE BAPTISTE,	Rapporteur
M. HENRI CASANOVA,	Rapporteur
M. JACQUES MOSSIÈRE,	Président
M. RAYMOND NAMYST,	Examinateur
M. YVES ROBERT,	Examinateur
M. DENIS TRYSTRAM,	Directeur de thèse

Table des matières

Table des figures

Introduction

Cette thèse parle d'ordonnancement. L'ordonnancement est un thème très vaste, qui regroupe selon les cas des problèmes d'emploi du temps dans des écoles, de planification de production dans des industries, de sélection de processus dans des systèmes d'exploitation informatique, et encore d'autres domaines d'applications variés. Dans le cadre de la théorie de l'ordonnancement, qui écarte le cadre des systèmes d'exploitation, on peut dire qu'il s'agit de l'allocation de ressources rares à diverses activités dans le but de minimiser une ou plusieurs mesures de performance. Les ressources et les activités peuvent prendre plusieurs formes ; dans cette thèse, il s'agira principalement de calcul informatique. Les ressources seront ainsi des ordinateurs, ou des processeurs, et les activités seront des programmes informatiques effectuant des calculs indéterminés.

Les travaux en ordonnancement ont d'abord été motivés par des applications industrielles, pour optimiser la gestion des lignes de production. Le développement du calcul parallèle a fait émerger un besoin de ce savoir-faire pour utiliser au mieux les différentes ressources d'un calculateur parallèle. Plus récemment, l'essor de l'informatique grand public a permis l'apparition de *grappes de calcul*, ou « *clusters* », qui sont des systèmes à mémoire distribués formés par l'interconnexion d'un grand nombre d'ordinateurs standard. Ces grappes ont des performances comparables à celles des grands calculateurs spécialisés, pour un coût beaucoup plus faible. Ces architectures nécessitent des logiciels et des techniques d'ordonnancement spécifiques pour une utilisation efficace, et le but de cette thèse est d'étudier des méthodes susceptibles de répondre à ces besoins.

De façon plus précise, nous nous intéressons à l'ordonnancement au niveau des *gestionnaires de ressources*. Ces logiciels sont chargés de répartir les ressources de calcul d'une grappe entre plusieurs applications. Bien que ces applications peuvent éventuellement aussi contenir du parallélisme, la gestion explicite de ce parallélisme n'est pas l'objet de cette thèse. C'est pourquoi nous travaillerons majoritairement dans le cadre du modèle des *Tâches Parallèles* (voir section 1.2.2), qui est le plus approprié pour décrire ce niveau de contrôle.

Vue d'ensemble de ce manuscrit

Ce document est construit selon une démarche d'aller-retour entre la théorie et la pratique, qui correspond au travail effectué pendant la thèse. Nous commençons donc, dans le chapitre 1, par énoncer précisément le contexte dans lequel s'est effectué ce travail : la description de l'architecture des grappes de calcul,

les différents modèles du calcul parallèle existants dans la littérature, et en particulier ceux qui ciblent ces architectures ; nous donnons également un aperçu des différents algorithmes et techniques classiques.

Nous nous intéresserons plus spécifiquement dans le chapitre 2 au problème de l'ordonnancement bicritère (makespan et somme pondérée des temps de complétion) de tâches modelables indépendantes. Nous y développons une approche théorique, en proposant un algorithme garanti sur les deux critères à la fois ; dans la suite du chapitre, nous présentons une simplification de cet algorithme qui réduit sa complexité, dans le but de pouvoir l'appliquer à des cas pratiques. Comme nous ne connaissons pas de bornes de garantie pour cet algorithme simplifié, nous montrons par une étude expérimentale qu'il obtient en moyenne de très bonnes performances, très stables sur les différentes instances testées, et équivalentes à celles de l'algorithme garanti.

Le chapitre 3 continue d'étudier le même problème, mais dans un cadre beaucoup plus pratique. Il s'agit d'appliquer les résultats décrits dans le chapitre précédent à un environnement de grappes de calcul réel, au sein du gestionnaire de ressource OAR. Nous montrons que malgré quelques différences profondes au niveau de la modélisation du problème, il est possible d'adapter l'algorithme simplifié pour ce logiciel. L'implémentation ainsi obtenue obtient de bons résultats en pratique sur des instances réelles tirées de traces d'utilisation de la grappe Icluster2 de Grenoble.

Dans le chapitre 4, nous revenons à une étude théorique pour analyser plus précisément une des différences de modélisation entre les chapitres 2 et 3 : la présence de réservations. Nous y proposons une modélisation pour l'ordonnancement de tâches parallèles en présence de contraintes d'indisponibilité des machines, ce qui n'a jamais vraiment été fait dans la littérature. Nous étudions également certains cas particuliers pour lesquels nous prouvons des garanties de performance pour les algorithmes de liste.

Contributions

Les contributions de cette thèse sont de plusieurs types. Le chapitre 1 contient des réflexions et un état de l'art sur l'ordonnancement sur grappes de calculs ; ces réflexions ont fait l'objet d'une publication dans l'« *International Journal of Foundations of Computer Science* » [18]. Il contient également la preuve d'une garantie pour les algorithmes de liste dans le cas des tâches rigides, qui est un peu plus précise que la garantie classique de 2, et qui est exactement égale à la borne inférieure. Dans le chapitre 2, on trouve une analyse théorique poussée d'un schéma d'algorithme existant dans la littérature ; grâce à l'introduction d'une procédure spécifique, nous obtenons des garanties meilleures que les bornes existantes. Ce chapitre contient également une étude expérimentale d'une variante plus simple de cet algorithme garanti, qui a été publiée dans la 16^{e} conférence SPAA [17].

Le chapitre 3 expose un travail d'implémentation de ces techniques, en mettant l'accent sur les aspects conceptuels et fondamentaux. Cela permet de faire ressortir les inadéquations ou insuffisances des modèles vis-à-vis d'un logiciel réel. Ces travaux sont en cours de publication dans l'« *International Journal of High*

Performance Computing and Applications » [20]. Le résultat de l'implémentation est de plus disponible dans la distribution de OAR.

Les travaux du chapitre 4 généralisent aux tâches parallèles les résultats précédents sur l'ordonnancement de liste en présence d'indisponibilité des machines. Nous y présentons également des bornes inférieures montrant que ces résultats sont d'une certaine manière les meilleurs possibles.

Chapitre 1

Ordonnancement sur grappes de PCs

Ce chapitre présente le contexte dans lequel s'est effectuée cette thèse. Nous allons tout d'abord introduire les principes architecturaux des *grappes de calcul*, qui forment l'essentiel des machines de calcul scientifique à travers le monde, ainsi que les logiciels et les approches utilisés pour gérer ces systèmes. Nous présentons dans un deuxième temps les différents modèles usuels introduits pour décrire le parallélisme d'un point de vue théorique, ainsi que les problèmes d'optimisation qui y sont liés. La suite de ce chapitre rassemble et analyse les différentes techniques algorithmiques utilisées dans la pratique des logiciels de gestion de ressources ou dans les études théoriques.

1.1 Les architectures du calcul parallèle

Au début des années 1990, l'essentiel des centres de calcul scientifique étaient équipés de machines parallèles monolithiques, dont le prix est élevé à cause de difficultés techniques de fabrication et de leur production en faible nombre. L'évolution rapide des performances des PCs « grand public » a fait évoluer le domaine du calcul parallèle, et ces machines ont progressivement été remplacées par des *grappes*, ou « *clusters* », qui sont formées de composants standard de l'industrie informatique et ont ainsi un coût d'achat bien moindre. Les chiffres du Top500 [67] le montrent bien : sur les 500 machines de calcul scientifique les plus performantes en Juin 2006, 364 sont des grappes vendues par IBM, HP ou Dell. Ces grappes représentent 72% du nombre de machines, mais seulement 50% de la performance totale : ces architectures sont donc moins performantes, mais bien plus répandues du fait de leur faible coût relatif.

Le principe architectural de ces systèmes distribués est assez simple : ils comptent entre plusieurs dizaines et quelques centaines d'ordinateurs dont l'architecture est standard et qui sont connectés par un réseau performant. Ces ordinateurs peuvent être de simples PCs de bureau (comme dans le cas de l'ancien i-cluster de Grenoble), ou bien des machines puissantes et récentes conçues spécifiquement pour le calcul haute performance. Avec cette définition, une salle informatique servant

aux travaux pratiques dans une université d'informatique peut être vue comme une grappe[1]. Cependant, les grappes de calcul sont le plus souvent des machines dédiées, et sont donc physiquement organisées de manière à optimiser la place occupée et le refroidissement des machines. De plus, l'exécution d'applications parallèles nécessite une interconnexion rapide entre ces ordinateurs que les cartes réseau standard ne permettent pas d'atteindre ; les ordinateurs composant ces grappes sont donc souvent équipés de cartes à haut débit, des cartes Ethernet GigaBit ou Myrinet par exemple. Ces ordinateurs, qui constituent l'élément atomique d'une grappe, sont appelés *nœuds* de calcul. Chaque nœud a ainsi une mémoire locale, un disque dur local, un ou plusieurs processeurs (mais rarement plus de deux), et une carte réseau.

Dans la classification des machines parallèles, on distingue les architectures à mémoire *partagée* de celles à mémoire *distribuée*. Quand la mémoire est partagée, tous les processeurs ont un accès direct à la mémoire, et peuvent lire et écrire à n'importe quel endroit. Les grappes sont des systèmes à mémoire distribuée, dans lesquels chaque nœud a une mémoire locale, et ne peut accéder aux informations contenues dans la mémoire des autres nœuds qu'en utilisant le réseau d'interconnexion pour envoyer un message. Ces architectures sont plus simples à fabriquer, mais plus complexes à programmer. En particulier, il est courant de considérer une machine à mémoire partagée comme une grande station de travail, sur laquelle on exécute plusieurs applications en même temps. Le système d'exploitation gère alors l'allocation des différentes applications sur les processeurs disponibles, et il est facile d'interrompre une application pour la reprendre plus tard sur d'autres processeurs : elle aura encore accès à ses données dans la mémoire centrale. Cette capacité de préemption et de migration est bien plus complexe à mettre en œuvre dans une architecture à mémoire distribuée comme une grappe de calcul.

La structure du réseau d'interconnexion est une caractéristique importante des machines à mémoire partagée. Aux débuts du parallélisme, les technologies ne permettaient de construire que des réseaux de faible taille. Il fallait donc organiser les nœuds selon des topologies spécifiques, et un message entre deux nœuds donnés transitait souvent par plusieurs autres nœuds avant d'arriver à destination. Les progrès technologiques permettent maintenant de construire des commutateurs qui relient directement entre eux un grand nombre de nœuds ; on parle donc de topologie en *clique* dans laquelle chaque nœud peut communiquer directement avec n'importe quel autre nœud. Le prix de ces commutateurs augmente cependant très vite lorsque l'on veut connecter plus de nœuds. Pour des machines de grande taille, plusieurs commutateurs d'interconnexion peuvent donc être nécessaires, créant ainsi un réseau hiérarchique moins régulier. Cependant, le routage n'est en tous cas jamais fait par les nœuds : un message traverse plusieurs commutateurs avant d'arriver à destination, mais ne transite jamais par un autre nœud. Comme les commutateurs ont de plus de très bonnes performances, on considère souvent en première approximation que ce réseau a une structure en clique, au moins d'un point de vue logique.

[1] Il y a d'ailleurs des logiciels dont le but est de permettre d'utiliser ces salles informatiques pour effectuer des calculs scientifiques pendant la nuit

L'administration d'une telle grappe est une tâche complexe. On ne peut parler d'une grappe comme une entité unique que si les nœuds sont fonctionnellement équivalents. Il faut donc installer, configurer et maintenir le même système d'exploitation sur tous les nœuds. Notons cependant que de la même façon que les composants matériels d'une grappe sont des composants standard, fabriqués en grande quantité, les systèmes d'exploitation déployés sur ces grappes sont le plus souvent très classiques, et sont souvent les mêmes que ceux qui sont utilisés par nos machines de bureau. Il existe quelques outils d'aide à l'administration qui permettent d'automatiser certaines parties du déploiement, mais dans toute grappe en utilisation, il y a régulièrement plusieurs nœuds hors service qui demandent une attention spécifique. De manière générale, on associe également aux nœuds de calcul proprement dits une ou plusieurs machines *frontales* qui servent de points d'entrée à la grappe, et à partir desquelles les utilisateurs peuvent réserver des machines de calcul. Il y a également toujours un ou plusieurs serveurs de fichiers qui regroupent les données des utilisateurs.

Pour exécuter un programme sur ces machines, la procédure standard nécessite d'une façon ou d'une autre une connexion à distance sur chacun des nœuds[2]. Sur un ordinateur classique, l'exécution d'une application passe par une phase de compilation, qui permet de transformer un programme écrit dans un langage quelconque en un fichier exécutable dans un format standard. De la même manière, les applications parallèles ne peuvent être exécutées par la grappe que sous la forme d'un fichier exécutable qui doit être exécuté localement par plusieurs nœuds.

Une application de bureautique, ou n'importe quelle application interactive, n'utilise souvent qu'une fraction des ressources d'une machine, c'est pourquoi il arrive fréquemment que l'on en exécute plusieurs en même temps sur le même ordinateur. En revanche, une application de calcul haute performance n'est pas interactive, et s'exécute donc aussi rapidement que le permettent les ressources disponibles sur les nœuds. C'est pourquoi on préfère n'exécuter qu'une seule application à la fois sur un nœud donné, et il est donc nécessaire de mettre en place un système de gestion de ces ressources pour allouer de manière exclusive les nœuds aux applications. De tels logiciels, appelés « *batch schedulers* », ont été développés à l'origine pour gérer les grosses machines monolithiques, qui présentent en ce domaine précis le même type de contrainte. La priorité était alors d'optimiser l'utilisation de la machine, dans un cadre où la demande est forte, et où un grand temps de réponse était admis (les utilisateurs n'ayant pas vraiment le choix). Le principe de base est simple, et est d'ailleurs toujours utilisé : l'utilisateur doit *soumettre* son calcul à un nœud de soumission, et le système l'exécute de façon automatisée lorsque les ressources nécessaires sont disponibles. Les résultats du calcul sont conservés sur le serveur de fichiers jusqu'à ce que l'utilisateur les efface.

Avec la baisse du prix des systèmes parallèles, on assiste à l'émergence de nouvelles attentes de la part des utilisateurs, qui sont souvent plus exigeants sur

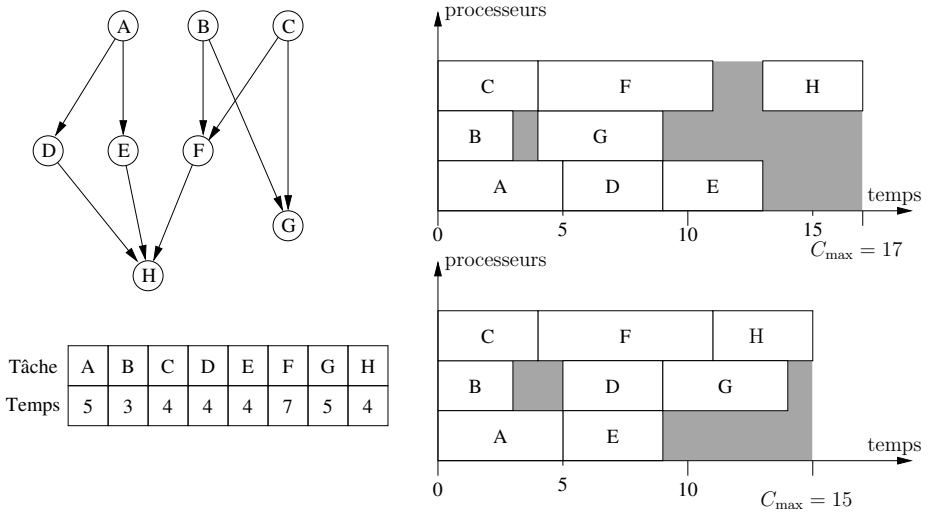
[2]La plupart des environnements de programmation parallèle, qui servent à faciliter l'écriture de programmes parallèles, contiennent cependant des outils pour automatiser ce processus de déploiement d'une application.

la réactivité du système. En particulier dans les communautés de développement informatique, il est apparu un besoin spécifique en travaux *interactifs* (par opposition aux travaux automatisés), qui permettent à l'utilisateur de suivre et de modifier le déroulement de son calcul. Cela lui permet de réagir à des évènements non prévus, dûs au fait que son logiciel se trouve dans une phase de test ou de prototypage. Contrairement à un système de production, ces travaux peuvent nécessiter une grande quantité de nœuds puisque les utilisateurs veulent souvent tester le passage à l'échelle de logiciels ou d'algorithmes. Ces nouvelles attentes ne changent pas réellement la modélisation de ces plates-formes de calcul, mais poussent à étudier une plus grande variété de critères d'optimisation. Une discussion sur ces critères est donnée dans la section 1.3.

Comme la demande en puissance de calcul continue d'augmenter, la prochaine étape naturelle consiste à relier plusieurs grappes, souvent réparties géographiquement à l'échelle régionale ou nationale, pour mettre en commun la puissance de calcul. Ces architectures, que l'on appelle parfois *grilles de calcul*, ont le double but de lisser la charge sur les différentes grappes qui les composent et de permettre aux utilisateurs d'effectuer des calculs plus importants (en terme de consommation mémoire et/ou de temps de calcul). Cependant, elles posent encore un certain nombre de problèmes techniques, dûs principalement à l'hétérogénéité du système ainsi formé :

- L'éloignement géographique des différentes grappes entraîne une hétérogénéité importante des liens de communications : on ne peut plus considérer que le système a une connexion en clique homogène. En particulier, la latence entre deux nœuds est très différente selon s'ils appartiennent à la même grappe ou pas. Les progrès effectués dans les réseaux à haut débit font que les différences de bande passante sont moins prononcées ; en revanche, il peut exister des phénomènes de *contention* des liens longue distance, et il est important de prendre ces phénomènes en compte.
- Comme chacune des grappes qui composent la grille a été achetée à des époques différentes, les matériels qui les composent sont différents, et il est délicat de maintenir une vision uniforme du calcul sur ce matériel hétérogène : il faut donc toujours un effort de l'utilisateur pour adapter son application aux différents environnements proposés.
- Comme chaque grappe appartient à des organismes administratifs différents, il est souvent nécessaire de spécifier et de mettre en œuvre une politique d'utilisation et de partage de ces ressources. La maintenance des machines est également répartie entre plusieurs centres administratifs, ce qui a tendance à augmenter sa qualité et sa réactivité par rapport à une maintenance centralisée ; cependant cela entraîne parfois (pour des raisons historiques ou du fait de contrats locaux) une hétérogénéité au niveau des choix techniques d'administration.

Dans cette thèse, nous nous intéresserons surtout au cas des grappes, tout en gardant à l'esprit cette évolution vers des systèmes à plus grande échelle et avec une grande répartition géographique.



Tâche	A	B	C	D	E	F	G	H
Temps	5	3	4	4	4	7	5	4

FIG. 1.1 – Exemple d'un graphe de tâches et de deux ordonnancements sur trois processeurs, représentés par des diagrammes temps × processeurs.

1.2 Modèles d'applications

Pour étudier ces plates-formes et les applications parallèles qui s'y exécutent, un certain nombre de modèles théoriques ont été développés. Nous allons présenter ici les modèles pour l'ordonnancement de ces applications, qui ont été définis afin d'optimiser l'arrangement de leur exécution, et obtenir ainsi la meilleure efficacité possible. Nous verrons également les notions classiques de l'ordonnancement, qui sont souvent valables dans tous les modèles.

1.2.1 Ordonnancement de graphes de tâches

En ordonnancement, le modèle du parallélisme le plus classique et le plus ancien est celui des *graphes de tâches*. Il part du principe que le parallélisme d'un calcul vient de ce que l'on a beaucoup de petits calculs atomiques à exécuter. L'approche consiste donc à diviser une application en *tâches*, qui par construction ont besoin pour s'exécuter d'un seul processeur, et prennent un certain temps que l'on suppose connu à l'avance. L'ordre d'exécution de ces tâches est contraint par un *graphe de précédence*, un graphe orienté acyclique qui spécifie ainsi un ordre partiel entre les tâches. L'application pourra ainsi être exécutée sur plusieurs processeurs si cet ordre permet à plusieurs de ces tâches séquentielles de s'exécuter en même temps. La figure 1.1 montre un exemple de graphe de tâches et deux ordonnancements possibles pour ce graphe. La représentation en diagramme temps × processeurs, dits *diagramme de Gantt*, est assez classique et est utilisée tout au long de cette thèse, avec toujours le temps en abscisse et les processeurs en ordonnée. Dans cette représentation, une tâche est un rectangle de largeur 1 et de longueur proportionnelle au temps d'exécution. Les zones grisées correspondent au temps d'inactivité, pendant lequel le nœud correspondant n'a rien à calculer.

Définition du modèle

Le problème correspondant à cette modélisation des applications, que l'on nomme dans la littérature *problème central de l'ordonnancement*, consiste à trouver la manière d'exécuter l'ensemble de ces tâches atomiques le plus efficacement possible sur un ensemble de m processeurs. Pour cela, on cherche un *ordonnancement*, c'est-à-dire une fonction σ qui donne la date de début de chacune des tâches, de sorte que les contraintes de ressources (ne pas utiliser plus de m processeurs) et de précédence (une tâche ne peut débuter avant que tous ses prédécesseurs aient terminé leur exécution) soient respectées. La performance d'un ordonnancement se mesure alors par le temps total nécessaire à exécuter toutes les tâches de l'application, c'est-à-dire le plus grand temps de complétion d'une tâche. Cette valeur, que l'on appelle le *makespan* de l'ordonnancement, est notée habituellement $C_{\max}$. Selon la notation standard en trois champs $\alpha \,|\, \beta \,|\, \gamma$ introduite par Graham [27], et que l'on peut retrouver dans [45], ce problème est noté par $P \,|\, \text{prec} \,|\, C_{\max}$.

Plus formellement, une instance de ce problème est composée d'un graphe orienté sans cycle $G = (V, E)$, d'un nombre m de processeurs et d'une pondération sur les nœuds du graphe, notée $p_i \in \mathbb{N}$ pour $i \in V$. Les arêtes dans le graphe correspondent aux contraintes de précédence : si $(i, j) \in E$ est une arête, cela signifie que la tâche j ne peut pas commencer son exécution tant que la tâche i n'est pas terminée. La pondération p_i représente quand à elle le temps d'exécution de la tâche i. Un ordonnancement est une fonction $\sigma : V \mapsto \mathbb{N}$, et σ_i est la date de début de la tâche i dans cet ordonnancement. Si l'on se fixe un ordonnancement, on peut alors définir pour chaque tâche $i \in V$ son *temps de complétion* selon l'ordonnancement σ par $C_i^\sigma \equiv \sigma_i + p_i$. Le *makespan* de cet ordonnancement est défini comme le plus grand de ces temps de complétion, $C_{\max}^\sigma \equiv \max_{i \in V} C_i^\sigma$ (pour simplifier les notations, on omettra l'indication de l'ordonnancement σ lorsqu'il n'y a pas de confusion possible). Le but du problème d'optimisation est de trouver une fonction d'ordonnancement σ qui soit *réalisable* et dont le makespan soit le plus petit possible. Une fonction est réalisable si elle vérifie les contraintes exposées plus haut, et que l'on peut formellement écrire de la façon suivante :

$$\forall t \in \mathbb{N}, \qquad |I_t| \leq m, \text{ où } I_t = \{i \in V \,|\, \sigma_i \leq t < \sigma_i + p_i\} \tag{1.1a}$$

$$\forall (i, j) \in E, \qquad \sigma_i + p_i \leq \sigma_j \tag{1.1b}$$

L'équation (1.1a) exprime la contrainte de ressources : à chaque instant t, l'ordonnancement ne peut pas utiliser plus de m nœuds. La deuxième équation (1.1b) formalise les contraintes de précédence.

Digression sur l'intégrité des valeurs : Nous pouvons faire tout de suite une remarque technique sur le problème tel que nous l'avons posé ci-dessus. En effet, nous avons défini toutes les valeurs temporelles comme éléments de $\mathbb{N}$, alors que l'on pourrait s'attendre à ce que le temps soit continu. Dans la théorie de la complexité et de l'algorithmique, il n'est pas possible de poser des problèmes qui travaillent sur des nombres réels : un ordinateur ne peut pas calculer à une précision infinie. De fait, en pratique, les temps que l'on manipule sont exprimés dans une certaine unité, par exemple en secondes dans le cas d'un gestionnaire de ressources. Il arrive cependant fréquemment (dans la littérature, et également dans cette thèse) que l'on parle de données qui ne sont pas

entières, comme par exemple « une tâche de longueur ϵ » pour $\epsilon \to 0$. Il faut alors comprendre que l'on peut toujours opérer un changement d'échelle, en multipliant toutes les valeurs par une même constante, afin de n'obtenir que des valeurs entières.

Solution optimale

Comme tous les problèmes que l'on va considérer au long de cette thèse, le but est ici de trouver, parmi un ensemble de solutions valides, une solution qui minimise un certain critère. La résolution *exacte* de ces problèmes consiste à trouver une solution avec la valeur la plus petite possible, c'est-à-dire telle qu'il n'existe aucune solution avec une valeur strictement plus petite du critère choisi. Une telle solution est dite *optimale* (notons qu'il peut très bien exister plusieurs solutions optimales différentes), et sa valeur sera notée tout au long de cette thèse par une étoile en exposant. Ainsi, si $C_{\max}$ est le makespan d'un ordonnancement, $C^*_{\max}$ est la valeur du makespan optimal.

Nous pouvons dès à présent introduire quelques notions importantes en ordonnancement, et qui vont être largement utilisées dans cette thèse. On peut ainsi définir le *travail* w_i d'une tâche comme étant la quantité de calcul qu'elle contient, du point de vue de l'algorithme d'ordonnancement. Pour une tâche séquentielle comme celles introduites précédemment, il s'agit simplement de son temps d'exécution. On peut alors définir le travail total W d'une instance : c'est simplement la somme des travaux de toutes les tâches. De façon évidente, pour ordonnancer toutes les tâches sur m processeurs, il va falloir un temps au moins égal à W/m. Cela donne donc une borne inférieure du makespan optimal : $C^*_{\max} \geq \frac{W}{m}$. Cette notion de travail, et la borne inférieure qui en découle, peut se généraliser à d'autres modèles de tâches, comme ceux que l'on va voir dans la section suivante.

Une autre borne inférieure peut être obtenue en calculant le plus long chemin dans le graphe de précédence. Comme toutes les tâches de ce chemin doivent être exécutées les unes après les autres, le makespan est au moins égale à la somme des temps d'exécution de ces tâches. Un tel chemin est appelé *chemin critique*, et on note souvent $l_{\max}$ sa longueur (que l'on appelle souvent aussi chemin critique par abus de langage). On a donc une deuxième borne inférieure : $C^*_{\max} \geq l_{\max}$.

Solution approchée

Cependant, trouver une solution qui atteint la valeur optimale du critère choisi s'avère souvent être un problème trop dur pour qu'il puisse être résolu en temps raisonnable. Techniquement, la plupart des problèmes d'ordonnancement appartiennent à la classe des problèmes NP-complets, dont on conjecture qu'il faut pour les résoudre un temps qui augmente de façon exponentielle avec la taille du problème, ce qui fait que l'on ne sait résoudre que des problèmes de taille relativement faible. Une approche classique est alors de chercher des algorithmes rapides (polynomiaux) qui fournissent une solution approchée, dont la valeur du critère n'est pas nécessairement la plus petite possible, mais n'est pas trop éloignée de la valeur optimale.

Pour pouvoir évaluer la performance de tels algorithmes sous-optimaux, on définit la notion de *garantie de performance* en faisant une analyse *au pire cas*

de la performance de l'algorithme : sa garantie est ainsi le plus mauvais rapport possible entre les valeurs approchée et optimale, qui est souvent atteint pour des instances très spécifiques pour lesquelles « tout se passe mal ». Cette définition se formalise classiquement de la façon suivante, comme on le trouve par exemple dans [29] :

Définition 1

Soit un problème $\mathcal{P}$ tel qu'à chaque instance I correspond un espace $\mathcal{S}(I)$ de solutions réalisables, et pour lequel on cherche $S \in \mathcal{S}(I)$ qui minimise une certaine fonction $f(S)$. Pour une instance I donnée, on note $f^*(I)$ la valeur minimale de cette fonction sur $\mathcal{S}(I)$.

Soit $\mathcal{A}$ un algorithme qui à partir d'une instance I produit une solution réalisable $\mathcal{A}(I)$. On dit que $\mathcal{A}$ est un algorithme de ρ-approximation *si, pour toute instance I, $f(\mathcal{A}(I)) \leq \rho \times f^*(I)$. La* garantie de performance *de $\mathcal{A}$ est la plus petite valeur de ρ tel que $\mathcal{A}$ soit une ρ-approximation.*

Il existe également des travaux sur l'étude des performances *moyennes* des algorithmes, qui se basent sur une mesure de probabilité sur les instances pour évaluer l'espérance du rapport de performance. Ces études sont plus délicates à mener, et nécessitent des hypothèses sur la répartition des probabilités des instances possibles. Dans cette thèse, nous nous intéresserons donc surtout à des études dans le pire des cas. Cependant, pour évaluer certains algorithmes que l'on n'arrive pas à prouver formellement, nous ferons également des études par simulation basées sur des modèles d'instances dérivés d'observations de systèmes réels.

Remarques finales

Ce problème central a été très largement étudié, et s'applique assez bien à l'optimisation de l'exécution d'une application parallèle. C'est un problème NP-difficile au sens fort [24], c'est-à-dire qu'il est très improbable que l'on trouve un jour un algorithme rapide pour le résoudre de façon exacte. Il existe de nombreux algorithmes d'approximation pour ce problème, dont le plus connu est certainement l'algorithme de « *List Scheduling* » imaginé par Graham en 1969 [26] et qui a une garantie de performance de 2.

De nombreuses extensions de ce modèle ont été proposées, pour prendre en compte par exemple les temps de communication (qui représentent un transfert de données entre différentes tâches), une éventuelle hétérogénéité des processeurs (certains calculant plus vite que d'autres), ou encore la possibilité de *préempter* une tâche (c'est-à-dire d'arrêter son exécution pour la reprendre plus tard). Ces extensions sont traitées en particulier dans [9]. On peut également trouver dans un article de Schuurman et Woeginger [61] une liste de dix questions ouvertes autour de ce problème et de ses extensions, la première étant de trouver un algorithme polynomial pour $P \,|\, \text{prec} \,|\, C_{\max}$ avec une garantie de performance meilleure que la borne de 2 de l'algorithme de liste de Graham.

D'un point de vue pratique, cette vision d'une application comme un ensemble de petits calculs dépendants donne d'assez bons résultats. Plusieurs environne-

ments de programmation, comme par exemple Kaapi [33] ou Cilk [57], permettent d'écrire des programmes parallèles en les décrivant de cette façon. Si l'on se limite à des applications dont le graphe de tâches ne dépend pas des paramètres d'entrée, on peut prévoir l'ordonnancement de façon statique, avant l'exécution du programme ; cela se fait dans certains compilateurs. Mais de façon générale, les tâches à exécuter « apparaissent » au fur et à mesure du déroulement du programme (on rejoint un modèle en-ligne, voir la discussion dans la section 1.2.4), et il faut un algorithme d'ordonnancement dynamique. La technique du *vol de travail* est couramment utilisée dans ce contexte : dès qu'un nœud est inactif, il essaie de contacter un nœud actif pour partager la charge de travail. Cela permet d'atteindre de bonnes performances en moyenne, tout en simplifiant l'écriture de programmes parallèles, sans avoir à gérer les communications et synchronisations de manière explicite.

1.2.2 Ordonnancer plusieurs applications

Dans le cadre qui nous intéresse ici, c'est-à-dire celui de la gestion des ressources pour une machine parallèle, il y a en fait plusieurs applications *différentes* entre lesquelles il faut partager les ressources. Il y a donc deux sources de parallélisme : on a plusieurs calculs indépendants, qui peuvent donc s'exécuter chacun sur un nœud différent ; de plus, chacun de ces calculs peut lui aussi être exécuté en parallèle (par exemple en appliquant les techniques de la section précédente). D'un point de vue théorique, on pourrait vouloir étendre la vision précédente et considérer l'ensemble de ces applications comme un graphe de petites tâches séquentielles, afin de reprendre les études effectuées dans le cadre précédent pour utiliser la plate-forme de façon très efficace. Cependant, il y a plusieurs raisons pour lesquelles cette vision n'est pas adaptée à ce cadre :

- Comme cela a été indiqué plus haut, pour des raisons techniques, les applications apparaissent au système de gestion de ressources comme un programme qu'il doit exécuter sur un certain nombre de nœuds. Le système n'a alors aucun moyen d'accéder à une représentation de ce calcul sous forme d'un graphe de tâches séquentielles. Pour que cela soit possible, il faudrait contraindre tous les utilisateurs à utiliser le même environnement de programmation parallèle, qui serait alors fortement couplé avec le système de gestion de ressources. Cette contrainte interdirait alors de réutiliser une application développée dans un autre cadre ; or il arrive fréquemment que les applications de calcul scientifique soient en développement depuis une dizaine d'années, et contiennent des codes propriétaires que l'on ne peut pas modifier.
- Le problème $P|\text{prec}|C_{\max}$ est déjà un problème difficile, même lorsque l'on considère une seule application. Même si, d'un point de vue théorique, le graphe de tâches de plusieurs applications est toujours un graphe de tâches et rentre donc dans le cadre précédent, cela revient à considérer des graphes bien plus gros que pour une seule application. Un système qui implémenterait cette vision aurait à gérer de grandes quantités de données et ne passerait certainement pas bien à l'échelle.

– Optimiser le critère $C_{\max}$ revient à considérer toutes les tâches séquentielles comme faisant partie d'un même calcul, et à vouloir terminer ce calcul le plus tôt possible. Dans le cadre de plusieurs applications, cette vision n'est plus appropriée. Vouloir terminer tout un groupe de calculs indépendants le plus vite possible revient à essayer d'obtenir la meilleure utilisation possible de la machine, ce qui est une attente sensée quand la puissance de calcul est rare ; mais il est intéressant également d'essayer d'augmenter la réactivité du système, en exécutant par exemple plus tôt les plus petits calculs.

Tâches parallèles

L'étude des systèmes de gestion de ressources nécessite donc une granularité moins fine. Ces systèmes ont en effet uniquement accès aux applications, et pas aux petites tâches séquentielles qui les composent. C'est pour cela qu'a été introduit le modèle des *tâches parallèles*, dans lequel l'unité atomique de calcul est au niveau de l'application. Ces tâches parallèles peuvent ainsi nécessiter plus d'un processeur pour s'exécuter. La façon dont elles utilisent les processeurs qui leur sont alloués n'est pas considérée par ce modèle ; l'ordonnanceur n'a aucun contrôle dessus. Comme ces tâches représentent un calcul complet, on suppose souvent dans ce modèle qu'elles sont indépendantes, c'est-à-dire qu'on peut les exécuter dans un ordre quelconque.

Ce modèle des tâches parallèles se raffine en trois grandes versions, suivant le niveau de contrôle qui est donné à l'algorithme d'ordonnancement [21] (voir la figure 1.2) :

– Dans le modèle des tâches *rigides*, il est spécifié dans l'instance le nombre de processeurs dont chaque application a besoin, ainsi que son temps d'exécution. L'application utilise alors tous ces processeurs pendant toute la durée de son exécution (elles sont représentées par des « rectangles » sur un diagramme temps × processeurs).
– Dans le modèle des tâches *modelables*[3], le nombre de processeurs qu'utilise une application n'est plus fixé : l'instance spécifie, pour chaque tâche, quel sera son temps d'exécution sur chaque nombre de processeurs possible, et l'algorithme d'ordonnancement peut choisir combien de processeurs allouer à chaque tâche. Cependant, une application donnée utilise le même nombre de processeurs tout au long de son exécution ; elle est donc toujours représentée par un rectangle dans un diagramme de Gantt, mais la forme de ce rectangle n'est pas fixé à l'avance.
– Dans le modèle des tâches *malléables*, cette dernière contrainte est supprimée : l'algorithme d'ordonnancement peut modifier le nombre de processeurs alloués à une application au cours de son exécution.

Un tour d'horizon des travaux sur les tâches parallèles peut être trouvé dans [16], qui se concentre sur des algorithmes polynomiaux pour les cas particuliers des pro-

[3]Cette appellation n'est pas uniforme. Les premiers travaux parlaient de « *parallelizable tasks* » ; le terme anglais largement accepté est maintenant « *moldable* ». Le terme français « malléable » a été utilisé pour désigner ce type de tâches, mais la confusion avec le terme anglais « *malleable* » qui désigne le troisième type de tâche a poussé à plutôt utiliser « modelable ».

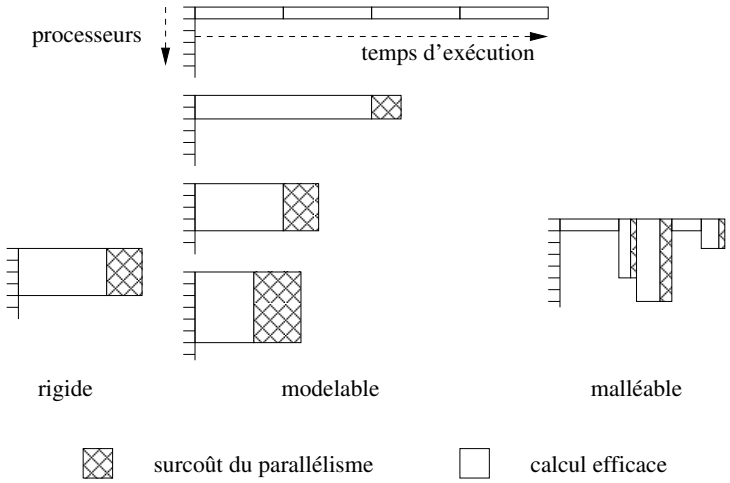


FIG. 1.2 – Les différents types de tâches parallèles

blèmes qui ne sont pas NP-difficiles, comme par exemple le cas préemptif, ou avec des temps d'exécution unitaires, lorsque le nombre de machines est fixé. On peut également trouver dans [19] un survol des algorithmes d'approximations pour les tâches parallèles.

À titre d'exemple, je donne ici la description formelle du problème d'ordonnancement dans le modèle rigide, que l'on note classiquement $P \,|\, \text{size}_j \,|\, C_{\max}$ si l'on veut minimiser le makespan :

Problème 1 (RIGIDSCHEDULING)

Instance : *Un entier m représentant le nombre de processeurs, et n tâches caractérisées par une durée d'exécution $p_i \in \mathbb{N}^*$ et un nombre de processeurs requis $q_i \in [1..m]$, pour chaque $1 \leq i \leq n$.*

Solution : *Une fonction d'ordonnancement $\sigma : [1..n] \mapsto \mathbb{N}$, telle que :*

$$\forall t \in \mathbb{N}, \quad \sum_{i \in I_t} q_i \leq m \quad \text{où} \quad I_t = \{i \in [1..n] \,|\, \sigma_i \leq t < \sigma_i + p_i\}$$

Pour ces problèmes, on peut considérer plusieurs critères d'optimisation différents. Nous reviendrons sur ce problème du choix du critère d'optimisation un peu plus loin, dans la section 1.3.

Bornes inférieures

Les bornes inférieures présentées pour le problème $P \,|\, \text{prec} \,|\, C_{\max}$ peuvent être facilement étendues à ces modèles de tâches parallèles. On peut ainsi définir le travail d'une tâche rigide comme le produit du nombre de processeurs requis par sa durée d'exécution : $w_i \equiv p_i q_i$. Cette valeur représente donc la surface de la tâche sur un diagramme temps $\times$ processeurs, et est d'ailleurs souvent également

appelée *surface*. Des arguments de surface permettent donc d'obtenir avec cette définition la même borne inférieure pour le makespan : $C^*_{\max} \geq \frac{W}{m}$. La notion de chemin critique n'a pas réellement de sens avec des tâches indépendantes, mais la borne inférieure correspondante est $p_{\max}$, la durée de la tâche la plus longue. Cette dernière borne inférieure est utile pour les instances avec peu de tâches, dans lesquelles la borne donnée par le travail est très en dessous de la valeur optimale du makespan.

Lien avec la géométrie

Le problème d'ordonnancement de tâches rigides est très lié à un autre problème, d'ordre plus géométrique, classiquement appelé StripPacking. Ce modèle a été introduit à l'origine pour trouver des découpes efficaces de pièces dans un matériau. L'énoncé du problème est le suivant : étant donnée une bande de largeur fixe (souvent fixée égale à 1) et de longueur infinie, étant donnés n rectangles dont les longueurs et les largeurs sont comprises entre 0 et 1, il faut trouver une découpe de ces n rectangles dans la bande de manière à minimiser le gaspillage, c'est-à-dire minimiser la longueur utilisée. On n'autorise pas la rotation des rectangles : leurs côtés doivent être parallèles à ceux de la bande. Dans les applications de découpes de pièces, cette contrainte apparaît souvent à cause de motifs du tissu ou des veines du bois.

Cet énoncé, qui ressemble grandement à celui de l'ordonnancement de tâches rigides pour minimiser le makespan, ajoute cependant une contrainte supplémentaire. Il est en effet imposé que le matériau utilisé pour découper un rectangle soit d'une seule pièce. En revanche, le problème des tâches rigides comme il a été défini plus haut n'impose aucune contiguïté sur les ressources alloués à une tâche, seulement une contrainte sur le *nombre* de ressources utilisées à un instant donné. Un ordonnancement valide pour RigidScheduling peut donc ne pas être une découpe valide pour StripPacking. On peut d'ailleurs montrer [19] qu'il existe des données en entrée (largeurs et longueurs des rectangles et des tâches) pour lesquelles la valeur du makespan optimale pour RigidScheduling est plus faible que la longueur optimale pour StripPacking.

Il existe beaucoup de travaux sur ce problème, en particulier un certain nombre d'algorithmes d'approximation ont été proposés. Une approche classique est de séparer les rectangles entre ceux qui sont « fins et longs » et ceux qui sont « larges et courts », chaque groupe étant alors composé de rectangles qu'il est facile d'agencer côte à côte. À titre d'exemples, Steinberg [66] a ainsi pu obtenir un algorithme avec une garantie de performance de 2, et Kenyon et Rémila [35] ont développé un algorithme avec une garantie asymptotique (c'est-à-dire valable à la limite, quand la longueur optimale tend vers l'infini) de $1 + \epsilon$ pour tout $\epsilon > 0$.

Les preuves de ces bornes de garanties sont souvent faites en se comparant à la surface totale de l'instance, et sont donc également valables pour l'ordonnancement rigide. Cependant, on connaît pour ce cas des algorithmes plus simples avec des performances similaires. On peut en revanche utiliser ces résultats de StripPacking lorsque les contraintes architecturales font que les tâches doivent être exécutées sur des nœuds contigus, par exemple pour faire en sorte que les

communications effectuées par deux tâches différentes n'interfèrent pas.

Intérêt des tâches modelables

Les études théoriques montrent que le fait de laisser à l'algorithme d'ordonnancement la possibilité de choisir l'allocation des applications permet de résoudre plus facilement les problèmes, ou du moins d'obtenir de meilleures garanties de performance. C'est pourquoi le modèle des tâches modelables, bien que peu utilisé en pratique dans les systèmes de gestion de ressources, est étudié de manière relativement large. Un des buts de ces études est de convaincre les concepteurs de ces gestionnaires que le fait de permettre l'expression de tâches modelables permettrait une meilleure performance globale du système.

En effet, la plupart des environnements de programmation d'applications parallèles, de même que quasiment tous les algorithmes distribués, ne supposent pas que le nombre de processeurs est fixe et spécifié à l'avance. Il est tout à fait possible de programmer une application de sorte qu'elle s'adapte au nombre de processeurs disponibles au moment où elle est exécutée. Cependant, la plupart des systèmes de gestion de ressources ne permettent d'exprimer que des tâches rigides : un utilisateur demande l'accès à un certain nombre de nœuds pour un certain temps, et les utilise pour effectuer ses calculs. Dans la plupart des cas, il aurait tout aussi bien pu effectuer ses calculs avec un peu plus ou un peu moins de processeurs, mais le système de soumission et d'ordonnancement ne permet pas d'exprimer cela.

L'intérêt des tâches modelables est de permettre à l'algorithme d'ordonnancement d'adapter l'allocation des tâches à la charge de travail présente sur la machine. Lorsqu'il n'y a que peu de tâches à effectuer, on peut allouer un grand nombre de processeurs à chacune des tâches ; le surcoût dû au parallélisme augmente, mais comme le système n'est pas très chargé, cela est compensé par un meilleur temps de réponse. Par contre, lorsque le système est très chargé, on peut allouer très peu de processeurs à chaque tâche, et effectuer les calculs de manière très efficace. Ainsi la charge de travail diminue le plus vite possible.

En revanche, programmer une application capable de s'exécuter sur un nombre de processeurs variable au cours de la même exécution est techniquement et algorithmiquement beaucoup plus difficile, sauf pour certains calculs qui ont un parallélisme assez simple. Le modèle malléable est donc bien moins adapté à une utilisation pratique d'une manière générale ; il est cependant très intéressant dans des cadres plus spécifiques, comme on va le voir dans la section suivante.

1.2.3 Applications multi-paramétriques

De nombreux systèmes de gestion de ressources, qui ciblent des architectures plus distribuées comme les *grilles de calcul*, se restreignent à une classe d'applications moins générale. Il arrive en effet fréquemment que l'on puisse naturellement découper une application en un grand nombre de petits calculs séquentiels indépendants, par exemple lorsque l'application consiste en la répétition du même calcul sur un grand ensemble de paramètres, ce qui peut permettre d'étudier en simulation l'effet des paramètres sur un système donné. La caractéristique principale de

ce type d'applications est que ces tâches qui la composent sont *indépendantes* et *nombreuses*, donc petites par rapport à la durée totale de l'application. Leur ordonnancement se fait typiquement selon le paradigme *maître - esclave* : un nœud maître dirige les opérations, et envoie du travail à faire à un groupe de nœuds esclaves qui le traitent et renvoient le résultat.

Cette simplicité de mise en œuvre permet à ces systèmes d'être relativement robustes. En effet, on peut très facilement changer en cours d'exécution le nombre d'esclaves qui prennent part au calcul, il suffit pour cela de re-répartir le travail à effectuer. Et même si un nœud tombe en panne et qu'il faut refaire le calcul qu'il avait commencé, le gaspillage n'est pas trop grand puisque les tâches séquentielles élémentaires sont relativement petites. Enfin, les communications sont simples à gérer, puisqu'elles n'ont lieu qu'entre le maître et les esclaves, jamais entre deux esclaves. C'est la raison pour laquelle ces systèmes peuvent être utilisés sur des plate-formes très distribuées. Le premier exemple très connu est le système *Seti@Home* [64], qui permet à tout internaute de participer à un gigantesque calcul à l'échelle planétaire. Cet exemple a été suivi par plusieurs systèmes, comme par exemple BOINC [1] ou *Folding@Home* [22] ; plus localement, la communauté grenobloise de calcul scientifique CIMENT [11] a développé un système nommé CiGri [10] qui permet d'utiliser les nœuds inactifs des grappes de calcul grenoblois pour effectuer des calculs multi-paramétriques.

D'un point de vue théorique, la simplicité de ces applications permet de prendre en compte d'autres aspects de l'architecture, comme par exemple les délais de communication. Dans le modèle assez classique des *tâches divisibles* [6], on suppose que le calcul peut se diviser en un nombre arbitraire de parties, et que le temps nécessaire pour envoyer les données dépend linéairement de la quantité de calcul. On peut ainsi déterminer une répartition efficace des calculs entre tous les esclaves, pour une seule application. D'autres études se penchent sur l'ordonnancement global de plusieurs applications de ce type, et l'on peut alors considérer que ce modèle est un cas particulier du modèle des tâches malléables, puisque le nombre de processeurs alloués à chaque tâche peut être modifié au cours de son exécution ; il s'agit d'un cas particulier parce que l'on peut négliger les surcoûts du parallélisme. Ces études parviennent alors à minimiser des critères complexes comme le *ralentissement* moyen ou maximal (appelé « *stretch* » en anglais)[4] [41], même en considérant des plate-formes hétérogènes.

Ces modèles de tâches multi-paramétriques permettent de considérer des plates-formes plus complexes, mais la restriction sur les applications les rend très peu appropriés pour les grappes de PCs, qui sont des architectures plus simples sur lesquelles on peut exécuter une plus grande palette d'applications.

1.2.4 Dates d'arrivées et modèles en ligne

Une caractéristique importante des systèmes de gestion de ressources est le fait que leur fonctionnement est ancré dans le temps : les requêtes des utilisateurs arrivent au fur et à mesure, et toutes les applications ne sont donc pas disponibles au même instant pour être exécutées. Les modèles précédents ne prennent pas cela

[4] Voir la section 1.3 pour la définition de ces critères.

en compte, puisqu'ils impliquent l'existence d'une date « zéro » à laquelle toutes les tâches peuvent commencer leur exécution. C'est pourquoi on rajoute souvent aux modèles une information de « *release date* », ou date d'arrivée. On associe ainsi à chaque tâche une valeur $r_i \in \mathbb{N}$, qui indique la date à partir de laquelle la tâche i peut commencer son exécution. Cela rajoute une contrainte sur la fonction d'ordonnancement σ, qui doit vérifier $\forall i,\ \sigma_i \geq r_i$.

Il existe deux manières d'appréhender ces dates d'arrivées. Les modèles *hors ligne* (« *off-line* » en anglais) posent le problème de façon directe : l'algorithme prend en entrée un ensemble de tâches, avec une date d'arrivée en plus de leurs autres caractéristiques, et doit en déduire un ordonnancement valide et performant. Il peut donc anticiper pour adapter l'ordonnancement des quelques premières tâches à l'arrivée d'une tâche qui deviendra disponible plus tard. À l'inverse, dans un modèle *en ligne* (« *on-line* » en anglais), l'algorithme d'ordonnancement ne devient conscient de l'existence et des caractéristiques d'une tâche qu'après sa date d'arrivée : il doit prendre toutes les décisions précédentes sans cette information, et ne peut plus revenir dessus.

Les problèmes avec dates d'arrivée sont souvent plus durs que les modèles qui n'intègrent pas cette contrainte. C'est pourquoi nombre d'études sont faites sur le problème plus simple sans dates d'arrivée, avec l'espoir que les techniques qui permettent d'obtenir un ordonnancement plus compact et plus performant seront transférables à ces modèles plus réalistes. Dans cette optique, le résultat de Shmoys et al. [65] est intéressant : si on dispose d'un algorithme de ρ-approximation sur le makespan pour le problème sans dates d'arrivée, on peut l'utiliser pour obtenir une 2ρ-approximation pour le problème en ligne avec dates d'arrivée. Le principe est de grouper les tâches disponibles par *lots*, de les ordonnancer par l'algorithme hors-ligne, et d'attendre jusqu'à la fin de l'exécution de ce lot pour ordonnancer alors toutes les tâches qui sont arrivées pendant ce temps.

De façon similaire, les techniques exposées à la section 2.2 permettent, à partir d'un algorithme d'ordonnancement sans dates d'arrivée, d'obtenir un algorithme garanti sur deux critères à la fois pour le modèle avec dates d'arrivée.

Clairvoyance

Un autre aspect du caractère dynamique des systèmes de gestion de ressources est qu'il est en réalité très difficile de prévoir à l'avance le temps que va nécessiter chaque calcul. La plupart des modèles supposent qu'un temps de calcul est associé à chaque tâche, et les algorithmes peuvent alors utiliser ces informations pour produire des ordonnancements efficaces. Ces modèles sont alors dits *clairvoyants*, pour bien montrer qu'ils donnent accès à des informations futures sur le déroulement des tâches. Par opposition, de nombreuses études ont été menées sur le modèle *non clairvoyant*, introduit en 1994 [52], dans lequel on n'a aucune information sur les durées des tâches, et donc finalement aucun moyen de différencier deux tâches. Comme l'on peut s'y attendre, il est délicat de concevoir des algorithmes efficaces dans ce cadre, car l'on peut souvent montrer que ce manque d'informations empêche d'obtenir des garanties de performance intéressantes. C'est pourquoi la plupart de ces études se placent dans un cadre où la préemption est autorisée, ce

qui permet dans une certaine mesure de corriger les erreurs faites à cause de ce manque d'informations. Dans cette thèse, nous avons fait le choix de ne considérer que des modèles clairvoyants. En effet, dans les systèmes de gestion de ressources sur grappes que nous étudions, les utilisateurs fournissent une estimation, même vague, du temps de calcul des tâches que l'on peut donc utiliser ; de plus, ces systèmes ne fournissent généralement pas de support pour autoriser la préemption des tâches. Notons cependant que certains algorithmes que nous allons étudier, en particulier les algorithmes de liste décrits à la section 1.4.1, n'utilisent pas les informations de durée des tâches et sont donc non-clairvoyants.

1.2.5 Synthèse

Il existe de nombreux modèles pour traiter des problèmes d'ordonnancement, chacun ayant été introduit pour répondre à une situation particulière. Celui qui nous intéresse plus particulièrement dans cette thèse est celui des Tâches Parallèles, qui s'applique bien au cadre de l'ordonnancement sur grappes de PCs. Le cadre réel d'un logiciel d'ordonnancement correspondrait à un modèle en ligne non clairvoyant ; cependant cela conduit à des problèmes bien trop difficiles d'un point de vue théorique. Nous allons donc dans la suite étudier des problèmes clairvoyants et souvent hors ligne afin d'améliorer la compréhension de l'ordonnancement dans ce cadre.

1.3 Critères d'évaluation

Nous avons donné jusqu'ici les paramètres et les contraintes que l'on exprime pour décider quelles sont les solutions réalisables. Pour poser entièrement le problème d'optimisation, il faut également un moyen de départager deux ordonnancements valides pour savoir lequel est le plus performant, ou préférable à l'autre selon un certain critère. En fonction du choix de ce critère, qui dépend des pratiques d'utilisation et des objectifs visés pour un site de calcul, l'administrateur va choisir un algorithme d'ordonnancement adapté. Il n'y a probablement pas de critère d'optimisation qui soit approprié à toutes les situations ; le travail des théoriciens qui conçoivent et prouvent des algorithmes est de proposer des algorithmes pour une large palette de critères, pour que les personnes confrontées à ces problèmes d'ordonnancement puissent choisir ce qui convient le mieux à leur situation particulière.

D'un point de vue théorique, un critère est une fonction d'évaluation, qui associe une « note » à une solution réalisable. Cette évaluation se fait donc toujours *a posteriori*, une fois que l'ordonnancement est terminé. Dans la plupart des études, on ne considère qu'un seul critère à la fois : une fois le critère sélectionné, on conçoit un algorithme spécifique pour ce critère, et le cas échéant on arrive à prouver qu'il a de bonnes performances. Il est cependant possible que cet algorithme donne des résultats très mauvais sur d'autres critères ; la notion de *performance* est très liée au critère d'évaluation que l'on s'est fixé. Il existe également des études, comme celle présentée dans le chapitre 2, où l'on considère plusieurs critères à la fois et

où l'on essaie de produire des ordonnancements qui soient performants pour tous ces critères simultanément.

1.3.1 Critères pour les tâches parallèles

Dans cette section, nous allons exposer une liste des critères les plus couramment étudiés et les plus pertinents pour les problèmes que nous considérons. Pour donner un aperçu de la difficulté relative de chacun de ces critères, nous donnons les résultats connus de complexité dans le cas de tâches indépendantes sur une seule machine.

Makespan

Le makespan, ou temps de complétion maximal (noté $C_{\max}$), est historiquement le premier critère considéré en ordonnancement, et donc également le plus étudié. Il s'applique très bien aux cas où toutes les tâches font partie du même calcul, et que c'est la date à laquelle le calcul entier est terminé qui est intéressante. Cependant, même dans le cas contraire, minimiser le makespan revient à maximiser l'utilisation de la machine (puisque cela permet d'effectuer une quantité de travail donnée en un temps le plus court possible), et peut donc être également intéressant.

Par contre, le makespan n'est pas forcément très approprié dans un cadre en ligne avec dates d'arrivée, puisque c'est probablement les dernières dates d'arrivée qui vont contraindre le plus le makespan, et que le comportement de l'algorithme avant cet instant n'est pas pris en compte dans l'évaluation.

Sur une seule machine, n'importe quel ordonnancement qui ne laisse pas la machine inutilisée alors qu'une tâche est prête à être exécutée est optimal pour le makespan.

Moyenne des temps de complétion

Ce critère, classiquement noté[5] $\sum C_i$, est plutôt orienté vers les utilisateurs, dans un cadre où chaque tâche appartient à un utilisateur différent : le principe est alors d'essayer de satisfaire les utilisateurs en moyenne. On peut également considérer une version pondérée de ce critère ; dans ce cas, l'instance contient également un poids ω_i quelconque pour chaque tâche, et le but est de minimiser $\sum \omega_i C_i$. Cette pondération permet ainsi d'exprimer des contraintes de priorité, qui sont quelquefois présentes dans les politiques d'utilisation exprimées par les utilisateurs des systèmes parallèles.

Sur une seule machine, sans dates d'arrivée, un résultat bien connu (voir [46] par exemple) est que l'on minimise $\sum \omega_i C_i$ en ordonnant les tâches par ordre de ω_i/p_i décroissant ; on appelle l'algorithme correspondant **WSPTF**, pour « *Weighted Smallest Processing Time First* ». Avec dates d'arrivées, le problème de minimiser $\sum C_i$ est NP-difficile au sens fort si l'on n'autorise pas la préemption [42]. Si la préemption est autorisée, ordonnancer toujours en premier la tâche à qui il

[5] Pour que ce soit réellement la moyenne, il faudrait multiplier par $\frac{1}{n}$, mais cela ne change qu'un facteur d'échelle ; ni le caractère optimal d'un ordonnancement ni les garanties de performance ne sont modifiées.

reste le moins de calcul à faire (cette heuristique est appelée **SRPT**, pour « *Shortest Remaining Processing Time* ») permet d'obtenir une solution optimale [2]. En revanche, même avec préemption, minimiser $\sum \omega_i C_i$ sur une seule machine avec dates d'arrivée est NP-difficile au sens fort également [37].

Moyenne des temps d'attente

Le temps d'attente d'une tâche (souvent appelé « *flow time* » en anglais) est défini dans un environnement avec dates d'arrivées de la manière suivante : $F_i \equiv C_i - r_i$. Il représente le temps qu'une tâche passe dans le système, entre le moment où elle est soumise et la fin de son exécution. La moyenne des temps d'attente est alors $\sum F_i$; ici aussi, on peut considérer la version pondérée $\sum \omega_i F_i$.

Ce critère est très similaire au précédent : comme les r_i sont des données du problème, les ordonnancement optimaux pour ces deux critères sont les mêmes. Cependant, les garanties de performance ne sont pas conservées d'un critère à l'autre : une garantie valable pour $\sum \omega_i F_i$ est également valable pour $\sum \omega_i C_i$, mais le contraire est faux. La moyenne des temps d'attente est donc un critère beaucoup plus difficile à traiter.

Comme les optimaux sont les mêmes, les résultats de complexité sur une machine pour $\sum \omega_i C_i$ s'appliquent également pour $\sum \omega_i F_i$.

Ralentissement moyen

Le ralentissement (« *stretch* » en anglais) d'une tâche est défini par $S_i \equiv \frac{C_i - r_i}{p_i}$. Le critère de ralentissement moyen est donc un cas particulier de la version pondérée du temps d'attente moyen. Cette notion été introduite pour refléter le point de vue des utilisateurs, qui considèrent souvent qu'il est plus raisonnable d'attendre longtemps pour exécuter une tâche longue qu'une tâche courte. Ce critère mesure ainsi le ralentissement que la tâche a subi du fait d'avoir dû partager la machine avec d'autres tâches.

Sur une seule machine, le problème de minimiser le ralentissement moyen pour des tâches indépendantes sans préemption est NP-complet [71]. En revanche, la complexité du problème sur une seule machine avec préemption est encore ouverte : on ne sait pas si minimiser le ralentissement moyen est un problème difficile ou pas, ce qui montre que ce critère est bien plus complexe à traiter que les autres. Par contre, **SRPT** est une heuristique avec une garantie de performance de 2 [55].

Maximum des temps d'attente

Les critères précédents qui s'intéressent à la moyenne des temps d'attente ou de complétion ont tendance à introduire de la *famine* pour les tâches les plus longues : elles peuvent être retardées indéfiniment par un flux de tâches courtes. En effet, pour tout algorithme garanti sur la moyenne des temps d'attente, il est possible de construire une instance dans laquelle une des tâches est sujette à la famine. Ce comportement, qui n'est pas souhaitable, amène à essayer de fournir des garanties sur le plus grand temps d'attente d'une tâche, et donc d'essayer de minimiser $\max_i F_i$. La mrearque précédente indique qu'il n'est pas possible d'obtenir un

algorithme avec une garantie à la fois sur la moyenne et sur le maximum des temps d'attente.

Sur une seule machine, ordonnancer les tâches par ordre d'arrivée dans le système (c'est-à-dire en utilisant l'algorithme **FCFS** (« *First Come First Served* », voir plus loin, section 1.4.1) permet d'être optimal pour $\max_i F_i$.

Ralentissement maximal

De la même façon, pour essayer d'éviter la famine, on peut préférer minimiser le plus grand ralentissement $\max S_i$.

Il existe un algorithme polynomial hors ligne pour minimiser ce critère sur une seule machine avec préemption [41], mais il est assez compliqué, avec une approche par programmation linéaire. Sans préemption, le problème est NP-difficile.

1.3.2 Critères dans d'autres modèles

Avec dates butoir

Dans certains modèles, on suppose que l'instance contient pour chaque tâche une *date butoir* d_i (en anglais « *deadline* » ou « *due date* » selon les cas), date à laquelle la tâche doit avoir terminé son exécution. Cette contrainte peut être dure, c'est-à-dire que l'on ne considère que des solutions dans lesquelles elle est vérifiée. Les dates butoirs sont dans ce cadre plutôt appelées des « *deadlines* », et le problème intéressant est souvent de décider s'il existe au moins une solution réalisable. En effet, en rajoutant une telle contrainte, il est facile de voir que l'on peut construire des instances qui n'ont aucune solution. On parle dans ce cas de problèmes de faisabilité. Certains modèles choisissent plutôt d'autoriser de rejeter des tâches, c'est-à-dire de ne pas les exécuter s'il est impossible de satisfaire leurs deadlines ; il est alors courant d'essayer de maximiser le nombre de tâches exécutées.

Si la contrainte de ces dates butoirs est faible, c'est-à-dire qu'une solution qui ne les respecte pas est acceptable, on parle de « *due date* ». Dans ce cas, on définit le *retard* d'une tâche comme étant 0 si elle est finie à l'heure (ou avant), et $C_i - d_i$ sinon. On note cette valeur $L_i \equiv \max_i (0, C_i - d_i)$. Il est courant dans ces modèles d'optimiser un critère en rapport avec ces dates butoirs. Par exemple, les critères souvent étudiés sont le retard maximal $L_{\max} \equiv \max_i L_i$, la somme des retards $\sum L_i$, et le nombre de tâches en retard.

Ces critères sont rarement étudiés dans le cadre des tâches parallèles, peut-être parce qu'il est rare que les systèmes de gestion de ressources permettent d'exprimer des dates butoir pour les tâches. En revanche, il existe de nombreuses études dans le cadre de systèmes de production industrielle, comme des machines-outils par exemple. Sur une seule machine, exécuter en priorité les tâches dont la date butoir est la plus proche permet souvent d'optimiser ces critères à base de dates butoir.

En régime continu

Une autre manière d'exprimer que l'on cherche à produire des ordonnancements qui conduisent à une forte utilisation de la machine est de considérer que chaque tâche est répétée un grand nombre de fois, et d'évaluer alors l'ordonnancement par le nombre de tâches effectuées par unité de temps. Ce critère s'appelle le *débit* (« *throughput* » en anglais), et suppose que l'on considère un régime permanent, c'est-à-dire que le nombre de répétitions des tâches est assez grand pour que l'on atteigne un ordonnancement périodique.

Ce modèle a été utilisé par Robert et al. [5] dans une série d'articles, pour modéliser en particulier des applications multi-paramétriques comme celles introduites dans la section 1.2.3 ou la diffusion de grandes quantités de données entre un serveur et plusieurs clients. La simplification introduite par l'hypothèse de régime permanent permet en effet de prendre en compte de façon fine les coûts de communication entre les nœuds, même sur une plate-forme non entièrement connectée, et de développer un algorithme polynomial qui atteint le débit optimal.

L'ordonnancement périodique a également été beaucoup étudié dans le cadre des compilateurs, pour optimiser les calculs effectués dans des boucles du programme ; cela s'appelle alors du *pipeline logiciel*. Les modèles développés dans cette optique intègrent cependant des relations de précédence entre différentes itérations de la boucle, ce qui rend le problème bien plus difficile.

1.4 Algorithmes et techniques standard

Cette section présente plusieurs techniques très classiques utilisées pour résoudre des problèmes d'ordonnancement. Nous commençons par exposer les principes et les résultats connus pour les algorithmes gloutons, qui sont très simples mais parfois très efficaces ; et nous verrons ensuite quelques approches plus complexes.

1.4.1 Algorithmes gloutons

Le principe d'un algorithme glouton est de prendre des décisions au fur et à mesure, sans jamais revenir sur les décisions précédentes. À chaque étape, toutes les décisions prises ont mené à un certain état, et la décision suivante est prise en s'adaptant à cet état, sans essayer de le changer. On résout ainsi le problème de façon incrémentale ; cela donne généralement des heuristiques qui s'exécutent très rapidement, mais l'ordonnancement qui en résulte n'est que rarement optimal. On peut cependant souvent prouver des garanties de performance pour ces algorithmes.

Dans le cadre qui nous intéresse, il y a deux manières classiques de concevoir un algorithme glouton, selon que les décisions soient basées sur les tâches ou sur les ressources. Pour des tâches indépendantes et séquentielles, les deux approches sont équivalentes ; mais dans le cas général elles sont très différentes.

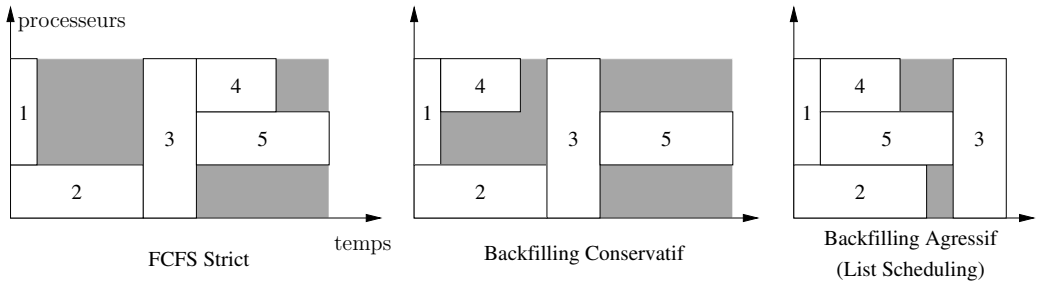


FIG. 1.3 – Effet des différentes variantes de backfilling. Les numéros des tâches correspondent à leur ordre dans la liste.

Décisions sur les tâches

Dans cette approche, la question à laquelle on doit répondre à chaque étape est la suivante : « Étant donnée une tâche, sur quelle ressource (ou ensemble de ressources) faut-il la placer ? ». Les algorithmes basés sur cette approche parcourent ainsi une liste de tâches dans l'ordre, et placent chacune des tâches le plus tôt possible étant données les décisions déjà prises pour les tâches précédentes. Dans la suite de cette thèse, les algorithmes basés sur ce principe seront dénotés par l'acronyme **FIFO**, pour « *First In First Out* », en précisant bien sûr quel est l'ordre de la liste utilisé.

Le représentant le plus connu de cette famille d'algorithmes est **FCFS** (pour « *First Come First Served* », Premier Arrivé Premier Servi), qui est un algorithme très couramment utilisé dans les systèmes de gestion de ressources. Pour **FCFS**, la liste est triée par ordre d'arrivée dans le système, ce qui fait qu'une tâche arrivée plus tôt sera bien plus prioritaire : les tâches suivantes seront ordonnancées de façon à ne pas la gêner. Les premières implémentations de **FCFS** étaient même plus strictes, car elles forçaient les tâches à être servies sur la machine dans l'ordre d'arrivée. Cela a pour effet qu'une tâche très « large » (qui demande beaucoup de processeurs et qui ne peut donc pas s'exécuter immédiatement) retarde toutes les tâches suivantes, même celles qui sont assez courtes pour terminer avant qu'elle ne commence.

La capacité à insérer des petites tâches dans les « trous » de l'ordonnancement a été appelée « *backfilling* » ; cela permet d'améliorer notablement l'utilisation de la machine, au prix d'une plus forte complexité algorithmique [3]. La figure 1.3 montre les différentes variantes de backfilling possible. La dernière variante correspond en fait à l'approche présentée plus bas, où les décisions portent sur les ressources.

HEFT [68], un autre algorithme classique, est également un représentant de cette famille. Il est conçu pour ordonnancer un graphe de tâches séquentielles sur des machines fortement hétérogènes. Son principe est de trier les tâches en fonction de leur position dans le graphe, et ensuite d'ordonnancer dans l'ordre de la liste chaque tâche de manière à ce qu'elle termine le plus tôt possible.

D'un point de vue théorique, l'algorithme **FCFS**, même avec backfilling, peut

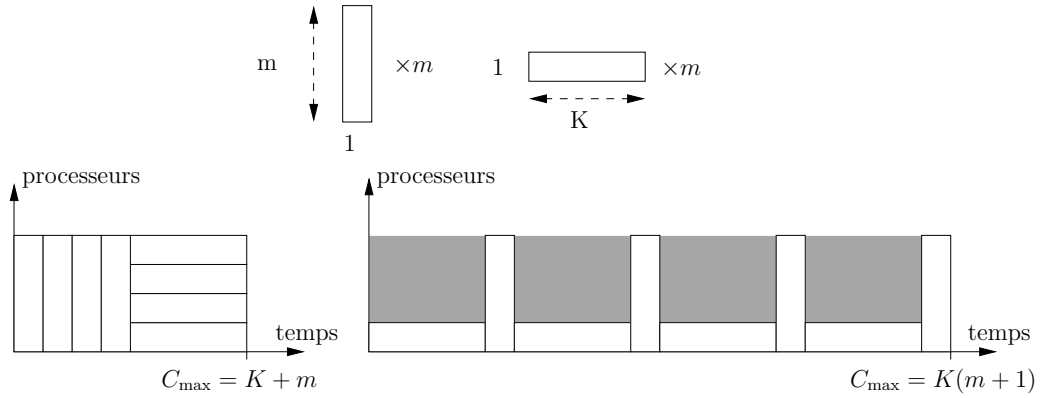


FIG. 1.4 – Pire cas pour l'algorithme **FCFS** : l'ordonnancement optimal à gauche, un ordonnancement possible pour **FCFS** à droite.

avoir des performances très mauvaises, en particulier sur le makespan. Le contre-exemple de la figure 1.4 montre que la garantie de performance est au moins égale à m sur les instances à m processeurs, et n'est donc pas bornée. Cette mauvaise utilisation entraîne également de mauvaises performances sur la plupart des autres critères standard. Comme on va le voir, cela vient principalement du fait que l'ordre des tâches dans la liste de **FCFS** n'a aucun rapport avec leurs dimensions.

En effet, quelques études ont montré de bonnes propriétés de l'algorithme **FIFO** dans certains cas, ou pour certains ordres de la liste dans les cas où il n'y a pas de dates d'arrivée. Schwiegelshohn et Yahyapour [63] ont ainsi montré, dans le cadre des tâches rigides, que lorsqu'aucune tâche n'utilise plus de $m/2$ nœuds, l'algorithme **FCFS** est une 3-approximation sur le makespan. Notons également que lorsque toutes les tâches utilisent strictement plus de $m/2$ processeurs, il est impossible d'ordonnancer deux tâches en même temps : on se retrouve alors dans un cas équivalent au cas d'une seule machine, et **FCFS** est optimal. Les cas difficiles pour **FCFS** sont donc des mélanges de tâches « fines » et « larges », comme le montre la figure 1.4.

Avec la même restriction sur les dimensions des tâches, mais en choisissant un ordre de la liste approprié, on peut également prouver des bornes de performance sur d'autres critères. Turek et al. ont par exemple étudié [69] le problème des tâches rigides et modelables pour le critère du temps de complétion moyen. Ils ont prouvé que si l'on trie les tâches par ordre croissant du travail[6] (donc en accordant une plus grand priorité aux tâches qui ont un petit travail), l'algorithme **FIFO** correspondant a une garantie de 2 sur la somme (non pondérée) $\sum C_i$.

Dans un autre article [70], et cette fois sans hypothèse sur les dimensions des tâches, Turek, Wolf et Yu ont étudié le critère du makespan, et ont montré que trier les tâches par p_i décroissant permet d'obtenir une 3-approximation sur le

[6]Rappelons que le travail w_i d'une tâche est défini comme le produit $p_i \times q_i$ du nombre de processeurs qu'elle requiert par son temps d'exécution. Il représente donc la quantité globale de calcul effectuée par cette tâche.

makespan pour l'algorithme **FIFO** ; ils ont également donné des contre-exemples montrant que l'on ne peut pas espérer prouver une meilleure borne si l'on n'autorise pas le backfilling.

Décisions sur les ressources

La deuxième approche pour concevoir un algorithme glouton est de centrer les décisions sur les ressources. Dans cette approche, à chaque étape l'algorithme doit répondre à la question : « Pour utiliser cette ressource (ou cet ensemble de ressources) libre, quelle tâche puis-je ordonnancer ? ». Ces algorithmes maintiennent également une liste, triée dans un certain ordre, mais ont pour objectif de ne jamais laisser une ressource inoccupée s'il y a une tâche prête à être exécutée dessus, même si cette tâche est très loin dans la liste de priorité. Cette famille d'algorithmes est connue sous le nom de *List Scheduling*, et a été étudiée pour la première fois par Graham [26], dans le cadre de l'ordonnancement d'un graphe de tâches séquentielles sur des machines homogènes. Dans cet article, Graham montre que l'algorithme de liste a une garantie de performance de $2 - \frac{1}{m}$ sur le makespan. Dans la suite, nous noterons **List** l'algorithme de liste général (c'est-à-dire avec un ordre de la liste quelconque).

Dans un autre article [25], Graham a également étudié l'algorithme de liste pour des tâches indépendantes, mais avec des contraintes de ressources. Dans ce cadre, il y a s types de ressources différents, et chaque tâche i nécessite une certaine quantité q_{il} de la ressource l. On peut alors montrer que l'algorithme **List** est une $s+1$-approximation sur le makespan. Dans le cas des tâches rigides, il n'y a qu'une seule ressource (les processeurs), donc $s = 1$, et l'on obtient alors une garantie de performance d'au plus 2. De plus, pour tout m, on peut construire une instance pour laquelle il existe un ordre de la liste qui produit un makespan $2 - \frac{1}{m}$ fois plus grand que le makespan optimal. Cette instance, assez classique, est représentée sur la figure 1.5 et comporte $m(m-1)+1$ tâches. Il y a $m(m-1)$ tâches de longueur $p = 1$ et de largeur $q = 1$, et 1 tâche de longueur $p = m$ et de largeur $q = 1$ (on peut déjà remarquer que ce contre-exemple est valable également pour les tâches séquentielles). L'ordonnancement optimal consiste à faire commencer la tâche a plus longue à l'instant 0, et l'on obtient un makespan de $C^*_{\max} = m$. Mais si l'algorithme **List** exécute en priorité les tâches les plus courtes, la tâche longue doit commencer à la date $m-1$, ce qui donne un makespan de $2m-1 = \left(2 - \frac{1}{m}\right) C^*_{\max}$.

Dans le cadre de cette thèse, nous avons prouvé que la garantie de performance de **List**[7] est en fait exactement de $2 - \frac{1}{m}$. Plus exactement, nous allons montrer que pour toute instance I, le makespan $C_{\max}$ de tout algorithme de liste vérifie :

$$C_{\max} \leq \left(2 - \frac{1}{m}\right) \max\left(p_{\max}, \frac{W}{m}\right)$$

où $p_{\max}$, la longueur de la plus grande tâche, et W/m, le travail total divisé par le nombre de ressources, sont deux bornes inférieures classiques du makespan optimal

[7] Il ne faut pas confondre cette garantie avec la précédente. La borne est la même, mais le cadre est un peu différent. En effet la première borne concerne le problème $P \,|\, \text{prec} \,|\, C_{\max}$, et celle-ci est valable pour le problème RigidScheduling, c'est-à-dire $P \,|\, \text{size}_j \,|\, C_{\max}$.

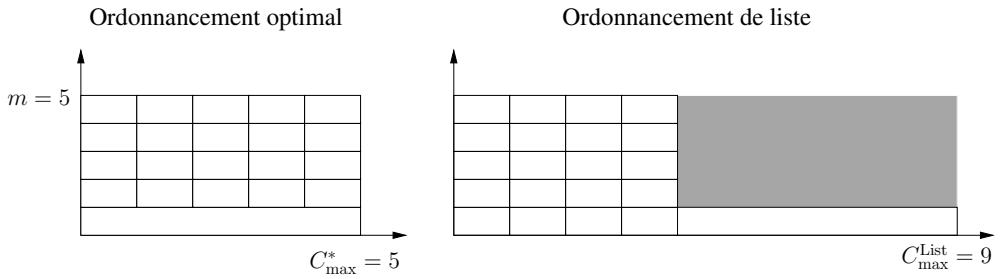


FIG. 1.5 – La garantie de performance de **List** est au moins de $2 - \frac{1}{m}$.

$C^*_{\max}$ (voir section 1.2.2).

Pour cela, nous commençons par prouver une propriété fondamentale de tout algorithme de liste. Étant donné un ordonnancement de liste σ, nous notons I_t l'ensemble des tâches en cours d'exécution au temps t : $I_t = \{i \in [1..n] \,|\, \sigma_i \leq t < \sigma_i + p_i\}$, et $r(t)$ le nombre de processeurs utilisés à l'instant t : $r(t) = \sum_{i \in I_t} q_i$.

Proposition 1

$$\forall t, t' \in [0,\ C^{\sigma}_{\max}[\ , \quad t' \geq t + p_{\max} \ \Rightarrow \ r(t) + r(t') \geq m + 1$$

Démonstration : Si $t' \geq t + p_{\max}$, alors il ne peut y avoir une même tâche en cours d'exécution à la fois au temps t et au temps t' : $I_t \cap I_{t'} = \emptyset$. Mais comme $t' \leq C^{\sigma}_{\max}$, il y a au moins une tâche, notons-la T_i, qui s'exécute au temps t'. Comme σ est un ordonnancement de liste, la tâche T_i n'a pas été exécutée au temps t parce qu'il était impossible de l'exécuter en même temps que les tâches de l'ensemble I_t. Comme les tâches sont indépendantes, cela ne peut venir que d'un manque de ressources, et l'on a nécessairement $r(t) + q_i > m$. On en déduit $r(t) + r(t') > m$, et comme $r(t)$ et $r(t')$ sont des entiers, il vient :

$$r(t) + r(t') \geq m + 1$$

■

Nous pouvons maintenant prouver le résultat principal :

Théorème 1

Pour toute instance I avec m processeurs,

$$C^{List}_{\max}(I) \leq \left(2 - \frac{1}{m}\right) \max\left(p_{\max}, \frac{W}{m}\right)$$

Démonstration : Si $C^{\text{List}}_{\max} \leq \left(2 - \frac{1}{m}\right) p_{\max}$, le résultat est vérifié. Supposons donc que $C^{\text{List}}_{\max} > \left(2 - \frac{1}{m}\right) p_{\max}$.

On a alors $p_{\max} < \frac{m}{2m-1} C^{\text{List}}_{\max}$, et le résultat précédent nous permet d'écrire :

$$\forall t \in \left[0,\ \frac{m-1}{2m-1}\right[, \qquad r\left(t\right) + r\left(t + \frac{m}{2m-1} C^{\text{List}}_{\max}\right) \geq m + 1$$

En intégrant cette relation, on obtient :

$$X \equiv \int_0^{\frac{m-1}{2m-1}C_{\max}^{\text{List}}} r(t) + r\left(t + \frac{m}{2m-1}C_{\max}^{\text{List}}\right) \mathrm{d}t \geq (m+1)\frac{m-1}{2m-1}C_{\max}^{\text{List}}$$

Quelques réarrangements permettent d'exprimer X en fonction du travail total :

$$\begin{aligned} X &= \int_0^{\frac{m-1}{2m-1}C_{\max}^{\text{List}}} r(t)\mathrm{d}t + \int_{\frac{m}{2m-1}C_{\max}^{\text{List}}}^{C_{\max}^{\text{List}}} r(t)\mathrm{d}t \\ &= \int_0^{C_{\max}^{\text{List}}} r(t)\mathrm{d}t - \int_{\frac{m-1}{2m-1}C_{\max}^{\text{List}}}^{\frac{m}{2m-1}C_{\max}^{\text{List}}} r(t)\mathrm{d}t \end{aligned}$$

et comme $r(t) \geq 1$ pour tout t,

$$\begin{aligned} &\leq \int_0^{C_{\max}^{\text{List}}} r(t)\mathrm{d}t - \frac{1}{2m-1}C_{\max}^{\text{List}} \\ &= W - \frac{1}{2m-1}C_{\max}^{\text{List}} \end{aligned}$$

On arrive donc au résultat suivant :

$$W \geq X + \frac{1}{2m-1}C_{\max}^{\text{List}} \geq \frac{(m+1)(m-1)+1}{2m-1}C_{\max}^{\text{List}}$$

On en déduit donc $W \geq m \times \frac{1}{2-\frac{1}{m}}C_{\max}^{\text{List}}$, d'où

$$C_{\max}^{\text{List}} \leq \left(2 - \frac{1}{m}\right)\frac{W}{m}$$

■

Lorsque l'on voit le contre-exemple de la figure 1.5, il paraît naturel d'essayer de prouver de meilleures performances de garantie pour des ordres bien précis de la liste. Il est en effet connu que pour les tâches séquentielles, le fait trier par p_i décroissant (l'algorithme correspondant est usuellement appelé **LPT**, pour « *Largest Processing Time* ») permet d'obtenir une garantie de $\frac{4}{3}$ sur le makespan. Il n'existe à ma connaissance aucune étude sur **LPT** dans le cadre de tâches parallèles. On peut cependant montrer, grâce à un contre-exemple venu de problèmes de BinPacking [32], que la garantie de performance de **LPT** est dans ce cadre au moins de $\frac{17}{10}$. Ce contre-exemple, donné sur la figure 1.6, est formé de tâches ayant toutes la même longueur ($p_i = 1$), et dans ce cas **LPT** se comporte exactement comme l'algorithme **FirstFit** classique pour BinPacking. Le pire cas arrive bien entendu lorsque **LPT** exécute les tâches par q_i croissant ; mais cela n'aiderait pas vraiment de spécifier un ordre sur les q_i en cas d'égalité des p_i. En effet, il est possible de rallonger d'une petite valeur ϵ les tâches fines pour que **LPT** n'ait pas de choix lors de l'ordonnancement du contre-exemple de la figure 1.6.

Avec d'aussi bonnes performances indépendantes de l'ordre de la liste, on pourrait être tenté d'utiliser l'algorithme **List** en triant la liste par ordre d'arrivée des

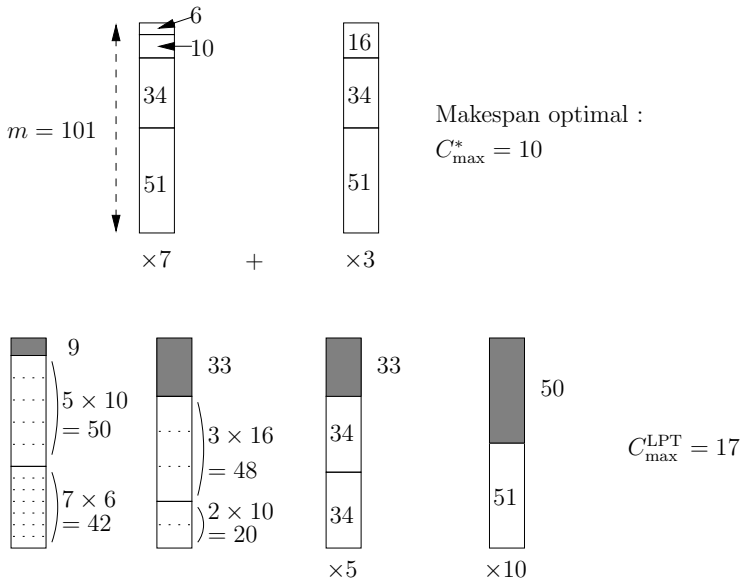


FIG. 1.6 – La garantie de performance de **LPT** est au moins de $\frac{17}{10}$.

tâches dans le système, afin d'obtenir à la fois une bonne utilisation et un bon temps de réponse. Cependant cet algorithme est presque « trop » indépendant de l'ordre de la liste, et a plutôt tendance à favoriser les tâches « fines », qui vont pouvoir utiliser les ressources plus tôt et ainsi passer devant des tâches qui attendent depuis plus longtemps. Il est ainsi possible de construire des instances où un flux continu de telles tâches empêche une tâche large de s'exécuter pour un temps arbitrairement long. On dit dans ce cas que cette tâche est en situation de *famine*. Le fait que la famine soit possible est la raison principale pour laquelle List Scheduling n'est pas utilisé dans des systèmes réels.

Approche hybride

Comme souvent, il est possible de définir un juste milieu entre ces deux approches. Un algorithme intermédiaire est par exemple l'algorithme *min-min*, qui est défini au départ dans le cadre d'ordonnancement de tâches séquentielles sur processeurs hétérogènes, mais qui pourrait s'étendre à des cadres plus simples. Le principe est ici de sélectionner à chaque étape le couple (tâche, ressource) dont la date de complétion est la plus faible parmi tous les couples possibles. On voit bien que la complexité de cet algorithme est plus grande que pour les approches précédentes, puisqu'il considère plus d'éventualités à chaque étape, mais cela doit lui permettre d'obtenir de meilleures performances. D'autres algorithmes ont également été développés avec cette approche (*max-min* et *Sufferage*). Ces heuristiques ne sont pas garanties théoriquement, car l'ordonnancement sur processeurs hétérogènes est un problème très difficile ; on peut cependant trouver dans [51] une étude expérimentale montrant les résultats obtenus.

1.4.2 Algorithmes par phases

De nombreux algorithmes d'ordonnancement sont basés sur un principe de décomposition en plusieurs phases du problème, c'est-à-dire en le découpant en plusieurs sous-problèmes plus simples et en considérant les sous-problèmes séparément. Je vais détailler ici quelques exemples de cette approche.

Tâches modelables

Dans le cadre des tâches modelables, le problème se décompose de manière naturelle en une phase d'*allocation*, où l'on décide combien de processeurs allouer à chaque tâche, et une phase d'*ordonnancement* proprement dit, où l'on décide l'ordre dans lequel on exécute chacune des tâches.

Un premier exemple d'ordonnancement par phases dans ce cadre est un algorithme de Turek et al. [70], qui commence par générer un ensemble d'allocations différentes dans lequel on peut prouver qu'au moins une a les propriétés voulues. L'algorithme consiste alors à obtenir un ordonnancement par *List Scheduling* à partir de chacune de ces allocations, et à garder le meilleur. Cet algorithme a une garantie de performance de 2 sur le makespan, et ne suppose pas que les tâches soient monotones[8].

Un autre algorithme a été développé dans le cas où l'hypothèse de monotonie est vérifié, et obtient alors une $\left(\frac{3}{2}+\epsilon\right)$-approximation [53]. Cet algorithme est en effet capable, étant donnée une valeur D du makespan, de fournir un ordonnancement de longueur $\frac{3}{2}D$ ou d'affirmer qu'il n'existe pas d'ordonnancement de longueur plus petite que D. Le principe est de déterminer l'allocation des tâches grâce à un algorithme de sac-à-dos, en séparant les tâches entre tâches parallèles rapides et les tâches moins parallèles et plus longues. Une fois cette séparation effectuée, l'ordonnancement se fait alors par étagères, en empilant les deux ensembles de tâches de manière intelligente.

Pour l'ordonnancement de tâches modelables avec contraintes de précédence, Lepère et al. [43] ont également conçu un algorithme en deux phases et obtenu une $\left(3+\sqrt{5}\right)$-approximation ; la garantie peut être améliorée à $\left((3+\sqrt{5})/2\right)$ lorsque les contraintes de précédence sont données par un graphe série-parallèle.

Clustering

Pour ordonnancer des graphes de tâches avec grands temps de communication[9], une approche par *clustering*, introduite dans [58], est souvent utilisée. Il s'agit de regrouper les tâches qui communiquent beaucoup pour les ordonnancer sur le même processeur. Pour cela, on résout (par une heuristique, car ce problème

[8] La définition formelle de la monotonie d'une tâche modelable est donnée dans la section 2.1 ; informellement, cela revient à dire que la « pénalité » due à la gestion du parallélisme augmente avec le nombre de processeurs.

[9] Dans le modèle délai classique, ces temps de communication retardent l'exécution d'une tâche si un de ses prédécesseurs s'exécute sur un autre processeur : il faut attendre que les informations transitent sur le réseau.

est toujours NP-complet) le problème d'ordonnancement sur une infinité de processeurs. Yang et Gerasoulis [72] ont proposé l'algorithme **DSC** pour traiter ce problème et ont montré à la fois son optimalité pour certaines classes de graphes de dépendance et ses bonnes performances sur des instances aléatoires.

Une fois ce regroupement fait, les temps de communication entre groupes sont moins importants par rapport à la taille des groupes, et on peut alors appliquer des techniques connues pour l'ordonnancement de graphes de tâches avec petits temps de communication. La deuxième phase consiste donc à fusionner les groupes de façon à « replier » l'ordonnancement obtenu sur une infinité de processeurs pour retrouver une solution réalisable pour le problème initial.

Techniques de relaxation

Une autre technique, assez générale, qui peut être vue comme une approche en deux phases, consiste à *relaxer* le problème traité pour revenir à un problème plus simple (souvent en enlevant une contrainte au problème). On peut ainsi résoudre le problème simple (parfois de façon optimale), et transformer la solution obtenue en une solution valide pour le problème initial, sans trop de perte de performance.

Un exemple classique est l'ajout de la préemption : beaucoup de problèmes d'ordonnancement sont plus simples à traiter quand on peut arrêter l'exécution d'une tâche pour la reprendre ensuite. Pour des tâches séquentielles indépendantes avec dates d'arrivée, Chakrabarti et al. [8] ont ainsi prouvé qu'on peut transformer n'importe quel ordonnancement préemptif en un ordonnancement non préemptif avec une perte de performance d'au plus 7/3 sur la moyenne des temps de complétion.

La relaxation est également utilisée avec des Programmes Linéaires en enlevant la contrainte d'intégrité des variables. Un des nombreux exemples est un article de Munier et König [54] qui étudie le problème d'ordonnancement d'un graphe de tâches avec communications et temps de calcul unitaire, sur une infinité de processeurs (problème $P_\infty \,|\, \text{prec}, p_j = 1, c_{ij} = 1 \,|\, C_{\max}$). Leur approche est donc de formuler le problème par un programme linéaire en nombres entiers, de résoudre la version relaxée (sans contrainte d'intégrité), puis d'utiliser cette solution pour reconstruire un ordonnancement valide pour le problème de départ. L'heuristique résultante a alors une performance de garantie de $\frac{4}{3}$.

1.5 Synthèse

Dans ce chapitre, nous avons présenté les approches et les modèles classiques pour l'ordonnancement sur grappes de calcul. Il apparaît que le modèle des Tâches Parallèles (rigides ou éventuellement modelables) est le plus approprié pour étudier ces problèmes. La question du critère d'optimisation à utiliser est par contre beaucoup moins claire ; le choix dépend en effet du contexte d'utilisation spécifique à chaque installation. Comme les auteurs de [36] le font remarquer, il n'y a certainement pas de critère qui soit approprié pour tous les cas. Dans la suite, nous allons nous intéresser au makespan et à la moyenne des temps de complétion, qui

sont deux critères généraux et assez antagonistes pour pouvoir couvrir une large gamme de cas d'utilisation pratiques.

Chapitre 2

Approche théorique de l'ordonnancement bicritère

Ce chapitre s'intéresse à la conception et la garantie d'un algorithme d'un point de vue théorique. Cette étude se place dans le cadre d'applications modelables indépendantes, dans l'espoir que de futurs systèmes d'ordonnancement supportent ce genre d'applications. Pour répondre aux besoins multiples et éventuellement contradictoires, nous avons décidé de concevoir un algorithme bicritère, c'est-à-dire qui cherche à optimiser simultanément deux critères différents, correspondant à deux visions différentes de la qualité d'un ordonnancement.

Après une définition formelle du problème, nous exposons un algorithme (appelé **Lots**) garanti sur les deux critères en nous appuyant sur quelques résultats précédents. Nous prouvons ainsi une garantie de 6 sur les deux critères ; dans un cadre hors-ligne, on peut améliorer la garantie sur le makespan à 3. En introduisant un paramètre aléatoire, on peut également obtenir une garantie en moyenne d'environ $4,1$ pour la somme des temps de complétion.

Nous nous intéressons ensuite à l'étude d'un algorithme basé sur le même principe, mais plus simple à mettre en œuvre et d'une complexité plus faible. La preuve de garantie ne pouvant s'étendre à cet algorithme, nous présentons des résultats de simulation en comparant ses performances avec des algorithmes existants ainsi qu'avec des bornes inférieures des valeurs optimales des deux critères.

2.1 Rappels et définition du modèle

2.1.1 Problème

Le problème d'ordonnancement de tâches modelables indépendantes a été introduit par Turek et al. [70]. Comme il a été exposé dans la section 1.2.2, ce modèle a pour but de séparer explicitement le parallélisme inter-application (qui vient du fait que l'on a plusieurs applications indépendantes) du parallélisme intra-application (puisque chaque calcul peut lui-même être exécuté en parallèle). Dans ce modèle, on suppose que chaque tâche peut être exécutée sur un nombre arbitraire de processeurs, mais que ce nombre doit rester fixe tout au long de son exé-

cution. La gestion du parallélisme intra-application n'est pas explicitement prise en compte ; on suppose seulement que le temps de calcul de chaque tâche dépend du nombre de processeurs qui lui ont été alloués selon une fonction propre à chaque tâche.

La notation à trois champs standard de ce problème est $P \,|\, \text{any} \,|\, C_{\max}$ (voir [16]). On peut le formuler de la façon suivante :

Problème 2 (MoldableScheduling**)**

Instance : *Un entier m représentant le nombre de processeurs, et un ensemble de n tâches T_i, chacune étant caractérisée par une fonction qui donne son temps de calcul en fonction du nombre de processeurs alloués : $p_i : [1..m] \mapsto \mathbb{N}$. Chaque tâche a également une priorité ω_i.*

Solution : *Une fonction d'*allocation *$\pi_i : [1..n] \mapsto [1..m]$ qui spécifie combien de processeurs sont alloués à la tâche i, et une fonction d'*ordonnancement *$\sigma_i : [1..n] \mapsto \mathbb{N}$ qui spécifie la date de début d'exécution de la tâche i. Une solution est réalisable si l'on peut respecter les dates de début d'exécution des tâches en utilisant au plus m processeurs :*

$$\forall t \in \mathbb{N},\ \sum_{i \in J_t} \pi_i \leq m \quad \text{où} \quad J_t = \{i | \sigma_i \leq t < \sigma_i + p_i(\pi_i)\}$$

J_t est l'ensemble des tâches qui sont en cours d'exécution au temps t.

Pour ce modèle, il est fréquent de faire une hypothèse additionnelle sur les fonctions p_i, qui exprime le comportement classique d'une application parallèle lorsque l'on change le nombre de processeurs sur lesquels elle est exécutée. Cette hypothèse, dite de *monotonie*, est en fait double. La première partie revient à supposer que le temps de calcul d'une application ne peut que diminuer quand le nombre de processeurs augmente. La deuxième partie exprime d'autre part que le travail total ne peut alors qu'augmenter. Cette augmentation vient du surcoût de synchronisation, qui est de plus en plus important lorsque l'on utilise plus de processeurs. Par exemple, le temps de calcul d'une application sur un seul processeur indique son travail effectif, puisqu'il n'y a alors aucun surcoût de synchronisation. Si l'on exécute cette application sur deux processeurs, le temps d'exécution va forcément être plus court que sur un seul (dans le cas contraire on aurait pu ignorer le deuxième processeur et gagner du temps), mais sera au moins la moitié de ce temps séquentiel, puisqu'il faut de toutes façons effectuer tous les calculs correspondant au travail effectif.

De façon formelle, l'hypothèse de monotonie suppose que toutes les tâches sont monotones, au sens suivant :

Définition 2

Une tâche est monotone si sa fonction p_i vérifie :

$$\forall q_1 \leq q_2 \in [0..m], \quad p_i(q_1) \geq p_i(q_2) \tag{2.1a}$$

$$\forall q_1 \leq q_2 \in [0..m], \quad q_1 p_i(q_1) \leq q_2 p_i(q_2) \tag{2.1b}$$

On donne souvent des noms aux deux comportements extrêmes autorisés par cette hypothèse. On parle ainsi de tâche *séquentielle* pour une tâche qui ne peut utiliser qu'un seul processeur, et dont le temps d'exécution est donc toujours le même quel que soit le nombre de processeurs alloués : $\forall q, p_i(q) = p_i(1)$. À l'inverse, on qualifie une tâche de *parfaitement parallèle* si elle n'a aucun surcoût de parallélisme, c'est-à-dire que son travail est constant quel que soit le nombre de processeurs alloués. On a alors $p_i(q) = \frac{p_i(1)}{q}$.

Pour ce problème d'ordonnancement, nous nous sommes intéressés à deux critères différents : le makespan et le temps de complétion moyen pondéré par les ω_i. Le but est d'essayer de répondre à la fois aux attentes de l'administrateur de la machine, qui veut que l'utilisation du matériel soit efficace, et à celles des utilisateurs, qui veulent un temps de complétion moyen le plus faible possible. Formellement, nous allons concevoir un algorithme qui a une garantie de performance sur les deux critères simultanément.

2.1.2 Résultats précédents

Pour le makespan

Comme on l'a vu rapidement dans la section 1.4, les algorithmes d'ordonnancement pour les tâches modelables séparent presque toujours le problème en deux phases : déterminer d'abord une allocation (c'est-à-dire le nombre de processeurs que chaque tâche va utiliser), puis l'ordonnancement lui-même, souvent avec un algorithme déjà connu pour les tâches rigides. Comme indiqué à la section 1.2.1, la preuve des garanties de performance de ces algorithmes pour les tâches rigides se fait presque toujours grâce à deux bornes inférieures complémentaires : le travail total et la longueur de la plus longue tâche. Ludwig et al. [50] ont montré qu'il est possible d'obtenir en temps polynomial une allocation qui minimise le maximum de ces deux bornes inférieures ; cela permet de transférer aux tâches modelables les garanties de performance pour les tâches rigides. L'algorithme List Scheduling décrit à la section 1.4.1 fournit ainsi une 2-approximation sur le makespan pour le problème des tâches modelables indépendantes.

La meilleure garantie obtenue à ce jour est de $3/2 + \epsilon$ pour tout $\epsilon > 0$, en utilisant l'algorithme **MRT** de Mounié et al. [53]. Cet algorithme spécialise un peu plus la phase d'allocation, de façon à obtenir une allocation qui facilite grandement la phase d'ordonnancement. L'observation clé consiste à noter que si l'ordonnancement optimal a un makespan de D, il ne peut y avoir plus de m processeurs occupés par des tâches de longueur strictement plus grande que $D/2$. La phase d'allocation va donc chercher à partitionner les tâches en deux ensembles : un ensemble de tâches « longues » qui ont un temps d'exécution plus grand que $D/2$, et un ensemble de tâches « courtes ». Par un algorithme de sac-à-dos, de complexité $O(nm)$, on peut trouver la partition qui minimise le travail total sous la contrainte que les tâches longues utilisent moins de m processeurs. Le travail total obtenu est nécessairement inférieur ou égal au travail de l'ordonnancement optimal. La phase d'ordonnancement consiste alors à faire subir à cette allocation une série de transformations qui permet d'ordonnancer ces tâches en moins de $3D/2$.

On obtient ainsi un algorithme qui, étant donné une valeur D du makespan, est capable de trouver un ordonnancement de makespan ρD (avec $\rho = 3/2$), ou bien de garantir qu'il n'existe aucun ordonnancement de makespan inférieur ou égal à D. Ce genre d'algorithme est appelé algorithme d'*approximation duale* [29]. Il est alors assez naturel de procéder à une dichotomie sur la valeur de D entre une borne inférieure I et une borne supérieure S. Cela permet alors d'obtenir, pour tout $\epsilon > 0$, une garantie de performance de $\rho + \epsilon$ en $\log_2\left(\frac{S}{I\epsilon}\right)$ étapes, chaque étape consistant en un appel de l'algorithme d'approximation duale.

Pour la moyenne des temps de complétion

Pour minimiser la somme des temps de complétion, une règle raisonnable est d'ordonnancer en premier les tâches les plus petites. En effet, si l'on exécute une tâche courte après une tâche longue, les deux tâches ont un grand temps de complétion. En revanche, si on les exécute dans l'ordre inverse, seule la tâche longue finit tard, et la moyenne est alors plus petite. Cette règle se retrouve dans le cas d'ordonnancement sur une seule machine, où l'algorithme **WSPTF**, très connu [46], qui consiste à ordonnancer par ordre croissant de p_i (ou de p_i/ω_i dans le cas pondéré), produit un ordonnancement optimal. La plupart des résultats pour les tâches parallèles sont obtenus en arrangeant les tâches par « étagères » et en les triant ensuite suivant cette même règle. Ce principe est ainsi utilisé par l'algorithme « smart SMART » de Schwiegelshohn et al. [62], qui est conçu pour ordonnancer des tâches rigides avec dates d'arrivée. En regroupant les tâches qui ont une longueur proche, puis en formant des étagères par un algorithme glouton avec une priorité bien choisie, cet algorithme obtient une garantie de performance de 8 en l'absence de pondération, et 8.53 pour la moyenne pondérée.

Avec une autre approche, comme évoqué précédemment dans la section 1.4.1, il a également été prouvé [69] que l'algorithme **FIFO**, lorsque les tâches sont triées par travail croissant et ne demandent pas plus de la moité des ressources disponibles, a une garantie de 2 sur la moyenne non pondérée des temps de complétion. Ce résultat n'est cependant valable qu'en l'absence de dates d'arrivées des tâches. Dans le même article, les auteurs étudient la version modelable du même problème, et arrivent, grâce à un couplage de poids minimal dans un graphe biparti, à obtenir une allocation optimale des tâches, restreinte pour satisfaire l'hypothèse précédente. Cela leur permet d'appliquer l'algorithme précédent et d'obtenir une garantie de performance de 2 également pour les tâches modelables, au prix d'une complexité en $O\left(n^3\right)$.

2.2 Une garantie sur les deux critères

Dans cette section, nous allons présenter l'algorithme **Lots**, basé sur une construction proposée par Hall et al. [28]. La contribution de cette section est un algorithme d'approximation duale spécifique aux tâches modelables, qui permet d'obtenir des garanties de performance meilleures que précédemment. Cette section introduit également un paramètre dans le schéma général qui permet d'exprimer un compromis entre les deux critères que l'on cherche à optimiser.

2.2.1 Schéma par lots

Le principe de l'algorithme (et de la construction de Hall et al. [28]) est de considérer des *lots* de tailles fixées, et d'ordonnancer un ensemble de tâches dans ces lots dont le poids est le plus grand possible. Ainsi, ordonnancer le plus de poids possible le plus tôt possible permet d'obtenir une garantie sur la moyenne pondérée des temps de complétion.

Pour ce faire, il est nécessaire d'avoir accès à un algorithme qui sélectionne les tâches de manière à ordonnancer un maximum de poids dans un temps donné. Il faut donc un algorithme qui résout le problème suivant[1] :

Problème 3 (Maximum Scheduled Weight (Mswp))

> **Instance :** *Un nombre de processeurs m, une date D, et un ensemble de tâches $\mathcal{T}$, chacune associée à un poids ω_i.*
>
> **Solution :** *Un sous-ensemble S de $\mathcal{T}$ dont le poids $\sum_{i\in S}\omega_i$ est maximal parmi tous les sous-ensembles ordonnançables en temps D.*

Bien entendu, il s'agit d'un problème difficile, et il est donc irréaliste de supposer que nous avons accès à un algorithme qui le résout de manière optimale. Nous allons donc supposer qu'il existe un algorithme polynomial $\mathcal{A}$ qui le résout de manière approchée, au sens suivant : s'il existe une solution de poids Ω ordonnançable en temps D, alors $\mathcal{A}$ fournit une solution de poids supérieur ou égal à Ω ordonnançable en temps ρD.

Une fois que l'on a un tel algorithme approché $\mathcal{A}$, l'idée est d'allouer des lots dont la taille augmente de façon exponentielle, afin de pouvoir ordonnancer les petites tâches au début (pour gagner sur la somme des temps de complétion) tout en ayant à la fin un lot assez grand pour obtenir une garantie sur le makespan. On va donc définir des instants $\tau_l \equiv \beta\alpha^l$, avec $\alpha > 1$ et $\beta > 0$, qui seront utilisées comme frontières de ces lots pour notre algorithme. Les constantes α et β seront fixées plus tard pour optimiser les garanties obtenues. L'algorithme **Lots** consiste alors à utiliser $\mathcal{A}$ pour ordonnancer les tâches disponibles entre deux τ_l consécutifs, voir ci-dessous l'algorithme 1.

Algorithme 1 Principe de l'algorithme **Lots**.

```
Calculer β
X ← T
l ← 1
while X ≠ ∅ do
    S ← A(X, (1/ρ)(τ_{l+1} − τ_l))
    Ordonnancer S entre τ_l et τ_{l+1}
    X ← (X \ S)
    l ← l + 1
end while
```

Tout d'abord, voyons comment construire un algorithme $\mathcal{A}$ de garantie $\rho = 3/2$.

[1] La nature des tâches est volontairement laissée vague, car ce schéma général peut s'adapter à une large classe de modèles d'application, et il n'est pas nécessaire de spécifier cette nature pour obtenir des résultats intéressants.

2.2.2 Adaptation de l'algorithme MRT

Comme exposé plus haut, le cœur de l'algorithme de **MRT** [53] consiste en un sac-à-dos (résolu par programmation dynamique) pour séparer les tâches entre longues et courtes. À chaque étape, on prend la décision qui minimise la surface totale occupée par les tâches, sous la contrainte que les longues tâches ne peuvent occuper plus de m processeurs. On définit ainsi $W(i,l)$ comme étant la surface minimale pour ordonnancer les i premières tâches, sous la contrainte que les longues tâches utilisent moins de l processeurs. La relation de récurrence est alors :

$$W(i+1,l) = \min \begin{cases} W(i,l) + w_i(D/2) & \text{alors } i \text{ est une tâche courte} \\ W\big(i, l - q_i(D)\big) + w_i(D) & \text{si } l \geq q_i(D)\,;\ i \text{ est une tâche longue} \end{cases}$$

où $q_i(t)$ est le nombre minimal de processeurs nécessaires à allouer à la tâche i pour avoir un temps d'exécution plus petit que t, et $w_i(t)$ est le travail correspondant : $q_i(t) = \min\{q \,|\, p_i(q) \leq t\}$ et $w_i(t) = q_i(t) \times p_i\big(q_i(t)\big)$.

Pour pouvoir utiliser cet algorithme pour le problème de Poids Maximal, il faut lui rajouter la possibilité de *rejeter* des tâches. Pour cela, il suffit de modifier l'algorithme de programmation dynamique pour séparer les tâches entre longues, courtes et rejetées.

Le principe est alors de calculer la surface minimale possible pour un ensemble de tâches dont le poids est supérieur à un poids donné : on définit $W(i,l,u)$ comme étant la surface minimale nécessaire pour ordonnancer un sous-ensemble des i premières tâches de poids supérieur ou égal à u, sous la contrainte que les tâches longues utilisent moins de l processeurs. On obtient la relation de récurrence suivante :

$$W(i+1,l,u) = \min \begin{cases} W(i,l,(u-\omega_i)_+) + w_i(D/2) \\ W(i,l-q_i(D),(u-\omega_i)_+) + w_i(D) \quad \text{si } l \geq q_i(D) \\ W(i,l,u) \end{cases}$$

où $(x)_+ = \max(x,0)$. Les valeurs de W pour $i = 0$ sont initialisées à $W(0,l,0) = 0$, et $W(0,l,u) = +\infty$ pour $u > 0$. Une fois que toutes les valeurs $W(i,l,u)$ sont calculées, il suffit de trouver le plus grand u tel que $W(n,m,u)$ soit inférieur ou égal à mD : comme il est impossible d'ordonnancer des tâches qui représentent plus de mD de travail en temps D, on a la garantie que ce poids u est au moins aussi grand que le poids maximal ordonnançable en temps D. Ensuite, l'algorithme **MRT** assure de pouvoir ordonnancer les tâches sélectionnées en temps au plus $3D/2$.

La complexité de cet algorithme est en $O(nm\Omega)$, où $\Omega \equiv \sum \omega_i$ est la somme de tous les poids des tâches. Si tous les poids sont inférieurs à $\omega_{\max}$, la complexité est donc en $O(n^2 m \omega_{\max})$. On peut cependant remarquer que dans la plupart des problèmes réels, l'écart entre les différents poids n'est pas très grand : si une tâche

est très prioritaire, il est plus logique de fixer son ordonnancement dès le début et d'ordonnancer les autres tâches en respectant cette contrainte[2].

2.2.3 Analyse

Nous allons exposer maintenant une preuve de la garantie de performance de l'algorithme **Lots** présenté sur la figure 1. Dans toute la suite, nous supposerons que l'algorithme $\mathcal{A}$ d'approximation pour le problème de poids ordonnancé maximal a une garantie de ρ. La technique de preuve utilisée ici est similaire à celle utilisée dans [28] pour le cas $\alpha = 2$. Nous généralisons ici à $\alpha > 1$ quelconque, ce qui donne une famille d'algorithmes dont le paramètre α fournit un compromis entre le makespan et la moyenne des temps de complétion.

Première étude

On va tout d'abord analyser la performance de cet algorithme pour les deux critères, quelle que soit la valeur de β suffisamment petite. Plus précisément, en notant $p_{\min}$ le plus petit temps d'exécution d'une tâche, nous allons prouver le théorème suivant :

Théorème 2

Pour tout $\alpha > 1$ et $0 < \beta < \frac{\rho}{\alpha-1} p_{\min}$, l'algorithme **Lots** *(1) est une ρ'-approximation pour la moyenne pondérée des temps de complétion et pour le makespan, avec $\rho' = \rho\frac{\alpha^2}{\alpha-1}$.*

Commençons par prouver la garantie sur la moyenne pondérée des temps de complétion ; soit σ^* un ordonnancement optimal pour ce critère. Pour étudier cet ordonnancement, nous allons définir des instants $\tau_l^* \equiv \frac{\alpha-1}{\rho}\tau_l$, avec toujours $\tau_l = \beta\alpha^l$. On va commencer par prouver le résultat de dominance suivant :

Lemme 1

Pour tout l, l'algorithme **Lots** *effectue au moins autant de poids entre 0 et τ_l que l'optimal n'en ordonnance entre 0 et τ_{l-1}^*.*

Démonstration : Notons $\mathcal{S}_l^* \equiv \{i \in [1..n] \,|\, \tau_{l-1}^* < C_i^* \leq \tau_l^*\}$ l'ensemble des tâches qui finissent dans l'ordonnancement optimal entre τ_{l-1}^* et τ_l^*, et $\omega(\mathcal{S}_l^*)$ la somme des poids de ces tâches. De la même manière, on note $\mathcal{S}_l$ l'ensemble des tâches ordonnancées à l'étape l de l'algorithme (donc entre τ_l et τ_{l+1}), et $\omega(\mathcal{S}_l)$ le poids correspondant.

Considérons, pour une étape l fixée, l'ensemble $V_l \equiv \cup_{k=1}^{l}\mathcal{S}_k^* \setminus \cup_{k=1}^{l-1}\mathcal{S}_k$ des tâches qui terminent avant τ_l^* dans l'optimal, mais qui ne sont pas encore ordonnancées par l'algorithme au début de l'étape l. Ces tâches sont dans $\mathcal{X}$ lors de l'appel à l'algorithme $\mathcal{A}$, et peuvent être ordonnancées en temps τ_l^* puisque l'ordonnancement optimal le fait. Or, par la définition des τ_l^*, on a $\tau_l^* = \frac{1}{\rho}(\tau_{l+1} - \tau_l)$, ce qui est la valeur de D passée à l'algorithme $\mathcal{A}$ à l'étape l (voir la ligne 5 dans

[2] La notion de *files de priorité* présente dans la plupart des logiciels de gestion de ressources repose d'ailleurs sur ce principe.

l'algorithme 1). L'hypothèse d'approximation sur cet algorithme assure donc que l'ensemble $\mathcal{S}_l$ de tâches retourné à cette étape a un poids supérieur à celui de V_l :

$$\omega(\mathcal{S}_l) \geq \omega(V_l) \geq \sum_{k=1}^{l} \omega(\mathcal{S}_k^*) - \sum_{k=1}^{l-1} \omega(\mathcal{S}_k)$$

D'où le résultat de dominance :

$$\forall l, \quad \sum_{k=1}^{l} \omega(\mathcal{S}_k) \geq \sum_{k=1}^{l} \omega(\mathcal{S}_k^*) \tag{2.2}$$

■

Pour obtenir à partir de ce résultat une garantie sur la moyenne des temps de complétion, on utilise ce petit lemme technique :

Lemme 2

Soit $N \in \mathbb{N}$, et $(x_k)_{1\leq k\leq N}$ et $(y_k)_{1\leq k\leq N}$ deux séquences de nombres réels telles que :

$$\forall l \in [1..N-1], \quad \sum_{k=1}^{l} x_k \geq \sum_{k=1}^{l} y_k \tag{2.3a}$$

$$\sum_{k=1}^{N} x_k \leq \sum_{k=1}^{N} y_k \tag{2.3b}$$

Alors pour toute séquence croissante $(a_k)_{1\leq k\leq N}$ de réels positifs, on a :

$$\sum_{k=1}^{N} a_k x_k \leq \sum_{k=1}^{N} a_k y_k \tag{2.4}$$

Démonstration : La preuve consiste en une simple réécriture de la somme $\sum_{k=1}^{N} a_k x_k$. Posons $X \equiv \sum_{l=1}^{N-1}(a_{l+1} - a_l)\sum_{k=1}^{l} x_k$. En inversant l'ordre de sommation dans X, on obtient :

$$\begin{aligned} X &= \sum_{k=1}^{N-1} x_k \sum_{l=k}^{N-1} (a_{l+1} - a_l) \\ &= \sum_{k=1}^{N-1} x_k (a_N - a_k) \\ &= a_N \sum_{k=1}^{N-1} x_k - \sum_{k=1}^{N-1} a_k x_k \\ &= a_N \sum_{k=1}^{N} x_k - \sum_{k=1}^{N} a_k x_k \end{aligned}$$

De la même manière, si l'on note $Y \equiv \sum_{l=1}^{N-1}(a_{l+1} - a_l)\sum_{k=1}^{l} y_k$, on a $Y = a_N \sum_{k=1}^{N} y_k - \sum_{k=1}^{N} a_k y_k$. L'hypothèse de dominance 2.3a et la croissance de a

impliquent $X \geq Y$; on a alors :

$$\begin{aligned}\sum_{k=1}^{N} a_k x_k &= a_N \sum_{k=1}^{N} x_k - X \\ &\leq a_N \sum_{k=1}^{N} y_k - Y \\ &\leq \sum_{k=1}^{N} a_k y_k\end{aligned}$$

■

Soit alors L tel que toutes les tâches de l'ordonnancement optimal soient terminées avant τ_L^*. Le résultat de dominance implique alors que l'algorithme termine également au plus tard à l'étape L, et nous permet d'appliquer ce lemme pour obtenir : $\sum_{l=1}^{L} \tau_{l+1}\omega(\mathcal{S}_l) \leq \sum_{l=1}^{L} \tau_{l+1}\omega(\mathcal{S}_l^*)$. Comme toutes les tâches de $\mathcal{S}_l$ terminent avant τ_{l+1}, on a $\sum \omega_i C_i \leq \sum_{l=1}^{L} \tau_{l+1}\omega(\mathcal{S}_l)$. De plus, comme les $\mathcal{S}_l^*$ forment une partition de l'ensemble des tâches[3], on a $\sum_{l=1}^{L} \tau_{l-1}^*\omega(\mathcal{S}_l^*) \leq \sum \omega_i C_i^*$. Comme $\tau_{l+1} = \frac{\rho\alpha^2}{\alpha-1}\tau_{l-1}^*$, on a bien le résultat voulu :

$$\sum \omega_i C_i \leq \frac{\rho\alpha^2}{\alpha - 1} \sum \omega_i C_i^*$$

Étudions maintenant la garantie de l'algorithme **Lots** sur le makespan. Si L est la dernière étape de l'algorithme **Lots**, cela veut dire que l'algorithme $\mathcal{A}$ n'a pas pu ordonnancer toutes les tâches à l'étape $L-1$. La propriété d'approximation de $\mathcal{A}$ garantit alors qu'il est impossible d'ordonnancer toutes les tâches en un temps $\frac{1}{\rho}(\tau_l - \tau_{l-1}) = \tau_{L-1}^*$. On en déduit donc que $C_{\max}^* \geq \tau_{L-1}^*$. Le makespan de la solution fournie par l'algorithme **Lots** est $C_{\max} \leq \tau_{L+1}$ puisqu'il a terminé à l'étape L, donc :

$$C_{\max} \leq \frac{\rho\alpha^2}{\alpha - 1} C_{\max}^*$$

Cette garantie de $\frac{\rho\alpha^2}{\alpha-1}$ est minimale lorsque $\alpha = 2$, on obtient alors pour l'algorithme **Lots** une garantie de 4ρ sur les deux critères à la fois. En utilisant comme algorithme $\mathcal{A}$ l'adaptation de **MRT** présentée dans la section 2.2.2, cela donne une garantie de 6.

Avec dates d'arrivées

Par souci de simplification, l'étude précédente a été faite dans un modèle sans dates d'arrivées, dans lequel toutes les tâches sont disponibles au début. Cependant, l'algorithme **Lots** peut très bien s'étendre à un cas où chaque tâche a une date d'arrivée : il suffit de rajouter, avant d'ordonnancer les tâches d'une étape l, toutes les tâches qui sont arrivées entre τ_{l-1} et τ_l. L'algorithme obtenu est alors

[3] C'est ici qu'intervient l'hypothèse $\beta < \frac{\rho}{\alpha-1} p_{\min}$, qui implique $\tau_0^* < p_{\min}$, et qui garantit donc qu'aucune tâche ne peut terminer avant τ_0^*.

un algorithme en ligne[4]. Cependant, il faut noter que le résultat de dominance n'est valable que si, à chaque étape, l'ensemble V_l est inclus dans $\mathcal{X}$; cela n'est garanti que si $\tau_l \geq \tau_l^*$. Cela restreint les choix possibles pour α : il faut et il suffit d'avoir $\alpha \leq \rho + 1$. De toutes façons, les garanties de performances sont optimisées en prenant $\alpha = 2$, qui vérifie bien cette contrainte.

Spécialisation hors-ligne

Dans un cadre hors-ligne (soit parce qu'il n'y a pas de dates d'arrivée, soit parce que l'on suppose que toutes les informations sont disponibles au début de l'ordonnancement), il est possible d'obtenir une meilleure garantie sur le makespan, en choisissant une valeur appropriée pour β. En effet, on peut déterminer à l'avance (en faisant une dichotomie sur D) la valeur minimale de D telle que $\mathcal{A}$ puisse ordonnancer toutes les tâches en temps ρD. Si ϵ est le pas de la dernière itération de la dichotomie, on a alors $C^*_{\max} \geq D - \epsilon$. Il suffit alors de choisir $\beta < \frac{\alpha-1}{\rho} p_{\min}$ de façon à avoir $\tau_L^* = D$ pour un certain L (par exemple, il suffit de choisir $L = \lfloor \log_\alpha(D/p_{\min}) \rfloor + 1$ et $\beta = \frac{\alpha-1}{\rho}\frac{D}{\alpha^L}$). Comme $\mathcal{A}$ sait ordonnancer toutes les tâches en temps ρD, l'algorithme aura nécessairement terminé à l'étape L, donc avec un makespan $C_{\max} \leq \tau_{L+1}$. On en déduit donc $C_{\max} \leq \frac{\rho\alpha}{\alpha-1}(C^*_{\max} + \epsilon)$.

L'algorithme **LotsOpt** ainsi obtenu a donc une garantie de $\frac{\rho\alpha}{\alpha-1}$ sur le makespan, et toujours une garantie de $\frac{\rho\alpha^2}{\alpha-1}$ sur la moyenne des temps de complétion (puisqu'elle était valable pour tout β suffisamment petit). Rappelons cependant qu'en présence de dates d'arrivée, ces garanties ne sont valables que pour $\alpha \leq \rho+1$.

Une famille d'algorithmes

Avec cette optimisation pour le cadre hors-ligne, on s'aperçoit que les garanties sur les deux critères ne changent pas de la même façon en fonction de α. La garantie sur le temps de complétion moyen est minimale pour $\alpha = 2$, on obtient alors une garantie de 4ρ ; en revanche, la garantie sur le makespan se rapproche de ρ lorsque α grandit. On peut donc dans une certaine mesure faire varier ce paramètre α pour choisir le compromis voulu entre les deux critères. La figure 2.1 montre l'ensemble des garanties atteignables de cette façon, avec $\rho = 3/2$. Les valeurs de α inférieures à 2 ne sont pas intéressantes, puisque les garanties sur les deux critères sont meilleures en prenant $\alpha = 2$; la partie correspondante de la courbe a donc été tracée en pointillés. De même, la partie correspondant aux valeurs de α supérieures à 2.5 n'est valable qu'en l'absence de dates d'arrivée, et a été tracée en pointillés plus larges.

2.2.4 Avec de l'aléatoire

Les bornes du théorème précédent sont valables dans le pire cas, et quelle que soit la valeur de β. Mais les instances qui atteignent ce pire cas dépendent

[4]Il est en fait *quasi* en ligne, car il nécessite d'avoir dès le début une information sur $p_{\min}$, la durée de la plus petite tâche. Mais ce n'est pas vraiment un problème, car cette information est souvent disponible dans les systèmes de gestion de ressources.

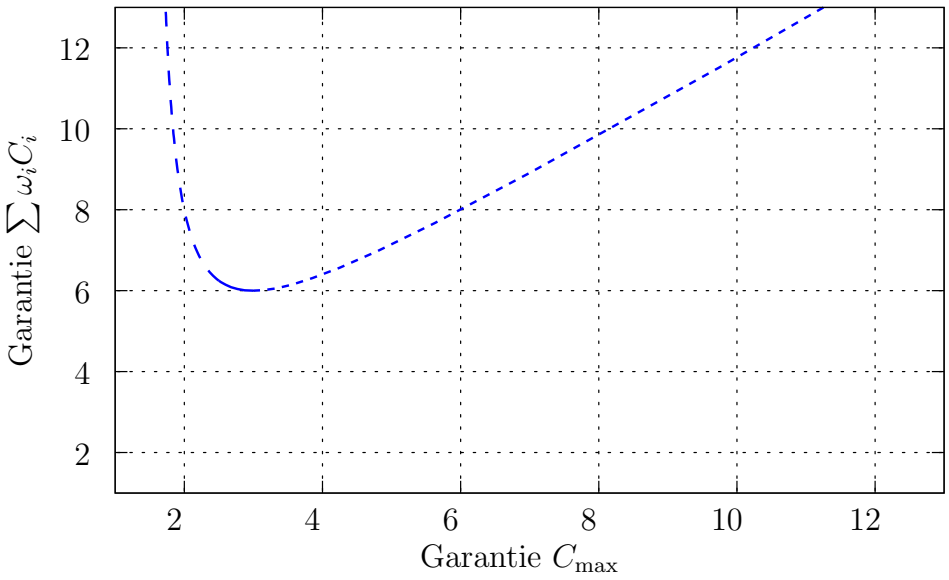


FIG. 2.1 – Garanties de l'algorithme **LotsOpt** lorsque α varie.

de β. Réciproquement, comme on l'a vu avec la spécialisation hors-ligne pour le makespan, il y a pour chaque instance des valeurs de β qui permettent d'obtenir une meilleure garantie.

Dans un cadre hors ligne, on peut ainsi essayer plusieurs valeurs, et garder l'ordonnancement qui donne la meilleure performance : on va ainsi obtenir avec une grande probabilité une garantie au pire cas (cela ne sera utile que pour la moyenne des temps de complétion, puisque l'on a vu que l'on est capable de trouver de manière déterministe une bonne valeur de β pour le makespan). Dans un cadre en ligne, on ne peut pas rejouer l'algorithme plusieurs fois ; cependant, si l'on tire β au hasard, on peut obtenir une garantie en moyenne. Notons bien que les résultats en moyenne que l'on va obtenir dans cette section ne sont que par rapport à la façon dont β est généré aléatoirement ; en particulier, les garanties sont valables pour toutes les instances. Il est possible qu'un mauvais tirage pour β fasse que la garantie fournie ne soit pas vérifiée, mais pour toute instance, si l'on exécute l'algorithme un nombre suffisant de fois, la moyenne des résultats obtenus vérifiera les bornes présentées ici.

Cette approche a été présentée pour la première fois dans [8] ; nous présentons ici une généralisation à α quelconque. Cela permet comme précédemment d'obtenir une plus vaste palette de facteurs d'approximations, mais également d'améliorer le facteur pour la moyenne des temps de complétion, car dans ce cadre le minimum est atteint pour $\alpha > 2$.

Nous allons donc prouver le théorème suivant :

Théorème 3

Pour tout $\alpha > 1$ et $\beta_0 < \frac{\rho}{\alpha-1} p_{\min}$, notons **LotsRand** *l'algorithme* **Lots** *ini-*

tialisé avec une valeur β aléatoire obtenue par $\beta = \alpha^{-X}\beta_0$, où X est tiré uniformément entre 0 et 1. Alors l'algorithme **LotsRand** *a une garantie de performance en moyenne de $\rho\frac{\alpha}{\ln(\alpha)}$.*

Démonstration : Considérons une instance I fixée, et σ' un ordonnancement fixé pour cette instance. Pour chaque tâche i, notons f_i l'indice tel que $i \in \mathcal{S}^*_{f_i+1}$, c'est-à-dire tel que $\tau^*_{f_i} \leq C'_i \leq \tau^*_{f_i+1}$. Comme elle dépend de la valeur de β, f_i est une variable aléatoire (on rappelle en effet que $\tau^*_{f_i} = \beta\alpha^{f_i}$). Nous allons maintenant déterminer l'espérance de $\tau^*_{f_i}$.

Pour cela, écrivons C'_i comme $C'_i = \alpha^{-\gamma_i}\beta_0\alpha^{l_i}$, avec l_i entier et $\gamma_i \in]0;1]$. On voit alors que $\tau^*_{l_i} \geq C'_i$ si et seulement si $X \leq \gamma_i$; comme de plus on a toujours $\tau^*_{l_i-1} \leq C'_i \leq \tau^*_{l_i+1}$, on en déduit que f_i vaut $l_i - 1$ si $X \leq \gamma_i$, et l_i sinon. On peut donc en déduire l'espérance de $\tau^*_{f_i}$:

$$\begin{aligned}
E\left(\tau^*_{f_i}\right) &= \int_0^1 \tau^*_{f_i}\mathrm{d}X \\
&= \int_0^{\gamma_i} \alpha^{-X}\beta_0\alpha^{l_i-1}\mathrm{d}X + \int_{\gamma_i}^1 \alpha^{-X}\beta_0\alpha^{l_i}\mathrm{d}X \\
&= \int_0^{\gamma_i} \alpha^{-X}\alpha^{\gamma_i-1}C'_i\mathrm{d}X + \int_{\gamma_i}^1 \alpha^{-X}\alpha^{\gamma_i}C'_i\mathrm{d}X \\
&= \alpha^{\gamma_i}C'_i\left(\frac{1}{\alpha}\int_0^{\gamma_i}\alpha^{-X}\mathrm{d}X + \int_{\gamma_i}^1\alpha^{-X}\mathrm{d}X\right) \\
&= \alpha^{\gamma_i}C'_i\left(\frac{1}{\alpha}\left(-\frac{\alpha^{-\gamma_i}}{\ln(\alpha)} + \frac{1}{\ln(\alpha)}\right) + \left(-\frac{\alpha^{-1}}{\ln(\alpha)} + \frac{\alpha^{-\gamma_i}}{\ln(\alpha)}\right)\right) \\
&= \frac{C'_i}{\ln(\alpha)}\left(1 - \frac{1}{\alpha}\right) = \frac{\alpha-1}{\alpha\ln(\alpha)}C'_i
\end{aligned}$$

Grâce à ce calcul d'espérance, nous pouvons maintenant affiner l'analyse précédente, dans laquelle le résultat de dominance et le lemme 2 amènent au résultat suivant :

$$\sum_i \omega_i C_i \leq \frac{\rho\alpha^2}{\alpha-1}\sum_{l=1}^{L}\tau^*_{l-1}\omega(\mathcal{S}^*_l)$$

En regroupant les sommes partielles, on obtient $\sum_{l=1}^{L}\tau^*_{l-1}\omega(\mathcal{S}^*_l) = \sum_i \omega_i\tau^*_{f_i}$. Le calcul précédent nous permet donc de conclure (en prenant comme ordonnancement de référence σ' un ordonnancement σ^* optimal pour la somme des temps de complétion) :

$$E\left(\sum_i \omega_i C_i\right) \leq \frac{\rho\alpha}{\ln(\alpha)}\sum_i \omega_i C^*_i$$

Pour le makespan, la même analyse que précédemment nous amène à écrire, si L est la dernière étape de l'algorithme **LotsRand** : $C_{\max} \leq \tau_{L+1} \leq \rho\frac{\alpha^2}{\alpha-1}\tau^*_{L-1}$. Considérons cette fois comme ordonnancement de référence σ' un ordonnancement σ^* optimal pour le makespan. Comme σ^* ne peut pas finir avant τ^*_{L-1}, si l'on note j la tâche qui termine en dernier dans cet ordonnancement, on a $\tau^*_{f_j} \geq \tau^*_{L-1}$. Donc $E\left(\tau^*_{L-1}\right) \leq E\left(\tau^*_{f_j}\right) = \frac{\alpha-1}{\alpha\ln(\alpha)}C^*_j$. Comme $C^*_{\max} = C^*_j$, on en déduit :

$$E\left(C_{\max}\right) \leq \rho\frac{\alpha}{\ln(\alpha)}C^*_{\max}$$

■

Spécialisation hors-ligne

L'optimisation hors-ligne précédente pour le makespan n'est pas applicable directement à l'algorithme **LotsRand**, puisque l'on ne peut plus choisir la valeur de β. En revanche, on peut toujours estimer la valeur de D comme ci-dessus, et l'utiliser pour fixer la longueur du dernier lot de l'algorithme à ρD. L'ordonnancement résultant a alors un makespan de $\tau_L + \rho D \leq \rho(\frac{\alpha}{\alpha-1}\tau^*_{L-1} + D)$. On obtient ainsi, de la même façon que ci-dessus, une garantie en moyenne sur le makespan de $\rho\left(1 + \frac{1}{\ln(\alpha)}\right)$.

En résumé

Il est donc possible d'utiliser l'algorithme **LotsRand** dans plusieurs cas différents, avec des garanties particulières pour chaque cas :

- Dans un cadre en ligne, on ne peut pas faire d'autre optimisation, et on obtient une garantie en moyenne de $\rho\alpha/\ln(\alpha)$ sur les deux critères. Contrairement au cas déterministe, la généralisation à $\alpha \neq 2$ permet d'améliorer les performances, puisque cette garantie est minimale égale à ρe lorsque $\alpha = e$. Cependant, la remarque précédente sur les dates d'arrivées est toujours valable, et pour $\rho = \frac{3}{2}$, limite α à être inférieur à $\frac{5}{2}$. Au total, l'algorithme **LotsRand** en utilisant l'adaptation de **MRT** comme algorithme $\mathcal{A}$ et en choisissant $\alpha = \frac{5}{2}$ est un algorithme en ligne avec une garantie en moyenne de $\frac{15}{4\ln(5/2)} \simeq 4,09$.
- Dans un cadre hors ligne, si l'on est intéressé par des garanties en moyenne, on peut utiliser l'optimisation précédente et obtenir des garanties en moyenne de $\rho\alpha/\ln(\alpha)$ sur le temps moyen de complétion et de $\rho\left(1 + \frac{1}{\ln(\alpha)}\right)$ sur le makespan. Une représentation graphique de ces garanties lorsque α varie est donnée sur la figure 2.2. À titre d'exemple, en prenant $\alpha = e$, on obtient une garantie de $\rho e \simeq 4,08$ sur le temps moyen de complétion, et de 3 sur le makespan.
- Toujours dans un cadre hors ligne, on peut cette fois chercher plutôt des garanties au pire des cas, en favorisant le temps moyen de complétion. Il suffit alors d'exécuter plusieurs fois l'algorithme **LotsRand** avec plusieurs tirages aléatoires, et de garder l'ordonnancement avec le meilleur temps moyen de complétion. Avec une forte probabilité, ce résultat sera en-dessous de la moyenne, et on a donc une garantie au pire cas de $\rho\alpha/\ln(\alpha)$ sur la $\sum \omega_i C_i$. En revanche, l'analyse en moyenne pour le makespan n'est plus valable, parce que l'on n'a aucune garantie qu'il existe un β pour lequel les garanties soient inférieures à la moyenne sur les deux critères à la fois. L'étude précédente pour l'algorithme **Lots** reste toutefois valable, et donne une garantie de $\frac{\alpha^2}{\alpha-1}\rho$. On peut améliorer cette garantie en reprenant l'amélioration ci-dessus qui consiste à diminuer la taille du dernier lot ; on obtient alors une garantie de $\frac{2\alpha-1}{\alpha-1}\rho$ sur le makespan. La courbe correspondante est également donnée sur la figure 2.2 sous la dénomination « $\sum \omega_i C_i$ moyen, pire $C_{\max}$ ». Encore

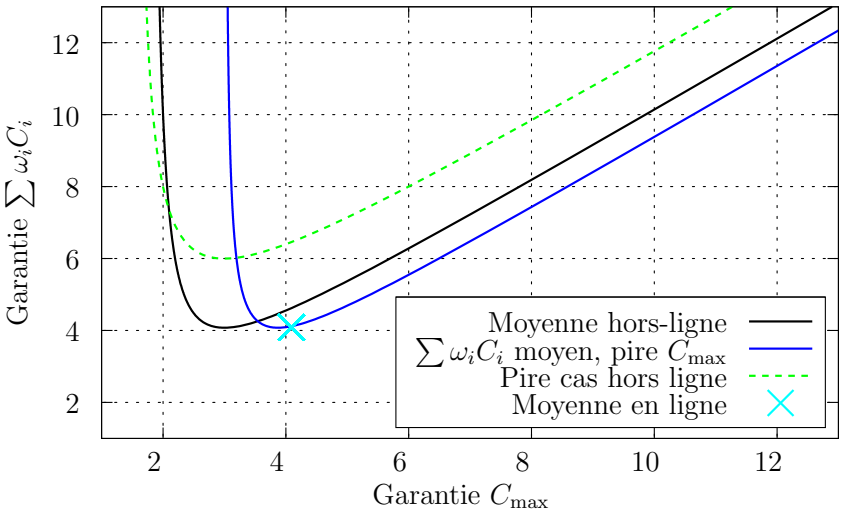


FIG. 2.2 – Comparaison des différentes garanties possibles pour **Lots** et **Lots-Rand**.

une fois, la meilleure garantie sur le temps moyen de complétion est obtenue avec $\alpha = e$ et vaut environ $4,08$; la garantie correspondante sur le makespan est d'environ $3,88$.

– Dans un cadre hors-ligne, si l'on cherche plutôt à favoriser le makespan, on n'a par contre pas besoin de l'aléatoire. En effet, la garantie sur le makespan obtenue avec l'optimisation hors ligne pour l'algorithme **Lots** est meilleure que la garantie en moyenne, puisque l'on peut calculer de manière déterministe la « bonne » valeur de β.

2.3 Bornes inférieures

Dans la suite de ce chapitre, nous allons concevoir une version simplifiée de cet algorithmes par lots, dans le but d'obtenir un algorithme ayant une complexité plus praticable. Cependant, cette simplification se fait au prix de la perte de la garantie de performance. Pour pouvoir valider cet algorithme, nous avons donc effectué des études expérimentales, consistant à générer aléatoirement un grand nombre d'instances et à évaluer les ordonnancements produits par notre algorithme et par d'autres algorithmes standard.

Pour avoir un point de comparaison objectif, il est de plus intéressant de se comparer également à l'ordonnancement optimal de chaque instance pour chacun des critères. Cependant, le problème d'ordonnancement étant NP-difficile au sens fort (que l'on considère l'un ou l'autre des deux critères), il n'est pas envisageable de calculer une solution optimale en temps raisonnable. Nous avons donc cherché à calculer de bonnes bornes inférieures.

2.3.1 Pour le makespan

Il existe deux bornes inférieures immédiates pour le makespan dans le cadre des tâches modelables, qui sont une généralisation des bornes de chemin critique et de travail exposées à la section 1.2.2 pour les tâches rigides. La première est la durée de la plus longue tâche, puisqu'il faut bien exécuter toutes les tâches. Comme la durée d'une tâche dépend du nombre de processeurs alloués, on peut seulement affirmer que le makespan est plus grand que la façon la plus rapide d'exécuter chaque tâche. Ainsi $B_1 = \max_i \min_q p_i(q)$ est une borne inférieure du makespan. L'hypothèse de monotonie implique que c'est avec m processeurs que chaque tâche s'exécute le plus vite, et donc dans ce cas $B_1 = \max_i p_i(m)$.

La deuxième borne inférieure classique est $\frac{W}{m}$, où W est la travail total à ordonnancer. Encore une fois, le travail d'une tâche modelable dépend de son allocation, et l'on peut seulement borner le travail dans un ordonnancement optimal par le plus petit travail possible. On obtient donc comme borne $B_2 = \frac{1}{m}\left(\sum_i \min_q q \times p_i(q)\right)$. Avec l'hypothèse de monotonie, c'est sur 1 processeur que chaque tâche a le plus petit travail, et la borne devient $B_2 = \frac{1}{m}\sum_i p_i(1)$.

Ces deux bornes inférieures classiques sont très faciles à calculer, mais sont un peu naïves. En particulier, elles ne concernent que les allocations extrêmes, sur 1 et m processeurs. Elles peuvent donc être très éloignées du vrai makespan optimal. Pour obtenir une meilleure borne, il est possible de recourir à l'approximation duale, comme on l'a vu dans la section 2.2. Le principe est toujours d'opérer une dichotomie pour trouver la plus grande valeur de D telle que l'algorithme **MRT** garantisse qu'il n'existe pas d'ordonnancement dont le makespan est inférieur à D. Si l'algorithme **MRT** répond négativement pour une valeur D, et positivement pour une valeur $D + \epsilon$ avec ϵ suffisamment petit (c'est-à-dire qu'il fournit un ordonnancement avec un makespan $\frac{3}{2}(D + \epsilon)$), on sait que la borne inférieure D est au pire à $\frac{2}{3}$ du makespan optimal.

2.3.2 Pour la moyenne des temps de complétion

Pour $\sum \omega_i C_i$, nous avons introduit une formulation par programmation linéaire, dont la relaxation permet d'obtenir une borne inférieure sur la valeur optimale. Cette formulation n'a pas pour but de mener à un ordonnancement réalisable, mais plutôt d'exprimer des contraintes qui doivent nécessairement être respectées par tout ordonnancement réalisable.

Le programme linéaire s'exprime de la façon suivante : on commence par diviser l'horizon temporel en intervalles $I_j =]t_j, t_{j+1}]$ pour j entre 0 et K. Les valeurs de t_j sont calculées comme précédemment, en augmentant de façon exponentielle à partir de $t_0 = p_{\min}$ de sorte qu'aucune tâche ne peut terminer avant t_0. Pour cette borne inférieure, nous avons choisi $\alpha = 2$, ce qui donne $t_j = p_{\min} 2^j$. Nous pouvons alors définir, pour chaque tâche i et chaque intervalle j :

- une variable de décision $x_{i,j}$, qui vaut 1 si la tâche i termine son exécution dans l'intervalle I_j, et vaut 0 dans le cas contraire ;
- la surface minimale occupée par la tâche i si elle termine avant t_{j+1} :

$$S_{i,j} \equiv \min_{1 \leq k \leq m} \{k p_i(k) | p_i(k) \leq t_{j+1}\}$$

Si l'ensemble est vide, cela signifie qu'il n'est pas possible d'ordonnancer la tâche i de manière à terminer avant t_{j+1}, et on définit alors $S_{i,j} = +\infty$

Avec ces valeurs, la formulation est :

$$\text{Minimiser} \quad \sum_{\substack{1\leq i\leq n\\ 0\leq j\leq K}} \omega_i t_j x_{i,j}$$

$$\text{De façon que} \quad \forall i, \quad \sum_j x_{i,j} \geq 1 \tag{2.5a}$$

$$\forall j < K, \quad \sum_{0\leq l\leq j} \sum_i S_{i,l} x_{i,l} \leq m t_{j+1} \tag{2.5b}$$

$$\forall i, \forall j, \quad x_{i,j} \in \{0,1\} \tag{2.5c}$$

La première contrainte (2.5a) exprime le fait que chaque tâche doit être exécutée au moins une fois. Le critère de minimisation implique qu'aucune tâche ne va être exécutée plus d'une fois : si deux variables $x_{i,j}$ et $x_{i,j'}$ d'une solution réalisable valent 1, on peut obtenir une solution meilleure et qui reste réalisable en annulant l'une des deux variables.

La deuxième contrainte (2.5b) est un argument de surface. Pour chaque intervalle I_j, on considère les tâches qui terminent avant la fin de cet intervalle : elles finissent dans un intervalle I_l pour $l \leq j$. Par définition, une tâche i qui termine dans un intervalle I_l utilise au moins une surface $S_{i,l}$. La somme de toutes ces surfaces doit être plus petite que la surface totale entre le temps 0 et t_{j+1}, qui vaut mt_{j+1}. Cette borne est optimiste, puisqu'elle ne prend pas en compte les collisions entre tâches : pour ordonnancer suivant une solution de cette formulation, il peut arriver d'avoir besoin de plus de m processeurs. On peut noter également que l'on n'impose pas cette contrainte pour le dernier intervalle I_K, ce qui fait que l'on autorise toutes les tâches qui n'ont pas terminé dans les intervalles précédents à terminer dans ce dernier intervalle. Ainsi le résultat du programme linéaire sera une borne inférieure quelle que soit la valeur de K ; augmenter K permet d'obtenir une borne plus précise. Pour obtenir la plus grande précision possible, nous avons choisi K de telle sorte qu'il soit possible d'ordonnancer toutes les tâches dans l'intervalle I_K, toujours grâce à l'approximation du makespan obtenue par approximation duale.

Ces deux contraintes sont respectées par tout ordonnancement réalisable S, auquel correspond donc une solution R de ce programme linéaire. Comme pour chaque tâche i, $\sum_j t_j x_{i,j} \leq C_i$, la fonction objective de la solution R est inférieure ou égale au critère $\sum \omega_i C_i$ de l'ordonnancement S. En particulier, ceci est vrai pour tout ordonnancement optimal, donc la valeur optimale de la fonction objective du programme linéaire est toujours plus petite que la valeur optimale du critère $\sum \omega_i C_i$ du problème d'ordonnancement.

Le programme linéaire (2.5) donne donc toujours une borne inférieure pour le problème d'ordonnancement. Cependant, comme c'est un programme linéaire en nombres entiers, sa résolution est trop longue pour être utilisable dans des simulations intensives sur de nombreuses instances. Nous avons donc utilisé la version relaxée de ce programme, obtenue en remplaçant la contrainte (2.5c) par

$x_{i,j} \in [0;1]$ (c'est-à-dire en éliminant la contrainte d'intégrité). Comme on ne fait que rajouter des solutions réalisables, la valeur optimale du programme linéaire reste une borne inférieure. Elle est certainement un peu plus éloignée de la valeur optimale du problème d'ordonnancement, mais se calcule plus rapidement.

2.4 Étude d'un algorithme plus praticable

L'algorithme garanti présenté à la section 2.2 est certes très intéressant, mais il a une complexité prohibitive, qui ne permet pas vraiment d'envisager de l'utiliser dans un système de production grandeur nature. C'est pourquoi nous avons conçu un algorithme un peu plus simple, en restreignant la forme de l'ordonnancement à l'intérieur des lots. Avec cette restriction, la preuve de la garantie de performance dans le pire cas n'est plus valable, mais la phase de sélection des tâches qui forment chaque lot est grandement simplifiée, ce qui va permettre de mettre l'algorithme en œuvre en pratique. Cette mise en œuvre est détaillée dans le chapitre 3. Nous présentons également dans cette section une évaluation expérimentale de cet algorithme par des simulations qui montre une très bonne performance en moyenne pour un temps d'exécution très faible.

2.4.1 Schéma en étagères

Dans l'algorithme précédent, la base de la performance de garantie vient de la structure en lots, dont la taille augmente exponentiellement avec le temps. C'est cette structure qui permet d'ordonnancer en priorité les tâches les plus petites, afin de favoriser le critère $\sum \omega_i C_i$. Au fur et à mesure que la taille des lots augmente, on commence à considérer des tâches de plus en plus grandes, ce qui donne plus de latitude à l'algorithme $\mathcal{A}$ qui résout le problème MSWP et permet d'obtenir également une garantie sur le makespan.

Afin de garder un algorithme qui ait un bon comportement pour le critère $\sum \omega_i C_i$, nous avons décidé de garder cette structure en lots dont les tailles augmentent de manière exponentielle. Mais le problème d'ordonnancement MSWP est conceptuellement trop complexe pour pouvoir être adapté à un système réel ; nous avons donc choisi de restreindre les ordonnancements utilisés pour remplir les lots et de ne considérer que des ordonnancements par *étagères*, c'est-à-dire où toutes les tâches d'un même lot commencent leur exécution au même instant. L'algorithme correspondant sera noté **SDE** (pour « Sac-à-Dos Étagères ») dans la suite ; son principe est représenté sur la figure 2.3.

2.4.2 Algorithme détaillé

La boucle principale de l'algorithme (lignes 3 à 11) correspond à la sélection des tâches qui vont être exécutées dans l'étagère courante. On commence (ligne 5) par sélectionner l'ensemble S des tâches qui ne sont pas trop longues, et qui sont donc candidates à être exécutées dans l'étagère ; à la ligne 6, on assigne à ces tâches le nombre minimal de processeurs qui leur permet de s'exécuter dans la durée de

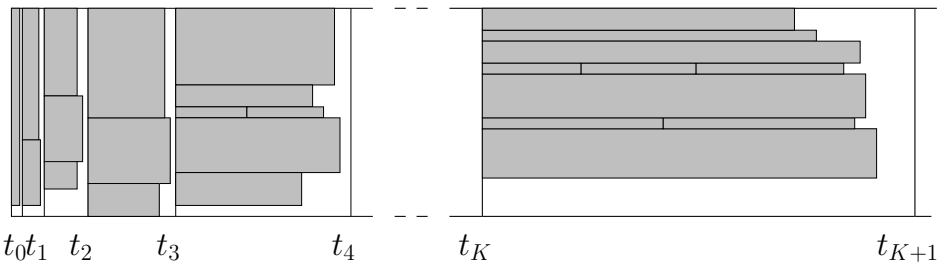


FIG. 2.3 – Principe de l'algorithme **SDE**.

Algorithme 2 Détail de l'algorithme **SDE**.

1: $t_0 = \min_{i,q}\{p_i(q)\}$; $k = 0$
2: $T \leftarrow \{1..n\}$
3: **while** $T \neq \emptyset$ **do**
4: $k \leftarrow k + 1$; $t_k = \alpha t_{k-1}$
5: $S = \{i \in T$ tel que $\exists q,\ p_i(q) \leq t_k\}$
6: $\forall i \in S, \pi_i = \min\{q | p_i(q) \leq t_k\}$
7: Regrouper les tâches séquentielles ($\pi_i = 1$) triées par poids décroissants.
8: Sélectionner $S_k \subseteq S$ de poids maximal tel que $\sum_{i \in S_k} \pi_i \leq m$ (en utilisant un sac à dos).
9: Ordonnancer toutes les tâches de S_k à la date t_j avec l'allocation π.
10: $T \leftarrow T \setminus S_k$
11: **end while**
12: Compacter l'ordonnancement par un algorithme glouton en utilisant l'ordre des étagères.

l'étagère. Le but de la sélection effectuée à la ligne 8 est alors de trouver le sous-ensemble de poids maximal que l'on peut ordonnancer ensemble, c'est-à-dire tel que le nombre total de processeurs soit inférieur à m.

Ce sous-problème est en fait exactement équivalent au problème du sac à dos, que l'on peut résoudre de manière exacte par un algorithme de programmation dynamique. Pour cela, on va calculer les valeurs $\Omega(i, j)$, définies pour $0 \leq i \leq n$ et $0 \leq j \leq m$ comme le plus grand poids d'un sous-ensemble des i premières tâches que l'on peut ordonnancer sur m processeurs. On a la relation de récurrence suivante :

$$\Omega(i, j) = max\left(\Omega(i-1, j), \Omega(i-1, j-\pi_i) + \omega_i\right)$$

En calculant les valeurs de $\Omega(i, j)$ à partir de $\Omega(0, j) = 0$, on peut accéder à la valeur de $\Omega(n, m)$ qui est le poids maximal qui peut être ordonnancé dans l'étagère. La complexité de cette sous-procédure est $O(nm)$.

La ligne 7 correspond à une optimisation que l'on effectue pour les petites tâches, et qui relaxe la contrainte des étagères sans augmenter la complexité de l'algorithme. Les tâches pour lesquelles π_i vaut 1 sont dites *petites* pour l'étagère courante : on ne peut plus augmenter leur temps de calcul en leur attribuant moins de processeurs. Si on les ordonnance en suivant la contrainte d'étagères, c'est-à-

dire en les faisant commencer toutes en même temps, l'étagère va contenir une grande quantité de temps d'inactivité, où les processeurs n'effectuent aucune tâche. Cependant, si l'ensemble S contient plusieurs de ces petites tâches dont la somme des temps de calcul est plus petite que la longueur de l'étagère, il est possible de les empiler pour qu'elles n'occupent qu'un seul processeur. Notre algorithme va donc les regrouper, en commençant par les tâches de plus grand poids, de façon à ce qu'elle soient interprétées comme une seule tâche séquentielle par le programme dynamique de sac à dos. On garde ainsi la propriété que l'algorithme de sélection ne travaille que sur la dimension d'espace, et ne prend pas en compte les durées des tâches.

Une fois toutes les étagères déterminées, on peut ordonnancer toutes les tâches de chaque étagère à la date de début de l'étagère. Cet ordonnancement est valide, mais il contient encore une grande quantité de temps d'inactivité, puisque toutes les tâches d'une même étagère ne durent pas nécessairement aussi longtemps. Une amélioration immédiate est possible : on fait commencer une tâche plus tôt si tous les processeurs qu'elle utilise sont disponibles. Cette amélioration résulte en un ordonnancement qui n'est plus en étagères, mais qui est nécessairement plus performant, puisque chaque tâche ne peut que s'exécuter plus tôt que dans l'ordonnancement en étagères. Cependant, la structure en étagères a permis de simplifier assez le problème pour pouvoir sélectionner les plus petites tâches en premier.

On peut même aller un peu plus loin : une fois les étagères déterminées, on trie les tâches suivant l'ordre des étagères (et pour deux tâches de la même étagère, on peut se donner un autre critère de priorité, comme le poids ω_i par exemple), et on utilise cet ordre comme entrée pour un algorithme glouton de type **FIFO** (voir la section 1.4.1) qui place chaque tâche dans l'ordre le plus tôt possible. L'idée est la même que pour l'optimisation précédente, mais on s'autorise éventuellement à changer l'ensemble de processeurs sur lequel une tâche va s'exécuter si cela permet de la commencer plus tôt. Cette compaction est effectuée à la ligne 12.

Pour terminer, on effectue une dernière étape d'optimisation : on mélange l'ordre des étagères plusieurs fois, et on conserve l'ordonnancement qui a la meilleure performance pour $\sum \omega_i C_i$ une fois compacté.

2.4.3 Analyse expérimentale

Cette section présente l'analyse expérimentale que nous avons effectuée pour valider cet algorithme **SDE**. Bien qu'il n'ait pas de garantie théorique connue, son comportement dans ces simulations montre une très bonne performance par rapport aux autres algorithmes connus, surtout sur la moyenne des temps de complétion.

Environnement d'expérimentation

Pour effectuer ces simulations, nous avons généré un grand nombre d'instances aléatoires à partir de paramètres réalistes que nous allons détailler ici. Les instances générées comportent toutes 200 processeurs, et un nombre de tâches variant de 25

à 400. Ce nombre de processeurs correspond à une grappe de taille standard, mais nous avons remarqué que les résultats ne dépendent pas vraiment du nombre de processeurs, mais plutôt du rapport entre le nombre de processeurs et le nombre de tâches. Il suffit donc de faire varier le nombre de tâches pour observer cette dépendance. Le poids de chaque tâche est tiré uniformément entre 1 et 10.

La spécification du temps de calcul d'une tâche dans le modèle modelable nécessite de fournir un vecteur de m temps de calcul : un pour chaque nombre de processeurs susceptible de lui être alloués. Pour générer un tel vecteur, nous avons (d'une façon assez classique) découplé la modélisation en deux parties : la première partie génère les temps séquentiels des tâches, ce qui correspond à la charge de travail qu'elles représentent, et la deuxième fournit à partir de ce temps séquentiel les temps de calcul sur plusieurs processeurs. Cette deuxième partie représente le surcoût de parallélisme des tâches, c'est-à-dire leur caractère plus ou moins parallèle. Pour chacune de ces deux parties, nous avons utilisé deux modèles différents, pour essayer de couvrir un large spectre d'instances.

Modèles de temps séquentiels Pour la première partie, nous avons utilisé un modèle simple uniforme et un modèle mixte. Dans le cas uniforme, les temps séquentiels sont générés suivant une distribution uniforme sur $[1, 30]$ (le nombre 30 correspond à peu près au rapport entre les plus petites et les plus grandes tâches sur une grappe standard). Dans le cas mixte, nous avons introduit deux classes de tâches : des petites et des grandes tâches. Les temps séquentiels sont alors générés suivant une loi normale centrée respectivement sur 1 et 20, avec des écarts-types respectifs de 1.0 et 20, la proportion des petites tâches étant de 70%.

Modèles de parallélisme Le premier modèle que nous avons utilisé pour générer les temps parallèles des tâches est celui utilisé dans [53]. Dans ce modèle, les temps de calcul successifs sont calculés par la formule $p_i(j) = p_i(j-1)\frac{X+j}{1+j}$, où X est une variable aléatoire entre 0 et 1. Cette formule garantit que la tâche obtenue vérifie bien l'hypothèse de monotonie. Selon la distribution de X, les tâches générées sont fortement parallèles (avec une accélération quasi-linéaire) si X est proche de 0, et faiblement parallèles (avec une accélération proche de 1) si X est proche de 1. Ces deux cas sont générés en utilisant respectivement une loi normale centrée sur 0.9 et 0.1, avec un écart-type de 0.2, dans laquelle toute valeur plus petite que 0 ou plus grande que 1 est ignorée et recalculée. Bien que ce modèle ne soit certainement pas très réaliste, nous l'avons utilisé afin de tester nos algorithmes dans les cas extrêmes de tâches très ou très peu parallèles. Quand ce modèle est utilisé avec des temps séquentiels mixtes, les petites tâches sont faiblement parallèles, et les grandes tâches sont fortement parallèles.

Le deuxième modèle que nous avons utilisé provient d'une enquête effectuée par Cirne et Berman [12] sur le comportement des utilisateurs de plusieurs centres de calcul (entre autres à la NASA, au NCSA, au NERSC et au NPACI). Cet article repose sur un modèle de Downey [14] pour le comportement des tâches parallèles, qui prend en compte deux paramètres : A, le parallélisme moyen de la tâche, et σ, qui représente la variance autour de ce parallélisme moyen tout au long de

son exécution. À partir de ces valeurs et d'un modèle assez simple de l'exécution d'une tâche, on peut calculer le temps d'exécution d'une tâche sur j processeurs. L'enquête menée par Cirne et Berman fournit des valeurs de nombre de processeurs minimal, optimal et maximal, qui sont plus parlantes pour les utilisateurs de ces centres de calculs. Les auteurs utilisent alors ces résultats pour en déduire les lois de distribution des paramètres de Downey :

- celle de A, qui est une loi log-uniforme de fonction de répartition $cdf(x) = \chi \log_2(x) + \rho$ de paramètres $\chi_A = 0.07468$ et $\rho_A = -0.009198$,
- celle de σ, qui est une loi normale de moyenne $\mu_\sigma = 1.209$ et de variance $\sigma_\sigma = 1.132$.

Pour effectuer les simulations, nous avons développé un programme spécifique. Pour chaque valeur des paramètres (nombre de tâches, modèle de génération des temps séquentiels, modèle du parallélisme), on effectue 30 expériences ; pour chacune d'elles on génère une instance aléatoire que l'on donne en entrée des algorithmes d'ordonnancement et des algorithmes qui calculent les bornes inférieures. On obtient ainsi un makespan $C^{\mathcal{A}}_{\max}$ et une moyenne des temps de complétion $\sum \omega_i C_i^{\mathcal{A}}$ pour chaque algorithme, ainsi que deux bornes inférieures $C^{BI}_{\max}$ et $\sum \omega_i C_i^{BI}$ respectivement du makespan optimal et de la moyenne des temps de complétion optimale. On définit donc deux ratios de performance pour chaque algorithme et pour chaque expérience :

$$r^{\mathcal{A}}_{C_{\max}} = \frac{C^{\mathcal{A}}_{\max}}{C^{BI}_{\max}} \quad \text{et} \quad r^{\mathcal{A}}_{\sum \omega_i C_i} = \frac{\sum \omega_i C_i^{\mathcal{A}}}{\sum \omega_i C_i^{BI}}$$

. La simulation pour ces valeurs des paramètres fournit alors deux valeurs pour chaque algorithme : la moyenne des $r_{C_{\max}}$ et la moyenne des $r_{\sum \omega_i C_i}$ sur l'ensemble des 30 expériences.

Algorithmes de comparaison

Comme nous l'avons expliqué précédemment, nous utilisons des bornes inférieures de la solution optimale pour chacun des critères comme référence pour évaluer la qualité de l'algorithme. Pour juger du comportement et de l'efficacité de notre approche, nous avons également comparé nos résultats avec ceux d'algorithmes standard, que nous détaillons ici.

MRT : il s'agit de l'algorithme **MRT** [53], garanti sur le makespan, que nous avons présenté au début de ce chapitre.

Gang : cet algorithme exécute chaque tâche sur tous les processeurs, en les triant par ordre croissant du rapport de leur poids par leur temps d'exécution. Il est optimal sur les deux critères pour les instances où les tâches ont une accélération linéaire.

Séquentiel (Séq.) : cet algorithme est l'inverse du précédent. Il exécute chaque tâche sur un seul processeur, en utilisant un algorithme glouton qui ordonnance d'abord les tâches avec un plus grand temps d'exécution (cet algorithme est noté dans la littérature LPTF, pour "Largest Processing Time

First"). On s'attend à ce que cet algorithme soit particulièrement performant lorsque les tâches sont faiblement parallèles, en particulier pour le makespan.

Algorithmes de liste : nous avons également utilisé trois algorithmes d'ordonnancement de liste multiprocesseurs (voir section 1.4.1), en utilisant l'allocation fournie par l'algorithme **MRT**, ce qui devrait donner une très bonne performance vis-à-vis du makespan. Les trois algorithmes ne diffèrent que par l'ordre des tâches dans la liste :

- Le premier, **LPTF**, est une variante classique qui trie les tâches par ordre décroissant de leur temps d'exécution, et qui a un très bon comportement vis-à-vis du makespan.
- Dans le deuxième, **SAF** (pour "Smallest Area First"), les tâches sont triées par ordre croissant de leur "aire", c'est-à-dire du produit du nombre de processeurs par le temps d'exécution. Le but de cette variante est d'améliorer la performance vis-à-vis du $\sum \omega_i C_i$.
- Une autre variante, qui a également de bonnes performances sur le $\sum \omega_i C_i$, est **WSPTF**, pour « *Weighted Smallest Processing Time First* », qui trie les tâches par ordre croissant du rapport p_i/ω_i. Comme on l'a vu dans la section 1.3, cet algorithme est optimal pour le $\sum \omega_i C_i$ sur une seule machine.

Random : il s'agit d'un algorithme qui choisit une allocation au hasard pour chacune des tâches, et les ordonnance ensuite avec l'ordre **SPTF**.

Résultats

Nous allons exposer les résultats en donnant une courbe pour chaque combinaison de modèle de temps d'exécution et de parallélisme qui nous semble pertinente. Pour faciliter les comparaisons, nous allons toujours utiliser la même échelle ; cela a pour effet que certains algorithmes ne seront pas entièrement (voire pas du tout) visibles sur certains graphes, car ils obtiennent de trop mauvaises performances.

La figure 2.4 expose les résultats pour le modèle des tâches uniformes faiblement parallèles. Il s'agit d'un cas défavorable pour **SDE**, puisqu'il utilise beaucoup de ressources pour accélérer des tâches qui ne peuvent pas les utiliser de manière efficace. Il en est de même, mais dans une plus grande mesure, pour les algorithmes **Gang** et **Random**, qui n'apparaissent presque pas dans l'intervalle présenté sur ces courbes. On voit ici que c'est l'algorithme **SPTF** qui obtient les meilleurs résultats sur la moyenne des temps de complétion, et **LPTF** pour le makespan. Comme l'on pouvait s'y attendre, l'algorithme **Séquentiel** a de très bonnes performances dans ce cas sur les deux critères. Notons également que même si **SDE** est parmi les moins efficaces, il obtient pour ce cas très défavorable des ratios de performances en-dessous de 2 pour le makespan et de $2,5$ pour la moyenne des temps de complétions.

La figure 2.5 montre les mêmes expériences avec des tâches fortement parallèles. Ce cas est nettement plus avantageux pour **SDE**, qui obtient les meilleurs résultats pour la moyenne des temps de complétion, même avec un faible nombre de tâches, et conserve des résultats honorables pour le makespan. Pour le makespan, c'est

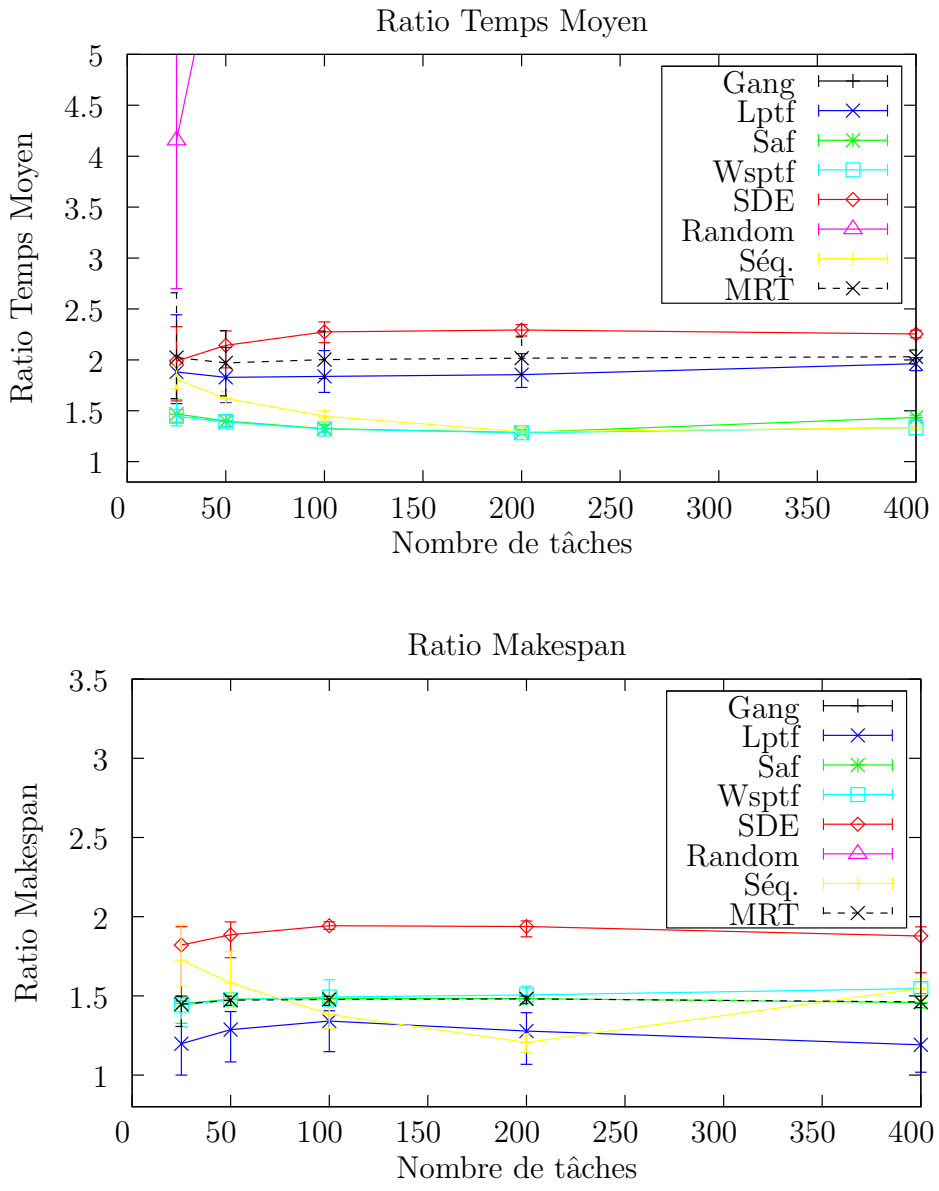


FIG. 2.4 – Ratios de performance pour des temps séquentiels uniformes et des tâches faiblement parallèles.

encore **LPTF** qui est le plus efficace, au prix de moins bonnes performances sur le deuxième critère. On remarque que **Gang** est plutôt bon lorsqu'il y a peu de tâches, et devient vraiment mauvais lorsque le nombre de tâches augmente ; à l'inverse, **Séquentiel** est bien meilleur dans les instances avec un grand nombre de tâches (comme chaque tâche n'occupe qu'un seul processeur, on comprend bien que ce n'est que lorsqu'il y a plus de tâches que de processeurs que **Séquentiel** est efficace).

L'expérience suivante, montrée sur la figure 2.6, contient des tâches mixtes : certaines petites et faiblement parallèles, d'autres grandes et fortement parallèles. La présence d'une majorité de tâches faiblement parallèles entraîne comme précédemment une performance relativement faible de **SDE**, par rapport à **SPTF** en particulier, qui bénéficie d'une allocation plus appropriée. L'algorithme **SDE** a cependant une performance relativement stable lorsque le nombre de tâches varie, et garde un ratio de performance d'environ 2 sur les deux critères. **LPTF** obtient encore une fois de très bons résultats sur le makespan, mais de très mauvais sur l'autre critère, surtout lorsqu'il y a un grand nombre de tâches. On voit ici clairement que la moyenne des temps de complétion est bien plus sensible à l'ordre des tâches. Notons également que de telles instances mixtes font que ni **Gang** ni **Séquentiel** n'obtiennent de bons résultats.

Lorsque toutes les tâches sont fortement parallèles, on obtient la figure 2.7. Le comportement est sensiblement le même pour le makespan, à part pour **Gang** qui obtient (de manière prévisible) de meilleurs résultats. En revanche, pour la $\sum \omega_i C_i$, on voit que les algorithmes de liste sont moins performants pour ce cas-là, et que **SDE** est le plus efficace, à part pour les instances avec très peu de tâches sur lesquelles **Gang** est meilleur.

Enfin, la figure 2.8 montre les résultats obtenus avec le modèle de parallélisme de Cirne et Berman [12]. Dans cette expérience, l'algorithme **SDE** garde la performance constante que l'on a pu constater jusqu'à présent, voire obtient même des ratios de performance un petit peu meilleurs. En revanche, les autres algorithmes, même **SPTF**, ont une performance fortement dégradée sur la moyenne des temps de complétion. Pour le makespan, c'est toujours **LPTF** qui obtient les meilleurs résultats, et **SPTF** est assez performant également.

Dans une dernière expérience (voir figure 2.9), nous avons repris les instances de la figure précédente, et nous avons inclus dans la simulation l'algorithme **Lots**, augmenté d'une phase de compaction comme celle de **SDE**. On peut noter que les performances obtenues par **Lots** sont très similaires à celles de **SDE** ; cependant, le temps d'exécution de l'algorithme **Lots** est environ 200 fois plus important pour les plus grosses instances. Sur cette figure, il n'y a pas de point pour $n = 400$ tâches, car l'algorithme **Lots** est trop lent pour ces instances et n'arrivait pas à produire un résultat.

Résumé

À partir de ces résultats, on peut faire plusieurs observations intéressantes. Le premier résultat frappant est que **SDE** est d'une stabilité remarquable : il obtient des ratios de performance sensiblement équivalents pour toutes les instances tes-

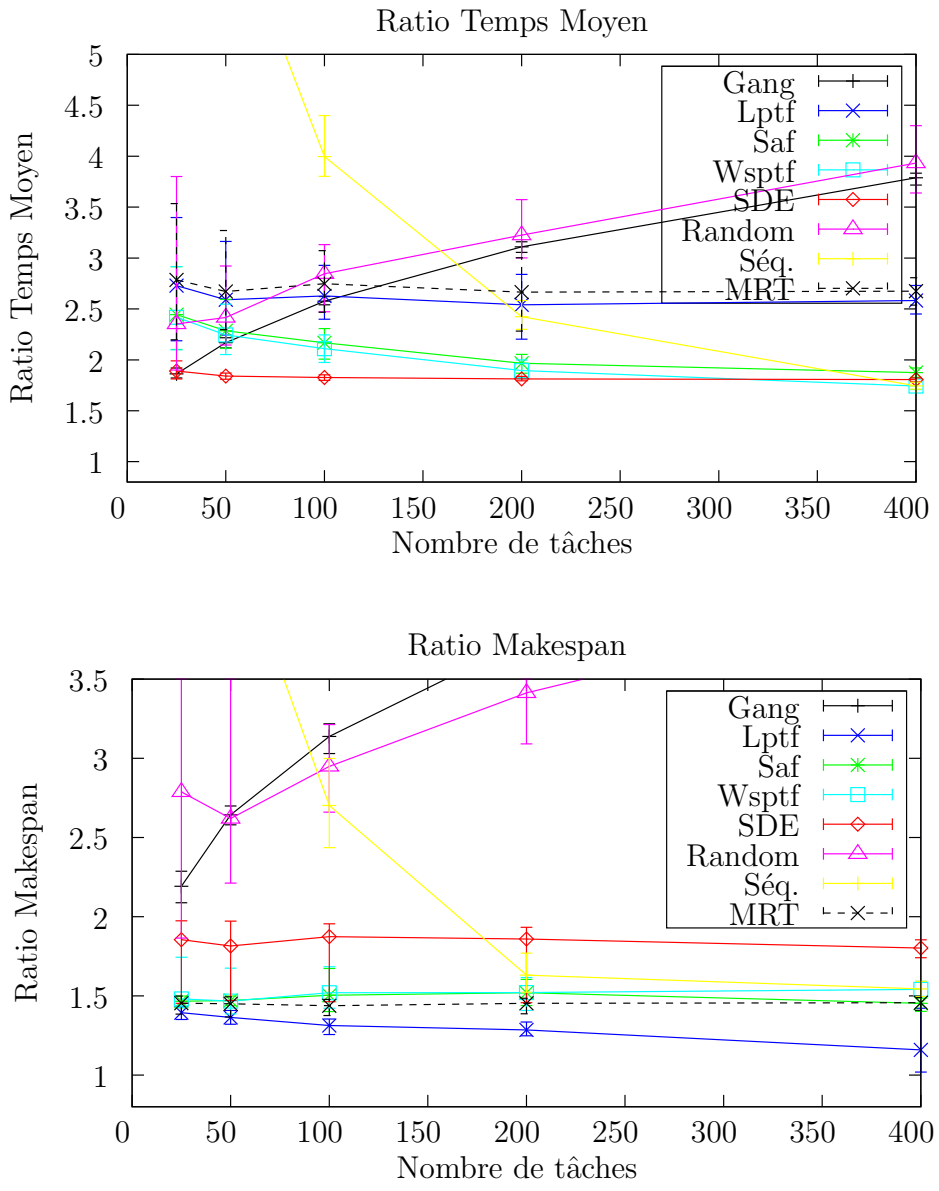


FIG. 2.5 – Ratios de performance pour des temps séquentiels uniformes et des tâches fortement parallèles.

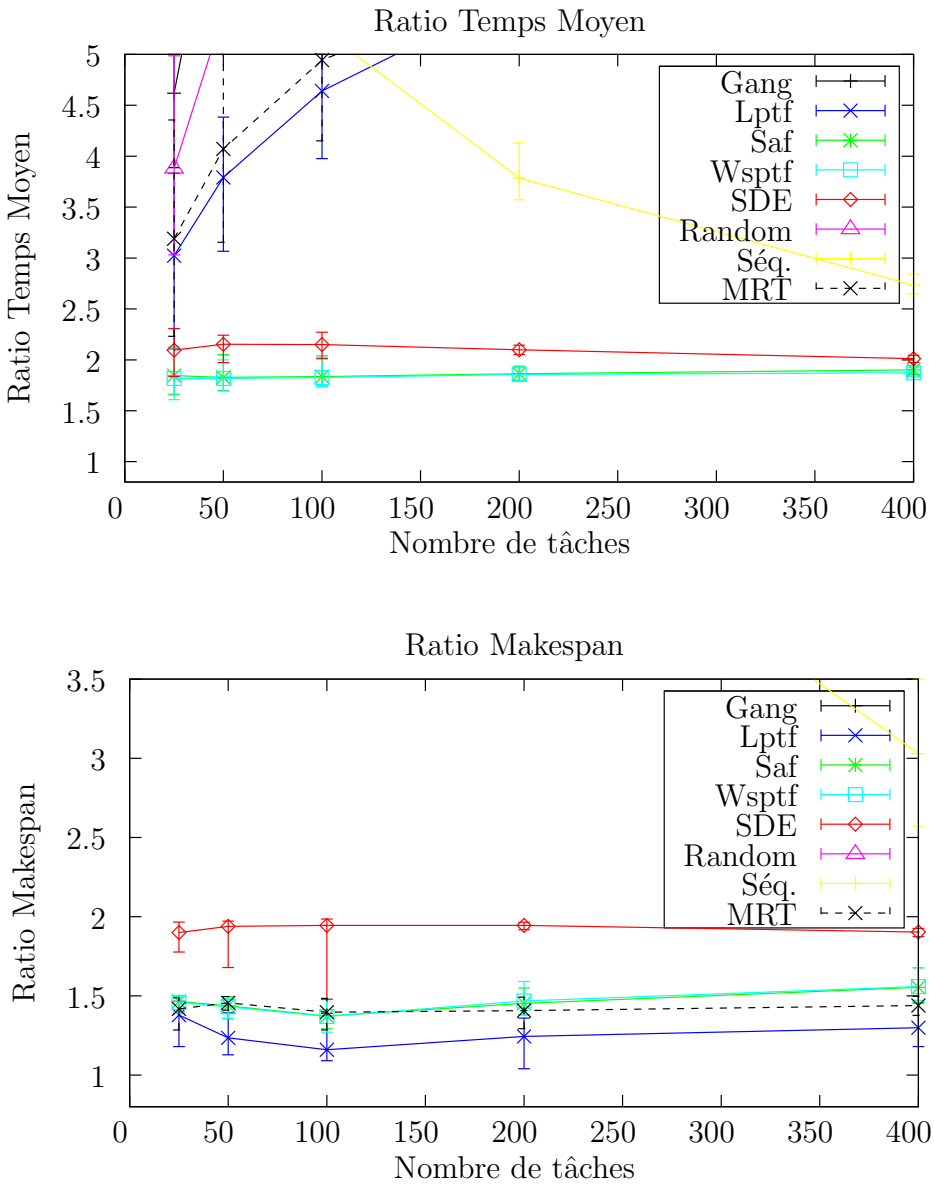


FIG. 2.6 – Ratios de performance pour des temps séquentiels mixtes, les petites tâches étant faiblement parallèles et les grandes fortement parallèles.

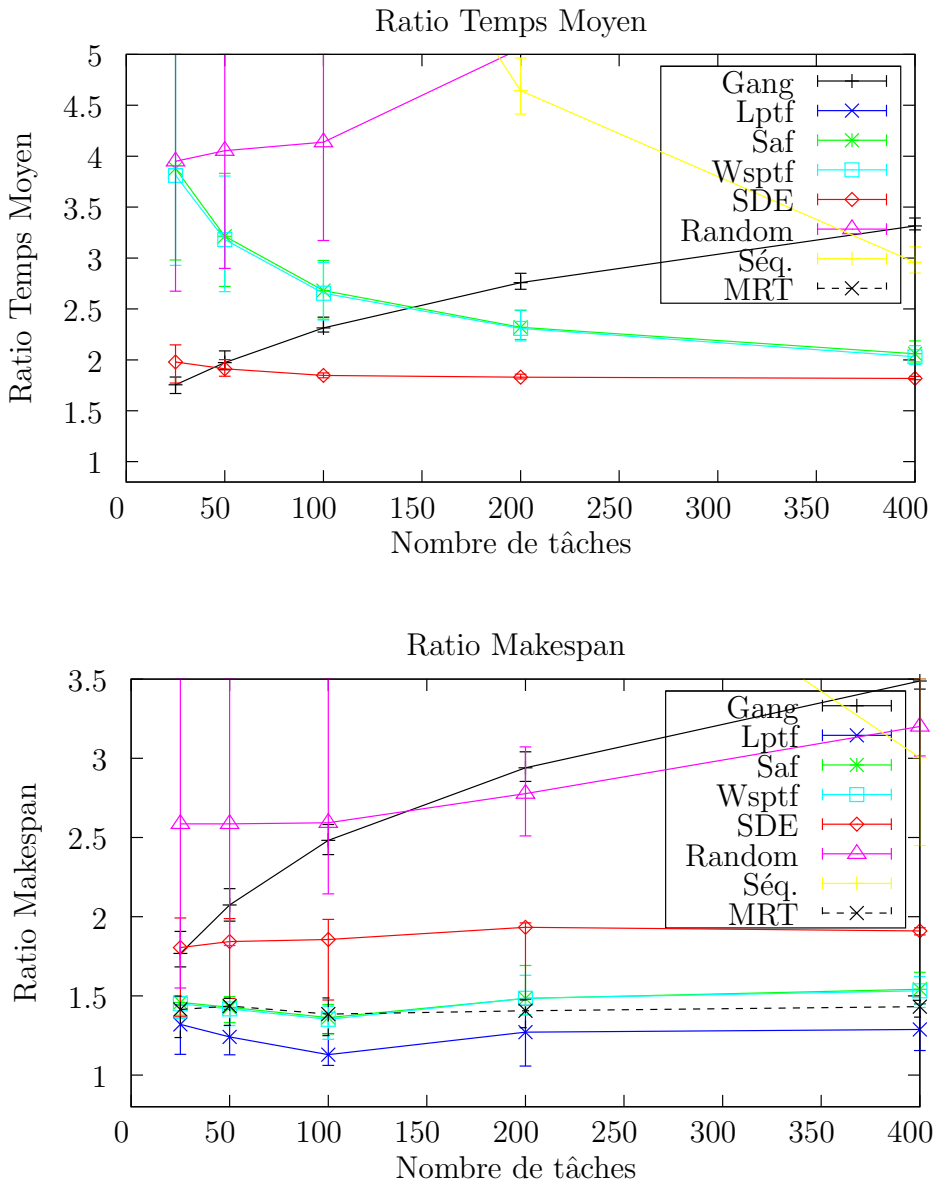


FIG. 2.7 – Ratios de performance pour des temps séquentiels mixtes et des tâches fortement parallèles.

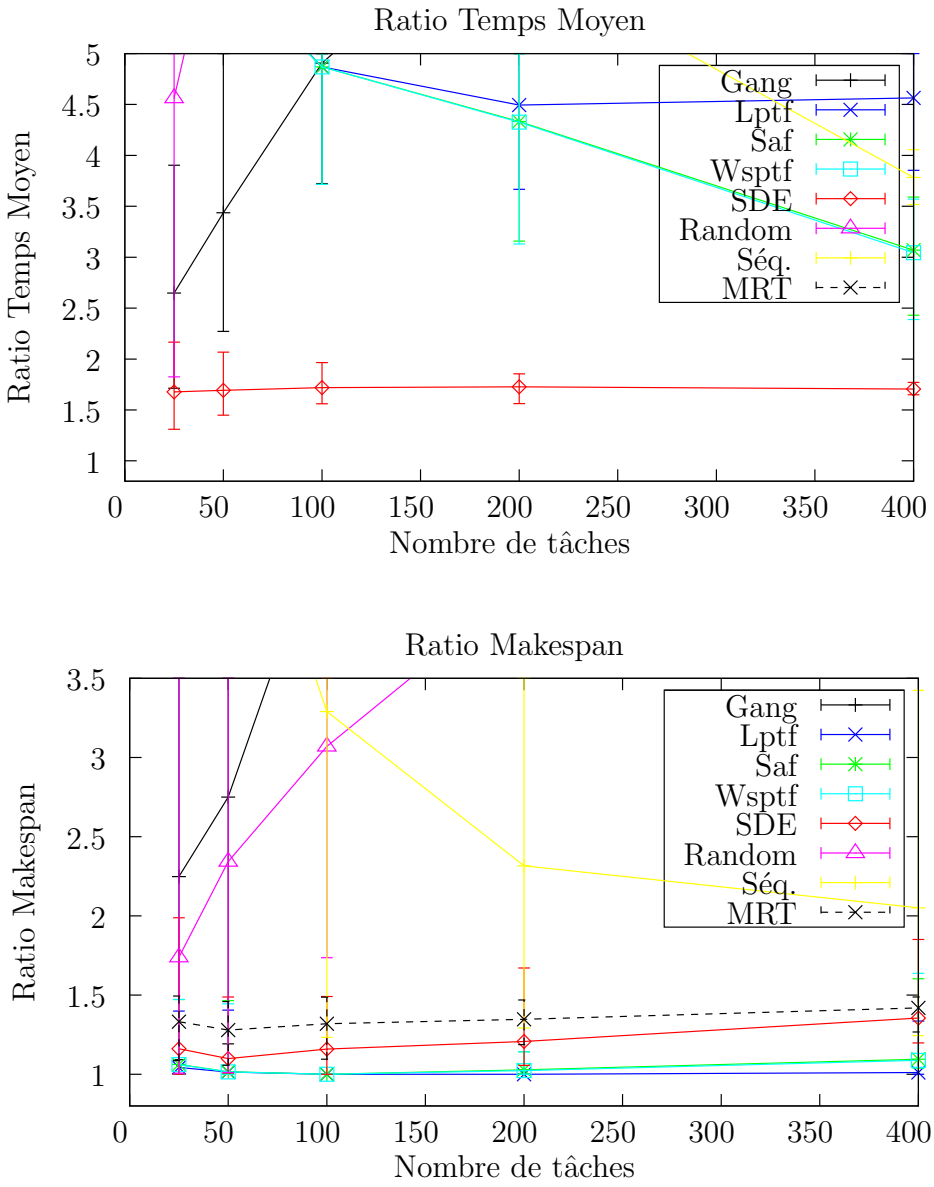


FIG. 2.8 – Ratios de performance pour des temps séquentiels mixtes et le modèle de parallélisme de Cirne et Berman [12].

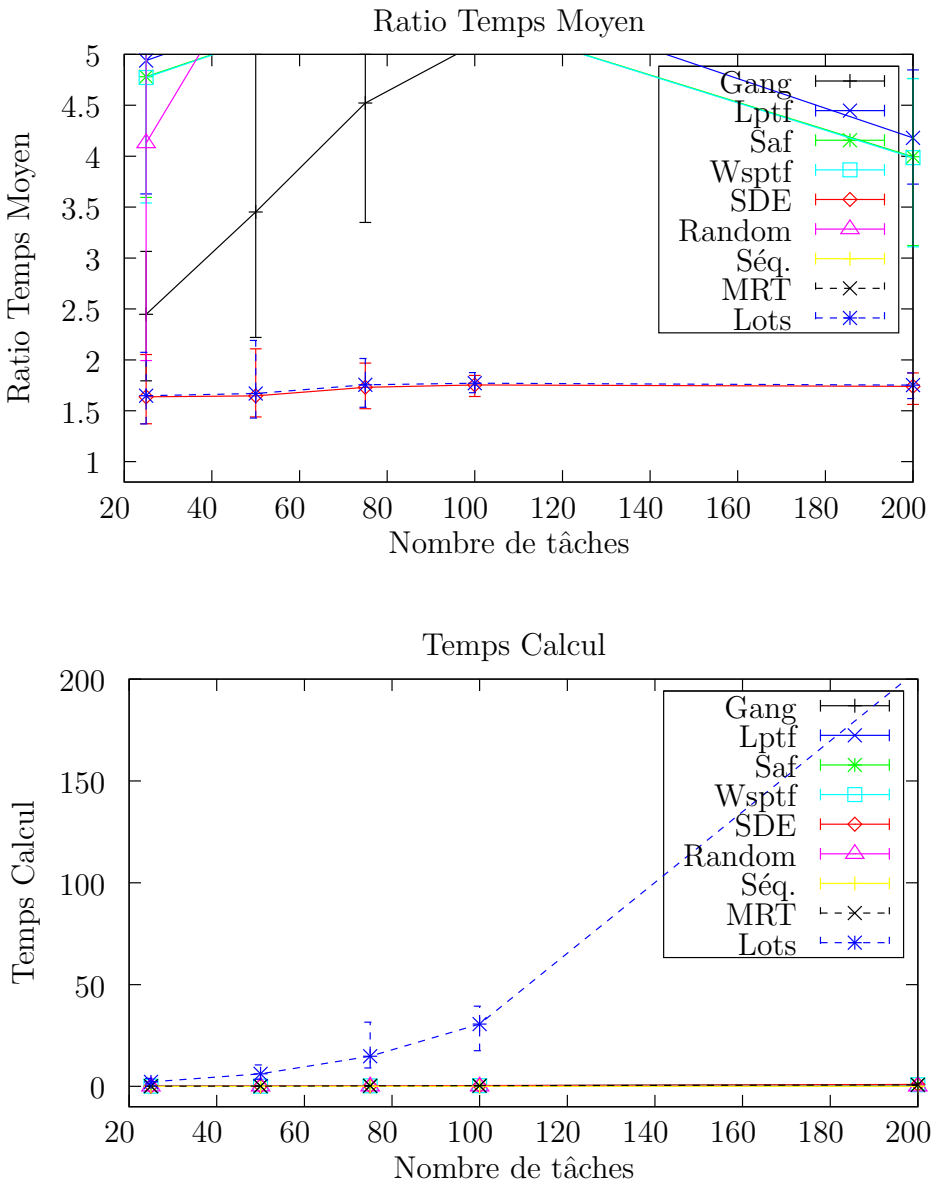


FIG. 2.9 – Ratios de performance pour la somme des temps de complétion et temps d'exécution en secondes avec l'algorithme **Lots**, pour les instances de la figure 2.8

tées, et quel que soit le nombre de tâches. En comparaison, les autres algorithmes ont des performances bien moins stables, meilleures dans certains cas mais bien plus mauvaises dans d'autres. Les ratios obtenus par l'algorithme **SDE** sont entre 2 et $2,5$ pour la $\sum \omega_i C_i$, et d'environ 2 sur le makespan ; cela représente de bons résultats, surtout pour la $\sum \omega_i C_i$ qui est bien plus difficile à optimiser.

On observe également que **LPTF** est pour le makespan un algorithme très performant, qui arrive à être très proche de la borne inférieure dans un grand nombre de cas. **SPTF** est également relativement performant sur le makespan, et obtient de très bons résultats sur la $\sum \omega_i C_i$ pour les instances relativement simples. Pour les instances plus complexes et plus réalistes, comme celle de la figure 2.8 basée sur le modèle de Cirne et Berman, l'allocation utilisée ne lui permet pas d'obtenir de bons résultats. L'approche en étagères se révèle être bien plus robuste, et devrait donc être adoptée dans des cas réels du fait de son insensitivité au comportement des applications tout en gardant un bon comportement sur le makespan. De plus, les simplifications effectuées par rapport à l'algorithme **Lots** permettent d'améliorer grandement le temps d'exécution, sans influencer le comportement en moyenne.

2.5 Synthèse

Dans ce chapitre, nous avons abordé en détail le problème de l'ordonnancement de tâches modelables indépendantes, en s'intéressant à deux critères antagonistes que sont le makespan et la moyenne pondérée des temps de complétion. Nous avons proposé une analyse approfondie d'un algorithme par lots, amélioré par une adaptation de l'algorithme **MRT** d'approximation pour le makespan. Grâce à l'introduction d'un paramètre dans cet algorithme, nous avons pu obtenir une large palette de garanties de performances, à la fois pour les cas hors-ligne et en ligne, ainsi que de meilleures garanties pour l'algorithme randomisé.

Dans un contexte plus pratique, nous avons également proposé une simplification de cet algorithme basée sur une approche en étagères. Ce nouvel algorithme, bien que n'ayant pas de garantie de performance théorique connue, obtient dans une campagne de simulation intensive de très bons résultats en moyenne, équivalents à ceux de l'approche précédente, avec une bonne propriété de stabilité vis-à-vis des diverses instances étudiées. De plus, la simplicité et la plus grande vitesse d'exécution apportées par ces modifications permet d'envisager l'implémentation pratique de cet algorithme **SDE** dans un environnement réel de gestion de ressources pour une grappe. Cette implémentation est présentée dans le chapitre suivant.

Chapitre 3

Approche pratique

Ce chapitre est consacré à la mise en œuvre pratique de l'algorithme bicritère exposé précédemment dans un logiciel conçu pour la gestion de ressources pour un système parallèle de type grappe de PCs. De tels logiciels reposent presque exclusivement, pour résoudre les problèmes difficiles d'ordonnancement, sur des heuristiques intuitives sans réels fondements théoriques. Comme nous l'avons vu dans le chapitre précédent, un certain nombre d'études théoriques récentes ont cependant introduit des modèles spécifiquement adaptés à ces architectures, dont celui des tâches parallèles.

Dans ce chapitre, nous analysons le chemin qui mène des modèles et algorithmes théoriques à leur implémentation dans un environnement réel. Nous présentons en détail les divergences entre les modèles standards et un vrai logiciel de gestion de ressources, et proposons des solutions pour y adapter l'algorithme **SDE** étudié précédemment. Les résultats expérimentaux montrent que cette implémentation obtient des performances comparables à l'heuristique originelle, et bien meilleures sur les instances difficiles. Nous espérons que l'utilisation future de cet algorithme dans un système de production montrera les bienfaits de l'interaction entre la théorie et la pratique.

Cette implémentation a été effectuée au sein du système de gestion de ressources OAR [7, 56]. Ce logiciel, développé au laboratoire ID-IMAG, est basé sur une architecture originale et le choix d'une faible complexité logicielle grâce à des outils de haut niveau. Il offre quasiment toutes les fonctionnalités présentes dans les autres systèmes de gestion de ressources, comme l'ordonnancement avec priorités (via des files de priorités), les réservations, le *backfilling*, et un support pour le calcul distribué à grande échelle (via le concept de tâches *besteffort*). Il est utilisé dans un grand nombre de grappes de calcul scientifique à Grenoble, et est actuellement l'un des blocs de base du projet de grille nationale française Grid'5000 [23].

En utilisant et en interagissant avec un environnement d'ordonnancement réel tel qu'OAR, nous avons vu apparaître des différences conceptuelles avec les modèles théoriques classiques. Ces différences sont souvent assez importantes pour nécessiter de repenser quasiment entièrement les modèles ; ce chapitre n'a donc pas la prétention de donner des réponses et des solutions complètes à ces pro-

blèmes. En revanche, nous essayons d'exposer les raisons de ces divergences, et quelles réponses pratiques peuvent être apportées pour s'adapter à un environnement réel tout en gardant les principes de base des résultats théoriques.

La première section de ce chapitre est consacrée à la description du contexte dans lequel s'est faite cette étude : nous verrons d'une part le fonctionnement et les fonctionnalités d'OAR et nous examinerons d'autre part les divergences que cela implique par rapport aux modèles standards dans les études théoriques. La section suivante (3.2) expose et motive les modifications qu'il a fallu apporter à l'algorithme **SDE** pour l'adapter à cet environnement. Enfin, la section 3.3 présente les expériences effectuées pour valider cette implémentation.

3.1 Contexte pratique

3.1.1 OAR, un système de gestion de ressources

Comme indiqué plus haut, OAR est un système de gestion de ressources destiné à faciliter l'utilisation d'une grappe de PCs. Il a été développé au sein du laboratoire ID, avec comme architectures cibles à la fois les machines de production des laboratoires de physique et de chimie de Grenoble (via le projet CIMENT [11]) et les machines d'expérimentation informatique de Grenoble et plus généralement de France, via le projet Grid'5000 [23]. Cette particularité explique les quelques différences de conception avec d'autres logiciels de gestion de ressources.

Le principe de fonctionnement d'OAR, commun à tous les systèmes de gestion de ressources, est le suivant : il y a un ensemble de nœuds de calcul, qui forment les ressources à partager entre les utilisateurs, et que l'on ne peut utiliser qu'après en avoir obtenu l'autorisation auprès du serveur d'OAR. Ce serveur rassemble toutes les requêtes des utilisateurs et monitore les nœuds de calcul afin de garantir que les ressources allouées à un utilisateur sont bien disponibles. Grâce à l'interface de soumission, il est possible de formuler une grande variété de requêtes pour demander l'accès aux ressources. La méthode simple et classique de le faire consiste à demander l'accès à un certain nombre de nœuds pendant un certain temps ; le serveur central se charge alors d'allouer cet ensemble de ressources le plus tôt possible. Lorsque cette allocation est déterminée et que tous les nœuds correspondants sont disponibles, le système se charge d'exécuter le calcul. Lorsque celui-ci se termine (soit de lui-même soit à la fin du temps imparti de la réservation), les nœuds sont libérés et deviennent disponibles pour d'autres calculs.

Nous faisons ici une liste des fonctionnalités importantes d'OAR d'un point de vue de l'ordonnancement.

Files de priorité

OAR gère la soumission de tâches dans différentes files de priorité. Ces files déterminent un ordre de priorité strict : l'ordonnancement des tâches de forte priorité se fait sans aucune considération des tâches de plus faible priorité, et cet ordonnancement devient ensuite une contrainte pour les files de priorités suivantes.

Grâce à la technique du backfilling, il est cependant possible qu'une tâche moins prioritaire profite de trous dans l'ordonnancement des files plus prioritaires et puisse démarrer plus tôt. La file de plus basse priorité est dénommée *best effort*, et le système est autorisé à arrêter l'une de ces tâches pour libérer des ressources afin de pouvoir exécuter une tâche plus prioritaire.

Grappes de SMPs

OAR supporte les machines composées de nœuds à plusieurs processeurs. Dans ce cas-là, il peut être utile de partager un nœud entre plusieurs tâches, si chacune n'utilise pas tous les processeurs disponibles. OAR intègre donc la notion du nombre de ressources occupées par une tâche ; dans OAR, cela s'appelle le *poids* de la tâche. La tâche utilise donc ce poids sur chacun des nœuds sur lesquels elle est ordonnancée, et il est ainsi possible d'ordonnancer sur un même nœud plusieurs tâches dont le poids total est inférieur ou égal au poids maximum du nœud.

Propriétés

OAR intègre également une notion de *propriétés*, qui permet aux utilisateurs d'exprimer des contraintes sur les nœuds qui peuvent être alloués à chaque tâche. Cette fonctionnalité est très utile pour les utilisateurs, car les nœuds d'un système réel ne sont pas toujours exactement homogènes : certains peuvent avoir plus de mémoire vive, ou des disques durs locaux moins robustes. L'hétérogénéité du réseau de communication entre également en jeu, puisque des nœuds qui sont physiquement connectés sur le même commutateur communiquent plus rapidement. Cependant, cette augmentation de l'expressivité des utilisateurs impose des contraintes assez fortes et non standard sur les algorithmes d'ordonnancement ; nous allons donc détailler ces effets un peu plus loin, dans la section 3.2.4.

Réservations

OAR permet d'effectuer des *réservations* : un utilisateur peut demander d'avoir accès à des nœuds de calcul à une date donnée. Le serveur central d'OAR peut alors refuser ou accepter cette requête. Si elle est acceptée, le système garantit que les nœuds seront disponibles à ce moment là. Cette façon d'obtenir des nœuds est bien différente de la méthode classique, qui consiste à demander un accès à des nœuds le plus tôt possible. Le système de réservations est spécialement utilisé pour permettre la *co-allocation* dans un contexte comme celui de Grid'5000, où l'on cherche à utiliser de façon commune plusieurs grappes distantes. On se heurte en effet à deux exigences contradictoires : l'administration de chaque grappe est faite par le site qui l'accueille (pour des raisons politiques et également pour éviter d'avoir un point central de contrôle), et les utilisateurs veulent pouvoir effectuer des calculs distribués sur plusieurs sites, ce qui nécessite d'avoir accès au même moment à des nœuds qui sont gérés par des systèmes de gestion de ressources différents. Il n'existe pas encore de système réellement au point[1] pour faire interagir

[1] Des mécanismes plus sophistiqués sont actuellement à l'étude, mais la communauté est encore très loin d'un consensus sur la bonne façon d'atteindre l'objectif de co-allocation. Le principe de

différents systèmes de gestion de ressources ; la solution la plus simple consiste alors à effectuer une réservation sur chacun des sites où l'on souhaite effectuer le calcul.

Une autre utilisation de la possibilité d'effectuer des réservations est de permettre d'effectuer des démonstrations du fonctionnement d'une application lors d'une réunion planifiée.

Tâches rigides

OAR implémente le modèle des tâches rigides, et il est impossible de soumettre des tâches modelables. Bien que la plupart des calculs soient intrinsèquement modelables (car la plupart des environnements de programmation ne supposent pas que le nombre de processeurs est connu à l'avance), il est très difficile de fournir au système une estimation du temps de calcul pour chaque allocation possible. La position des concepteurs d'OAR est que trop peu d'utilisateurs seraient prêts à faire cet effort pour que cette fonctionnalité soit prioritaire.

L'ordonnancement dans OAR

L'algorithme d'ordonnancement implémenté dans OAR de façon native est « First Come First Served », ou **FCFS**, avec un backfilling conservatif. Comme mentionné à la section 1.4.1, il s'agit d'un algorithme glouton dont les décisions sont basées sur les tâches, qui considère les tâches dans l'ordre d'arrivée et ordonnance chacune le plus tôt possible étant données les décisions précédentes. Cette approche a l'avantage d'être simple à implémenter, ce qui permet de prendre en compte aisément les contraintes additionnelles de réservations et de propriétés. De plus, le principe est également simple à comprendre pour les utilisateurs de la machine.

Cependant, il est connu que ce genre d'algorithme peut conduire à une faible utilisation de la machine dans des cas dégénérés, et donc à de mauvaises performances. En particulier, il est arrivé que des utilisateurs se plaignent de son manque d'intelligence en ce qui concerne les allocations de ressources : si une tâche arrivée plus tôt peut utiliser n'importe quelles ressources, il est possible que celles qui lui sont allouées soient nécessaires à l'exécution d'une autre tâche, qui dont alors attendre qu'elles soient à nouveau libérées.

3.1.2 Limitations des modèles

Cadre en ligne

Comme pour tous les systèmes de gestion de ressources, l'environnement d'OAR est très dynamique : tous les évènements qui ont lieu après la date d'ordonnancement courante sont complètement inconnus du système. Le système ne peut effectivement pas connaître les tâches qui vont être soumises plus tard, mais il ne sait pas non plus précisément quand vont terminer les tâches en cours d'exécution. En effet, la sémantique du système de soumission est que les utilisateurs demandent l'accès à un certain nombre de nœuds pour un certain temps (ce temps est appelé

réservations, bien qu'imparfait sur bien des plans, a l'avantage d'être simple à mettre en œuvre.

le « *walltime* » de la tâche), mais ne spécifient pas d'estimation du temps réel d'exécution. Si une tâche dure plus longtemps que son walltime, elle est annulée (tous les résultats sont ainsi perdus) afin de pouvoir garantir l'accès à la machine pour les autres tâches en attente. Le walltime est donc une borne supérieure sur le temps réel d'exécution, et le plus souvent une borne très grossière, puisque les utilisateurs ne veulent pas que leurs calculs soient perdus à cause d'une mauvaise estimation[2].

Le cadre d'OAR ne correspond donc pas au modèle clairvoyant, pas assez réaliste, qui suppose que les temps d'exécution des tâches sont connus de façon précise. En revanche, les modèles non clairvoyants sont trop pessimistes et supposent que l'on n'a absolument aucune information sur ces temps d'exécution. Ils sont donc bien adaptés à un environnement de type station de travail, mais pas vraiment pour les grappes de calcul. On se trouve donc dans un cadre intermédiaire entre les deux extrêmes théoriques que sont les modèles clairvoyant et non-clairvoyant. Quelques études récentes [31] ont porté sur la *robustesse* des algorithmes d'ordonnancement à des modifications des données du problèmes, et pourraient être intéressantes dans ce contexte ; mais elles supposent classiquement que ces modifications ne sont pas trop grandes.

Dans OAR, ce cadre dynamique est géré de manière conservative : à chaque instant, toutes les décisions sont prises comme si l'information disponible était exacte. Quand un évènement non prévu a lieu (la soumission d'une nouvelle tâche ou une tâche qui termine avant son walltime), l'algorithme d'ordonnancement est relancé avec en entrée le nouvel état du système ; bien entendu, les tâches en cours d'exécution ne sont pas ré-ordonnancées. Ce comportement implique que les décisions d'ordonnancement concernant le futur ne sont que des prévisions. Bien qu'elles soient disponibles pour fournir des informations aux utilisateurs, elles changent régulièrement avec l'état du système. Cependant, comme l'algorithme d'ordonnancement est **FCFS**, ces changements ne peuvent pas retarder les dates d'exécution prévues des tâches.

Ordonnancement en présence de réservations

D'un point de vue théorique, la présence de réservations ajoute des contraintes dans le problèmes d'ordonnancement, et le rend alors beaucoup plus difficile. Les études existantes sont plutôt motivées par des périodes d'indisponibilité des machines, par exemple pour des raisons de maintenance, mais cela revient finalement au même. Lorsque l'on rajoute ces contraintes, même des problèmes très simples, comme minimiser le makespan sur une seule machine, deviennent NP-difficiles. C'est pour cette raison que l'indisponibilité des machines n'a quasiment été étudiée que dans des modèles assez simples. Il est en effet courant de considérer des modèles où la préemption est autorisée, ou avec une seule machine, ou avec un petit nombre de périodes d'indisponibilité. Un survol de ces études est donné dans [40] ; cependant, aucune étude n'a été faite dans le cadre des tâches parallèles. Nous reviendrons sur l'étude des réservations dans ce cadre dans le chapitre 4.

[2]Dans cette optique, il serait intéressant d'étudier l'effet d'une politique d'ordonnancement favorisant les tâches qui ont un petit walltime sur les habitudes de soumissions des utilisateurs.

Remarques sur l'hétérogénéité

La présence du concept de « propriétés » change profondément le modèle de plate-forme : avec OAR, comme la plupart des systèmes de gestion de ressources destinés à des grappes, il est possible de distinguer deux nœuds de calcul.

Du point de vue de l'utilisateur, cette différentiation des nœuds permet d'exprimer, via un prédicat booléen, le sous-ensemble des nœuds qui sont candidats pour exécuter son calcul. Le gestionnaire sélectionne alors parmi ces candidats les nœuds qu'il va réellement lui allouer. Cette sélection est désirable à cause de différences physiques qui peuvent exister entre les nœuds d'une même grappe (mémoire disponible, vitesse du disque dur local, carte réseau, etc.). L'hétérogénéité est également présente sur le réseau d'interconnexion : dès qu'une grappe est d'une taille raisonnable, il est impossible de connecter physiquement tous les nœuds sur un seul commutateur. La latence entre deux nœuds connectés sur des commutateurs différents est alors un petit peu plus élevée (on atteint là la limite du modèle de plate-forme « entièrement connectée »). Cette différence est insensible pour la plupart des applications, mais celles qui font un usage intensif du réseau (comme en particulier des tests de performance d'algorithmes de transmission de données) ont besoin d'être ordonnancées sur des nœuds qui partagent le même commutateur. Dans un environnement de développement informatique comme Grid'5000, ce genre d'applications est bien plus présent que sur une grappe de production.

Dans les modèles théoriques classiques, l'hétérogénéité est presque toujours vue comme une différence de vitesse de calcul. Les trois modèles les plus connus sont ainsi le modèle *homogène* dans lequel tous les nœuds sont identiques, le modèle « *related* » dans lequel chaque nœud a une vitesse différente, et le modèle « *unrelated* », le plus général, dans lequel la vitesse d'exécution dépend à la fois du nœud et de la tâche à exécuter. Dans tous les cas, on ne prend en compte l'hétérogénéité qu'en termes de vitesse à laquelle les machines peuvent effectuer des calculs. En contraste, dans un système de gestion de ressources, il y a une hypothèse implicite que tous les nœuds calculent à la même vitesse[3], et la notion de propriétés qui restreint l'ensemble des machines sur lesquelles une tâche donnée peut être exécutée. Bien sûr, il serait possible d'utiliser le modèle « *unrelated* », qui est le plus général, pour prendre cela en compte : il suffit d'affecter un temps de calcul très grand à une combinaison (tâche, nœud) interdite. Mais ce modèle est bien trop général et trop complexe pour que l'on arrive à concevoir des algorithmes efficaces ; il n'est même pas clair que l'on puisse définir un modèle de tâches parallèles sur une architecture « *unrelated* ».

D'autres modèles traitant de différentes versions de l'hétérogénéité ont été étudiés, de manière plus anecdotique. Par exemple, le modèle set_j [13], qui a été introduit à la même époque que celui des tâches modelables, en est un peu une généralisation dans un cadre hétérogène. Dans ce modèle, une tâche est spécifiée avec l'ensemble des allocations qu'il est possible de lui affecter, ainsi que son temps d'exécution pour chacune de ces allocations. Il s'agit donc d'une façon possible de modéliser les propriétés d'OAR ; cependant elle ne permet pas d'exprimer de façon

[3] En effet, le walltime d'une tâche, qui est l'information la plus proche de son temps d'exécution, est donné indépendamment des nœuds sur lesquels elle va être ordonnancée.

concise les requêtes du type : « cette tâche peut s'exécuter sur 10 nœuds parmi ces 30 nœuds-là », qui sont les requêtes les plus courantes dans un environnement comme OAR.

Un autre modèle plus simple est le modèle hiérarchique, étudié par exemple dans [4], qui s'intéresse à des plate-formes dans lesquelles la communication s'effectue de manière hiérarchisée : l'hétérogénéité est donc vue comme une différence de structure plutôt que de vitesse de calcul. Les architectures ciblées par ce modèle sont par exemple des grappes composées de nœuds multi-processeurs, ou bien des grilles composées de plusieurs grappes ; ce sont des architectures composées d'un regroupement d'éléments parallèles dans lesquelles la communication à l'intérieur d'un élément est bien plus rapide que la communication entre deux éléments. Ce pourrait être un modèle bien adapté pour décrire la communication dans une grappe, mais il n'est pas vraiment assez général pour englober tout ce que le système de propriétés d'OAR peut exprimer ; de plus il n'existe pour l'instant que peu de travaux dans ce modèle.

Enfin, le modèle étudié dans [41] est un modèle « *related* » avec *disponibilités restreintes*, c'est-à-dire que chaque tâche ne peut être exécutée que sur un sous-ensemble des nœuds. Cela correspond très bien au modèle d'OAR ; cependant cet article s'intéresse à un type de tâches particulier et très simple, qui autorise la résolution du problème dans le cadre « *unrelated* » plus général. La spécificité de ce modèle d'architecture n'y est donc pas réellement étudiée.

Encore une fois, le modèle de plate-forme des environnements de grappes est intermédiaire entre les deux modèles classiques extrêmes que sont le modèle homogène et « unrelated », et ne correspond pas vraiment à aucun autre modèle plus exotique déjà étudié.

3.2 De la théorie à la réalité

Dans cette section, nous décrivons les choix concrets que nous avons dû faire pour adapter l'algorithme **SDE** à l'environnement OAR. Nous espérons ainsi montrer qu'il est effectivement possible d'implémenter un algorithme sophistiqué à un cadre réel tout en gardant son efficacité, malgré toutes les différences avec les modèles de départ.

3.2.1 Adaptation au cadre dynamique

L'algorithme **SDE** a été pensé aussi bien pour le cadre en ligne qu'hors ligne. Cependant, la structure même de l'algorithme, en étagères de tailles croissantes, suppose qu'il y a une « origine », un instant où tout commence. Il serait impraticable de l'utiliser tel quel dans un environnement réel, dont la durée de vie est de l'ordre de trois ans : les étagères deviendraient très vite beaucoup trop grandes, sans aucun rapport avec la longueur des tâches. D'un autre côté, il ne serait pas non plus intéressant de recommencer l'algorithme avec des petites étagères à chaque fois qu'une tâche est soumise, car cela aurait pour effet de favoriser grandement les petites tâches. En effet, aucune tâche longue ne pourrait s'exécuter tant qu'il

reste des petites tâches : on rencontrerait alors nécessairement des scénarios de famine des grandes tâches.

Pour garder la structure en étagères de l'algorithme, nous avons décidé de prendre une approche intermédiaire en remettant périodiquement les longueurs des étagères à zéro. On rajoute ainsi artificiellement des « origines des temps », le principe étant de les faire coïncider avec les débuts des périodes de travail. Ainsi, l'algorithme **SDE** repart avec des petites étagères tous les jours au début de la matinée et de l'après-midi. Entre deux redémarrages, l'ordonnancement est fait comme dans l'algorithme **SDE** original, et réagit à la soumission d'une tâche en l'insérant dans la structure en étagères au moment où elle est soumise. L'idée est d'essayer de favoriser les petites tâches pendant la journée, lorsque l'interactivité est plus importante, et d'exécuter les tâches plus longues pendant la nuit.

3.2.2 Tâches rigides

L'algorithme **SDE** a été conçu pour ordonnancer des tâches modelables, avec l'idée que ce modèle serait supporté par les gestionnaires de ressources futurs. La possibilité de changer l'allocation des tâches permet de garantir qu'il n'y a pas une trop grande différence de longueur entre les tâches contenues dans une étagère : si une tâche est trop courte, on peut lui allouer moins de processeurs pour qu'elle rentre mieux dans l'étagère ; les processeurs ainsi libérés peuvent être utilisés par d'autres tâches, et on évite d'avoir de trop gros « trous » dans l'ordonnancement.

Le cadre original d'ordonnancement par lots de tailles croissantes est cependant très générique, et peut être adapté à de nombreux modèles de tâches différents. De la même manière, il est possible d'utiliser l'algorithme **SDE** pour des tâches rigides, en court-circuitant la phase d'allocation qui n'a plus lieu d'être. On espère alors que la phase de compaction permettra de retrouver une bonne utilisation, même avec des étagères moins bien remplies.

Notons cependant qu'il est prévu que la prochaine version d'OAR puisse gérer les tâches modelables. Notre algorithme sera alors naturellement approprié dans un cadre où des tâches rigides coexistent avec des tâches modelables, et ne nécessitera que très peu de modifications.

3.2.3 Gestion des réservations

La possibilité pour les utilisateurs de faire des réservations à l'avance rajoute encore une contrainte à l'algorithme d'ordonnancement. Pour l'algorithme **SDE**, cela implique qu'il est possible que certains nœuds ne soient disponibles que durant une partie du temps que dure l'étagère. Choisir un ensemble de tâches qui peuvent s'exécuter ensemble dans l'étagère peut alors devenir très difficile ; or on ne veut pas que l'implémentation ait une complexité trop élevée.

L'avantage principal d'ordonnancer les tâches par étagères vient du fait que, lors de la phase de sélection, la dimension temporelle n'est pas prise en compte : on ne choisit les tâches qu'en fonction des processeurs qu'elles utilisent, et non en fonction de combien de temps elles vont les utiliser. Cela permet de baisser considérablement la complexité de l'algorithme, et il semble donc important de

conserver cette propriété. C'est pourquoi nous avons décidé de considérer pendant cette phase qu'un nœud n'est utilisable que s'il est disponible durant toute la longueur de l'étagère, les autres nœuds étant considérés absents. Cela revient finalement à étendre les réservations pour que leurs frontières coïncident avec les débuts et fins d'étagère. Ainsi, nous pouvons encore garantir que les tâches sélectionnées pourront être ordonnancées toutes en même temps dans l'étagère.

Cette limitation n'est en place que pendant la phase de sélection. Lors de la compaction, qui utilise un algorithme plus simple de « backfilling » conservatif pour ordonnancer les tâches plus tôt lorsque c'est possible, les réservations sont considérées avec leurs dates originelles. Cependant, il faut noter qu'avec la présence de réservations, le résultat de la sélection pour une étagère donnée dépend de la date de début de cette étagère. Si les tâches sélectionnées pour l'étagère précédente sont toutes sensiblement plus courtes que la longueur théorique de cette étagère, on peut savoir dès la phase de sélection suivante que toutes les tâches vont pouvoir commencer plus tôt. Nous avons donc choisi d'effectuer une phase de compaction juste après chaque phase de sélection. Cela ne modifie pas le comportement de l'algorithme lorsqu'il n'y a pas de réservations, mais permet de fixer la date de début de chaque étagère le plus tôt possible, et donc de faire en sorte que les nœuds disponibles considérés lors de la phase de sélection soient le plus proche possible des nœuds qui seront utilisés ensuite par la phase de compaction.

3.2.4 Plate-forme non homogène

La plus grande difficulté rencontrée lors de l'implémentation de **SDE** a été le modèle d'hétérogénéité de OAR. C'est encore une fois la phase de sélection qu'il a fallu modifier, mais cette fois de manière bien plus profonde. En effet, le concept de « propriétés » rajoute des contraintes qui font que l'on ne peut plus considérer toutes les machines comme indiscernables ; or l'algorithme de sac-à-dos utilisé dans la phase de sélection utilise fortement cette hypothèse pour pouvoir obtenir une solution optimale avec une complexité raisonnable.

Dans un tel contexte, même la formulation du problème que la phase de sélection doit résoudre est délicate. Formellement, les entrées du problème sont :

- un ensemble $\mathcal{N}$ de nœuds, chaque nœud n_i ayant un nombre de processeurs disponibles π_i,
- un ensemble $\mathcal{T}$ de tâches, chaque tâche t_j étant représentée par un nombre q_j de nœuds requis, un nombre w_j de processeurs requis par nœud, un ensemble $\mathcal{P}_j \subseteq \mathcal{N}$ de nœuds autorisés, et un poids ω_j.

Le but est de fournir un sous-ensemble $\mathcal{S}$ de $\mathcal{T}$ et une fonction d'allocation $\sigma : \mathcal{S} \to \mathcal{P}(\mathcal{N})$ tels que :

- on ait alloué à chaque tâche sélectionnée le bon nombre de nœuds parmi les nœuds autorisés :

$$\forall t_j \in \mathcal{S}, \quad \sigma(t_j) \subseteq \mathcal{P}_j \text{ et } |\sigma(t_j)| = q_j$$

- La somme des processeurs utilisés par les tâches alloués à un nœud donné

ne dépasse pas la capacité de ce nœud :

$$\forall n_i \in \mathcal{N}, \quad \sum_{j \in B_i} w_j \leq \pi_i \text{ où } B_i = \{j \in \mathcal{S} \,|\, n_i \in \sigma(t_j)\}$$

Il faut de plus que le poids total de ce sous-ensemble de tâches sélectionnées soit le plus grand possible, c'est-à-dire que l'on veut maximiser $\omega(\mathcal{S}) \equiv \sum_{t_j \in \mathcal{S}} \omega_j$.

C'est un problème très difficile : il contient le problème du sac-à-dos quand il n'y a qu'un seul nœud, mais il peut également être vu comme une instance de SetPacking [24] lorsque tous les π_i sont égaux à 1. Le problème SetPacking est NP-difficile au sens fort, contrairement au sac-à-dos (qui est NP-complet au sens faible).

Bien que ce problème soit trop difficile pour être résolu dans le cas général, les cas pratiques que l'on rencontre dans OAR sont assez spécifiques, et il y a souvent de faibles différences, quand il y en a, entre les $\mathcal{P}_j$. En effet, la plupart des utilisateurs n'expriment aucune contrainte spéciale sur les nœuds que peuvent utiliser leurs tâches, et ceux qui le font expriment les mêmes contraintes pour toutes les tâches. Il est donc possible de considérer comme indiscernables les nœuds qui apparaissent toujours ensemble dans les $\mathcal{P}_j$, ce qui permet de former des *groupes* de nœuds indiscernables. De cette façon, la taille de l'espace de recherche est grandement réduite, et on peut appliquer un algorithme de programmation dynamique pour le résoudre de manière exacte dans ces cas simples. Bien entendu, la taille de l'espace d'états et la complexité de l'algorithme sont exponentiels en le nombre de groupes différents, ainsi qu'en le plus grand π_i.

Ces valeurs étant souvent faibles, cette implémentation est très efficace dans la plupart des cas. Bien sûr, il arrive quand même que l'on rencontre des instances difficiles, pour lesquelles le temps d'exécution est prohibitif. Pour pouvoir y répondre quand même, cette procédure de sélection est automatiquement remplacée par un algorithme glouton quand la taille du problème est trop élevée (c'est-à-dire quand il y a trop de groupes), ou quand l'algorithme par programmation dynamique dépasse un certain temps.

3.3 Analyse expérimentale

3.3.1 Description de l'environnement

Pour analyser et valider notre implémentation de l'algorithme **SDE**, nous avons décidé de rejouer des instances obtenues à partir de traces récoltées sur des grappes de PCs gérées par OAR. Bien qu'il existe des travaux récents [49] qui proposent des modèles de génération de traces synthétiques, l'utilisation de ces traces réelles permet d'avoir des instances qui contiennent toutes les spécificités qu'OAR permet d'exprimer, et également de valider l'implémentation dans le contexte même où elle est amenée à être utilisée. Il aurait en effet été difficile de concevoir un modèle de génération aléatoire d'instances qui spécifie des propriétés pour les tâches d'une manière assez réaliste pour que la validation ait un sens.

Comme OAR est le gestionnaire utilisé sur toutes les grappes du projet national Grid'5000 [23], nous avons accès à plusieurs traces obtenues sur chacune des

grappes. Cependant, et bien que Grid'5000 soit utilisé par une grande partie de la communauté du calcul parallèle française, il n'y a sur la plupart des grappes qu'une faible compétition pour les ressources, et les traces correspondantes ne fournissent pas des instances très intéressantes pour notre propos. La seule exception notable est le Icluster2 [30], composé de 104 nœuds bi-Itanium 2 ; il a en effet été opérationnel plus tôt que les autres grappes et possède donc une plus grande base d'utilisateurs. Icluster2 a donc connu un certain nombre de périodes de haute activité qu'il est alors intéressant de rejouer pour étudier quel peut être l'effet de l'utilisation d'un autre algorithme d'ordonnancement, et quelles sont les performance de notre implémentation de **SDE**.

3.3.2 Simulations

Nous avons développé un simulateur à évènements qui a un comportement *strictement* équivalent à celui d'OAR dans la même situation. En particulier, l'interface avec le système d'ordonnancement est la même, ce qui permet d'utiliser exactement la même implémentation des algorithmes d'ordonnancement que dans OAR. Pour évaluer notre algorithme, nous avons comparé le résultat à l'ordonnancement produit par l'algorithme **FCFS** originel. Nous avons donc rejoué, avec chacun des deux algorithmes, plusieurs scénarios de soumission de tâches qui ont eu lieu sur la grappe[4]. Le résultat de chaque simulation est un ordonnancement qui spécifie pour chaque tâche sa date de début et les machines qui lui ont été affectées. On peut alors étudier pour chacune de ces tâches la différence entre ces dates de début pour les deux algorithmes.

Remarquons cependant que les traces de soumission originales ont été obtenues dans un environnement qui utilisait l'algorithme **FCFS**, et avec lequel les utilisateurs ont accès non seulement à l'état courant d'occupation de la machine, mais également aux prédictions concernant les décisions futures d'ordonnancement. Comme il est courant que les tâches soumises soient adaptées à l'état de la machine au moment de la soumission, on peut s'attendre à ce que les résultats des simulations soient en faveur de **FCFS**, qui va recréer un ordonnancement similaire à l'original et donc bien adapté.

La table 3.1 résume les résultats de ces simulations, et donne pour chaque instance le nombre total de tâches ainsi que le nombre de tâches *identiques*, qui démarrent exactement au même moment avec les deux algorithmes. Ces tâches sont souvent très courtes ou très petites, et démarrent juste au moment où elles sont soumises ; elles ne sont donc pas vraiment représentatives de l'intelligence de l'algorithme d'ordonnancement. Nous avons donc retiré ces tâches du reste de l'analyse, pour insister plus sur les différences entre les algorithmes. La table 3.1 donne également la différence totale des temps de démarrage de toutes ces tâches *différentes*, avec la moyenne et l'écart-type. Les figures 3.1 et 3.2 présentent les

[4]Nous n'avons pas réutilisé l'ordonnancement effectué dans la réalité par OAR, parce qu'il est impossible de soumettre notre algorithme aux mêmes conditions de façon exacte. En effet, pour certains évènements, notamment les pannes de machines, l'information de la date à laquelle ils ont lieu n'est pas présente dans les traces, puisque ces pannes ne sont détectées par OAR que lorsqu'une tâche commence ou termine son exécution. On ne peut donc pas vraiment savoir ce qui se serait passé avec un autre algorithme.

Instance	Nb tâches	Identiques	Diff. totale	Diff. moyenne	Écart-type
3.1(a)	1391	362	113133	109.945	117.073
3.1(b)	1074	749	1824.79	5.61475	23.1384
3.1(c)	575	221	121.904	0.344362	10.6313
3.2(a)	429	364	-94.8261	-1.45886	25.825
3.2(b)	1292	757	-757.596	-1.41607	11.7721
3.2(c)	239	198	-93.0825	-2.2703	16.1927

TAB. 3.1 – Résultats numériques des simulations. Les temps sont donnés en heures ; une valeur est positive si la tâche a été ordonnancée plus tôt avec **SDE** qu'avec **FCFS**.

histogrammes de chaque distribution correspondant à chaque instance, ainsi que la moyenne (représentée par un petit segment sous l'histogramme) et les quartiles (représentés par les lignes en pointillés). Dans les deux cas, une tâche pour qui la différence est positive a été exécutée plus tôt avec **SDE** qu'avec **FCFS**.

La première observation que l'on peut faire sur ces résultats est que l'écart-type est relativement grand. Cela s'observe également sur les histogrammes, où l'on voit que la distribution est assez large, et que la différence des temps de démarrage de quelques tâches est très élevée en valeur absolue. Que ce soit en positif ou en négatif, cela montre que ces tâches ont été exécutées très tard par un des algorithmes, alors qu'il était possible de les exécuter bien plus tôt. Cela arrive aux très grandes tâches, qui durent plusieurs jours, et ont un walltime d'une ou deux semaines. Il arrive que de telles tâches puissent être exécutées dès leur soumission, mais qu'un délai, même petit, les force à être exécutées après une réservation faite environ une semaine à l'avance. En changeant d'algorithme, il arrive donc que la date de début de ces tâches soit grandement modifiée.

Pour la dernière moitié de ces expériences, la moyenne des différences est autour de -2 heures, ce qui montre que **SDE** un comportement moins performant en terme de moyenne que **FCFS**. Cependant, la médiane est dans ces cas-là positive, ce qui veut dire que **SDE** améliore la performance d'une majorité des tâches, et en contrepartie pénalise beaucoup une minorité ; au total la tendance est plutôt à une moins bonne performance. Dans l'expérience 3.1(c), on assiste au phénomène inverse, avec quelques tâches très avantagées par **SDE**, ce qui donne une moyenne positive avec une médiane négative.

Les deux premières expériences font exception, puisqu'elles ont à la fois une moyenne et une médiane positive. Ces expériences correspondent à des instances plus difficiles, avec en particuliers plus de tâches qui ont des contraintes de propriétés. Cela montre que la gestion plus intelligente que fait l'algorithme **SDE** de ces propriétés permet d'améliorer notablement les performances de l'ordonnancement produit.

De plus, l'expérience 3.1(a) est très favorable envers **SDE**, et on peut attribuer cela à deux facteurs supplémentaires. Premièrement, la charge de la grappe est très importante dans cette instance, ce qui entraîne des temps d'attente très élevés

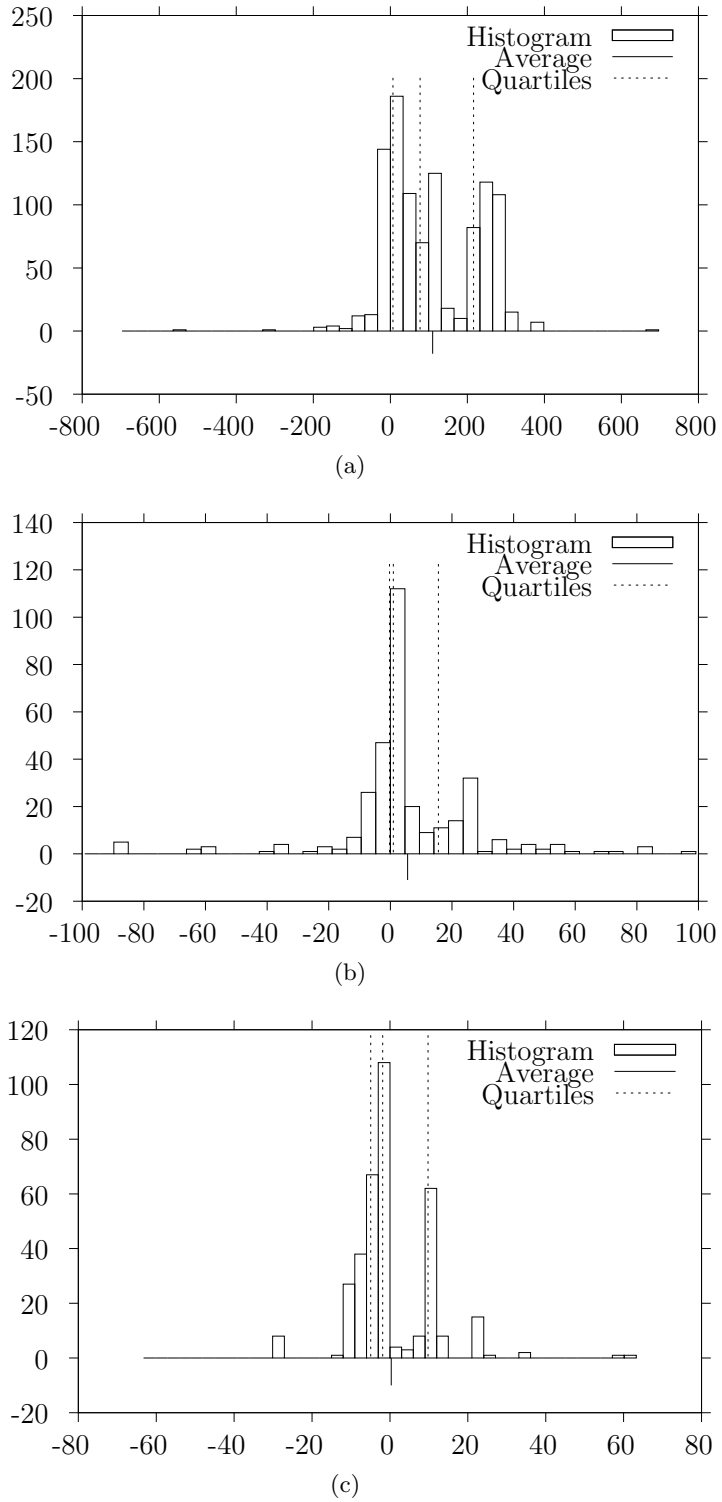
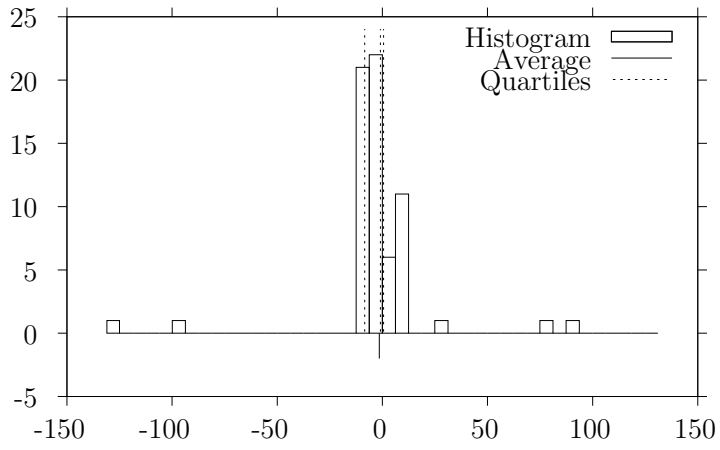
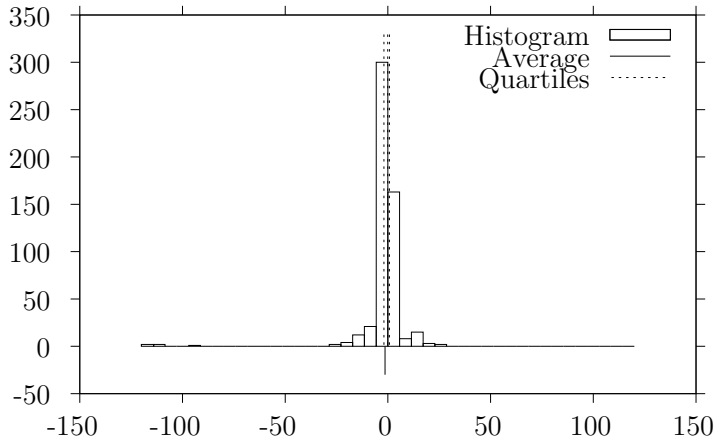


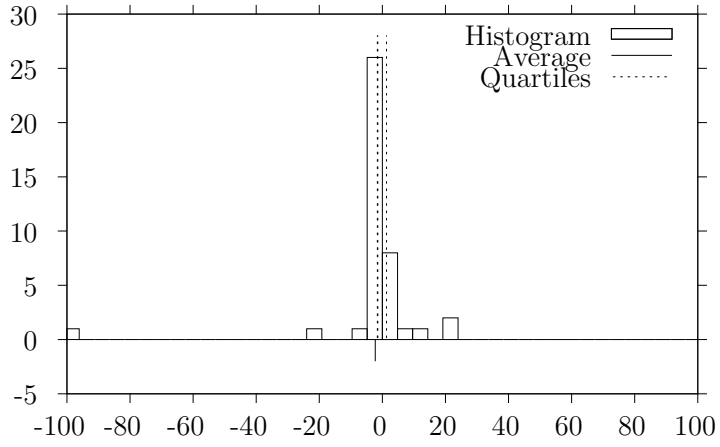
FIG. 3.1 – Ces histogrammes montrent la distribution des différences de temps. Les différences sont en abscisse (en heures), en ordonnées se trouvent le nombre de tâches qui ont été avancées ou retardées de la valeur correspondante. Une valeur est positive si la tâche a été ordonnancée plus tôt avec **SDE** qu'avec **FCFS**.



(a)



(b)



(c)

FIG. 3.2 – Suite de la figure 3.1.

pour les tâches, et donc plus de liberté pour que l'ordonnanceur puisse prendre des décisions intelligente. Deuxièmement, cette instance vient de la partie la plus ancienne des traces du Icluster2, à une époque où l'implémentation de l'algorithme **FCFS** n'était pas exactement la même que celle d'aujourd'hui. Le biais envers **FCFS** qui se retrouve dans les autres instances n'est donc plus présent, puisque les soumissions ont été faites dans un cadre un petit peu différent. Dans ce contexte, notre implémentation de **SDE** a de bien meilleures performances que **FCFS**, qu'elle atteint principalement en retardant quelques grandes tâches pour faire de la place à certaines plus petites ou plus contraintes, et également en réordonnant les tâches pour que les tâches qui ont des propriétés compatibles soient ordonnancées ensemble.

La prochaine étape pour valider cette implémentation sera de la mettre en place sur une plate-forme en cours d'utilisation. Cela permettra de voir comment les utilisateurs réagissent à cette nouvelle politique d'ordonnancement, et peut-être d'effectuer des tests avec un biais dans l'autre sens, pour mieux apprécier l'importance de ce biais.

Nous pensons également convertir des traces classiques provenant de grappes qui ne sont pas gérés par OAR [15, 49] pour pouvoir comparer la performance des deux algorithmes sur ces instances. Cependant, l'infrastructure des plates-formes où ont été collectées ces traces est différente (il n'y a en particulier pas de réservations ni de propriétés), et ces expériences ne permettront pas de mettre en évidence toutes les caractéristiques des algorithmes.

3.4 Synthèse

Le travail présenté dans ce chapitre a permis de décrire en détails les divergences conceptuelles qui peuvent exister entre les modèles classiques d'ordonnancement, même ceux qui sont spécialement adaptés aux grappes, et la réalité d'environnements réels de gestion de ressources. Ce travail a également débouché sur l'implémentation dans un tel environnement d'un algorithme basé sur des fondements théoriques solides. Les résultats sont très encourageants, et permettent d'espérer que cela entraînera d'autres travaux de coordination entre ces deux branches. À titre d'exemple, nous avons démarré suite à ces expériences une étude théorique sur l'ordonnancement de tâches parallèles en présence de réservations. Ces travaux sont exposés dans le chapitre suivant. Réciproquement, la deuxième version d'OAR qui est actuellement en phase de conception et de prototypage permettra de soumettre des tâches modelables.

Chapitre 4

Étude d'algorithmes d'ordonnancement avec réservations

Dans le chapitre précédent, nous avons vu l'implémentation de l'algorithme **SDE** dans OAR. Entre autres choses, cette implémentation a fait ressortir l'absence d'une étude théorique sur l'effet de la présence de réservations sur l'ordonnancement de tâches parallèles. Ce chapitre présente quelques résultats préliminaires dans cette direction, dans un cadre un peu simplifié afin de faciliter l'étude de ce problème difficile. Nous nous intéresserons donc uniquement à des tâches rigides, indépendantes, et sans dates d'arrivée, le tout dans un cadre totalement hors-ligne où les réservations sont connues à l'avance. Il en résulte quand même un problème dur et intéressant, dont nous étudions ici différentes variations.

La présence de réservations implique en fait que le nombre de processeurs disponibles pour exécuter des tâches varie au cours du temps. Des études sur l'ordonnancement avec nombre de machines variables ou avec machines indisponibles ont déjà été menées dans le cadre de tâches séquentielles ; un exposé de ces travaux est donné dans la section 4.1. Nous présentons ensuite formellement, dans la section 4.2, un modèle d'ordonnancement avec réservations ; nous y étudions également sa complexité et discutons de modèles alternatifs. La section 4.3 s'intéresse à des cas particuliers dérivés du problème général, pour lesquels nous donnons des performances de garantie et des bornes inférieures proches pour les algorithmes de liste.

4.1 Travaux apparentés

Il existe une littérature assez fournie dans le domaine de la Recherche Opérationnelle qui s'intéresse à l'ordonnancement d'activités lorsque les machines ne sont pas disponibles en continu. Un quantité importante de ces travaux porte sur des problèmes de logistique et de planification de production ; un bon tour d'horizon peut être trouvé dans [40]. Ces problèmes sont fortement apparentés aux problèmes d'ordonnancement sur machine parallèle qui nous intéressent ici, mais

avec des modèles d'exécution plus spécifiques, comme le modèle « *flow-shop* » dans lequel chaque tâche doit être effectuée en séquence sur chacune des machines, qui modélise plutôt une chaîne de production industrielle. Une revue des travaux avec contraintes de disponibilités dans un cadre plus informatique se trouve dans [60]. Dans tous les cas, il s'agit toujours uniquement de modèles à base de tâches séquentielles. Nous présentons ici quelques-uns de ces résultats qui sont les plus marquants et les plus proches du problème qui nous intéresse, en nous restreignant au critère du makespan sur plusieurs machines parallèles.

4.1.1 Sans préemption

Il existe relativement peu de résultats dans un cadre non préemptif. Dans un article de Lee [39], qui étudie le cas particulier où les périodes d'indisponibilité ne se trouvent qu'au début de l'ordonnancement[1], il est montré que l'algorithme **LPT** (« *Largest Processing Time* »), qui consiste à ordonnancer en priorité les plus longues tâches, a une garantie de performance de $\frac{3}{2}$ [39], au lieu de $\frac{4}{3}$ sans périodes d'indisponibilité. Dans le même article, l'auteur propose une version modifiée, **MLPT**, qui a également une garantie de $\frac{4}{3}$. La meilleure garantie connue pour ce problème est de $\frac{5}{4}$ par une approche basée sur le « *Bin Packing* » [34]. Notons que dans ce cadre, l'algorithme **SPT** (qui ordonnance les tâches les plus courtes d'abord) est optimal pour le critère $\sum C_i$ [60].

Dans un autre article [38], Lee étudie le cas où chaque machine n'a qu'une seule période d'indisponibilité, mais sans la restriction qu'elle soit au début de l'ordonnancement. Il suppose également qu'une des machines est disponible en continu. Dans ce cadre, il montre tout d'abord que l'algorithme de liste générique a une garantie de performance de m, puis que **LPT** a une garantie de $\frac{m+1}{2}$. Il n'y a à ma connaissance aucun résultat non préemptif avec plus d'une réservation par machine.

4.1.2 Avec préemption

Le problème est bien plus facile lorsque l'on autorise les préemptions. En effet, on voit dans [59] qu'il est possible d'ordonnancer de manière optimale des tâches indépendantes séquentielles avec indisponibilité quelconque des machines, même si l'on rajoute des dates butoirs pour chaque tâche. On peut également obtenir des algorithmes optimaux en présence de contraintes de précédence : dans [47], les auteurs étudient des cas particuliers de graphes de précédence (chaînes et forêts) pour lesquels certains algorithmes de liste sont optimaux.

Il existe également des études, motivées par des applications industrielles où les machines peuvent tomber en panne à tout moment, dans lesquelles les périodes d'indisponibilité ne sont pas connues à l'avance. Cela donne lieu à des études stochastiques [48] où l'on montre l'optimalité stochastique d'algorithmes de listes. Lorsque les périodes d'indisponibilité sont dues à des activité de maintenance, il est possible de ne plus considérer qu'elles sont fixées à l'avance, mais plutôt qu'elles

[1] Ce cas est appelé « *increasing pattern of availability* » dans [60], et correspond aux réservations décroissantes de la section 4.3.1

doivent être ordonnancées en même temps que les tâches. C'est alors plutôt le critère $\sum C_i$ qui est intéressant ; le lecteur intéressé par ces applications est invité à lire [40] pour plus de détails.

4.2 Modèle et discussion

Nous présentons dans cette section une modélisation naturelle des réservations, comme une extension au problème RigidScheduling du chapitre 1. Comme l'on peut s'y attendre, cette modélisation donne lieu à un problème très difficile non seulement à résoudre, mais également à approcher. Nous proposons également à la fin de cette section une discussion sur quelques modélisations alternatives que l'on pourrait imaginer.

4.2.1 Tâches rigides avec réservations

Voici une description formelle du problème d'ordonnancement que nous allons considérer :

Problème 4 (ResaScheduling)

Instance :
- *Un entier m représentant le nombre de machines,*
- *un ensemble de n tâches indépendantes $(T_i)_{i=1..n}$ caractérisées par une durée $p_i > 0$ et un nombre de processeurs requis $q_i \in [1..m]$,*
- *et n' réservations $(R_j)_{j=n+1..n+n'}$, caractérisées par une durée $p_j > 0$, un nombre de processeurs $q_j \in [1..m]$ et une date de début $r_j > 0$.*

Solution : *Un ordonnancement réalisable de makespan minimal.*

De la même manière que pour le problème RigidScheduling décrit à la section 1.2.2, un ordonnancement de ResaScheduling est réalisable si le nombre de processeurs utilisé à chaque instant est inférieur ou égal à m. On cherche donc une fonction $\sigma : [1..n] \mapsto \mathbb{N}$, telle que $\forall t, \sum_{i \in I_t} q_i + \sum_{j \in J_t} q_j \leq m$, où là encore $I_t \equiv \{i \,|\, \sigma_i \leq t < \sigma_i + p_i\}$ est l'ensemble des tâches en cours d'exécution à l'instant t, et $J_t \equiv \{j \,|\, r_j \leq t < r_j + p_j\}$ est l'ensemble des réservations actives à l'instant t.

On peut immédiatement noter que l'existence d'un ordonnancement réalisable n'est garantie que si les réservations sont elles-mêmes ordonnançables, c'est-à-dire n'utilisent jamais plus de m processeurs. On va donc s'intéresser uniquement à des instances raisonnables, pour lesquelles l'hypothèse suivante est vérifiée :

$$\forall t \geq 0, \quad \sum_{j \in J_t} q_j \leq m \tag{4.1}$$

On remarque alors qu'il est équivalent de considérer, au lieu des n' réservations, que l'on a accès à une fonction d'*indisponibilité* $U : \mathbb{N} \mapsto [0..m]$, qui donne pour chaque instant t le nombre de processeurs inutilisables du fait des réservations. Elle est donc définie par $U(t) = \sum_{j \in J_t} q_j$, et vaut nécessairement 0 à partir d'un certain temps, lorsque toutes les réservations sont terminées. Il est aisé de voir que cette fonction U résume la donnée des n' réservations pour toute instance

raisonnable. Nous noterons également $\mu(t) \equiv m - U(t)$ le nombre de ressources disponibles à l'instant t.

Le makespan d'un ordonnancement est comme précédemment défini comme le plus grand temps de complétion de toutes les tâches : $C_{\max} = \max_i \sigma_i + p_i$. Il est intéressant de noter que l'on ne prend pas en compte les temps de complétion des réservations (ce qui explique que l'on peut les résumer par la fonction U) : on s'intéresse ici uniquement à l'ordonnancement des tâches avec un nombre variable de processeurs disponibles. Une discussion sur le modèle alternatif qui consiste à définir le makespan comme le plus grand temps de complétion d'une tâche ou d'une réservation est présentée dans la section 4.2.3.

Remarque Cette formulation du problème conserve la propriété d'*indiscernabilité* des machines qui était déjà présente dans RigidScheduling. En effet, on insiste uniquement sur le nombre de processeurs disponibles pour effectuer les calculs ; on ne s'intéresse pas du tout à déterminer quels sont ces processeurs. Cela est parfaitement adapté au cas de l'ordonnancement en présence de réservations, comme c'est le cas dans OAR, car elles se comportent exactement comme des tâches dont la date d'ordonnancement est fixée : pour chaque réservation, il est possible de ne décider qu'au moment où elle démarre quels processeurs elle va utiliser.

En revanche, une partie des études précédentes sur l'indisponibilité (en particulier dans un cadre de *flow-shop* ou d'*open-shop*) justifie son approche par la nécessité d'effectuer de la maintenance sur les machines. Imaginons une situation où l'on veut prévoir une maintenance d'une heure sur dix machines, mais où la nécessité d'un opérateur humain pour cette maintenance fait que l'on ne peut l'effectuer que sur une machine à la fois. On pourrait effectuer une réservation pour chacune de ces périodes de maintenance, chaque réservation utilisant une machine et démarrant juste après la fin de la réservation précédente. Cependant, le modèle défini ci-dessus autorise alors l'exécution d'une tâche parallèle de dix heures utilisant 9 machines, puisqu'il y a à tout moment 9 machines disponibles. En réalité, les 9 machines disponibles ne sont jamais les mêmes, et aucune machine n'est disponible pendant les dix heures que dure la maintenance.

Le choix de cette modélisation permet cependant de simplifier grandement à la fois la formulation et l'analyse du problème ; il est de plus parfaitement adapté au but initial que l'on s'est posé pour comprendre l'influence des réservations sur l'ordonnancement dans un système de gestion tel que OAR.

4.2.2 Difficulté du problème

Le problème ResaScheduling est une généralisation directe du problème RigidScheduling, et est donc NP-difficile au sens fort également. Mais la présence de réservations augmente de beaucoup la difficulté : on prouve en effet dans cette section qu'il est impossible, à moins d'avoir $P = NP$, d'obtenir un algorithme avec une garantie de performance finie, et ce même si l'on se restreint à des instances avec une seule réservation ($n' = 1$) ou une seule machine ($m = 1$). L'idée

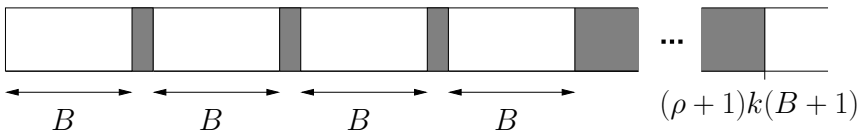


FIG. 4.1 – Transformation de RESASCHEDULING à partir de 3PARTITION

de cette preuve est qu'il est, pour ce problème, aussi difficile de trouver une solution approchée qu'une solution optimale, car il est toujours possible de fabriquer une instance avec une réservation suffisamment longue pour retarder indéfiniment toute solution non optimale.

Rappelons tout d'abord la définition du problème 3PARTITION [24], un des problèmes NP-complets les plus connus, et que nous allons utiliser dans la preuve.

Problème 5 (3PARTITION)

Instance : *un ensemble de* $3k$ *entiers* x_i*, et une borne* B *telle que* $\sum_i x_i = kB$*.*

Solution : *déterminer s'il existe une partition de* $[1..n]$ *en* k *groupes* G_l*, tous de cardinal 3, telle que* $\forall l, \sum_{i \in G_l} x_i = B$*.*

Le nom de ce problème vient du fait que l'on cherche une partition faite de groupes de cardinal 3 exactement ; mais on peut montrer que cette restriction peut être supprimée sans modifier la difficulté. En effet, en modifiant légèrement les entiers x_i, il est possible de transformer toute instance en une autre, pour laquelle toutes les partitions qui vérifient $\forall l, \sum_{i \in G_l} x_i = B$ ne contiennent que des groupes de cardinal 3.

Théorème 4

Si $P \neq NP$*, il n'existe pas d'algorithme polynomial pour le problème* RESASCHEDULING *qui a une garantie de performance finie, même dans les cas particuliers* $m = 1$ *(une seule machine) ou* $n' = 1$ *(une seule réservation).*

Démonstration : Nous allons faire la preuve pour le cas $m = 1$, en utilisant une réduction à partir du problème 3PARTITION. Le cas $n' = 1$ se traite aisément de manière similaire, à partir du problème RIGIDSCHEDULING.

Nous allons prouver le théorème par contradiction, en supposant l'existence d'un algorithme $\mathcal{A}$ qui résout le problème RESASCHEDULING avec une garantie de performance ρ ; nous allons alors montrer que cet algorithme résout de manière exacte le problème 3PARTITION. Pour cela, considérons une instance I_P du problème 3PARTITION, avec les notations ci-dessus. Nous construisons alors une instance I de RESASCHEDULING, à une seule machine, de telle sorte que le temps entre deux réservations successives soit exactement B (voir la figure 4.1) :

- $m = 1$;
- $n = 3k$ tâches avec $\forall i, q_i = 1$ et $p_i = x_i$;
- k réservations $(R_j)_{j=n+1..n+k}$ définies par $q_j = 1, r_{n+1} = B$, et $r_j = r_{j-1} + B + 1$ pour $n+1 < j \leq n+k$ [2]. Les durées des réservations sont $p_j = 1$ pour

[2] c'est-à-dire $r_j = (j-n)(B+1) - 1$ pour $n+1 \leq j \leq n+k$

$j \neq n+k$, et la durée de la dernière réservation est $p_{n+k} = \rho k(B+1)+1$ (elle termine donc au temps $(\rho+1)k(B+1)$).

S'il existe une solution $(G_l)_{1\leq l\leq k}$ au problème 3PARTITION pour l'instance I_P, alors il est possible de construire un ordonnancement de makespan $C^*_{\max} = k(B+1)-1$ en ordonnançant les tâches correspondant au groupe G_l entre la $(l-1)^{\text{e}}$ et la l^{e} réservation. Cet ordonnancement est clairement optimal, puisque la machine est utilisée pour exécuter une tâche à chaque fois qu'elle est disponible.

Comme $\mathcal{A}$ a une garantie de performance de ρ, il doit donc fournir un ordonnancement dont le makespan vérifie $C^{\mathcal{A}}_{\max} \leq \rho(k(B+1)-1) < \rho k(B+1)$. Comme il est impossible d'ordonnancer une tâche entre les instants $k(B+1)-1$ et $(\rho+1)k(B+1)$, on a alors nécessairement $C^{\mathcal{A}}_{\max} = C^*_{\max}$. L'ordonnancement fourni par $\mathcal{A}$ permet donc de déterminer une solution à l'instance I_P, en affectant les tâches ordonnancées entre deux réservations successives au même groupe de la partition.

Réciproquement, s'il n'existe pas de solution à l'instance I_P, $\mathcal{A}$ produit un ordonnancement dont le makespan est strictement supérieur à $\rho k(B+1)$, puisqu'il est impossible d'ordonnancer toutes les tâches avant la fin des réservations. ■

4.2.3 Discussion

Ce résultat d'inapproximabilité diminue l'intérêt de ce modèle d'ordonnancement avec réservations – il est difficile d'imaginer quels résultats positifs on va pouvoir prouver. Le problème vient du fait qu'il est très facile de construire des contre-exemples grâce aux réservations : celles-ci ajoutent des contraintes tellement fortes qu'il est possible de faire en sorte que seule la solution optimale ait une performance raisonnable. C'est pourquoi la suite de ce chapitre s'intéresse à des cas spécifiques dans lesquels on restreint les possibilités des réservations, et qui correspondent à des hypothèses raisonnables en pratique. Dans cette discussion, nous allons examiner d'autres critères que l'on pourrait envisager pour le problème RESASCHEDULING, afin de voir s'il est possible d'obtenir des résultats plus satisfaisants.

Élargir le makespan

Une première idée peut être, comme on l'a évoqué précédemment, de considérer réellement les réservations comme des tâches dont l'ordonnancement est fixé, et ainsi d'ajouter leur temps de complétion à la définition du makespan. Cela pourrait d'ailleurs être plus pertinent, puisque dans le cadre d'OAR, les réservations constituent également des calculs à effectuer. Dans ce cas, les constructions de la preuve précédente ne fonctionnent plus, puisque la grande réservation fait que l'ordonnancement optimal a un makespan presque aussi grand que l'ordonnancement approché.

Avec une telle définition, il est possible d'obtenir assez facilement des algorithmes d'approximation en revenant au problème RIGIDSCHEDULING. En effet, à partir d'un algorithme $\mathcal{A}$ de ρ-approximation pour RIGIDSCHEDULING, on peut construire un algorithme $\mathcal{A}'$ qui est une $\rho+1$-approximation pour RESASCHEDULING. $\mathcal{A}'$ est défini de la façon suivante : attendre (et ne rien faire) jusqu'à t_1, date

de fin de la dernière réservation ; puis ordonnancer toutes les tâches par l'algorithme $\mathcal{A}$ à partir de t_1. On a donc $C_{\max}^{\mathcal{A}'} = t_1 + C_{\max}^{\mathcal{A}}$; or d'après la définition du makespan, $t_1 \leq C_{\max}^*$, de plus l'hypothèse sur $\mathcal{A}$ permet d'écrire $C_{\max}^{\mathcal{A}} \leq \rho C_{\max}^*$. On obtient donc bien que $C_{\max}^{\mathcal{A}'} \leq (\rho + 1) C_{\max}^*$.

Ainsi, si l'on part par exemple d'un algorithme de liste comme décrit dans la section 1.4.1 et dont la garantie de performance est 2, on obtient une 3-approximation.

Juger l'utilisation

Dans un modèle sans réservation, il y a deux justifications principales pour le critère du makespan. En effet, si l'on considère que toutes les tâches appartiennent au même calcul global, c'est alors la date de fin de la dernière tâche qui détermine quand le résultat du calcul complet est disponible ; c'est la justification usuelle pour le problème $P \,|\, \text{prec} \,|\, C_{\max}$ (voir section 1.2.1). Ou alors, on peut adopter le point de vue du propriétaire d'une machine parallèle qui la rendrait disponible à une communauté pour qu'elle effectue des calculs. Le résultat exact de ces calculs n'a pas d'importance réelle pour ce propriétaire ; c'est l'*utilisation* de la machine qui importe, c'est-à-dire effectuer le plus de calcul possible dans un temps donné. De façon symétrique, cela revient à mettre le moins de temps possible pour effectuer un ensemble de calculs donnés. Cette justification est souvent plus adaptée dans les cadres où l'on considère des tâches parallèles indépendantes.

Il est donc assez naturel de généraliser cette notion d'utilisation dans un contexte avec réservations, en comptabilisant les ressources dont on a eu besoin pour exécuter toutes les tâches. Si $C_{\max}$ est la date de complétion de la dernière tâche (sans compter les réservations), alors ce critère peut être défini par $R \equiv \sum_{0 \leq t \leq C_{\max}} \mu(t)$: pour chaque instant t, on compte le nombre de ressources disponibles à cet instant. On évalue ainsi le nombre de ressources utilisées par l'ordonnancement, soit pour effectuer réellement des calculs, soit perdues à cause de la fragmentation. Une longue période sans aucune ressource disponible, comme celle de la preuve d'inapproximabilité ci-dessus, n'est alors pas comptabilisée dans le critère d'évaluation et ne pénalise plus les algorithmes d'ordonnancement par rapport à l'optimal. On peut de plus remarquer que cette définition de R est effectivement une généralisation du makespan : en l'absence de réservations, on a $U(t) = 0$ pour tout t, et donc $R = mC_{\max}$.

Cependant, cette définition n'empêche pas vraiment de construire des contre-exemples similaires au précédent. On peut par exemple imaginer une instance avec $m = 2$ processeurs, dans laquelle toutes les tâches nécessitent $q_i = 2$ processeurs, et toutes les réservations $q_j = 1$ processeur. On se retrouve alors dans une situation équivalente à celle de la preuve précédente ; mais cette fois le temps passé pendant la grande réservation sera comptabilisé dans R, même si c'est à un rythme deux fois plus faible que dans $C_{\max}$. Il est donc également impossible d'approcher R dans un contexte avec réservations.

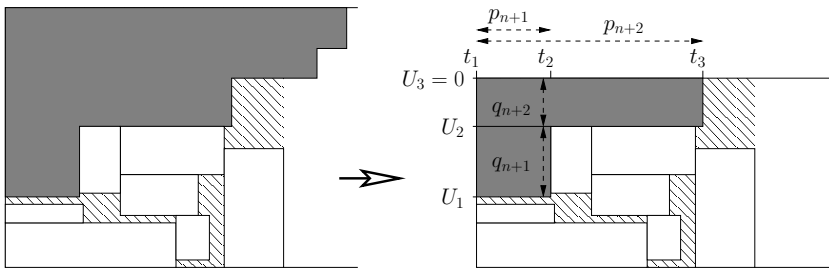


FIG. 4.2 – Un exemple de réservations décroissantes et de la transformation utilisée dans la preuve de la proposition 2

4.3 Cas particuliers

Dans la suite de cette étude, nous allons définir deux restrictions des instances de RESASCHEDULING pour lesquelles il est possible de concevoir des algorithmes garantis. Pour chacune de ces restrictions, nous analysons les performances des algorithmes de liste.

4.3.1 Réservations décroissantes

Nous nous intéressons tout d'abord au cas de réservations décroissantes, ce qui revient à dire que la disponibilité des machines augmente au cours du temps. Bien que cette hypothèse ne soit pas très réaliste lorsque l'on considère les réservations dans OAR, elle correspond parfaitement à un ordonnancement dans un cadre en ligne. En effet, il est courant que lorsque l'on ordonnance l'ensemble des tâches qui sont arrivées dans le système depuis la dernière phase d'ordonnancement, quelques anciennes tâches soient encore en cours d'exécution. Dans ce cas, certains processeurs sont occupés, et se libèrent progressivement au fur et à mesure que ces tâches se terminent. C'est également pour cette raison que cette hypothèse de disponibilités croissantes a déjà été étudiée dans la littérature, comme on l'a vu dans la section 4.1 (par exemple dans [39]).

Avec une telle hypothèse, le problème n'est pas beaucoup plus difficile que le problème RIGIDSCHEDULING dont on est parti. Nous pouvons en effet prouver le même rapport de performance pour l'algorithme de liste :

Proposition 2

Pour toute instance I avec réservations décroissantes, c'est-à-dire telle que U est décroissante au sens large, on a :

$$C_{\max}^{List} \leq \left(2 - \frac{1}{m}\right) C_{\max}^{*}$$

Démonstration : Considérons une instance I avec réservations décroissantes, et σ un ordonnancement de liste pour cette instance. Nous allons prouver la garantie de σ en ramenant l'instance I à une instance équivalente, mais pour le problème

RigidScheduling. Nous définissons tout d'abord une instance I_1 en « oubliant » les réservations qui se trouvent après le makespan de σ, noté $C^{\sigma}_{\max}$.[3]

I_1 contient donc les mêmes tâches que I, un nombre de processeurs $m^{I_1} = \mu^I(C^{\sigma}_{\max})$, et des réservations telles que $\mu^{I_1}(t) = \mu^I(t)$ pour tout $t \leq C^{\sigma}_{\max}$. Il est clair que tout ordonnancement réalisable pour I_1 l'est également pour I, qui a plus de ressources disponibles ; de plus, tout ordonnancement pour I de makespan inférieur ou égal à $C^{\sigma}_{\max}$ est également réalisable pour I_1, puisque I_1 a les mêmes ressources disponibles jusqu'à la fin de cet ordonnancement. Il en résulte en particulier que $C^{*,I_1}_{\max} = C^{*,I}_{\max}$.

Nous allons maintenant convertir les réservations de l'instance I_1 en tâches, de façon à obtenir une instance de RigidScheduling pour laquelle l'ordonnancement correspondant aux réservations sera un ordonnancement de liste. Pour cela, notons $U_1, \ldots, U_k$ les k différentes valeurs que prend la fonction U^{I_1}, de sorte que $U^{I_1}(t) = U_j$ pour $t_j \leq t < t_{j+1}$. On a donc $t_1 = 0$, $t_{k+1} = \infty$, et $U_k = 0$. On définit alors $k-1$ tâches $T_{n+1}, \ldots, T_{n+k-1}$ par $q_{n+j} = U_j - U_{j+1}$ et $p_{n+j} = t_{j+1}$, de sorte qu'elles utilisent les mêmes ressources que les réservations si elles commencent toutes à la date 0 (voir figure 4.2). L'instance I_2 de RigidScheduling est alors obtenue en ajoutant ces tâches supplémentaires aux n tâches de I_1. Il est alors aisé de transformer tout ordonnancement réalisable pour I_1 en ordonnancement réalisable pour I_2, en ajoutant simplement 0 comme date de début de chaque tâche supplémentaire. En particulier, l'ordonnancement optimal pour I_1 est réalisable pour I_2, et on a donc $C^{*,I_2}_{\max} \leq C^{*,I_1}_{\max}$.

En faisant subir cette transformation à l'ordonnancement σ, on obtient un ordonnancement σ' qui est lui-même un ordonnancement de liste pour I_2 : il suffit de rajouter les tâches correspondant aux réservations au début de la liste utilisée pour produire l'ordonnancement σ. De plus, $C^{\sigma'}_{\max} = C^{\sigma}_{\max}$, puisque toutes les tâches supplémentaires, qui ne sont pas comptées dans $C^{\sigma}_{\max}$, terminent avant la fin de σ.

D'après le théorème 1, on a donc $C^{\sigma'}_{\max} \leq \left(2 - \frac{1}{m^{I_2}}\right) C^{*,I_2}_{\max}$, d'où l'on déduit le résultat :

$$C^{\sigma}_{\max} \leq \left(2 - \frac{1}{m^I}\right) C^{*,I}_{\max}$$

■

4.3.2 Réservations restreintes

Nous allons maintenant considérer une restriction sur la quantité de ressources utilisées par les réservations, de sorte qu'il soit toujours possible d'exécuter une tâche normale en même temps qu'une réservation. Pour cela, étant donné un paramètre $\alpha \in]0;1]$, nous définissons le problème α-ResaScheduling en ne considérant que les instances où il y a toujours une fraction α des processeurs qui est disponible. Ce genre de contrainte se rapproche des politiques qui peuvent être mises en œuvre dans certaines grappes de calcul pour essayer de faciliter l'ordonnancement et d'améliorer l'utilisation de la grappe. Formellement, nous ajoutons les contraintes suivantes :

[3] Comme nous allons manipuler plusieurs instances différentes, nous adoptons ici la notation suivante : chaque paramètre portera en exposant le nom de l'instance auquel il fait référence. Ainsi l'instance I contient m^I processeurs, et $C^{*,I}_{\max}$ est le makespan optimal de l'instance I.

$$\forall t \geq 0, \quad U(t) = \sum_{j \in J_t} q_j \leq (1-\alpha)m$$

$$\forall i \leq n, \quad q_i \leq \alpha m$$

Ces contraintes garantissent que l'on peut toujours exécuter au moins une tâche, et empêchent donc les contre-exemples pathologiques de la section précédente. Nous étudions donc encore une fois les performances de l'algorithme de liste, en fournissant une borne inférieure et une borne supérieure de sa garantie de performance.

Nous pouvons également noter que cette contrainte est en fait une généralisation de l'hypothèse faite dans [38] qu'il y a toujours au moins une machine disponible, et qui revient à prendre $\alpha = \frac{1}{m}$.

Borne supérieure

Nous allons prouver tout d'abord que, pour tout α, l'algorithme **List** est une $\frac{2}{\alpha}$-approximation. Ce résultat en lui-même n'est pas très fort ; il revient informellement à dire qu'au pire, **List** utilise αm processeurs alors que l'ordonnancement optimal en utilise m. La garantie pour α-ResaScheduling vient alors du fait que **List** est une 2-approximation pour le problème RigidScheduling, comme on l'a vu à la section 1.4.1. Cependant, nous allons voir avec la borne inférieure que cette garantie est assez précise, et qu'il serait difficile de prouver une meilleure garantie pour l'algorithme **List** générique.

Proposition 3

Quel que soit $\alpha \in]0;1]$, l'algorithme **List** *est une $\frac{2}{\alpha}$-approximation pour le problème α-ResaScheduling.*

Démonstration : Pour prouver cette garantie, il suffit de remarquer que comme **List** peut toujours utiliser au moins αm processeurs, on peut écrire une propriété similaire à la proposition 1 pour le problème RigidScheduling :

$$\forall t, t' \in [0, C_{\max}^{\text{List}})[, \quad t' \geq t + p_{\max} \;\Rightarrow\; r(t) + r(t') > \alpha m$$

où $r(t)$ est le nombre de processeurs utilisé à l'instant t par l'ordonnancement de liste.

Donc si $C_{\max} > \frac{2}{\alpha} p_{\max}$, on peut écrire

$$\forall t \leq \frac{1}{\alpha} C_{\max}, \quad r(t) + r\left(t + \frac{1}{\alpha} C_{\max}\right) > \alpha m$$

Et en intégrant cette relation, on arrive de la même manière que pour la preuve du théorème 1 à $C_{\max} \leq \frac{2}{\alpha} \frac{W}{m}$. ■

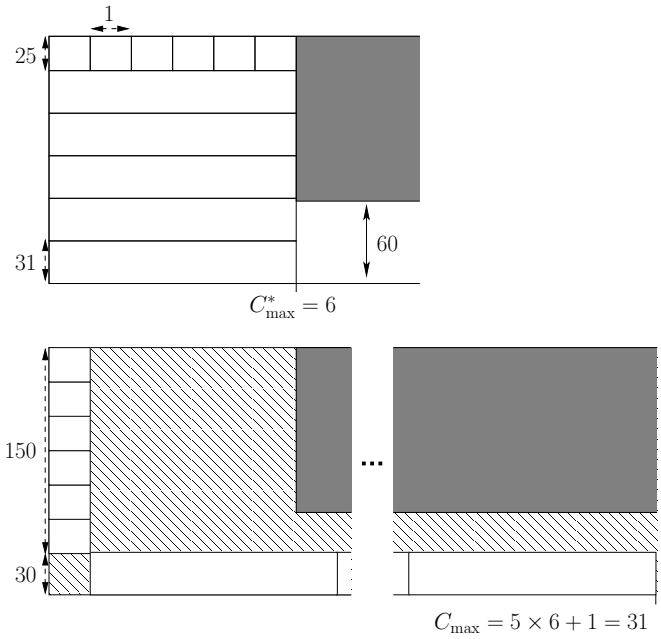


FIG. 4.3 – Un ordonnancement optimal pour α-RESASCHEDULING et l'ordonnancement de liste correspondant, pour $\alpha = \frac{1}{3}$ ($m = 180$).

Borne inférieure

Nous donnons dans cette section plusieurs bornes inférieures pour l'algorithme **List** général, et nous verrons finalement que la borne supérieure précédente est assez précise. Nous commençons par le cas particulier où $\alpha = \frac{2}{k}$, où k est un entier. Pour ce cas, qui est plus simple à appréhender, nous donnons la construction complète du contre-exemple ; les autres cas sont très similaires et plus fastidieux, aussi nous passerons plus rapidement dessus. Nous construisons donc une instance pour le cas $\alpha = \frac{2}{k}$, où k est un entier, dans laquelle l'ordonnancement optimal utilise tout le temps m processeurs, mais où il existe un ordre de la liste pour lequel **List** n'utilise que αm processeurs pendant la majorité du temps.

Proposition 4

Si $\frac{2}{\alpha}$ est entier, la garantie de performance de **List** *est au moins de $\frac{2}{\alpha} - 1 + \frac{\alpha}{2}$.*

Démonstration : Supposons que $\alpha = 2/k$, avec $k \in \mathbb{N}$. Nous construisons une instance I avec $m = k^2(k-1)$ processeurs, qui contient deux types de tâches différents (voir la figure 4.3) :

- k tâches courtes, de T_1 à T_k, avec $p_i = 1$ et $q_i = (k-1)^2$;
- $k-1$ tâches longues, de T_{k+1} à T_{2k-1}, avec $p_i = k$ et $q_i = k(k-1)+1$;

De plus, I contient une réservation qui commence au temps k, et occupe $m(1-\alpha) = m - 2m/k = k(k-1)(k-2)$ processeurs pendant $2k^2$ unités de temps.

Comme $(k-1)\times(k(k-1)+1)+(k-1)^2 = (k-1)(k(k-1)+k) = (k-1)k^2 = m$, il est possible d'ordonnancer toutes les tâches avant le temps k, qui est le début de la réservation. Le makespan optimal pour cette instance est donc $C^*_{\max} = k$.

Si les tâches sont ordonnées par i croissant, l'algorithme **List** ordonnance dès l'instant 0 les k tâches courtes, qui sont de largeur $(k-1)^2$; cela est possible puisque $k \times (k-1)^2 \leq m$. Mais il n'est alors pas possible de commencer une tâche longue à la date 0, puisqu'il ne reste que $k(k-1)$ processeurs disponibles. Et comme la réservation ne laisse que $2m/k = 2k(k-1)$ processeurs libres, il n'est alors plus possible d'ordonnancer deux tâches longues en même temps. Ces tâches doivent donc être ordonnancées les unes après les autres, ce qui conduit à un makespan de $C_{\max}^{\text{List}} = 1 + (k-1) \times k = \left(\frac{2}{\alpha} - 1 + \frac{\alpha}{2}\right) C_{\max}^{*}$. ■

Pour α général, on peut construire une instance similaire, mais la construction est un peu fastidieuse. Notons cependant que c'est lorsque $\frac{2}{\alpha}$ approche d'un entier par au-dessus que la borne inférieure se rapproche de la borne supérieure, et qu'il y a même des valeurs de α pour lesquelles ces deux bornes sont arbitrairement proches.

Nous construisons donc une instance avec $n_1 = \left\lceil \frac{2}{\alpha} - 1 \right\rceil$ tâches longues de largeur $h_1 = (\alpha/2 + \epsilon)\, m$ et de longueur K, de sorte qu'il est impossible d'ordonnancer 2 tâches longues en même temps que la réservation. Cette réservation commence à l'instant K et ne laisse que αm processeurs disponibles. On ajoute également n_2 tâches courtes, de largeur $h_2 = (1 - n_1 h_1)\, m$ et de longueur K/n_2, en choisissant n_2 de sorte que si l'on ordonnance les n_2 tâches courtes à l'instant 0, il n'y ait plus de place pour une tâche longue ; il suffit de choisir $n_2 = \left\lfloor \frac{1-\alpha/2}{h_2} \right\rfloor + 1$. Pour K assez grand, m assez grand et ϵ assez petit, l'ordonnancement de liste qui en résulte a donc pour longueur $n_1 K + \frac{K}{n_2}$, alors que l'ordonnancement optimal est de longueur K. La garantie de performance de **List** vérifie donc $\rho \geq n_1 + \frac{1}{n_2}$, soit :

$$\begin{aligned} \rho &\geq \left\lceil \frac{2}{\alpha} \right\rceil - 1 + \frac{1}{\left\lfloor \frac{1-(\alpha/2)}{1-(\alpha/2)(\lceil 2/\alpha \rceil - 1)} \right\rfloor + 1} \equiv B_1 \\ &\geq \left\lceil \frac{2}{\alpha} \right\rceil - \frac{\lceil 2/\alpha \rceil - 1}{2/\alpha} \equiv B_2 \end{aligned}$$

La borne B_2 est obtenue en simplifiant un peu l'expression B_1 ; elle est un peu moins précise, mais plus simple à exprimer. La figure 4.4 montre les différentes garanties obtenues ainsi ; on voit en particulier que les bornes B_1 et B_2 sont bien une généralisation de la borne $\frac{2}{\alpha} - 1 + \frac{\alpha}{2}$ pour le cas particulier précédent, et peuvent être arbitrairement proches de la borne supérieure $\frac{2}{\alpha}$. Cela suggère que pour prouver une garantie meilleure pour **List** (à supposer que ce soit possible), il faudrait différencier selon la proximité de $2/\alpha$ à un entier, ce qui semble bien plus délicat.

4.4 Synthèse

Dans ce chapitre, nous avons entamé l'étude théorique de l'effet des réservations sur l'ordonnancement de tâches parallèles, en proposant une modélisation naturelle de ce problème. Nous avons surtout fourni des contre-exemples afin de montrer ce

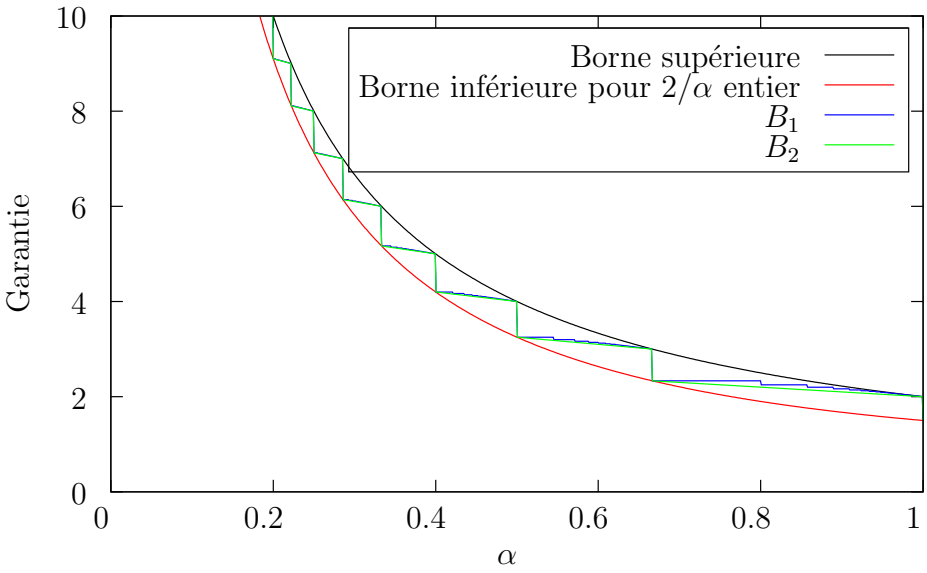


FIG. 4.4 – Encadrement de la garantie de performance de **List** pour le problème α-RESASCHEDULING en fonction de α.

que l'on peut attendre d'algorithmes d'approximation dans ce cadre. Nous avons également étudié deux cas particuliers qui sont une généralisation des études faites dans le cadre des tâches séquentielles, et pour lesquels nous avons pu obtenir des garanties pour les algorithmes de liste qui sont très proches des bornes inférieures.

Nous avons montré que la garantie de $2 - \frac{1}{m}$ en l'absence de réservations peut s'étendre au cas où les réservations sont décroissantes, et elle est donc exactement égale à la borne inférieure. Dans un cadre de réservations restreintes qui correspond à une hypothèse classique dans le cas des tâches séquentielles, nous avons montré une garantie de performance et une borne inférieure qui sont arbitrairement proches pour certaines valeurs de α.

Conclusions et perspectives

Le domaine de l'ordonnancement sur machines parallèles est un domaine très vaste, avec un grand nombre d'applications et de modèles différents. Cette thèse apporte une contribution par une approche originale, en menant des études théoriques avec un ancrage fort dans a pratique et la réalité. Cette approche a ainsi permis de considérer de nouveaux problèmes, comme l'introduction de réservations dans l'ordonnancement de tâches parallèles, mais également d'apporter des réponses nouvelles, fondées sur des bases théoriques solides, à des problèmes pratiques.

Nous avons ainsi étudié l'ordonnancement de tâches modelables indépendantes, et obtenu des garanties de performance simultanément de 3 sur le makespan et de 6 sur la moyenne pondérée des temps de complétion ; cette dernière améliore les meilleures garanties connues pour ce problème. Dans un cadre en ligne, nous donnons également un algorithme randomisé avec des garanties de performance d'environ $4,1$ en moyenne. Nous avons ensuite proposé une simplification de cet algorithme, qui obtient de bons résultats dans une étude expérimentale, avec des performances très stables vis-à-vis des différentes instances possibles.

La simplification apportée à cet algorithme a permis de l'implémenter dans OAR, un logiciel de gestion de ressources développé au laboratoire ID-IMAG. Ce travail a permis de mettre en lumière les divergences entre les modèles théoriques et la pratique courante dans de tels logiciels. L'algorithme résultant de cette implémentation a cependant de très bons résultats par rapport à l'heuristique **FCFS** originellement en place dans OAR.

Enfin, nous avons étudié l'ordonnancement de tâches rigides en présence de réservations, car c'est l'une des contraintes rencontrées lors de cette implémentation. Nous avons proposé un modèle et étudié des cas particuliers qui généralisent les études faites précédemment dans le cadre des tâches séquentielles. Nous avons également prouvé des garanties de performance pour les algorithmes de liste dans ces cas particuliers, ainsi que des bornes inférieures qui montrent que ces garanties sont d'une certaine manière les meilleures possibles.

À l'issue de ces travaux, il reste des pistes pour continuer les recherches dans cette direction. L'étude faite sur les réservations est assez préliminaire, et n'a pas réellement permis de conclure sur quels algorithmes utiliser pour gérer les réservations dans un gestionnaire de ressources. On pourrait en particulier chercher d'autres algorithmes avec de meilleures garanties, ou étudier d'autres critères que le makespan. Une autre question qui apparaît dans ce genre de cadre, mais que l'on

n'a pas abordée, porte sur la politique d'acceptation des réservations : comment choisir si l'on accepte ou non un ensemble de réservations, de façon à optimiser l'utilisation de la machine lors de l'ordonnancement des tâches ?

Les travaux d'application du chapitre 3 peuvent également être poursuivis. En effet, une nouvelle version du logiciel OAR est actuellement en phase de test, et les fonctionnalités ajoutées dans cette version complexifient encore le modèle d'ordonnancement correspondant. En particulier, il est maintenant possible de soumettre des tâches modelables ; l'expressivité des propriétés a également augmenté. Il serait donc intéressant d'étudier les effets de cette évolution sur l'implémentation d'algorithmes d'ordonnancement, et analyser les nouveaux problèmes théoriques que cela apporte.

Une autre piste importante est l'étude, d'un point de vue théorique, de modèles en ligne bien adaptés à ces environnements de grappe. En particulier, des critères comme le « *stretch* » ou le flot semblent de meilleures mesures de performance dans un tel cadre ; mais leur analyse théorique n'est pas encore bien maîtrisée et il reste des problèmes intéressants à résoudre dans ce domaine.

La pratique des systèmes parallèles est encore en pleine évolution ; l'essor des *grilles de calcul* et l'apparition de processeurs multi-cœurs, par exemple, font que l'on s'intéresse de plus en plus à des architectures irrégulières et hiérarchiques. Je pense que cette approche résolument théorique, mais liée à la pratique, que j'ai suivie au cours de cette thèse peut permettre de suivre au plus près ces évolutions. Elle permet aussi éventuellement d'influencer les décisions prises dans la conception des logiciels, afin qu'elles soient raisonnées par des études plus fondamentales. Les systèmes à grande échelle apportent un grand nombre de nouveaux problèmes dans des domaines très variés, pour lesquels il est encore nécessaire de trouver des modélisations appropriées ; les approches et les techniques développées dans cette thèse pourront être utiles à ces travaux futurs.

Bibliographie

[1] D. P. Anderson. Boinc : A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, 2004.

[2] K. Baker. *Introduction to Sequencing and Scheduling.* John Wiley & Sons, 1974.

[3] M. Baker, G. Fox, and H. Yau. Cluster computing review, 1995.

[4] E. Bampis, R. Giroudeau, and J.-C. König. An approximation algorithm for the precedence constrained scheduling problem with hierarchical communications. *Theoretical Computer Science*, 290(3) :1883–1895, 2003.

[5] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters. *Int. J. of Foundations of Computer Science*, 2005.

[6] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems.* IEEE Computer Society Press, September 1996.

[7] N. Capit, G. D. Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, 2005.

[8] S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. In *ICALP '96 : Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, pages 646–657, London, UK, 1996. Springer-Verlag.

[9] B. Chen. Parallel scheduling for early completion. In Leung [44], chapter 9.

[10] Le projet ciment grid. `http://cigri.ujf-grenoble.fr/`.

[11] Le projet ciment. `http://ciment.ujf-grenoble.fr/`.

[12] W. Cirne and F. Berman. A model for moldable supercomputer jobs. In *15th Intl. Parallel & Distributed Processing Symp.*, 2001.

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

[14] A. Downey. A model for speedup of parallel programs. Technical Report CSD-97-933, University of California, Berkeley, CA, USA, 1997.

[15] A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *Performance Evaluation Rev.*, 26(4) :14–29, Mar 1999.

[16] M. Drozdowski. Scheduling parallel tasks – algorithms and complexity. In Leung [44], chapter 25.

[17] P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *Symposium on Parallel Algorithm and Architectures*, pages 125–132, Barcelona, 2004.

[18] P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Scheduling on large scale distributed platforms : from models to implementations. *Intl. Journal of Foundations of Computer Science*, 16(2) :217–237, 2005.

[19] P.-F. Dutot, G. Mounié, and D. Trystram. Scheduling parallel tasks – approximation algorithms. In Leung [44], chapter 26.

[20] L. Eyraud. A pragmatic analysis of scheduling environments on new computing platforms. *Intl. Journal of High Performance Computing and Applications*, 2006. to be published.

[21] D. G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *IPPS '96 : Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26, London, UK, 1996. Springer-Verlag.

[22] The folding@home website. `http://folding.stanford.edu`.

[23] Le projet grid'5000. `http://www.grid5000.org`.

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

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

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

[27] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. *Annals of Discrete Mathematics*, 5 :287–326, 1979.

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

[29] D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. Pws, September 1996.

[30] La plate-forme icluster2. `http://www.inrialpes.fr/i-cluster2/`.

[31] A. M. J.C. Billaut and E. Sanlaville, editors. *Flexibilité, robustesse en Ordonnancement*. Informatique et Systèmes d'Information. Hermes, Lavoisier, 2005.

[32] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4) :299–325, 1974.

[33] Le logiciel kaapi. http://kaapi.gforge.inria.fr/.

[34] H. Kellerer. Algorithms for multiprocessor scheduling with machine release times. *IIE Transactions*, 30 :991–999, 1998.

[35] C. Kenyon and E. Rémila. A near optimal solution to a two-dimensional cutting stock problem. *MOR : Mathematics of Operations Research*, 25, 2000.

[36] J. Krallmann, U. Schwiegelshohn, and R. Yahyapour. On the design and evaluation of job scheduling algorithms. In *IPPS/SPDP '99/JSSPP '99 : Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 17–42, London, UK, 1999. Springer-Verlag.

[37] J. Labetoulle, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Preemptive scheduling of uniform machines subject to release dates. In *Progress in combinatorial optimization*, pages 245–261. Academic Press, 1984.

[38] C.-Y. Lee. Machine scheduling with an availability constraint. *Journal of Global Optimization*, 9 :395–416, 1996.

[39] C.-Y. Lee. Parallel machines scheduling with nonsimultaneous machine available time. *Discrete Applied Mathematics*, 30 :53–61, 1997.

[40] G. Lee. Machine scheduling with availability constraints. In Leung [44], chapter 22.

[41] A. Legrand, A. Su, and F. Vivien. Off-line scheduling of divisible requests on an heterogeneous collection of databanks. In *Proceedings of the 14th Heterogeneous Computing Workshop*, Denver, Colorado, USA, apr 2005. IEEE Computer Society Press.

[42] J. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1 :343–362, 1977.

[43] R. Lepère, D. Trystram, and G. Woeginger. Approximation scheduling for malleable tasks under precedence constraints. In *9th Annual European Symposium on Algorithms - ESA 2001*, number 2161 in LNCS, pages 146–157. Springer-Verlag, 2001.

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

[45] J. Y.-T. Leung. Introduction and notation. In Leung [44], chapter 1.

[46] J. Y.-T. Leung. Some basic scheduling algorithms. In Leung [44], chapter 3.

[47] Z. Liu and E. Sanlaville. Profile scheduling by list algorithms. In P. Chrétienne, J. E.G. Coffman, J. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications*. J. Wiley, 1994.

[48] Z. Liu and E. Sanlaville. Stochastic scheduling with variable profile and precedence constraints. *SIAM Journal on Computing*, 26(1) :173–187, 1997.

[49] U. Lublin and D. Feitelson. The Workload on Parallel Supercomputers : Modeling the Characteristics of Rigid Jobs. *Journal of Parallel and Distributed Computing*, 63(11), 2003.

[50] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *SODA '94 : Proceedings of the fifth ACM-SIAM Symposium on Discrete Algorithms*, pages 167–176, 1994.

[51] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. *hcw*, 00 :30, 1999.

[52] R. Motwani, S. Phillips, and E. Torng. Nonclairvoyant scheduling. *Theor. Comput. Sci.*, 130(1) :17–47, 1994.

[53] G. Mounié, C. Rapine, and D. Trystram. A 3/2-dual approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM J. of Computing*, 2006. to appear.

[54] A. Munier and J.-C. König. A heuristic for a scheduling problem with communication delays. *Operations Research*, 45 :145–147, 1997.

[55] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize average stretch. In *FOCS '99 : Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 433, Washington, DC, USA, 1999. IEEE Computer Society.

[56] Le logiciel oar. http://oar.imag.fr.

[57] K. H. Randall. *Cilk : Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.

[58] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[59] G. Schmidt. Scheduling independent tasks with deadlines on semi-identical processors. *Journal of the Operational Research Society*, 39 :271–277, 1988.

[60] G. Schmidt. Scheduling with limited machine availability. *European Journal of Operational Research*, 121(1) :1–15, 2000.

[61] P. Schuurman and G. J. Woeginger. Polynomial time approximation algorithms for machine scheduling : Ten open problems. *Journal of Scheduling*, pages 203–213, 1999.

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

[63] U. Schwiegelshohn and R. Yahyapour. Analysis of first-come-first-serve parallel job scheduling. In *Proceedings of the 9th SIAM Symposium on Discrete Algorithms (SODA 98)*, pages 629–638, 1998.

[64] The seti@home website. http://setiathome.berkeley.edu.

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

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

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

[68] H. Topcuouglu, S. Hariri, and M. you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3) :260–274, 2002.

[69] J. Turek, W. Ludwig, J. L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P. S. Yu. Scheduling parallelizable tasks to minimize average response time. In *6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 200–209, 1994.

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

[71] F. Vivien and A. Legrand. Sum-stretch minimization. Internal communication, 2005.

[72] T. Yang and A. Gerasoulis. A fast static scheduling algorithm for dags on an unbounded number of processors. In *Supercomputing '91 : Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 633–642, New York, NY, USA, 1991. ACM Press.