Nouvelles approches pour l'ordonnancement d'applications parallèles sous contraintes de déploiement d'environnements sur grappe.

Feryal Windal

Université Joseph Fourier de Grenoble

N° attribué par la bibliothèque

Thèse

pour obtenir le grade de

Docteur de l'Université Joseph Fourier de Grenoble

Spécialité : "Informatique : Systèmes et Communications"

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

présentée et soutenue publiquement

par

Feryal-Kamila MOULAÏ

Le 13 Décembre 2007

Nouvelles approches pour l'ordonnancement d'applications parallèles sous contraintes de déploiement d'environnements sur grappes

---

Directeurs de thèse :
M Jean-François Méhaut
M Grégory Mounié

Jury

M Georges-Pierre Bonneau	Univ. Joseph Fourier, Grenoble	Président
M Jean-Claude König	Univ Montpellier II	Rapporteur
M Christophe Cérin	Univ Paris XIII	Rapporteur
M Arnaud Legrand	CNRS, Grenoble	Examinateur
M Grégoy Mounié	ENSIMAG Grenoble	Examinateur
M Jean-François Méhaut	Univ. Joseph Fourier, Grenoble	Examinateur (directeur de thèse)

À mes parents et mes trois grandes sœurs
Amel, Amina et Nabila

Ne dites pas « J'ai trouvé la vérité », mais plutôt « J'ai trouvé une vérité »
Khalil GIBRAN, Le prophète

Remerciements

Je dédie cette thèse à mes parents et mes trois sœurs, pour leur amour et leur soutien durant toutes les épreuves que j'ai eu a passer. C'est grâce à eux et pour eux que je suis la.

Je voudrais remercier les membres du jury pour avoir accepté d'évaluer ces travaux. Je tiens à remercier Georges-Pierre Bonneau, professeur à l'université Joseph Fourier, pour m'avoir fait l'honneur de présider ce jury de thèse. Je voudrais, tout naturellement, exprimer mes remerciements à Christophe Cérin professeur à l'université Paris XIII, et à Jean-Claude König, professeur à l'université Montpellier II, pour leurs rapports détaillés témoignant de l'intérêt porté à ces travaux.

Merci aussi à Arnaud Legrand, chargé de recherche CNRS, d'avoir accepté de participer à ce jury de thèse, mais également pour sa disponibilité et pour partager sa passion de la recherche.
Merci à mon directeur de thèse Monsieur Jean-François Méhaut pour avoir soutenue jusqu'au bout.

À toi Radja la meilleur amie qu'une personne puisse avoir, merci pour avoir exister et fait partie de ma vie.

A mes amis, Fethi, Lyes, Sarah, Lina et Linda pour les bons moments passer ensemble, leur soutien et leur énergie.

Merci à Isabelle Mullie, mon professeur de Yoga Iyengar, pour m'avoir montrer le chemin et avoir cru en moi lorsque moi, je doutais.

Merci à mes amis et collègues du laboratoire ID-IMAG, à toi Krzysztof pour les bon moments passer ensemble dans le bureau 113, à toi Éstelle pour ta grande sagesse, à toi Lucas pour ton soutien, et à toi Max pour ton écoute.

Table des matières

Liste des tableaux

Table des figures

Liste des Algorithmes

Introduction

De nos jours, pour réduire les coûts du développement et des essais réels, de nombreux domaines industriels ont recours à des simulations de phénomènes physiques par des modèles mathématiques. Certains domaines particuliers comme par exemple l'imagerie médicale, la météo et l'annotation du génome, nécessitent l'analyse de très grande quantité de données. Cette énorme quantité d'information générée nécessite une infrastructure informatique de plus en plus performante pour analyser et extraire l'information pertinente. Il est alors essentiel de pouvoir acquérir, traiter, stocker, récupérer, analyser et disséminer cette information de la meilleure façon possible. On peut dire qu'aujourd'hui les besoins en calcul scientifique ne font que croître et dépassent largement les possibilités des microprocesseurs actuels.

Dans les années 90, le calcul sur la grille a été introduit (plus communément connu sous l'appellation «grid computing» [FK03]). En théorie, le calcul sur les grilles permet d'exploiter des ressources de calcul, de stockage et de visualisation. Ces ressources sont distribuées sur différents sites géographiques interconnectés par un réseau à grande échelle. Évidement, en pratique l'utilisation effective d'une telle plate-forme soulève des questions liées à l'administration, l'allocation, la volatilité de ressources distribuées et hétérogènes, la sécurité des données, et encore bien d'autres problématiques qui donnent actuellement lieu à de nombreux travaux scientifiques de par le monde.

La recherche française est très active dans ce domaine et se structure pour une bonne partie autour du programme national GRID'5000 initié fin 2003 par le Ministère de la Recherche et des Nouvelles Technologies. Ce programme, vise à promouvoir la mise en place d'une grille expérimentale de recherche constitué d'une dizaine de centres équipés d'ordinateurs multiprocesseurs. L'objectif d'une telle plate-forme est de fournir aux scientifiques les moyens expérimentaux pour mener à bien des recherches dans le domaine des grilles.

Contexte

Grappes dans Grid'5000

Grid'5000 est une plate-forme expérimentale qui a été conçue pour répondre aux demandes des scientifiques en ce qui concerne la puissance de calcul et la capacité de stockage. Elle rend également possible l'accès et l'échange d'importantes bases de données à travers Internet.

Actuellement, Grid'5000 compte 2400 processeurs répartis géographiquement de manière relativement équilibrée sur un ensemble de neuf sites. Chaque site de Grid' 5000 est constitué d'une ou plusieurs *grappes*. Une *grappe* est structurée en un ensemble homogène de machines (appelées nœuds) connectées par un réseau dédié. Les travaux de recherche présentés dans la suite de ce document seront relatifs à des plates-formes parallèles de type *grappes*.

Les problèmes qui apparaissent avec cette nouvelle architecture sont souvent liés en premier temps à la gestion de ressources, plus précisément les problèmes d'ordonnancement, et en second temps aux déploiements d'environnements. Pour le premier cas, il s'agit de déterminer pour un calcul, une allocation spatio-temporelle sur une plate-forme parallèle. La théorie de l'ordonnancement est apparue dans les années 1950, mais les caractéristiques des nouvelles grilles de calcul sont très différentes de celles des calculs sur ordinateurs du XX-ème siècle. Pour le second cas, il s'agit de déployer un environnement spécifique au calcul sur les machines dédiées, ce qui nécessite l'élaboration de nouveaux modèles et algorithmes.

Modèle de tâches et ordonnancement

L'étude des systèmes de gestion de ressources nécessite 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 cette raison qu'a été introduit le modèle de tâches parallèles, dans lequel l'unité élémentaire de calcul est l'application. Ces tâches parallèles peuvent nécessiter plus d'un processeur pour s'exécuter et un temps d'exécution fixé (estimation). Chacune de ces tâches représente un calcul complet, on suppose alors dans ce modèle qu'elles sont indépendantes. Le problème d'ordonnancement, bien qu'ayant ses origines dans le domaine de la recherche opérationnelle (optimisation combinatoire), est aussi de plus en plus utilisé dans le contexte du parallélisme et des réseaux.

Le problème traditionnel d'ordonnancement des tâches sur des machines parallèles vise à minimiser le temps d'exécution global des tâches. Il faut noter aussi, qu'il y a trois phases principales pour l'ordonnancement dans la grille [FK03]. La

première phase détermine les différentes ressources potentielles (processeurs) et la deuxième phase détermine pour chaque tâche selon ses besoins le meilleur ensemble des ressources. Au cours de la troisième phase, la tâche est exécutée, les résultats transférés et la mémoire libérée.

La plus grande difficulté réside pendant la deuxième phase. Déterminer les paires (tâche, allocation) est un problème $\mathcal{NP}$-complet [D.F89]. L'un des thèmes principaux de ce travail a donc été l'étude de la gestion efficace de ressources et plus particulièrement l'ordonnancement de tâches sur plates-formes parallèles.

Déploiement d'environnements

Afin de résoudre les problèmes d'homogénéité des environnements logiciels parmi les nœuds des *grappes*, de reproductivité et de maintenance, il est nécessaire de développer la possibilité de déployer un environnement pour chaque nœud. Un environnement contient différentes couches logicielles comme un système d'exploitation, des bibliothèques, des logiciels personnalisés, des systèmes de fichiers, des intergiciels (middleware) et des applications. Généralement, les opérations de déploiement sur une *grappe* ont lieu seulement à l'heure de la phase de démarrage ou de la phase de maintenance de la *grappe*. La question fréquemment posée est «comment un utilisateur peut-il faire ses calculs sur une *grappe* si l'environnement disponible n'est pas approprié ?».

En fait, cette approche présente un nouveau mode d'exploitation des *grappes*. L'utilisateur ne doit pas faire des modifications sur chaque nœud d'une *grappe*, mais il doit simplement déployer l'environnement des nœuds avant de mener ses expériences.

Le processus de déploiement n'est pas sans conséquence. Il peut causer le vieillissement accéléré des disques par le redémarrage excessif des machines, ou l'arrêt des calculs causé par l'échec de la phase de déploiement.

Il est donc nécessaire de proposer une approche qui minimise les risques de pannes des nœuds et qui permette ainsi un déploiement d'environnements rapide et fiable.

Optimisation multicritère

Dans le cadre de l'optimisation combinatoire classique, la qualité d'une solution est fréquemment mesurée à l'aide d'un critère unique (ou fonction objectif). Cependant, la prise de décision est rarement faite selon un seul critère. Afin d'apprécier la qualité d'une solution, il est nécessaire de prendre en compte plusieurs critères. Ces critères sont souvent conflictuels [CS02, EG00, Ehr01]. L'optimisation combinatoire multicritère tire sa particularité de l'analyse simultanée de plusieurs critères dans un

processus d'optimisation. Cela permet de modéliser de façon plus réaliste un problème concret. Si la notion d'optimum est claire lorsqu'une seule fonction objective est considérée, ce n'est pas le cas avec plusieurs critères. Les conflits qu'engendre l'étude de plusieurs objectifs écartent en général l'existence d'une solution réalisable satisfaisant au mieux tous les critères. On définit ainsi un optimum en optimisation combinatoire multicritère, comme étant un ensemble de solutions deux à deux incomparables [Par96]. Toutefois, la détermination de ces solutions reste problématique et cela est dû principalement au fait que ces problèmes sont $\mathcal{NP}$-difficiles, c'est-à-dire que l'on ne peut pas espérer trouver une solution exacte en un temps raisonnable de calcul. Il est donc naturel que l'approximation se soit imposée en optimisation combinatoire multicritère.

La recherche d'un algorithme d'approximation avec garantie de performances est un domaine très important [Hoc96]. En approximation monocritère, le rapport d'approximation ρ permet de mesurer l'écart entre les valeurs d'une solution approchée notée A et la valeur d'un optimum notée par OPT. On a alors $A \leqslant \rho OPT$ ou $A \geqslant \rho OPT$ selon que le problème est un problème de minimisation ou de maximisation. Un algorithme polynomial ρ-approché pour un problème de minimisation (resp. maximisation) retourne une solution dont la valeur ne dépasse pas ρOPT où $\rho \geqslant 1$ (resp. dépasse ρOPT où $\rho \leqslant 1$).

Transposer la notion d'algorithme d'approximation avec garantie de performance au domaine multicritère n'est pas direct car entre autre, la définition d'un optimum n'y est pas unique. De plus, une telle définition dépend de l'approche suivie.

Sujet de la thèse et contributions

L'objectif de la thèse est de définir de nouvelles techniques d'ordonnancement de tâches parallèles qui soient couplées avec le déploiement d'environnement. D'un point de vue plus concret, c'est de proposer aux utilisateurs de *grappes* la possibilité de soumettre au gestionnaire de ressources des programmes (tâches) et d'associer pour chaque requête un environnement. Cet environnement peut être un système d'exploitation, une bibliothèque ou des fichiers de données. À partir d'une image système déployée et le système redémarré, l'ordonnanceur tentera de maximiser le nombre de tâches s'exécutant avec un environnement sur la *grappe* et minimiser ainsi le nombre de déploiements qui peuvent avoir des conséquences sur la fiabilité du processus d'exécution des tâches.

Plan de la thèse

Le document est organisé en cinq chapitres. Nous commencerons dans le chapitre 1, par présenter le contexte dans lequel s'est effectué ce travail : Grid'5000 avec sa description matérielle, son utilisation et sa spécificité. Nous décrirons le processus de déploiement, et les différents problèmes liés au déploiement d'environnements sur les *grappes* de Grid'5000. Cela nous permet de poser d'une manière pratique le problème étudié dans ce travail de thèse.

Toutes les notations et concepts de base sont présentés dans le chapitre 2. On peut distinguer dans ce chapitre deux parties. La première partie est consacrée au problème d'ordonnancement monocritère, ce qui nous permet de définir les différents modèles de tâches parallèles, de faire un tour d'horizon autour du problème d'ordonnancement et de définir les différentes approches utilisées (comme les algorithmes de liste). Dans la seconde partie du chapitre, nous définissons le problème d'ordonnancement bicritère. Cela permet de passer en revue un certain nombre de définitions propres à l'optimisation multicritère et de dresser un panorama des différentes approches utilisées pour aborder ce genre de problème.

Le chapitre 3 est consacré aux résultats nouveaux apportés par cette thèse. Nous nous pencherons d'abord sur le problème d'ordonnancement bicritère avec déploiement. Afin de bien placer le contexte de notre approche, nous ferons un état de l'art dans le domaine de l'ordonnancement multicritère. Nous définissons ainsi «L'approche budget relaxée» basée sur l'approche budget avec relaxation des contraintes d'optimalités pour les deux critères. Cette approche nous permet de définir une rapport d'approximation (α, β)-budget-approché-relaxé pour la solution engendrée. On présente un algorithme basé sur cette approche que l'on note par **GLS** pour *Group list scheduling*. Afin de minimiser le nombre de déploiements d'environnement, la première phase de l'algorithme consiste à regrouper les tâches utilisant le même environnement sous forme d'une boîte dont la largeur est fixée par un nombre de processeurs et la longueur par un temps d'exécution. Les blocs ainsi formés sont ordonnancés suivant un algorithme de liste qui donne une bonne garantie de performance pour minimiser le second critère du problème c'est-à-dire le *makespan*. Ainsi, **GLS** fournit une solution $(4, 2)$-approche-budget-relaxée pour le problème d'ordonnancement bicritère.

Dans le chapitre 4, nous nous penchons sur le problème de l'approximation de Pareto par le biais de notre algorithme **GLS**. Il s'agit de déterminer à l'aide de **GLS**, un ensemble de solutions qui dominent approximativement toutes les autres solutions

et qui soient au plus $(4 + \epsilon, 2)$ de la courbe de Pareto.

Le chapitre 5, décrit le travail d'implémentation, de vérification et de comparaison entre **GLS** et différentes heuristiques de liste. Nous présentons dans un premier temps le modèle d'instance qui caractérise les tâches rigides sur une plate-forme parallèle. Dans un second temps, nous comparons **GLS** à des heuristiques de listes, afin d'analyser et de valider les performances de **GLS**.

Chapitre 1

Contexte du travail et problématique

Ce chapitre présente le contexte technologique de notre travail. Il s'agit de présenter d'abord la grille Grid'5000 avec son architecture matérielle et ses outils logiciels pour ainsi définir la problématique liée au déploiement d'environnements au niveau de ses grappes de calcul.

1.1 Les grilles

En 1998, Ian Foster et Carl Kesselman ont défini le paradigme de la grille informatique dans leur ouvrage «The grid : Blueprint for a New Computing Infrastructure» [FK03]. Ils comparent les services et ressources informatiques à l'énergie électrique produite par des centrales thermiques ou nucléaires. L'utilisation d'une grille est comparée à celle d'un usagé qui branche et qui ne sait pas d'où provient l'énergie alimentant son appareil. L' accès à une grille devrait être aisé et transparent quelque soit la puissance nécessaire et la complexité des équipements matériels et logiciels mis en œuvre. L'idée de connecter et partager des ressources informatiques dispersées était déjà présente dans les années 1960 avec l'avènement des premiers systèmes distribués. Récemment, les avancées technologiques ont donné à cette idée des perspectives relativement concrètes. Cependant, la réalisation des grilles informatiques se heurte encore à de nombreuses difficultés. On constate des difficultés d'ordre technique car il s'agit de faire communiquer et coopérer des matériels et logiciels distants et hétérogènes, de gérer et de distribuer efficacement les ressources cumulées du réseau, de mettre au point des environnements de programmation adaptés au caractère diffus et parallèle de l'exécution des tâches confiées à la grille, etc... On

est aussi face à des difficultés de nature sociologique, voir économique ou politique dans la mesure où la constitution d'une grille suppose de convaincre plusieurs entités administratives - un organisme public, entreprises privées ou individus - de mettre leurs propres ressources à la disposition d'une entité collective. Cette coopération met en avant de nouveaux problèmes, comme par exemple les problèmes de sécurité. Si chacun des nœuds d'une grille possède des données ou des logiciels qu'il juge confidentiels et ne souhaite pas les communiquer il faut définir des politiques pour garantir la confidentialité par des techniques appropriées à la sécurité des échanges au sein d'une grille. Le concept de la grille peut englober des architectures matérielles et logicielles très différentes en fonction des objectifs recherchés. On identifie deux classes de plate-formes différentes qu'il convient de séparer afin d'éviter de compromettre les performances de chacune d'elles. Il s'agit des grilles de productions et des grilles expérimentales.

1.1.1 Grilles de production

Les grilles de production sont des plates-formes applicatives, qui doivent fournir les mêmes services (heures CPU, temps de réponse) de manière ininterrompue et fiable. L'utilisateur soumet son travail en mode non interactif (batch) et attend que la plate-forme qu'il utilise, ait une architecture stable avec des services de qualité. Un premier exemple de ce type de grille est le projet Globus [Glo01] qui a relié les super calculateurs d'une dizaine de centres de calcul du continent nord-américain. L'objectif est d'obtenir un hyper calculateur parallèle et virtuel, qui offre la possibilité à chaque centre de soumettre des calculs en utilisant la puissance de tous les autres. En Europe, la grille *Data Grid* [Dat01] destinée à la physique des particules offre le même type de services. L'objectif avec cette grille est de stocker et d'analyser les nombreux péta-octets [1] de données que produit le collisionneur de particules LHC du CERN.

1.1.2 Grilles expérimentales

Les grilles expérimentales sont motivées par les travaux de recherche expérimentale. Il s'agit d'une part de recherches en informatique (systèmes distribués, systèmes répartis, calcul parallèle, réseaux, base de données, fouille de données). Ce type de plate-forme n'offre pas de garantie de service, dans la mesure où des modifications peuvent être apportés sur l'infrastructure ou sur le système d'exploitation, afin d'étudier de nouveaux concepts ou algorithmes, de tester un nouvel intergiciel (*mid-*

[1] 1 Péta-octets = 10^{15} octets, l'équivalent d'environ 100 milliards de pages de livres

dleware), ou mettre en œuvre un nouveau protocole réseau. L'administration des grilles de recherche est décentralisée. Les plate-formes informatiques expérimentales, sous toutes leurs formes ont largement contribué à l'avancement de la recherche. Une des applications la plus marquante est celle des bases de données distribuées qui sont passées dans le monde industriel avec le succès qu'on connaît pour ORACLE, compagnie liée à la machine N-CUBE. Le I-cluster (qui n'est pas une grille expérimentale) de Grenoble (ID-IMAG) a permis à une large communauté de chercheurs de faire des expérimentations sur une grappe de plusieurs centaines de processeurs. Cette machine a aussi donné la possibilité de changer d'échelle aux chercheurs de l'INRIA, STIC-CNRS et à d'autres disciplines applicatives.

1.2 Grid'5000

À l'origine, Grid'5000 [Gri06] est le nom d'un programme français initié par l'ACI Grid. Ce programme, d'envergure nationale, vise à promouvoir la mise en place d'une grille expérimentale de recherche constituée d'une dizaine de centres équipés de processeurs communs, ceux du modèle PC. L'objectif d'une telle plate-forme est de fournir aux scientifiques les moyens expérimentaux pour mener à bien des recherches dans le domaine des grilles.

1.2.1 Grappes dans Grid'5000

Actuellement, Grid'5000 compte 2400 processeurs sur les 5000 prévus. Ils sont répartis de manière relativement équilibrée sur un ensemble de neuf sites : Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse. Chaque site de Grid'5000 est constitué d'une ou plusieurs *grappes* ou *clusters*. Une *grappe* est constituée d'un ensemble homogène de machines (appelées nœuds) connectées par un réseau dédié. Ces nœuds sont des machines modernes équipées en biprocesseurs d'AMD (Opterons), de biprocesseurs Intel (Itanium/Xeon) ou de processeurs IBM (PowerPc) cadencés à des fréquences d'environ 2 GigaHertz. Elles disposent également d'un espace de mémoire vive de l'ordre du gigaoctet et de disques durs de capacité supérieure à 70 gigaoctet (IDE/SATA/SCCI). Au niveau de la connectivité, elles sont dotées d'interfaces réseaux Gigabits en standard ce qui leur permet d'avoir une connectivité intrasite de 1 Gigabits. Parmi les nœuds qui composent la *grappe*, on distingue un nœud principal (une frontale) qui agit comme une passerelle avec l'extérieur. Ces serveurs frontaux au niveau de la *grappe* offrent un certain nombre services généraux. On trouve un système de fichier (NFS), un gestionnaire de

ressources *batch scheduler* OAR et un service d'identification (LDAP). L'architecture matérielle d'une *grappe* change d'un site à un autre. Néanmoins les services sont les mêmes (et peuvent être regroupés dans un seul serveur qui sert aussi de frontale). La figure 1.1 montre cette organisation au niveau d'une *grappe* de Grid'5000. Il convient de rappeler que notre problématique de recherche se situe au niveau de l'étude du comportement de la *grappe* de calcul et non du comportement au niveau global de Grid'5000.

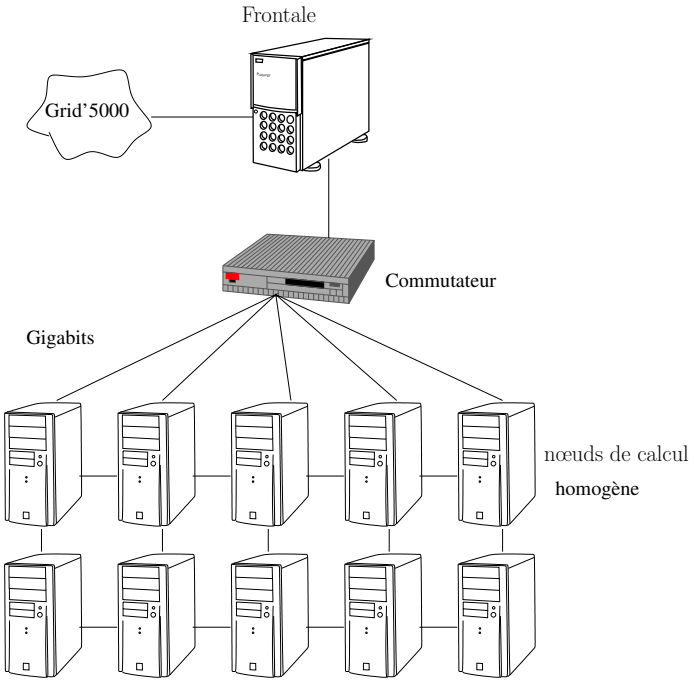

FIG. 1.1 – Représentation d'une *grappe* de Grid'5000

1.2.2 OAR

OAR est un outil de gestion de ressources *batch scheduler* développé au sein du laboratoire ID [OAR06, CCG+05]. Les nœuds de calcul constituent des ressources qui doivent être partagées par différents utilisateurs. Ces ressources ne peuvent être utilisées qu'après avoir obtenu l'autorisation du serveur OAR. Ce serveur reçoit toutes les requêtes des utilisateurs et s'occupe de la surveillance de tous les nœuds de calcul afin de garantir que les ressources allouées à un utilisateur à un instant donné soient bien disponibles. L'algorithme d'ordonnancement implémenté au niveau d'OAR est du

type «FIFO», *First In, First Out* avec un backfiling conservatif [OAR06, CCG+05]. Cependant, il a été montré que ce type d'algorithmes peut conduire à une mauvaise performance et à une faible utilisation des ressources. Dans ses travaux de thèse, Lionel Eyraud [Eyr06] a implémenté un algorithme de réservation de ressource au niveau d'OAR basé sur l'algorithme de liste pour des modèles de tâches modelables.

1.2.3 Kadeploy

Kadeploy est un outil de déploiement qui a été conçu et développé au sein du laboratoire ID-IMAG [Kad06]. L'idée est de définir un outil permettant de déployer un environnement homogène sur les nœuds de calcul au niveau des *grappes*. Déployer un environnement c'est déployer un système d'exploitation, une bibliothèque, un intergiciel (middleware) et des logiciels nécessaires pour l'exécution de calcul. Le but est de proposer aux utilisateurs de Grid'5000 un outil de déploiement d'environnements sur chaque nœud réservé pour leur expérimentation. L'architecture de Kadeploy est centrée sur une base de données et un système de diffusion.

Base de données

La base de données regroupe les informations nécessaires pour le déploiement. Elles couvrent toutes les informations concernant la configuration d'un nœud (le schéma de partition, environnement déployé sur toutes les partitions, les droits sur les partitions). Elle contient aussi les informations concernant les différents environnements déployés sur les partitions (le noyau, système de fichiers). Elle permet une identification (login) de l'utilisateur pour la session de déploiement.

Mécanisme de diffusion

Comme nous l'avons rapidement indiqué plus haut, Kadeploy est basé sur deux systèmes de diffusion [BCC+06]. En premier lieu, il utilise l'outil «*sentinelle*» qui permet l'exécution d'une commande depuis un serveur (PXE, LDAP, TFTP, DHCP) ou d'une machine de travail vers tous les nœuds. Il s'agit d'un outil d'exploitation de grappes. La *sentinelle* lance une connexion distribuée (en *ssh* ou *rsh*) entre le serveur (ou la machine de travail) et les nœuds réservés en utilisant une topologie d'arbre. Une fois la phase de connexion terminée, l'outil d'exploitation crée alors un nouveau processus pour exécuter les commandes spécifiques sur les nœuds. Kadeploy emploie en second lieu un mécanisme efficace de diffusion de fichiers (qui constituent l'environnement de travail) entre les nœuds d'une grappe, il utilise pour cela un outil

appelé *mcatSeg*. La topologie la plus efficace pour ce type de propagation est une chaîne (pipelinée).

Ainsi, Kadeploy présente un mécanisme de diffusion qui mélange la topologie d'arbre pour la soumission de commandes et une topologie en chaîne pour la propagations des fichiers, voir figure 1.2.

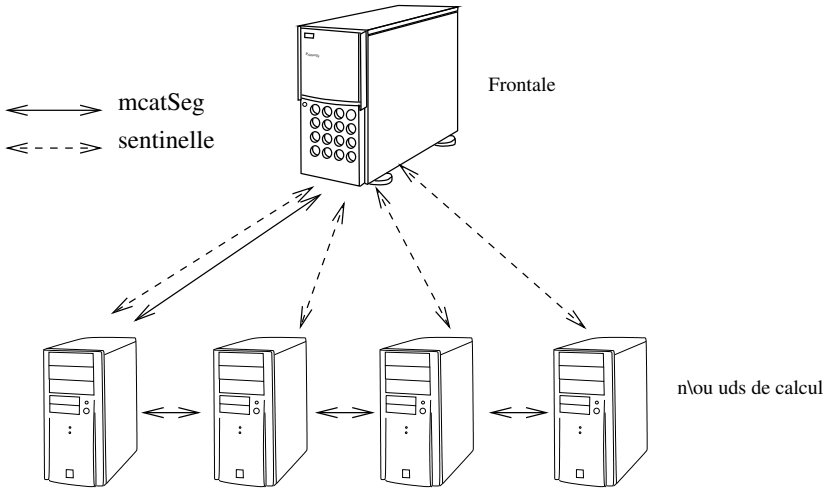

FIG. 1.2 – Le mécanisme de diffusion de Kadeploy

1.3 Processus de déploiement

Le déploiement des environnements n'est pas trivial lorsqu'on considère tous les cas possibles. Les réservations et les déploiements peuvent concerner des centaines de machines. La gestion d'un nombre si important de machines n'est pas à l'abri des problèmes matériels. Cette phase se déroule en trois étapes :

- Réservation des machines de calcul,
- Création de l'environnement,
- Déploiement de l'image (ou des images) sur les machines réservées.

On distingue deux types de réservations. La première concerne la réservation d'une seule machine destinée à la création de l'environnement. Le second type de réservation concerne les machines de travail. Il s'agit de réserver les nœuds sur lesquels on va effectuer les expériences. Les deux réservations sont indépendantes. Néanmoins, elles utilisent le *batch scheduler* OAR.

L'étape de déploiement est basée sur un mécanisme de diffusion en arbre pour les commandes entre la machine de travail et les machines de calcul et un mécanisme en

chaîne pour diffusion des fichiers entre les machines calcul.

1.3.1 Réservation des nœuds

Cette phase consiste à obtenir les ressources nécessaires à l'exécution de l'application à exécuter. Pour cela, des critères de réservation sont spécifiés, tels que le nombre de processeurs ou la durée de la réservation. Pour réserver des nœuds sur un site, on utilise *batch scheduler* OAR [OAR06, CCG+05]. Grâce à cet outil, l'utilisateur peut demander à 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 partir de la date indiquée.

1.3.2 Création d'un environnement

Dans ce contexte, le terme environnement désigne un système d'exploitation et un ensemble de logiciels que configure l'utilisateur en vue de mener ses expériences. Il se présente sous forme d'une image ou archive (*tar*). Afin de faire gagner du temps à l'utilisateur, un certain nombre d'environnements minimaux sont fournis. Ces derniers comportent tous les réglages nécessaires pour les rendre fonctionnels sous Grid'5000. La plupart sont basés sur des distributions Linux [2] mais il en existe aussi pour FreeBSD et Solaris. Finalement, la procédure de création consiste à réserver une machine de travail contenant plusieurs partitions et déployer sur l'une de ces partitions un des ces environnements minimaux. Plus précisément, il s'agit de décompresser l'archive correspondante dans une partition de la machine de travail et de redémarrer (rebooter) la machine sur le système contenu dans l'archive. On rentre alors en fonction les différents protocoles de reboot (PXE, TFTP, DHCP) pour finaliser l'installation de l'environnement. Il est ensuite possible de modifier selon ses besoins l'environnement et de le sauvegarder sous la forme d'une nouvelle archive puis de l'enregistrer dans le système.

1.3.3 Déploiement des environnements

Après la création de l'environnement, commence alors le processus d'identification, de contrôle de permission et les droits sur les machines des calculs effectués par la base de données de Kadeploy. À l'étape suivante, le mécanisme de diffusion entre les différents serveurs (PXE, LDAP, TFTP, DHCP) et les nœuds de calcul est mis en place. Ainsi, le mini système est diffusé et monté sur la RAM de chaque nœud

[2] Debian, Fedora, Ubuntu

de calcul. Il prépare le nœud au déploiement, il est ainsi équipé de tous les outils nécessaires pour le déploiement (mécanisme de diffusion, logiciels, pipes ....). Ainsi il permet de faire les modifications nécessaires sur le disque comme copier l'environnement sur une partition et le configurer pour le déploiement. Un reboot sur ce mini système est indispensable pour chaque nœud. Le processus de déploiement prépare alors la partition sur laquelle l'environnement va être déployé. Il propage ainsi l'image de l'environnement sur les autres nœuds en utilisant le mécanisme de diffusion en chaîne voir figure 1.3. La dernière étape consiste à redémarrer les nœuds sur la partition où l'environnement va être installé. Dans cette étape rentrent en fonction alors les différents protocoles de reboot (PXE, TFTP, DHCP) pour finaliser l'installation de l'environnement.

Ainsi l'utilisateur peut exploiter les nœuds et mener ses expériences. Une fois la session terminée, l'information est inscrite au niveau de la base de données et ainsi transmise au *batch scheduler* OAR. Un redémarrage des nœuds sur la partition de référence (initiale) constitue la dernière étape [BCC+06].

1.4 Problématique

L'objectif premier de Grid'5000 est de donner aux chercheurs de différentes disciplines les moyens expérimentaux pour mener à bien leur travaux. En outre, elle met à leur disposition des outils comme OAR et Kadeploy qui sont nécessaires à la mise en œuvre des applications sur la grille.

La configuration matérielle de Grid'5000 est différente au niveau de chaque site. On trouve des *grappes* de taille différente et au niveau matériel différent (modèle, CPU, mémoire, ...). Néanmoins les nœuds de calcul qui constituent une *grappe* sont identiques.

Cette plate-forme est extrêmement intéressante car elle donne la possibilité de réalisation d'expériences à grande échelle dans un environnement très contrôlé et hautement configurable. Globalement, son fonctionnement est très satisfaisant et prometteur, mais elle connaît néanmoins des perturbations d'ordre materiels et de gestion de ressources.

Le travail de recherche de cette thèse s'inscrit au niveau *grappe* de Grid'5000 donc d'un point de vue homogène.

Notre travail considère une *grappe* avec un certain nombre (noté par m) de nœuds de calcul (processeurs) identiques, reliés entre elles par un réseau Gigabits. Ces machines sont aussi connectées aux différents serveurs qui permettent aux utilisateurs de se connecter, de réserver des nœuds, de déployer et d'exécuter leurs travaux.

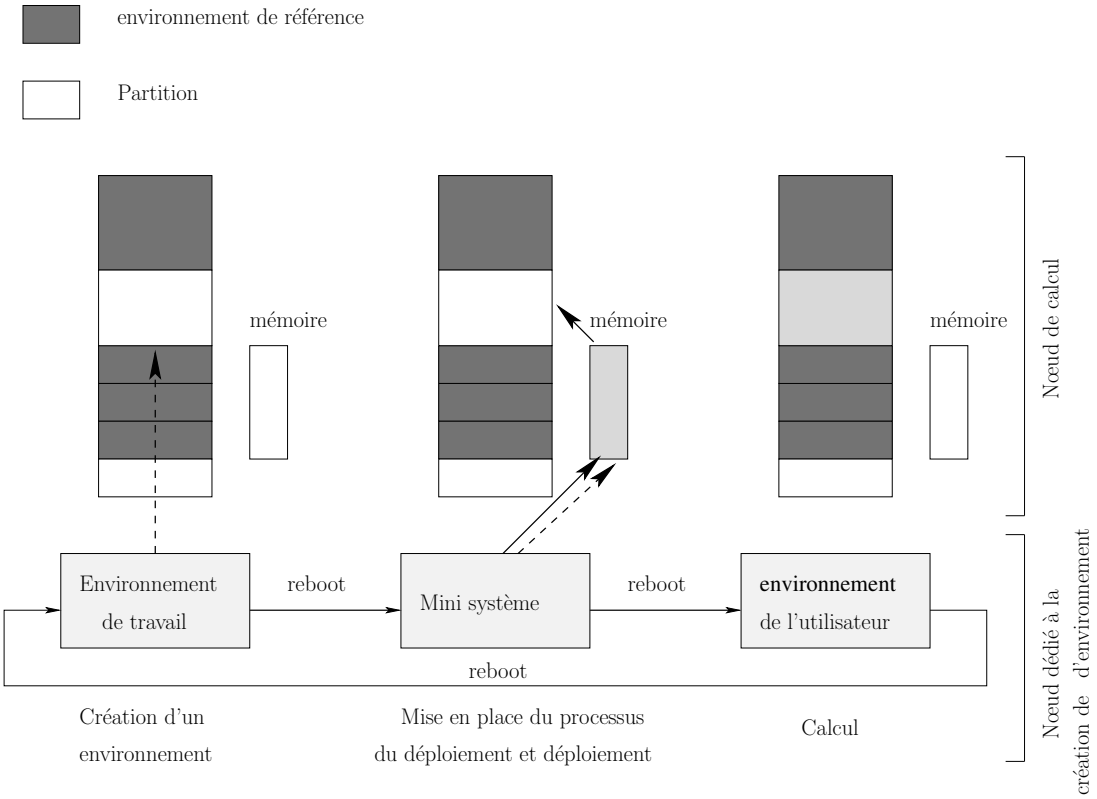

FIG. 1.3 – Les différentes étapes de déploiement sur un nœud

1.4.1 Problématique de déploiement

Comme nous l'avons indiqué plus haut, le processus de déploiement n'est pas sans conséquences sur la fiabilité des nœuds et les calculs. Une fois que l'utilisateur a réservé des nœuds, commence alors le processus de déploiement. La première difficulté que l'on rencontre est la défaillance des machines. En effet, le redémarrage excessif des nœuds à chaque nouveau déploiement peut causer le vieillissement accéléré des disques. Une deuxième difficulté est lié au mauvais démarrage des machines. Un nœud qui a mal été redémarré ne peut pas déployer un environnement sur une de ses partitions puisque des résidus d'un précédent utilisateur peuvent entraver le processus de déploiement. Un troisième problème que l'on rencontre souvent est relatif aux serveurs de boot. Il arrive souvent que ces serveurs soient submergés de travail, cela rend la communication très difficile avec les différents nœuds de calcul et peut causer ainsi leur arrêt.

Si un nœud est en arrêt pour l'une des raisons que l'on vient d'évoquer, il est alors impossible de diffuser l'image de l'environnement à déployer sur les autres nœuds.

Afin de répondre aux exigences de plus en plus croissantes des utilisateurs de Grid'5000 au niveau des *grappes*, nous avons mené une étude qui tente de «minimiser» les problèmes liés au déploiement. Nous avons ainsi étudié ce problème et nous l'avons modélisé sous forme d'un problème d'ordonnancement. Nous avons considéré la minimisation du temps de terminaison des travaux *makespan*. Et en ce qui concerne le problème de déploiement, nous avons défini un nouveau critère. Ce critère comptabilise pour chaque nœud différent (processeurs) le nombre de déploiements effectués pour définir le nombre total de déploiement sur l'ensemble des nœuds.

1.5 Conclusion

Les grilles informatiques sont des plate-formes de calcul à grande échelle, hétérogènes et distribuées. On distingue deux types de grilles, les «grilles de production» et les «grilles expérimentales». Nous avons présenté la distinction à faire entre les deux types de grilles. Ainsi nous avons introduit le contexte de notre travail, *grappe* de Grid'5000 qui est comme nous l'avons souvent rappelé dans ce chapitre, une plateforme expérimentale. Nous avons présenté ces *grappes* d'un point de vue matériel et d'un point de vue fonctionnel, ce qui nous a permis d'introduire les différents outils qui font la spécificité de Grid'5000. En effet, Grid'5000 offre à ses utilisateurs la possibilité de soumettre leur travaux en déployant eux mêmes leur environnement sur les nœuds de calcul. Néanmoins, le processus de déploiement n'est pas sans conséquences. Nous avons présenté les différents problèmes engendrés par ce processus.

Dans la suite, nous allons modéliser ce problème sous forme d'un problème d'ordonnancement bicritère. Nous avons défini un nouveau critère qui vise à minimiser le nombre de déploiements et d'offrir aux utilisateurs la possibilité de traiter leur tâches au plus vite.

Chapitre 2

Modélisation

Nous présentons dans ce chapitre, les notations et les définitions propres à l'optimisation multicritère. Nous présenterons également des notions d'optimalité et de complexité, utiles aux chapitres suivants et nous introduirons la modélisation du problème traité dans cette thèse.

2.1 Introduction

Un bon modèle doit avoir deux qualités : être réaliste et simple. Le réalisme est un critère antagoniste à la simplicité. Plus le modèle est simple donc abstrait, moins il est proche de la réalité. À l'opposé, un modèle réaliste est très souvent complexe à utiliser. Il est difficile de raisonner avec un modèle trop complexe.

Une représentation mathématique doit faire ressortir le plus fidèlement possible le comportement global et les principales caractéristiques de la plate-forme : *grappe*.

Nous allons considérer dans cette thèse un modèle de *grappe* composé de m machines identiques (homogènes) reliées entre elles via un réseau. L'objectif de cette architecture est de nous fournir une plate-forme parallèle haute performance pour l'exécution des applications parallèles.

Le problème auquel nous nous intéressons dans cette thèse consiste à décider d'un lieu (processeur) sur notre plate-forme et d'un ordre d'exécution pour les applications parallèles. C'est cette décision que nous appelons *Ordonnancement*. Suivant ce contexte, l'objectif de l'ordonnancement est de prendre une décision judicieuse afin de minimiser la durée totale de l'exécution des applications parallèles, le *makespan*. Nous nous intéressons plus particulièrement aux algorithmes statiques, c'est-à-dire à ceux qui tirent partie de la connaissance fine d'une application à l'aide de données comme le nombre de processeurs utilisés ou le temps d'exécution nécessaire.

Nous nous intéressons dans ce qui suit aux questions liées à la représentation des applications qui utilisent ce type d'information.

Ce chapitre a pour objectif de clarifier les notions, les concepts et les modèles que nous utiliserons tout au long de cette thèse. Il permet également de bien comprendre l'intérêt pratique du problème d'ordonnancement que nous avons abordé durant ces trois années de recherche. Dans un premier temps, nous présentons une formalisation classique du problème d'ordonnancement d'application sur un grappe. Ensuite nous exposerons les différents modèles d'applications parallèles. Nous introduirons ensuite la modélisation du problème que nous avons étudier. Nous ferons un tour d'horizon des travaux déjà effectués sur les problèmes d'ordonnancement de tâches rigides. Nous terminerons ce chapitre en expliquant les spécificités de notre problème et en particulier celles dues au caractère bicritère.

2.2 Notions et concepts

Afin de pouvoir ordonnancer une application parallèle, il est nécessaire d'utiliser une représentation de cette application (qui est donc une donnée du problème d'ordonnancement associé). Cette représentation doit exprimer le parallèlisme présent dans cette application. La représentation la plus couramment utilisée est le graphe de précédence. Plus formellement, soient un graphe orienté sans cycle $G = (V, E)$, et une pondération sur les nœuds du graphe $p(T_i) \in \mathbb{N}$ pour toutes tâches $T_i \in V$. Les arêtes du graphe G représentent les contraintes de précédence : si $(T_i, T_j) \in E$ est une arête, cela signifie que la tâche T_j ne peut pas être exécutée tant que la tâche T_i n'a pas terminé son exécution. La pondération $p(T_i)$ représente le temps d'exécution de la tâche T_i (ou *processing time* en anglais). Un ordonnancement est une fonction qui détermine pour toutes les tâches une date de début d'exécution. Soit la fonction $\sigma : V \mapsto \mathbb{N}$ et soit $\sigma(T_i)$ la date de début d'exécution de la tâche T_i. On définit aussi le temps de terminaison de la tâche T_i, $C_i = \sigma(T_i) + p_i$. La performance d'un ordonnancement pour ce type de données se mesure par le temps total nécessaire à exécuter toutes les tâches de l'application, c'est-à-dire le plus grand temps de terminaison des tâches $T_i \in V$ i.e. $\max C_i$. Cette valeur, que l'on appelle le *makespan* de l'ordonnancement est notée C_{max}. Évidement, le problème d'optimisation lié au problème d'ordonnancement consiste à minimiser le temps total pour exécuter toutes les tâches. Une solution réalisable pour le problème d'ordonnancement doit vérifier les contraintes de précédence mais aussi de ressources, plus formellement, cela s'exprime aussi :

1. $\forall t \in \mathbb{N}, |\{T_i \in V | \sigma(T_i) \leqslant t < \sigma(T_i) + p(T_i)\}| \leqslant m$

2. $\forall (T_i, T_j) \in E,\ \sigma(T_i) + p(T_i) \leqslant \sigma(T_j)$

La première condition indique qu'à chaque instant au plus m machines soient utilisées, la seconde vérifie les relations de précédence entre les tâches. Un ordonnancement *optimal* est un ordonnancement qui vérifie les conditions précédentes avec le plus petit *makespan* noté C^*_{max}. Cette représentation des applications est utilisée depuis les années 70 [CD73]. Comme les problèmes d'ordonnancement sont d'une grande variété, une notation synthétique pour classifier les schémas a été proposé par [Gra69]. Suivant cette notation standard en trois champs $\alpha|\beta|\gamma$ [LMB96], le problème d'ordonnancement de graphe de tâches est noté $P|prec|C_{max}$.

La figure 4.6 illustre un exemple de graphe de tâches avec un ordonnancement possible. Celui-ci est représenté par le *diagramme de Gantt*. Cette représentation qui est la représentation la plus classique est aussi utilisée tout au long de ce document.

L'ordre d'exécution des tâches est contraint par le graphe de précédence. Dans cette représentation, une tâche est un rectangle de largeur 1 et de longueur proportionnelle à son temps d'exécution. Les périodes d'inactivité des processeurs sont représentées en gris dans le *diagramme de Gantt*. Nous conseillons aux lecteurs intéressés par une étude approfondie de l'ordonnancement de se rapporter aux ouvrages [JEE+96] et [M.P95].

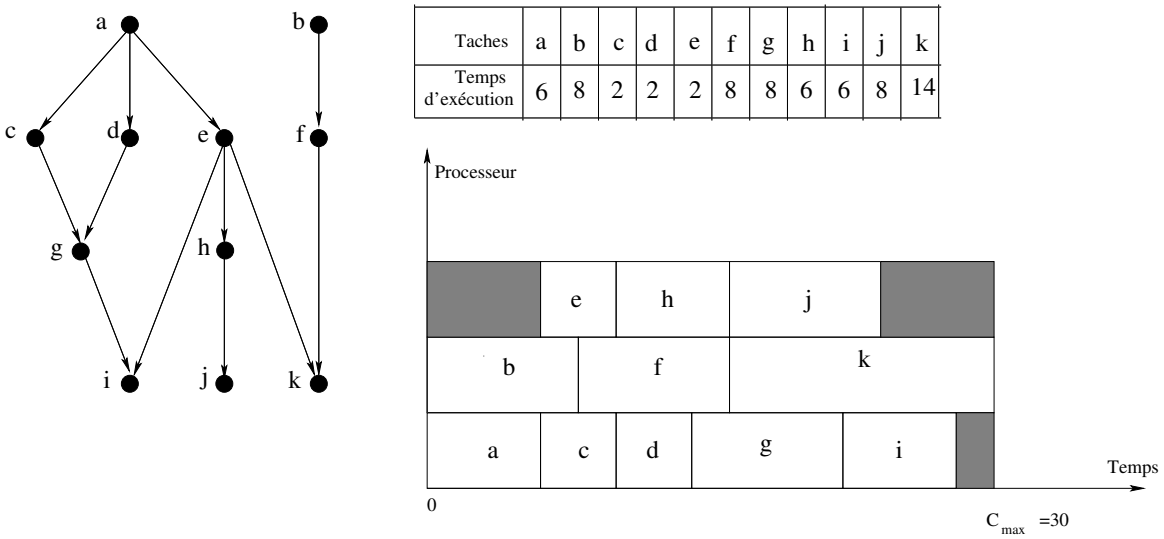

FIG. 2.1 – Exemple d'un graphe de tâches et de leur ordonnancement sur trois processeurs

2.2.1 Tâches parallèles

Dans toute la suite de cette thèse, par souci de simplicité, nous confondrons l'application et la tâche. En pratique une application parallèle est un ensemble de tâches qui sont reliées entre elles par des liens de précédence ou des temps de communication. De manière informelle, dans cette thèse, une tâche parallèle représente un calcul complet, qui contient elle-même suffisamment de parallélisme pour être exécutées par plus d'un processeur [CCG+05]. On suppose, qu'elles sont indépendantes, c'est-à-dire qu'on peut les exécuter dans un ordre quelconque. Il est possible de distinguer trois classes de tâches parallèles. La figure 2.2 représente les différents types de tâches parallèles.

Les tâches rigides

Le nombre de processeurs dont chaque tâche a besoin est spécifiée dans l'instance. La tâche utilise alors tous ces processeurs pendant toute la durée de son exécution. (Classe de tâches étudiée dans cette thèse).

Dans un *diagramme de Gantt*, une tâche rigide est représentée par un rectangle tel que la longueur indique le temps d'exécution et la largeur le nombre de processeurs nécessaire pour l'exécution de cette tâche.

Les tâches modelables

Le nombre de processeurs nécessaire pour une tâche n'est pas fixé par l'utilisateur. L'instance spécifie, pour chaque tâche, un temps d'exécution en fonction du nombre de processeurs possibles. Ainsi, l'algorithme d'ordonnancement utilisé choisit le nombre de processeurs à allouer à chaque tâche. Une tâche utilise le même nombre de processeurs tout au long de son exécution.

Les tâches malléables

Une tâche peut être allouée sur un nombre quelconque de processeurs variable au cours du temps. Le temps d'exécution de cette tâche dépend alors du nombre de processeurs sur lesquels elle est allouée. Évidement comme précédemment, cette allocation est exclusive, c'est-à-dire qu'un processeur n'exécute qu'une tâche à la fois.

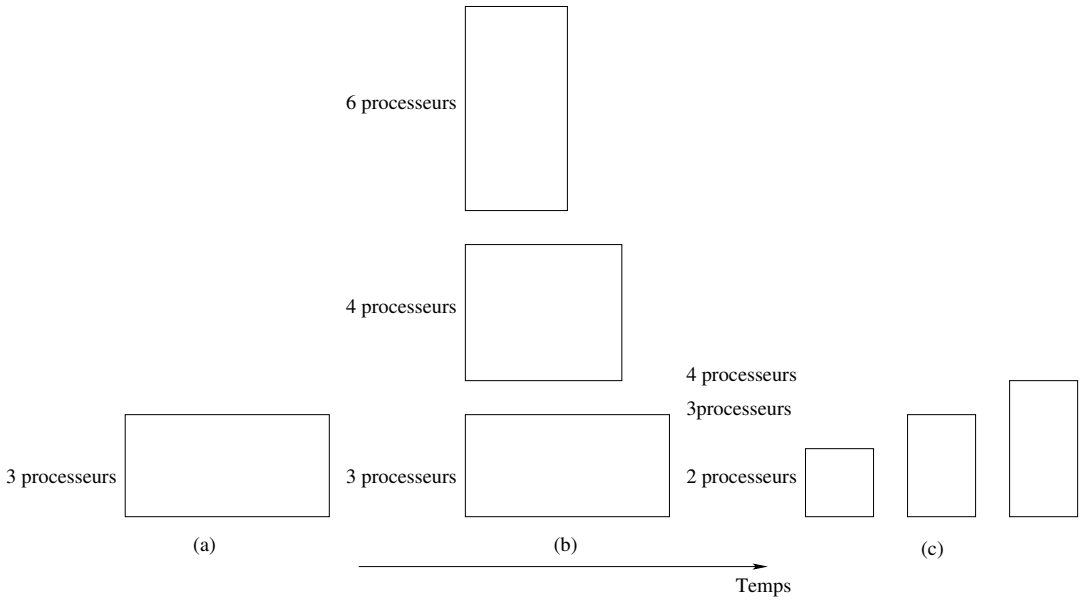

FIG. 2.2 – (a) Tâche rigide, (b) Tâche modelable, (c) Tâche malléable

2.3 Ordonnancement des tâches rigides avec déploiement

Soit un ensemble de tâches $\Gamma = \{T_1, T_2, .., T_n\}$. Les tâches sont indépendantes c'est-à-dire qu'il n'existe pas de liens de précédence ni de liens de communications entre elles. Le temps d'exécution pour une tâche T_i est noté p_i [1]. Les tâches s'exécutent sur une plate-forme composée de m processeurs homogènes connectés entre eux via un réseau. Soit $E = \{e_1, \cdots, e_s\}$ un ensemble de S environnements de travail disponibles sur la plate-forme. Chaque tâche T_i a besoin pour s'exécuter d'un nombre de processeurs q_i ($q_i \leqslant m$) et d'un environnement de travail $e_i \in E$. Cet environnement doit être déployé sur les q_i processeurs pour que la tâche T_i puisse commencer son exécution. Comme précédemment, les tâches ne sont pas préemptibles c'est-à-dire que si une tâche T_i commence son exécution, alors les q_i processeurs dédiés commencent le travail au même moment et ne s'arrêtent qu'après p_i unité de temps. On définit le *travail* w_i d'une tâche T_i, comme la quantité de calcul qu'elle contient,

[1] Nous définissons dans cette thèse les valeurs temporelles comme élément de $\mathbb{N}$. C'est vrai que le temps est continu, mais dans la théorie de la complexité et de l'algorithmique en général, on ne peut pas considérer des problèmes qui manipulent les nombres réels et pour cause, les ordinateurs ne peuvent pas calculer à une précision infinie.

c'est-à-dire la surface d'une tâche dans le *diagramme de Gantt* : $w_i = p_i q_i$.

L'ordonnancement des tâches consiste à trouver pour chaque tâche T_i une date de début d'exécution $\sigma(T_i)$ et une allocation de processeurs libres à cette date, sur la *grappe*. Avant le début de l'exécution de la tâche T_i, l'environnement de travail e_i doit être déployé sur les q_i processeurs nécessaire à l'exécution de T_i. Le temps nécessaire pour déployer un environnement sur un processeur est supposé être nul tout au long de cette thèse.

Nous avons donné jusqu'à présent les paramètres et les contraintes que l'on exprime pour décider quelles sont les solutions réalisables. Pour poser complètement le problème comme un problème d'optimisation, il nous faut des critères qui nous permettent de départager deux ordonnancements valides.

Selon le choix de ce critère, qui dépend des objectifs des utilisateurs, un algorithme d'ordonnancement est alors utilisé. Le *makespan*, ou le temps de terminaison maximal C_{max}, est historiquement le premier critère considéré et le plus étudié en ordonnancement. Ce qui intéresse les utilisateurs dans notre cas c'est de pouvoir exécuter leurs tâches au plus vite, ce qui revient à minimiser le *makespan*. Du point de vue de l'administrateur, c'est de sauvegarder la vie du matériel (processeurs) le plus longtemps possible, et assurer une fiabilité pour le déploiement c'est-à-dire minimiser le nombre de boot nécessaire pour déployer un environnement avant et après l'exécution d'une tâche. Cela revient à considérer un nouveau critère D_{sum} qui représente le nombre total de déploiements pour une instance du problème. La résolution *exacte* des problèmes d'ordonnancement consiste à trouver une solution avec la valeur la plus petite possible pour un critère donné. Une telle solution est dite *optimale*[2]. La valeur de la solution optimale selon le critère considéré sera notée par une étoile en exposant. Ainsi, si C_{max} et D_{sum} sont respectivement le makespan et le nombre de déploiements d'un ordonnancement, alors C^*_{max} et D^*_{sum} sont respectivement les valeurs optimales du makespan et du nombre de déploiements.

2.4 Les problèmes d'ordonnancement des tâches rigides sans déploiement

Trouver un temps raisonnable qui atteint une valeur optimale d'un critère choisi pour un problème donné, peut s'avérer être une tâche très dure. Les problèmes d'ordonnancement appartiennent généralement à la classe des problèmes $\mathcal{NP}$-difficiles, [GJ79] dont on conjecture qu'il faut pour les résoudre un temps qui augmente de

[2]Il faut noter qu'il peut exister plusieurs solutions optimales différentes pour un problème donné

façon exponentielle avec la taille du problème. Une solution pour contourner la difficulté consiste à se tourner vers les *algorithmes d'approximation.*

2.4.1 Algorithmes d'approximation

Intuitivement l'idée des algorithmes d'approximation consiste à *sacrifier* une partie de la qualité de la solution obtenue en échange d'un temps de calcul polynômial. La solution obtenue par les *algorithmes d'approximation* est dite *approchée*, c'est-à-dire que la valeur de cette solution n'est pas très éloignée de la valeur optimale. Afin d'évaluer la performance de tels algorithmes, on définit la notion de *garantie de performance* qui consiste à faire une analyse au pire cas de l'algorithme considéré. Cette *garantie de performance* est ainsi le rapport entre la valeur approchée et l'optimale. D'une façon plus formelle, comme on peut le trouver dans [Leu04] et [Eyr06] :

Définition 1. *Soit un problème $\mathcal{P}$ tel qu'à chaque instance $\mathcal{I}$ correspond un espace $\mathcal{S}(\mathcal{I})$ de solutions réalisables, et pour lequel on cherche $\mathcal{S} \in \mathcal{S}(\mathcal{I})$ qui minimise une certaine fonction $f(\mathcal{S})$.*
Soit $\mathcal{A}$ un algorithme qui à partir d'une instance $\mathcal{I}$, produit une solution réalisable $\mathcal{A}(\mathcal{I})$. On dit que $\mathcal{A}$ est une $\rho-$approximation si pour toute instance $\mathcal{I}$, $f(\mathcal{A}(\mathcal{I})) \leqslant \rho \times f^*(\mathcal{S}(\mathcal{I}))$. La garantie de performance de $\mathcal{A}$ est la plus petite valeur de ρ tel que $\mathcal{A}$ soit une $\rho-$approximation.

Dans les problèmes d'ordonnancement, la comparaison avec une solution optimale est compliquée. Pour cause, la connaissance de cette solution optimale est déjà en soi un problème difficile. La comparaison est alors souvent effectuée par rapport à une valeur que l'on sait inférieure à l'optimale. Dans les preuves sur les ordonnancement de tâches rigides (le modèle de tâches étudié dans cette thèse), les deux bornes inférieures classiques pour le *makespan* sont :

1. Le temps de la tâche la plus longue :

$$\forall T_i \in \Gamma, \quad C^*_{max} \geqslant p_i \tag{2.1}$$

2. Le travail moyen effectué par les processeurs :

$$C^*_{max} \geqslant \Big(\sum_{T_i \in \Gamma} \frac{w_i}{m} \Big) \tag{2.2}$$

2.4.2 Représentation géométrique de l'ordonnancement

À première vue, le problème d'ordonnancement des tâches rigides est lié au problème de «Strip Packing». Ce problème d'ordre géométrique a été étudié dans les années 80 par [RCR80]. On peut modéliser le problème comme suit : soit une bande de largeur fixe (m par exemple) et de longueur infinie. Étant donné n rectangles de largeur inférieure à la largeur fixée de la bande et de longueur différentes, le but est de trouver une découpe pour les rectangles de sorte à couvrir la surface du sol en un minimum de longueur. On voit bien que ce problème ressemble grandement à celui de l'ordonnancement des tâches rigides pour minimiser le makespan. Cependant, il y a une contrainte supplémentaire à prendre en compte pour le problème de «Strip Paking». Les rectangles ne se découpent pas, ils sont considérés comme étant des blocs. En revanche, le problème des tâches rigides étudié dans cette thèse n'impose aucune *contiguïté* sur les processeurs alloués à une tâche, seulement une contrainte sur le nombre de processeurs alloués à un instant donné. Dans sa thèse Dutot [Dut04] montre qu'un ordonnancement valide pour un problème d'ordonnancement de tâches rigides n'est pas forcément une découpe valide pour un problème de «Strip Packing». Il montre d'ailleurs l'existence d'une donnée pour laquelle la valeur du *makespan* optimale pour le problème d'ordonnancement des tâches rigides est plus faible que la longueur optimale pour le problème de «Strip Packing». Steinberg [A.S97] propose un algorithme avec une garantie de performance de 2 pour ce problème. Son approche fut de séparer les rectangles en deux catégories, les «longs» et les «larges». Il a ainsi formé des groupes pour chaque catégorie ce qui a permis un agencement efficace des groupes.

2.4.3 Tour d'horizon sur les problèmes d'ordonnancement des tâches rigides

Comme la plupart des problèmes d'ordonnancement qui optimisent le *makespan*, notre problème est inclus dans le traditionnel problème $P||C_{max}$ qui consiste à ordonnancer des tâches rigides sur m processeurs où m fait partie de l'instance. En effet, $P||C_{max}$ est équivalent au problème de 3-PARTITIONS, connu pour être $\mathcal{NP}$-difficile au sens fort. Même si l'on considère le problème d'ordonnancement des tâches rigides où le temps d'exécution est unitaire, c'est-à-dire $P|q_i, p_i = 1|C_{max}$, ce problème est aussi montré comme étant $\mathcal{NP}$-difficile par [E.L81] la preuve est basée sur une réduction au problème $P||C_{max}$, indépendamment [BDJ86] l'ont montré avec une réduction pour 3-PARTITIONS.

Pour m fixé (ne faisant pas partie de l'instance), le problème $P_m|q_i|C_{max}$ est

$\mathcal{NP}$-difficile, inclus dans le problème $P_2||C_{max}$ qui est équivalent au problème connu pour être $\mathcal{NP}$-difficile i.e. PARTITION [MD79]. Pour $m = 5$, $P5|q_i|C_{max}$, ce problème d'ordonnancement des tâches rigides est déjà $\mathcal{NP}$-difficile. Lorsque $m = 2, 3$, le problème est résolus en temps pseudopolynomial [DL89]. Concernant le problème $P4|q_i|C_{max}$ s'il est $\mathcal{NP}$-difficile reste une question sans réponse. Quand le temps d'exécution est unitaire le problème $Pm|q_i, p_i = 1|C_{max}$ a été résolu en un temps $\mathcal{O}(n)$ en utilisant la programmation linéaire [BDJ86]. Ils définissent des sous-ensembles de tâches Γ^j tel que $\forall T_i \in \Gamma^j$, T_i utilise j processeurs pour son exécution et $|\Gamma^j| = n_j$. Leur approche était de définir un vecteur $b = [b_1, b_2, .., b_k]$, appelé vecteur de processeurs réalisables, tel que chaque élément du vecteur b_j représente le nombre de processeurs m_j disponible à un instant donné pour exécuter des tâches du sous-ensemble Γ^j. Ce qui est équivalent à trouver une combinaison linéaire entre les vecteurs $n = [n_1, n_2, .., n_k]$ et $b = [b_1, b_2, .., b_k]$.

$$\min \sum_{j=1}^{k} x_j$$

$$\sum_{j=1}^{k} x_j b_j = m$$

Le lecteur intéressé par ces questions trouvera un éventail d'information très large concernant les tâches rigides dans le Handbook of scheduling [Leu04].

2.4.4 L'ordonnancement de liste

Historiquement, l'un des premiers algorithmes d'approximation proposé, l'a été dans le contexte de l'ordonnancement. Il s'agit des algorithmes de liste introduit par Graham [Gra66]. Remarquons aussi qu'à cette époque la notion de problème $\mathcal{NP}$-Complet n'existait pas encore ; elle sera introduite peu après par Cook [Coo71].

Les algorithmes de liste sont répertoriés dans la catégorie des algorithmes glouton dont le principe est de prendre des décisions au fur et à mesure sans jamais revenir sur les décisions précédentes. L'idée de l'algorithme est basée sur l'allocation gloutonne des tâches aux processeurs : un processeur ne peut être inoccupé à un instant t s'il peut exécuter une tâche à ce moment là. Ce principe est originellement appelé la «règle de Jackson» [Jac55].

[Gra69] a étudié l'algorithme de liste pour les tâches rigides avec une contrainte de ressources. Son modèle est représenté par un ensemble de n tâches rigides, et s types de ressources différentes. Chaque tâche T_i nécessite une quantité r_{ij} de la ressource

j. Il suppose qu'à chaque instant, il n'est pas possible d'utiliser plus d'une quantité prédéterminée de chaque ressource. Il montre alors que l'algorithme de liste est $s+1$-approximation pour le makespan. Dans le cadre de cette thèse nous considérons des tâches rigides avec une seule ressource i.e. les processeurs, $s = 1$. Nous allons ainsi montrer que nous obtenons une garantie de performance d'au plus 2.

Théorème 2.4.1. *[GG75] Tout algorithme de liste ordonnance une succession de tâches rigides multiprocesseurs avec une garantie de performance égale à 2 pour le makespan.*

Démonstration. Nous allons faire une démonstration par contradiction. Soit $r(t)$ la quantité de processeurs utilisés par les tâches à l'instant t. Une des conséquences des algorithmes de liste est :

$$\forall t_1, t_2 \in [0, C_{max}) \quad t_1 \leqslant t_2 - C^*_{max}$$

$$r(t_1) + r(t_2) > m \tag{2.3}$$

Si l'équation 2.3 n'est pas vérifiée cela revient à dire que les tâches qui sont ordonnancées à l'instant t_2 pouvaient être exécutées à l'instant t_1

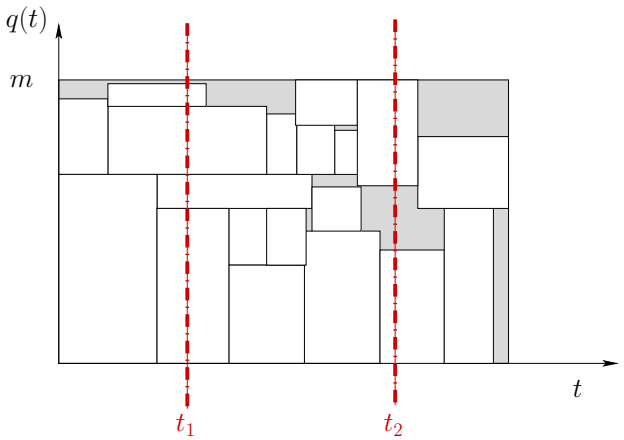

FIG. 2.3 – Contrainte de ressources de Graham

Il est important de noter que le temps d'exécution de toutes les tâches et que la longueur de la surface utilisée par les tâches sont inférieurs C^*_{max} 2.1, 2.2 ce qui correspond aux bornes inférieures du makespan optimal.

Supposons que $C_{max} > 2C^*_{max}$.

$$\sum_{i=1}^{n} p_i q_i = \int_0^{C_{max}} r(t)dt \leqslant mC^*_{max} \tag{2.4}$$

ce qui est équivalent à :

$$\int_0^{2C^*_{max}} r(t)dt + \int_{2C^*_{max}}^{C_{max}} r(t)dt \leqslant mC^*_{max} \tag{2.5}$$

$$\int_0^{C^*_{max}} \underbrace{(r(t) + r(t+1))}_{>m} dt + \int_{2C^*_{max}}^{C_{max}} \underbrace{r(t)}_{>0} dt \leqslant mC^*_{max} \tag{2.6}$$

Le deuxième terme de l'inéquation 2.6 reste positif mais le premier terme est strictement plus grand que m ce qui amène à une contradiction.

□

2.5 Le problème d'ordonnancement bicritère

Notre but dans cette thèse est de trouver une fonction d'ordonnancement efficace pour que les tâches rigides puissent s'exécuter en un temps minimum sur une *grappe* de calcul, et aussi, assurer aux utilisateurs le bon déroulement de ces tâches en minimisant le nombre de déploiements. Il s'agit d'un problème bicritère, tel que les objectifs à optimiser sont le temps de terminaison de toutes les tâches i.e. *makespan* et le nombre de déploiements des environnement nécessaire pour l'exécution des tâches, D_{sum}.

La figure 2.4 montre un *diagramme de Gantt* représentant 2 processeurs, 3 environnements (représentés par des rectangles avec des différents motifs) et 4 tâches. Les tâches sont séquentielles et nécessitent chacune un environnement. Le cas (a) montre que si le nombre de déploiements n'est pas minimisé, le processeur $P2$ comptabilise l'installation de deux environnements différents pour les tâches T_1 et T_4 pour un *makespan* $C_{max} = 6$. Le cas (b) montre par contre que si le nombre de déploiements est minimisé, les tâches utilisant le même environnement c'est-à-dire T_1 et T_3 sont exécutées successivement sur le même processeur.

Remarque 1. *Comme le montre la figure 2.4, le temps de déploiement n'a pas beaucoup d'impact sur le makespan ($C_{max} = 6$ pour les deux cas). L'impact du temps de déploiement est minime sur le makespan par rapport au temps d'exécution des tâches, puisqu'il est très petit (de l'ordre de quelques minutes).*

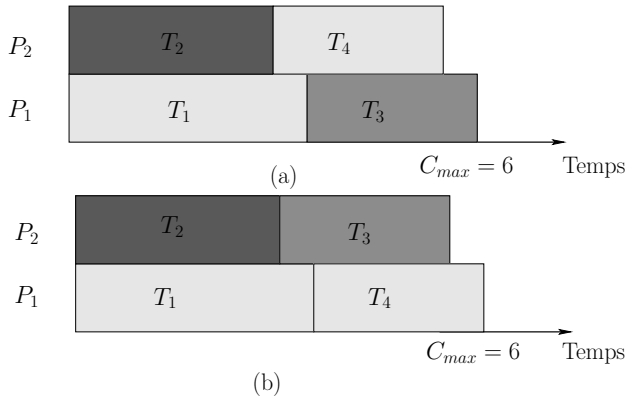

FIG. 2.4 – (a) Nombre de déploiements $D_{sum} = 4$, (b) Nombre de déploiements $D_{sum} = 3$

Minimiser le nombre de déploiements est en lui même un problème très facile. Il suffit pour cela d'exécuter les tâches nécessitant le même environnement successivement. Néanmoins, cela ne minimise guère le temps de terminaison de toutes les tâches. Comme on peut le constater sur la figure 2.5. On considère une instance composée de m processeurs et n tâches séquentielles, chacune des tâches nécessite le même temps d'exécution p et le même environnement. On peut voir que si l'on souhaite minimiser le makespan il suffit d'exécuter les tâches parallèlement sur les m processeurs ce qui nous amène à faire m déploiements. D'un autre côté si l'on minimise le nombre de déploiements, le plus simple est de déployer l'environnement sur un seul processeur et exécuter les n tâches successivement ce qui augmente le *makespan*.

Nous arrivons ainsi à la conclusion suivante : "Nos deux critères sont antagonistes".

Définition 2. *Suivant la notation standard en trois champs $\alpha|\beta|\gamma$ [LMB96], le problème bicritère d'ordonnancement des tâches multiprocesseurs avec déploiement d'environnements est noté $P|q_i, e_i, p_i|(D_{sum}, C_{max})$.*

En optimisation combinatoire, pour un ensemble fini de solutions réalisables trouver la recherche de la meilleure solution pour un critère revient à trouver efficacement cette solution dans cet ensemble. En multicritère, cela revient à trouver efficacement les meilleures solutions non dominées.

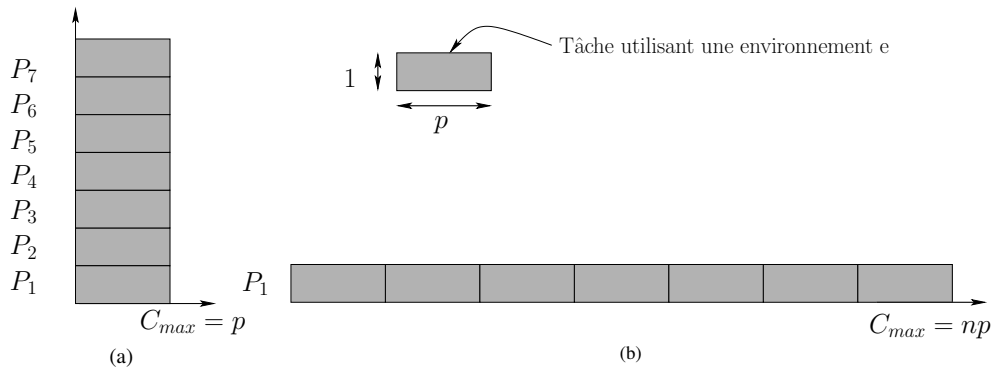

FIG. 2.5 – (a) Minimiser le makespan, (b) Minimiser le nombre de déploiements

2.5.1 Problèmes multicritères d'optimisation combinatoire

Dans le cadre de l'optimisation combinatoire classique, la qualité d'une solution est mesurée généralement à l'aide d'un seul critère. Cependant, une décision n'est jamais prise selon un critère unique. En pratique, plusieurs critères souvent conflictuels sont nécessaires pour apprécier la qualité d'une solution [Hoo92, VB01, CS02, EG00, Ehr01]

L'optimisation combinatoire multicritères tire sa particularité de la considération simultanée de plusieurs critères dans un processus d'optimisation. Ce degré supérieur de liberté permet de modéliser plus fidèlement un problème concret mais il n'est pas sans inconvénient.

La particularité de ces problèmes réside dans le fait que $k \geqslant 2$ critères ou k fonctions objectives sont considérées. On peut alors définir un problème d'optimisation multicritère comme suit :

1. Un ensemble fini de solution réalisable $\mathcal{S}$
2. Un vecteur $\vec{f} = (f_1, .., f_k)$ de k fonctions objectives où $f_i : \mathcal{S} \longmapsto \mathbb{Q}$ pour $i = 1, ..., k$.
3. Un vecteur $\vec{b} = (but_1, ..., but_k)$ de k buts où $but_i \in \{\min, \max\}$ détermine si la fonction f_i doit être minimisée ou maximisée.

Par conséquent, un problème d'optimisation combinatoire multicritères s'exprime : $(but_1 f_1(s), ..., but_k f_k(s))$ sous contrainte que $s \in \mathcal{S}$.

Afin de simplifier les notations, on définit $but_i = \min$ pour tout $i = 1, ..., k$.

L'*image* d'une solution $s \in \mathcal{S}$ est un vecteur $\vec{f}(s) \in \mathbb{Q}^k$. On note $\mathcal{E} = \{\vec{f}(s) | s \in \mathcal{S}\}$ l'image de $\mathcal{S}$ par le vecteur $\vec{f}$. Ainsi, $\mathcal{S}$ est appelé *espace de décision* tandis que $\mathcal{E}$ est appelé *espace objectif*.

Il est important de noter que la notion du multicritère d'un problème est liée au fait que les fonctions objectives sont en *conflit*, c'est-à-dire qu'une solution optimale pour un critère ne l'est pas forcément pour un autre, une notion d'antagonisme. Cette propriété essentielle fait que l'existence d'une solution optimale pour toutes les fonctions objectives est souvent écartée.

2.5.2 Optimalité de Pareto

Historiquement, le premier à évoquer les situations impliquant plusieurs critères en conflit est Vilfredo Pareto en 1896 [Par96]. Il n'est alors pas question d'optimalité mais d'*ophélimité*. Ce mot tire de la racine grecque ophelimos, et en terme de multicritères, une solution jouit d'une ophélimité maximale lorsque toute amélioration selon un critère engendre une détérioration selon au moins un autre critère. Pour Vilfredo Pareto, une solution est dite *Pareto optimale* si elle jouit d'une ophélimité maximale.
L'optimalité de Pareto est basée sur la notion de *dominance*. Formellement, une solution s domine une solution s' si elle est au moins aussi bonne sur tous les critères et strictement meilleure sur au moins un critère.

Définition 3. *Une solution s domine une solution s', noté $s < s'$, si les deux points suivants sont vérifiés :*
- $f_i(s) \leqslant f_i(s')$ *pour* $i = 1, ..., k$.
- *Il existe au moins un critère* i^* *tel que* $f_{i^*}(s) < f_{i^*}(s')$

Il existe des cas où la solution s ne domine pas s' et que s' ne domine pas s (voir figure 2.6), la dominance n'étant qu'un ordre partiel. Les solutions s et s' sont dites alors *incomparable*. La notion de dominance s'applique aussi bien aux solutions qu'aux valeurs qu'elles atteignent. Ainsi $\vec{f}(s)$ domine $\vec{f}(s')$, et on note $\vec{f}(s) < \vec{f}(s')$ si $s < s'$.

Définition 4. *Une solution $s \in \mathcal{S}$ est dite Pareto optimale s'il n'existe pas de solution $s' \in \mathcal{S}$ qui la domine. Si $s \in \mathcal{S}$ est pareto optimale alors son image $\vec{f}(s)$ est dite efficace.*

Définition 5. *Pour une instance d'un problème multicritères, l'ensemble de Pareto, noté $\mathcal{S}_{Pareto} \subseteq \mathcal{S}$, est l'ensemble des solutions Pareto optimales.*

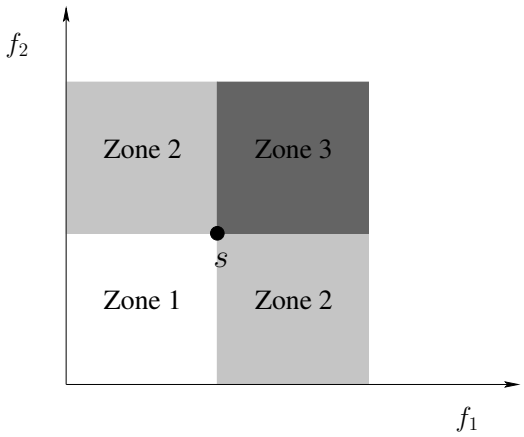

FIG. 2.6 – La solution s domine (resp. est dominée par) toutes solutions se trouvant dans la Zone 1 (resp. Zone 3). La solution s et toutes solutions se trouvant dans la Zone 2 sont incomparables.

Définition 6. *La courbe de Pareto d'une instance d'un problème multicritères noté $\mathcal{P}$, est l'ensemble des images des solutions Pareto optimales :*

$$\mathcal{P} = \{\vec{f}(s) | s \in \mathcal{S}_{Pareto}\}$$

Comme en optimisation monocritère, on fait l'hypothèse que si deux solutions distinctes ont la même image alors une seule suffit. On parle alors d'*ensemble minimal de Pareto*, à savoir un ensemble $\mathcal{S}_{ParetoMin} \subseteq \mathcal{S}_{Pareto}$ vérifiant :

$$\{\vec{f}(s) | s \in \mathcal{S}_{ParetoMin}\} = \{\vec{f}(s) | s \in \mathcal{S}_{Pareto}\}$$

Remarque 2. *La définition de la courbe de Pareto repose sur l'image des solutions i.e. espace objectif et non les solutions elles mêmes i.e. espace de décision. Néanmoins, lorsque le contexte est clair, il est courant de confondre les notions d'ensemble de Pareto et de courbe de Pareto. La figure 2.7 reprend les notions d'optimalité, d'ensemble et de courbe de Pareto pour un problème bicritère.*

Certains problèmes en optimisation combinatoire peuvent être résolus de manière efficace quand il s'agit d'optimiser un critère, mais deviennent $\mathcal{NP}$-difficile lorsque deux objectifs sont simultanément pris en compte. À titre d'exemple on peut citer le problème d'ordonnancement sur une machine où la somme pondérée des dates doit être minimisée. Ce problème noté $1||\sum w_i C_i$ a été résolu en temps polynômial. En effet, Smith [Smi56] a prouvé qu'il suffit de placer les tâches selon leurs rapports $\frac{w_i}{p_i}$

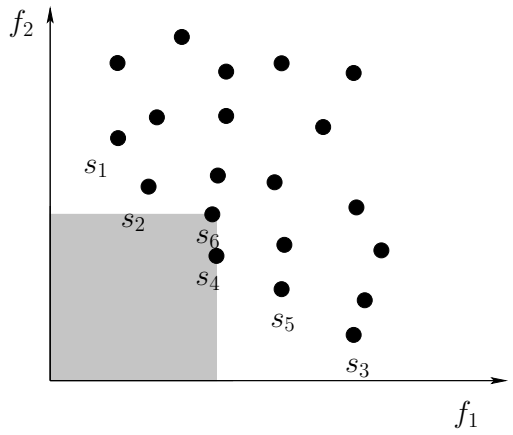

FIG. 2.7 – Les solution s_1 à s_5 sont Pareto optimales : $\mathcal{S}_{Pareto} = \{s_1, s_2, s_3, s_4, s_5\}$ et $\mathcal{P} = \{\vec{f}(s_1), \vec{f}(s_2)\vec{f}(s_3), \vec{f}(s_4), \vec{f}(s_5), \vec{f}(s_6)\}$

décroissants pour obtenir un ordonnancement optimal. Si l'on ajoute pour chaque tâche T_i un coût c_i ainsi qu'une fonction objective a minimiser appelée *coût total*, ce problème est alors noté $1||(\sum w_i C_i, \sum c_i C_i)$. Hoogeveen [Hoo92] a montré que ce problème est $\mathcal{NP}$-difficile à l'aide d'une réduction du problème PARTITION. Grouvès dans ces travaux a montré à l'aide d'une instance que l'ensemble des images des optima de Pareto pouvait être de taille exponentielle. En conclusion, la détermination de plusieurs optima est problématique pour deux raisons :

- Le nombre des solutions est souvent exponentiel.
- En déterminer ne serait-ce qu'une est souvent $\mathcal{NP}$-difficile.

C'est donc naturellement que l'approximation s'est imposée en optimisation combinatoire multicritère. Les méthodes dans le domaine de l'approximation peuvent être classifiée en trois catégories [Tal] :

- les approches basées sur la transformation du problème multicritère en un problème monocritère ;
- les approches non-Pareto où chaque critère est considéré séparément ;
- les approches Pareto où l'optimalité de Pareto est au cœur du processus d'optimisation.

Suivant ces trois approches, des algorithmes d'approximation avec ou sans garantie de performance ont été construits pour les problèmes d'optimisation souvent bicritère [Gou05, EG00, Tal]

2.5.3 La particularité du problème d'ordonnancement bicritère avec déploiement

Le problème d'ordonnancement bicritère avec déploiement est déjà $\mathcal{NP}$-difficile dans sa version monocritère 2.4.3. Notre première approche pour essayer de résoudre notre problème fut un algorithme de liste. Nous nous sommes vite rendus compte que cet algorithme donne une très bonne approximation par rapport au *makespan* mais le nombre de déploiements peut devenir alors très grand. La figure 2.8 illustre ces propos. Il s'agit d'un ensemble de tâches où chaque environnement est utilisé par deux tâches de taille différente. Une tâche large qui utilise m processeurs et un temps d'exécution égal à 1, et une tâche longue qui a un temps d'exécution p et nécessite qu'un seul processeur. Si on dispose de m tâches de largeur m et m tâches de longueur m alors on constate qu'avec un ordonnancement liste dont la politique est de minimiser le nombre de déploiements on obtient $D_{sum} = m \times m$ et $C_{max} = m(p+1)$. Avec un ordonnancement de liste dont la politique est de minimiser le *makespan*, on obtient $C_{max} = p + m$ avec un nombre de déploiements $D_{sum} = m \times m + m - 1$.

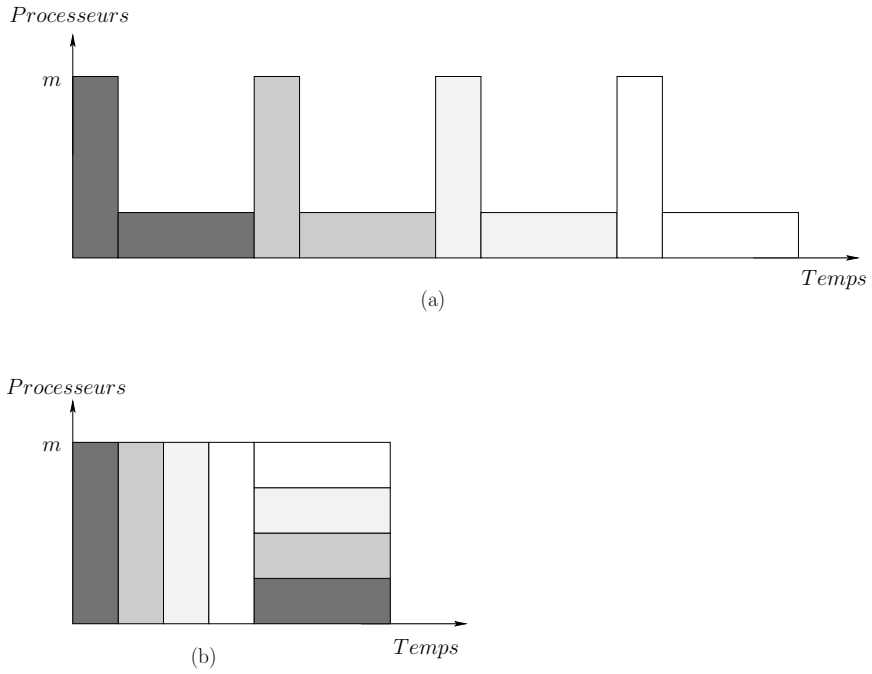

FIG. 2.8 – (a) Ordonnancement de liste qui minimise le nombre de déploiements, (b) Ordonnancement de liste qui minimise le *makespan*

Remarque 3. *Il n'existe pas d'algorithme garanti à un facteur constant par rapport aux optimums de chaque critère.*

2.5.4 Approche budget

L'approche budget consiste à optimiser un critère alors que les autres sont contraints à ne pas dépasser des bornes fixées. On peut penser au problème du sac à dos [GJ79] par exemple. Formellement, pour k critères où $B_2, ..., B_k$ fixés, on peut définir cette approche de la façon suivante :

$$\begin{aligned} \min_{s \in \mathcal{S}} \quad & f_1(s) \\ \text{sous contraintes :} & \\ f_2(s) \quad & \leqslant B_2 \\ \vdots & \\ f_k(s) \quad & \leqslant B_2 \end{aligned}$$

Dans le cadre de l'approximation avec garantie de performance, on peut chercher à retourner une solution s vérifiant toutes les contraintes et qui soit ρ-approchée sur f_1 :

$$f_1(s) \leqslant \rho\Big(\min_{s' \in \mathcal{S}} \{ f_1(s') | f_i(s') \leqslant B_i, i = 2, ..., k \} \Big)$$

Ainsi, seul un scalaire ρ est nécessaire pour qualifier la finesse de l'approximation. Cependant, une définition plus générale de l'approximation peut être donnée en supposant que les contraintes sont relaxées et que les bornes peuvent être dépassées.

Définition 7. *Pour $(B_2, ..., B_k)$ fixé et $B_1 = \min_{s \in \mathcal{S}} \{ f_1(s) | f_i(s) \leqslant B_i$ pour $i = 2, ..., k \}$, une solution $s \in \mathcal{S}$ est dite $(\rho_1, ..., \rho_k)$-budget-approchée si $f_i(s) \leqslant \rho_i B_i$ pour $i = 1, ..., k$*

On en déduit qu'une solution ρ-approchée est une solution $(\rho, 1, ..., 1)$-budget approchée.

Pour un problème d'optimisation bicritère, cette approche est intéressante lorsqu'il est facile de trouver un optimum pour un critère. Comme nous l'avons préciser plus haut le caractère $\mathcal{NP}$-difficile de notre problème dans sa version monocritère, nous allons définir dans une nouvelle approche pour aborder notre problème d'ordonnancement aborder notre problème d'ordonnancement bicritère avec déploiement. Néanmoins, cette approche est basée sur l'approche budget avec relâchement de contrainte. En effet, il ne s'agit plus de fixer un critère à l'optimum et définir les solutions

qui respectent les bornes sur les autres critères. Il s'agit de fixer une valeur pour un critère qui représente une solution possible (réalisable), ce qui nous permet de définir la borne inférieure pour le second critère et de déterminer ainsi les solutions (ρ_1, ρ_2)-budget-approchées-relaxées.

2.5.5 Approche budget relaxée pour le problème d'ordonnancement bicritère avec déploiement

L'approche budget relaxée pour le problème d'ordonnancement bicritère avec déploiement consiste à supposer que s'il existe un ordonnancement de longueur fixée à une valeur réalisable (plus grand que l'optimal supposé) on peut alors définir la plus petite valeur possible pour le nombre de déploiements.

Définition 8. *Pour un λ fixé et $D_\lambda \leqslant \min_{s\in\mathcal{S}}\{D_{sum}(s)|C_{max}(s) \leqslant \lambda\}$, une solution $s \in \mathcal{S}$ est dite (ρ_1, ρ_2)-budget-relaxée-approchée si $C_{max}(s) \leqslant \rho_1\lambda$ et $D_{sum}(s) \leqslant \rho_2 D_\lambda$*

Comme nous l'avons vu dans la section 2.5.3, nos deux critères sont antagonistes et les bornes très éloignées. L'approche budget relaxée pour ce problème consiste à fixer un budget que l'on note λ pour le premier critère i.e. le *makespan* et déterminer l'ensemble des solutions tel que la borne inférieure que l'on note D_λ pour le deuxième critère vérifie la contrainte $D_\lambda \leqslant \rho_2 \min_{s\in\mathcal{S}}\{D_{sum}|C_{max} \leqslant \rho_1\lambda\}$.

D'une certaine façon, il s'agit de trouver un "bon compromis" entre les deux critères. Ainsi, si nous fixons une valeur correcte pour un critère, alors on sait définir la borne inférieur pour le second.

La figure 2.9 illustre une façon de trouver une solution par rapport à l'exemple de la figure 2.4 qui ne soit pas trop "mauvaise" pour les deux critères.

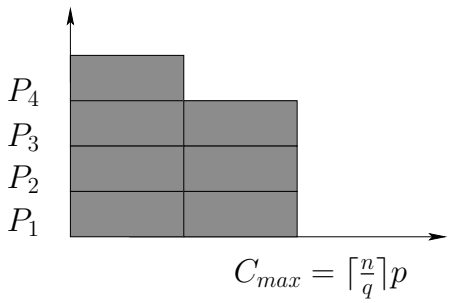

FIG. 2.9 – But : Le compromis entre le makespan et le nombre de déploiement

Pour un ordonnancement de longueur λ, on détermine la borne inférieure que l'on note D_λ du nombre de déploiements nécessaire pendant ce temps qui a les propriétés suivantes :

– Pour chaque environnement e il est nécessaire de faire q^e_{max} déploiements tel que q^e_{max} est le nombre de processeurs utilisés par sa plus grande tâche.
– Pour toutes les tâches utilisant le même environnement il est nécessaire de déployer l'environnement sur $\left\lceil \sum_{T_i \in e} \frac{w^e_i}{\lambda} \right\rceil$ processeurs pour un ordonnancement de longueur λ.

Formellement,

$$D_\lambda = \sum_{e \in E} \max \left(q^e_{max}, \left\lceil \sum_{T_i \in e} \frac{W^e_i}{\lambda} \right\rceil \right)$$

On peut remarquer dans la figure 2.10 une solution optimale pour l'approche budget relaxée n'est généralement pas Pareto optimale.

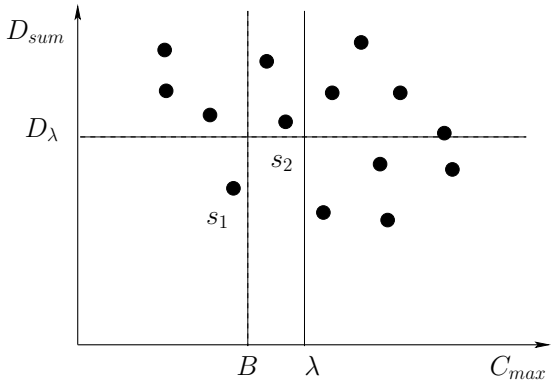

FIG. 2.10 – La solution s_1 minimise la fonction nombre de déploiements D_{sum} tout en respectant la contrainte $C_{max} \leqslant B$ et s_1 est Pareto optimale. La solution s_2 est incluse dans l'ensemble de solutions avec D_λ borne inférieure tout en respectant la contrainte $C_{max} \leqslant \lambda$

2.6 Conclusion

La considération de plusieurs objectifs dans les problèmes d'ordonnancement permet une certaine liberté. En intégrant la notion de compromis entre plusieurs objectifs conflictuels, les problèmes multicritères permettent de modéliser plus fidèlement des cas pratiques en optimisation. Cependant, ce degré supérieur de liberté implique une complexité accrue et bon nombre de problèmes polynômiaux en optimisation classique deviennent $\mathcal{NP}$-difficiles lorsque plusieurs critères sont pris en compte. Ce

chapitre retrace les principaux concepts et notions que l'on trouve dans les problèmes d'ordonnancement. Un tour d'horizon sur les tâches rigides et les algorithmes d'approximation sont présentés. Nous avons fait une introduction sur les problèmes multicritères, la notion d'optimalité et d'approximation pour ce genre de problème.

Nous nous sommes intéressés au problème d'ordonnancement bicritère avec déploiement $P|p_j, e_j, q_j|(D_{sum}, C_{max})$ dont la particularité est la non approximabilité simultanée (non approximable sur les deux critères à un facteur constant). Nous avons défini l'approche budget relaxée qui nous a permis de définir plus finement le compromis entre les deux critères étudiés pour ce problème. Reste à définir un algorithme qui respecte ce compromis.

Chapitre 3

Le problème d'ordonnancement bicritère avec déploiement

Dans ce chapitre nous traitons le problème d'ordonnancement bicritère avec déploiement. Nous définissons un algorithme basé sur l'approche budget relaxée qui permet de déterminer ainsi un ordonnancement d'au plus 4λ pour le *makespan* et de $2D_\lambda$ pour le nombre de déploiements. Les résultats présentés dans ce sont issus de l'article [MM07].

3.1 Ordonnancement multicritère

La théorie de l'ordonnancement est apparue dans les années 1950. Dès lors, des problèmes de plus en plus complexes ont été abordés notamment des problèmes issus des applications industrielles. La littérature montre que dans la majorité des cas traités, les problèmes d'ordonnancement nécessitent souvent la prise en compte de plusieurs critères. Pourtant, ils ont fait l'objet de nombreuses études lorsqu'il s'agit d'optimiser un critère unique, mais beaucoup moins, lorsqu'il s'agit de plusieurs critères. Plus généralement, les premiers travaux traitant de problèmes d'optimisation multicritère remontent au début des années 1970. La littérature dans ce domaine est très conséquente. La plupart des résultats sur l'ordonnancement multicritère sont basés sur les travaux en optimisation multicritère [EBK01, CJK98, TGC02]. Le lecteur intéressé peut lire par exemple le livre de T'kinds et Billaut [VB01], ou celui de Collette et Siarry [CS02]. On trouve aussi dans les travaux de Steuer [Ste86] et de Goicoechea [GHD82] des définitions ainsi que des résumés complets sur le domaine.

Dans les travaux de [DS88, FAH89, Hoo92, NSD93, CW97] les différents auteurs, définissent quatre types d'approches pour les problèmes d'ordonnancement multicritère. La première approche fixe un critère (à l'optimal) et tente d'optimiser le second. La deuxième considère la combinaison convexe des critères. La troisième approche consiste en la détermination de tous les optimums stricts de Pareto. La quatrième étudie la qualité de l'approximation qui pourrait être obtenue simultanément pour les divers critères. On trouve des résultats similaires lorsqu'il s'agit de plusieurs machines [NHH95].

Les premières études concernant les problèmes d'ordonnancement multicritère sont relatives à une seule machine. Smith [Smi56] a été le premier, en 1956 à étudier le problème d'ordonnancement bicritère sur une machine. Il a considéré le problème de minimisation de la somme des temps de terminaison des tâches, et le maximum des retards. Il a prouvé qu'il suffit d'ordonnancer les tâches selon leurs temps d'exécution croissants pour un ordonnancement optimal. Ce procédé est connue sous le nom de «règle de Smith». Van Wassenhove et Gelders [WG80] utilisent cette règle pour résoudre le problème d'ordonnancement bicritère où il s'agit de minimiser la somme des temps d'exécution et le retard (tardiness). Ils définissent ainsi la courbe de Pareto pour ce problème. Dans l'article [Bag89], Bagchi étudie plusieurs problèmes bicritère d'ordonnancement et propose un algorithme paramétrique pour déterminer un ensemble de solutions Pareto optimales. Hoogeveen mentionne dans sa thèse [Hoo92] que la méthode utilisée par Bagchi peut être appliquée au problème d'ordonnancement où il s'agit de minimiser le coût total et le poids total. Pour ce même problème, [EBG05] détermine des solutions supportées par l'agrégation des critères.

Shmoys et Tardos [ST93] définissent un algorithme pour optimiser le temps de terminaison des tâches, *makespan*, et la moyenne des temps d'exécution sur des machines indépendantes. Leur approche consiste à fixer un critère, le *makespan*, et à minimiser la moyenne des temps. Pour un problème similaire, Chakrabarti *et al.* [Cha96] (voir aussi [HSSJ97], [HSJ96]) avec la même approche définissent des algorithmes qui optimisent simultanément le *makespan* et la somme pondérée des temps de terminaison. Leur approche est un algorithme qui prend en entrée un ensemble de tâches pondérées et un temps D, et essaye de minimiser le temps nécessaire pour faire autant de poids que l'optimal (pour le poids) en temps D. L'approche ρ-approximation duale ([PCTW97]) donne lieu à un algorithme qui produit une solution au plus 2.89ρ pour le *makespan* et au plus 4ρ pour la moyenne des poids. Pour le même problème Stein et Wein [CW97], améliorent ce résultat pour $\rho = 1$ et fournissent une solution $(2, 2)$-approchée pour les deux critères. Hoogeveen dans

[Hoo05] fait un bilan complet de tous les problèmes d'ordonnancement bicritère.

3.1.1 Complexité des problèmes d'ordonnancement multicritère

Chen et Bulfin dans [CB93, CB94] présentent un résumé très complet sur la complexité des problèmes d'ordonnancement multicritère sur une et plusieurs machines. Suivant les différentes approches présentées dans la section précédente, ils classifient les problèmes d'ordonnancement et présentent ainsi leur complexité. Nous nous somme inspiré de leurs résultats pour énoncer le théorème qui suit :

Théorème 3.1.1. *Si $P||C_{max}$ est $\mathcal{NP}$-difficile alors $P|p_j, e_j, q_j|(C_{max}, D_{sum})$ l'est aussi.*

Démonstration. S'il existe un algorithme polynômial pour résoudre le problème $P|p_j, e_j, q_j|(C_{max}, D_{sum})$ alors ce même algorithme polynômial permet de résoudre le problème $P||C_{max}$.

□

3.2 Rappel et définition du modèle

Bien qu'ayant déjà été introduit dans le chapitre 2, le problème d'ordonnancement bicritère étudié est rappelé ici afin de fixer les notations utilisées. Le problème d'ordonnancement bicritère prend en entrée un ensemble de tâches indépendantes et rigides $\Gamma = \{T_1, \cdots, T_n\}$, un ensemble de processeurs $\Pi = \{\pi_1, \cdots, \pi_m\}$ et un ensemble $E = \{1, \cdots, s\}$ de S environnements. Chaque tâche T_i a besoin pour s'exécuter d'une quantité de processeurs $q(T_i)$ telle que $q : \Gamma \longrightarrow \mathbb{N}$ $(q(T_i) \leqslant m)$ et d'un environnement de travail $e(T_i) = j \in E$ où $e : \Gamma \longrightarrow E$. La quantité de travail d'une tâche T_i est définie par $w(T_i) = q(T_i)p(T_i)$ où $p(T_i)$ est la durée d'exécution de T_i ($p : \Gamma \longrightarrow \mathbb{N}$ et $w : \Gamma \longrightarrow \mathbb{N}$). Soit σ un ordonnancement réalisable. Alors, σ est un ordonnancement où, à un instant donné, chaque tâche $T_i \in \Gamma$ est exécutée par $q(T_i)$ processeurs. Un processeur exécute au plus une tâche et celle-ci commence son exécution seulement si son environnement a été préalablement déployé sur les processeurs dédiés.

On définit le temps de terminaison d'un ordonnancement σ par :

$$C_{max}(\sigma) = \max_{T_i \in \Gamma} C^{\sigma}(T_i), \tag{3.1}$$

où $C^{\sigma}(T_i)$ est le temps de terminaison de la tâche T_i pour l'ordonnancement σ.

Soit la fonction *déploiement* définie par $D : \Pi \mapsto \mathbb{N}$, qui détermine pour chaque processeur $\pi_s \in \Pi$ le nombre de déploiements qui sont effectués. Ainsi, le nombre de déploiements pour un ordonnancement σ est donné par :

$$D_{sum}(\sigma) = \sum_{s=1}^{m} (D(\pi_s))^{\sigma}. \tag{3.2}$$

Le problème d'ordonnancement bicritère consiste à déterminer un ordonnancement avec le plus petit C_{max}, et un minimum de déploiement d'environnements. C'est-à-dire déterminer le plus petit D_{sum}.

3.3 Les Différents Compromis

Le caractère $\mathcal{NP}$-difficile du problème d'ordonnancement bicritère avec déploiement ne nous permet pas de trouver, en un temps polynômial (si $\mathcal{P} \neq \mathcal{NP}$) et pour une longueur (la plus petite possible) un ordonnancement qui minimise le nombre de déploiements. Comme nous l'avons vu dans la section 2.5.3, utiliser un algorithme qui donne la meilleure garantie de performance sur le *makespan*, pour un problème d'ordonnancement de tâches rigides avec déploiement, ne permet pas de minimiser le nombre de déploiements.

Notre approche pour ce problème consiste à fixer une valeur réalisable pour le *makespan* et de définir un arrangement de tâches permettant de minimiser le nombre de déploiements. Nous nous sommes inspirés de l'une des approches utilisée pour traiter les problèmes multicritères. C'est l'approche dite «*approche budget*», introduite dans [Gou05], qui consiste à optimiser un critère en se fixant des bornes pour les autres critères.

Par conséquent, comme nous l'avons vu dans la section 2.5, nos deux critères sont antagonistes. En effet, trouver simplement un ordonnancement de longueur λ sans que cet ordonnancement puisse nous fournir le plus petit nombre de déploiement est un problème $\mathcal{NP}$-difficile. Nous avons donc décidé de relâcher les contraintes d'optimalité.

Ainsi, s'il existe un ordonnancement de longueur λ telle que celle-ci soit une valeur pour un ordonnancement réalisable, alors le plus petit nombre de déploiements possible pour exécuter toutes les tâches est déterminé par D_λ.

3.4 Approche budget relaxée

Étant donnée une valeur λ sur le makespan, l'approche budget relaxée consiste à déterminer un ordonnancement dont le nombre de déploiement est bornée par D_λ.

Propriété 1. *Soit λ une valeur pour un ordonnancement réalisable. Alors ,*

$$\forall T_i \in \Gamma, \quad \lambda \geqslant \max\left(\max_{T_i \in \Gamma} \quad p(T_i), \sum_{T_i \in \Gamma} \frac{w(T_i)}{m}\right). \tag{3.3}$$

Propriété 2. *Soit D_λ le nombre de déploiements pour un ordonnancement de longueur λ. Alors,*

$$D_\lambda = \sum_{j \in E} \max\left(\max_{T_i \in \Gamma_{E_j}} q(T_i), \sum_{T_i \in \Gamma_{E_j}} \left\lceil \frac{w(T_i)}{\lambda} \right\rceil\right). \tag{3.4}$$

où Γ_{E_j} est un sous ensemble de tâches tel que $\Gamma_{E_j} \subseteq \Gamma$ et où les tâches ont un environnement j. $w(T_i)^j$ étant la quantité de travail d'une tâche T_i de l'environnement $j \in E$.

La définition d'une solution (α, β)-budget-relaxée-approchée dans notre contexte est rappelée dans la définition suivante et illustrée dans la figure 3.6.

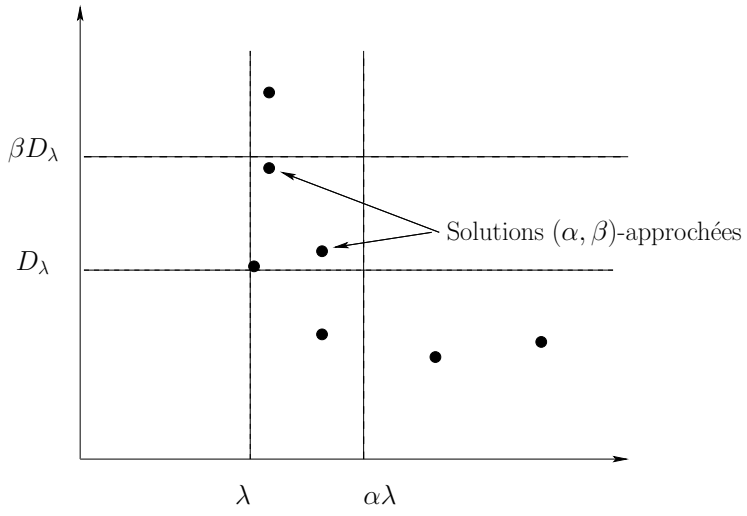

FIG. 3.1 – Illustration d'une solution (α, β)-approchée pour l'approche budget relaxée

Définition 9. *Soient λ un budget pour le makespan et D_λ une borne inférieur du nombre de déploiement pour ce makespan. On appelle solution (α, β)-budget-relaxée-approchée, tout ordonnancement dont le makespan est au plus $\alpha\lambda$, et dont le nombre de déploiements est au plus βD_λ.*

Nous nous sommes inspirés de cette approche pour définir l'algorithme **GLS** qui construit un ordonnancement $(4, 2)$-budget-relaxé-approché. Ainsi, on définit le problème d'ordonnancement qui minimise le nombre de déploiements pour un *makespan* fixé.

3.5 Algorithme GLS

L'algorithme que nous allons définir est construit en deux phases : une phase de construction de groupes de tâches et une phase d'ordonnancement, décrites comme suit :

1. Pour la première phase, deux classes de sous-ensembles de tâches sont définies. La classe des groupes feuilletés $\mathcal{G}_f$ et la classe des groupes canoniques $\mathcal{G}_C$. Les tâches appartenant à un même groupe ont le même environnement. Ainsi, pour chaque environnement, sont définis un ensemble de groupes feuilletés et un groupe canonique.
2. La deuxième phase est l'ordonnancement des tâches. Cet ordonnancement est basé sur l'ordonnancement de liste. Chaque groupe de tâches construit représente alors une MétaTâche telle que le nombre de processeurs et le temps d'exécution soient suffisant pour accomplir toutes les tâches du groupe.

3.5.1 Première phase

La première partition des tâches se fait suivant leur environnement. Toutes les tâches appartenant à un environnement sont regroupées dans des sous-ensembles de tâches $\Gamma_{E_j} \subset \Gamma$ telle que $\Gamma_{E_j} = \{T_i / e(T_i) = j\}$. Les tâches sont ensuite triées suivant un ordre décroissant de leur nombre de processeurs. Par conséquent, $\Gamma_{E_j} = \{T_1, T_2, \cdots, T_r\}$ tel que $q(T_1) \geqslant q(T_2) \geqslant \cdots \geqslant q(T_r)$. De cet ensemble de tâches ainsi défini, on construit les différents groupes feuilletés et canoniques pour chaque environnement.

Groupe Feuilleté

Les tâches appartenant à un groupe feuilleté sont ordonnancées d'une façon séquentielle suivant l'ordre décroissant du nombre de processeurs. Chaque tâche du groupe utilise plus de la moitié du nombre de processeurs utilisés par la première tâche du groupe (voir contrainte 3.5 ci dessous). Formellement, si on définit par G_{f}^{j} un groupe feuilleté de l'environnement $j \in E$ alors le nombre de processeurs utilisés par le groupe correspond au nombre de processeurs nécessaires pour l'exécution de la première tâche du groupe (c'est-à-dire, celle qui a le plus grand nombre de processeurs) (voir contrainte3.6). Le temps d'exécution du groupe est définit par la somme des temps d'exécution de toutes les tâches du groupe.

$$q(T_k) \geqslant q(T_{k+1}) \geqslant \cdots \geqslant q(T_l) > \frac{1}{2} q(T_k), \tag{3.5}$$

$$G_{\mathrm{f}}^{j} = \{T_k, T_{k+1}, \cdots T_l\}, \quad q(G_{\mathrm{f}}^{j}) = q(T_k), \tag{3.6}$$

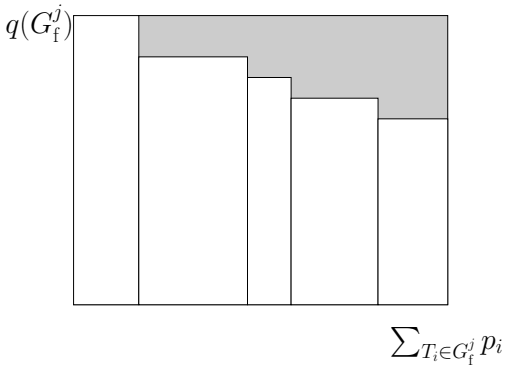

FIG. 3.2 – Groupe Feuilleté

Le principe est de regrouper des tâches qui utilisent à peu près la même quantité de processeurs en un seul bloc. La construction de chaque groupe feuilleté est caractérisée par la sélection de la première tâche. C'est celle qui définit la quantité de processeurs utilisés par le groupe en cours de construction.

La procédure *InitFeuilleté* permet de démarrer le processus de construction d'un groupe feuilleté. Elle définit à chaque appel la première tâche du groupe. On distingue deux catégories d'applications pour cette procédure. La première catégorie d'applications consiste à initialiser, pour un environnement donné, l'étape de construction des groupes feuilletés. Cela revient à définir la première tâche du premier

groupe feuilleté. Dans la seconde catégorie d'applications, l'appel de cette procédure se fait à chaque début de construction d'un nouvel groupe feuilleté pour le même environnement.

Pour la première catégorie d'application, son schéma est le suivant :

Procédure `InitFeuilleté`(Γ_{E_j})

Données : Ensemble de tâches Γ_{E_j}
Résultat : Le premier groupe feuilleté de l'environnement E_j
début
  Soit T_1 une tâche de Γ_{E_j} qui a le plus grand nombre de processeurs
  $DebutG \longleftarrow T_1$
  $FinG \longleftarrow T_1$
  $G_{\mathrm{f}}^{j} = [DebutG, FinG]$
  $q(G_{\mathrm{f}^j} = q(T_1)$
  $p(G_{\mathrm{f}^j} = p(T_1)$
  $\Gamma_{E_j} = \Gamma_{E_j} \backslash \{T_1\}$
fin

Pour la seconde catégorie d'application, le schéma est le suivant :

Procédure `InitFeuilleté`(Γ_{E_j})

Données : Ensemble de tâche Γ_{E_j}
Résultat : Un groupe feuilleté
début
  Soit $FinG$ la dernière tâche du groupe feuilleté précédent
  $DebutG \longleftarrow FinG + 1$
  $FinG \longleftarrow DebutG$
  $G_{\mathrm{f}}^{j} = [DebutG, FinG]$
  $p(G_{\mathrm{f}}^{j}) = p(DebutG)$
  $q(G_{\mathrm{f}}^{j}) = q(DebutG)$
  $\Gamma_{E_j} = \Gamma_{E_j} \backslash \{FinG + 1\}$
fin

Une fois la première tâche définie, l'étape suivante consiste à déterminer le reste des tâches constituant le groupe feuilleté en cours de construction. La condition qui permet d'ajouter une nouvelle tâche au groupe est spécifique à la tâche sélectionnée, c'est-à-dire,

– Une tâche est ajoutée au groupe feuilleté si son nombre de processeurs est

supérieur à la moitié du nombre de processeurs du groupe.

$$q(T_i) > \frac{1}{2} q(G_{\mathrm{f}}^j). \tag{3.7}$$

Soit T_i une tâche candidate pour être ajoutée au groupe feuilleté G_{f}^j en cours de construction. Si T_i ne vérifie pas la condition sur les processeurs 3.7, alors le groupe $G\mathrm{f}^j = [DebutG, FinG]$ est terminé et la tâche $T_i = FinG + 1$ est sélectionnée pour être la première tâche du prochain groupe feuilleté $G_{\mathrm{f}+1}^j$ (voir figure 3.3).

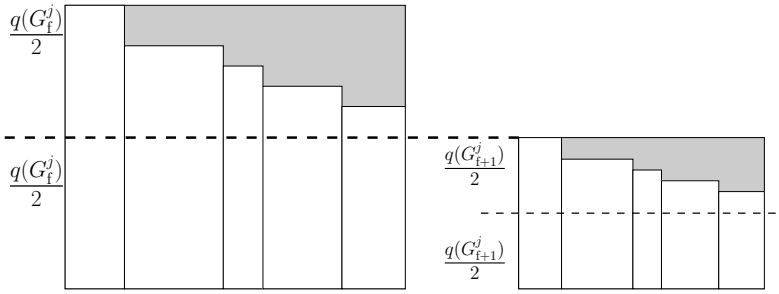

FIG. 3.3 – Condition d'arrêt pour un groupe feuilleté

La procédure *AjoutTache* permet d'ajouter une tâche au groupe feuilleté en cours de construction. Elle met ainsi à jour le temps d'exécution du groupe engendré. Le schéma de la procédure est le suivant :

Procédure `AjoutTâche`(Γ_{E_j}, G_f^j)

Données : Ensemble de tâches Γ_{E_j} et un groupe feuilleté G_{f}^j en cours de construction
Résultat : Un groupe feuilleté
début
  Soit $FinG$ la dernière tâche du groupe feuilleté
  $FinG \longleftarrow FinG + 1$
  $p(G_{\mathrm{f}}^j) = p(G_{\mathrm{f}}^j) + p(FinG)$
  $\Gamma_{E_j} = \Gamma_{E_j} \backslash \{FinG\}$
fin

Groupe canonique

Un groupe canonique est un groupe feuilleté "dégénéré". Lorsque l'algorithme construit les différents groupes feuilletés, ces derniers sont soumis à un test lors de

leur construction, par rapport à leur temps d'exécution. Comme nous l'avons précisé dans la section 3.5.1, le principe des groupes feuilletés est de regrouper des tâches qui utilisent la même quantité de processeurs dans un bloc. Ce bloc est donc limité par rapport au temps d'exécution. Si le groupe de tâches feuilletés est très grand alors il est transformé en un groupe canonique. Il faut noter aussi que toutes les tâches restantes du même environnement sont ajoutées au groupe canonique. Ces tâches sont ordonnées suivant un ordre décroissant du nombre de processeurs. Ainsi, les tâches qui n'appartiennent à aucun groupe ont forcément un nombre de processeurs inférieur ou égal aux tâches du groupe canonique.

Ainsi, on définit le nombre de processeurs $q(G_c^j)$ comme étant le plus grand entre le plus petit nombre de processeurs nécessaires pour exécuter une quantité de travail de tâches en un temps λ, et le nombre de processeurs de la plus grande tâche (voir contrainte 3.8). Les tâches du groupe canonique sont ordonnancées sur $q(G_c^j)$ processeurs par un algorithme de liste tel que $T_i \in G_c^j$ où,

$$q(G_c^j) = \max\left(\left\lceil \sum_{T_i \in G_c^j} \frac{w(T_i)}{\lambda} \right\rceil, \max_{T_i \in G_c^j} q(T_i)\right). \tag{3.8}$$

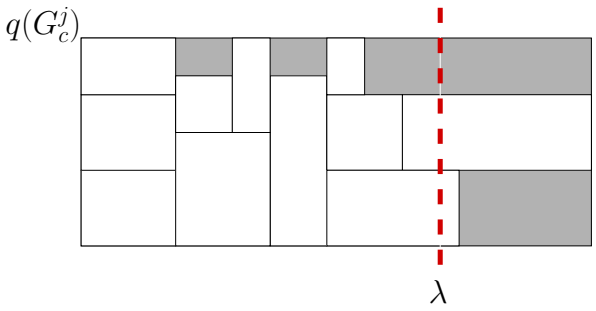

FIG. 3.4 – Groupe Canonique

Si une tâche T_i vérifie la condition 3.7 sur le nombre de processeurs, mais ne vérifie pas la condition sur le temps d'exécution, c'est-à-dire que $p(T_i) + p(G_f^j) > 2\lambda$, alors le groupe G_f^j et la tâche T_i sont transformés en groupe canonique (voir figure 3.5).

La procédure *GroupeCanonique* permet de définir l'ensemble de tâches qui constituent le groupe canonique pour un environnement donné. Elle calcule ainsi le

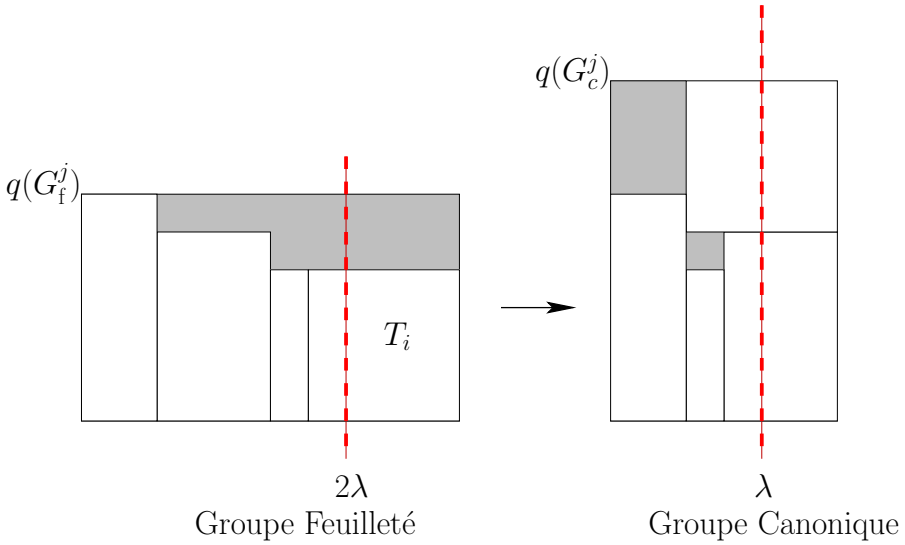

FIG. 3.5 – Transformation d'un groupe feuilleté en un groupe canonique dans le cas où la condition sur le temps d'exécution n'est pas vérifiée

nombre initial de processeurs $q(G_c^j)$, qui représente le nombre de processeurs nécessaire pour exécuter une quantité de travail relative aux tâches sélectionnées, pour qu'elles s'exécutent en un temps λ.

Procédure `GroupeCanonique`(G_c^j)

Données : Un ensemble de tâches
Résultat : Le nombre de processeurs pour le groupe canonique
début
  $G_c^j \longleftarrow [DebutG, T_r]$
  $q(G_c^j) = \max(\lceil \sum_{T_i \in G_c^j} \frac{w(T_i)}{\lambda} \rceil, q(T_i))$ où $T_i \in G_c^j$
  List Scheduling des tâches de G_k^c en utilisant $q(G_c^j)$ processeurs. Ceci permet de calculer $p(G_c^j)$
fin

Remarque 4. *Une fois que la procédure de construction du groupe canonique commence, toutes les tâches restantes de* Γ_{E_j} *feront alors partie du groupe canonique.*

Le groupe canonique a toujours un nombre de processeurs plus petit que tous les groupes feuilletés pour un environnement donnée.

Une fois $q(G_c^j)$ fixé, il est alors comparé aux nombres de processeurs de tous les groupes feuilletés construits.

Soit G_{f}^j le dernier groupe feuilleté, c'est-à-dire celui qui a le plus petit nombre de processeurs. Alors, il est comparé en premier à G_c^j. Si le nombre de processeurs de G_{f}^e est inférieur à celui de G_c^j, alors les tâches de G_{f}^e sont insérées dans le groupe canonique. On met alors à jour $q(G_c^j)$ afin de reconsidérer sa comparaison aux autres groupes feuilletés de l'environnement e. On a alors,

$$q(G_{\mathrm{f}}^j) \geqslant q(G_c^j). \tag{3.9}$$

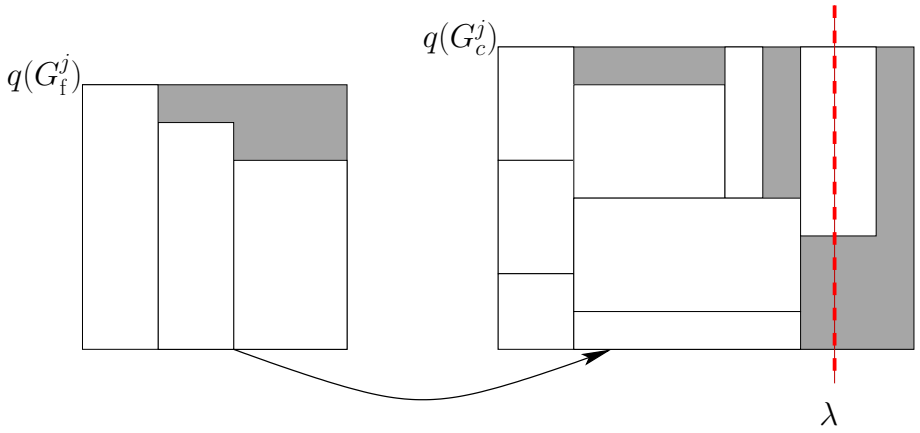

FIG. 3.6 – Cas de fusion : le nombre de processeurs du plus petit groupe feuilleté est inférieur au nombre de processeurs du groupe canonique.

Lemme 3.5.1. *Le temps d'exécution des tâches d'un groupe canonique est au plus égale à* 2λ

Démonstration. Comme nous l'avons vu dans la section 2.4.4, dans le cadre de l'ordonnancement des tâches rigides sur m machines homogènes, l'algorithme de liste a une garantie de performance de 2 sur le *makespan* [Gra69]. Le temps minimum pour exécuter toutes les tâches rigides de G_c^j est λ. En utilisant un algorithme de liste pour ordonnancer ces tâches sur $q(G_c^j)$, on obtient un temps d'exécution pour le groupe canonique d'au plus 2λ. □

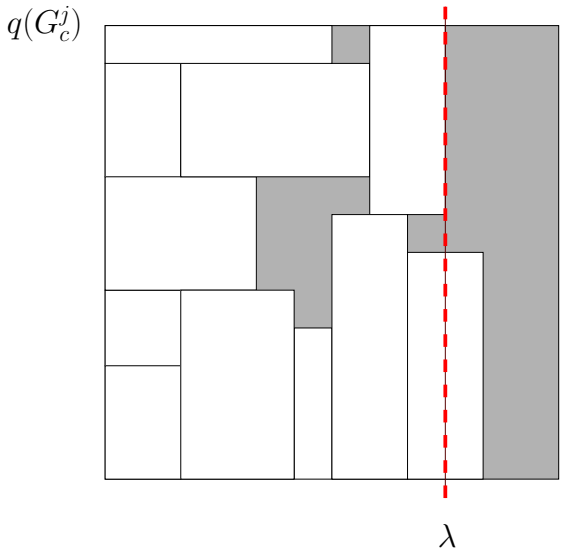

FIG. 3.7 – Groupe canonique après sa fusion avec le groupe feuilleté (suite de la figure 3.6)

3.5.2 Deuxième phase

Les propriétés des groupes et ordonnancement

Tous les groupes appartenant aux classes $\mathcal{G}_{\mathrm{f}}$ et $\mathcal{G}_c$ possèdent les propriétés suivantes :

Propriété 3. *La quantité totale de travail des tâches dans un groupe est supérieure à la moitié de la surface du groupe. Ce qui s'écrit :*

$$\sum_{T_i \in G_k^j} w(T_i) \geqslant \frac{p(G_k^j) q(G_k^j)}{2}. \tag{3.10}$$

Propriété 4. *Le temps d'exécution d'un groupe c'est-à-dire le temps d'exécution de toutes les tâches d'un groupe est au plus égale à 2λ, soit :*

$$p(G_k^j) \leqslant 2\lambda. \tag{3.11}$$

Propriété 5. *Le nombre de processeurs de chaque groupe est inférieur au nombre total de processeurs m, soit :*

$$q(G_k^j) \leqslant m. \tag{3.12}$$

L'équation 3.10 montre que les groupes ainsi construits n'induisent pas un temps de non activité trop important des processeurs .

En résumé, les tâches d'un groupe feuilleté sont ordonnancées de façon séquentielle, et celles du groupe canonique suivent un ordonnancement de liste. Les groupes ainsi construits ont un temps d'exécution, valant au plus 2λ, et un nombre de processeurs au plus égale à m.

Par conséquent, on considère chaque groupe de tâches comme étant une «MétaTâche» avec, un temps d'exécution, un nombre de processeurs et un environnement.

Soit $\mathcal{G}_{\mathcal{F}}$ la classe de toutes les MétaTâches feuilletées et $\mathcal{G}_{\mathcal{C}}$ la classe de toutes les MétaTâches canoniques. La deuxième phase de l'algorithme consiste à ordonnancer toutes les MétaTâches de $\mathcal{G}_{\mathcal{F}}$ et $\mathcal{G}_{\mathcal{C}}$ en utilisant un algorithme de liste sur m processeurs.

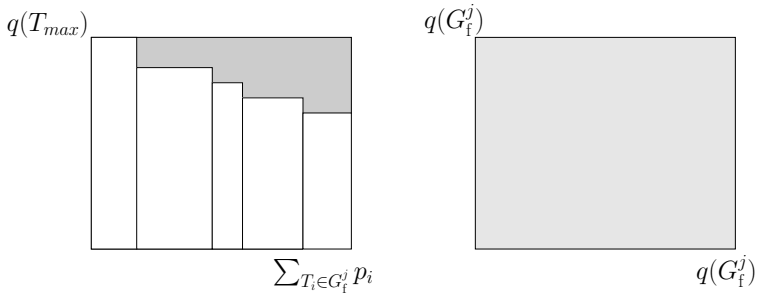

FIG. 3.8 – Un groupe Feuilleté transformé en MétaTâche

Rapport d'approximation pour le makespan

Théorème 3.5.2. *L'algorithme **GLS** construit un ordonnancement avec un makespan au plus égal à* 4λ.

Démonstration. Nous nous sommes inspirés de la preuve de Graham [Gra69], vue dans la section 2.4.4 pour montrer le rapport d'approximation sur le *makespan*. Pour pouvoir utiliser les résultats de Graham pour la construction de l'algorithme, on utilise les propriétés suivantes par rapport à l'optimal :

$$C^*_{max} \geqslant \max_{T_i \in \Gamma} p(T_i), \tag{3.13}$$

et

$$C^*_{max} \geqslant \sum_{T_i \in \Gamma} \frac{w(T_i)}{m}. \tag{3.14}$$

Algorithme 5 : GLS Algorithm

Données : Ensemble de tâches $\Gamma = \{T_1 = (e(T_1), p(T_1), q(T_1)),$
$T_2 = (e(T_2), p(T_2), q(T_2)), ..., T_n = (e(T_n), p(T_n), q(T_n))\}$
E_j Environnement tel que $j = 1, .., s$
Résultat : Un ordonnancement de tâches sur $m$processeurs
début
- $\lambda \geqslant \max(\sum_{T_i \in \Gamma} \frac{w(T_i)}{m}, p(T_i))$
- Soit $w(T_i) = p(T_i)q(T_i)$ quantité de travail utilisée par la tâche T_i
- $\mathcal{G}_\mathcal{C} \longleftarrow \emptyset$
- $\mathcal{G}_\mathcal{F} \longleftarrow \emptyset$
- **pour** j = *1 à* s **faire**
  - $\Gamma_{E_j} = \{T_i / e(T_i) = j\}$
- **pour** *chaque* Γ_{E_j} **faire**
  - Ordonner les tâches suivant un ordre décroissant de leur nombre de processeurs :
  - $\Gamma_{E_j} = \{T_1, T_2, \cdots, T_r\}$ tel que $q(T_1) > q(T_2) > \cdots > q(T_r)$
  - `InitFeuilleté` (Γ_{E_j})
  - **tant que** $FinG \neq T_r$ **faire**
    - **si** $p(G_f^j) \leqslant 2\lambda$ **alors**
      - **si** $q(FinG + 1) > \frac{1}{2}q(DebutG)$ **alors**
        - `AjoutTâche` $(\Gamma_{E_j}, G_{\mathrm{f}}^j)$
      - **sinon**
        - `InitFeuilleté` (Γ_{E_j})
    - **sinon**
      - `GroupeCanonique` (G_{f}^j)
    - **pour** *Tout* ${G_f}^j \in {\mathcal{G}_\mathcal{F}}^j$ *dans le sens inverse* **faire**
      - **si** $q(G_c^j) > q(DebutG)$ **alors**
        - Ajouter toutes les tâches de G_{f}^j dans G_c^j
        - ${\mathcal{G}_\mathcal{F}}^j \longleftarrow {\mathcal{G}_\mathcal{F}}^j \backslash G_{\mathrm{f}}^j$
        - `GroupeCanonique` (G_{f}^j)
      - Ordonnancement des tâches de G_c^j sur $q(G_c^j)$ processeurs en utilisant l'algorithme de liste
- Soient ${\mathcal{G}_\mathcal{F}}^j$ et $\mathcal{G}_c^j$ ensembles des MétaTâches tel que
- MétaTâche $= ([DebutG, FinG], p(G_k^j), q(G_k^j))$
- $\mathcal{G}_\mathcal{C} \longleftarrow \mathcal{G}_\mathcal{C} \cup G_c^j$
- $\mathcal{G}_\mathcal{F} \longleftarrow \mathcal{G}_\mathcal{F} \cup \mathcal{G}_F^j$
- Ordonnancement des *MétaTâches* de $\mathcal{G}_\mathcal{F}, \mathcal{G}_\mathcal{C}$ sur m processeurs en utilisant l'algorithme de liste

fin

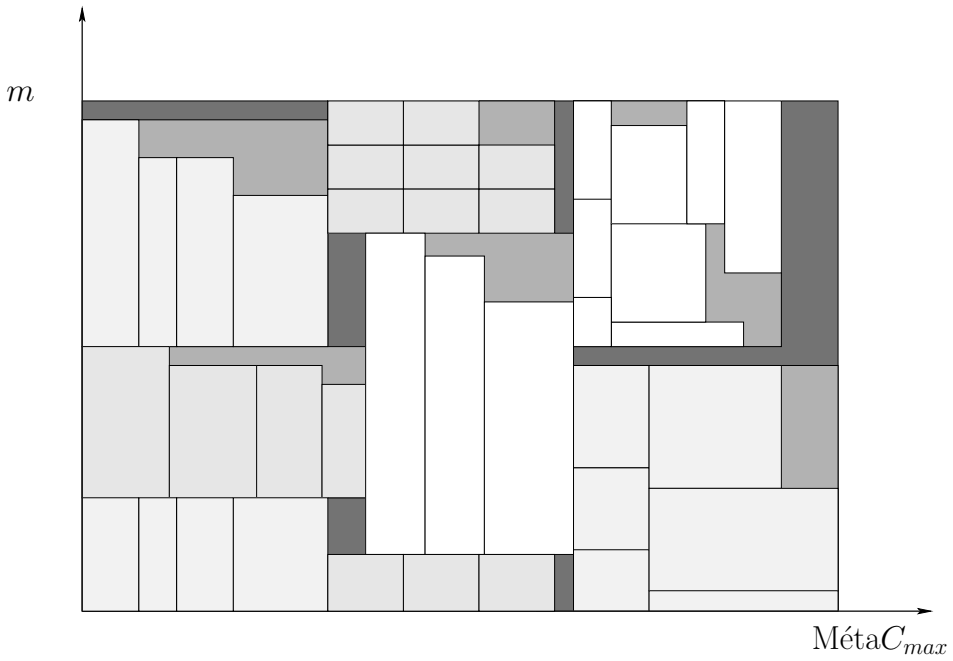

FIG. 3.9 – Ordonnancement des MétaTâches

L'approche budget relaxée pour notre problème consiste à choisir une valeur de λ qui induise un ordonnancement réalisable. C'est-à-dire, $C^*_{max} \leqslant \lambda$. Nous allons aussi supposer que λ ne dépasse pas la valeur d'un ordonnancement de liste pour le problème d'ordonnancement des tâches rigides soit, $\lambda \leqslant 2C^*_{max}$.

Considérons maintenant la nouvelle instance pour le problème d'ordonnancement. Il s'agit là d'ordonnancer des MétaTâches qui nécessitent un nombre de processeurs, un temps d'exécution et un environnement sur m processeurs.

Les résultats de Graham sur cette instance induisent les propriétés suivantes :

$$\max_{G^j_k \in \mathcal{G}_\mathcal{F}, \mathcal{G}_\mathcal{C}} p(G^j_k) \leqslant 2\lambda, \tag{3.15}$$

et

$$\sum_{G^j_k \in \mathcal{G}_\mathcal{F}, \mathcal{G}_\mathcal{C}} \frac{w(T_i)}{m} \leqslant 2\lambda, \tag{3.16}$$

où $p(G^j)$ est la longueur de la plus grande MétaTâche. $\sum_{G_i \in \mathcal{G}_\mathcal{F}, \mathcal{G}_\mathcal{C}} \frac{w(T_i)}{m}$ représente ici le travail total de toutes les MétaTâhes divisées par le nombre de processeurs qui sont deux bornes inférieures du *makespan* pour une instance de MétaTâches. Nous

allons démontrer notre résultat par l'absurde.
Notons par MétaC^*_{max} le *makespan* optimal tel que Méta$C^*_{max} \leqslant 2\lambda$. Soit Méta$C_{max}$ le *makespan* MétaC_{max} de tout algorithme de liste pour une instance composée de MétaTâches.

Supposons que

$$\text{Méta}C_{max} \geqslant 2\text{Méta}C^*_{max}. \tag{3.17}$$

Notons par $r(t)$ la quantité de processeurs utilisés par les MétaTâches à l'instant t. Comme nous l'avons précisé dans la section 2.4.4, une des propriétés fondamentales de tout algorithme de liste est qu'à deux instants distincts t_1 et t_2, il ne peut y avoir des tâches qui s'exécutent simultanément à ces deux instants.

Formellement :

$$\forall t_1, t_2 \in [1, \text{Méta}C_{max}) \quad \text{tels que} \quad t_1 \leqslant t_2 - \text{Méta}C^*_{max},$$

on a

$$r(t_1) + r(t_2) > m. \tag{3.18}$$

En intégrant cette relation on obtient :

$$\int_0^{\text{Méta}C_{max}} r(t)dt \leqslant m \ \text{Méta}C^*_{max}.$$

Ce qui se réécrit,

$$\int_0^{2\text{Méta}C^*_{max}} r(t)dt + \int_{2\text{Méta}C^*_{max}}^{\text{Méta}C_{max}} r(t)dt \leqslant m \ \text{Méta}C^*_{max},$$

ou encore,

$$\int_0^{\text{Méta}C^*_{max}} \underbrace{(r(t) + r(t+1))}_{>m} dt + \int_{2\text{Méta}C^*_{max}}^{\text{Méta}C_{max}} \underbrace{r(t)}_{>0} dt. \leqslant m \ \text{Méta}C^*_{max}$$

Le second terme est strictement positif et suivant l'équation 3.18, le premier terme est supérieur a m, ce qui contredit l'hypothèse 3.17. Alors,

$$\text{Méta}C_{max} < 2 \ \text{Méta}C^*_{max} \leqslant 4\lambda.$$

□

Rapport d'approximation pour le nombre de déploiements

Théorème 3.5.3. *L'algorithme **GLS** construit un ordonnancement avec un nombre de déploiements au plus égal à* $2D_\lambda$.

Démonstration. Soit D_λ le nombre de déploiements d'environnement définis par l'approche budget relaxé pour une valeur donnée du budget λ. Soit $D_{sum} = \sum_{j \in E} D^j_{sum}$ le nombre de déploiements définis par l'algorithme **GLS** pour un ordonnancement, dont la longueur est MétaC_{max}. On note par $q_{max}(G^j_{\mathrm{f}})$ le nombre de processeurs du groupe feuilleté le plus large de l'environnement j.

Trois cas sont possibles :

Cas 1 Si pour chaque environnement un seul groupe est induit, et si celui-ci est un groupe canonique, alors on a :

$$D_{sum} = \sum_{j \in E} \Big\lceil \sum_{T_i \in \Gamma_{E_j}} \frac{w(T_i)}{\lambda} \Big\rceil = D_\lambda$$

Cas 2 Si pour chaque environnement un seul groupe est induit, et si celui-ci est un groupe feuilleté, alors on a :

$$D_{sum} = \sum_{T_i \in \Gamma_{E_j}} q(T_i) = D_\lambda$$

Cas 3 Le cas général est le cas où chaque environnement engendre un ensemble de groupes feuilletés et un groupe canonique. On sait que le nombre de processeurs d'un groupe canonique est plus petit que le nombre de processeurs du plus petit groupe feuilleté pour un environnement. Alors, on a

$$D^j_{sum} \leqslant q_{max}(G^j_{\mathrm{f}}) + \frac{q_{max}(G^j_{\mathrm{f}})}{2} + \frac{q_{max}(G^j_{\mathrm{f}}}{4} + \ldots + \frac{q_{max}(G^j_{\mathrm{f}})}{2^i} + q(G^j_c).$$

En utilisant l'équation 3.9, on obtient :

$$D^j_{sum} \leqslant q_{max}(G^j_{\mathrm{f}}) + \frac{q_{max}(G^j_{\mathrm{f}})}{2} + \frac{q_{max}(G^j_{\mathrm{f}})}{4} + \ldots + \frac{q_{max}(G^j_{\mathrm{f}})}{2^i} + \frac{q_{max}(G^j_{\mathrm{f}})}{2^{i+1}}.$$

Soit,

$$D^j_{sum} \leqslant 2q_{max}(G^j_{\mathrm{f}}).$$

En sommant D^j_{sum}, on obtient :

$$\sum_{j \in E} D^j_{sum} \leqslant 2 \sum_{j \in E} q_{max}(G^j_{\mathrm{f}}).$$

Ce qui donne

$$D_{sum} \leqslant 2D_\lambda$$

□

3.6 Conclusion

Nous avons présenté dans ce chapitre un algorithme pour le problème d'ordonnancement bicritère avec déploiement. Les critères considérés sont le *makespan* et le nombre de déploiements. Nous avons défini une nouvelle approche pour aborder ce genre de problème. Étant donné la complexité d'un problème de minimisation du *makespan* pour l'ordonnancement de tâches rigides, il n'est pas facile de reconsidérer un autre critère à minimiser. «L'approche budget relaxée» définie s'inspire des approches abordées dans les problèmes d'optimisation multicritère, *approche budget* mais avec une version relaxée. Il s'agit de fixer une valeur λ, représentant un budget pour le premier critère *makespan* et essayer de construire un ordonnancement de sorte que la borne inférieure pour le nombre de déploiements pour cet ordonnancement soit donnée par D_λ. Nous avons ainsi construit un algorithme **GLS** basé sur cette approche. Pour chaque environnement, l'algorithme regroupe les tâches dans des blocs et considère ces blocs comme des MétaTâches. Il définit ainsi un temps d'exécution et un nombre de processeurs nécessaires pour exécuter toutes les tâches constituant les MétaTâches. Une fois la phase de construction de groupe terminée, vient alors la phase d'ordonnancement. Pour ce faire, nous utilisons un algorithme de liste pour ordonnancer toutes les MétaTâches sur m processeurs.

L'algorithme construit ainsi un ordonnancement dont le *makespan* est au plus égale à 4λ et le nombre de déploiements au plus égale à $2D_\lambda$.

Chapitre 4

Approximation de la courbe de Pareto

Ce chapitre a pour but d'étudier le problème d'ordonnancement bicritère avec déploiment avec l'approche de la courbe de Pareto. Il s'agit de définir la qualité de la solution fournit par l'algorithme **GLS** en la comparant à la courbe de Pareto.

4.1 Introduction

Déterminer la courbe de Pareto pour des problèmes d'optimisation combinatoire n'est pas facile, et cela est du généralement pour deux raisons. La première est à cause du nombre de solutions Pareto optimales qui est généralement expenentiel (intractable). La seconde est du fait que trouver un point Pareto est très souvent $\mathcal{NP}$-difficile. Néaumoins, il y a eu récement un interet et beaucoup de progrets dans le domaine de l'optimisation combinatoire [PY00, EHP01, Ehr01, ARSY99, CW97]. Il s'agit de montrer qu'il est possible de calculer une approximation de la courbe de Pareto en un temps polynômial. De façon informelle, une courbe ϵ-Pareto est un ensemble de solution qui dominent toutes les solutions approximatives pour tout les objectifs.

Dans le domaine de l'ordonnnancement multicritère, on trouve quelques recent travaux qui considèrent l'approche de Pareto. Cheng et al. [CJK98] proposent un algorithme qui calcul en temps polynômial une courbe de Pareto pour le problème d'ordonnancement sur une machine avec des temps d'exécutions qui dependent de la ressources. Ils définissent deux critères a optimiser, le poids des tâches (la consommation de la ressource) et un critère de régularité (un critère qui dépend du temps d'exécution des tâches). Angel et al. [EBK01] ont étudiés un problème d'ordon-

nancement de tâches indépendantes sur des machines parallèles avec deux fonctions objectives à optimiser le *makespan* et le coût. Ils proposent ainsi une FPTAS pour calculer la courbe de Pareto approché. Dans un autre article les même auteurs [EBK03] proposent une méthode générique pour déterminer une FPTAS pour les problèmes d'ordonnancement bicritère qui considère

4.2 Approche de la courbe de Pareto

A la différences des différentes approches abordées dans le chapitre 2, on cherche a retourner non pas une mais plusieurs solutions qui idéalement, prennent la forme d'un ensemble de Pareto. Le caractère souvent $\mathcal{NP}$-difficile de la détermination d'un élément de cet ensemble fait que l'on s'intéresse à un objectif moins ambitieux, celui de la recherche d'un ensemble de solutions deux à deux incomparables dont l'image approche au mieux la courbe de Pareto. Dans ce cas la, on parle d'*ensemble de Pareto approché* dont l'image est une *courbe de Pareto approchée*. Dans le cadre de l'approximation avec garantie de performance, on parle de d'*ensemble de Pareto $\vec{\epsilon}$-approché* dont l'image est une *courbe de Pareto $\vec{\epsilon}$-approchée*. Un tel ensemble domine approximativement pour tout les critères considérés toutes les autres solutions.

Définition 10. *Soit un vecteur $\vec{\epsilon} = (\epsilon_1, ..., \epsilon_k)$, une solution $s \in \mathcal{S}$, $\vec{\epsilon}$-domine une solution s' si $f_i(s) \geqslant (1 + \epsilon_i) f_i(s')$ pour $i = 1, ..., k$.*

Définition 11. *Pour un problème à k critères, un ensemble de Pareto $\vec{\epsilon}$-approché, noté $\mathcal{S}_{\vec{\epsilon}}$, est un ensemble de solution réalisable tel que pour toute solution $s' \in \mathcal{S}$, il existe une solution $s \in \mathcal{S}_{\vec{\epsilon}}$ qui $\vec{\epsilon}$-domine s'.*

Définition 12. *On note $P_{\vec{\epsilon}}$, une courbe de Pareto $\vec{\epsilon}$-approchée est l'ensemble des images des solutions d'un ensemble de Pareto $\vec{\epsilon}$-approché.*

On trouve dans l'article de Papadimitriou et Yannakakis [PY00], une preuve de l'existence pour tout problème multicritère d'une courbe de Pareto $\vec{\epsilon}$-approchée comportant un nombre de solution polynômial en la taille de l'instance et de $\frac{1}{\epsilon}$. [PY00, SO95] discutent sur l'existence de schémas d'approximation polynômiaux pour la construction de la courbe de Pareto $\vec{\epsilon}$-approchées. Nous rappelons ici les résultats fondamentaux de [PY00] qui sont à la base de nos travaux pour l'approche de la courbe de Pareto pour le problème d'ordonnancement bicritère.

Théorème 4.2.1. *[PY00] Pour le problème multicritère et pour tout ϵ, il existe une courbe de Pareto $P_{\vec{\epsilon}}$ dont le nombre de solutions est polynômial en la taille de l'instance et de $\frac{1}{\epsilon}$*

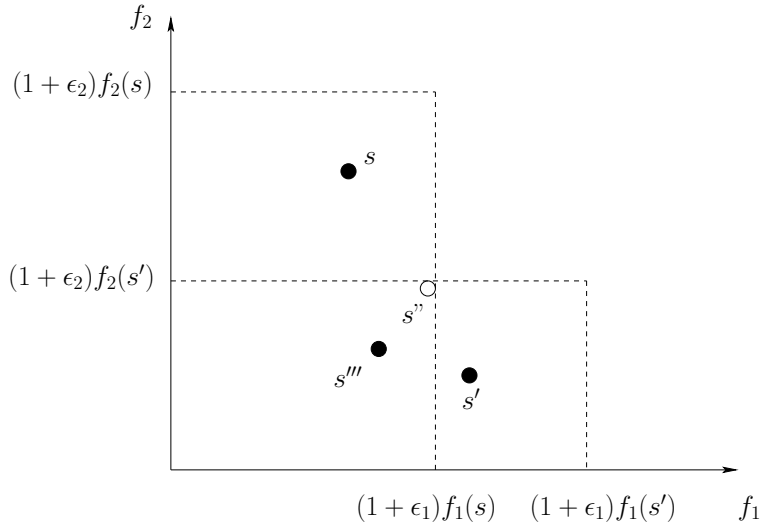

FIG. 4.1 – La solution s" $\vec{\epsilon}$-domine les solutions s et s'

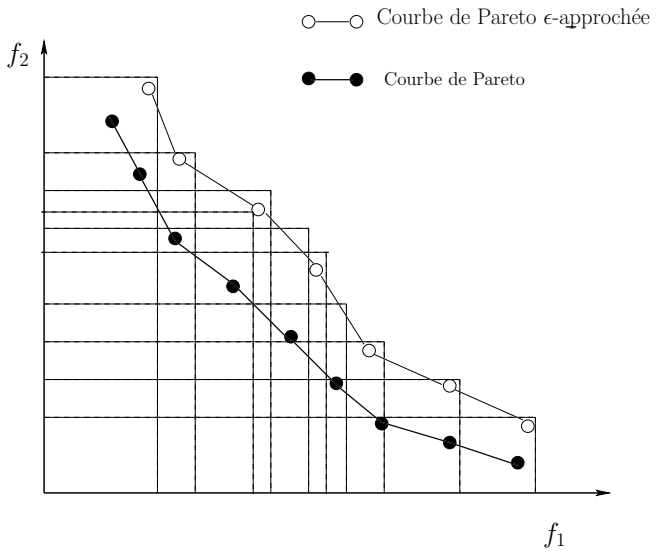

FIG. 4.2 – Courbe de Pareto $\vec{\epsilon}$-approchée

Démonstration. Notons que toutes solution du problème multicritère a une évaluation comprise entre $\frac{1}{2^{p(|x|)}}$ et $2^{p(|x|)}$ où p est un polynôme et x l'instance du problème. Afin de garantir qu'à tout moment le nombre de solution est polynômial, nous utilisons une grille géométrique pour que l'on puisse restreindre le nombre de solutions par boite dans la grille 4.3. Les valeurs atteignables sur chacun des critères f_i sont divisées en intervalles géométriques de raison $(1+\epsilon)$ tel que $\epsilon > 0$, alors il existe $\mathcal{O}(\frac{(2p(|x|))^2}{\epsilon^2})$ intervalles. Soit P ensemble des solutions incomparables, on note par $P_{\vec{\epsilon}} \subseteq P$ un ensemble qui, muni d'une règle déterministe permet de choisir au maximum une solution dans une boite de la grille. On peut alors remarquer que $P_{\vec{\epsilon}}$ est une $\vec{\epsilon}$-approximation de P car pour toute solution $s \in P$ il existe nécessairement dans $P_{\vec{\epsilon}}$ une solution s' tel que $f(s') \leqslant (1+\epsilon_i)f(s)$

□

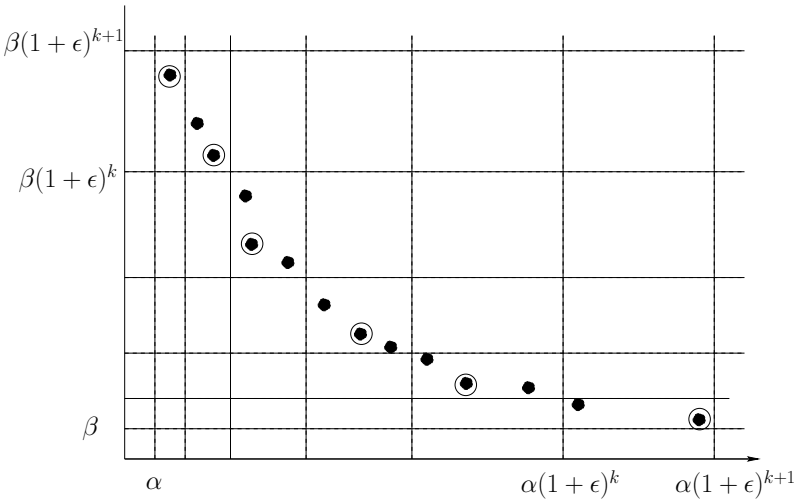

FIG. 4.3 – Subdivision géométrique de l'espace, avec α et β bornes inférieurs

Théorème 4.2.2. *[PY00]*

Il existe un algorithme pour la construction de la courbe de Pareto $P_{\vec{\epsilon}}$ polynômial en la taille de l'instance et $\frac{1}{\epsilon}$ si et seulement si le problème suivant peut être résolus : pour une instance x et les valeurs $(b_1, .., b_k)$, l'algorithme retourne soit une solution s tel que $f_i(x,s) \geqslant b_i$ pour tous i soit retourne une réponse de non existence de s' tel que $f_i(x,s') \geqslant b_i(1+\epsilon)$.

Démonstration. ($\Rightarrow$) supposons qu'il existe un algorithme polynômial en $|x|$ et $\frac{1}{\epsilon}$ pour construire une $\vec{\epsilon}$-approximation. En subdivisant le k-espace en une grille géométrique

(4.2.1), de raison $(1+\epsilon')$ tel que $\epsilon' = \sqrt{1+\epsilon} - 1 \approx \frac{\epsilon}{2}$. Il est facile de vérifier pour une approximation s'il existe ou non une telle solution.
($\Leftarrow$) Supposons que nous avons l'ensemble $P_{\vec{\epsilon}}$. On peut utiliser un algorithme qui vérifie pour les points de la grille s'il existe ou non une solution et qui conserve les solutions dominantes pour obtenir ainsi une $\vec{\epsilon}$-approximation.

□

4.3 Approche de la courbe de Pareto pour le problème bicritère

Afin d'évaluer la performance de notre algorithme **GLS** on définit la notion de la courbe de Pareto ϵ-approchée. Il s'agit de trouver parmis les solutions définit par l'algorithme **GLS**, un ensemble de solutions de taille polynômial qui soit ϵ-approchés de la courbe de Pareto. Comme nous l'avons vu dans le théorème 4.2.1, un nombre polynômial de solutions suffit pour approcher la courbe de Pareto avec une précision aussi petite que l'on veut. On montre ainsi que l'algorithme **GLS** générent une courbe de Pareto ϵ-approchée.

La courbe de Pareto est un ensemble de points dans lequel on distingue deux points particulier les points extrèmes. Évidement, déterminer un point de la courbe de Pareto est un $\mathcal{NP}$-difficile. Néaumoins, ils ont la particularité pour chaque point d'avoir la plus petite valeur pour un des critères parmis toutes les solutions.

Pour l'exemple de la section 2.5.3, on définit dans la figure 4.4 deux solutions qui représentent les points extrèmes de la courbe de Pareto pour cette instance. Le cas (a) est une solution Pareto qui représente un ordonnancement qui a le plus petit nombre de déploiement parmis tout les ordonnancement qui minimisent le *makespan*. Le cas (b) représente un ordonnancement qui a le plus petit *makespan* parmis tout les ordonnancement qui minimisent le nombre de déploiement.

On en déduit que les points Pareto pour cette instance sont allignés sur une droite comme le montre la figure 4.6.

4.3.1 Algorithme de détermination de la courbe de Pareto approchée

Nous présentons dans cette section l'algorithme **CPA** (*Courbe de Pareto approchée*) dont le but est de construire un ensemble de solutions approchant la courbe de Pareto. À chaque étape de l'algorithme, l'ensemble des solutions de l'approche

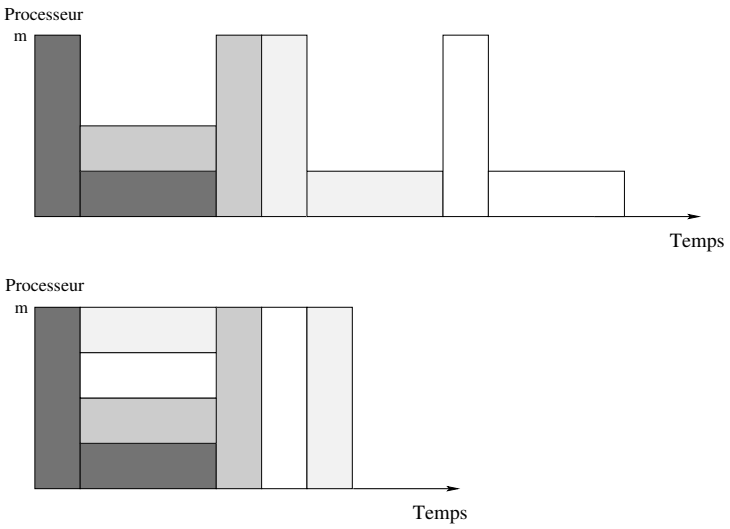

FIG. 4.4 – Diagramme de Gantt illustrant deux solutions Pareto optimales pour l'exemple2.5.3 . Cas(a) Ordonnancement Pareto optimal pour le plus petit nombre de déploiement. Cas(b) Ordonnancement Pareto optimal pour le plus petit makespan

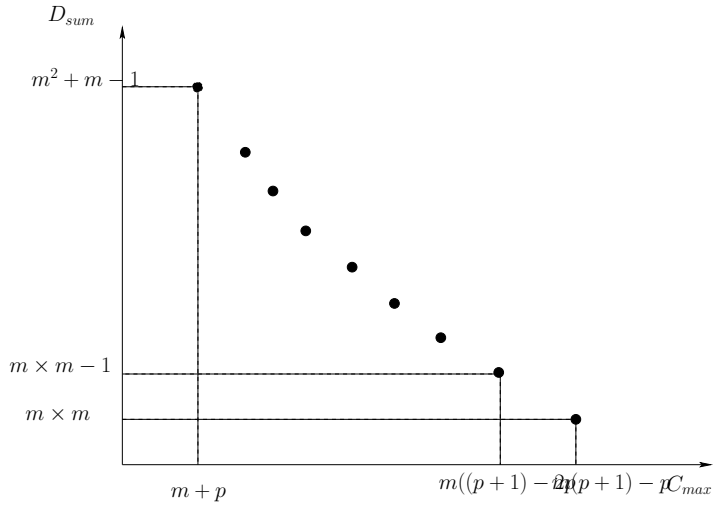

FIG. 4.5 – Courbe de Pareto pour l'exemple

budget relaxée est remplacé par un ensemble de solutions $\vec{\epsilon}$-approché de la courbe de Pareto.

On cherche donc à produire en temps polynômial un ensemble $P_{\vec{\epsilon}}$ de solutions $\vec{\epsilon}$-approché tel que pour tout ordonnancement σ^* (c'est-à-dire un ordonnancement Pareto optimal) il existe dans $P_{\vec{\epsilon}}$ un ordonnancement vérifiant la définition suivante :

Définition 13. *Une courbe de Pareto (α, β)-approchée pour le problème $P|q_j, e_j, p_j|(D_{sum}, C_{max})$ est un ensemble $P_{\alpha,\beta}$ tel que pour toute ordonnancement Pareto optimale σ^*, il existe une solution dans $P|q_j, e_j, p_j|(D_{sum}, C_{max})$ vérifiant :*

$$C_{max}(\sigma) \leqslant \alpha C_{max}(\sigma^*) \tag{4.1}$$

$$D_{sum}(\sigma) \leqslant \beta D_{sum}(\sigma^*) \tag{4.2}$$

Dans ce qui suite, on essaie de mettre à profit l'approche budget relaxée pour construire une courbe de Pareto approchée.

Définition 14. *On définit par σ_1 un ordonnancement de tâches rigides obtenu en utilisant un algorithme qui minimise le makespan (algorithme de liste) sur une instance du problème d'ordonnancement avec déploiement.*

Définition 15. *On définit par σ_2 un ordonnancement de tâches rigides obtenu en utilisant un algorithme qui minimise le nombre de déploiement (algorithme séquentiel) sur une instance du problème d'ordonnancement avec déploiement.*

Algorithme CPA

L'objectif de l'algorithme **CPA** est de construire un ensemble de solutions Pareto approchées sur une instance quelconque pour le problème $P|q_j, e_j, p_j|(D_{sum}, C_{max})$. Suivant le critère considéré dans un l'algorithme basé sur l'approche budget-relaxée **BAR**, c'est-à-dire le *makespan*, les valeurs atteignables sur ce critère sont divisées en intervalles géométrique de raison $(1 + \frac{\epsilon}{4})$ où $\epsilon > 0$ un paramètre. Soit B_{min} le temps de terminaison d'une solution qui minimisant le *makespan*. Soit B_{max} le temps de terminaison d'une solution qui minimisent le nombre de déploiement. Ainsi, on considère les intervalles $[0, B_{min}]$, $]B_{min}, (1+\epsilon)B_{min}], ...,](1+\epsilon)^M B_{min}, B_{max}]$. Pour toutes ces valeurs, l'algorithme basé sur l'approche**BAR**, détermine des solutions (α, β)-budget-approchée-relaxée. Ainsi, l'algorithme **CPA** retourne pour toute solution Pareto optimale σ^* une solution σ vérifiant : $C_{max}(\sigma) \leqslant \alpha C_{max}(\sigma^*)$
$D_{sum}(\sigma) \leqslant \beta D_{sum}(\sigma^*)$.

Algorithme 6 : L'algorithme CPA

Data : un ϵ positif, n tâches rigides,m processeurs **BAR**
Result : $P_{\vec{\epsilon}}$ un ensemble de solution Pareto approchées
begin
  σ_1 une solution définit par un ordonnancement de liste
  σ_2 une solution définit par un ordonnancement séquentiel
  $P_{\vec{\epsilon}} = \{\sigma_1, \sigma_2\}$
  $B_{min} = C_{max}(\sigma_1)$
  $B_{max} = C_{max}(\sigma_2)$
  Soit M tel que $(1+\epsilon)^M B_{min} \leqslant B_{max} \leqslant (1+\epsilon)^{1+M} B_{min}$
  for $i = 1$ *à* M **do**
    $P_{\vec{\epsilon}} := P_{\vec{\epsilon}} \cup \{\mathbf{BAR}(1+\epsilon)^i B_{min})\}$
  return $P_{\vec{\epsilon}}$
end

Théorème 4.3.1. *L'algorithme **CAP** construit une courbe de Pareto $(\alpha(1+\epsilon), \beta)$-approchée pour le problème $P|q_j, e_j|(D_{sum}, C_{max})$ si **BAR** est un algorithme (α, β)-budget-approché-relaxé pour le problème $P|q_j, e_j, C_{max} \leqslant \lambda|D_{sum}$.*

Démonstration. Soit σ^* une solution Pareto optimal vérifiant

$$(1+\epsilon)^{k-1} B_{min} \leqslant C_{max}(\sigma^*) \leqslant (1+\epsilon)^k B_{min}$$

Soit σ une solution (α, β)-budget-approchée-relaxée pour le problème $P|q_j, e_j, C_{max} \leqslant \lambda|D_{sum}$ où $\lambda = (1+\epsilon)^k B_{min}$ on a :

$$D_{sum}(\sigma^*) \geqslant ND_\lambda$$

où ND_λ est le nombre minimum de déploiement pour le problème $P|q_j, e_j, C_{max} \leqslant (1+\epsilon)^k B_{min}|D_{sum}$. Ainsi on peut déduire que :

$$D_{sum}(\sigma) \leqslant \beta ND_\lambda \leqslant \beta D_{sum}(\sigma^*)$$

Comme par définition $C_{max}(\sigma) \leqslant \alpha(1+\epsilon)^k B_{min}$, on peut déduire des inégalités 4.1 et 4.2 que :

$$C_{max}(\sigma) \leqslant \alpha(1+\epsilon) C_{max}(\sigma^*)$$
$$D_{sum}(\sigma) \leqslant \beta D_{sum}(\sigma^*)$$

□

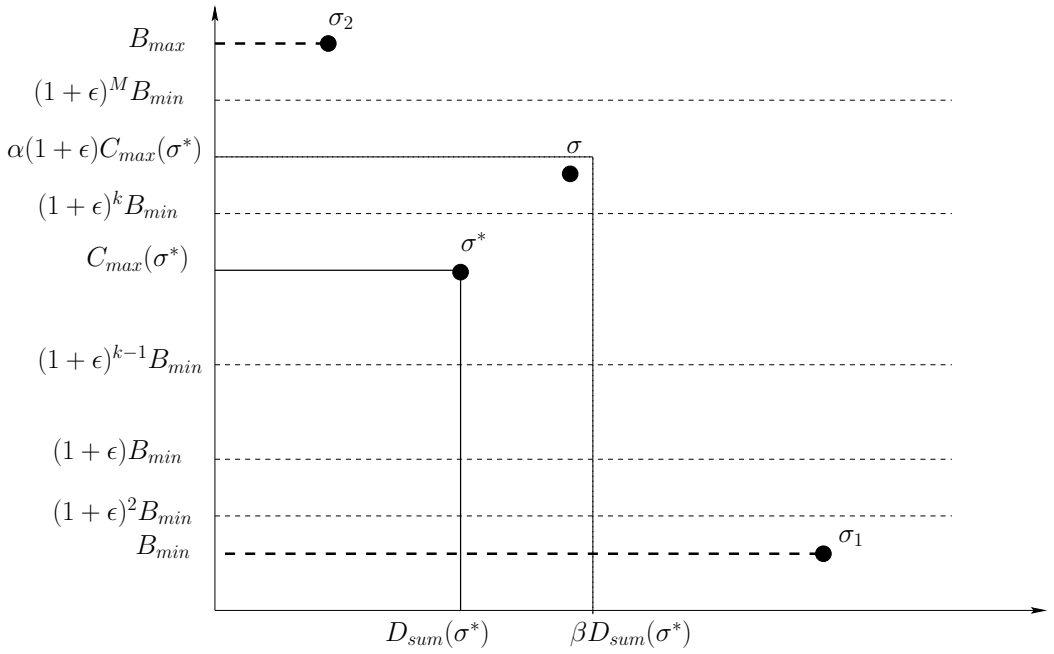

FIG. 4.6 – Illustration de l'algorithme CPA

Théorème 4.3.2. *Il existe un algorithme qui construit en temps Polynômial une courbe de Parto* $(4+\epsilon, 2)$*-approchée pour le problème* $P|q_j, e_j|(D_{sum}, C_{max})$.

Démonstration. Considérons l'algorithme **CPA** appliqué à une instance du problème d'ordonnancement bicritère $P|q_j, e_j, p_j|(D_{sum}, C_{max})$. Soient B_{min} et B_{max} deux solutions tels que, B_{min} est définit par l'algorithme qui

Pareto optimales pour le problème, tel que B_{min} est le plus petit *makespan* pour une solution qui minimise le temps de terminaison des tâches, et B_{max} le plus petit *makespan* pour une solution qui minimise le nombre de déploiement. Étant donné que l'algorithme **GLS** détermine pour toutes les valeurs de λ tel que $\lambda = (1+\frac{\epsilon}{4})B_{min}$ une solution $(4,2)$-budget-approchée-relaxée pour le $P|q_j, e_j, C_{max} \leqslant \lambda|D_{sum}$. Alors **CPA** retourne pour toute solution Pareto optimale σ^* une solution vérifiant :

$$C_{max}(\sigma) \leqslant 4C_{max}(\sigma^*)$$

$$D_{sum}(\sigma) \leqslant 2D_{sum}(\sigma^*)$$

Comme la quantité de budgets considéré ici est polynômial alors il est évident de dire que la courbe de Pareto approchée est calculée en temps polynômial. □

4.4 Conclusion

Nous nous sommes penchée sur le problème d'ordonnancement bicritère. Le caractère $\mathcal{NP}$-difficile du problème monocritère étant déjà connu, nous nous sommes intéressé à l'approche de la courbe de Pareto.

Chapitre 5

Analyse des performances

Ce chapitre présente l'analyse expérimentale de l'algorithme **GLS** pour le problème d'ordonnancement bicritère avec déploiement. Nous présentons dans définissent des distributions et des paramètres associés pour construire des instances d'entrée. Dans leur modèle, les tâches sont représentées par un temps d'exécution et un degré de parallélisme. Nous ajoutons à ce modèle une distribution qui représente les différents environnements associés à chacune des tâches.

5.1 Modèle expérimental

Pour analyser et valider notre algorithme, nous avons décidé de générer des instances synthétiques. Bien qu'il existe des traces sur les grappes de PCs gérées par OAR, celles-ci ne sont pas suffisantes pour nous permettent pas de mener à bien nos expériences. Le nombre de traces n'est pas très représentatif des tâches déployant un environnement. Ainsi, il est nécessaire d'anticiper une étude sur le comportement des futurs utilisateurs. Celui-ci sera forcément différent du comportement actuel sachant que nous nous dirigeons vers une installation automatique des environnements ce qui permettra une utilisation plus intensive du déploiement.

Nos modèles de charge de travail proviennent d'une enquête effectuée par Lublin et Feitelson [LG03] sur le comportement des tâches rigides multiprocesseurs dans un environnement parallèle. Ils définissent des distributions et des paramètres associés pour construire des instances d'entrée. Dans leur modèle, les tâches sont représentées par un temps d'exécution et un degré de parallélisme. Nous ajoutons à ce modèle une distribution qui représente les différents environnements associés à chacune des tâches.

5.1.1 Taille des tâches

L'enquête a montré que la taille des tâches est dominée par une puissance de deux. Même si cette taille n'est pas définie par l'utilisateur, les files d'attente des différents systèmes sont généralement délimitées par une puissance de deux. Les tâches séquentielles sont aussi très observées et représentent une fraction importante.

La distribution définie dans [LG03] partitionne les tâches en trois classes : les tâches de puissance de deux, les tâches séquentielles et les autres tailles de tâches. La procédure de sélection des tâches est définie comme suit : avec une distribution uniforme et une probabilité p_1, les tâches générées sont séquentielles. Dans le cas contraire, le nombre de processeurs est choisi comme nouveau paramètre d'entrée. Avec une distribution uniforme et une probabilité p_2, les tâches générées sont des puissance de deux sinon, elles ont des tailles différentes non spécifiques et supérieures à un. Cette procédure est définie dans l'enquête de Feitelson par distribution "two stage uniform".

5.1.2 Temps d'exécution des tâches

Comme nous l'avons noté plus haut, le temps d'exécution est une caractéristique majeure des tâches rigides. La distribution des temps d'exécution est vraiment différente de celle des tailles des tâches. Il s'agit d'un cas continu, qui devient discret et qui prend un intervalle de valeurs bien plus important. La meilleure distribution retenue dans l'enquête de Feitelson est la distribution "hyper-Gamma" avec les paramètres α_1, β_1, α_2, β_2, et la probabilité p où α_i, β_i (avec $i = 1, 2$) définissent l'amplitude pour un mode de temps. Cette distribution est bimodale, et permet de représenter des tâches de différentes longueurs [LG03]. Ainsi, avec la probabilité p le temps d'exécution de la tâche sélectionnée suit la première distribution de Gamma donc des petits temps d'exécution, sinon la tâche sélectionnée suit la deuxième distribution de Gamma donc de grand temps d'exécution.

5.1.3 Environnement des tâches

La spécificité de notre modèle réside bien sûr au niveau des environnements de tâches. Il n'existe pas dans la littérature de distribution qui représente un modèle de tâches avec choix d'environnements. Nous avons décidé de définir une distribution uniforme sur le nombre d'environnements pour chaque instance.

5.2 Plan d'expérience

Nous avons développé un générateur de tâches rigides avec déploiement d'environnement sur une plate-forme synthétique. Les instances générées comportent toutes 200 processeurs, un nombre de tâches variant entre 100 à 500 et de deux niveaux valeurs pour le nombre d'environnement : 4 et 25. Ce nombre de processeurs correspond à une grappe de taille standard dans Grid'5000. 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. Pour cela nous avons fait varier le nombre de tâches afin d'observer cette dépendance. Pour plus de réalisme dans nos expériences, nous ne négligeons plus le temps de déploiement de chaque environnement, il vaut une unité de temps.

Une des spécificités de notre problème réside au niveau du degré de parallélisme des tâches, puisque le nombre de déploiements lui est fortement lié. C'est pour cette raison que nous avons testé quatre modèles de charge de travail différents. Un premier modèle de tâches de base, avec les paramètres de Feitelson, un deuxième modèle de tâches séquentielles, un troisième modèle de tâches parallèles et le quatrième modèle de tâches mixtes (modèle qui considère la même proportion de tâches parallèles et de tâches séquentielle). Pour chacun de ces modèles, il est nécessaire d'étudier l'impact des temps d'exécution des tâches. Pour cette raison, on considère pour chaque modèle trois sous modèles de tâches : des tâches avec un petit temps d'exécution, des tâches avec des grands temps d'exécution et des tâches mixtes qui considèrent les deux tranches de temps. Les tableaux 5.1, 5.2, 5.3 et 5.4 présentent un résumé numérique des différentes valeurs des paramètres pour les quatre modèles de tâches.

	Temps d'exécution
	$p = 0.70$
$p_1 = 0.24$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$
$p_2 = 0.75$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$

TAB. 5.1 – Valeurs des paramètres pour le modèle de Feitelson

	Petit temps d'exécution	Grand temps d'exécution	Temps mixtes
	$p = 0.10$	$p = 0.80$	$p = 0.50$
$p_1 = 0.90$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$
$p_2 = 0.10$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$

x

TAB. 5.2 – Valeurs des paramètres pour le modèle de tâches séquentielles

	Petit temps d'exécution	Grand temps d'exécution	Temps mixtes
	$p = 0.10$	$p = 0.80$	$p = 0.50$
$p_1 = 0.10$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$
$p_2 = 0.90$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$

TAB. 5.3 – Valeurs des paramètres pour le modèle de tâches parallèles

	Petit temps d'exécution	Grand temps d'exécution	Temps mixtes
	$p = 0.10$	$p = 0.80$	$p = 0.50$
$p_1 = 0.50$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$	$\alpha_1 = 4.00$, $\beta_1 = 1.00$
$p_2 = 0.50$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$	$\alpha_2 = 500.0$, $\beta_2 = 0.02$

TAB. 5.4 – Valeurs des paramètres pour le modèle de tâches mixtes

5.2.1 Métriques d'évaluation

Notre objectif est d'évaluer la qualité de la solution de l'algorithme **GLS**. Pour ce faire, nous allons le comparer à des heuristiques de liste avec différentes politiques d'ordonnancement que nous détaillerons plus bas (paragraphe 5.2.2).

Pour chaque valeur des paramètres (nombre de tâches, modèle de tâches séquentielles, modèle de tâches parallèles et le modèle des tâches de Feitelson), nous effectuons 30 expériences. Pour chacune d'elles, nous générons une instance aléatoire que l'on donne en entrée aux algorithmes d'ordonnancement. Concernant les métriques d'évaluation, nous nous comparons aux performances d'un ordonnancement idéal pour le *makespan*. Ainsi, pour chaque instance, nous calculons la borne C^{lb}_{max} pour le *makespan* et $D_{C^{lb}_{max}}$ pour le nombre de déploiements. Il est claire que $D_{C^{lb}_{max}}$ n'est pas une borne inférieure absolue. Comme nous l'avons vu dans le chapitre 3, $D_{C^{lb}_{max}}$ est le plus petit nombre de déploiement pour un ordonnancement de longueur $\lambda = C^{lb}_{max}$. Nous avons ainsi normalisé les résultats pour chaque expérience, et calculé la moyenne arithmétique des ratios pour les 30 expériences de chaque modèle.

On définit ainsi les deux ratios de performance pour chaque expérience et chaque heuristique par :

$$r_{C_{max}} = \frac{C_{max}}{C^{lb}_{max}} \quad \text{et} \quad r_{D_{sum}} = \frac{D_{sum}}{D_{C^{lb}_{max}}}. \tag{5.1}$$

Remarque 5. *Le ratio du nombre de déploiements peut être dans un certain cas inférieur à 1. En présence de nm tâches séquentielles, m processeurs et n environnements tel que $n < m$, l'ordonnancement qui donne le nombre de déploiements optimal consiste à ordonnancer les tâches sur m processeurs, ainsi le nombre de déploiements D_{sum} est égal au nombre d'environnements n, dans ce cas $D_{C^{lb}_{max}}$ est supérieure à D_{sum} puisque la borne inférieure pour le makespan est égale à n, ainsi le nombre minimum de déploiements pour un makespan égal à n est d'au moins m 3. Cependant, comme dans nos expériences, on ne considère que des ordonnancements de listes qui ne pourront jamais produire un ordonnancement sur uniquement n processeurs, il y a peu de chance d'obtenir $r_{D_{sum}} < 1$ (voir 5.1).*

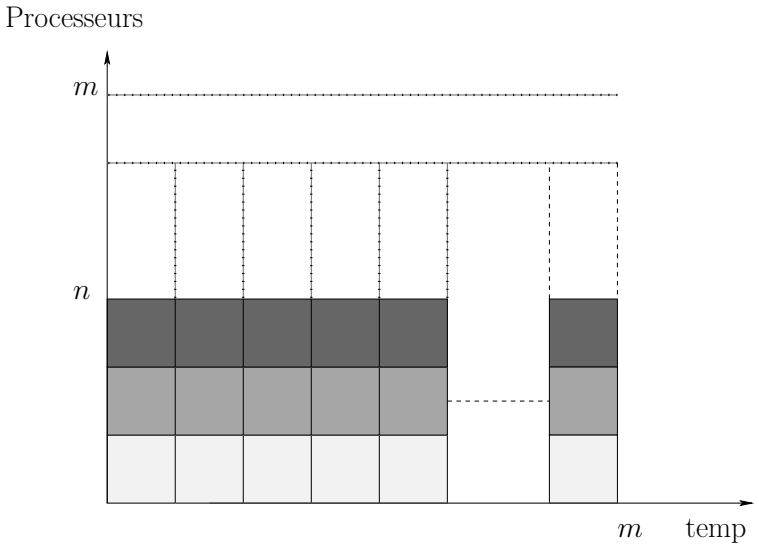

FIG. 5.1 – Illustration de la remarque 5

5.2.2 Algorithme de comparaison

Pour évaluer et juger du comportement et de l'efficacité de l'algorithme **GLS**, nous avons implémenté 6 autres algorithmes de liste avec différentes politiques d'ordonnancement. Ces heuristiques nous permettent aussi d'évaluer le comportement

de notre algorithme, afin de déterminer les paramètres qui influencent le plus les résultats.

- **Processeurs décroissants** Cet algorithme de liste ordonnance les tâches avec une priorité suivant l'ordre décroissant du nombre de processeurs.
- **Processeurs croissants** Cet algorithme de liste ordonnance les tâches avec une priorité suivant l'ordre croissant du nombre de processeurs.
- **Temps décroissant** Cet algorithme de liste ordonnance les tâches avec une priorité suivant leur temps d'exécution décroissant.
- **Temps croissant** Cet algorithme de liste ordonnance les tâches avec une priorité suivant leur temps d'exécution croissant.
- **Environnements décroissants** Cet algorithme de liste dans une première étape, regroupe les tâches suivant leur environnement. Par la suite ordonnance les groupes de tâches suivant un ordre décroissant du nombre de processeurs des groupes.
- **Environnements alternés** Cet algorithme regroupe les tâches suivant leur environnement. Il exécute ainsi les tâches suivant un ordre décroissant du nombre de processeurs, en alternant suivant un Round-robin (tourniquet) par rapport aux environnement.

5.3 Analyse des résultats

Nos expériences nous ont permis de remarquer que le temps d'exécution n'avait pas d'influence sur les ratios de performance. Cela est dû principalement au choix de la borne inférieure du *makespan* qui permet de normaliser les résultats. Pour cette raison, nous n'exposons que les courbes présentant des tâches avec des temps mixtes
.

En ce qui concerne les paramètres qui peuvent influencer les résultats, nous avons observé une corrélation entre le nombre de tâches et le nombre de déploiements et entre le nombre de tâches et le *makespan*. Nous avons ainsi calculé ces cœfficients pour chaque modèle de tâches. Les valeurs sont affichées pour chaque heuristique dans les tableaux 5.6 et 5.5.

Comme on pouvait s'y attendre, en ce qui concerne les cœfficients de corrélation pour le *makespan* on remarque une forte corrélation pour tout les modèles de tâches.

Par contre, la première observation que l'on peut faire au niveau du tableau 5.6, est que lorsqu'il s'agit du modèle de tâches parallèles, du modèle de tâches mixtes ou du modèle tâches de Feitelson, les cœfficients de corrélation avec 25 environnements

sont dans l'ensemble plus importants que pour le modèle de tâches séquentielles. Cette différence est due au degré de parallélisme des tâches des différents modèles et la proportion du nombre de tâches par environnement. Lorsque les tâches sont séquentielles, et avec un nombre important d'environnements, le nombre de déploiements n'est pas aussi important que les autres modèles de tâches, même si le nombre de tâches est conséquent.

Aussi, pour modèle de tâches séquentielles, du modèle de tâches mixtes ou du modèle tâches de Feitelson, les cœfficients de corrélation avec 4 environnements sont dans l'ensemble plus importants que pour le modèle de tâches Parallèles. Cette différence est due au nombre d'environnements, le nombre de tâches par environnements et le degré de parallélisme de celles-ci.

On observe aussi pour chaque heuristique une légère différence au niveau des cœfficients de corrélations entre le modèle avec 4 environnements et le modèle avec 25 environnements. En effet, les quatre modèles ont des cœfficients de corrélation plus importants avec 25 environnements que ceux avec 4 environnements pour toutes les heuristiques. La différence la plus significative est au niveau de **GLS**. Cette différence s'explique par le degré de parallélisme des tâches et le nombre d'environnement disponible. On peut dire qu'en présence de beaucoup plus de tâches et peu d'environnements, **GLS** sera en mesure d'avoir les meilleurs performance en terme du nombre de déploiements par rapport aux autres heuristiques de comparaison. Effectivement, **GLS** reconstruit des tâches (MétaTâches) avant de les ordonnancer. Celles-ci sont regroupées en longueur d'abord, ainsi, le nombre de déploiements pour chaque MétaTâche est comptabilisé pour qu'un petit nombre de tâches (voir chapitre 3).

		PC	PD	TC	TD	EA	ED	GLS
Tâches Feitelson	4 environnements	0.98	0.98	0.97	0.98	0.98	0.98	0.95
	25 environnements	0.98	0.98	0.98	0.98	0.98	0.98	0.97
Tâches séquentielles	4 environnements	0.95	0.96	0.95	0.94	0.95	0.96	0.91
	25 environnements	0.92	0.93	0.92	0.92	0.94	0.94	0.82
Tâche parallèles	4 environnements	0.98	0.98	0.98	0.98	0.98	0.98	0.97
	25 environnements	0.99	0.99	0.99	0.99	0.99	0.99	0.97
Tâches mixtes	4 environnements	0.97	0.97	0.97	0.97	0.97	0.97	0.95
	25 environnements	0.96	0.96	0.96	0.96	0.97	0.97	0.95

TAB. 5.5 – Tableau des corrélations pour le makespan

		PC	PD	TC	TD	EA	ED	GLS
Tâches Feitelson	4 environnements	0.92	0.55	0.85	0.60	0.65	0.74	0.23
	25 environnements	0.97	0.93	0.96	0.96	0.94	0.98	0.94
Tâches séquentielles	4 environnements	0.85	0.79	0.85	0.78	0.86	0.96	0.75
	25 environnements	0.89	0.90	0.89	0.90	0.92	0.96	0.87
Tâche parallèles	4 environnements	0.92	0.49	0.82	0.51	0.57	0.67	0.19
	25 environnements	0.97	0.95	0.97	0.96	0.95	0.98	0.91
Tâches mixtes	4 environnements	0.90	0.66	0.82	0.73	0.74	0.83	0.43
	25 environnements	0.96	0.94	0.95	0.96	0.94	0.98	0.93

TAB. 5.6 – Tableau des corrélations pour le nombre de déploiements

Afin de valider la moyenne comme métrique d'évaluation, nous présentons sur les figures 5.2 les histogrammes représentant, pour une instance, la densité des distributions des solutions de **TD** pour le *makespan* et le nombre de déploiements. Comme on peut le voir sur ces figures la distribution est gaussienne, ce qui valide la moyenne arithmétique comme métrique de comparaison.

Nous exposons les résultats en donnant une courbe pour chaque combinaison de modèle de temps d'exécution et de parallélisme qui nous semble la plus pertinente.

On remarque que dans tous les modèles de tâches, **GLS** donne les meilleures performances pour le nombre de déploiements. En ce qui concerne le *makespan* c'est **TD** qui fournit les meilleurs résultats. Néanmoins, les ratios de performances varient d'un modèle à un autre, et c'est là que se fera notre analyse.

Considérons dans la figure 5.3 le modèle de tâches de Feitelson. Nous remarquons qu'avec 4 environnements, le ratio pour le déploiement atteint 5.1 pour **PC** avec 500 tâches. On peut voir aussi que ce ratio est constant pour **GLS**. Concernant **PD, TC** et **TD**, le ratio est entre 1.2 et 2.4 avec 100 tâches et entre 2.3 et 3.5 avec 500 tâches. Pour **ED** ce ratio est constant et au plus 4. Le modèle de tâches de Feitelson considère un pourcentage de tâches parallèles important. Ainsi, avec 4 environnements le nombre de tâches par environnements est plus conséquent, les tâches dans cette heuristique sont regroupées en blocs pour être ordonnancées en séquentielements.

Comparé au modèle de tâches avec 25 environnements, le ratio reste le même pour **ED** et atteint 4 dès 200. Ce modèle de tâches est très proche du modèle de tâches de Feitelson avec une proportion de tâches séquentielles très petite. On peut voir que le ratio diminue pour **EA, PD, TC** et **TD** par rapport au modèle de tâches avec 4 environnements. Celui-ci augmente pour **GLS** mais reste le plus petit par rapport

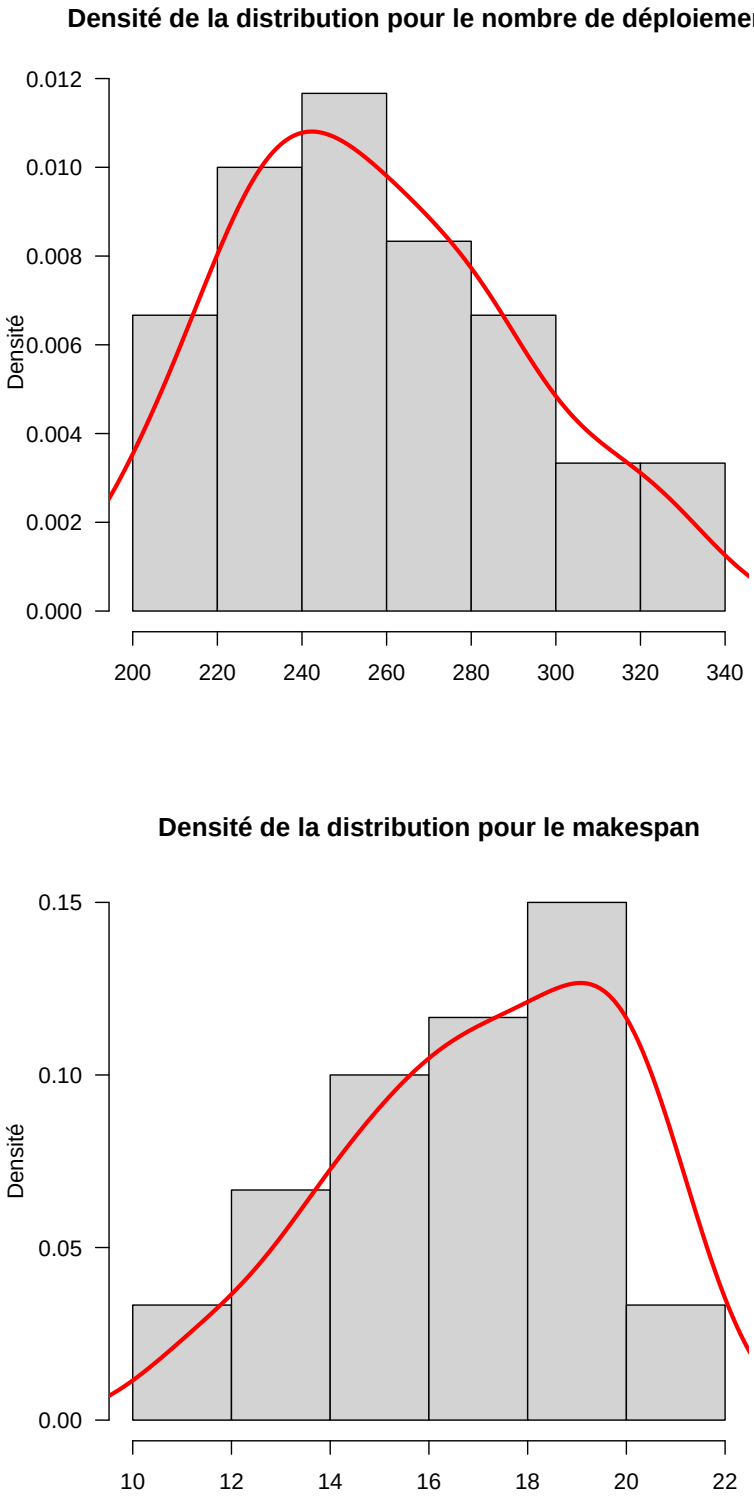

FIG. 5.2 – Densité de distribution des solutions pour une instance de 200 tâches

aux autres heuristiques.

Dans la figure 5.4 du modèle de tâches parallèles, on remarque qu'avec 4 environnements et des tâches fortement parallèles, le ratio pour le déploiement est le plus élevé par rapport aux autre modèles de tâches. Ce dernier atteint 5.4 pour **PC** avec 500 tâches. Les courbes pour **EA, ED, PD, TC** et **TD** restent quasiment constantes lorsque le nombre de tâches augmente. Pour **GLS**, le ratio est le même pour toutes les expériences et vaut 1.2. Celui-ci augmente avec le nombre de tâches pour toutes les heuristiques, au niveau de la deuxième courbe de la figure 5.4 avec 25 environnements. Comparé à la figure au dessus (5.4 avec 4 environnements), le ratio est inférieur pour **PC, PD, TC, TD** et **EA** sauf **ED** ou il est supérieur et atteint 4.8. Pour **GLS** il augmente mais reste inférieur à 2. Cette différence au niveau du ratio s'explique par le fait que lorsqu'on a 25 environnements, le nombre optimal de déploiements augmente d'où un ratio pour le déploiement plus petit. On remarque aussi que **PD** donne une bonne performance comparé aux autres heuristiques.

En ce qui concerne le modèle des tâches séquentielles (avec 4 et 25 environnements), on peut observer sur les courbes de la figure 5.5 que le ratio de performance est moins important que dans le modèle des tâches parallèles pour les heuristiques **PD, PC, TD, TC, EA** et **ED**. Cela est dû principalement au degré de parallélisme des tâches. Pour **GLS** le ratio de performance est légèrement au dessus de 1.2 pour le modèle de tâches, avec 4 et 25 environnements et cette tendance est constante. Les heuristiques **TC, TD** et **TD** ont un ratio largement inférieur à 2 avec 25 environnements.

La figure 5.6 montre un modèle de tâches mixtes : certaines sont petites et faiblement parallèles, alors que d'autres grandes et fortement parallèles. La présence d'un certain nombre de tâches faiblement parallèles entraîne, comme précédemment, un ratio relativement faible pour la plupart des heuristiques. Celui-ci est même constant pour **GLS** avec 4 environnements. Là aussi, on remarque que **PD** a une bonne performance comparé aux autres heuristiques. Le ratio de celle-ci est aussi largement inférieur à 2 avec 25 environnements. On peut conclure sur les courbes de la figure 5.6 que cette tendance confirme bien les résultats observés au niveau des modèles de tâches précédents.

Nous considérons dans la figure 5.7 le modèle de tâches de Feitelson concernant le *makespan*. Nous observons pour ce modèle que les heuristiques **PD, PC, TD, TC, ED** et **EA** donnent de meilleures performances que **GLS**. Avec 25 environ-

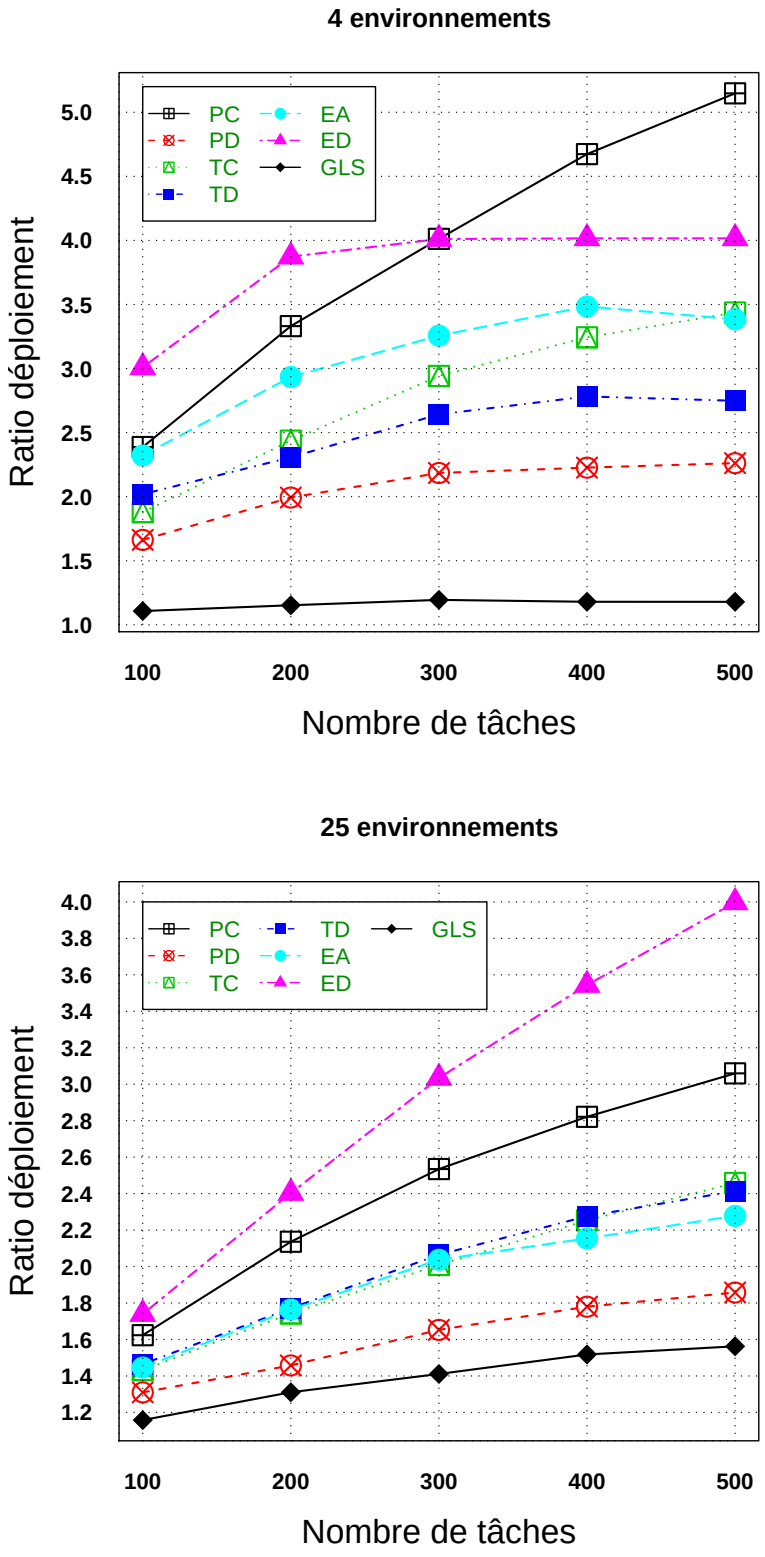

FIG. 5.3 – Ratios de performance du nombre de déploiements pour le modèle de tâches de Feitelson

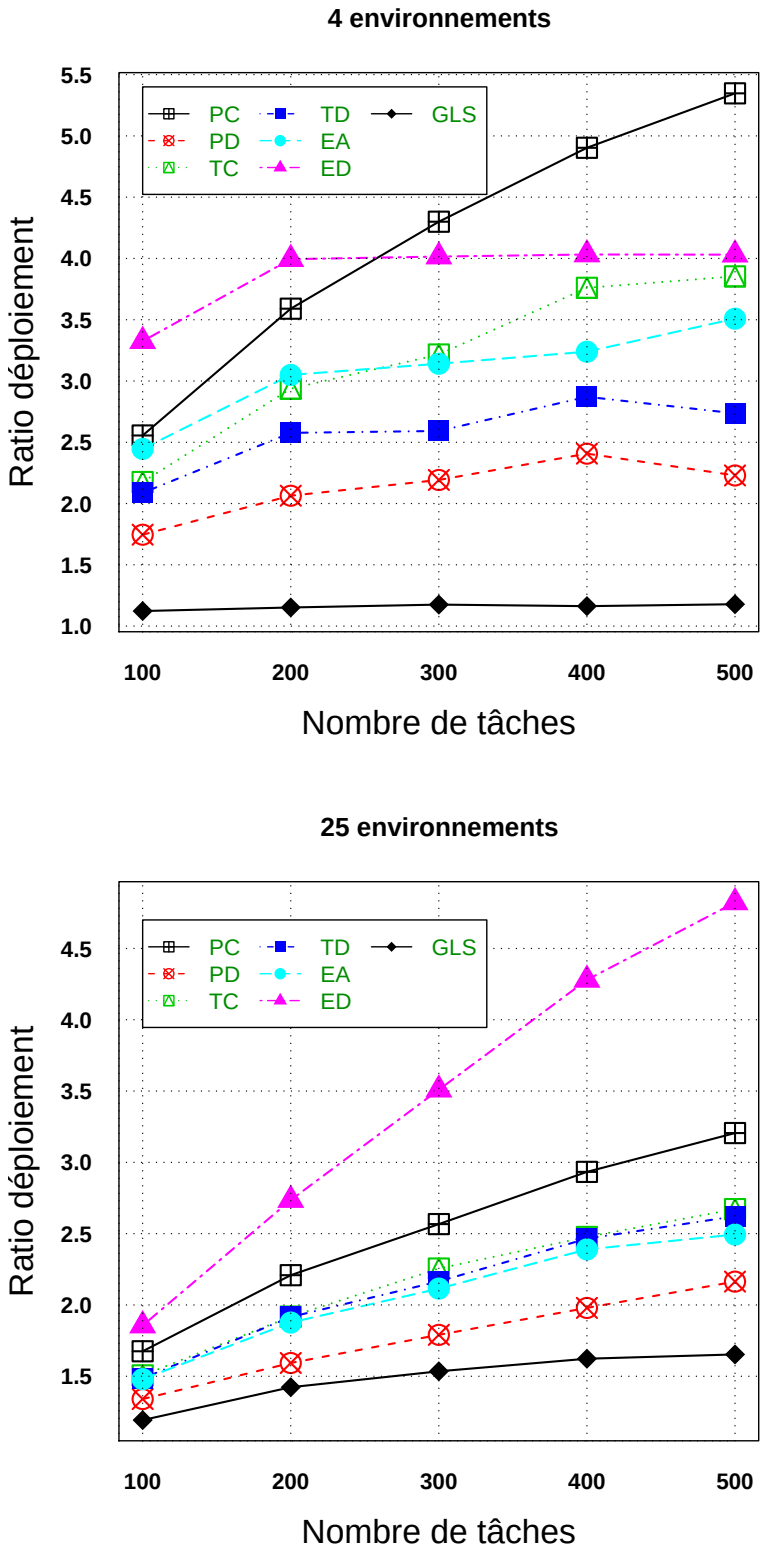

FIG. 5.4 – Ratio de performance du nombre de déploiements pour le modèle de tâches parallèles avec les différents modes de temps d'exécution

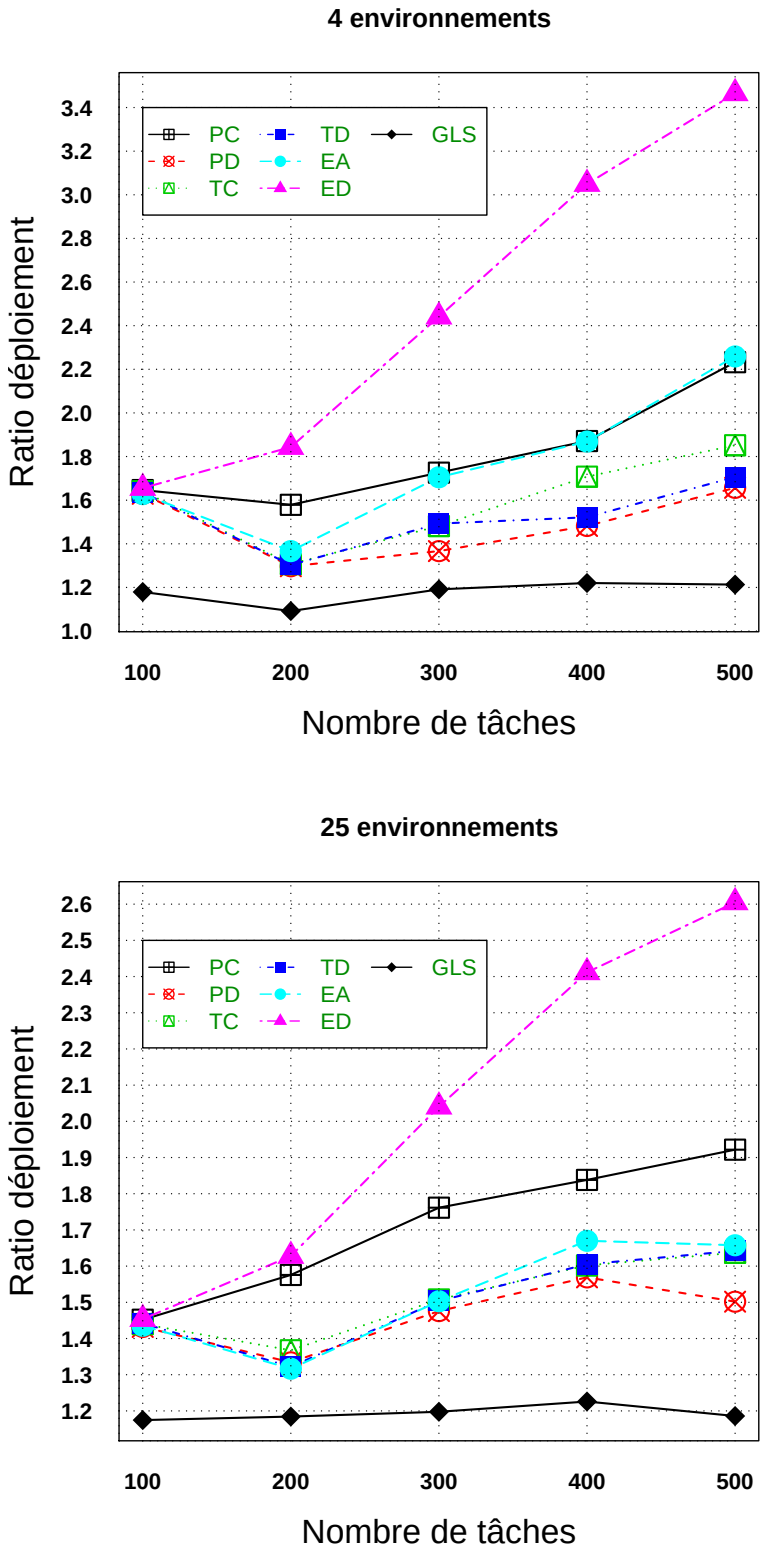

FIG. 5.5 – Ratio de performance du nombre de déploiements pour le modèle de tâches séquentielles avec les différents modes de temps d'exécution

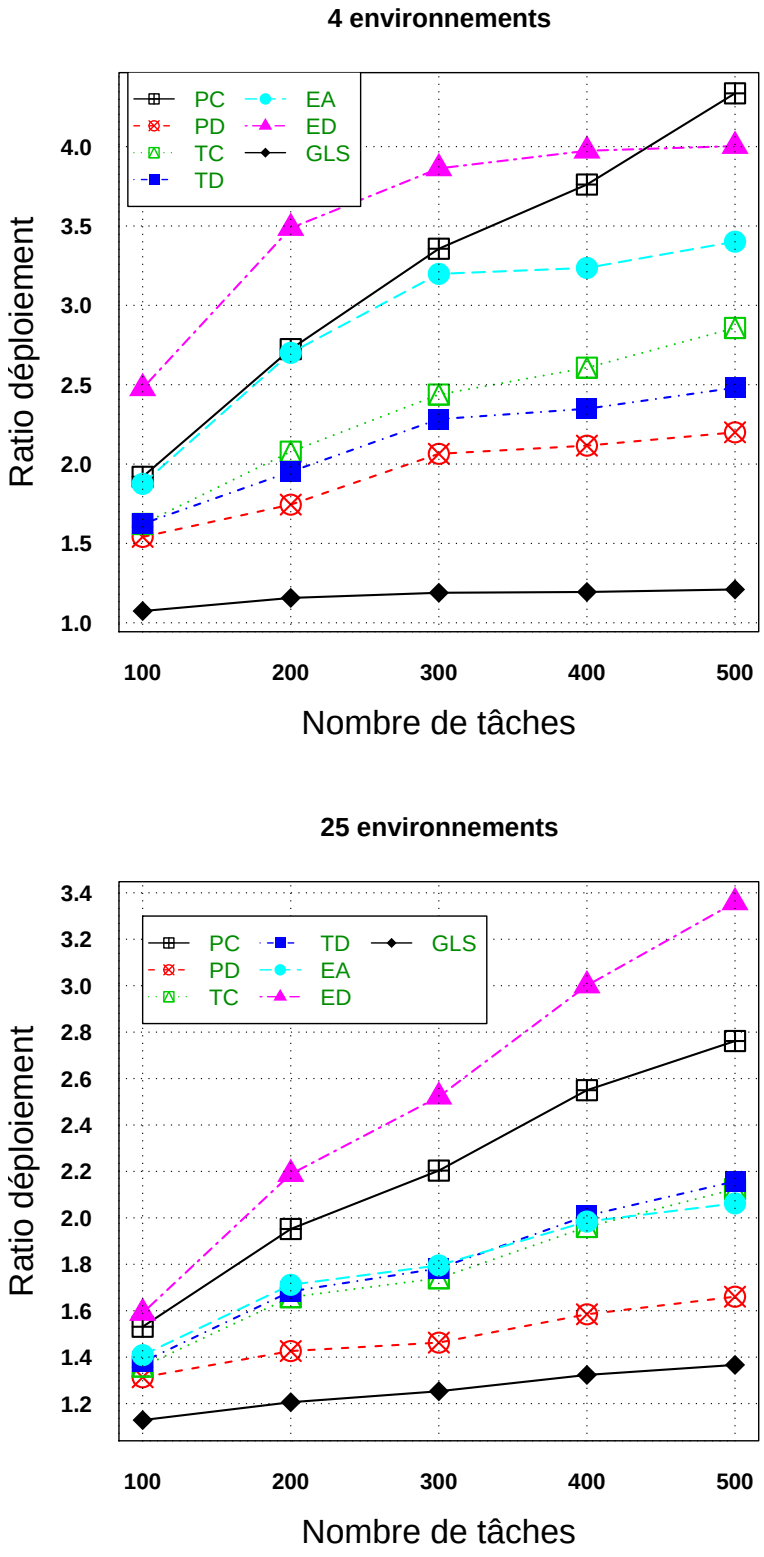

FIG. 5.6 – Ratios de performance du nombre de déploiements pour le modèle de tâches mixtes avec les différents modes de temps d'exécution

nements, **GLS** donne un bonne performance pour le *makespan* comparé aux autres modèles de tâches. Celui-ci décroît pour atteindre 1.3 lorsque le nombre de tâches est le plus élevé. Avec 4 environnements, ce ratio atteint 2.4 avec 500 tâches. Cette différence pour **GLS** entre le modèle de tâches avec 4 environnements et celui avec 25 environnements réside au niveau de la longueur des MétaTâches. Lorsque le nombre de tâches par environnement est important, les MétaTâches construites sont plus longues. Effectivement, avec 25 environnements, le nombre de tâches par environnement n'est pas très élevé. Ainsi, à la phase 1 de l'algorithme **GLS**, les tâches sont regroupées sous forme de MétaTâches. Ces dernières forment des petits groupes dont la longueur varie suivant le nombre de tâches du groupe. À la phase d'ordonnancement, on considère alors un ordonnancement de petites MétaTâches pour chaque environnement.

La figure 5.8 montre les résultats pour les tâches parallèles. On observe pour le modèle de tâches parallèles que les heuristiques **PD, PC, TD, TC, ED** et **EA** donnent les meilleurs performances par rapport à **GLS**. Avec 25 environnements, on peut remarquer que c'est pour le modèle de tâches parallèles que **GLS** donne son meilleur ratio. Celui-ci décroît pour atteindre 1.2 lorsque le nombre de tâches est le plus élevé. Ce ratio dépasse légèrement 2 avec 4 environnements.

La figure 5.9 concerne le modèle de tâches séquentielles. A l'inverse du modèle précédent, on peut voir que le ratio pour **GLS** est inférieur à 2.1 lorsque l'on est en présence de 4 environnements. Avec 25 environnements, celui-ci passe au dessus de 2 avant de décroître à 1.9 pour 500 tâches. On peut observer pour ce modèle de tâches que **GLS** a le même comportement que les autres heuristiques avec un ratio plus grand. Ce résultat est évident du fait que les tâches (séquentielles) ont la même largeur et que **GLS** commence par construire des groupes feuilletées donc des MétaTâches plus longues. En présence de 25 environnements, on a en moyenne le même nombre de tâches par environnement. Ceci implique des MétaTâches feuilletés séquentielles et longues. Le phénomène est presque le même avec 4 environnements seulement, dans ce cas, les MétaTâches construites sont des MétaTâches canoniques. Elles ne sont donc pas toutes séquentielles, ce qui a pour effet de réduire le ratio du makespan.

La figure 5.10 représente le modèle des tâches mixtes en présence de 25 environnements. On observe un ratio très élevé pour le *makespan* comparé aux autres modèles de tâches. Ce ratio décroît progressivement en fonction du nombre de tâches. Comme constatée dans les modèles précédents, cette tendance montre qu'avec beau-

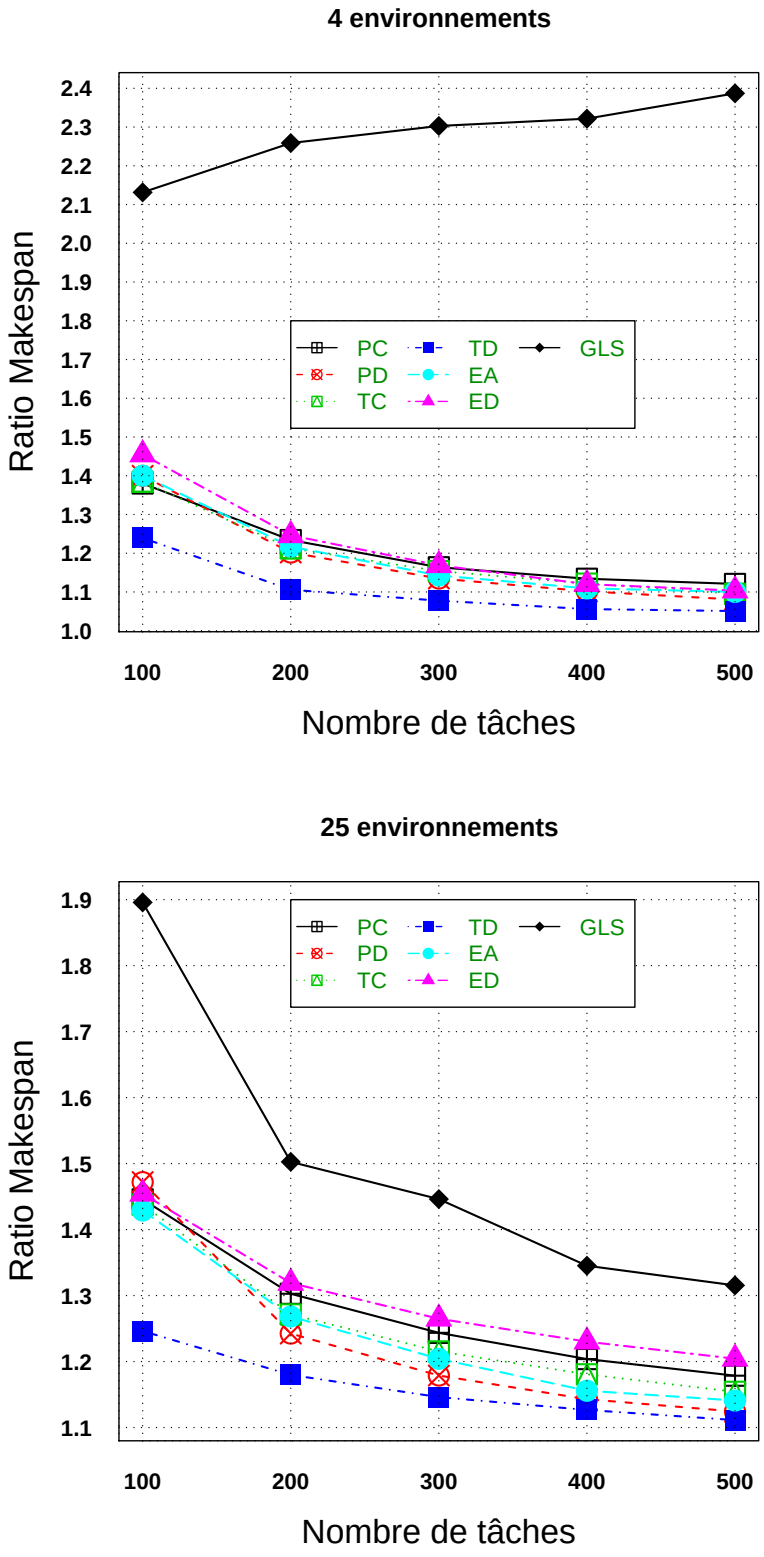

FIG. 5.7 – Ratios de performance du *makespan* pour le modèle de tâches de Feitelson

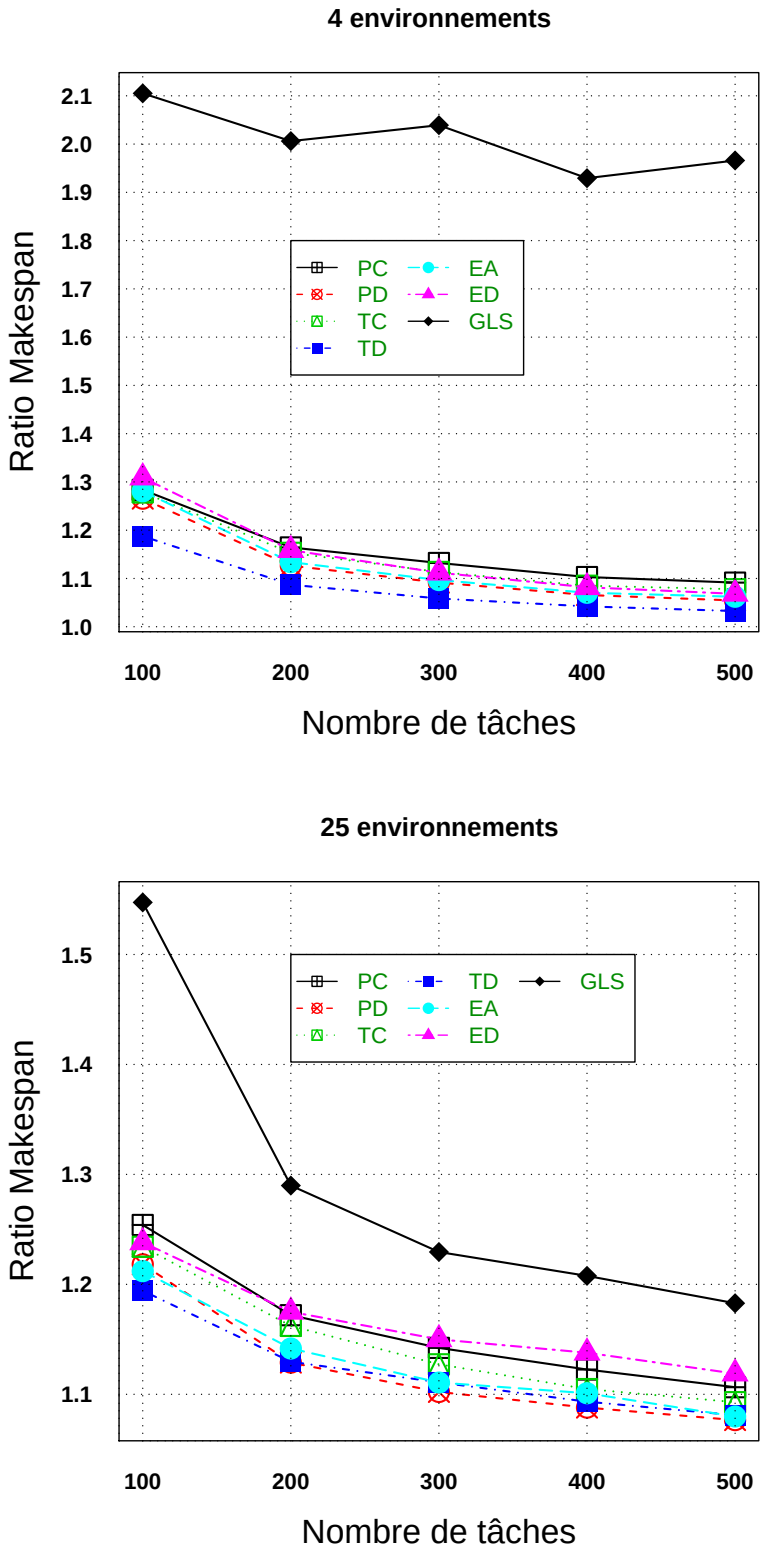

FIG. 5.8 – Ratio de performance du *makespan* pour le modèle de tâches parallèles avec les différents modes de temps d'exécution

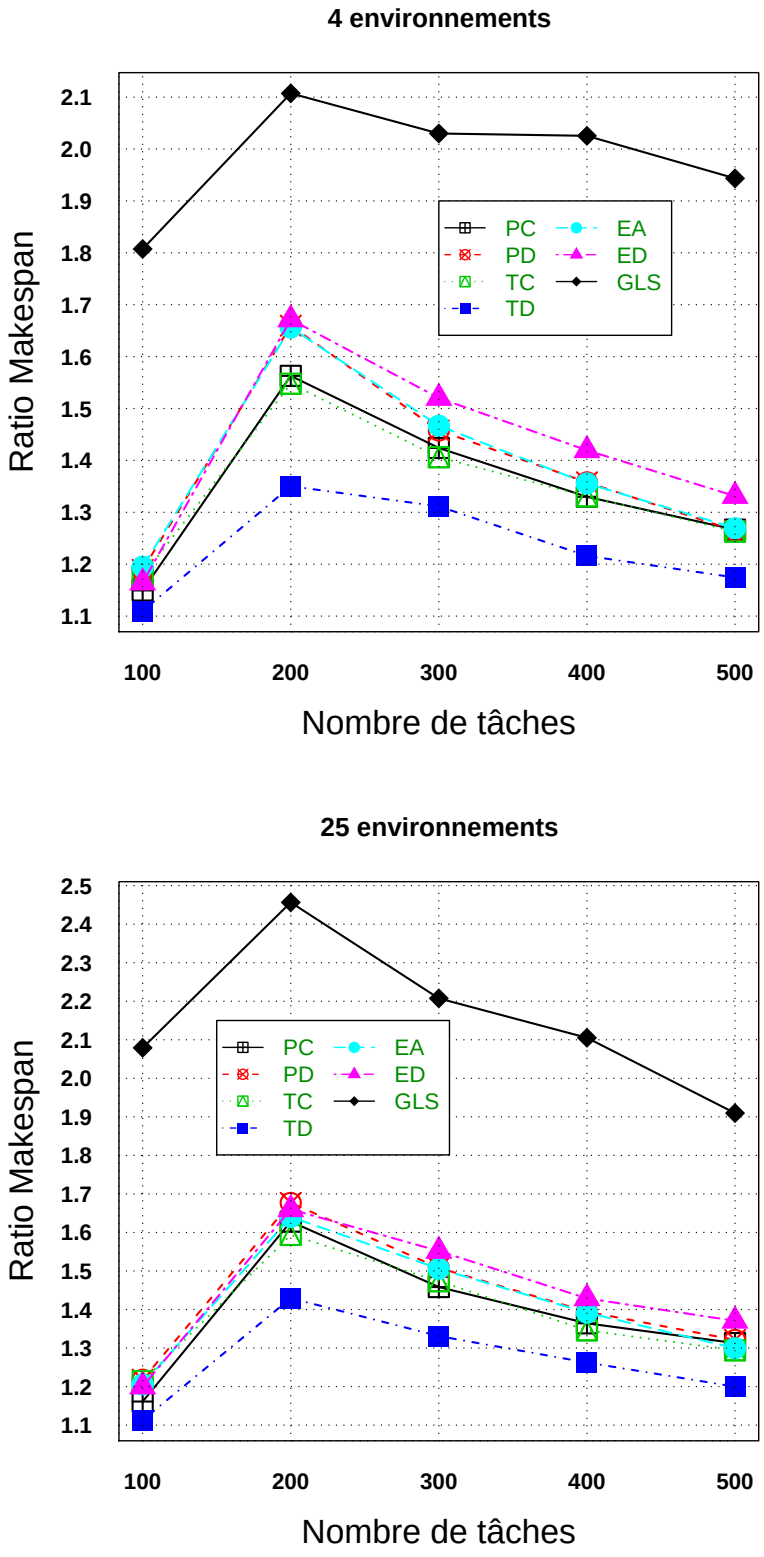

FIG. 5.9 – Ratio de performance du *makespan* pour le modèle de tâches séquentielles avec les différents modes de temps d'exécution

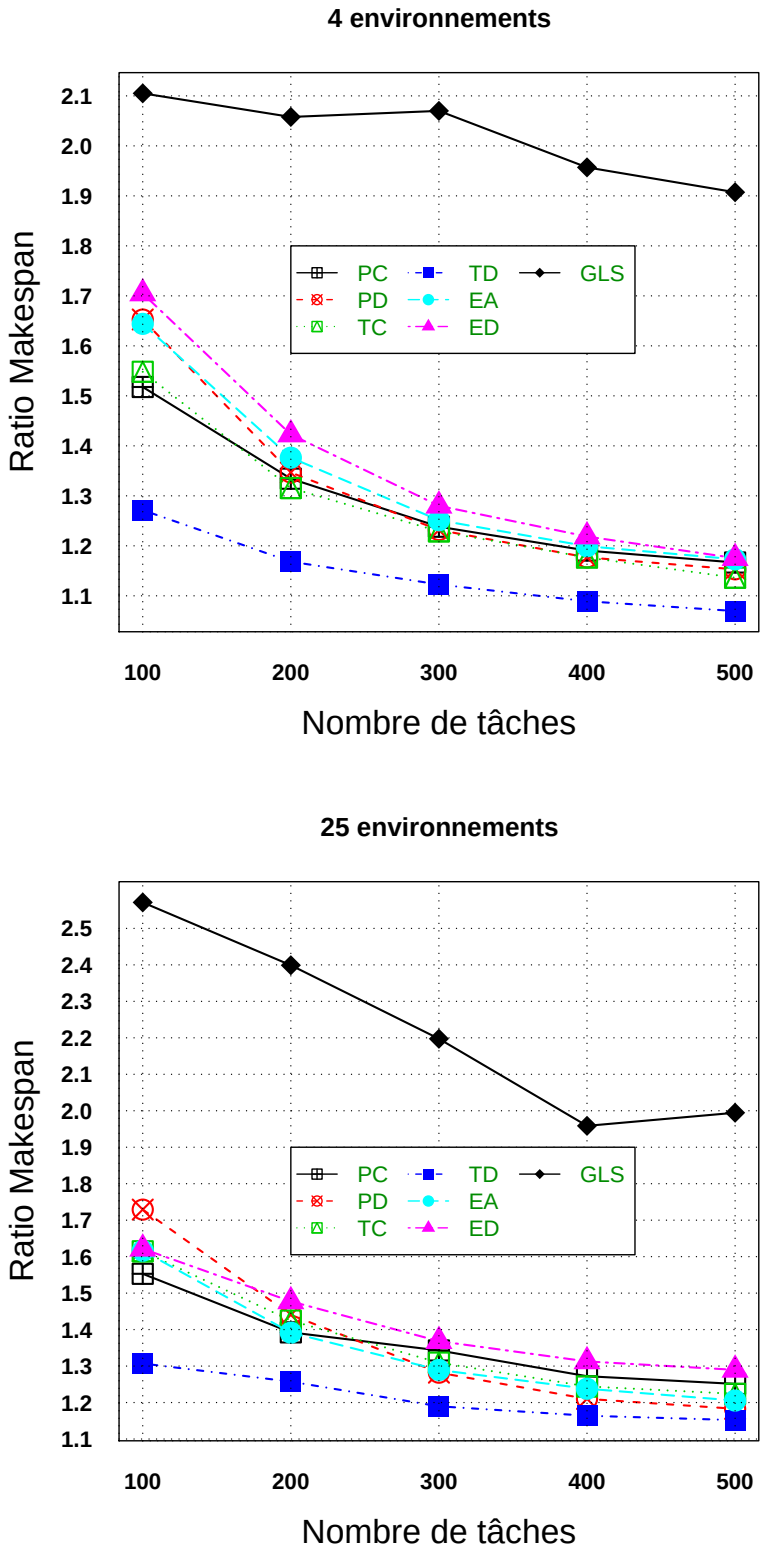

FIG. 5.10 – Ratios de performance du *makespan* pour le modèle de tâches mixtes avec les différents modes de temps d'exécution

coup d'environnements, les groupes de tâches formés sont plus longs, ce qui donne un *makespan* plus grand. En effet, Avec 4 environnements, cette tendance se confirme puisque la quantité de travail des tâches par environnement est plus importante, ce qui donne des MétaTâches plus larges et plus courtes.

5.4 Conclusion

Dans ce chapitre nous nous sommes intéressés à l'évaluation de **GLS** et nous l'avons ainsi comparé à d'autres heuristiques. Pour ce faire, nous avons implémenté un simulateur basé sur le modèle de Lublin et Feitelson [LG03]. Il s'agit d'un modèle qui considère des tâches multiprocesseurs avec différents temps d'exécution. Nous avons ajouté à ce modèle une distribution uniforme afin de prendre en compte pour chaque tâches un environnement. Comme la spécificité de notre problème réside au niveau du degré de parallélisme des tâches, nous avons considéré quatre modèles de tâches (tâches de Feitelson, tâches fortement parallèles, tâches séquentielles, tâches mixtes) afin de considérer le plus de cas possible. Nous avons aussi implémenté des heuristiques basées sur les algorithmes de listes, afin de les comparer à **GLS**. Nous avons constaté que **GLS** a d'excellentes performances en ce qui concerne le nombre de déploiements. Toutefois ces performances sont moins bonnes pour le *makespan*. D'autre part, le ratio concernant le *makespan* pour les quatre modèles de tâches ne dépasse pas les 2.5. Une des premières conclusions que nous pouvons faire est que ces expériences confirment les rapports d'approximation obtenue pour nombre de déploiements et pour le *makespan*. Nous pensons pouvoir améliorer les performances au niveau du *makespan* en diminuant la longueur des MétaTâches, c'est-à-dire en diminuant la valeur de λ.

Conclusion & perspectives

Dans cette thèse nous avons étudié un problème spécifique rencontré au niveau de Grid'5000.

À l'origine, Grid'5000 [Gri06] est le nom d'un programme français initié par l'ACI Grid. L'objectif premier de Grid'5000 est de donner aux chercheurs de différentes disciplines les moyens expérimentaux pour mener à bien leurs travaux. On trouve sur Grid'5000 des outils tels un gestionnaire de ressources, OAR et un processus de déploiement, Kadeploy qui sont nécessaires à la mise en œuvre efficace des applications sur la grille.

Cette plate-forme est extrêmement intéressante car elle ouvre les portes sur la réalisation d'expériences à grande échelle dans un environnement très contrôlé et hautement configurable. Globalement, son fonctionnement est très satisfaisant et prometteur. Néanmoins, elle connaît des perturbations.

L'objectif de la thèse est de définir de nouvelles techniques d'ordonnancement de tâches parallèles qui soient couplées avec le déploiement d'environnements. D'un point de vue plus concret, cela consiste à proposer aux utilisateurs de grappes au niveau de Grid'5000 la possibilité de soumettre au gestionnaire de ressources OAR, des tâches en associant pour chacune d'elles un environnement. Cet environnement peut être un système d'exploitation, une bibliothèque ou des fichiers de données. À partir d'une image système déployée et redémarrée, l'ordonnanceur tentera de maximiser le nombre de tâches s'exécutant avec un environnement sur la grille afin de minimiser le nombre de déploiements qui peuvent avoir un impact fort sur la fiabilité du processus d'exécution des tâches.

Nous avons ainsi étudié ce problème et nous l'avons modélisé sous forme d'un problème d'ordonnancement. Afin de répondre aux exigences des utilisateurs concernant le temps de traitement de leurs tâches, nous avons associé un critère sur le temps de terminaison, le *makespan*. En ce qui concerne le problème de déploiement, nous avons défini un nouveau critère qui comptabilise pour chaque machine (processeur), le nombre de déploiements effectués. Ainsi, on peut définir le nombre total de déploiements dans la *grappe*.

Le caractère $\mathcal{NP}$-difficile de ce problème d'ordonnancement bicritère avec déploiement ne nous permet pas de trouver pour une longueur (la plus petite possible), un ordonnancement qui minimise le nombre de déploiements. Nous avons ainsi définit une nouvelle approche dite «approche budget relaxée» basée sur l'approche budget avec relaxation des contraintes d'optimalité pour les deux critères. Ceci nous a permis de définir un rapport d'approximation (α, β)-budget-approché-relaxé, pour la solution engendrée. Nous présentons un algorithme basé sur cette approche que l'on note par **GLS** pour *Group list scheduling*. Cet algorithme se fait en deux phases. Une première phase consiste à regrouper des tâches utilisant le même environnement sous forme de blocs. Il considère ensuite ces blocs comme des MétaTâches et définit ainsi un temps d'exécution et un nombre de processeurs nécessaires à exécuter toutes les tâches constituant les blocs. Une fois la phase de construction de MétaTâches terminée, vient alors la phase d'ordonnancement. Les MétaTâches formées sont ordonnancées suivant un algorithme de liste sur m processeurs. Ainsi, **GLS** fournit une solution $(4, 2)$-budget-approchée-relaxée pour le problème d'ordonnancement bicritère.

La notion d'optimalité pour un problème multicritère peut prendre différents aspects, créant ainsi parfois une confusion. L'optimalité de Pareto semble aujourd'hui bien admise, capturant ainsi la notion du "meilleur" compromis possible. Nous nous sommes inspirés des travaux de Papadimitriou et Yannakasis [PY00] et nous avons défini un algorithme polynômial qui détermine, à l'aides des solutions produites par **GLS**, un ensemble de solutions qui soient au plus $(4 + \epsilon, 2)$ approchées de la courbe de Pareto.

Pour finir, nous avons testé et comparé **GLS** à des algorithmes de liste. Nous avons implémenté un simulateur de tâches rigides nécessitant un environnement et qui s'exécutent sur une plate-forme parallèle. Nous avons aussi implémenté des heuristiques basées sur les algorithmes de listes afin de les comparer à **GLS**. On constate que **GLS** donne la meilleure performance en ce qui concerne le nombre de déploiements. Par contre, cet algorithme donne de moins bons résultats pour le *makespan*. Nous avons aussi remarqué que les rapports d'approximation calculés au chapitre 3 sont respectés. Effectivement, pour un λ et un D_λ, fixés le ratio pour le déploiement est toujours inférieur à 2.

À l'issue de ces travaux, il reste des pistes à explorer pour rendre l'étude encore plus complète. À court terme, il serait intéressant de faire varier la valeur de λ afin de trouver un meilleur compromis entre le *makespan* et le nombre de déploiements. D'un point de vue des utilisateurs qui soumetentsouvent une seule tâche, il serait

intéressant de considérer à la place du *makespan* un autre critère comme par exemple la moyenne des temps de réponse. On pourrait alors tenter de déterminer un nouveau compromis entre les deux critères.

D'un point de vue théorique, il serait très utile de considérer un modèle en ligne, puisqu'il est plus adapté à l'environnement de *grappes*. On peut prévoir par exemple une pré-installation d'environnements sur des machines dédiées. Néanmoins, l'analyse théorique des modèles en ligne n'est pas encore maîtrisée. Cette étude peut être menée à long terme.

Les systèmes parallèles sont en plein développement et de plus en plus utilisés. Les architectures sont de plus en plus irrégulières et hiérarchiques. Nous pensons que l'approche étudiée dans cette thèse, bien que théorique, pourra s'adapter à un environnement hétérogène. On pourra dans un premier temps considérer les tâches qui nécessitent plusieurs environnements pour leur exécution, et ainsi les déployer au même moment sur les processeurs dédiés.

Bibliographie

[ARSY99] J.A. Aslam, A. Rasala, C. Stein, and N.E. Young. Improved bicriteria existance theorems for scheduling. *SODA*, 1999.

[A.S97] A.Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2), 1997.

[Bag89] U. Bagchi. Simultaneous minimisation of mean and variation of flow time and waiting time in single machine systems. *Operations Research*, 37, 1989.

[BCC+06] R. Bolze, F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S.Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, O. Richard, E. Talbi, and I. Touché. Grid'5000 : A large scale and highly reconfigurable grid experimental testbed. *International Journal of High Performance Computing Applications*, 20(4), 2006.

[BDJ86] J. Blazewicz, M. Drabowski, and J.Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transaction on computers*, 35, 1986.

[CB93] C.L Chen and R.L Bulfin. Complexity of single machine, multi-criteria scheduling problems. *European Journal of Operations Research*, 70, 1993.

[CB94] C.L Chen and R.L Bulfin. Complexity of multiple machines, multi-criteria scheduling problems. *3rd Industrial Engineering Research Conference (IERC'94) Atlanta(USA)*, 1994.

[CCG+05] N. Capit, G.D. Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neuron, and O. Richard. A batch scheduler with high level compenents. *Cluster Computing and Grid CCGrid'2005*, 2005.

[CD73] E.G. Coffman and P.J. Denning. *Operating system theory*. Prentice-Hall, 1973.

[Cha96] *Improved scheduling algorithmes for minsum criteria*, Automata Languages and Programming. Springer, 1996.

[CJK98] T.C.E. Cheng, A. Janiak, and M.Y. Kovalyov. Bicriteria single machine scheduling with resource dependent processing times. *SIAM Journal on Optimization*, 8(2), 1998.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.

[CS02] Y. Colette and P. Siarry. *Optimisation multiobjectif*. Eyrolles, October 2002.

[CW97] C.Stein and J. Wein. On the existence of schedules that are near-optimal for both makespan and total weighted completion time. *Operations Research Letters*, 21, 1997.

[Dat01] The data grid project. http ://eu-datagrid.web.cern.ch/eu-datagrid/, 2001.

[D.F89] D.Fernandez. Allocating modules to processors in a distributed system. *IEEE Transactions On Software Engineering*, SE-15(11), November 1989.

[DL89] J. Du and J.Y.-T. Leung. Complexity of scheduling parallel task systems. *SIJDM : SIAM Journal on Discrete Mathematics*, 2, 1989.

[DS88] P. Dileepan and T. Sen. Bicriterion static scheduling research for single machine. *Omega*, 16(1), 1988.

[Dut04] P-F. Dutot. *Algorithmes d'ordonnancement pour les nouveaux support d'exécution*. PhD thesis, Institut Polytechnique de Grenoble, 2004.

[EBG05] E.Angel, E. Bampis, and L. Gourves. Approximation results for a bicriteria job scheduling problem on a single machine without preemption. *Information processing letters*, 94, 2005.

[EBK01] E.Angel, E. Bampis, and A. Kononov. A fptas for approximatin the unrelated parallel machines schedulin problem with costs. *Prceeding of European Symposium on Algorithms (ESA)*, 2001.

[EBK03] E.Angel, E. Bampis, and A. Kononov. On the approximate tradeoff for bicriteria batching and parallel machine schedulin problems. *Theoretical computer science*, 306, 2003.

[EG00] M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multiobjective combinatoril optimisation. *OR Spektrum*, 22, 2000.

[EHP01] T. Erlebach, H.Kellerer, and U. Pferschy. Approximating multi-objective knapsack problems. *Proceedings of the Seventh International Workshop on Algorithms and Data Structures (WADS)*, 2001.

[Ehr01] M. Ehrgott. Multicriteria optimization. *Lecture notes in Economics and Mathematical System*, 491, 2001.

[E.L81] E.Lloyd. Concurrent tasks system. *Operation Research*, 29, 1981.

[Eyr06] L. Eyraud. *Théorie et pratique de l'ordonnancement d'applications sur les systèmes distribués*. PhD thesis, Institut Polytechniques de Grenoble, 2006.

[FAH89] T.D. Fry, R.D Armstrong, and H.Lewis. A framwork for single machine multiple objective sequencing research. *Omega*, 17(16), 1989.

[FK03] I. Foster and C. Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[GG75] M.R. Garey and R.L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SICOMP : SIAM Journal on Computing*, 4, 1975.

[GHD82] A. Goicoechea, D.R Hansen, and L. Duckstein. *Multiobjective Decision Analysis with Engineering and Business Application*. Wiley, 1982.

[GJ79] M.R. Garey and D. Johnson. Bounds on multiprocessor scheduling with ressource constraints. *SIAM Journal on Computing*, 4, 1979.

[Glo01] Globus alliance. http ://www.globus.org/, 2001.

[Gou05] L. Gourvès. *Approximation polynomiale et optimisation combinatoire multicritère*. PhD thesis, Université d'Évry Val d'Essonne, 2005.

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

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

[Gri06] Grid'5000. http ://www.grid5000.fr, 2006.

[Hoc96] D. Hochbaum. *Approximation Algorithms for NP-hard Problems*. PWS, 1996.

[Hoo92] H. Hoogeveen. *Single Machine Bicriteria Scheduling*. PhD thesis, Eindhoven University of Technology, 1992.

[Hoo05] H. Hoogeveen. Multicriteria scheduling. *European Journal of Operations Research*, 167, 2005.

[HSJ96] L.A. Hall, D.B. Shmoys, and J.Wein. Scheduling to minimize average completion time : Off-line and on-line approximation algorithmes. *Proceeding of the 7th ACM-SIAM Symposium on Discrete Algorithms*, 1996.

[HSSJ97] L.A. Hall, A.S. Schulz, D.B. Shmoys, and J.Wein. Scheduling to minimize average completion time : Off-line and on-line approximation algorithmes. *Mathematics of Operations Research*, 22(3), 1997.

[Jac55] J.R. Jackson. Scheduling a production line to minimize maximum tardiness. Technical report, Mgmt. Sci. UCLA, 1955.

[JEE+96] J.Blazewicz, K.H Ecker, E.Pesch, G. Schmidt, and J. Weglarz. *Scheduling Computer and Manufacturing Processes*. Springer and Verglag, 1996.

[Kad06] Kadeploy. http ://gforge.inria.fr/projects/kadeploy, 2006.

[Leu04] Joseph Y-T Leung. *Handbook of Scheduling*, chapter 25-26. CRC Computer and Information Science Series. Chapman et Hall, 2004.

[LG03] Uri Lublin and Dror G.Feitelson. The workload on parallel supercomputers modeling the characteristics of regid jobs. *J. Parallel and Distributed Comput*, 63(11) :1105–1122, nov 2003.

[LMB96] J.K. Lenstra, M.Veldhorst, , and B.Veltman. The complexity of scheduling trees with communication delays. *Journal of Algorithms*, 20, 1996.

[MD79] M.R.Garey and D.S.Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[MM07] F.K. Moulaï and G. Mounié. Bicriteria scheduling algorithm with deployment in cluster. *The Sixteenth International Heterogeneity in Computing Workshop*, 2007.

[M.P95] M.Pinedo. *Scheduling : Theory, Algorithm, and System*. Prentice Hall, 1995.

[NHH95] A. Negar, J. Haddock, and S.S. Heragu. Multiple and bicriteria scheduling : A literature survey. *European Journal of Operations Research*, 81, 1995.

[NSD93] R.T. Nelson, R.K. Sarin, and R.L. Daniels. Scheduling with multiple performance measures : the one-machine case. *Management Science*, 62, 1993.

[OAR06] Oar. http ://oar.imag.fr/, 2006.

[Par96] V. Pareto. *Cours d'Éconnomie Politique*. Rouge and Cie, 1896.

[PCTW97] C.A. Phillips, C.Stein, E. Torng, and J. Wein. Optimal time critical scheduling via ressource augmentation (extended abstract). *Proceeding of the 29th annual ACM symposium on Theary of Computing*, 1997.

[PY00] C.H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. *Proceeding 41th Annual IEEE Symposium on Foundations of Computer Science*, 2000.

[RCR80] R.Baker, E.G. Coffman, and R.L Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4), 1980.

[Smi56] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3, 1956.

[SO95] H.M. Safer and J.B. Orlin. Fast approximation schemes for multi-criteria combinatorial optimization. 1995.

[ST93] D.B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming A*, 62, 1993.

[Ste86] R. Steuer. *Multiple criteria optimization : theory computation and application.* Wiley, 1986.

[Tal] E. Talbi. Métaheuristique pour l'optimisation combinatoire multi-objectif :état de l'art. Technical report, Laboratoire Fondamentale de Lille.

[TGC02] R. Tadei, A. Grosso, and F. Della Croce. Finding the pareto-optima for the total and maximum tardiness single machine problem. *Discrete Applied Mathematics*, 124, 2002.

[VB01] V.T'Kindt and J-C. Billaut. Multicriteria scheduling problems : a survey. *RAIRO Operations Research*, 35, 2001.

[WG80] L.N. Van Wassenhove and F. Gelders. Solving a bicriterion scheduling problem. *European Journal of Operations Research*, 4, 1980.

Résumé

Nouvelles approches pour l'ordonnancement d'applications parallèles sous contraintes de déploiement d'environnements sur grappes.

Cette thèse s'inscrit dans le cadre des *grappes* dans le projet Grid'5000 (Projet Français pour les grilles). Grid'5000 est une plate-forme expérimentale qui offre la possibilité aux chercheurs de soumettre aux gestionnaires de ressource des programmes (travaux) et d'associer pour chaque requête un environnement. Une *grappe* est un ensemble de nœuds de calcul, connectés entre eux via un réseau dédié. Le processus de déploiement d'environnement sur les nœuds de calcul n'est pas sans conséquence. Un des problèmes que l'on rencontre est la défaillance des machines. Le démarrage excessif lors de de la phase déploiement peut causer un endomagement de celles-ci. Nous avons ainsi modélisé ce problème sous forme d'un problème d'ordonnancement bicritère. Le premier critère à minimiser comptabilise pour chaque machine (processeur) le nombre de déploiements effectués. Il permet ainsi permet de définir le nombre total de déploiements sur toutes les machines. Nous avons également considéré un second critère à minimiser, le *makespan*.

Nous avons défini un algorithme *Groups List Scheduling*, basé sur une approche budget, avec un relâchement des contraintes d'optimalité. Cette approche nous a permis de définir une solution (α, β)-budget-relaxée-approchée pour un problème d'optimisation bicritère. Dans le cadre du problème d'ordonnancement bicritère avec déploiement, l'algorithme **GLS** donne ainsi une solution $(4, 2)$-budget-approchée-relaxée.

Nous avons ensuite abordé ce problème d'ordonnancement bicritère avec déploiement en utilisant l'approche «courbe de Pareto». Nous avons défini un algorithme polynômial, qui permet de construire une courbe de Pareto $(4 + \epsilon, 2)$-approchée, à partir des solutions fournies par l'algorithme **GLS**.

Une analyse expérimentale nous a permis d'évaluer les performances de l'algorithme **GLS** et de valider ainsi les rapports d'approximation trouvés.

Mots-clés : Ordonnancement, optimisation multicritère, déploiement, Grid'5000

Abstract

New approaches for scheduling the parallel applications with the environments deployment constraints on clusters.

This thesis considers the *Clusters* in Grid'5000 (French Project for the grids). Grid'5000 is an experimental platform which makes possible for researchers to submit their programs associating them with an excution environment. Usually, the deployment process on nodes has some problems. One of them is the failure of the machines because the excessive boot of the deployment phases can cause their endomagement. Thus, we consider the bicriteria scheduling problem to solve the scheduling problem with deployment on *cluster*. The first criteria to minimize is the number of deployments of all machines. The second one is to minimize the makespan.

We define an algorithm "Groups List Scheduling" denoted by **GLS**, based on a bujet approach with relaxation of the optimality constraints. Using this approach, we define a (α, β)-budget-relaxed-approximate solution for the bicriteria optimisation problem. When we consider the bicriteria scheduling problem with deployment, the **GLS** algorithm gives a $(4, 2)$-budget-relaxe-approximate solution.

We have defined a polynomial algorithm that allows the construction of a $(4 + \epsilon, 2)$-approximate Pareto curve from the results obtained by the **GLS** algoritm.

In the next step we consider the bicriteria scheduling problem with deployment, using the Pareto curve approach. We define a plynomial algorithm which builds, from GLS algorithm, a $(4 + \epsilon, 2)$-approximate Pareto curve solutions.

An experimental analysis gives the performance of the **GLS** algorithm and allows us to validate the approximation ratios.

Keywords : Scheduling, multicriteria optimisation, deployment, Grid'5000.