Contributions aux techniques d'ordonnancement sur plates-formes parallèles ou distribuées

Lamiel Toch

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Contributions aux techniques d'ordonnancement sur plates-formes parallèles ou distribuées

Lamiel Toch

SPIM

Thèse de Doctorat

UFC

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE FRANCHE-COMTÉ

présentée à L'U.F.R. DES SCIENCES ET TECHNIQUES DE L'UNIVERSITÉ DE FRANCHE-COMTÉ pour obtenir LE GRADE DE DOCTEUR DE L'UNIVERSITÉ DE FRANCHE-COMTÉ
Spécialité : Informatique

Contributions aux techniques d'ordonnancement sur plates-formes parallèles ou distribuées

par **Lamiel Toch**

Soutenue le 9 novembre 2012 devant la Commission d'Examen :

Rapporteurs	Frédéric Desprez	Directeur de recherche à l'INRIA , Laboratoire Informatique du Parallélisme , Ecole Normale Supérieure de Lyon, INRIA Grenoble Rhône-Alpes, Lyon, Grenoble
	Denis Trystram	Professeur, Laboratoire d'Informatique de Grenoble, ENSIMAG, INPG, Grenoble
Membres du jury	Christophe Cérin	Professeur, Laboratoire d'Informatique Paris Nord, Université Paris 13
	Stéphane Chrétien	Maître de conférences, Laboratoire de Mathématiques de Besançon, invité
	Jean-Marc Nicod	Professeur, ENSMM Institut Femto-St, Besançon, co-directeur
	Pierre-Cyrille Héam	Professeur, Université de Franche-Comté, Institut Femto-St, Besançon, président
	Laurent Philippe	Professeur, Université de Franche-Comté, Institut Femto-St, Besançon, directeur

Résumé

Les travaux présentés dans ce document portent sur l'ordonnancement d'applications parallèles sur des plates-formes parallèles (cluster) ou distribuées (grilles de calcul). Dans nos travaux de recherche nous nous sommes concentrés sur l'ordonnancement d'applications modélisées par un DAG, graphe orienté sans cycle, pour les grilles de calcul et sur l'ordonnancement pour les (cluster, machines multiprocesseurs) de programmes parallèles (jobs parallèles) représentés sous la forme de surface rectangulaire dont les deux dimensions sont le nombre de processeurs requis et la durée d'exécution. Les recherches s'articulent autour de trois grands axes. Le premier axe concerne l'ordonnancement d'un ensemble d'instances d'une application pour les grilles de calcul. Le deuxième axe est l'ordonnancement de jobs parallèles dans les clusters. Le troisième est l'ordonnancement d'un lot de jobs parallèles pour les machines parallèles. Cette thèse apporte des contributions sur les trois axes. La première contribution associée au premier axe est l'étude expérimentale avancée de trois algorithmes pour l'ordonnancement d'un ensemble d'instances d'une application sur une plate-forme hétérogène où les coûts de communication sont négligeables : un algorithme de liste, un algorithme de régime permanent et un algorithme génétique. D'autre part nous apportons l'intégration des communications dans cet algorithme génétique. La deuxième contribution associée au deuxième axe est la conception d'une nouvelle technique d'ordonnancement de jobs parallèles pour les clusters : le pliage de jobs qui utilise la virtualisation des processeurs. La dernière contribution porte sur la conception d'une nouvelle technique inspirée du domaine des statistiques et du traitement du signal appliquée à l'ordonnancement de jobs parallèles dans une machine multiprocesseur. Enfin nous donnons quelques travaux de recherches qui on été réalisés mais qui n'ont pas abouti à des résultats significatifs pour l'ordonnancement.

Mots clés : Ordonnancement, graphes de tâches, job parallèle, lots de graphes de tâches, grilles de calcul, cluster, machine multiprocesseur.

Abstract

Works presented in this document tackle scheduling of parallel applications in either parallel (cluster) or distributed (computing grid) platforms. In our researches we were concentrated on either scheduling of applications modeled by a DAG, directed acyclic graph, for computing grid or scheduling of parallel programs (parallel jobs) represented by a rectangular shape whose the two dimensions are the number of requested processors and the execution time. The researches follow three main topics. The first topic concerns the scheduling of a set of instances of an application for computing grid. The second topic deals with the scheduling of parallel jobs in clusters. The third one tackles the scheduling of parallel jobs in multiprocessor machines. We brought contributions on these three topics. The first contribution under the first topic consists of the advanced experimental study of three algorithms for scheduling a set of instances of an application on a heterogeneous platform without communication costs : a list-based algorithm, a steady-state algorithm and genetic algorithm. Moreover we integrate communications in this genetic algorithm. The second contribution under the second topic is the design of a new technique for scheduling parallel jobs in clusters : job folding which uses virtualization of processors. The third contribution deals with a new technique which comes from statistics and signal pro-

cessing applied to scheduling of parallel jobs in a multiprocessor machine. Eventually we give some works that we carried out but which did not give significant results for scheduling.

Keywords : Scheduling, task graphs, parallel job, batches of task graphs, computing grid, cluster, multiprocessor machine.

Table des matières

Introduction

La pensée humaine est une activité complexe qui englobe le raisonnement, la perception, les sensations, les souvenirs, les idées et l'imagination. Mais pour arriver à un résultat ou atteindre un but, l'être humain se confronte à des problèmes qu'il est amené à résoudre. Ces problèmes peuvent être résolus en partie ou totalement par le calcul.

Nos ancêtres très lointains utilisaient leurs dix doigts pour compter. Cependant ce moyen ne permet pas de calculer ou de manipuler de grands nombres. C'est en Mésopotamie qu'il y a plus de 4000 ans, a été inventé une numération sexagésimale. À Nippur les instituteurs enseignaient aux écoliers le calcul avec des tables de multiplications, d'inverses, de carrés et de racines carrées et cubiques. Puis différents moyens matériels ont été utilisés afin de calculer plus vite et ceux-ci ont évolué au fil des siècles : bouliers chinois, bâtons de Neper, Pascaline, tables logarithmiques, tables trigonométriques, et d'autres. Jusqu'à ce qu'en 1945, l'armée réalise une machine électrique à base d'ampoules pour calculer quelques milliers de trajectoires de missiles : il s'agit de l'ENIAC (*Electronic Numerator Integrator Analyser Computer*). Néanmoins l'ENIAC ne possédait pas de mémoire et à chaque exécution d'un programme il fallait reconfigurer la machine. En 1949, a été créé le premier ordinateur par Max Newmann : le Manchester Mark 1. L'invention, puis la démocratisation du transistor a marqué un pas dans cette évolution. Ainsi les premiers superordinateurs ont vu le jour à la fin des années 1970 lorsque la société Cray Research Inc a équipé le laboratoire d'énergie "Los Alamos National Laboratory" (LANL) d'un Cray-1. Il s'agissait d'une machine qui réalisait des opérations sur des vecteurs. Au fil du temps le transistor est devenu de plus en plus petit, si bien que les premiers microprocesseurs ont vu le jour dans les années 1970, conçus par la société Intel. Commença alors la course à la miniaturisation et à la vitesse conduisant à la démocratisation des ordinateurs personnels.

Les besoins des scientifiques et ingénieurs en ressources de calcul et en vitesse ne cessent de croître. Ces besoins poussent les industriels à concevoir des superordinateurs de plus en plus puissants. Les scientifiques réalisent des calculs numériques pour simuler des phénomènes physiques et utilisent des superordinateurs pour obtenir des résultats. Par exemple au LANL, les scientifiques réalisent des simulations en physique nucléaire. On peut classer les architectures des superordinateurs en trois grandes catégories. La première est l'architecture vectorielle, comme le Cray-1 qui en un cycle d'horloge est capable d'appliquer une opération sur l'ensemble des valeurs d'un vecteur. La seconde est l'architecture massivement parallèle : elle se compose d'une multitude de processeurs peu puissants qui appliquent une même opération sur un grand nombre de données avec les mêmes paramètres. Nous pouvons citer la Connection Machine CM-2 de la société Thinking Machines ou nos actuelles cartes graphiques. La dernière architecture qui a le vent en poupe ces dernières années est l'architecture en grappes (ou *cluster*). Un cluster

est constitué d'ordinateurs interconnectés par un réseau très rapide. On peut citer la machine Sequoia d'IBM pour la National Nuclear Security Administration (NNSA) : en juin 2012 ce cluster est composé de 98 304 machines interconnectées, de 1.6 millions de cœurs et dispose de 1.6 pétaoctets de mémoire vive.

D'autres architectures permettent de réaliser des calculs hautement parallèles : les processeurs graphiques (GPU : *Graphics Processing Unit*). Ces processeurs sont constitués de centaines de cœurs et sont potentiellement aptes à délivrer une grande puissance de calcul. À l'origine conçus pour effectuer des rendus graphiques et des traitements vidéos, il n'était pas simple pour les scientifiques d'exécuter du code parallèle général sur des GPUs (GPGPU : General Purpose computing on GPU) car ils devaient passer par des bibliothèques de rendu graphique comme OpenGL. Mais depuis une dizaine d'années, les constructeurs de GPUs ont facilité le développement d'applications parallèles sur ces GPUs en modifiant leur architecture et en proposant à la communauté scientifique un modèle général de programmation parallèle comme OpenCL ou CUDA de nVidia. Les gains de performances des GPUs par rapport aux CPUs peuvent être très importants comme le montre le travail de Suchard et al. [84]. Si les clusters de CPUs sont très courants dans les centres de calcul, il n'est pas rare d'y trouver des cluster de GPUs.

D'autre part avec l'explosion de l'internet, de plus en plus de ressources de calcul (téléphones portables, ordinateurs, superordinateurs) peuvent se connecter au réseau mondial. Toutes ces ressources sont potentiellement aptes à délivrer une puissance de calcul pour exécuter des applications dont les sous-tâches sont distribuées à ces ressources. Ce sont des applications dites distribuées. Cette infrastructure composée de ressources de calcul différentes est appelée grille de calcul. En fait, le concept de la grille est apparue à la fin des années 1990 aux États-Unis, quand les informaticiens commençaient à interconnecter les grands centres de calcul. À ce moment là le projet I-WAY a été lancé; il consistait à interconnecter des ressources de calcul à plus large échelle. Ce projet intègre un intergiciel, logiciel qui vise à faire abstraction de l'hétérogénéité de la grille vis à vis de l'utilisateur. Par la suite, Ian Foster et Carl Kesselman se sont inspirés du projet I-WAY pour réaliser un intergiciel pour les grilles de calcul : Globus [48]. Grâce à cet intergiciel, l'utilisateur exécute son programme sur la grille comme si l'ensemble des ressources n'étaient regroupées qu'en un seul lieu.

Les différentes sous-tâches d'un programme sur une grille peuvent être interconnectées entre elles et cette construction est modélisée par un DAG (*Directed Acyclic Graph*), un graphe orienté sans cycle. Très souvent un tel programme est lancé en plusieurs exemplaires en parallèle sur des données différentes. C'est ce que l'on appelle une instance du programme. Nous utilisons le terme *workflow* pour désigner un ensemble d'instances d'un ou plusieurs programmes décrits par un DAG.

Afin d'optimiser l'exécution d'un programme de l'utilisateur, il est nécessaire de calculer les meilleurs placements des sous-tâches du programme sur les différents nœuds de la grille, ainsi que les meilleures routes à prendre pour l'acheminement des données d'une tâche à l'autre. Il s'agit d'un problème d'ordonnancement. Les problèmes d'ordonnancement sont également présents dans le contexte des clusters de calcul. En effet, quand un cluster est partagé entre plusieurs utilisateurs, les ressources n'étant pas infinies, il faut placer les programmes des utilisateurs sur les ressources disponibles afin d'optimiser l'utilisation de la machine.

Toutes les ressources de calcul que nous venons d'aborder sont des éléments clés pour le

calcul hautes performances (HPC : *High Performance Computing*) qui servent aux scientifiques désireux de réaliser des simulations de grande taille. En effet, plus les ressources de calcul sont performantes et plus les scientifiques obtiennent rapidement des résultats. Dans cette course à la vitesse, la partie logicielle qui comporte un ordonnanceur joue un rôle déterminant car c'est elle qui optimise l'utilisation des ressources. Le HPC est une solution à un problème de calcul intensif qui comporte le matériel pour l'exécution et le logiciel pour l'optimisation. L'ordonnancement joue ici un rôle d'optimisation.

Les travaux présentés ici participe à la recherche de l'équipe CARTOON du Département Informatique des Systèmes Complexes de l'Institut FEMTO-ST à Besançon. Ils font suite aux travaux de l'équipe dans le domaine de l'ordonnancement d'un workflow sur des grilles de calcul. En cours de thèse, le Mésocentre de Franche-Comté a été créé avec l'installation d'un cluster dans le bâtiment hébergeant la plupart des chercheurs en informatique de l'Université de Franche-Comté. Nous nous sommes intéressés aux problématiques d'ordonnancement de jobs (travaux) sur les clusters. Dans ce contexte, comme notre cluster offre seulement un millier de cœurs et comme les demandes en ressources de calcul sont importantes, nous avons proposé une technique visant à remplir les cycles inutilisés des cœurs de calcul. Nous appelons cette technique le *pliage de jobs*. Elle utilise la virtualisation des machines pour créer des cœurs virtuels pour faire croire aux jobs qu'ils tournent réellement sur le nombre de cœurs demandé au départ alors qu'ils tournent en réalité sur un nombre inférieur de processeurs physiques. Pour aller plus loin, nous avons eu besoin d'outils mathématiques puissants pour optimiser les ordonnancements d'un lot de jobs sur une machine multiprocesseurs. C'est pourquoi nous avons collaboré avec le Laboratoire de Mathématiques de Besançon pour adapter les techniques d'optimisation à nos problèmes d'ordonnancement et proposer une approche originale à ce problème.

Durant cette thèse, nous nous sommes intéressés aux problèmes d'ordonnancement dans le contexte des clusters et des grilles de calcul. L'ordonnancement peut être considéré soit en tant que moyen pour planifier les tâches toutes définies à un instant donné et que l'on doit affecter à des ressources, soit en tant que moyen pour organiser des tâches qui arrivent au cours du temps et les affecter à des ressources. Pour les deux aspects de la problématique de l'ordonnancement, l'objectif est d'optimiser un critère. Dans notre contexte, les ressources sont des plates-formes parallèles (clusters) ou distribuées (grilles de calcul), et les tâches sont calculatoires. La théorie de l'ordonnancement est très vaste, c'est pourquoi notre problématique se définit simplement à partir des motivations qui ont conduit à nos travaux de recherche. La problématique est la suivante : sur une grille de calcul, comment planifier au mieux l'exécution d'un lot de travaux ($j$obs) décrites par un DAG appliqués à un jeu de données différentes ? Dans un cluster partagé entre plusieurs utilisateurs, comment organiser au mieux le partage des unités de calcul pour l'exécution de leurs applications ? Dans un cluster ou une machine multiprocesseurs, comment planifier au mieux un lot d'applications d'un utilisateur pour les terminer au plus tôt ? Mais avant de répondre à chacune de ces questions, il est nécessaire de définir un critère à optimiser pour chacun des problèmes.

Pour répondre à cette problématique, en premier lieu (partie I), nous présentons le contexte dans lequel s'inscrit notre travail : l'ordonnancement pour le calcul hautes performances. Nous définissons plus exactement ce qu'est l'ordonnancement et nous donnons un état de l'art de l'ordonnancement pour le calcul haute performance. Nos contributions portent, comme nous venons de le voir, sur trois problèmes d'ordonnancement. Ainsi chaque problème fait l'objet

d'un chapitre.

Dans le premier chapitre de la partie dédiée aux contributions (chapitre 3), nous abordons un problème d'ordonnancement d'un workflow sur des grilles de calcul. Nous évaluons d'abord trois algorithmes sans tenir compte des coûts de communication inhérents aux grilles de calcul. Puis, nous proposons des modifications d'un des trois algorithmes, un algorithme génétique, afin d'intégrer le coût des communications. Nous comparons ensuite les trois algorithmes.

Dans un deuxième chapitre des contributions (chapitre 4), nous abordons le problème d'ordonnancement de jobs parallèles dans le contexte des clusters. Dans ce chapitre nous proposons une nouvelle technique d'ordonnancement pour les cluster avec le *pliage de jobs* associée à la virtualisation des processeurs. Cette technique crée des cœurs virtuels sur des cœurs physiques. Les travaux ou *jobs* s'exécutent de manière logique sur autant de cœurs virtuels que demandés alors que le nombre de cœurs physiques alloués est plus petit. L'objectif de cette technique est de remplir les cycles inutilisés des cœurs physiques. Une étude expérimentale nous a permis d'évaluer les gains possibles apportés par cette technique.

Le dernier chapitre 5 traite de recherches pour l'ordonnancement d'un lot de jobs indépendants sur un cluster ou une machine multiprocesseurs. Les jobs rigides et moldables sont traités. Le but est alors de minimiser le temps de complétion (*makespan*) de l'ordonnancement, c'est à dire la durée entre la date de début de l'exécution de la première tâche et la date de fin d'exécution de la dernière tâche. Ce problème a largement été abordé dans la littérature. Il est connu pour être NP-Difficile et différentes heuristiques ont déjà été proposées. Nous abordons ce problème sous un angle différent de l'approche habituelle. Ce que nous proposons repose sur des approximations linéaires successives du problème posé cette fois sous la forme d'un problème creux. L'idée est de relaxer la résolution d'un programme linéaire en nombres entiers du problème ainsi formulé et d'utiliser une linéarisation de la norme ℓ_p comme la fonction objectif dans le but de forcer les valeurs de la solution à tendre vers des nombres entiers. Nous comparons notre approche avec l'algorithme de liste classique LTF (*Largest Task First*). Nous avons conçu différents algorithmes reposant sur des approximations linéaires successives problème formulé ainsi. La contribution de ces travaux tient en la conception de ces algorithmes et l'intégration d'une méthode mathématique dans l'ordonnancement. En fin de chapitre, nous synthétisons des recherches menées sur la base d'outils mathématiques tels que la résolution de programmes linéaires, quadratiques, non linéaires.

Première partie

Présentation du contexte

Chapitre 1

Les ressources de calcul et leur utilisation

Quand les scientifiques sont amenés à réaliser des calculs intensifs, fastidieux et longs, ils ont recours à des ressources de calcul performantes : ordinateurs, clusters ou grilles de calcul. Tout ce qui a une relation avec les calculs intensifs fait partie du domaine du *HPC*. Notre travail sur l'ordonnancement apporte des contributions pour le HPC puisque nous manipulons des modèles d'applications et de plates-formes propres à ce domaine de l'informatique. Nous présentons dans ce chapitre, le contexte sur les différentes ressources de calcul afin de positionner et de donner une visibilité à notre travail.

1.1 Les différentes ressources de calcul

Nous présentons ici trois grands types de ressources de calcul : les fermes de calcul ou clusters, les grilles de calcul et les clouds dont le paradigme sous-jacent est l'informatique en nuage.

1.1.1 Les clusters ou fermes de calcul

Les scientifiques ont besoin d'effectuer des calculs de plus en plus massifs dans leur travail de recherche ; c'est ce qui a poussé le développement d'ordinateurs toujours plus puissants : les superordinateurs. Dans les deux dernières décennies, afin de satisfaire cette demande en superordinateurs, les industriels ont regroupé des ordinateurs afin d'accroître les capacités de calcul : ce sont les fermes de calcul (cluster). Plutôt que fabriquer un superordinateur avec un processeur très puissant, les industriels préfèrent concevoir des ordinateurs sur la base de processeurs existants afin de constituer une ferme de calcul. Avec ce procédé de tels superordinateurs reviennent moins cher à fabriquer.

Une ferme de calcul, aussi appelée grappe de serveurs ou cluster, est constituée de différents serveurs composés d'un ou de plusieurs processeurs. Ces serveurs sont connectés au moyen de réseaux de communication très rapides (par exemple Infiniband). L'utilisateur se connecte à la ferme de calcul sur un serveur frontal et exécute son application parallèle qui est conçue pour s'exécuter en parallèle sur un ensemble de processeurs. Généralement les serveurs qui

composent une ferme de calcul sont tous similaires : ce sont des plates-formes homogènes. De plus, les serveurs d'une ferme de calcul sont physiquement localisés au même endroit.

De nos jours les fermes de calcul sont très répandues dans le monde scientifique, dans les universités, les centres de recherche pour les calculs hautes performances (HPC). Elles sont utilisées pour effectuer des simulations de tout genre : simulations physiques (aérodynamique, nucléaire, et autres), prévisions météorologiques (superordinateur K de Fujitsu), application à la médecine ou encore à la génétique (Blue Gene). Pour avoir un ordre d'idée des plus grands clusters de calcul nous pouvons nous référer au site web TOP500 [4] : en juin 2012, la première place était détenue par la machine IBM Sequoia (Blue Gene/Q) qui peut atteindre 20 pétaflops en puissance de calcul. Cette machine est composée de 1 572 864 cœurs de processeurs Power BQC. La deuxième place était occupée par le superordinateur K de Fujitsu. Cette machine peut atteindre une puissance de calcul de 11 pétaflops et est composée de 705 024 cœurs de processeurs Sparc64.

1.1.2 Les grilles informatiques

Une grille informatique ou grille de calcul (grid computing) peut être considérée comme un superordinateur virtuel qui est physiquement constitué de plusieurs machines différentes et délocalisées. Cette infrastructure est dite hétérogène car les éléments qui la constituent ont des vitesses de calcul différentes et des capacités de stockage différentes ; les éléments peuvent aller d'une ferme de calcul à un ordinateur personnel. Ces machines sont interconnectées par des réseaux de communication beaucoup moins rapides que ceux d'une ferme de calcul. Elles sont par exemple interconnectées au moyen de câbles ethernet ou du wifi. Les grilles sont soit privées ou soit publiques et dans ce cas elles sont accessibles à tous depuis internet.

Les grilles informatiques peuvent être classées dans trois grandes familles :

- les grilles pair-à-pair (peer-to-peer grids ou desktop grids) sont constituées d'une multitude de machines de petite taille, comme des ordinateurs personnels. Le problème à résoudre est découpé en sous problèmes résolus indépendamment sur les machines, avant qu'une machine maître ne récolte les différents résultats.

- les grilles de ressources sont constituées de machines de plus grande taille, telles que des fermes de calcul, comme Grid'5000[1].

- les grilles de services proposent aux utilisateurs des applications (de calcul scientifique par exemple) déployées sur la grille. Ce sont des architectures orientées services (*Service Oriented Architecture* ou SOA). Par exemple l'utilisateur fait la demande d'une application à la grille. En cas de disponibilité l'utilisateur fournit ses données à l'application puis récupère ses résultats de manière transparente sans se soucier, ni de la localisation du service, ni de la manière dont le service est réalisé.

L'apparition des grilles de calcul a permis d'analyser et d'effectuer des calculs intensifs sur d'immenses quantités de données. Ces données sont générées lors d'acquisitions par les grands appareils de mesures utilisés pour des expériences scientifiques. Par exemple l'accélérateur de

[1] `https://www.grid5000.fr/`

particules du LHC (Large Hadron Collider) du CERN génère plus de 25 pétaoctets[2] par an. Les scientifiques qui utilisent ces techniques de calculs intensifs sur les grilles font de l'e-science [86]. De nos jours l'e-science se développe de plus en plus : les grilles de calcul sont utilisées par exemple en traitement numérique d'images médicales [63, 77, 49], en géosciences [30], en astronomie [78] et dans d'autres domaines.

1.1.3 L'informatique en nuage

L'informatique en nuage (cloud computing) est un paradigme qui propose à l'utilisateur de délocaliser ses ressources de calcul et ses ressources de stockage. Les ressources qui servent de support au cloud computing sont généralement des clusters car ceux-ci sont plus homogènes et disponibles que les grilles de calcul. À la manière d'un réseau électrique, les entreprises paient la consommation réelle des services proposés par le cloud du fournisseur qui leur garantit une qualité de calcul et de stockage. Avec ce concept, elles n'ont plus besoin de se préoccuper de l'achat de serveurs de calcul et de stockage ou de l'organisation des infrastructures réseaux. Nous ne développons pas davantage le concept de l'informatique en nuage car notre travail n'apporte aucune contribution dans ce domaine. Nous disons juste que ce concept est en pleine expansion et qu'il peut être utilisé pour réaliser des calculs intensifs.

1.2 Des Problèmes

Nous abordons à présent des problématiques importantes liées à deux grands types de ressources de calcul : les fermes de calcul et les grilles.

1.2.1 Problématiques liées aux fermes de calcul

Qui dit ressources de calcul dit aussi nécessité d'une bonne utilisation de celles-ci. En effet, les ressources sont néanmoins en quantité finie alors que les programmes qui s'exécutent sont de plus en plus gourmands en calcul. De plus les utilisateurs peuvent être nombreux à se connecter en même temps à un même superordinateur et demander pour leur besoin une partie des ressources de calcul disponibles. Il se dégage le problème de la bonne utilisation de la ferme de calcul. En effet les statistiques d'utilisation des centres de calcul montrent que les machines présentent des cycles inutilisés (*idle-time*). Des recherches dans ce domaine ont été menées afin d'optimiser à la fois le placement des programmes des différents utilisateurs sur les ressources de calcul et leur utilisation tout en satisfaisant chaque utilisateur : l'ordonnancement est le terme qui englobe ces problèmes d'optimisation. Prenons un exemple : supposons qu'une machine possède 100 processeurs, qu'un programme parallèle A arrive à la date $t = 0$ et demande 60 processeurs et dure 1 heure. Un autre programme parallèle B arrive à la date $t = 30$ minutes et demande 50 processeurs et dure 10 minutes. S'il fonctionne de manière très basique, le logiciel qui s'occupe du placement des tâches sur les processeurs, appelé ordonnanceur de tâches, choisit d'exécuter la tâche A et au bout de 60 minutes, comme 60 processeurs sont libérés, l'ordonnanceur exécute la tâche B. Le problème est que pendant 60 minutes (entre $t = 0$ et $t = 60$ minutes) la machine présente des cycles inutilisés dans 40 processeurs, puisqu'elle attend que la tâche A libère des

[2] `http://wlcg.web.cern.ch/`

processeurs pour que la tâche B puisse s'exécuter. Ce sont ces problèmes d'optimisation des ordonnancements des tâches qui nous intéressent.

1.2.2 Problématiques liées aux grilles informatiques

Dans une ferme de calcul, les machines sont généralement homogènes (vitesses de calcul), leur nombre est connu à l'avance, et elles sont interconnectées par des réseaux de communication très rapides. Des outils comme MPI (*Message Passing Interface*) ou OpenMP (*Open Multi-processing*), permettent de tirer profit de l'homogénéité des machines et des vitesses de communication entre machines. Grâce à l'homogénéité des machines, les applications parallèles exécutées dans les fermes de calcul ne rencontrent pas le problème d'un goulet d'étranglement lié à une machine plus lente que les autres. De plus grâce au réseau très rapide des fermes de calcul, les données peuvent transiter rapidement d'une machine à l'autre.

Sur une grille, une application qui exploite la puissance de calcul des nombreuses ressources est dite distribuée. Pour maintenir de bonnes performances, exécuter une application distribuée sur une grille demande plus de travail en amont, qu'exécuter une application parallèle sur une ferme de calcul. En effet, les communications entre machines peuvent être plus lentes que dans une ferme de calcul ; les machines sont hétérogènes et ont des vitesses de calcul différentes ; les capacités de mémoire vive sont différentes et il se peut qu'une bibliothèque nécessaire à l'exécution d'un programme ne soit pas installée sur toutes les ressources de calcul.

Les applications distribuées sont constituées de multitudes de tâches à exécuter dans un certain ordre avec des dépendances entre les tâches. Il est primordial d'utiliser au mieux les ressources. Placer quelles tâches sur quelles machines et déterminer à quelle date elles commencent, afin d'obtenir une vitesse d'exécution la plus grande pour une application distribuée est tout l'enjeu des problèmes d'ordonnancement dans ce contexte. Ainsi, si on ordonnance mal les tâches d'une application distribuée, une machine de faible vitesse peut globalement ralentir toute l'exécution.

1.3 Jobs et fermes de calcul

Les fermes de calcul coûtent cher, par conséquent elles sont en nombre limité. Elles sont généralement partagées entre plusieurs utilisateurs. Nous abordons dans cette section les techniques existantes utilisées pour gérer ce partage. Souvent, dans les centres de calcul comme le Centre de Calcul de l'Institut National de Physique Nucléaire et de Physique des Particules (CC-IN2P3), une seule ferme de calcul est installée et partagée entre plusieurs utilisateurs, des scientifiques par exemple. La ferme de calcul est mutualisée.

Afin d'utiliser les ressources disponibles, les scientifiques et les chercheurs soumettent leurs tâches, appelées jobs, au cluster par l'intermédiaire d'un ordonnanceur de jobs. Ce logiciel décide quel job s'exécute à quel moment selon sa priorité. En tant qu'utilisateur, chacun aimerait que son job s'exécute si possible au plus tôt. Cependant les ressource ne sont pas infinies, mais limitées. Il faut donc partager équitablement les ressources entre les jobs des différents utilisateurs. Ce pose alors le problème de l'instauration d'une politique d'équité (*fairshare policy*). Généralement les logiciels mettant en œuvre ces politiques analysent les traces d'utilisation des processeurs afin d'ajuster le comportement de l'ordonnanceur de jobs. De nombreuses poli-

tiques d'équité sont implémentées dans les ordonnanceurs de jobs comme dans l'ordonnanceur Maui [60]. Maui offre la possibilité de choisir sa politique d'équité et de la configurer. Nous pouvons citer la politique d'équité qui s'appuie sur le taux d'utilisation exprimé en processeurs-heures.

Quand un utilisateur soumet un job au cluster, il précise ses besoins en termes de nombre de processeurs, de quantité de mémoire, et il indique le temps limite pour l'exécution de son job. Sur la base de ces données et en tenant compte de ses politiques d'ordonnancement, l'ordonnanceur calcule la priorité du job qui déterminera son ordre de passage. De nombreux algorithmes d'ordonnancement apparaissent dans la littérature, comme l'algorithme *FCFS* (*First Come First Served* : premier arrivé premier servi), implémenté dans le logiciel IBM LoadLeveler [62] ou la technique du *Backfilling* utilisée dans les ordonnanceurs OAR [20] et PBS [55] (*Portable Batch Scheduler*). Certains chercheurs ont combiné l'ordonnanceur de tâches Maui et PBS pour obtenir de bonnes performances [13]. Ces algorithmes sont qualifiées de *online* : l'ordonnanceur ne connaît pas à l'avance la date d'arrivée des jobs. Bien que l'utilisateur donne un temps d'exécution pour son job, ce temps n'est qu'une borne supérieure au bout duquel l'ordonnanceur l'interrompt. Le temps d'exécution donné par l'utilisateur est loin d'être exact. Dans ce cas l'algorithme d'ordonnancement est dit non clairvoyant, car l'ordonnanceur ne connaît pas la durée exacte des jobs. En revanche un algorithme est dit clairvoyant s'il peut estimer à l'avance la durée de tous les jobs.

Nous nous apercevons dans le monde des clusters que leur usage n'est pas optimal car ordonnancer de manière optimale des jobs parallèles est très complexe. Dans les centres de calcul, il n'est pas rare que des clusters présentent des cycles inutilisés.

Pour éviter des cycles inutilisés des processeurs et pour optimiser leur utilisation, les problèmes d'ordonnancement de jobs parallèles ont largement été étudiés. Dans la littérature on distingue trois types de jobs parallèles. Les jobs rigides [70] sont exécutés avec le nombre de processeurs requis. Les jobs *moldables* dont le modèle a été introduit par Turek et al. dans [89], peuvent s'exécuter avec un nombre différent de processeurs que celui prévu à la soumission. Celui-ci ne peut pas changer en cours exécution. Les jobs *malléables* [75, 39] peuvent modifier le nombre de processeurs qui leur sont alloués au cours de leur exécution. Le modèle des jobs rigides peut facilement être utilisé dans la plupart des cas pour ordonnancer des jobs parallèles. Les deux autres modèles supposent une interaction entre l'ordonnanceur et le programme parallèle.

1.4 Workflows et grilles de calcul

Afin d'automatiser les traitements et gagner du temps, les scientifiques ont besoin de rassembler leurs tâches logicielles et de les connecter ensemble pour former une application. À l'entrée de l'application, des données commencent à être traitées par les tâches qui n'ont pas de dépendance, puis celles-ci transmettent les résultats aux suivantes. Généralement la structure d'une telle application se présente sous la forme d'un *DAG* (*Directed Acyclic Graph* : graphe orienté sans cycle) : un graphe dont chaque sommet est une application et chaque arc est une contrainte de dépendance. Une dépendance modélise également une communication entre deux tâches.

Ceci nous amène à la notion de *workflow*. Dans la littérature la notion de workflow est généralement définie comme une structure abstraite d'une application [86, 66, 91] décrite par un DAG. D'autres modèles pour décrire les workflows existent depuis quelques décennies. Les réseaux de Petri[3] sont utilisés pour modéliser les workflows. L'article [5] discute de l'utilisation de réseaux de Petri dans le contexte des workflows. Quel que soit le modèle de workflow, les travaux de recherche distinguent l'exécution d'une application munie d'une structure workflow qui réalise un traitement sur des données d'entrée et l'exécution en parallèle de plusieurs instances d'une application munie une structure workflow qui réalisent un traitement sur un ensemble de données d'entrée. Afin de simplifier l'emploi du terme *workflow* dans notre contexte de travail, nous donnons une autre définition du terme *workflow*. Pour nous, un worfklow est un ensemble d'instances d'une ou plusieurs applications décrites par un DAG.

Exécuter des workflows peut prendre énormément de temps. C'est pourquoi quand la taille des données devient très importante ou quand les calculs deviennent extrêmement intensifs, il est nécessaire d'utiliser des plates-formes distribuées comme des grilles de calcul, afin d'obtenir des gains de temps. L'hétérogénéité des plates-formes d'exécution rendent difficile le déploiement d'applications distribuées sur ces grilles. C'est pourquoi des techniques sont mises en œuvre afin de faciliter le travail des utilisateurs non experts en informatique.

Une technique commode pour faciliter le travail des scientifiques est d'utiliser les architectures orientées services (SOA : en anglais *Service Oriented Architecture*). Les grilles qui intègrent l'architecture SOA mettent en œuvre des services : chaque nœud de la grille propose un ensemble de fonctions. Pour l'utilisateur scientifique, les fonctions sont appelées de manière transparente, il n'a pas besoin de savoir où les fonctions sont localisées, il invoque sa fonction à distance, envoie ses données d'entrée et reçoit le résultat. Ces architectures sont disponibles à travers des logiciels comme DIET [21] ou NINF-G [85].

Le cycle de vie du processus des workflows est divisé en quatre parties [86] : la composition du DAG, c'est-à-dire, l'écriture du programme sous forme de DAG, le *placement* (l'ordonnancement des tâches sur les ressources), l'exécution et la gestion des méta-données (les traces par exemple). Ce qui nous intéresse est l'étape du placement : l'ordonnancement. Avec le développement des grilles de calcul et de leur hétérogénéité, l'ordonnancement est un point crucial pour obtenir de bonnes performances et gagner du temps [11]. Bien ordonnancer un workflow sur une grille de calcul, c'est optimiser l'allocation des différentes tâches sur les différentes ressources disponibles.

Des ordonnanceurs de workflows existent, comme Condor, présenté en détail dans [68] et [87], qui a été conçu pour exploiter la puissance de calcul des ordinateurs de bureau connectés à internet pendant leurs temps de cycle inutilisés. Afin d'exploiter la puissance des grilles de calcul, le module Condor-G[4] a été créé. L'utilisateur envoie son fichier de soumission à Condor-G lorsqu'il demande une exécution d'une instance de son workflow sur la grille. Le logiciel DAGMan (*Directed Acyclic Graph Manager*) est très utilisé avec Condor et est chargé d'exécuter l'instance du workflow modélisé par un DAG. D'autres ordonnanceurs de workflows existent comme le logiciel Pegasus [29] ou GradSoft [73].

En résumé, les techniques visant à faciliter le travail des scientifiques sont : la modélisation de leurs applications sous la forme de DAGs, une grille de calcul où chaque nœud propose des

[3] Le modèle des réseaux de Petri a été introduit par Carl Adam Petri en 1962, au cours de son doctorat.
[4] `research.cs.wisc.edu/condor/condorg/`

services (SOA), des outils (ordonnanceurs de workflows) pour placer les différentes tâches des instances des applications sur les nœuds adéquats de la grille au bon moment.

Chapitre 2

Les problèmes d'ordonnancement

Les problèmes d'ordonnancement peuvent être classés en deux grandes catégories :

1. les problèmes d'ordonnancement en ligne (*online*) [92, 44] pour lesquels la date d'arrivée (*release date*) des jobs n'est pas connue à l'avance ;
2. les problèmes d'ordonnancement hors ligne (*off-line*) pour lesquels les dates d'arrivées des jobs (généralement ils sont tous prêts à $t = 0$ et toutes leurs caractéristiques sont connues avant l'ordonnancement. Ces problèmes ont été très largement étudiés pour les jobs séquentiels [50] et pour les jobs parallèles [37, 43].

Les problèmes d'ordonnancement online sont généralement plus difficiles que les problèmes off-line puisque nous ne connaissons qu'une partie des données du problème. En effet, les décisions prises pour le placement ou l'exécution de tâches ne tiennent pas compte des données manquantes car on ne peut pas prévoir l'avenir. Jiri Sgall a mené une étude exhaustive sur les algorithmes d'ordonnancement online [79].

2.1 Les critères d'optimisation

Quand nous concevons un algorithme d'ordonnancement pour un problème particulier, nous cherchons à optimiser un certain critère. Comme nous allons le voir, ce critère dépend du problème à traiter et il n'existe pas pour tous les problèmes d'ordonnancement un critère d'optimisation universel. Néanmoins nous pouvons donner quelques critères d'optimisation fréquemment utilisés.

Pour les problèmes off-line d'ordonnancement d'une collection de jobs ou d'un graphe de tâches dont les propriétés sont connues à l'avance, un critère d'optimisation souvent utilisée est la date de terminaison du dernier job ou de la dernière tâche du graphe. Il s'agit du **temps de complétion maximal** ou *makespan* noté C_{max}. Il correspond au temps passé par le système à réaliser tout son travail.

Quand une collection de tâches indépendantes provenant d'utilisateurs différents doit être ordonnancée, un autre critère d'optimisation est utilisé : le **temps de complétion moyen** noté $\sum C_i$. Chaque utilisateur souhaite que sa tâche finisse le plus tôt mais il n'attend pas que le système exécute globalement l'ensemble des tâches le plus rapidement possible. Ainsi le temps

de complétion moyen est la moyenne des temps de complétion C_i de toutes les tâches i de la collection. Comme nous divisons la somme $\sum C_i$ par le nombre de jobs n, nous minimisons en fait $\sum C_i$.

Dans un cluster les ressources de calcul ne sont pas illimitées, si bien que, quand les processeurs sont tous occupés, les jobs de certains utilisateurs sont mis dans une file d'attente. Á cause de cette file d'attente, un critère d'optimisation est alors le **temps d'attente moyen** (*flow time* noté $\sum F_i$) qu'il faut minimiser. Il s'agit de la moyenne des temps F_i écoulés entre l'arrivée du job i dans le cluster (à la date a_i) et la fin de son exécution (à la date C_i) : $F_i = C_i - a_i$. Dans [9], Bender et al. préconise plutôt de minimiser max F_i. En effet, minimiser une moyenne des temps d'attente a tendance à allonger les temps d'attente des petits jobs.

Dans le contexte des problèmes d'ordonnancement de jobs pour les clusters, les travaux de Bender et al. [9] aborde un critère d'optimisation fréquemment utilisé : le stretch. Le stretch S_i reflète le **ralentissement** engendré par l'exécution en concurrence avec d'autres jobs dans le cluster : $S_i = \frac{F_i}{C_i^*}$ avec F_i le temps d'exécution totale du job i et C_i^* le temps de calcul du job i s'il s'exécutait tout seul sur le cluster. Le calcul du stretch moyen correspond à la moyenne arithmétique de l'ensemble des stretchs S_i : $\frac{1}{N}\sum_i S_i$. Pour des raisons de risque de famine, nous préférerons minimiser le max stretch qui est le maximum des S_i : $\max_i\ S_i$.

Pour l'ordonnancement d'un graphe de tâches exécuté un grand nombre de fois, il est judicieux d'utiliser le **débit** comme critère d'optimisation, notamment pour l'ordonnancement de flux d'une même application. Avec le débit, nous considérons une fraction de tâche réalisée par unité de temps. Nous définissons la **période** comme étant le temps moyen séparant deux exécutions terminées de deux instances d'une application consécutives.

Ainsi, il n'y a pas de critère d'optimisation universel. Cependant le choix du critère d'optimisation a une grande importance. Comme nous venons de le voir à travers la minimisation du temps d'attente moyen des F_i, cette optimisation conduit à des effets indésirables. Il est alors judicieux de remplacer ce critère par le maximum des temps d'attente.

2.2 Ordonnancement de programmes parallèles dans les systèmes multiprocesseurs

Dans cette section nous présentons des exemples d'approches pour ordonnancer des programmes parallèles sur une machine multiprocesseur.

Dans [43] Dror G. Feitelson présente une étude sur l'ordonnancement de programmes parallèles concurrents devant s'exécuter sur un système comportant plusieurs unités de calcul (en anglais *Processing Elements* : PEs). Ces programmes parallèles comportent des threads. Dans cette étude il distingue deux problèmes d'ordonnancement de ces programmes. Il définit le premier problème à un seul niveau comme suit : un programme parallèle avec Nt threads demande au système de s'exécuter ; ce dernier est chargé de trouver sur quels PEs et quand exécuter chacun des Nt threads. Feitelson définit le second problème à deux niveaux : un programme parallèles avec Nt threads demande au système N PEs ; ce dernier trouve une allocation pour ce programme qui se débrouille lui même pour ordonnancer ses propres Nt threads sur les N PEs. Contrairement à l'ordonnancement à deux niveaux, dans l'ordonnancement à un seul niveau le système exécute tous les threads de tous les jobs en temps partagé. On retrouve cette technique

dans les systèmes d'exploitation des ordinateurs de bureau.

Au sein du problème d'ordonnancement à un seul niveau, Feitelson distingue trois approches. La première approche revient pour le système à trouver un partitionnement des PEs pour les jobs soumis, et ensuite d'exécuter un seul thread sur chaque PE. La deuxième approche consiste à ordonnancer plus de threads qu'il n'y a de PEs. La troisième approche est le "gang scheduling".

Le gang scheduling est défini comme suit :

- les threads sont regroupés dans des groupes
- les threads de chaque groupe s'exécutent simultanément sur des PEs avec un mapping one-to-one (un thread sur un PE)
- les threads d'un groupe sont tous suspendus ou repris en même temps

Généralement tous les threads d'un groupe sont les threads d'une même application, mais d'autres versions de gang scheduling sont proposées dans [76] par Parsons et al.

Les machines multiprocesseurs peuvent être classées dans deux catégories : les machines à mémoires partagées et les machines à mémoires distribuées. Généralement dans les machines à mémoires partagées, le système utilise l'ordonnancement à deux niveaux, car cette approche vise à réduire le changement de contexte grâce au partitionnement de l'ensemble des PEs. Chaque application dispose de son propre espace et se débrouille pour ordonnancer ses threads. Généralement dans les machines à mémoire distribuées il n'y a qu'un seul thread par PE.

2.3 La virtualisation pour l'ordonnancement

À présent nous parlons brièvement de la virtualisation car, dans nos travaux de recherche pour l'ordonnancement, nous l'abordons à travers la mise en place du *pliage de job* dans le contexte des clusters. La virtualisation est une technique qui crée des machines virtuelles sur des machines physiques.

Ce qui nous intéresse est que depuis quelques années la virtualisation s'intègre dans les nouveaux ordonnanceurs de jobs dans les clusters. Le logiciel Xen Grid Engine [40] combine un ordonnanceur de jobs (Sun grid Engine) et la virtualisation (Xen Hypervisor) afin d'optimiser l'ordonnancement des jobs séquentiels et parallèles. Dans [83], Stillwell et al. utilisent la virtualisation des nœuds d'un cluster pour gérer le temps passé par chaque job sur des processeurs partagés. D'autres travaux comme ceux de Sotomayor, et al. [82] cherchent à optimiser la satisfaction des utilisateurs à l'aide d'une approche utilisant la virtualisation et un système de bail. Dans ces travaux, les utilisateurs effectuent une requête à l'ordonnanceur et indiquent s'ils désirent que leurs jobs s'exécutent au plus tôt (*best effort*) ou bien à une date spécifiée. Afin de réaliser cet ordonnancement, les jobs peuvent être migrés et suspendus ou repris grâce à la virtualisation. Dans le chapitre 4 nous abordons une nouvelle technique d'ordonnancement appelée *pliage de jobs* fondée sur la virtualisation des processeurs visant à améliorer l'ordonnancement classique des jobs dans le contexte des clusters. L'arrivée de systèmes d'exploitation telles que CentOS [1], intégrant la virtualisation favorise l'amélioration des implémentations d'ordonnanceurs de jobs.

2.4 Des techniques d'optimisation pour l'ordonnancement

Comme un grand nombre de problèmes d'ordonnancement sont NP-Difficiles, les chercheurs utilisent des méthodes d'approximation pour résoudre des problèmes de taille moyenne ou de grande taille. Afin d'optimiser les ordonnancements nous pouvons utiliser des techniques d'optimisation classiques.

Parmi les méthodes classiques d'optimisation nous pouvons citer les méthodes itératives telle que le *hill climbing* qui réalise une recherche locale afin d'optimiser une fonction $f(x)$ définie sur $\mathbb{R}^n$. À chaque itération, l'algorithme essaie de modifier un élément du vecteur x. Tant que $f(x)$ est améliorée, il garde ce vecteur x. Cette méthode ressemble à la méthode de descente du gradient. Néanmoins, dans l'algorithme de descente du gradient, la direction à prendre est dirigée par la pente du polytope, autrement dit le gradient de $f(x)$. Nous pouvons aussi citer la Recherche Tabou qui, afin d'éviter de se trouver dans un optimum local, enregistre un historique des positions. Nous pouvons mentionner la méthode du recuit simulé inspirée des méthodes du domaine de la métallurgie [45], ou encore la méthode des colonies de fourmis [46].

Nous nous concentrons dans la suite sur les techniques d'optimisation suivantes : les algorithmes génétiques, les chaînes de Markov et les programmes linéaires/quadratiques car nous avons utilisé ces outils dans nos recherches afin d'optimiser les ordonnancements.

2.4.1 Les algorithmes génétiques

Les algorithmes génétiques sont des algorithmes évolutionnaires qui permettent d'obtenir une solution à des problèmes complexes d'optimisation. John Henry Holland en est l'inventeur dans les années 1960 [57].

Ces algorithmes s'inspirent de la théorie de l'évolution des êtres vivants. Dans le monde du vivant, les individus d'une espèce d'une population donnée se reproduisent entre eux, ils engendrent de nouveaux individus et s'intègrent dans cette population. Les caractéristiques des individus, appelées phénotypes, sont codées dans leurs gènes[1] dont le support matériel est le chromosome. Pendant le processus de reproduction, chacun des parents apporte ses propres chromosomes et une partie du génotype des parents est transmise à leur enfant. Les enfants se reproduisent à leur tour avec d'autres individus de la population. Dans la sélection naturelle les individus les plus adaptés à l'environnement ont plus de chance de se reproduire. Ainsi la population a tendance à contenir des individus qui ont les caractéristiques leur permettant de s'adapter au mieux à l'environnement. Au sein d'une population, il peut y avoir des mutations génétiques dues aux facteurs environnementaux, c'est-à-dire, qu'un individu peut voir un de ses gènes changer. Afin d'améliorer les algorithmes génétiques, des techniques de migration de populations peuvent être utilisées. En appliquant ce phénomène naturel d'évolution des individus afin d'optimiser un critère sur un système donné, les algorithmes génétiques suivent le paradigme suivant :

- une configuration du système est un individu ;
- un ensemble de configurations du système est une population ;

[1] une partie ou l'ensemble de ces gênes constitue le génotype

- une configuration du système est codée à l'intérieur d'un chromosome ;
- la capacité de l'individu à s'adapter à son environnement (*fitness*) est évaluée ;
- la sélection des individus pour la reproduction est réalisée avec un opérateur de sélection ;
- un croisement de chromosome est réalisé avec un opérateur de croisement ;
- une mutation est réalisée avec un opérateur de mutation ;
- une solution du problème d'optimisation est l'individu qui obtient le meilleur score d'adaptation (*fitness*) après un certain nombre de générations ou tout autre critère d'arrêt.

La figure 2.1 montre les étapes nécessaires d'un algorithme génétique. L'algorithme commence par créer une population initiale. Il évalue les fitness des individus s'ils sont inconnus. Puis il sélectionne les individus pour le croisement et les mutations. L'algorithme génétique se termine quand le critère d'arrêt est vérifié. L'algorithme génétique termine par exemple quand le nombre de générations (le nombre d'itérations) a atteint une valeur maximale, ou quand les individus de la population ont des patrimoines génétiques qui se ressemblent d'un point de vue statistique. L'algorithme génétique renvoie alors la solution portée par l'individu qui a le meilleur fitness.

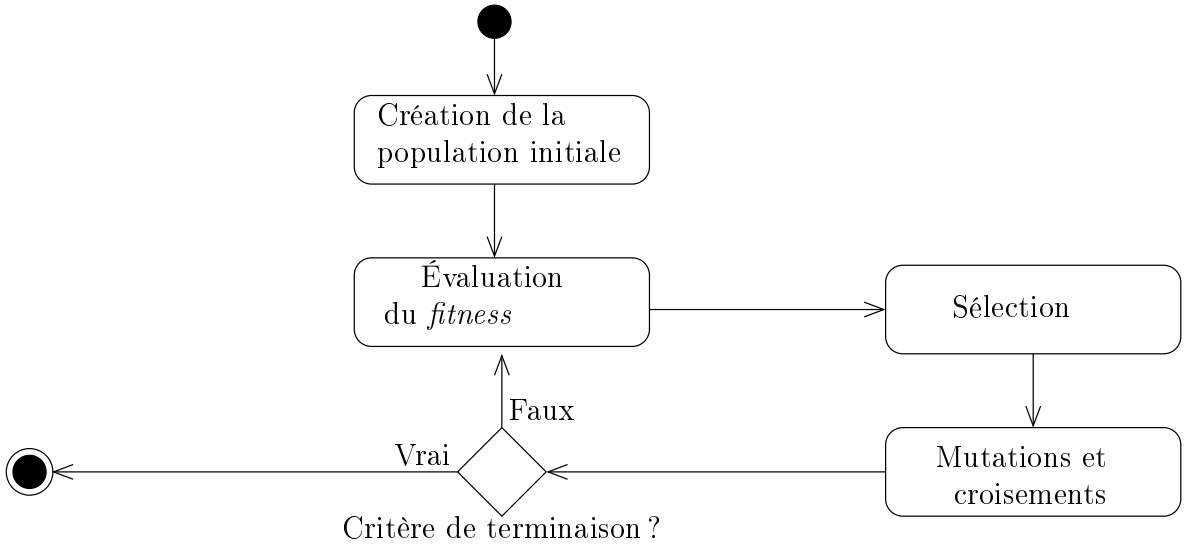


Figure 2.1 – Étapes d'un algorithme génétique

2.4.2 Les chaînes de Markov

Les chaînes de Markov sont présentées dans [71] par Grinstead et Snell. Une chaîne de Markov ou processus de Markov permettent de prédire le futur d'un système à états discrets à partir de son état présent. Le processus de Markov est sans mémoire. Le système passe d'un état à un autre suivant certaines probabilités de transition. Ces transitions sont consignées dans une matrice de transitions P (voir équation 2.1).

$$P = \begin{array}{c} \begin{array}{ccc} n(s_1) & n(s_2) & n(s_3) \end{array} \\ \begin{pmatrix} \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{8} & \frac{1}{8} & \frac{3}{4} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{pmatrix} \end{array} \begin{array}{c} s_1 \\ s_2 \\ s_3 \end{array} \tag{2.1}$$

Dans cet exemple, si le système est dans l'état s_1 alors la probabilité pour le système d'arriver aux états s_1, s_2 et s_3 sont respectivement $\frac{1}{2}$, $\frac{1}{4}$ et $\frac{1}{4}$.

On note $p_{13}^{(2)}$ la probabilité d'arriver à l'état s_3 depuis s_1 avec 2 transitions.

$$p_{13}^{(2)} = p_{11}p_{13} + p_{12}p_{23} + p_{13}p_{33}$$

On note $P^{(n)}$ la matrice de transition après n transitions.

Les chaînes de Markov sont par exemple utilisées en bioinformatique [74]. Dans ces travaux les chercheurs utilisent les chaînes de Markov afin de reconstruire l'arbre d'évolution d'un groupe d'organisme. Les chaînes de Markov permet également de résoudre des problèmes d'optimisation combinatoire. Ainsi l'algorithme 1 permet d'approcher une solution optimale à un problème d'optimisation combinatoire, sous réserve d'avoir au préalable une procédure $GENERER\_VOISIN(x_n)$ qui à partir de l'état x_n du système génère un nouvel état voisin y_n. Cet algorithme sert à trouver l'état du système afin de minimiser[2] une fonction objectif F. L'instruction à la ligne 6 calcule la probabilité α de conserver l'état voisin y_n. Supposons que le nouvel état y_n est tel que $F(y_n) \gg F(x_n)$, alors $-F(y_n) \ll -F(x_n)$. Ainsi $\alpha = \frac{e^{-\beta.F(y_n)}}{e^{-\beta.F(x_n)}} \ll 1$. Puisque nous minimisons $F(x)$ nous souhaitons conserver avec une forte probabilité x_n qui a une valeur F beaucoup plus petite que celle du nouveau voisin y_n, et générer un autre voisin à partir de x_n.

Soient :

- x le système ;
- $F(x)$ la fonction objectif du système x à minimiser ;
- N le nombre d'itérations.

2.4.3 Les programmes linéaires

Les programmes linéaires sont très largement utilisés en recherche opérationnelle afin de prendre des décisions optimales ou proches de l'optimal face à des problèmes complexes.

Un programme linéaire s'exprime sous la forme d'une expression à maximiser ou à minimiser en respectant un ensemble de contraintes.

En notation matricielle un programme linéaire s'exprime de la manière suivante :

Soient :

- x le vecteur d'inconnues $x \in \mathbb{R}^n$ qui contient les n variables $x_1, x_2, \ldots x_n$;

[2] Pour maximiser une fonction objectif il suffit de minimiser son opposé

Algorithme 1: Principe d'une chaîne de Markov pour approchée une solution optimale

```
1  x_0 ← structure de données initiales
2  n ← 0
3  tant que n < N faire
4      β ← log(n + 1)
5      y_n ← GENERER_VOISIN(x_n)
6      α ← min{1, e^{-β.F(y_n)} / e^{-β.F(x_n)}}
7      p ← GENERER_BERNOUILLI(1, α)
8      si p = 1 alors
9          x_{n+1} ← y_n
10     sinon
11         x_{n+1} ← x_n
12     n++
13 retourner min_n(x_n)
```

- c les coefficients des x_i dans l'expression à optimiser : $c \in \mathbb{R}^n$ et l'expression à optimiser est $c^T x$
- $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$. L'inégalité vectorielle $Ax \leq b$ exprime les m contraintes d'inégalité sur n variables x_i.

L'objectif est de déterminer les valeurs de chaque composante du vecteur x afin d'optimiser l'expression : $c^T x$ sous les contraintes $Ax \leq b$.

Un algorithme très populaire qui permet de résoudre un programme linéaire est l'algorithme du simplexe de George Dantzig[3]. En terme de géométrie, l'inégalité vectorielle $Ax \leq b$ définit un polytope convexe dans un espace à n dimensions. L'algorithme du simplexe est fondé sur la propriété d'après laquelle si le problème est réalisable et borné alors une solution optimale existe et est finie. Une solution optimale est située sur un sommet du polytope. En moyenne l'algorithme du simplexe est polynomial par rapport au nombre de variables [14].

Si dans la formulation quelques valeurs du vecteur x sont restreintes à des valeurs entières, le programme linéaire est dit mixte. Généralement la résolution de tels programmes ne s'effectue pas en temps polynomial. Néanmoins ils sont très utilisés dans l'ordonnancement des tâches de processus pétrochimiques [58, 47]. Par ailleurs si toutes les valeurs de x sont des entiers alors il s'agit d'un programme linéaire en nombres entiers. Généralement la résolution de ces programmes est NP-Difficile. Pour des cas simples la résolution de tels programmes est possible avec CPLEX [4] ou Gurobi [5]

Cette technique est également utilisée dans les travaux pour l'ordonnancement [7], Beaumont et al. présentent la technique de l'ordonnancement en régime permanent (*steady-state*), pour ordonnancer un ensemble infini d'applications décrites par un DAG au moyen d'un programme linéaire en nombres rationnels. Ils recherchent le débit optimal en régime permanent.

[3] `http://en.wikipedia.org/wiki/George_Dantzig`
[4] `www.ibm.com/software/integration/optimization/cplex-optimizer/`
[5] `http://www.gurobi.com/`

2.4.4 Les programmes quadratiques

Les programmes quadratiques servent à optimiser une fonction quadratique, de la forme

$$f(x_1, \ldots, x_n) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} H_{i,j} x_i x_j + \sum_{i=1}^{n} g_i x_i$$

sous les contraintes :

$$\forall i \in \{1, \ldots, m\}, \ \sum_{j=1}^{n} A_{i,j} x_j - b_i \leq 0$$

Dans la fonction objectif, $H_{i,j}$ représente le coefficient que multiplie le produit $x_i x_j$ qui forme la partie quadratique de la fonction et g_i représente le coefficient que multiplie x_i qui forme la partie linéaire de la fonction.

Un tel programme quadratique peut s'exprimer sous forme matricielle où les coefficients $H_{i,j}$ sont les éléments de la matrice H et les coefficients g_i sont les composantes du vecteur g.

optimiser :

$$f(x) = \frac{1}{2} x^T H x + g^T x$$

sous la contrainte :

$$Ax \leq b$$

Si H est une matrice définie positive[6], un programme quadratique peut être résolu en temps polynomial.

Dans une étude [81], Skutella utilise un programme quadratique pour résoudre un problème d'ordonnancement de jobs munis de dates d'arrivée. L'objectif est de minimiser le temps de complétion moyen pondéré par les poids de chaque job.

2.5 Algorithmes d'ordonnancement

Nous présentons dans ce paragraphe les algorithmes d'ordonnancement principaux. Nous présentons d'abord les algorithmes d'ordonnancement de jobs, puis les algorithmes d'ordonnancement d'applications décrites par un DAG.

2.5.1 Algorithmes d'ordonnancement de jobs *online*

De nombreux algorithmes d'ordonnancement sont décrits et étudiés dans la littérature. Nous allons décrire l'algorithme *FCFS* (*First Come First Served* : premier arrivé premier servi) et l'algorithme de *Backfilling*.

[6] Une matrice M $n \times n$ à valeurs réelles est définie positive si $\forall z > 0$, $z \in \mathbb{R}^n$ $z^T M z > 0$

a L'algorithme *FCFS*

L'algorithme *FCFS* est un algorithme d'ordonnancement très simple. Le premier job qui arrive dans le système est le premier job qui est exécuté. En fait cet algorithme exécute les jobs dans l'ordre d'arrivée. Comme le montre un exemple dans [27], cet algorithme peut entraîner une sous utilisation du cluster. Le cas peut être illustré avec un cluster de 100 processeurs sur lequel un job A est en cours d'exécution et utilise 55 processeurs. Si le job B arrive et demande 50 processeurs avant qu'un job C demande 10 processeurs alors les deux jobs B et C attendent tous les deux que le job A termine, alors que le petit job C pourrait s'exécuter sans retarder B.

b *L'algorithme de* Backfilling

Pour comprendre l'idée du l'algorithme de *Backfilling*, nous reprenons l'exemple précédent. Comme le job B ne peut être ordonnancé nous sommes tentés d'ordonnancer le job C car il dispose de suffisamment de processeurs disponibles pour s'exécuter. Cette approche est sujette aux famines car elle favorise les petits jobs, ce qui risque de repousser les gros jobs indéfiniment. Pour empêcher cette famine, l'idée est d'imposer que les jobs qui passent devant ne repoussent pas l'ordonnancement des jobs arrivés les premiers. Cet algorithme est appelé *Conservative Backfilling*.

Une autre version du *Backfilling* appelé *Agressive Backfilling* consiste à faire passer devant les jobs en attente du moment qu'ils ne repoussent pas leur ordonnancement du premier job en attente. *Agressive Backfilling* est utilisé dans l'ordonnanceur EASY (*Extensible Argonne Scheduling sYstem*) [80].

2.5.2 Algorithmes d'ordonnancement de jobs offline

Dans la section précédente nous avons décrit deux algorithmes pour ordonnancer des jobs qui arrivent au fil du temps. Néanmoins la situation où les jobs seraient tous présents à un moment donné et où on souhaite les exécuter le plus rapidement possible sur une machine multiprocesseur peut se présenter. Nous devons alors résoudre un problème d'ordonnancement multiprocesseur (*multi-processor scheduling problem*).

Un problème d'ordonnancement multiprocesseur se définit comme suit. Soient une machine à m processeurs et un ensemble de n jobs indépendants. Chacun de ces jobs i a une durée d'exécution connue à l'avance $reqtime_i$ et s'exécute sur un seul processeur : ce sont des jobs séquentiels. Le but est de minimiser le temps d'exécution de l'ensemble des jobs (noté C_{max}). Le problème peut se décliner en deux versions. Les processeurs sont soit homogènes soit hétérogènes. Dans les deux cas ce problème est connu pour être NP-difficile [50].

Un autre problème similaire existe pour les jobs parallèles toujours dans le but de minimiser C_{max}. Soient une machine à m processeurs et un ensemble n de jobs indépendants. Chacun de ces jobs i ont une durée d'exécution connue à l'avance $reqtime_i$ et s'exécute sur $reqproc_i \leq m$ processeurs. Le problème est toujours de minimiser le temps d'exécution de l'ensemble des jobs. Ce problème a largement été étudié [37, 43] et il est NP-difficile.

Comme les deux problèmes d'ordonnancement de jobs séquentiels et de jobs parallèles sur une machine multiprocesseur sont NP-difficiles. Les recherches pour résoudre ces problèmes ne

peuvent se reposer que sur des heuristiques. Les heuristiques les plus simples et bien connus sont le *LPTF* (*Largest Processing Time First*) pour les jobs séquentiels et le *LTF* (*Largest Task First*) pour les jobs parallèles. Ces deux algorithmes sont des algorithmes de liste. L'algorithme *LPTF* trie les jobs i dans l'ordre décroissant des $reqtime_i$, puis il les ordonnance dans cet ordre. L'algorithme *LTF* trie les jobs i dans l'ordre décroissant des $reqtime_i \times reqproc_i$, puis il les ordonnance.

Il existe une version de *LTF* pour les jobs moldables qui est donnée par l'algorithme 2 conçu par Dutot et al. dans [38]. Cet algorithme a une complexité de $\mathcal{O}(m \times n \times ln(n))$.

Algorithme 2: Version de LTF pour les jobs moldables

pour p *de* 1 *à* m **faire**
  $\forall i\ reqproc'_i \leftarrow reqproc_i$
  $\forall i\ reqtime'_i \leftarrow reqtime_i$
  pour chaque *job* i **faire**
    si $p < reqproc_i$ **alors**
      $reqtime'_i \leftarrow \left\lceil \frac{reqproc_i}{p} \right\rceil$
      $reqproc'_i \leftarrow p$
    trier les jobs i dans l'ordre décroissant de $reqtime'_i * reqproc'_i$
    ordonnancer les jobs i dans cet ordre
retourner l'ordonnancement avec le makespan minimal

2.5.3 Algorithmes d'ordonnancement d'applications décrites par un DAG

Les problèmes d'ordonnancement de décrites par un DAG ont été très largement étudiés dans la littérature. Minimiser la date de la dernière tâche (*makespan*) d'un DAG sur une plate-forme distribuée homogène avec un nombre limité de ressources est un problème connu pour être NP-Difficile [67]. Le même problème avec une plate-forme hétérogène est plus complexe, donc reste nécessairement NP-Difficile.

Nous trouvons des travaux qui se concentrent sur l'ordonnancement d'une application décrite par un DAG [10, 90]. Des stratégies orientées makespan pour ordonnancer une application sur les grilles sont présentées dans les travaux de Mandal et al. [72] qui décrivent une application concrète dans le monde médical. Nous trouvons aussi des travaux sur l'ordonnancement d'applications qui arrivent dans la grille au fil de l'eau [59] et des travaux qui se concentrent sur l'ordonnancement d'un lot d'applications identiques [7]. Pour ce dernier cas chaque instance de l'application traite un ensemble de données d'entrée.

Dans la littérature, les solutions pour optimiser le makespan d'un ensemble de tâches sont fondées généralement sur des algorithmes de liste. Elles utilisent des heuristiques telles que *Earliest Finish Time* [88] (EFT) qui donne une grande priorité aux tâches qui s'exécutent le plus tôt ou elles utilisent des heuristiques qui reposent sur le chemin critique [65]. Kwok et al. font une étude [64] pour ordonnancer des DAGs sur une plate-forme homogène. Dans [15] Braun et al. présentent une étude comparative qui évalue onze heuristiques par exemple : Min-min, Max-min. D'autres études sur l'ordonnancement d'applications sur une grille de calcul existent intégrant d'autres critères d'optimisation que le makespan. Les recherches présentées

dans [10] par Benoit et al. utilisent des réseaux de Petri et des pipelines afin d'optimiser le débit d'applications répliqués sur une plate-forme hétérogène. Nous trouvons d'autres critères d'optimisation comme la latence [90]. Enfin, dans [7], Beaumont et al. présentent la technique de l'ordonnancement en régime permanent (steady-state), pour ordonnancer un lot en nombre infini d'applications identiques. Ils prennent le débit comme critère d'optimisation. Dans [32], Diakité et al. présentent les travaux menés pour ordonnancer un ensemble fini d'applications identiques au moyens de trois algorithmes adaptés à la situation afin de minimiser le makespan : l'algorithme d'ordonnancement en régime permanent [7], un algorithme génétique [28] et un algorithme de liste [88].

Dans la suite nous présentons divers algorithmes d'ordonnancement de d'applications modélisés sous la forme de DAGs. Nous rappelons d'abord le principe des algorithmes *HEFT* et *EFT* pour ordonnancer une application décrite par un DAG sur une plate-forme hétérogène, puis nous présentons un algorithme génétique pour l'ordonnancement d'une application sur une plate-forme hétérogène.

a Algorithmes HEFT et EFT

Ici nous rappelons le principe des algorithmes *HEFT* et *EFT*. Soit un DAG $G = D(V, E)$ où V est l'ensemble des tâches et E est l'ensemble des dépendances entre tâches. Soit un cluster à P processeurs. L'algorithme HEFT [88] fonctionne de la manière suivante. Les tâches i du graphe sont rangées dans l'ordre décroissant des $rang(i)$:

$$rang(i) = w(i) + \max_{j \in succ(i)} (c_{i,j} + rang(j))$$

avec $w(i)$ le coût de la tâche i, $c_{i,j}$ le coût (temps par exemple) de la transmission de la tâche i vers la tâche j, $succ(i)$ l'ensemble des successeurs de la tâche i. Soit s le puits du graphe, on définit $rang(s) = w(s)$. Le rang peut être interprété comme le coût total maximal d'une tâche i du graphe G vers le puits s (temps total avant la sortie par exemple). Pour chaque tâche i l'algorithme attribue les $w(i)$ et les $c_{i,j}$ en calculant les moyennes des coûts sur les processeurs et les liens de communication. Il calcule pour chaque tâche i $rang(i)$. Les tâches i du graphe sont rangées dans l'ordre décroissant des $rang(i)$ et placées dans une liste L.

Algorithme 3: L'algorithme HEFT

```
1 pour chaque tâche i de la liste L faire
2     sélectionner la première tâche de la liste L
3     pour chaque processeur p faire
4         calculer EFT(i, p)
5     choisir p avec le plus petit EFT(i, p)
6     retirer i de L
```

La fonction $EFT(i, p)$ (*Earliest Finish Time*) représente la date de terminaison au plus tôt de la tâche i sur le processeur p. Elle se calcule récursivement avec une fonction auxiliaire $EST(i, p)$ qui représente la date de début au plut tôt de la tâche i sur le processeur p.

	T_0	T_1	T_2	T_3	T_4	T_5
p_0	5	1	3	4	2	1
p_1	1	4	5	2	7	6

Table 2.1 – Plate-forme sur laquelle est exécutée l'application 2.2

$$\forall p,\ EST(tache_{entree}, p) = 0$$

$$EST(i,p) = max\left\{avail(p), \max_{j \in pred(i)}(finishtime(j) + c_{j,i})\right\}$$

$$EFT(i,p) = duration_{i,p} + EST(i,p)$$

avec $avail(p)$ la date où p est disponible, $pred(i)$ les tâches parents de i, $finishtime(j)$ la date de fin de la tâche j et $duration_{i,p}$ la durée d'exécution de la tâche i sur le processeur p.

b Algorithme génétique pour l'ordonnancement d'un DAG

Les algorithmes génétiques ont déjà été utilisés en ordonnancement. Par exemple dans [51] Goh et al. utilisent un algorithme génétique pour ordonnancer des tâches munies de dépendances sur une plate-forme totalement hétérogène (processeurs hétérogènes et liens de communication hétérogènes). Nous pouvons aussi citer les travaux de Daoud et al. [28] qui ont travaillé sur un algorithme génétique pour l'ordonnancement d'un DAG (GATS : *Genetic Algorithm for Tasks Scheduling*) sur une plate-forme hétérogène où les coûts des communications sont négligeables.

Dans cette section nous présentons brièvement le principe de l'algorithme génétique de Daoud et al. Comme tout algorithme génétique, il suit le paradigme présenté en 2.4.1.

La figure 2.2 montre un exemple d'un graphe de tâches. Les figures 2.4(a) et 2.4(b) représentent deux chromosomes qui codent chacun un ordonnancement du DAG donné à la figure 2.2. Un chromosome représente un individu de la population. Il code seulement les allocations de chaque tâche à un processeur ainsi que l'ordre des tâches exécutées sur un processeur. Avec le chromosome seulement, nous ne pouvons pas déterminer la date de début de chaque tâche. Pour construire un ordonnancement à partir d'un chromosome, il faut s'aider des contraintes de précédence du graphe de tâches. Ainsi avec toutes ces données un ordonnancement peut être construit.

Le tableau 2.1 représente les temps de calcul d'une plate-forme à deux nœuds (P0 et P1) pour chaque tâche de l'application de la figure 2.2. Nous prenons l'exemple du chromosome de la figure 2.4(a) et à travers la figure 2.3 nous donnons son ordonnancement après construction en tenant compte des contraintes de précédence données par la figure 2.2.

c Ordonnancement de lots d'instances d'une application décrite par un DAG

Dans [8], Legrand et al. ont mis au point un algorithme qui travaille sur des lots d'instances d'une application décrite par un DAG : le *Steady-state* qui est un algorithme fondé sur le

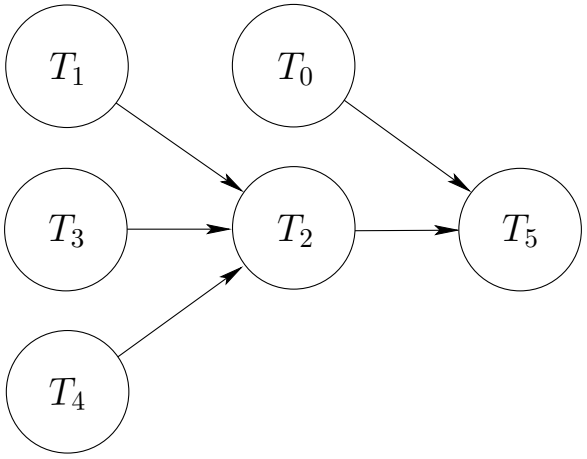


Figure 2.2 – Exemple d'une application

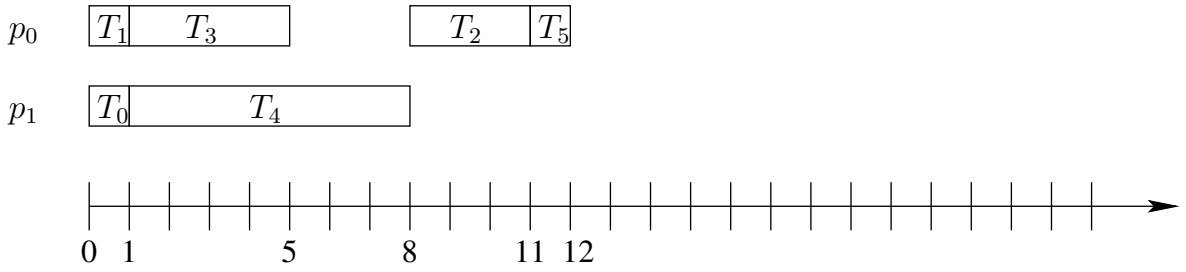


Figure 2.3 – Exemple d'un ordonnancement

Figure 2.4 – Exemple de chromosomes

principe sur l'ordonnancement en régime permanent introduit par Bertsimas et Gamarnik [12]. Asymptotiquement, quand le nombre d'instances du lot tend vers l'infini, cet algorithme trouve au moyen d'un programme linéaire en nombres rationnels un ordonnancement optimal qui donne le débit maximal. Dans [31] Diakité et al. se sont inspirés de l'algorithme du Steady-State proposé dans [7] pour ordonnancer un lot borné de DAGs. Dans cette étude le makespan doit être minimisé.

2.6 Synthèse

Dans cette partie nous avons abordé le contexte dans lequel s'inscrit notre travail. Nous avons brièvement présenté les grilles et les fermes de calcul en tant que ressources de calcul pour le HPC. Nous avons vu que l'utilisation de ces ressources soulevaient des questions. Ce qui nous a amené à définir et présenter les problématiques d'ordonnancement dans le contexte des grilles et des fermes de calcul. Nous avons vu que les problèmes d'ordonnancement sont des problèmes d'optimisation.

Cette partie nous a permis de définir des éléments de base afin que le lecteur puisse comprendre la portée de nos contributions. Nous posons notamment une autre définition du workflow. Nous avons abordé des techniques d'optimisation que nous utilisons dans nos travaux de recherche. Nous montrons dans la suite comment nous nous sommes appropriés les outils pour concevoir de nouvelles techniques afin de contribuer à l'ordonnancement sur plates-formes parallèles ou distribuées.

Deuxième partie

Contributions

Chapitre 3

Des algorithmes pour ordonnancer un workflow

Durant sa thèse [33], Sékou Diakité a réalisé des expériences sur le steady-state, sur l'algorithme génétique de Daoud et al., appelé GATS (*Genetic Algorithm for Task Scheduling*) ainsi que sur un algorithme de liste de type EFT ; ces algorithmes ont été introduits au chapitre précédent. Il a adapté en particulier le GATS et l'algorithmes EFT, à l'origine conçus pour traiter une seule application, afin qu'ils traitent un workflow de plusieurs instances d'une application. Quant à l'algorithme steady-state qui est conçu pour traiter un workflow constitué d'un nombre non borné d'instances d'une application, Sékou Diakité l'a adapté afin qu'il traite un workflow fini d'instances d'une application.

Sa thèse terminée, il a laissé des données de trace générées par ses simulations. À partir de ces données nous avons mené des recherches plus avancées. Grâce à des statistiques réalisées sur ces données, nous montrons que dans la majorité des cas l'algorithme génétique et le steady-state donne de bons ordonnancements. De plus grâce à ces données, nous montrons dans quels cas il est préférable d'utiliser l'un ou l'autre des trois algorithmes cités plus haut.

Dans ce chapitre nous étudions le problème de l'ordonnancement d'un workflow sur une plate-forme hétérogène. Le but est de minimiser le makespan de l'ordonnancement. Le workflow est composé de plusieurs instances d'une application unique modélisée par un DAG. Une instance de l'application suit un patron donné par le DAG, composé de plusieurs tâches liées par des contraintes de précédence. Le DAG est un anti-arbre, un graphe orienté sans cycle et sans bifurcation. Nous supposons que chaque tâche est associée à un type, et que la plate-forme peut traiter l'ensemble F des types de tâches du DAG. De plus chaque nœud de la plate-forme est capable de traiter un sous ensemble de types de tâches de F.

Optimiser le makespan de l'ordonnancement d'un DAG sur une plate-forme hétérogène est un problème NP-difficile [67]. Nous ne pouvons donc que nous reposer sur des heuristiques afin de résoudre notre problème. Nous reprenons les travaux de Diakité et al. afin de tirer des conclusions plus approfondies sur les algorithmes : l'algorithme steady-state, le GATS et un algorithme de liste EFT. Nous évaluons comparativement ces trois algorithmes appliqués à un workflow.

3.1 La description du modèle

3.1.1 Le modèle de la plate-forme

La plate-forme d'exécution considérée est hétérogène : les différents nœuds mettent des temps différents pour réaliser la même opération. Dans notre modèle, nous supposons que nous connaissons avant l'exécution des instances d'une même application les informations suivantes :

- les fonctions disponibles sur chaque nœud ;
- les temps d'exécution des fonctions disponibles sur chaque nœud. Comme la plate-forme est hétérogène il n'existe aucune relation entre les temps d'exécution des différents nœuds pour une même fonction.

La plate-forme PF est hétérogène et composée de m machines. Elle est modélisée par un graphe non orienté $PF = (P, L)$ où les sommets de $P = \{p_1, \ldots, p_m\}$ représentent les machines et où les arêtes de L sont des liens de communication entre les machines. PF est connexe.

Chaque nœud p_i de P est capable de réaliser un sous ensemble des fonctions dont l'application a besoin. Soient F_i le sous ensemble de fonctions que le nœud p_i est capable de réaliser et $F = \cup_i F_i$ l'ensemble des fonctions disponibles sur la plate-forme, avec $1 \leq i \leq n$ et $p_i \in P$. De plus, chaque lien de communication (p_i, p_j) a une bande passante $bw(p_i, p_j)$ qui est la taille des données qui peuvent être transférées à travers un lien de communication par unité de temps.

La figure 3.1 donne un exemple de plate-forme. Dans cette figure, les lettres A à D représentent quatre fonctions déployées sur les nœuds de la plate-forme.

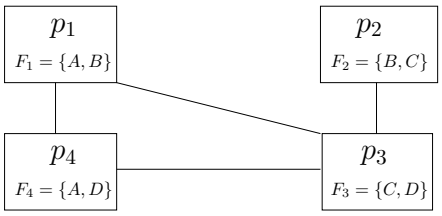


Figure 3.1 – Un exemple d'une plate-forme avec $F = \{A, B, C, D\}$

Le tableau 3.1 donne un exemple pour décrire une plate-forme. Les valeurs correspondent aux temps nécessaires pour réaliser une fonction donnée sur un nœud donné. Quand une case du tableau ne contient aucune valeur, cela signifie que le nœud n'est pas capable de réaliser cette fonction. Par exemple, le nœud p_4 est capable de réaliser les fonctions A et D mais est incapable de réaliser ni B ni C.

P		p_1	p_2	p_3	p_4
Fonction	A	20	-	-	15
	B	10	10	-	-
	C	-	10	10	-
	D	-	-	10	10

Table 3.1 – Un exemple d'une plate-forme et ses performances

a Modèle de communication

Dans notre étude les machines sont connectées par des liens de communication point-à-point. Cette représentation peut modéliser une grille. Dans la littérature, plusieurs modèles de communication existent, tels que le modèle 1-port ou le modèle multi-port [7, 24]. Nous avons choisi d'utiliser le modèle 1-port du fait de sa facilité de modélisation et d'implémentation tout en restant réaliste. Dans ce modèle, seulement une donnée peut être transmise au même moment à travers un lien de communication et un nœud ne peut au plus que réaliser une seule réception et une seule émission. Nous définissons $\mathcal{R}(p_k, p_i) = \{(p_j, p_{j'}) \in \mathcal{L}\}$ comme étant la route de p_k à p_i.

3.1.2 Le modèle de l'application

Nous venons de définir le modèle de la plate-forme afin d'exécuter plusieurs fois une application modélisée sous la forme d'un DAG avec des jeux d'entrée différents. Chacune de ces tâches doit être exécutée selon des contraintes d'ordre.

Nous supposons que l'ensemble des données d'entrée sont prêtes au tout début et que la même application est appliquée sur cet ensemble de données. Ainsi, nous devons ordonnancer un workflow constitué d'instances d'une même application sur la plate-forme dont le modèle a été décrit plus haut. De plus il n'y a pas de date butoir (*deadline*) pour les tâches. Dans cette étude nous nous concentrons uniquement sur l'exécution d'un seul workflow.

Une application J est modélisé par un graphe orienté sans cycle (en anglais DAG : *Directed Acyclic Graph*) : $J = (\tau, D)$ où $\tau = \{T^1, \ldots, T^N\}$ est un ensemble de N tâches T^k $(1 \leq k \leq N)$ et $D \subset \tau \times \tau$ représente les contraintes de précédence entre les tâches. Si une contrainte de dépendance (T^k, T^i) au niveau de l'application J existe entre la tâche T^k et la tâche T^i cela se traduit par un passage de données sous la forme d'un fichier $file^{k,i}$ envoyé de la tâche T^k à la tâche T^i pour que cette dernière puisse s'exécuter. La taille de ce fichier est $size(file^{k,i})$.

Soit $F(J)$ l'ensemble des fonctions utilisées par l'application J et $F^k(J)$ la fonction dont la tâche $T^k \in \tau$ a besoin avec $1 \leq k \leq N$. Ainsi, $F(J) = \cup_k \{F^k(J)\}$ avec $1 \leq k \leq N$ et $T^k \in \tau$. Pour exécuter l'application J, nous supposons que $F(J) \subset F$. Rappelons que F est l'ensemble des fonctions que la plate-forme P est capable de réaliser. Comme nous utilisons un modèle de plate-forme dans lequel il n'existe aucune relation entre les temps d'exécution des différents nœuds pour une même fonction, le temps nécessaire pour exécuter la tâche T^k dépend du nœud p_i.

Soit un workflow $B = \{J_1, \ldots, J_n\}$ qui contient n instances J_j de l'application J. Nous reprenons les notations introduites en ajoutant un indice j afin de différencier les composants des différentes instances de l'application. Ainsi chaque instance J_j est modélisé par un graphe (τ_j, D_j). Nous écrivons les notations suivantes :

- T_j^i : la tâche T^i de l'instance J_j ;
- (T_j^k, T_j^i) : la contrainte de dépendance entre la tâche T_j^k et la tâche T_j^i de l'instance J_j ;
- $file_j^{k,i}$: le fichier envoyé de la tâche T^k à la tâche T^i de l'instance J_j

De plus, nous définissons $a(i, j)$ de telle sorte que $p_{a(i,j)}$ est la machine sur laquelle la tâche T_j^i

est assignée. Nous appelerons *job* une instance d'une application.

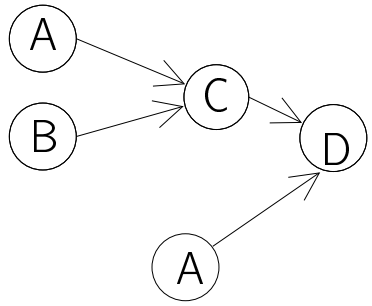


Figure 3.2 – Un exemple d'application

La figure 3.2 donne un exemple d'application que nous devons ordonnancer n fois, n étant le nombre de d'instances de l'application dans le workflow. L'application est constitué de cinq tâches $\{T_1, \ldots, T_5\}$. Chaque cercle indique une tâche et son type de tâche (dans cet exemple l'application est constitué de deux tâches de type A, une de type B, une de type C et une de type D). Les flèches indiquent des contraintes de précédence. On remarque que chaque instance de cette application a besoin d'exécuter deux fois le même type de fonction A.

Le problème traité ici, selon la classification $\alpha|\beta|\gamma$ (plate-forme | application | critère à optimiser) des problèmes d'ordonnancement donnée par Graham et al. dans [53], se définit par $R|\text{workflow}|C_{max}$: la plate-forme est hétérogène, ainsi il n'existe aucune relation entre les temps d'exécution des différents nœuds pour une même fonction (dénoté par R) et nous optimisons le makespan C_{max} d'un workflow. La taille du workflow peut varier entre une dizaine et des milliers d'instances d'une même application.

3.2 Évaluations des algorithmes pour ordonnancer un workflow sans tenir compte des coûts de communication

Ce paragraphe est dédié à l'évaluation des algorithmes cités plus haut sans prendre en compte les coûts de communication liés aux transferts entre deux nœuds d'une paire. Ce cas arrive très souvent dans les applications à gros grains dont les différentes tâches ont des temps d'exécution qui dominent la durée des temps de transfert. Ainsi nous posons $\forall j, k, i,\ size(file_j^{k,i}) = 0$. L'étude avec les coûts de communication est faite au paragraphe 3.3.

3.2.1 L'algorithme de Liste

L'algorithme de Liste que nous évaluons ici est une adaptation de l'algorithme EFT (en anglais : *Earliest Finish Time*) pour un workflows d'instances d'une même application et des plates-formes hétérogènes. L'algorithme 4 montre le fonctionnement de l'ordonnanceur d'un workflow qui utilise EFT. Cet ordonnanceur est appliqué sur une plate-forme, une application et un nombre d'instances de l'application. D'abord il instancie *nombre* fois *application* avec la procédure `creer_workflow`. Puis il parcourt chaque tâche libre[1] et applique EFT pour choisir sur quel processeur P de la plate-forme exécuter la tâche libre. La fonction `plus_ancienne`

[1] une tâche libre est une tâche dont toutes les tâches qui la précèdent ont été exécutées

renvoie une tâche libre T de l'instance la plus ancienne en cours d'exécution.

Algorithme 4: `ordonnanceurEFT(plateforme, application, nombre)`

```
1 workflow ⟵ creer_workflow(application, nombre);
2 tant que non estVide(workflow) faire
3     libres ⟵ maj_libres(workflow, libres);
4     T ⟵ plus_ancienne(libres);
5     P ⟵ EFT(plateforme, libres);
6     affecter(T, P);
7     si estVide(libres) alors
8         attendreUneTacheLibre(workflow);
```

Initialement l'algorithme EFT n'est pas adapté à l'ordonnancement d'un workflow. Dans un problème d'ordonnancement d'un workflow, la première tâches de chaque application est une tâche libre, par conséquent il faut attendre que toutes les premières tâches soit exécutées pour exécuter les deuxièmes tâches. Ceci est pénalisant quand on souhaite minimiser le makespan global. Diakité et al. ont modifié l'algorithme 4 permettant de limiter ce phénomène : ils obtiennent l'algorithme 5. Ils ont ajouté la procédure `attendreUneRessourceOisive`. Supposons une plate-forme composée de deux processeurs P1 et P2 et une application G composée de deux taches ($A \rightarrow B$), on ordonnance un workflow de 10 instances de l'application G avec l'algorithme 5, le processeur P1 exécute la tâche A de l'instance 1 et en parallèle la tâche A de l'instance 2 sur P2. Comme l'instance 1 est la plus ancienne et que B est libre, l'ordonnanceur choisit d'exécuter la tâche B de l'instance 1, et ainsi de suite.

Algorithme 5: `ordonnanceurEFT(plateforme, application, nombre)`

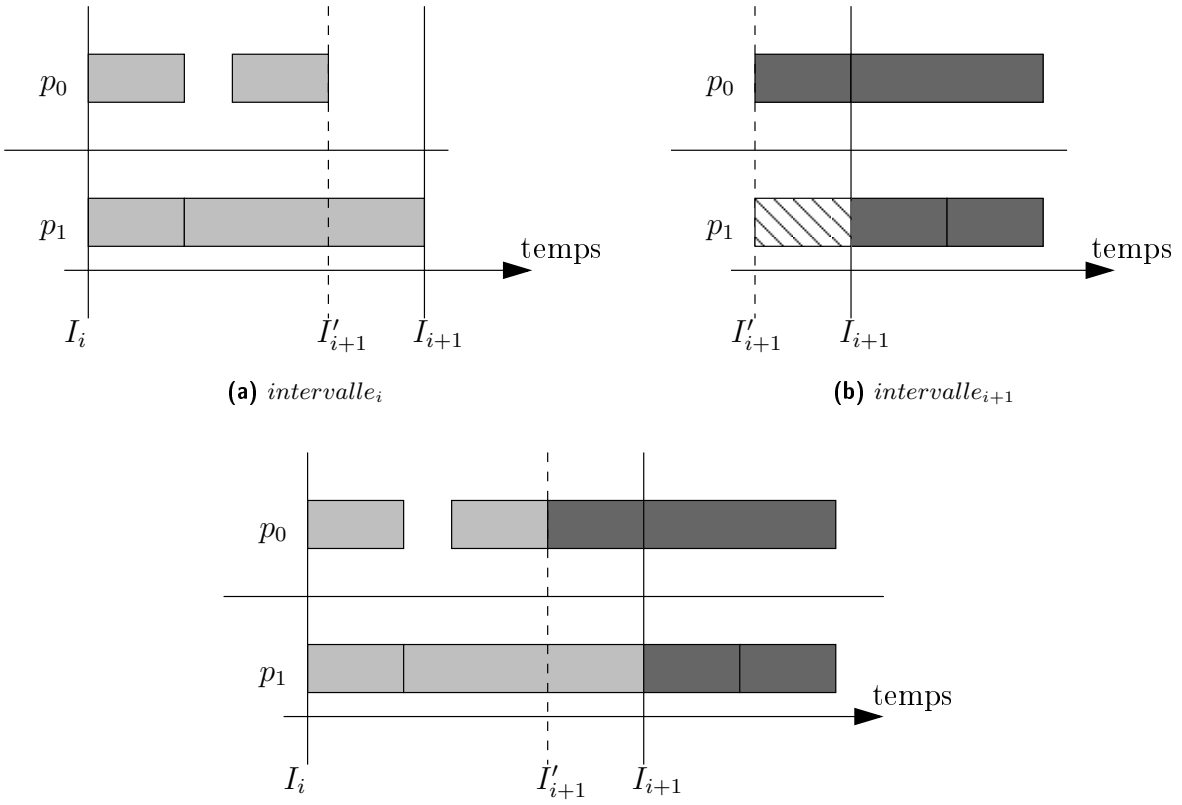
```
1 workflow ⟵ creer_workflow(application, nombre);
2 tant que non estVide(workflow) faire
3     libres ⟵ maj_libres(workflow, libres);
4     T ⟵ plus_ancienne(libres);
5     P ⟵ EFT(plateforme, libres);
6     affecter(T, P);
7     si estVide(libres) alors
8         attendreUneTacheLibre(workflow);
9     attendreUneRessourceOisive(plateforme);
```

3.2.2 L'algorithme génétique

L'algorithme génétique que nous évaluons dans ce paragraphe est une adaptation de l'algorithme génétique de Daoud et al. [28] pour ordonnancer un workflow sur une plate-forme hétérogène. Cette adaptation a été réalisée durant la thèse de Sékou Diakité. Nous rappelons comment Diakité et al. ont adapté le GATS pour traiter un workflow de m instances d'une même application.

Afin d'adapter le GATS pour le traitement de n instances d'une même application, ils ont introduit la notion d'intervalle d'ordonnancement. Ils découpent les n instances en x intervalles

de y instances ($n = x \times y$). La figure 3.3 montre un exemple de deux intervalles successifs. Au lieu de faire suivre les intervalles les uns après les autres, ils profitent du fait qu'une tâche sur un processeur finisse avant la fin de l'intervalle courant pour commencer à l'intérieur de l'intervalle courant une tâche de l'intervalle suivant.



(a) $intervalle_i$ **(b)** $intervalle_{i+1}$

(c) Ordonnancement

Figure 3.3 – Recouvrement des intervalles $intervalle_i$ et $intervalle_{i+1}$.

3.2.3 L'algorithme du steady-state

Nous rappelons le principe de l'algorithme du steady-state appliqué à un problème d'ordonnancement d'un nombre infini d'instances d'une même application. Diakité et al. ont adapté cet algorithme pour ordonnancer un nombre borné d'instances d'une même application fini sur une plate-forme hétérogène.

Le steady-state est caractérisé par les variables $\alpha(p_i, T^k)$ qui représentent la fraction de durées moyennes prises par le nœud p_i pour exécuter la tâche t_k pendant une unité de temps. Ainsi on définit des contraintes sur $\alpha(p_i, T^k)$ car il faut respecter le fait qu'un nœud exécute les tâches qu'il peut réaliser et que son utilisation n'excède pas ses capacités. Nous définissons la consommation $cons(p_i, T^k)$ qui est le nombre de tâches T^k qui sont exécutées par le nœud p_i en une unité de temps et $w_{i,k}$ qui est le temps passé par le nœud p_i pour exécuter la tâche T^k. La consommation $cons(p_i, T^k)$ est donc le débit de la tâche T^k sur le nœud p_i. Ainsi maximiser le débit ρ d'une plate-forme revient à maximiser le nombre global de jobs exécutés en une unité de temps, c'est à dire le nombre de tâches T^{end} sur chaque nœud où T^{end} est la dernière tâche

dans le DAG.

Nous pouvons alors écrire un programme linéaire qui détermine le débit optimal d'une plate-forme et une application donnée. Le programme linéaire 3.1 donne une solution optimale pour le problème d'ordonnancement d'un nombre infini d'instances d'une même application. Ce programme linéaire doit être résolu dans le domaine des nombres rationnels pour exécuter les instances d'une application avec un débit optimal ρ. La première contrainte définit le ratio de temps passé par chaque nœud p_i à exécuter une tâche de type T^k. La deuxième contrainte s'assure que $\alpha(p_i, T^k)$ soit positif et que chaque nœud ne puisse pas passer plus de 100% de son temps sur une tâche de type T^k. La dernière contrainte s'assure qu'un nœud ne prenne pas prendre plus de 100% de son temps pour l'ensemble des tâches qu'il exécute.

$$\begin{array}{l} \text{maximiser } \rho = \sum_{p_i \in P} cons(p_i, T^{end}) \\ \text{sous les contraintes :} \\ \left\{ \begin{array}{lrcl} \forall p_i \in P, \forall T^k \in \tau, & \alpha(p_i, T^k) & = & cons(p_i, T^k) w_{i,k} \\ \forall p_i \in P, \forall T^k \in \tau, & 0 \leq \alpha(p_i, T^k) & \leq & 1 \\ \forall p_i \in P, & \sum_{T^k \in \tau} \alpha(p_i, T^k) & \leq & 1 \end{array} \right. \end{array} \tag{3.1}$$

Bien que la solution du programme linéaire soit rationnelle ($cons(p_i, T^k)$) il est possible de construire un ordonnancement valide à l'intérieur d'une période. La période L est le plus petit commun multiple de tous les dénominateurs de $cons(p_i, T^k)$. Donc en multipliant $cons(p_i, T^k)$ par L, la quantité de jobs à l'intérieur d'une période est un entier. Il est alors possible de calculer les allocations avec un algorithme qui s'exécute en temps polynomial pour des DAGs en forme d'anti-arbres. Dans le cas de DAGs de forme générale, le problème du calcul des allocations est NP-Difficile, c'est pourquoi nous limitons notre étude aux anti-arbres.

Il est possible d'adapter la technique à un nombre fini d'instances d'une même application. Avec cette adaptation de la technique, l'ordonnancement résultant est composé de trois phases, comme le montre la figure 3.5 sur un exemple d'application (Figure 3.4) : (1) une phase d'initialisation sous-optimale qui calcule les tâches nécessaires pour alimenter la phase optimale en régime permanent (2) ; la phase en régime permanent est suivie d'une phase sous-optimale de terminaison qui s'occupe des tâches restantes. Cet ordonnancement approche l'optimal quand la taille du workflow augmente, car le poids de la phase d'initialisation et de la phase de terminaison diminue par rapport à phase en régime permanent.

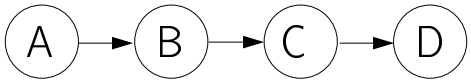


Figure 3.4 – Exemple d'application avec 4 tâches

3.2.4 Résultats expérimentaux

Diakité et al. ont développé un simulateur qui implémente l'algorithme du Liste, l'algorithme génétique et l'algorithme steady-state. Ce simulateur calcule un ordonnancement d'un workflow et simule l'exécution de ce workflow. Le simulateur est fondé sur la librairie Simgrid et son

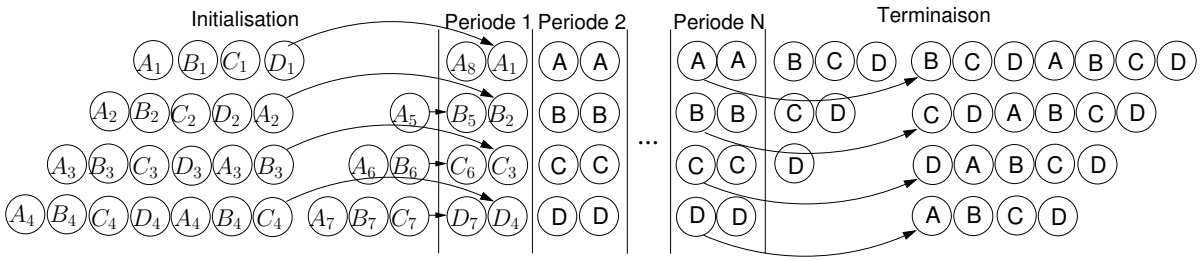


Figure 3.5 – Un ordonnancement avec le steady-state

API MSG [23, 22, 25] et suivant le paradigme maître/esclave. Un maître exécute l'un des trois algorithmes, calcule un ordonnancement puis envoie les tâches à exécuter aux esclaves en fonction de l'ordonnancement calculé. Pour chaque taille de workflow, Diakité et al. ont lancé une simulation avec les trois algorithmes et ont mesuré le temps simulé total pour exécuter un workflow, c'est à dire le makespan de l'ordonnancement calculé par chaque algorithme. Comme l'algorithme de Liste et l'algorithme du steady-state ont un comportement déterministe, une seule exécution de chacun des deux algorithmes suffit pour obtenir un résultat. Pour l'algorithme génétique, comme la solution dépend de valeurs aléatoires, ils ont réalisé une moyenne sur 10 exécutions.

a La Métrique

Pour évaluer l'algorithme génétique, nous introduisons la notion de rapport du makespan à l'optimal (RMO), qui est égal au quotient du makespan de l'ordonnancement calculé par un des algorithmes $makespan_r$ sur le makespan optimal $makespan_o$. Ainsi, si $makespan_r = makespan_o$, alors $RMO = 1$. Cela signifie que l'ordonnancement calculé a un makespan optimal. Plus la valeur RMO de l'algorithme génétique est proche de 1, meilleur est son ordonnancement en terme de makespan. Comme ce problème d'ordonnancement est NP-Difficile et que la taille du problème est importante, nous ne pouvons pas évaluer le makespan optimal $makespan_o$. Cependant, le nombre de DAGs n à ordonnancer divisé par le débit calculé par le steady-state (ρ) est une borne inférieure correcte. Ainsi nous avons :

$$makespan_o = \frac{n}{\rho}$$

et

$$RMO = \frac{makespan_o}{makespan_r}$$

donc on a

$$RMO = \frac{n}{\rho \times makespan_r}$$

Nous devons définir un indicateur qui montre la qualité des expériences pour un ensemble de résultats. C'est une question difficile. En effet une moyenne n'est pas nécessairement un bon indicateur car elle néglige la distribution. Comme l'algorithme steady-state trouve une solution quasi-optimale quand le nombre de d'instances augmente, la métrique utilisée est la proportion des expériences pour lesquelles l'algorithme est proche de l'optimal (90%).

b Mise en place des expériences

Des simulations ont été lancées avec des workflows de 1 à 5000 jobs (un job est une instance d'une application). Les simulations utilisent un jeu de données généré aléatoirement avant les simulations : 400 couples (plate-forme, application) sont générés à l'aide d'une loi uniforme. Les plates-formes sont composées de 4 à 10 nœuds. Pour chaque nœud nous calculons un sous ensemble de 10 types de tâches qui sont disponibles pour ce nœud durant les simulations. Ces types de tâches sont également sélectionnés suivant une loi uniforme. Quant aux applications, elles sont aussi générées aléatoirement suivant une loi uniforme. Elles ont la forme d'un anti-arbre et sont composées de 4 à 12 nœuds.

De plus, nous définissons un facteur d'hétérogénéité afin d'étudier l'impact de l'hétérogénéité de la plate-forme sur le comportement des algorithmes. Dans une plate-forme hétérogène, pour chaque nœud nous sélectionnons un facteur h aléatoirement entre 1 et le facteur d'hétérogénéité. Ensuite, ce facteur h est appliqué sur le temps d'exécution d'un type de tâche sur le nœud de la plate-forme homogène de référence. Ces temps d'exécution restent constant durant toute l'expérience.

c Plates-formes homogènes

Nous étudions dans un premier temps le comportement des trois algorithmes avec des plates-formes homogènes et un ensemble de 50 expériences. Dans une plate-forme homogène, le temps nécessaire pour exécuter un type de tâche donné est toujours le même quel que soit le nœud. La figure 3.6 montre la proportion des expériences pour lesquelles les algorithmes atteignent 90% du débit optimal, en fonction de la taille du workflow. Nous remarquons que l'algorithme génétique est le meilleur algorithme jusqu'à 500 jobs où il atteint 90% du débit optimal dans plus de 90% des cas. Pour les workflows de 500 jobs et plus, le steady-state réalise les meilleures performances : il atteint même les 90% du débit optimal dans 100% des cas à partir de 2500 jobs. L'algorithme de Liste donne de bonnes performances mais il n'est jamais le meilleur. Entre 1 et 200 jobs ses performances sont meilleures que celles du steady-state. Ses performances augmentent rapidement quand le nombre de jobs augmente, puis elles se stabilisent. Au contraire, le steady-state souffre de sa phase d'initialisation et de sa phase de terminaison, ainsi ses performances augmentent lentement avec le nombre de jobs. Quand le nombre de jobs atteint le seuil critique à partir duquel le nombre de jobs ordonnancés à l'intérieur des périodes optimales est supérieur au nombre de jobs ordonnancés à l'intérieur de la phase d'initialisation et la phase de terminaison, les performances du steady-state augmentent très rapidement avec la taille du workflow et finalement dans 100% des cas le rapport du makespan à l'optimal atteint plus de 90%.

d Plates-formes hétérogènes

Nous étudions maintenant le comportement des trois algorithmes avec des plates-formes hétérogènes et un facteur d'hétérogénéité de 10 et de 100. Les simulations ont été réalisées avec deux ensembles de 50 expériences. La figure 3.7 montre la proportion des expériences pour lesquels les algorithmes atteignent 90% du débit optimal, en fonction de la taille du workflow et avec un facteur d'hétérogénéité de 10. Nous remarquons que l'algorithme génétique est le

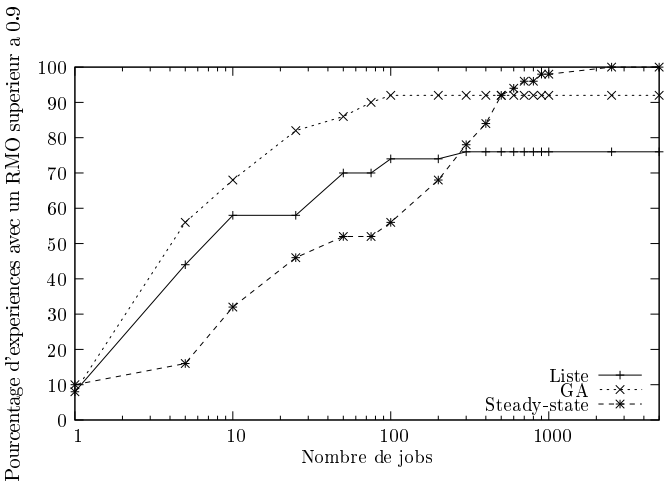


Figure 3.6 – Comparaison des trois algorithmes avec des plates-formes homogènes

meilleur algorithme et à partir de 700 jobs. Il atteint 90% du débit optimal pour plus de 96% des cas. Les deux autres algorithmes montrent des comportements similaires par rapport au cas homogène. Néanmoins, même si le steady-state est meilleur que l'algorithme de Liste à partir de 2500 jobs, il atteint 90% du débit optimal pour seulement 94% des cas avec un workflow de 5000 jobs. Il n'arrive pas à atteindre les performances de l'algorithme génétique. Avec un haut niveau d'hétérogénéité, le steady-state souffre d'un problème d'alignement lié au plus petit multiple commun dans le calcul de la taille de la période. Malheureusement, plus la période est grande, et plus les phases d'initialisation et de terminaison sont grandes.

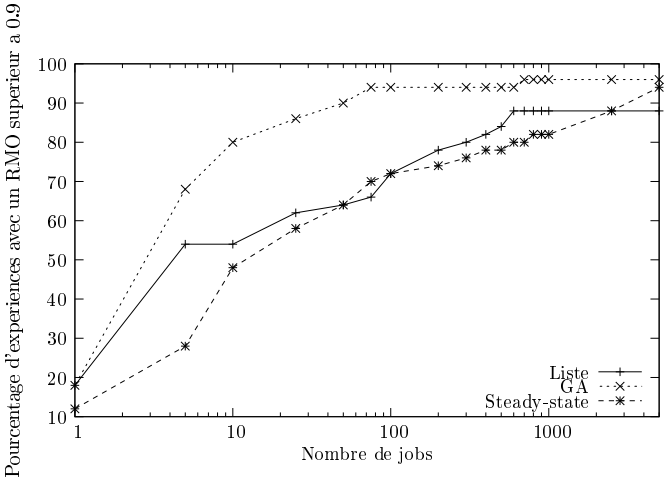


Figure 3.7 – Comparaison des trois algorithmes avec des plates-formes hétérogènes et un facteur d'hétérogénéité de 10

La figure 3.8 montre la proportion des expériences pour lesquels les algorithmes atteignent 90% du débit optimal, en fonction de la taille du workflow et avec un facteur d'hétérogénéité de 100. Nous remarquons que l'algorithme génétique est le meilleur algorithme et atteint 90% du débit optimal dans 100% des cas pour des workflows de plus de 100 jobs. Les performances du steady-state sont les moins bonnes, moins bonnes qu'avec un facteur d'hétérogénéité de 10.

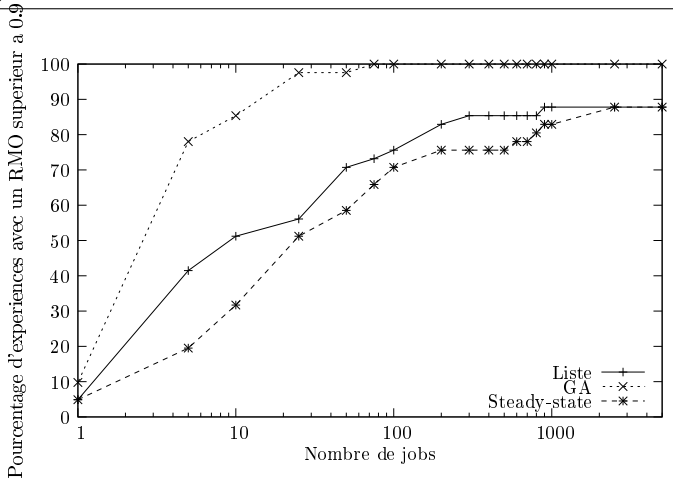


Figure 3.8 – Comparaison des trois algorithmes avec des plates-formes hétérogènes et un facteur d'hétérogénéité de 100

e Plates-formes hétérogènes avec un mélange de facteurs d'hétérogénéité

Nous étudions dans ce paragraphe le comportement des trois algorithmes avec des plates-formes hétérogènes et des facteurs d'hétérogénéité de 1, 10 et de 100. Les simulations ont été réalisées avec trois ensembles de 50 expériences. La figure 3.9 montre la proportion des expériences pour lesquels les algorithmes atteignent 90% du débit optimal, en fonction de la taille du workflow. Dans le cas général, l'algorithme génétique donne les meilleures performances. Pour des petites tailles de workflow, l'algorithme de Liste vient juste après l'algorithme génétique. Sinon, pour des grandes tailles de workflow, le steady-state est le deuxième algorithme.

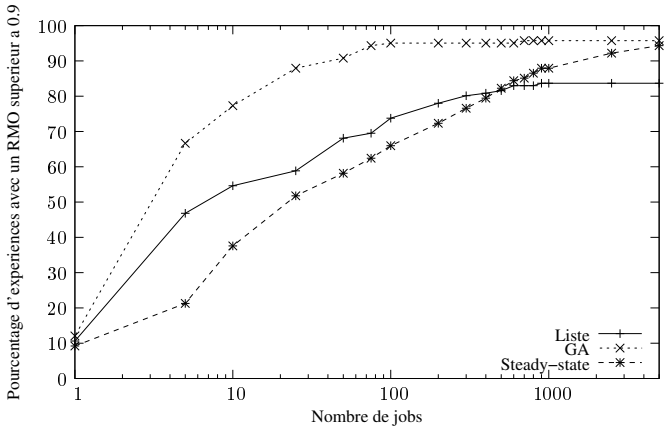
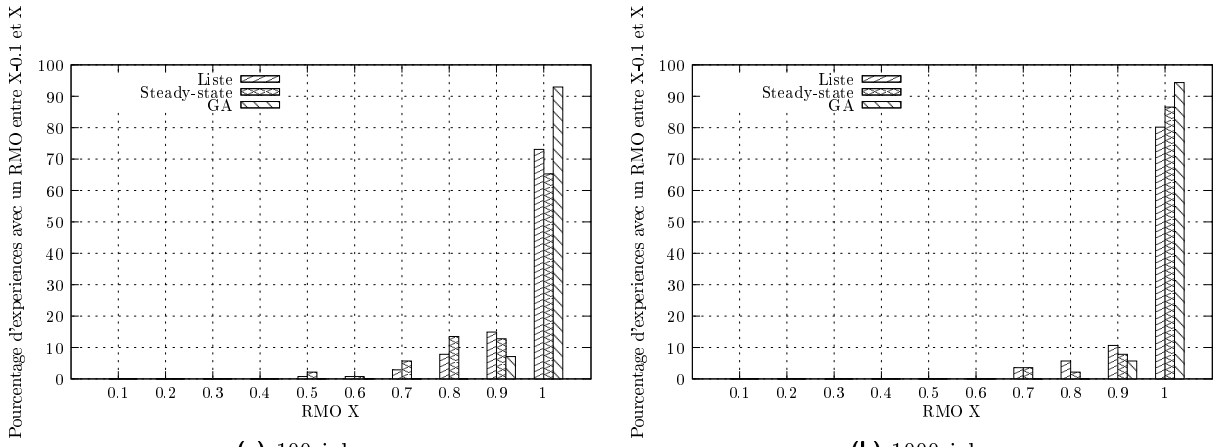


Figure 3.9 – Comparaison des trois algorithmes avec des plates-formes hétérogènes et un mélange de facteurs d'hétérogénéité

f Distribution du RMO

Dans les résultats précédents, la métrique utilisée est la proportion des expériences pour lesquelles un algorithme donné atteint un RMO de plus de 90%. L'inconvénient de cette métrique est qu'elle limite la vision aux seules expériences avec un RMO qui dépasse le seuil des 90%. Nous présentons dans cette section la distribution du RMO des trois algorithmes. La figure 3.10

montre la répartition du RMO pour des expériences avec 100 jobs (figure 3.10a) et 1000 jobs (figure 3.10b) avec des plates-formes et un mélange de facteurs d'hétérogénéité. Nous remarquons que pour les trois algorithmes la plupart des expériences donnent un RMO entre 0.9 et 1.0. Nous pouvons faire aussi deux remarques importantes. Premièrement, toutes les distributions sont globalement groupées vers la droite. Cela signifie que dans la majorité des cas, les algorithmes donnent des ordonnancements proches de l'optimale et d'après les deux histogrammes, aucune des expériences ne donne un RMO inférieur à 0.5. Deuxièmement, contrairement aux autres algorithmes, le steady-state ne montre pas une progression dans sa distribution : il y a plus d'expériences avec un RMO entre 0.7 et 0.8 qu'entre 0.8 et 0.9 avec 100 jobs. Cela peut être interprété par le fait que le steady-state dépend plus du contexte, c'est-à-dire que les caractéristiques de la plate-forme et du workflow ont un plus grand impact sur ses performances que sur celles des deux autres algorithmes.



(a) 100 jobs **(b)** 1000 jobs

Figure 3.10 – Répartition du RMO avec un mélange de facteurs d'hétérogénéité pour 100 jobs et 1000 jobs

g Impact des plates-formes et des workflows

Nous avons essayé d'identifier les causes des impacts des plates-formes et des workflows sur le RMO des différents algorithmes. Nous avons donc extrait des résultats de simulation, les deux configurations, c'est-à-dire les deux couples plate-forme/application, en faveur respectivement de l'algorithme génétique (Figures 3.11 et 3.12) et du steady-state (Figures 3.13 et 3.14). Pour toutes les expériences, nous n'avons pas pu trouver de configuration en faveur de l'algorithme de Liste.

L'algorithme génétique donne de bons résultats dans de nombreux cas, comme le montre la figure 3.9. Néanmoins ses performances ne sont pas si bonnes avec des applications complexes (grand nombre de tâches) et des grandes plates-formes hétérogènes pour lesquelles les nœuds peuvent exécuter plusieurs types de tâches avec différentes vitesses. La figure 3.13 montre le cas où les résultats de l'algorithme génétique sont 10% en dessous du steady-state à partir de 300 jobs comme le montre la figure 3.14. L'intérêt de l'algorithme génétique est sa capacité à fournir de bons ordonnancements même pour des petits workflows (Figures 3.11 et 3.12). Il convient de noter que les résultats présentés sur les courbes sont fondés sur la borne théorique du débit qui est calculée à partir du steady-state. Ainsi il est évident qu'aucun algorithme n'a

de bonnes performances avec un workflow de 10 jobs, puisque le nombre de tâches dans un job est trop petit pour qu'elles puissent être correctement distribuées sur les nœuds et atteindre un débit suffisant. Donc l'algorithme génétique est globalement très efficace (il trouve même des ordonnancements optimaux quand les configurations sont simples) pour des petits workflows.

Comme prévu, plus le nombre de job est grand et meilleurs sont les résultats du steady-state : quelles que soient la plate-forme et l'application, ses performances sont très bonnes avec un workflow de quelques milliers de job. Nous remarquons que son comportement est imprévisible pour des workflows inférieurs à 1000 jobs et ses performances sont médiocres pour ces tailles de workflow. Dans le cas où les configurations sont complexes, c'est-à-dire, des applications et des plates-formes complexes, il se comporte beaucoup mieux que l'algorithme génétique quand des workflows de plus de 1000 jobs sont ordonnancés.

P		p_1	p_2	p_3	p_4
Type	A	10	10	10	-
	B	-	-	-	10
	C	-	-	-	10
	D	10	-	-	-
	E	-	-	10	-
	F	-	10	-	-
	G	-	-	10	-
	H	10	-	-	-
	I	-	10	10	10
	J	-	-	-	10

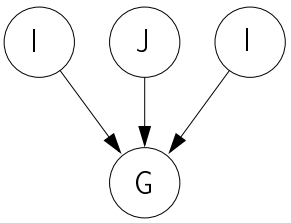


Figure 3.11 – Un couple plate-forme/application en faveur de l'algorithme génétique

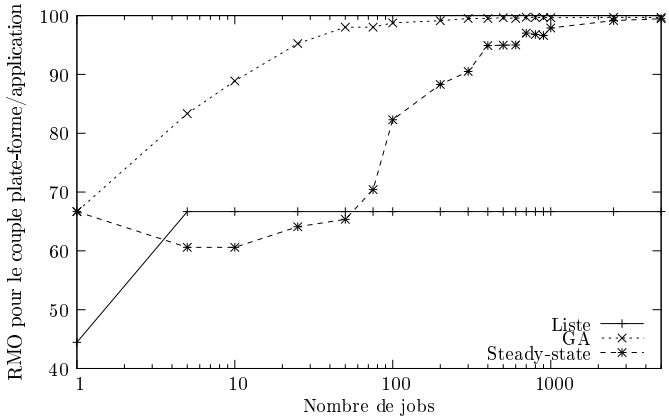


Figure 3.12 – RMO des algorithmes pour le couple plate-forme/application en faveur de l'algorithme génétique

h Temps de calculs et complexité des implémentations

La figure 3.15 montre le temps CPU nécessaire pour chaque algorithme pour calculer des ordonnancements de workflows de 1 à 5000 jobs. Pour des workflows de 5000 jobs, l'algorithme génétique met environ 1 heure pour calculer un ordonnancement, l'algorithme de Liste met 1 minute et le steady-state 1 seconde. Donc pour des grandes tailles de workflow, le steady-state est clairement la solution la moins coûteuse en terme de temps de calcul et il trouve de bons

P		p_1	p_2	p_3	p_4	p_5	p_6	p_7
Type	A	-	20	-	-	-	-	-
	B	50	-	20	-	100	-	-
	C	-	-	-	40	-	-	-
	D	-	-	50	-	-	-	-
	E	-	-	-	-	-	-	40
	F	-	20	50	-	-	-	-
	G	-	-	-	-	-	-	100
	H	-	-	20	-	-	-	50
	I	-	50	-	20	-	100	-
	J	-	-	-	70	-	70	80

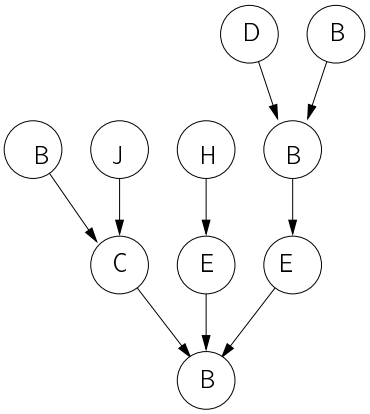


Figure 3.13 – Un couple plate-forme/application en faveur du steady-state

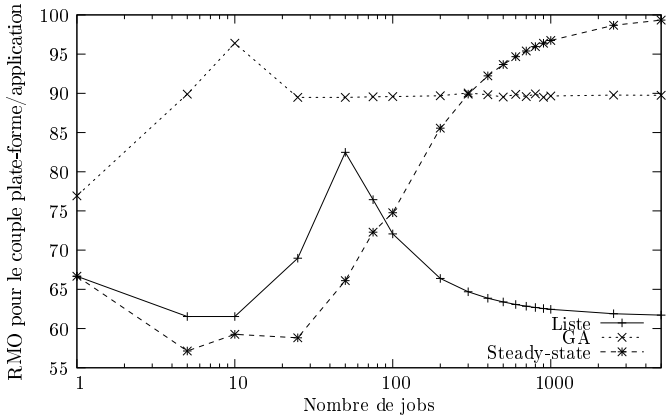


Figure 3.14 – RMO des algorithmes pour le couple plate-forme/application en faveur du steady-state

ordonnancements. Il convient de remarquer que nous avons choisi une échelle logarithmique pour le temps CPU et le nombre de jobs, donc le gradient pour l'algorithme de Liste est clairement linéaire et le gradient pour l'algorithme génétique est polynomial. Ces résultats sont cohérents avec la complexité des algorithmes. Pour le steady-state, la complexité est une constante $\mathcal{O}(1)$: dès que la période, les phases d'initialisation et de terminaisons sont calculées, nous pouvons ajouter autant de périodes que l'on souhaite sans ajouter aucun surcoût. Pour l'algorithme de Liste, la complexité est linéaire $\mathcal{O}(m)$: il dépend du nombre de jobs m et du nombre de nœuds n qui est constant. Pour l'algorithme génétique, la complexité dépend du nombre de tâches pour calculer les individus initiaux et le nombre de tâches pour appliquer les opérateurs génétiques, donc la complexité est aussi linéaire en $\mathcal{O}(m)$.

3.2.5 Analyse

En résumé, nous donnons quelques résultats généraux présentés dans ce chapitre. Comme l'algorithme génétique utilise un algorithme de Liste pour générer sa population initiale, il n'est pas surprenant que l'algorithme génétique fasse mieux que l'algorithme de Liste. Les amélioration varient entre 10% et 75% pour des workflows de plus de 10 jobs, et le meilleur gain

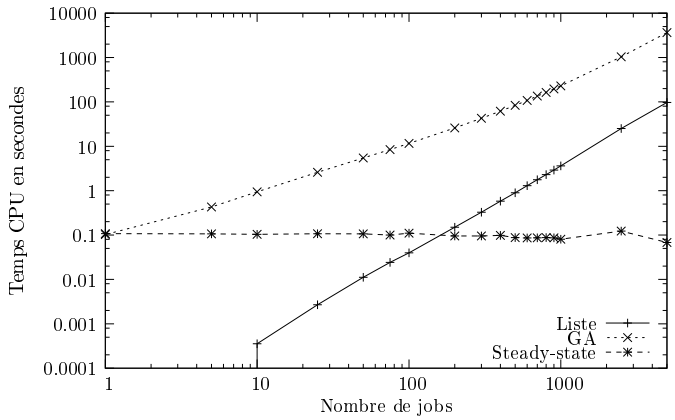


Figure 3.15 – Temps CPU moyen pour 150 expériences

est atteint pour des workflows entre 10 et quelques centaines de jobs.

Si nous comparons l'algorithme génétique avec l'algorithme du steady-state, nous remarquons que l'algorithme génétique donne de meilleurs résultats avec des plates-formes hétérogènes. La raison de cette observation, tient au fait que plus la plate-forme est hétérogène et plus la période du steady-state est grande. La taille de la période a un impact direct sur la taille de la phase d'initialisation et de la phase de terminaison. Comme ces phases sont sous optimales, l'algorithme du steady-state ne donne pas de bons résultats. Le steady-state devient meilleur que l'algorithme génétique quand le nombre de jobs augmente car les parties sous optimales de l'ordonnancement deviennent de moins en moins importantes par rapport à la partie optimale des périodes qui s'enchaînent. Le seuil à partir duquel le steady-state devient meilleur que l'algorithme génétique dépend de l'hétérogénéité de la plate-forme : quand les plates-formes sont homogènes, il est de l'ordre de 1000 de jobs et pour les plates-formes hétérogènes, de l'ordre de 5000 jobs.

La taille et la complexité de la plate-forme a aussi un impact sur les performances : des plates-formes et/ou des applications complexes sont plus difficiles à ordonnancer pour l'algorithme génétique, alors que l'algorithme du steady-state trouve toujours des ordonnancements optimaux. D'autre part, pour des plates-formes et/ou des applications plus simples, l'algorithme génétique donne aussi des solutions qui atteignent l'optimalité.

L'inconvénient de l'algorithme génétique est son coût de calcul pour trouver un ordonnancement qui peut être un problème pour des grosses plates-formes. En résumé, nous recommandons d'utiliser l'algorithme génétique quand la taille du workflow est inférieure à 1000 jobs et pour les couples de plates-formes/applications pas trop complexes, et d'utiliser l'algorithme du steady-state, quand la taille du workflow dépasse quelques milliers de jobs. Ensuite, entre ces deux valeurs (mille et quelques milliers), le choix dépend de l'hétérogénéité et de la complexité de la plate-forme.

Ainsi dans le cadre de cette thèse, nous reprenons les données issues des expériences réalisées par Sékou Diakité. Grâce à des statistiques réalisées sur ses données, nous apportons une contribution supplémentaire à ses travaux. Nous pouvons désormais déterminer quelles plates-formes et applications favorisent l'algorithme génétique et celles qui favorisent le steady-state. Pour ces données nous n'observons pas de cas où l'algorithme de Liste est favorisé. Grâce à la

distribution statistique du RMO, nous pouvons désormais affirmer que dans la plupart des cas les algorithmes donnent des ordonnancements proches de l'optimal.

L'intérêt de ce travail est double. Nous apportons d'une part des contributions au travail de Sékou Diakité et d'autre part il sert de base au travail de recherche décrit au paragraphe suivant.

3.3 Un algorithme génétique pour l'ordonnancement d'un workflow avec des coûts de communication

Après avoir étudié l'ordonnancement d'un workflow d'instances d'une même application sans tenir compte des coûts de communication nous étudions à présent le problème d'ordonnancement d'un workflow d'instances d'une même ou de différentes applications en intégrant le coût des communication. Le but est d'optimiser le makespan. Pour résoudre ce problème, nous proposons de reprendre l'algorithme génétique précédent et de le modifier afin qu'il prenne en compte les coûts de communication.

3.3.1 L'intégration des communications dans un algorithme génétique

Le premier problème auquel nous avions du faire face est l'intégration des communications dans l'algorithme génétique. Comme l'algorithme génétique représente un ordonnancement par un chromosome en affectant les tâches du DAG sur les nœuds de la plate-forme, nous avons introduit alors la notion de *nœud de communication* (figure 3.16) entre chaque couple de nœuds de calcul et la notion de *tâche de communication* entre une tâche parente et une tâche fille dans le graphe de tâches. Nous avons étudié plusieurs solutions pour réaliser l'intégration des communications dans l'algorithme génétique.

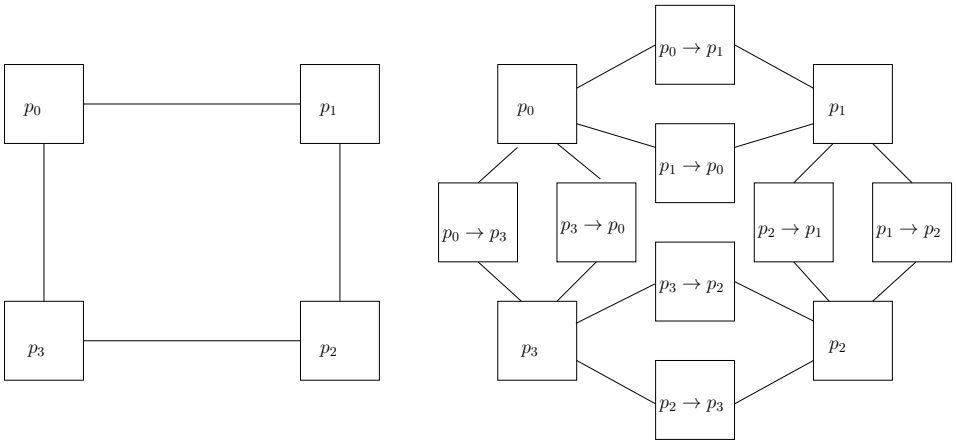


Figure 3.16 – Une plate-forme et son équivalent avec des nœuds de communication

D'abord nous pensions intégrer les communications à l'intérieur des chromosomes. Si nous intégrions les nœuds de communication dans les chromosomes, cela entraînerait des problèmes au niveau du croisement de deux chromosomes. Dans chaque ordonnancement une tâche est allouée à un nœud. Imaginons deux chromosomes représentant deux ordonnancements d'une application sur une plate-forme. Maintenant, appliquons un croisement de ces deux chromosomes selon la

méthode donnée par Daoud et al. [28] avec des nœuds de communication et des tâches de communication. Nous pouvons aisément imaginer que ce croisement génère dans la plupart des cas une route incohérente pour acheminer une les données d'une tâche à une autre. En effet, puisque la route dépend du choix des deux nœuds de calculs auxquels sont allouées une tâche de calcul parente et une tâche de calcul fille, le croisement peut générer une route qui n'existe pas.

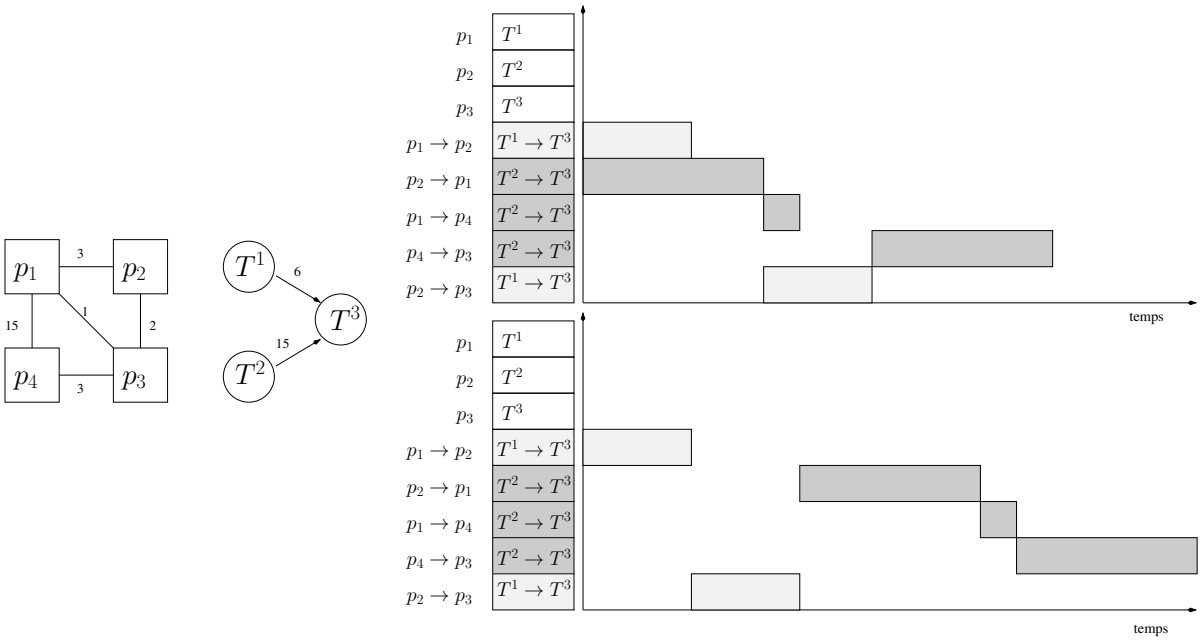


Figure 3.17 – Exemple d'ordonnancements d'une tâche de communication

Même si nous réussissions à implémenter un croisement pour les tâches de communication par rapport à des nœuds de communication, nous ne pouvons coder dans le chromosomes la date à laquelle débute une tâche de communication sur un nœud de communication et la date à laquelle elle finit. La figure 3.17 montre un exemple de deux ordonnancements d'une tâche de communication. En fait, cet exemple montre que pour un même codage d'une tâche de communication dans un chromosome, il correspond plusieurs ordonnancements possibles de cette tâche de communication. En effet la tâche de communication $T^2 \to T^3$ peut être ordonnancée de deux manières différentes.

Une manière commode d'introduire les communications au sein de l'algorithme génétique d'origine consiste à ignorer les communications dans le codage du chromosome et d'évaluer la performance du chromosome (fitness) en tenant compte des communications. Puisque le placement d'une tâche de communication dépend des placements des tâches de calculs parente/fille, intuitivement un bon placement de ces dernières peut implicitement donner un bon placement des tâches de communication.

3.3.2 Implémentations de l'intégration des communications dans l'algorithme génétique

Comme un algorithme génétique a un caractère stochastique, il est difficile d'intégrer des communications dans un algorithme génétique à partir de modèles. Nous avons donc, dans nos

recherches, essayé progressivement diverses solutions dont les implémentations sont simples et qui donnent d'assez bons résultats. Les figures 3.18a et 3.18b montrent le *RMO* des ordonnancements obtenus avec différentes implémentations de l'intégration des communications dans l'algorithme génétique.

Pour une première implémentation, nous définissons au préalable une route pour chaque couple de nœuds de calcul. Ces routes sont choisies aléatoirement et ne sont pas nécessairement les plus rapides. Dans cette première version, au niveau du décodage du chromosome, nous ne tenons compte ni des contentions à travers les routes ni du modèle 1-port. Nous avons réalisé des simulations mais pas plus de 10% des expériences donnent une efficacité supérieure à 0.9 (figure 3.18a).

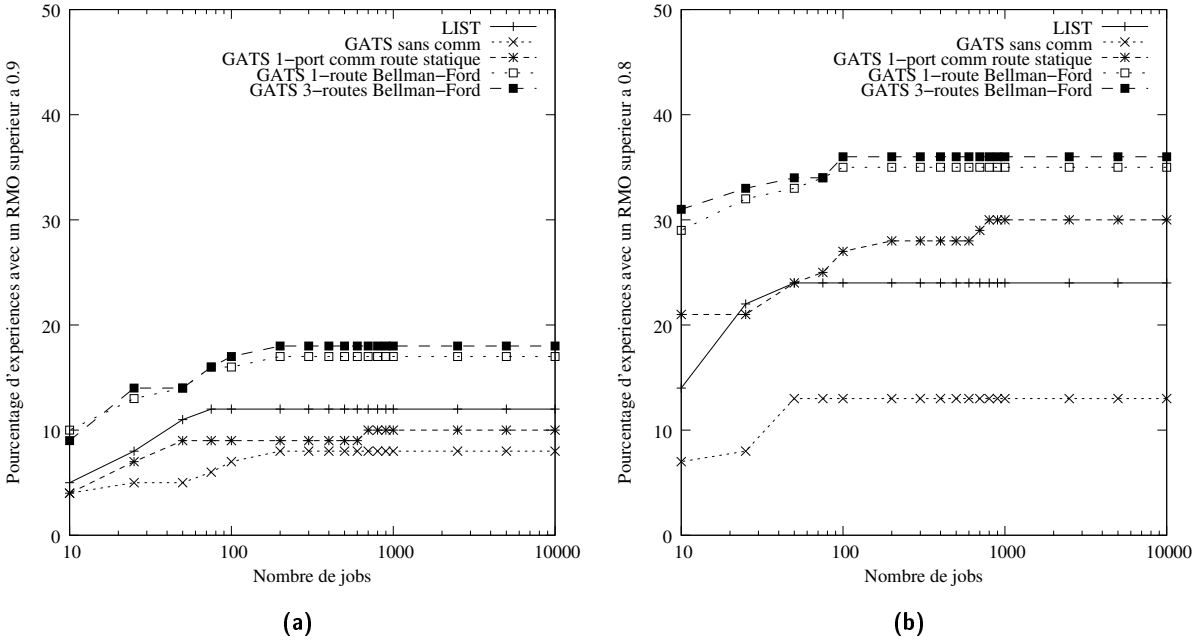


Figure 3.18 – Évolution des implémentations du GATS et pourcentage d'expérience avec une efficacité supérieure à 0.9 et 0.8 (quand les temps de calculs sont 10 fois plus importants que les temps de communication)

Ensuite nous avons réalisé une deuxième implémentation qui est une amélioration de la première. Cette implémentation tient compte des contentions à travers les routes mais en revanche elle ne respecte pas le modèle 1-port. Elle suit l'algorithme 12. La route entre chaque couple de nœuds de calcul est toujours statique, donc les routes générées ne sont pas nécessairement les plus rapides. Cette simple implémentation permet déjà d'obtenir de meilleures performances que l'algorithme de Liste (figure 3.18a et 3.18b).

L'implémentation suivante prend en compte à la fois les contentions à travers les routes ainsi que le modèle de communication 1-port. De plus, nous calculons la route la plus rapide entre chaque couple de nœuds de calcul, à l'aide de l'algorithme de plus court chemin de Bellman-Ford. Ce calcul est effectué une fois au début de l'algorithme. D'après la figure 3.18b l'amélioration par rapport à la version précédente est d'environ 5 à 10%.

Enfin nous avons réalisé une implémentation dans laquelle nous laissons le choix des 3

Algorithme 6: Évaluation du fitness d'un chromosome

Entrées : PF : la plate-forme et B : collection de workflows
Sorties : $f(ch)$: la fitness du chromosome ch avec les temps de communication
Données : $\mathcal{T}$: l'ensemble des tâches à ordonnancer,
$\mathcal{T}_{ToSched}$: l'ensemble des tâches restantes à ordonnancer,
$C(T_j^i)$: la date de fin de T_j^i,
$\sigma(T_j^i)$: la date de début de T_j^i sur $p_{a(i,j)}$,
$\delta(p_u)$: la date à laquelle p_u ne fait rien, $p_{a(i,j)}$ la machine sur laquelle T_j^i est assignée,
$w(T_j^i, p_i)$: la durée d'exécution de la tâche T_j^i sur p_i,
$CT(file_j^{k,i})$: le temps de communication pour envoyer $file_j^{k,i}$ à travers la route $\mathcal{R}(p_{a(k,j)}, p_{a(i,j)})$

$\mathcal{T}_{ToSched} \leftarrow \mathcal{T}$
tant que $\mathcal{T}_{ToSched} \neq \emptyset$ **faire**
  choisir une tâche libre de ses dépendances $T_j^i \in \mathcal{T}_{ToSched}$ (heuristique Earliest Finish Time)
  $\mathcal{T}_{pred} \leftarrow \{T_j^k | (T_j^k, T_j^i) \in \mathcal{D}_j\}$
  $\sigma(T_j^i) \leftarrow 0$
  pour chaque *tâche* $T_j^k \in \mathcal{T}_{pred}$ **faire**
    $\sigma(T_j^i) \leftarrow \max(\sigma(T_j^i), C(T_j^k) + CT(file_j^{k,i}))$
  $\sigma(T_j^i) \leftarrow \max(\delta(p_{a(i,j)}), \sigma(T_j^i))$
  $C(T_j^i) \leftarrow \sigma(T_j^i) + w(T_j^i, p_{a(i,j)})$
  $\delta(p_{a(i,j)}) \leftarrow C(T_j^i)$
  $\mathcal{T}_{ToSched} \leftarrow \mathcal{T}_{ToSched} \setminus \{T_j^i\}$
retourner $f(ch) = \frac{1}{C_{max}} = \frac{1}{\max\limits_{T_j^i \in \mathcal{T}}(C(T_j^i))}$

meilleures routes pour chaque couple de nœuds de calcul. En effet en effectuant un calcul des 3 plus court chemins avec l'algorithme de Bellman-Ford, nous espérions limiter la contention. Cependant les résultats (figure 3.18b) montrent que cette implémentation améliore très peu la précédente. On peut expliquer ce phénomène par le fait qu'au moment de l'évaluation du fitness du chromosome, l'algorithme fait uniquement un choix local d'une des trois routes, et ne tient pas compte des autres communications en cours.

3.3.3 Analyses expérimentales de l'ordonnancement d'un workflow d'instances d'une même application

Construire un modèle mathématique de l'algorithme génétique n'est pas réaliste dû à son caractère stochastique et la complexité du problème. De plus mener des expériences sur une grille réelle avec une implémentation de l'algorithme génétique ne permet pas d'obtenir des résultats reproductibles. C'est pourquoi nous avons implémenté un simulateur de grille pour évaluer les performances de l'algorithme génétique. Ce simulateur a été implémenté à l'aide de SimGrid et de son API MSG [23, 22, 25].

a La Métrique

Nous utilisons la même métrique qu'en section 3.2.4 : le rapport du makespan donné par l'algorithme par rapport à au makespan optimal (RMO).

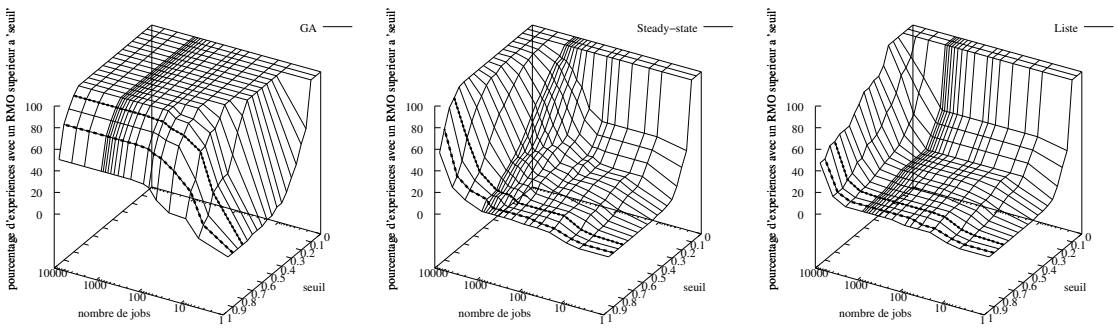
b Mise en place des expériences

Toutes les simulations ont été réalisées avec des tailles de de jobs allant de 1 à 10000. Les plates-formes et les applications sont générées aléatoirement à l'aide d'une loi uniforme. Les plates-formes ont entre 4 et 10 nœuds. Les applications ont entre 4 et 12 tâches. Dans le cas d'un workflow d'instances d'une même application, nous utilisons 10 plates-formes et 10 applications, donc nous avons $10 \times 10 = 100$ simulations pour chaque taille de workflow.

Nous utilisons la définition de "rapport des communications sur les calculs" (en anglais *communication to computation ratio* : CCR) d'une simulation qui est le rapport moyen des temps de communication sur les temps de calcul. Afin d'évaluer l'impact des communications sur les performances de l'algorithme génétique, nous lançons plusieurs simulations avec différents CCR, 1 (les temps de communications ont le même ordre de grandeur que les temps de calcul), 1/100 (les communications durent 100 fois moins longtemps que les calculs) et 100 (les communications durent 100 fois plus longtemps que les calculs). Les vitesses de calcul des nœuds sont indépendantes les unes des autres. Nous évaluons également l'impact de l'hétérogénéité de la plate-forme sur les performances : les temps d'exécutions varient de 1 à 10 et les temps de communication varient de 1 à 4. La population initiale de l'algorithme génétique est de 200 individus et l'algorithme génétique effectue 100 itérations.

Comme nous exécutons 100 simulations pour chaque taille de workflow, nous ne pouvons donner une valeur scalaire du RMO pour l'ensemble des expériences. Une valeur moyenne n'est pas suffisante pour avoir une idée réelle de la performance de l'algorithme. En effet les ordonnancements longs auront un plus grands poids que les plus petits. Donc nous calculons le pourcentage d'expériences qui ont un RMO plus grand qu'une valeur seuil t. Par la suite, nous présentons la distribution du RMO des expériences selon ce seuil t et la taille du workflow. Les deux lignes épaisses de la surface mettent en valeur la distribution du RMO des expériences avec comme valeur de seuil respectivement 0.8 et 0.9.

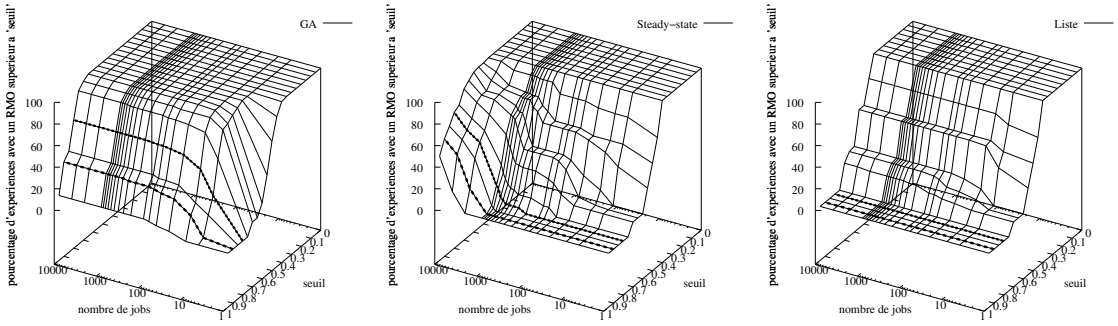
c Performances



(a) Algorithme Génétique **(b)** Steady-state **(c)** Liste

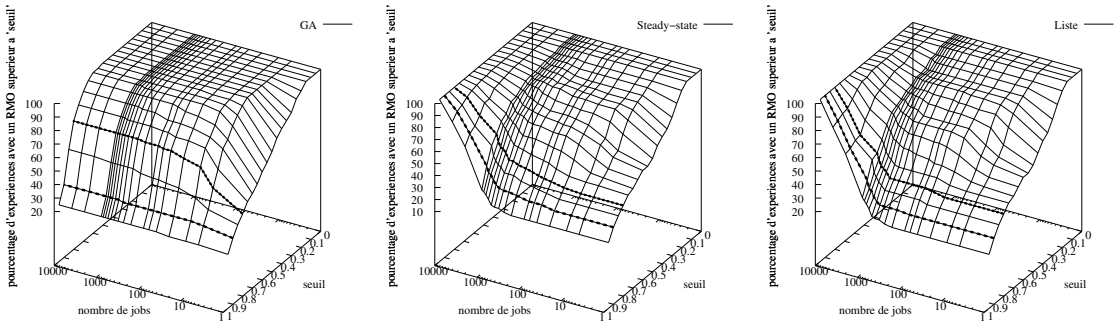
Figure 3.19 – Simulations avec des plates-formes homogènes, $CCR \approx 0.01$

Plates-formes homogènes Les figures 3.19a, 3.19b et 3.19c donnent les résultats avec des plates-formes homogènes (les nœuds de calcul ont les mêmes vitesses d'exécution et les bandes



(a) Algorithme Génétique **(b)** Steady-state **(c)** Liste

Figure 3.20 – Simulations avec des plates-formes homogènes, $CCR \approx 1$

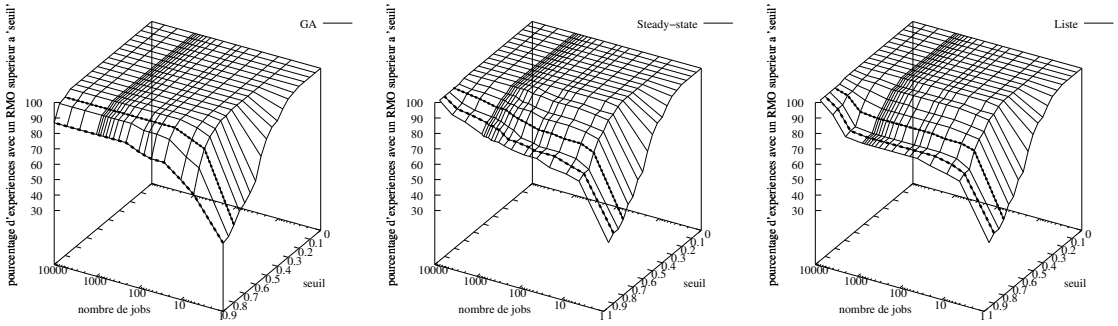


(a) Algorithme Génétique **(b)** Steady-state **(c)** Liste

Figure 3.21 – Simulations avec des plates-formes homogènes, $CCR \approx 100$

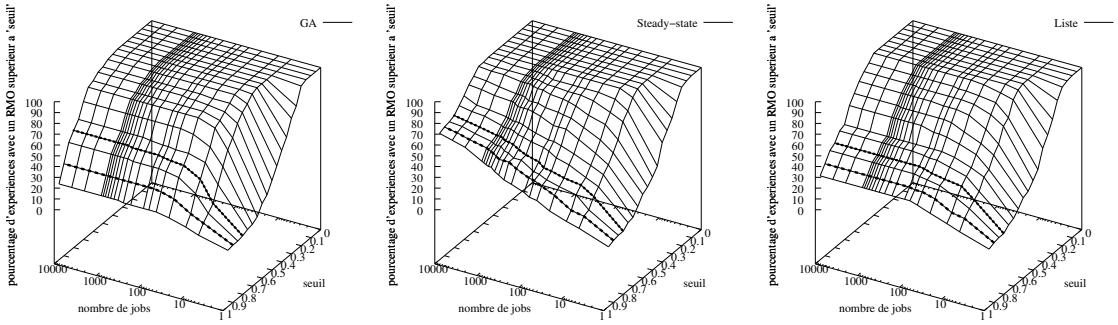
passantes des liens de communication sont identiques) et des temps de calculs des applications plus importants que les temps de communication ($CCR \approx 0.01$). Dans ce cas, l'algorithme génétique donne quasiment des résultats optimaux pour 85% des expériences. Les figures 3.20a, 3.20b et 3.20c montrent des résultats avec toujours des plates-formes homogènes mais les applications ont des temps de communication dont l'ordre de grandeur est le même que celui des temps de calcul ($CCR \approx 1$). Dans ce cas, l'algorithme génétique, donne les meilleurs résultats pour des workflows de taille inférieure à 2500 jobs puis il est dépassé par le Steady-state pour les workflows de plus de 10000 jobs. Il atteint néanmoins l'optimalité dans presque 70% des cas.

Les figures 3.19a, 3.19b et 3.19c donnent les résultats avec des plates-formes homogènes et les temps de communications des applications plus importants que les temps de calculs ($CCR \approx 100$). Nous pouvons noter que l'algorithme génétique est moins bon que les deux autres algorithmes pour ordonnancer plus de 10000 jobs. De ces expériences, nous pouvons conclure que l'algorithme génétique a globalement des performances meilleures que le Liste. Cependant, les résultats sont moins bons lorsque l'on introduit les communications car elles complexifient le problème.



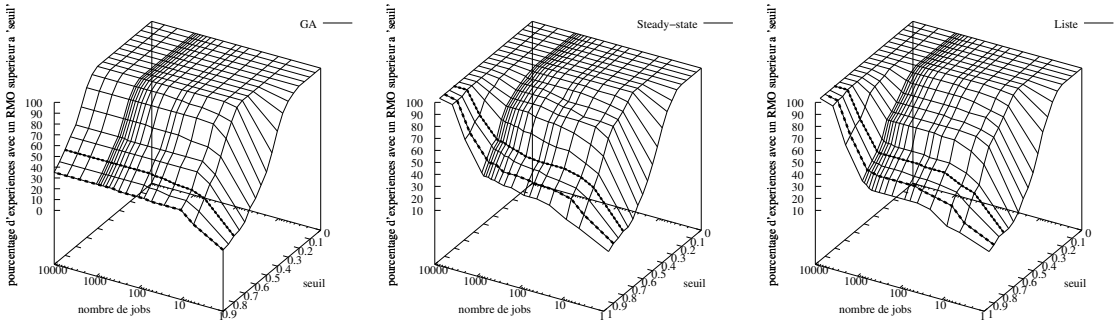
(a) Algorithme Génétique **(b)** Steady-state **(c)** Liste

Figure 3.22 – Simulation sur plates-formes hétérogènes, $CCR \approx 0.01$



(a) Algorithme Génétique **(b)** Steady-state **(c)** Liste

Figure 3.23 – Simulation sur plates-formes hétérogènes, $CCR \approx 1$



(a) Algorithme Génétique **(b)** Steady-state **(c)** Liste

Figure 3.24 – Simulation sur plates-formes hétérogènes, $CCR \approx 100$

Plates-formes hétérogènes Les figures 3.22a, 3.22b et 3.22c montrent les résultats obtenus avec des plates-formes hétérogènes et un $CCR \approx 0.01$. Les figures 3.23a, 3.23b et 3.23c montrent les résultats avec $CCR \approx 1$. Les figures 3.24a, 3.24b et 3.24c montrent les résultats obtenus avec des plates-formes hétérogènes et un $CCR \approx 100$. Dans le cas où les communications et les calculs ont des poids équivalents, $CCR \approx 1$, aucun algorithme ne donne de bons résultats du fait de la complexité du problème. Le Steady-state tend vers l'optimalité. Néanmoins l'algorithme génétique donne les meilleurs résultats pour des workflows de tailles inférieures à 2000 jobs. Dans le cas où les calculs sont plus intenses (c-à-d., $CCR \approx 0.01$), l'algorithme génétique est le meilleur et ses performances ne sont pas différentes du cas où la plate-forme est homogène. Nous pouvons noter que lorsque les coûts de communications sont dominants ($CCR \approx 100$) par rapport aux calculs, les algorithmes de Liste et du Steady-state atteignent l'optimal alors que l'algorithme génétique atteint son maximum à partir d'une cinquantaine de jobs, ce qui est très loin de l'optimal. l'algorithme génétique souffre des routes statiques précalculées et ne prend pas à compte la dynamicité des communications, ce qui explique ses faibles performances quand le poids des communications est plus élevé que celui des calculs.

d Les temps de calculs

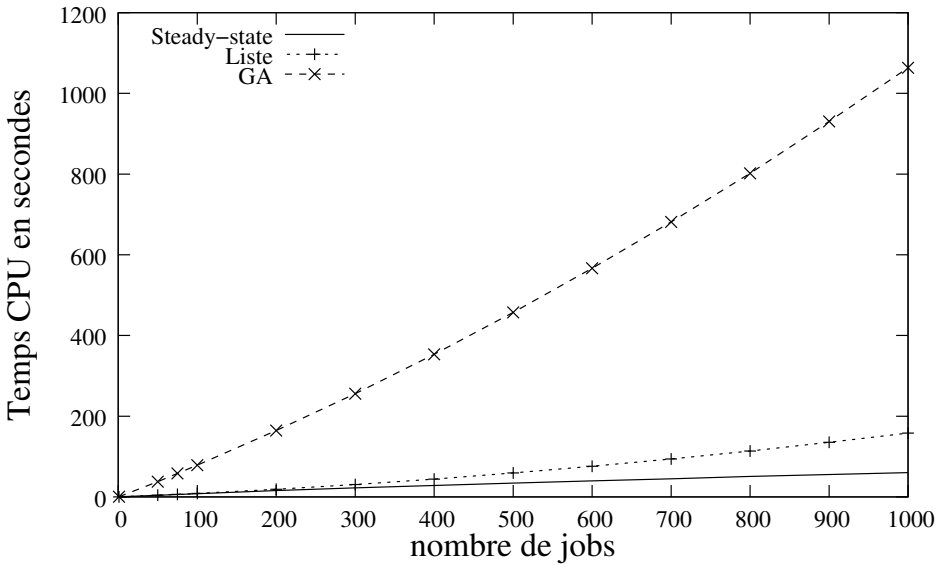


Figure 3.25 – Temps de calculs

Les simulations ont été exécutées sur des bi-processeurs 2.8 GHz Intel Xeon à quatre cœurs. La figure 3.25 montre les temps de calcul des trois algorithmes. Le temps nécessaire pour calculer un ordonnancement par l'algorithme génétique varie linéairement avec la taille de workflow : environ 1 seconde par DAG. Le temps nécessaire pour le Liste varie linéairement aussi. Il est environ 5 fois plus rapide que l'algorithme génétique. Finalement le temps d'exécution du Steady-state est très bas quelque soit la taille du workflow : environ 75 secondes pour ordonnancer 1000 DAGs. Donc l'algorithme génétique et l'algorithme de Liste sont très gourmands en temps de calcul alors que l'algorithme Steady-state est très rapide. Néanmoins, pour de grand nombre de jobs, l'algorithme génétique prend généralement moins de 20 minutes.

3.3.4 Résultats avec des DAGs généraux

Dans le cas de DAGs de forme générale, le problème du calcul des allocations dans l'algorithme du steady-state est NP-difficile. On ne peut donc pas utiliser l'algorithme du Steady-state pour ordonnancer un workflow d'instances d'une ou de plusieurs applications décrites par un DAG de forme générale. Au contraire, l'algorithme génétique et l'algorithme de Liste peuvent ordonnancer un workflow dans lequel les instances proviennent d'une ou de plusieurs applications décrites par un DAG de forme générale. C'est pourquoi nous limitons notre étude sur l'ordonnancement d'un tel workflow au comportement de l'algorithme génétique par rapport à celui de l'algorithme de Liste.

a La Métrique

À cause de la forme des DAGs il n'est plus possible de se comparer à l'optimale en utilisant l'algorithme du Steady-state et d'utiliser le RMO comme métrique. Comme nous comparons seulement deux algorithmes, nous évaluons pour chaque taille de workflow le gain apporté par l'algorithme génétique par rapport à l'algorithme de Liste.

b Mise en place des expériences

Toutes les simulations ont été réalisées avec des tailles d'ensemble de jobs allant de 1 à 10000. Les plates-formes et les applications sont générées aléatoirement à l'aide d'une loi uniforme. Les plates-formes ont entre 4 et 10 nœuds et sont hétérogènes et $CCR \approx 1$. Les applications ont entre 4 et 12 tâches. Les applications sont différentes les unes des autres ; 200 plates-formes et 10000 DAGs sont générés aléatoirement. Pour chaque couple (plate-forme, taille de workflow) différentes applications sont choisies aléatoirement parmi les 10000 DAGs. Ainsi, nous générons 1900 couples (plate-forme, application), donc 1900 simulations.

c Résultats expérimentaux et analyses

La figure 3.26 montre l'amélioration du makespan pour un workflow d'instances de plusieurs applications décrites par un DAG général sur une plate-forme hétérogène. Chaque courbe montre le pourcentage d'expériences de l'algorithme génétique qui obtiennent une amélioration du makespan respectivement de plus 0%, 10%, 20% et 30% en comparaison avec l'algorithme de Liste. Nous remarquons que plus de 80% des expériences de l'algorithme génétique donne un makespan qui s'améliore de plus de 10% par rapport à l'algorithme de Liste. La courbe 0% montre que l'algorithme génétique ne dégrade pas ou très peu les performances par rapport à l'algorithme de Liste.

La figure 3.27 montre l'amélioration du makespan pour un workflow d'instances d'une même application décrite par un DAG général sur une plate-forme hétérogène. Nous remarquons dans ce cas, que l'algorithme génétique apporte des améliorations moins importantes par rapport au Liste. Pour chaque courbe 0, 10, 20 et 30%, les améliorations de l'algorithme génétique baissent de 10% en comparaison avec les résultats des expériences obtenues en ordonnançant des instances de plusieurs applications. Nous remarquons aussi que dans 10% à 20% des expériences l'algorithme génétique donnent un ordonnancement moins efficace que le Liste.

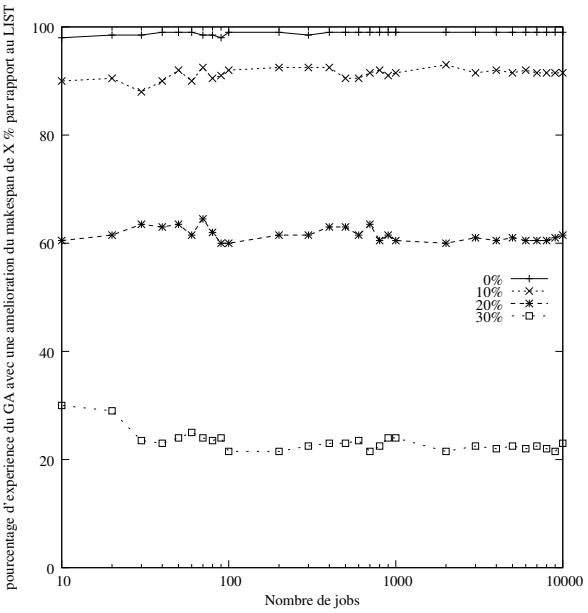


Figure 3.26 – Amélioration avec des instances de plusieurs applications décrites par un DAG général

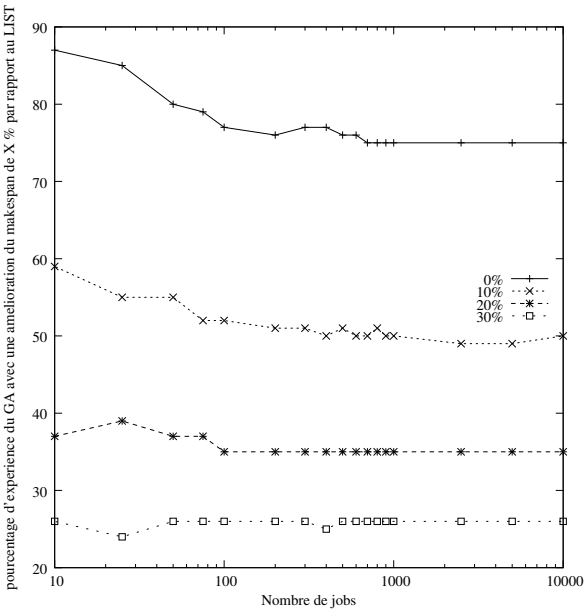


Figure 3.27 – Amélioration avec des instances d'une même application décrite par un DAG général

En général plus de 70% des expériences sur l'algorithme génétique donnent un makespan avec une amélioration d'au moins 10% pour n'importe quel taille de workflow. La diversité des applications permet de promouvoir l'algorithme génétique devant le Liste : comme l'algorithme génétique essaie aléatoirement de nombreuses combinaisons, il peut tirer parti des instances d'applications différentes alors que le Liste qui est dirigé par un choix glouton a plus de difficulté.

3.3.5 Performance de l'algorithme génétique pour ordonnancer un seul DAG

L'algorithme génétique de Daoud et al. n'est conçu que pour ordonnancer un DAG sur une plate-forme hétérogène sans communication. Avec l'introduction des communications, nous vérifions qu'il peut toujours ordonnancer un seul DAG en étant meilleur que l'algorithme de Liste. Nous considérons ici l'ordonnancement d'un seul DAG. Dans ce contexte, nous utilisons la métrique *longueur normalisée d'ordonnancement* (en anglais *Normalized Schedule Length* ou *NSL*) [6] afin d'évaluer les performances de l'algorithme génétique par rapport à l'algorithme de Liste.

Le NSL pour un DAG donné D est défini comme suit :

$$NSL(D) = \frac{makespan}{\sum_{t \in CP(D)} c_{t,a}}$$

avec CP le chemin critique du DAG D, $c_{t,a}$ le temps d'exécution de la tâche t sur la machine a le plus rapide qui sait réaliser t. Comme le chemin critique est une borne inférieure au makespan, le NSL représente un ratio du makespan une borne inférieure.

Les tableaux 3.2 et 3.3 font une comparaison des performances de l'algorithme génétique pour ordonnancer un DAG, en terme de NSL, sur une plate-forme hétérogène pour laquelle les temps de calcul et les temps de communication sont du même ordre de grandeur. On s'aperçoit

Nombre de tâches du DAG (anti-arbre)	Liste	GA
10	3.49	3.12
20	4.18	3.19
40	4.50	3.43
60	5.89	3.92
80	8.58	5.07
100	8.57	5.17

Table 3.2 – Comparaison du NSL moyen entre l'algorithme génétique et le Liste pour les anti-arbres, la plate-forme est hétérogène et $CCR \approx 1$.

Nombre de tâches du DAG général	Liste	GA
10	5.60	4.39
20	7.11	5.20
40	9.07	5.71
60	12.37	7.16
80	18.67	9.95
100	16.06	9.08

Table 3.3 – Comparaison du NSL moyen entre l'algorithme génétique et le Liste pour les DAGs généraux, la plate-forme est hétérogène et $CCR \approx 1$.

que l'algorithme génétique est plus performant que l'algorithme de Liste. En outre, plus le nombre de tâches du DAG est important et plus l'écart est marqué.

3.3.6 Synthèse

Dans ce chapitre nous avons proposé un algorithme génétique qui résout le problème d'ordonnancement d'un workflow sur un ensemble de nœuds connectés par des liens de communication. Nous avons repris l'algorithme génétique de Diakité el al. pour intégrer les communications. Nous montrons que pour un workflow d'instances de plusieurs applications, l'algorithme génétique obtient de meilleures performances que le classique algorithme de Liste. Dans le cas où les instances proviennent d'une même application décrite par un anti-arbre nous pouvons comparer les résultats de cet algorithme avec une borne inférieure du makespan optimal et montrer que les résultats tendent vers l'optimalité pour des lots de plus de 1000 jobs. De plus les résultats obtenus sont comparables à ceux de l'implémentation du steady-state en pratique. Cependant, nous retiendrons que le problème d'ordonnancement d'un workflow avec l'algorithme génétique prenant en considération des coûts de communication est très difficile. Dans ce contexte, le problème est également difficile pour l'adaptation de la technique du steady-state.

Chapitre 4

Virtualisation et pliage de jobs online

Au chapitre précédent nous nous sommes intéressés à l'ordonnancement d'applications sur des grilles de calculs par nature hétérogènes. Maintenant nous nous attaquons au problème d'ordonnancement d'applications séquentielles ou parallèles sur un cluster.

Dans un cluster homogène chaque nœud est constitué de processeurs identiques qui sont eux-même constitués de cœurs identiques. Les politiques d'ordonnancement utilisé dans la plupart des clusters homogènes ne tiennent pas compte de l'exacte répartition géographique des cœurs alloués à un job parallèle. Pour cette raison, par la suite, nous ne considérons que le nombre de cœurs alloués qui sont assimilables à des unités de calcul que l'on notera *PEs* (*Processing Elements*) [43].

De manière plus générale dans un ordinateur muni de N PEs, si N processus sont lancés par l'utilisateur, le système d'exploitation utilisent ces N PEs pour les exécuter. Bien qu'il y ait des changements de contexte à cause des processus systèmes, globalement chacun de ces processus s'exécute sur son propre PEs. En revanche si plus de N processus sont lancés, le système d'exploitation est obligé de partager des PEs entre les processus.

En particulier dans un cluster de calcul, en général, l'ordonnanceur de jobs ne lance pas plus de jobs séquentiels que de PEs. Quant au jobs parallèles, comme les jobs MPI, ils sont constitués de processus assimilables à des jobs séquentiels. Chacun de ses processus s'exécute sur un PE et par conséquent globalement chaque job parallèle s'exécute sur son propre ensemble de PEs. Il n'y a pas de partage de PEs entre les jobs.

Avant l'exécution d'un programme MPI, l'utilisateur indique le nombre n de PEs sur lequel il souhaite réaliser ses calculs. Au lancement d'une telle application, le processus parent se duplique en $n-1$ processus fils. Ces processus exécutent la même portion de code sur des données différentes. Typiquement dans un tel programme, les processus se synchronisent à un point de rendez-vous pour exécuter la suite du programme. Les processus peuvent aussi communiquer entre eux, par exemple, quand l'un a besoin des résultats de calcul d'un autre. Ce sont ces types d'applications parallèles que nous considérons dans ce chapitre.

Dans ce chapitre nous abordons une nouvelle technique d'ordonnancement appelée *pliage de jobs* associée à la virtualisation des PEs et visant à améliorer l'ordonnancement classique

des jobs dans le contexte des clusters. En général un cluster exécute un job parallèle sur un nombre de PEs fixé par l'utilisateur avant la soumission du job. L'idée générale du pliage de jobs consiste à exécuter un job parallèle avec moins de PEs physiques mais sur le bon nombre de PEs virtuels. L'exécution est plus longue que prévue car les PEs virtuels se partagent les PEs physiques mais cette technique permet au job de commencer son exécution plus tôt et finalement de finir plus tôt.

Notre objectif est l'optimisation de l'utilisation des ressources. Contrairement à d'autres travaux, nous ne permettons ni la migration de jobs, ni la suspension/reprise de jobs (préemption). Dans cette étude nous réalisons des simulations afin d'évaluer le gain de performances apporté par la virtualisation. À l'heure actuelle la gestion de la virtualisation dans les versions des systèmes d'exploitation de la majorité des clusters qui l'utilisent, comme CentOs [1], n'est pas encore suffisante pour implémenter nos algorithmes. Dans les prochaines versions, nous pourrons sûrement implémenter un vrai ordonnanceur de jobs avec gestion de la virtualisation.

Un travail préliminaire sur la virtualisation a été réalisé et les expériences avec KVM [3] ont montré que le surcoût lié à l'exécution sur des PEs virtuels était limité à 5%. Nous avons également observé que, quand deux tâches sont exécutées sur le même PE physique mais sur deux machines virtuelles différentes, ce partage apporte des effets positifs et négatifs. Le point positif est le recouvrement du temps du PE provoqué par les entrées/sorties. En effet les processus d'une application parallèle communiquent entre eux, et se synchronisent. Cette synchronisation engendre des cycles inutilisés dans un des PEs physiques. Si ce dernier est partagé avec un deuxième job grâce à la virtualisation de deux PEs virtuels sur ce PE physique, ce job peut utiliser ces cycles pour effectuer des calculs par exemple (c'est ce qui se passe en faisant du temps partagé). Le point négatif est le fait qu'il y aura des changements de contexte du cache du PE. Le gain apporté par le recouvrement peut être très important par rapport aux pertes des changements de contexte, ou bien les pertes de changement de contexte sont plus importantes que le gain apporté par le recouvrement. C'est pourquoi dans notre étude nous supposons que ces effets positifs et négatifs s'annulent. Ainsi l'idée du pliage de jobs est fondé sur le modèle des jobs moldables sans pénalité.

4.1 Le modèle du pliage de jobs

Dans la littérature, des travaux sur le pliage de jobs existent. Nous pouvons citer le lemme de Brent [17] et en donner une brève description. Dans le modèle PRAM [1], si une application parallèle n'écessite t unités de temps et est composée de q opérations - chacune dure 1 ut - alors cette application peut être exécutée avec m PEs et dure au plus $t + \frac{q-t}{m}\ ut$. Ce lemme permet aussi de démontrer que si une heuristique d'ordonnancement fonctionne avec m PEs et a une garantie ρ, alors cette même heuristique fonctionnant sur m' PEs, tels que $m' < m$, a une garantie $\rho + 1$.

Comme nous pouvons le constater, avec le lemme de Brent, seuls la durée initiale de l'application parallèle et le nombre d'opérations qui la constitue sont des données d'entrée, alors que dans notre modèle les applications parallèles sont des boîtes noires dont on ne connaît pas le nombre d'opérations.

[1] Parallel Random Access Machine

Dans notre problème, puisqu'un job est caractérisé principalement par sa durée et son nombre de PEs physiques requis, nous représentons graphiquement un job sous la forme d'une surface rectangulaire dont les deux dimensions sont le nombre de PEs physiques requis et la durée d'exécution associée. Cette surface représente sa géométrie.

Dans notre travail nous supposons qu'en changeant le nombre de PEs physiques sur lesquels le job s'exécute nous modifions son temps d'exécution d'origine. Par exemple, si un job requiert 10 PEs pendant 100 *ut* (unité de temps) et si l'ordonnanceur ne lui donne que 5 PEs, alors le temps requis sera multiplié par 2, c'est-à-dire 200 *ut*. Intuitivement cette hypothèse se vérifie : si l'ordonnanceur donne deux fois moins de PEs à un job que ce qu'il a demandé, chaque PE aura deux fois plus de travail et donc le temps d'exécution sera multiplié par 2. Quand nous exécutons un job parallèle sur des PEs virtuels nous changeons sa géométrie car nous modifions ses deux dimensions.

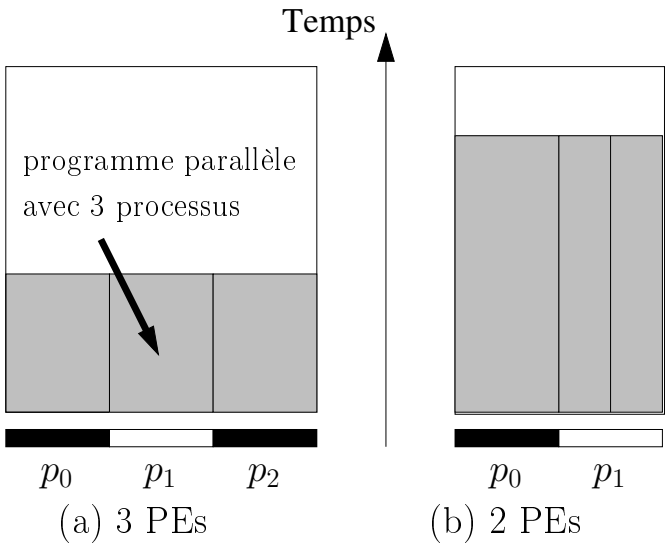


Figure 4.1 – Le principe du pliage de jobs

Néanmoins le temps d'exécution n'est pas une fonction linéaire du nombre de PEs alloués. Que se passe-t-il si le nombre de PEs est divisé par un nombre rationnel, par exemple $\frac{3}{2}$? La Figure 4.1(a) montre une exécution d'un job sur trois PEs alors que la figure 4.1(b) considère l'exécution du même job sur deux PEs seulement. Comme chaque processus est séquentiel, deux des trois processus doivent s'exécuter sur un PE et le troisième sur le second PE : p_0 exécute un seul processus et p_1 doit traiter deux processus. Ainsi la durée d'exécution est imposée par le PE qui a la plus grande charge de travail. De plus, à cause des synchronisations internes, par exemple les barrières de synchronisation, les passages de messages, dans l'application parallèle, des cycles inutilisés sont introduits dans p_0 qui exécute un seul processus et la géométrie du job reste une surface rectangulaire.

Nous posons l'hypothèse de travail suivante : si un job demande à s'exécuter durant t secondes avec n PEs, il durera $t \times \lceil \frac{m}{m'} \rceil$ secondes avec m' PEs, $m' < m$. Cette formule nous amène à la notion fondamentale de tableau d'allocations associé à un job. Une allocation est le nombre de PEs attribués à un job. Ainsi chaque job présente au plus m choix possibles d'allocation.

Le tableau 4.1 montre un exemple de tableau d'allocations pour un job qui demande au

départ 6 PEs et qui dure 10 *ut*. On dit qu'un pliage d'un job est entier quand le nombre de PEs alloué au départ à un job est divisé par un nombre entier, sinon on parle de pliage non entier. Nous remarquons qu'avec 5 et 4 PEs la surface est plus grande qu'avec 1, 2, 3 et 6 PEs. De plus avec 5 PEs la surface est plus grande qu'avec 3 PEs pour la même durée d'exécution. Cet exemple montre que pour la même durée d'exécution il est préférable de choisir l'allocation avec le plus petit nombre de PEs.

Table 4.1 – Exemple d'allocations pour un job

nombre de PEs	temps d'exécution	surface
1	60	60
2	30	60
3	20	60
4	20	80
5	20	100
6	**10**	**60**

Contrairement aux travaux réalisés dans le contexte des jobs moldables [89] ou malleables [75], nous ne considérons pas le cas où l'ordonnanceur donnerait plus de PEs que ce que demande l'utilisateur puisque nous n'avons pas accès au contenu des jobs et nous ne le modifions pas. Ainsi les jobs peuvent être exécutés avec moins de ressources que prévu grâce à la virtualisation.

4.2 Illustration du pliage

À travers la figure 4.2, nous montrons un exemple illustrant le pliage de jobs et ses possibilités. Dans cet exemple, nous considérons 5 jobs (A à E) et un cluster à 3 PEs. Les jobs A, B et C sont soumis à la date $t = 0$ alors que les jobs D et E sont soumis respectivement à la date $t = 1$ et $t = 4$. Ces jobs requièrent 1 PE et 7 *ut* pour le job A, 1 PE et 2 *ut* pour le job B, 2 PEs et 2 *ut* pour le job C, 3 PEs et 1 *ut* pour le job D et 2 PEs et 2 *ut* pour le job E. On considère un ordonnanceur FCFS et le makespan de l'ordonnancement des jobs A à E est de 11 *ut* (figure 4.2(a)). En utilisant le pliage de jobs (figure 4.2(b)), on réorganise l'ordonnancement : le job C s'exécute sur 2 PEs virtuels sur le même PE physique, ce qui ne lui permet pas de finir plus tôt mais de libérer le PE p_2. Ainsi, le job D exécute ses 3 processus sur le PE p_2 à la date $t = 2$ (au lieu de $t = 8$ avec l'ordonnanceur FCFS) et termine à $t = 5$ (au lieu de $t = 9$). Finalement le job E peut s'exécuter à $t = 5$ avec son nombre maximal de PEs sans augmenter sa durée. Le makespan avec l'ordonnanceur qui intègre le pliage de jobs est de 8 *ut*.

Cet exemple met en avant le fait qu'utiliser le pliage de jobs c'est-à-dire utiliser plusieurs PEs virtuels sur un PE physique peut améliorer l'ordonnancement de jobs en exploitant les cycles inutilisés des PEs.

4.3 Formulation du problème

Notre problème est l'optimisation du makespan de jobs pliables qui sont soumis au fil de l'eau. Un job est caractérisé par un nombre des PEs $reqproc_i$ et une durée d'exécution $reqtime_i(reqproc_i)$. Comme nous venons de le voir, chaque job i possède un tableau d'alloca-

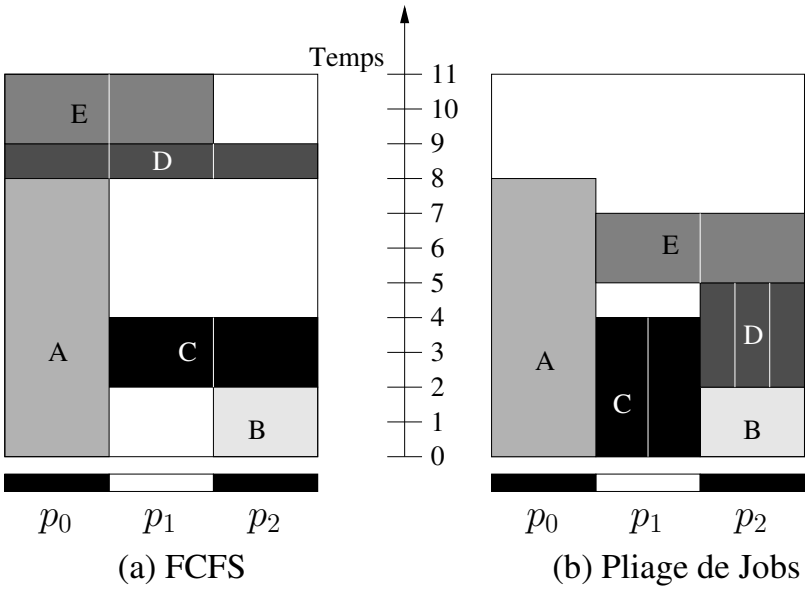


Figure 4.2 – *FCFS* vs Pliage

tion qui associe à chaque nombre μ_i de PEs une durée d'exécution $reqtime_i(\mu_i)$. Cette durée d'exécution $reqtime_i(\mu_i)$ est une borne supérieure pour la durée réelle (non connu du système) $runtime_i(\mu_i)$ du job i (tableau 4.1). En effet avec les ordonnanceurs de jobs classiques tel que Sun Grid Engine, l'utilisateur fournit un nombre de PEs et une durée d'exécution pour son job. Dès que le job atteint son quota de temps, l'ordonnanceur l'arrête. Ainsi nous pouvons faire l'hypothèse que pour un job donné, $\forall \mu_i,\ 1 \leq \mu_i \leq reqproc_i,\ runtime_i(\mu_i) = \alpha_i \times reqtime_i(\mu_i)$ où α_i est une constante et $0 \leq \alpha_i \leq 1$.

Dans ce chapitre nous utilisons les notations suivantes.

- i : le job considéré ;
- a_i : la date de soumission du job i ;
- $reqproc_i$: le nombre de PEs demandé au départ par le job i ;
- $1 \leq \mu_i \leq reqproc_i$: μ_i représente un nombre de PEs possible alloués au job i ;
- $runtime_i(m)$: la durée réelle du job J avec m PEs ;
- $reqtime_i(m)$: la durée requise par le job J avec m PEs ;
- α_i : le rapport entre la durée réelle d'exécution du job i et la durée requise par le job i ;
- $starttime_i(m)$: la date de début d'exécution du job i sur m PEs ;
- $starttime\_without\_folding_i$: la date de début d'exécution du job i s'il avait été ordonnancé avec *FCFS*.

L'objectif est de trouver un ordonnancement qui minimise le makespan C_{max}, la date de terminaison du dernier job.

4.4 Classe de complexité du problème

Trouver le meilleur ordonnancement optimisant le makespan de tâches moldables indépendantes est un problème NP-Difficile [38]. Dutot et al. ont fondé leur preuve sur celle de l'ordonnancement de tâches séquentielles indépendantes [50].

Du et Leung ont démontré dans [36] que pour des jobs rigides sans dates d'arrivée le problème est NP-Difficile. Nous démontrons que notre problème est un cas plus général du problème traité par Du et Leung. Notons $\mathcal{P}$ le problème de décision associé à notre problème d'optimisation et $\mathcal{Q}$ le problème de décision associé au problème de Du et Leung.

Proposition 1. *$\mathcal{P}$ est NP-Complet.*

Démonstration. $\mathcal{P}$ s'exprime de la manière suivante. Soient n jobs pliables qui arrivent aux dates a_i, ils requièrent au plus $reqproc_i$ PEs et $reqtime_i(reqproc_i)$ ut. Chacun des jobs i possède un tableau d'allocation. Etant donné un nombre y, existe-t-il un ordonnancement tel que $C_{max} \leq y$?

Montrons que $\mathcal{P}$ est dans NP. Étant donnée une solution, le temps pris pour vérifier si elle est valide et si $C_{max} \leq y$ est clairement en $\mathcal{O}(n)$.

$\mathcal{Q}$ s'exprime de la manière suivante. Soient n' jobs rigides qui sont déjà prêts, ils requièrent exactement $reqproc'_i$ PEs et durent exactement $reqtime'_i$ ut. Chacun des jobs i possède un tableau d'allocation. Etant donné un nombre y', existe-t-il un ordonnancement tel que $C'_{max} \leq y'$?

Nous réduisons au moyen d'une opération polynomiale f, $\mathcal{Q}$ vers $\mathcal{P}$. Soit I une instance quelconque du problème $\mathcal{Q}$. Construisons une instance $f(I)$ de $\mathcal{P}$ comme suit : $n = n'$; $a_i = 0$; $reqproc_i = reqproc'_i$; $\mu_i \in \{reqproc_i\}$; $\alpha_i = 1$; $runtime(\mu_i) = reqtime(\mu_i) = reqtime'_i$ et $y = y'$.

Nous montrons que I est une oui-instance de $\mathcal{Q}$ si et seulement si $f(I)$ est une oui-instance de $\mathcal{P}$. Supposons qu'un ordonnancement de $\mathcal{Q}$ est tel que $C'_{max} \leq y'$, alors trivialement un ordonnancement de $\mathcal{P}$ est tel que $C_{max} \leq y$. La réciproque est trivialement vérifiée.

Ainsi $\mathcal{P}$ est un NP-Complet et notre problème d'optimisation est NP-Difficile.

□

Par conséquent, nous ne pouvons approcher l'optimal que par des heuristiques.

Notre technique appelée *pliage de job* ressemble à la technique des *tâches moldables*, mais dans notre cas un job parallèle qui n'est pas à l'origine conçu pour être moldable peut s'exécuter sur un nombre de PEs inférieur à ce qui a été prévu.

4.5 Heuristiques utilisant le pliage de jobs

Dans ce paragraphe nous présentons nos contributions dans la conception de trois heuristiques utilisant le pliage de jobs. Nous avons choisi de concevoir trois heuristiques afin de montrer l'intérêt du pliage de jobs. Deux d'entre elles servent à montrer l'intérêt du pliage et le deuxième à montrer qu'en réalisant du pliage non entier, les gains sont plus importants que ceux d'un pliage entier. La troisième heuristique reposant sur l'utilisation de l'algorithme *backfilling*

montre que les performances par rapport à cet algorithme de référence sont réelles et que notre contribution sur le pliage de jobs n'est pas négligeable.

Dans un premier temps, ces heuristiques ne font intervenir aucune réservation, c'est-à-dire qu'on choisit de prendre le premier job de la file d'attente, en utilisant le même principe qu'un ordonnancement *FCFS*. Une des trois heuristiques reposant sur le principe du *backfilling* visent à boucher les trous inutilisés (cycles inutilisés) et à améliorer l'utilisation du cluster.

L'algorithme *FCFS* est simple à comprendre et à implémenter. C'est pourquoi il nous sert d'algorithme de référence pour comparer et évaluer nos heuristiques sans réservation. De plus nous montrons l'intérêt du pliage de jobs grâce au simple fait d'ajouter ce principe à l'algorithme *FCFS*.

Dans un deuxième temps, les heuristiques utilisent des *fenêtres de jobs*. Une fenêtre de jobs est un sous ensemble de jobs de la file d'attente. Ils font intervenir des réservations.

L'algorithme *backfilling* est simple et donne de bonnes performances. C'est pourquoi il nous sert aussi de référence pour comparer et évaluer nos heuristiques avec réservation. Ainsi nous montrons l'intérêt du pliage de job avec fenêtres de jobs grâce à l'intégration de ce principe dans l'algorithme *backfilling*.

4.5.1 Pliage entier H_1

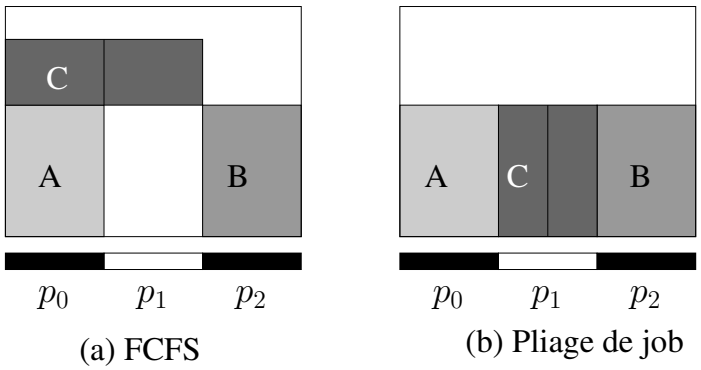


Figure 4.3 – Le principe du pliage de job entier

L'heuristique H_1 (algorithme 7) vise à améliorer l'algorithme *FCFS* en faisant terminer les jobs plus tôt et en leur attribuant, par pliage, moins de PEs qu'ils en ont besoin au départ. L'algorithme détermine le nombre approprié de PEs m de telle sorte que le job puisse terminer plus tôt que sa date de début s'il avait été ordonnancé avec l'algorithme *FCFS* (voir figure 4.3). Cet algorithme divise le nombre de PEs demandé au départ par un facteur entier.

Nous obligeons l'heuristique H_1 à limiter la date de fin du job i à la date de début du job s'il avait été ordonnancé avec *FCFS* car en pratique le temps réel du job $runtime_i(m)$ est inférieur au temps indiqué par l'utilisateur $reqtime_i(m)$. Ainsi quel que soit le nombre choisi m' de PEs avec $m' < m$, comme $starttime_i(m') + runtime_i(m') < starttime\_without\_folding_i$, le job i termine plus tôt avec l'heuristique H_1 qu'avec *FCFS*.

Algorithme 7: Pliage entier H_1

1 **pour chaque** *job i de la file d'attente* **faire**
2 choisir $max(m)$ PEs tel que$\Big(reqproc_i \ mod \ m = 0$ and $reqtime_i(m) + starttime_i(m) \leq starttime\_without\_folding_i\Big)$
3 attendre que m PEs soient libres
4 exécuter le job i

4.5.2 Pliage non entier H_2

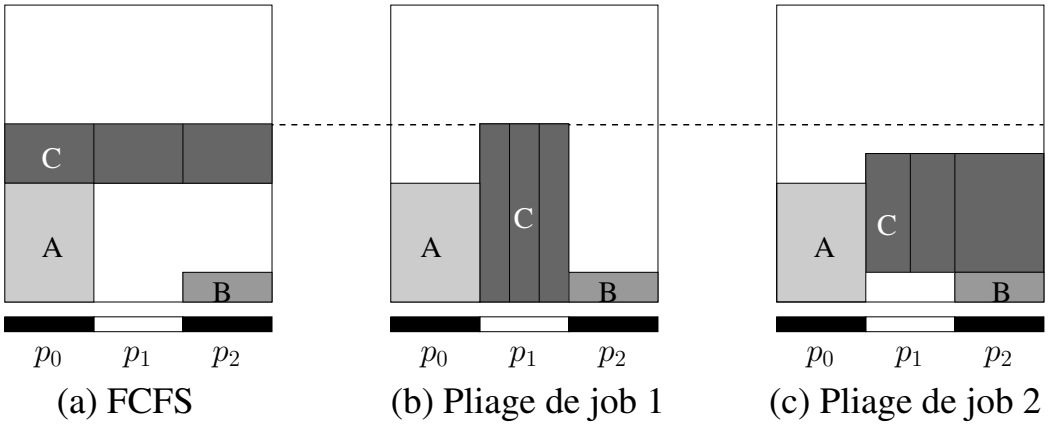


Figure 4.4 – Le principe du pliage de job non entier

L'heuristique H_2 (algorithme 8) détermine le nombre adéquat de PEs pour que le job termine le plus tôt possible. Dans la figure 4.4, le job C arrive dans le système et a besoin de 3 PEs. En ordonnançant le job C avec *FCFS*, il commence son exécution uniquement quand p_0 est libéré, alors qu'avec la technique du pliage, le job C commence plus tôt et finit plus tôt. Une première option est d'effectuer un pliage entier (ordonnancement 4.4(b)) en allouant seulement 1 PE au job C, mais dans ce cas, il termine au même moment que s'il avait été ordonnancé avec *FCFS* (ordonnancement 4.4(a)). Comme le montre la figure 4.4(c), avec $\frac{3}{2}$ fois moins de PEs le job termine plus tôt avec un pliage non entier. Dans ce cas, l'algorithme H_2 choisit la solution du pliage non entier (ordonnancement 4.4(c)).

Algorithme 8: Pliage non entier H_2

1 **pour chaque** *job i de la file d'attente* **faire**
2 choisir $max(m)$ PEs tel que $\Big(reqtime_i(m) + starttime_i(m)\Big)$ soit minimal
3 attendre que m PEs soient libres
4 exécuter le job i

4.5.3 Pliage entier H_1 + Backfilling

La technique du *backfilling* et l'heuristique H_1 sont combinées et cette nouvelle heuristique est décrite par l'algorithme 9. Elle prévoit des réservations pour chaque job de la file d'attente

dans une table d'ordonnancement en utilisant l'heuristique H_1. Quand une ressource est libérée, l'ordonnanceur applique un algorithme de *backfilling* sur les jobs qui sont assignés dans la table d'ordonnancement pour déterminer quel job exécuter. La figure 4.5 présente le principe de cette heuristique. Trois jobs A, B, C arrivent dans cet ordre dans le cluster à 4 PEs, d'autres jobs sont toujours en cours d'exécution. Le job A a besoin de 4 PEs, B en a besoin de 2, et C en a besoin de 4. Ces jobs sont placés en file d'attente et chaque réservation est calculée avec l'heuristique H_1. La figure 4.5(b) montre que la réservation calculée avec l'heuristique H_1 est plus compacte que celle utilisant l'algorithme *FCFS* (voir figure 4.5(a)). Dans la figure 4.5(b) les jobs A et B restent à la même place qu'avec l'ordonnancement *FCFS*. Le job C peut terminer plus tôt avec un seul PE, donc le job C est plié. Le job X termine plus tôt que prévu, 3 PEs sont libérés (voir figure 4.5(c)) et nous appliquons l'algorithme du *backfilling* sur les jobs A, B et C. Donc B peut commencer plus tôt que prévu par rapport à sa réservation dans la table d'ordonnancement 4.5(b). En effet, il ne perturbe pas la réservation du job A. Il est noter que l'on pouvait choisir le job C également, puisqu'il ne perturberait pas la réservation du job A.

Algorithme 9: Pliage entier H_1 + backfilling

```
1 pour chaque job i de la file d'attente faire
2     placer le job i dans la table T avec l'heuristique H1
3 en parallèle : quand une ressource est libérée début
4     appliquer backfilling sur les jobs dans T pour choisir le job e à exécuter
5     exécuter le job e
```

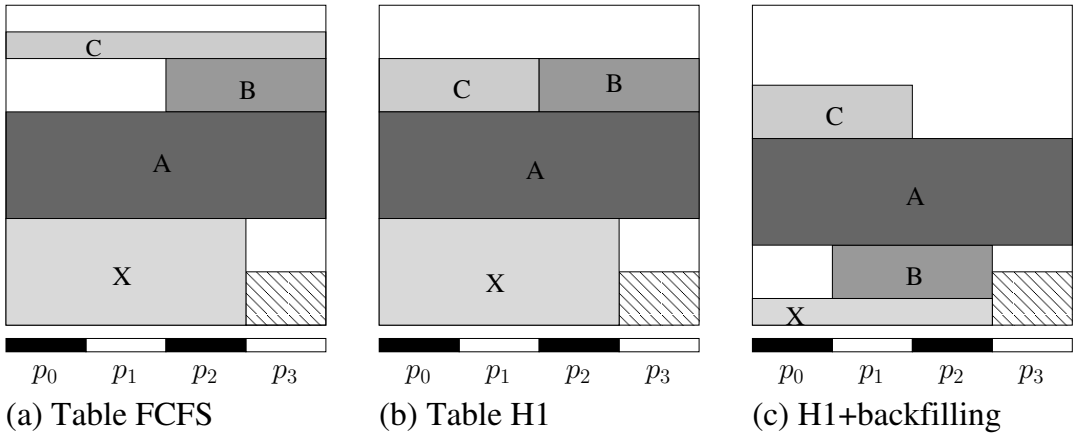


Figure 4.5 – Le principe de l'heuristique H_1 + *backfilling*

4.5.4 Heuristiques avec des fenêtres de jobs

Précédemment nous avons seulement considéré un job à la fois pour l'ordonnancement, le job en tête de la file d'attente. Nous considérons maintenant une *fenêtre de jobs*, c'est-à-dire un sous ensemble de jobs de la file d'attente. Soit z la taille de la fenêtre qui est égale au nombre maximal de jobs qui peuvent être ordonnancés. L'algorithme 10 montre le procédé pour ordonnancer des jobs avec une fenêtre de taille z. Cet algorithme est composé d'un ordonnanceur et d'un sélecteur de jobs. L'ordonnanceur de jobs prépare des réservations à l'avance à l'aide de

l'heuristique $HSched \in \{H_1, H_2\}$ pour les z premiers jobs de la file d'attente : début d'exécution + nombre de PEs alloués. Quand un ensemble de PEs est libéré, le nombre de PEs oisifs peut différer du nombre de PEs attendus par la réservation des jobs. Donc le sélecteur de jobs sert à choisir quel job exécuter.

Dans ces travaux, nous utilisons un simple sélecteur de jobs $HChoose \in \{EFT, backfilling\}$. Quand n PEs se libèrent, il regarde sa table de réservation et choisit le job qui a besoin d'au plus n PEs et qui finit le plus tôt : $HChoose = EFT$ (*Earliest Finish Time*) avec $H_1(z)$ et $H_2(z)$ et $HChoose = backfilling$ avec la combinaison $H_1(z) + backfilling$.

Algorithme 10: Principe de l'ordonnancement avec une fenêtre de jobs

```
1 La file d'attente n'est pas vide début
2     pour z premiers jobs i de la file d'attente faire
3         L'ordonnanceur HSched(i) effectue une réservation pour le job i
4 Des ressources sont libérées, m PEs sont disponibles début
5     chercher un job e avec le sélecteur de job HChoose(m)
6     executer le job e
```

4.6 Simulations et résultats

Dans cette section nous présentons les résultats des expériences que nous avons réalisées sur les heuristiques de pliage. Mener des expériences sur un vrai cluster est difficile car il coûte cher et nous ne pouvons pas en monopoliser un pour notre propre besoin. De plus les expériences menées avec des vraies applications et un vrai cluster ne sont pas reproductibles et peuvent durer longtemps. Pour ces raisons, nous avons décidé de développer un simulateur de cluster homogène. Ce simulateur est fondé sur une architecture maître/esclave. Il utilise SimGrid [25] et son API MSG.

Pour alimenter ce simulateur, il a besoin de jobs fictifs qui arrivent au fil du temps. Ces jobs ne font rien mais consomment virtuellement du temps. Ils possèdent des caractéristiques qui peuvent être enregistrées dans un fichier appelé *workload* (*fichier de trace décrivant l'historique de la charge de travail d'un cluster*).

4.6.1 Les workloads

Un workload est un historique de l'utilisation d'un cluster qui retrace chaque événement qui se produit dans l'ordonnanceur de jobs du cluster. Quelques workloads existent déjà et sont disponibles sur Internet et Feitelson [41] propose un format standard pour enregistrer les workloads. Les informations suivantes doivent au moins être enregistrées :

- la date de soumission du job,
- le nombre de PEs requis,
- la durée requise,
- le temps d'exécution réel.

Dans nos simulations, nous utilisons à la fois un workload réel provenant de traces d'un cluster existant et des workloads synthétiques. Les workloads réels fournissent une bonne base pour tester nos algorithmes pour des cas réels d'utilisation alors que les workloads synthétiques permettent d'étudier l'impact des différents paramètres sur leurs comportements, comme la charge ou la taille des jobs. Des workloads synthétiques issus d'analyses des workloads réels existent à travers les modèles de Lublin [70] ou de Downey [35].

Les caractéristiques des workloads synthétiques ont été générées aléatoirement avec une loi uniforme pour la durée d'exécution requise, la durée d'exécution réelle et le nombre de PEs requis. Nous avons choisi d'utiliser notre propre modèle de synthèse de workloads afin de mesurer l'impact de l'intervalle de temps entre chaque arrivée d'un job sur les performances des heuristiques.

Dans le livre de Feitelson [42], le temps entre chaque arrivée d'un job peut être modélisé par une loi de Poisson. Soit k le nombre de jobs qui arrivent pendant un intervalle de temps et λ le nombre moyen de jobs qui arrive pendant un intervalle de temps dans le système. La probabilité que pendant un intervalle de temps d'avoir exactement k jobs qui arrivent dans le système est $P(X = k) = e^{-\lambda}\frac{\lambda^k}{k!}$. Nous en déduisons que le temps moyen entre deux arrivées successives est $\frac{1}{\lambda}$. Dans nos simulations ce temps moyen entre deux arrivées successives est généré aléatoirement par un générateur de nombres aléatoires qui suit une loi de Poisson.

4.6.2 Résultats avec diverses valeurs de λ

Dans ces expériences nous simulons un cluster de 1024 PEs et nous avons 10000 jobs à ordonnancer. La mise en place des expériences est la suivante :

- $timeInterval = 3600\,s$: l'intervalle de temps pour le paramètre λ ;
- $rtMin = 500\,s$: le minimum de temps requis par un job ;
- $rtMax = 20\,000\,s$: le maximum de temps requis par un job ;
- $diffMax = 40$: le pourcentage maximum entre le temps requis par un job et son temps d'exécution réel ;
- $procMin = 1$: le nombre minimum de PEs requis par un job ;
- $procMax = 1024$: le nombre maximum de PEs requis par un job ;
- $\lambda \in \{10, 50, 100, 150, 200, 250, 300, 500, 750,\ 1000\}$: le nombre moyen de jobs arrivant par intervalle de temps $timeInterval = 3600\,s$.

Pour chaque valeur λ, 10 workloads synthétiques sont générés. Puis pour chaque valeur λ, la mesure est calculée au moyen d'une moyenne arithmétique des 10 valeurs pour éviter des résultats extravagants qui pourraient être dus à des cas spéciaux.

Dans ces travaux nous considérons divers métriques pour évaluer l'efficacité des algorithmes de deux points de vue : d'un point de vue système et d'un point de vue utilisateur. D'abord nous considérons l'amélioration du makespan apporté par chaque algorithme par rapport à la solution basique *FCFS* en fonction de diverses valeurs de λ.

La figure 4.6 montre les performances des résultats expérimentaux des différentes heuristiques. Nous remarquons qu'utiliser des fenêtres pour le *backfilling* et l'heuristique H_1 + *backfilling* améliore de manière drastique les performances par rapport à *FCFS*. De plus sans fenêtre de jobs, les améliorations des performances n'excède pas les 5% pour le *backfilling* et nos heuristiques. Avec la même taille de fenêtre, la combinaison de $H_1(5)$ + *backfilling* est 2% meilleur que *backfilling(5)* seul. Cette figure montre aussi l'intérêt de réaliser un pliage non entier par rapport à un pliage entier. En effet, l'amélioration apportée par un pliage non entier par rapport à un pliage entier est d'environ 3% avec ou sans fenêtre.

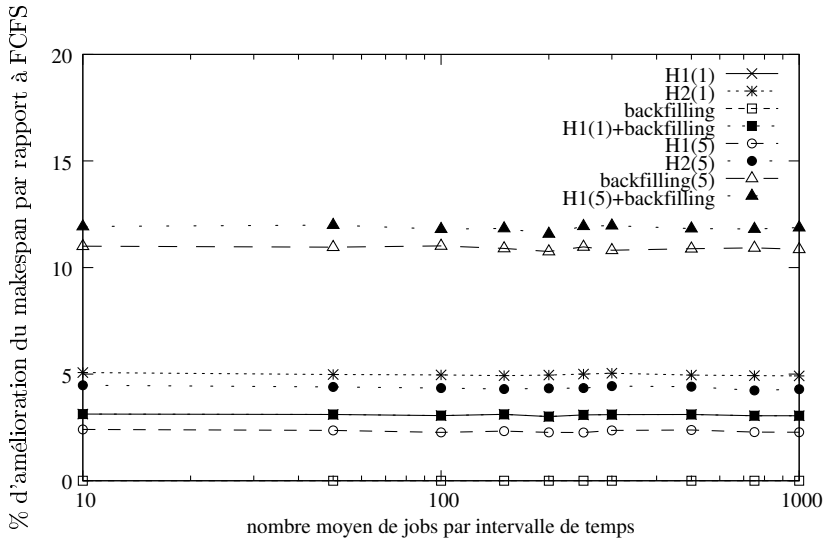


Figure 4.6 – Amélioration du makespan par rapport à *FCFS* en fonction de λ

Après avoir considéré le makespan comme métrique, nous considérons le stretch. La figure 4.7 montre le stretch moyen des ordonnancements obtenus avec *FCFS*, *backfilling* et nos heuristiques appliquées sur des workloads synthétiques en fonction des valeurs de λ. On remarque que le meilleur stretch est obtenu par l'heuristique $H_1(5)$ + *backfilling*. *FCFS* et *backfilling* donne les pires stretchs.

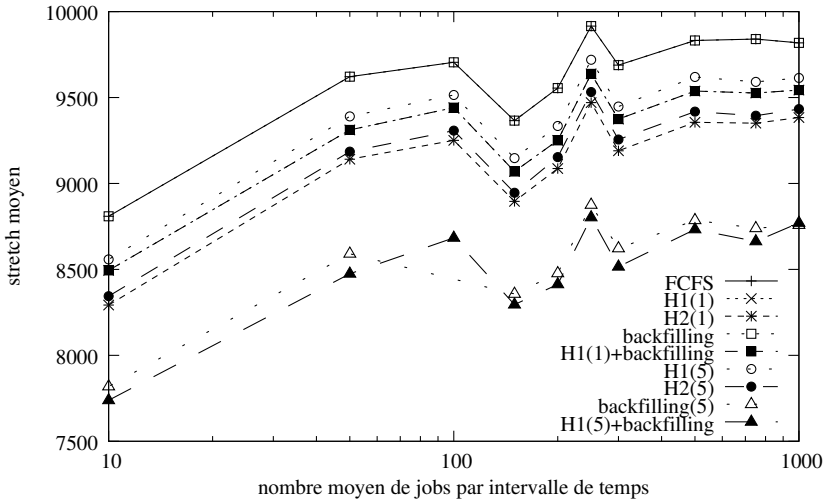


Figure 4.7 – Stretch moyen des ordonnancements obtenus avec les différentes heuristiques en fonction de λ

La figure 4.8 montre le stretch maximum des ordonnancements obtenus avec les différents algorithmes. $H_1(5) + backfilling$ reste le meilleur algorithme pour cette métrique.

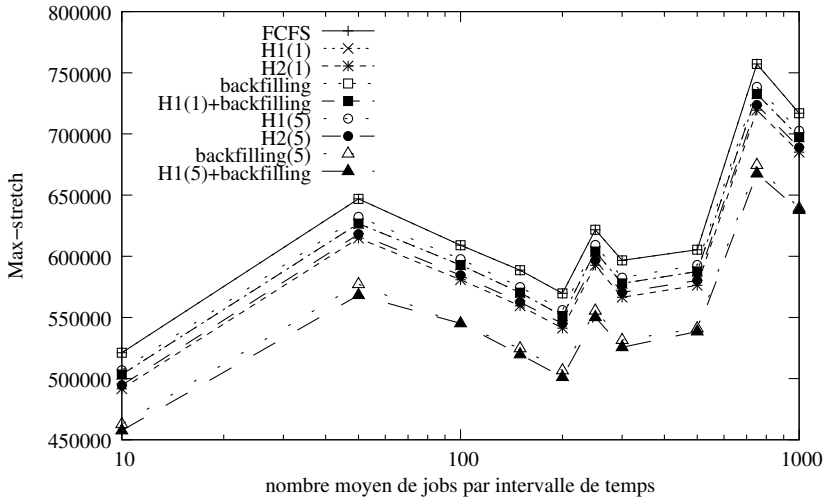


Figure 4.8 – Max stretch des ordonnancements obtenus avec les différentes heuristiques en fonction de λ

4.6.3 Résultats avec différentes valeurs pour le nombre de PEs requis

Dans cette section nous évaluons l'impact du niveau de parallélisme des jobs, c'est-à-dire, plus le nombre de PEs requis par un job est élevé plus son parallélisme est grand. Dans les expériences suivantes, nous simulons un cluster de 1024 PEs. Il y a 10000 jobs à ordonnancer. La mise en place des expériences est réalisée comme suit :

- les paramètres *timeInterval, rtMin, rtMax, diffMax, procMax* sont inchangés
- $procMin \in \{1, 5, 50, 100, 250, 500, 750, 1000\}$: le nombre minimal de PEs requis par un job
- $\lambda = 100$: le nombre moyen de jobs soumis par heure

La figure 4.9 montre l'amélioration du makespan par rapport à *FCFS* en fonction du nombre minimum de PEs requis par l'ensemble des jobs. Nous remarquons que pour $procMin \leq 500$ le meilleur algorithme est $H_1(5) + backfilling$. Le deuxième est $backfilling(5)$ mais quand $procMin > 500$, *backfilling(5)* donne les pires résultats. Nous pouvons expliquer ce phénomène. Quand *procMin* est petit, à cause de la loi uniforme pour générer le nombre de PEs requis, statistiquement, nous avons autant de jobs qui requièrent beaucoup de PEs et des jobs qui requièrent peu de PEs. Cette hétérogénéité permet de remplir les trous. À l'inverse, quand *procMin* est grand, le workload ne possède que des jobs qui demande un grand nombre de PEs, donc le *backfilling* est moins efficace. Donc quand le parallélisme est grand les performances des algorithmes décroissent.

4.6.4 Résultats avec des workloads réels

Afin de montrer les performances de nos algorithmes, nous les évaluons en utilisant trois workloads réels. Nous avons téléchargé ces workloads depuis le site web des archives de Dror G.

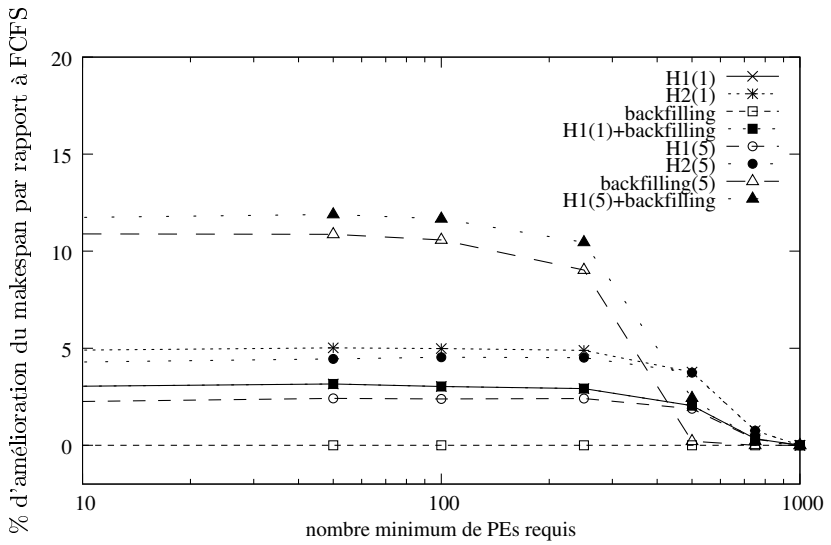


Figure 4.9 – Amélioration du makespan par rapport à *FCFS* en fonction du nombre minimum de PEs requis

Feitelson [41]. Nous évaluons les performances des algorithmes avec une fenêtre de jobs $z = 1$ et $z = 50$ sur les métriques stretch moyen, max stretch et makespan. Le premier workload provient de l'IBM SP2 qui comporte 100 nœuds au Swedish Royal Institute of Technology (KTH) à Stockholm. Ce workload contient les statistiques de 28489 jobs. Le deuxième workload est celui de l'iPSC de la NASA qui comporte 128 nœuds. Ce workload contient 18239 enregistrements. Le dernier workload que nous avons utilisé est celui de la CM-5 du LANL comportant 1024 nœuds. Il contient 122060 enregistrements.

Afin de pouvoir calculer le stretch pour un job nous avons considéré son temps d'exécution réel enregistré dans le workload comme le temps qu'il aurait pris s'il était seul dans le cluster. Néanmoins, ces workloads réels présentent des durées d'exécution égales à 0 seconde. Utiliser ces données brutes engendrerait des divisions par zéro. C'est pourquoi nous considérons pour le calcul du stretch que ces jobs ont une durée d'exécution de 1 seconde s'ils étaient seuls dans un cluster.

Les Tableaux 4.2, 4.3 et 4.4 montrent les résultats expérimentaux pour ces workloads respectifs. Pour les trois workloads nous remarquons que le makespan n'est pas amélioré par nos heuristiques par rapport à *FCFS*. Cela s'explique par le fait que dans les workloads les derniers jobs sont soumis à des dates telles qu'il n'est plus possible de faire mieux qu'un simple placement par *FCFS*. Bien que nous avons optimisé le makespan avec les jobs précédents, les derniers jobs arrivent trop tardivement pour réaliser une optimisation. Concernant le stretch moyen ou maximum, H_1 améliore pour les trois workloads *FCFS*. De plus nous remarquons comme avec les workloads synthétiques, que l'association $backfilling + H_1$ améliore grandement *backfilling*.

Le tableau 4.5 montre le temps CPU moyen nécessaire à nos heuristiques pour ordonnancer un job. Nous pouvons constater, pour les workloads du KTH et de la NASA dont les machines sont de petites tailles, de l'ordre d'une centaine de nœuds, sont de l'ordre de la milliseconde. Pour une machine de taille plus importante comme celle du LANL qui comporte 1024 nœuds les temps de calcul sont de l'ordre de la centaine de millisecondes. En effet, plus la machine est importante, plus les jobs demandent des ressources de calcul, plus les tableaux d'allocations deviennent grands. Ainsi le temps de calcul dépend du niveau de parallélisme des jobs. Les

Table 4.2 – Résultats d'un workload réel du KTH

Algorithmes	stretch moyen	max stretch	makespan
FCFS	13065	1011537	29379608
backfilling(1)	2260	362780	29366789
$H_1(1)$	1194	244086	29363626
backfilling(1) + $H_1(1)$	1194	244086	29363626
$H_2(1)$	940	203917	29363626
backfilling(50)	195	129164	29363626
$H_2(50)$	900	186632	29376950
$H_1(50)$	1165	238455	29377850
backfilling(50) + $H_1(50)$	86	51604	29363626

Table 4.3 – Résultats d'un workload réel de la NASA

Algorithmes	stretch moyen	max stretch	makespan
FCFS	1.02598	87.7174	7949022
backfilling(1)	1.02139	87.2044	7949022
$H_1(1)$	1.001756	7.4938	7949022
backfilling(1) + $H_1(1)$	1.001756	7.4938	7949022
$H_2(1)$	1.00595	35.4594	7949022
backfilling(50)	1.01175	73.1667	7949022
$H_2(50)$	1.00596	35.4594	7949022
$H_1(50)$	1.001764	7.4328	7949022
backfilling(50) + $H_1(50)$	1.00033	3.93827	7949022

temps de calcul de nos heuristiques montrent qu'elles ne sont pas gourmandes en temps.

4.7 Respect des deadlines par les heuristiques

Nous considérons à présent la question du respect des deadlines par les différentes heuristiques présentées précédemment. Quand un job arrive dans le système on lui attribue une deadline qui correspond à sa date de fin dans l'ordonnancement par *FCFS*. Ce deadline associé à chaque job est une contrainte que nous nous fixons afin de garantir que nos heuristiques sans fenêtres de jobs améliorent dans tous les cas l'algorithme *FCFS*.

Nous montrons ici que les algorithmes sans fenêtres H_1, H_2, $H_1 + backfilling$ et l'algorithme avec fenêtres $H_1(z) + backfilling$ respectent ces deadlines.

4.7.1 Notations

Nous posons les notation suivantes :

- W : l'ensemble des jobs du workload à ordonnancer
- $w = Card(W)$: le nombre de jobs dans W
- $sh1_i$: la date réelle de commencement du job i avec l'algorithme H_1 ;
- $eh1_i$: la date réelle de fin du job i avec l'algorithme H_1 ;
- s_i : la date de commencement du job i avec l'algorithme *FCFS* ;
- e_i : la date de fin du job i avec l'algorithme *FCFS* ;

Table 4.4 – Résultats d'un workload réel du LANL

Algorithmes	stretch moyen	max stretch	makespan
FCFS	1087	368784	62293404
backfilling(1)	675	247575	62293404
$H_1(1)$	857	304025	62293404
backfilling(1) + $H_1(1)$	857	304025	62293404
$H_2(1)$	851	305620	62293404
backfilling(50)	93	32905	62293404
$H_2(50)$	939	330099	62293404
$H_1(50)$	902	314748	62293404
backfilling(50) + $H_1(50)$	69	29283	62293404

Table 4.5 – Temps CPU moyen (ms) pour ordonnancer un job

Algorithmes	KTH	NASA	LANL
FCFS	0.99	0.71	9.83
backfilling(1)	1.59	0.87	13.19
$H_1(1)$	0.63	0.75	7.06
backfilling(1) + $H_1(1)$	1.19	0.97	15.28
$H_2(1)$	0.61	0.74	7.22
backfilling(50)	6.51	0.86	103.8
$H_2(50)$	9.68	0.75	146.8
$H_1(50)$	10.40	0.76	144.1
backfilling(50) + $H_1(50)$	4.58	0.98	122.8

- $nh1_i$: le nombre réel de PEs utilisés par le job i avec l'algorithme H_1 ;
- n_i : le nombre de PEs utilisés par le job i avec l'algorithme *FCFS* ;
- $uh1(t)$: le nombre réel de PEs utilisés à l'instant t avec l'algorithme H_1 ;
- $u(t)$: le nombre de PEs utilisés à l'instant t avec l'algorithme *FCFS* ;

4.7.2 Preuve de la terminaison d'un job avant *FCFS* avec l'heuristique H_1 et H_2

Comme nous l'avons déjà mentionné, dans notre problème la relation entre la durée réelle d'exécution $runtime_i(\mu_i)$ d'un job i et sa durée requise $reqtime_i(\mu_i)$ avec μ_i PEs est donnée par $runtime_i(\mu_i) = \alpha_i \times reqtime_i(\mu_i)$ avec $0 \leq \alpha_i \leq 1$.

Proposition 2. *Étant donnés une date t, un ensemble de jobs j en cours d'exécution et un job i prêt à être ordonnancé par l'heuristique H_1. Pour ce job i, $eh1_i \leq e_i$.*

Démonstration. L'heuristique H_1 réalise le pliage d'un job sur μ_i PEs, si, compte tenu de sa durée d'exécution requise, il termine plus tôt qu'en utilisant *FCFS*. Cette condition s'écrit :

$$sh1_i + reqtime_i(\mu_i) \leq s_i + reqtime_i(reqproc_i)$$

En remplaçant $reqtime_i(\mu_i)$ par son expression en fonction de $reqproc_i$, nous obtenons :

$$sh1_i + reqtime_i(reqproc_i) \times \left\lceil \frac{reqproc_i}{\mu_i} \right\rceil \leq s_i + reqtime_i(reqproc_i)$$

Maintenant, nous exprimons $reqtime_i(reqproc_i)$ en fonction de $runtime(reqproc_i)$. Nous obtenons :

$$sh1_i + \frac{1}{\alpha} \times runtime_i(reqproc_i) \times \left\lceil \frac{reqproc_i}{\mu_i} \right\rceil \leq s_i + \frac{1}{\alpha} \times runtime_i(reqproc_i)$$

Comme $0 \leq \alpha$, nous pouvons multiplier chaque membre de cette inégalité sans changer son sens et nous obtenons :

$$\alpha \times sh1_i + runtime_i(reqproc_i) \times \left\lceil \frac{reqproc_i}{\mu_i} \right\rceil \leq \alpha \times s_i + runtime_i(reqproc_i) \tag{4.1}$$

Nous rappelons que le principe du pliage de jobs réside dans le fait que nous les faisons exécuter plus tôt qu'avec l'algorithme *FCFS*. Ainsi nous écrivons

$$sh1_i \leq s_i$$

Comme $0 \leq (1-\alpha)$, nous pouvons multiplier chaque membre de cette inégalité sans changer son sens et nous obtenons :

$$(1-\alpha) \times sh1_i \leq (1-\alpha) \times s_i \tag{4.2}$$

En additionnant membre à membre les inégalités 4.1 et 4.2, nous obtenons :

$$sh1_i + runtime_i(reqproc_i) \times \left\lceil \frac{reqproc_i}{\mu_i} \right\rceil \leq s_i + runtime_i(reqproc_i)$$

et cette inégalité peut aussi s'écrire :

$$eh1_i \leq e_i$$

□

Avec l'heuristique H_1 nous venons de montrer qu'à tout instant, pour un job prêt à être ordonnancé, la condition de pliage est suffisante pour que le job termine réellement avant *FCFS*.

Pour l'heuristique H_2, comme elle choisit parmi les pliages du job prêt à être ordonnancé celle qui lui permet de finir le plus tôt, cette condition de choix est un cas particulier de celle de H_1. Donc la démonstration est exactement la même pour H_2.

4.7.3 Preuve du respect des deadlines par les heuristiques H_1 et H_2

Proposition 3.

$$\forall j \in W,\ eh1_j \leq e_j$$

Démonstration. Nous démontrons que l'algorithme H_1 respecte les deadlines par une démons-

tration par récurrence sur w, la taille du workload.

Pour $w = 1$, la proposition est vraie, il s'agit de la démonstration de la proposition 2.

Dans le cas général $w > 1$, nous suppose qu'il y a $w-1$ jobs j qui ont été ordonnancés avec l'algorithme H_1 et qu'ils finissent avant leur deadline e_j : $\forall j\ 1 \leq j \leq w-1\ eh1_j \leq e_j$

Soit x le w^{eme} job que l'on doit ordonnancer. Nous montrons qu'il finit avant la date e_x. Nous avons deux cas, parmis les $w-1$ autres jobs y déjà ordonnancés :

1. soit tous les $w-1$ jobs finissent avant s_x avec l'algorithme *FCFS* : $e_y \leq s_x$
2. soit il existe des jobs y parmi les $w-1$ jobs qui finissent après s_x avec l'algorithme *FCFS* : $e_y > s_x$

cas 1

Nous avons $e_y \leq s_x$, or d'après l'hypothèse de récurrence $eh1_y \leq e_y$. Nous en déduisons que $eh1_y \leq s_x$, ce qui veut dire que quand tous les jobs y ont fini avec l'algorithme H_1, le job x peut commencer au plus tard à s_x et prendre le maximum de PEs qu'il lui est permis et que son temps d'exécution sera exactement celui prévu dans l'ordonnancement *FCFS*. Il finira bien avant e_x. Ainsi $eh1_x \leq e_x$ et la proposition est vraie dans ce cas.

cas 2

Nous avons un sous ensemble de jobs y des $w-1$ jobs tels que $e_y > s_x$. Comme x arrive après les $w-1$ jobs, le job x commence nécessairement après le commencement du $(w-1)^{eme}$ job avec l'algorithme *FCFS*. Nous avons donc $s_y \leq s_x < e_y$.

D'après le principe de l'algorithme *FCFS* nous avons $\forall a, b \in \{y\}$ tel que a soit soumis avant b , $s_a \leq s_b$. Par conséquent $s_a \leq s_b \leq s_x < e_a$ et $s_a \leq s_b \leq s_x < e_b$. Ainsi ces jobs y sont nécessairement ordonnancés parallèlement les uns aux autres avec l'algorithme *FCFS*. C'est pourquoi dans l'ordonnancement avec l'algorithme *FCFS* à l'instant s_x les y jobs utilisent au total $u(s_x)$ PEs (égalité 4.3).

Comme nous avons supposé que nous avons un sous ensemble de jobs y des $w-1$ jobs tels que $e_y > s_x$, les autres jobs z tels que $e_z \leq s_x$, respectent l'hypothèse de récurrence (cas 1), c'est à dire, $eh1_z \leq e_z$, donc dans l'algorithme H_1, à l'instant s_x les jobs z ont terminés. Ainsi nous ne considérons que les jobs y. Les jobs y utilisent dans l'ordonnancement H_1 au maximum n_y PEs (inégalité 4.4) et ceux ci peuvent se réorganiser (inégalité 4.5).

Donc nous avons

$$u(s_x) = \sum_{j \in y} n_j \tag{4.3}$$

$$\geq \sum_{j \in y} nh1_j \tag{4.4}$$

$$\geq uh1(s_x) \tag{4.5}$$

L'inégalité (inégalité 4.5) signifie qu'à l'instant s_x dans l'ordonnancement H_1, dans il y a autant de PEs libres que dans l'ordonnancement *FCFS* et que le job x peut commencer au plus

tard à l'instant s_x avec n_x PEs. Donc il respectera son deadline et la proposition est vraie dans ce cas.

Nous venons de démontrer que la proposition est vraie quelle que soit la taille du workload.

□

Nous démontrerions de la même façon que H_2 respecte les deadlines.

4.7.4 Preuve du respect des deadlines par l'heuristique $H_1 + Backfilling$

Les jobs i ordonnancés avec un algorithme X, ont une date de fin eX_i et le but de l'algorithme *backfilling* est de faire exécuter des jobs plus tôt sans repousser la fin d'exécution prévue des jobs. Si nous appelons eB_i la date de fin du job i exécuté plus tôt avec *backfilling* alors nous avons

$$eX_i \geq eB_i$$

Or d'après ce qui précède, avec $X = H_1$, $e_i \geq eh1_i \geq eB_i$. Donc $e_i \geq eB_i$. Donc chaque job i terminera avec $H_1 + backfilling$ plus tôt qu'avec FCFS.

4.7.5 Respect ou non respect des deadlines par les heuristiques avec les fenêtres de jobs

Les algorithmes H_1 et H_2 avec fenêtres ne respectent pas les deadlines. Comme $HChoose = EFT$, il suffit qu'un petit job soit ordonnancé à une date de fin faible, et qu'une ressource se libère, pour qu'il passe devant un gros job et repousse sa date de fin. Donc les jobs avec $H_1(z)$ et $H_2(z)$ ne terminent pas nécessairement plus tôt qu'avec *FCFS*.

En revanche si $HChoose = backfilling$, on montrerait de même façon que dans le paragraphe 4.7.4 que $H_1(z) + backfilling$ avec fenêtres respecte les deadlines en remplaçant l'algorithme X par $H_1(z)$.

4.8 Synthèse

Dans ce chapitre, nous avons présenté une nouvelle technique d'ordonnancement, appelée pliage de job, pour ordonnancer des jobs dans les clusters homogènes. Cette technique s'appuie sur la virtualisation qui crée des PEs virtuels et change ainsi le nombre de PEs physiques utilisés par les jobs parallèles : la virtualisation fait croire aux jobs qu'ils tournent sur le nombre de PEs qu'ils ont demandés, alors qu'ils tournent réellement sur moins de PEs. Le problème que nous traitons est un problème d'ordonnancement online qui a pour objectif d'optimiser l'utilisation du cluster. Nous avons proposé d'utiliser la métrique *makespan* associée au respect des deadlines attribué à chaque job dès leur arrivée dans le cluster. Nous avons conçu des heuristiques polynomiales pour ce problème NP-Difficile. Nous avons exécuté des simulations de manière intensive afin d'évaluer les performances des différentes heuristiques utilisant le pliage de job. Nous avons également comparé ces heuristiques en considérant d'autres métriques centrées utilisateur, comme le stretch moyen et stretch maximum. Avec trois workloads réels,

les améliorations sont significatives en terme de stretch moyen et par rapport à l'algorithme *FCFS*. Nous avons montré que la combinaison du *backfilling* et du pliage de job apporte une plus grande amélioration en mélangeant un fort parallélisme et un faible parallélisme au niveau des jobs. Nous avons aussi montré que l'algorithme H_1, H_2 et $H_1 + backfilling$ respectent les deadlines imposés. En revanche avec les fenêtres de jobs, seule l'heuristique $H_1 + backfilling$ respecte les deadlines.

Chapitre 5

Virtualisation et pliage de jobs off-line

Dans le chapitre précédent nous avons abordé une technique appelée pliage de jobs pour mieux utiliser un cluster de calcul partagé entre plusieurs utilisateurs. Nous avons conçu plusieurs heuristiques de pliage qu'il reste à implémenter dans un vrai ordonnanceur de jobs. Maintenant, certaines des contraintes de travail changent. Nous avons à notre disposition un cluster homogène dédié à un seul utilisateur. Cet utilisateur dispose d'une collection de jobs parallèles qu'il souhaite terminer au plus tôt et il connaît à l'avance la durée exacte de ses jobs. En pratique, il peut utiliser le pliage de jobs pour les terminer plus tôt. Ces jobs sont alors assimilables à des jobs moldables dans le cas où il utilise le pliage, et à des jobs rigides dans le cas contraire.

C'est pourquoi, nous traitons dans ce chapitre le problème d'ordonnancement d'une collection de jobs indépendants dans une machine multiprocesseur. À la fois les jobs rigides et les jobs moldables sont traités. L'hypothèse selon laquelle il n'y a qu'un seul utilisateur sur le cluster est importante, car elle implique pour notre problème de choisir le bon critère d'optimisation. Comme l'utilisateur souhaite terminer ses jobs au plus tôt, le but est de minimiser le makespan de l'ordonnancement. Ce problème est connu pour être NP-Difficile et différentes heuristiques ont déjà été proposées. Nous pouvons citer les travaux de Jansen et Porkolab [61] qui ont formulé en 1999 le problème de minimisation du makespan d'une collection de jobs rigides (*NPTS* : non-malleable parallel task scheduling) et d'une collection de job moldables (*MPTS* : malleable parallel task scheduling) sous la forme d'un programme linéaire mixte. En fait comme nous pouvons le constater dans l'étude de Martello et al. [69] le problème de minimisation du makespan d'une collection de jobs rigides ressemble à un problème de strip packing à deux dimensions où il faut minimiser la hauteur de la bande qui représente le *makespan*. Ce qui différencie le problème de strip packing et le problème d'ordonnancement de jobs est la possibilité dans ce dernier de placer des jobs sur des PEs de manière non contigüe. Par ailleurs, dans cette étude nous constatons que, ce problème avec les jobs rigides peut aussi se ramener à problème de bin packing à deux dimensions où le nombre de boîtes de hauteur unitaire à utiliser représente le *makespan*.

Nous introduisons une nouvelle approche qui repose sur des approximations linéaires successives. L'idée est de relaxer un programme linéaire en nombres entiers et d'utiliser la linéarisation de la norme ℓ_p dans la fonction objectif pour forcer les valeurs de la solution à tendre vers des

nombres entiers. Nous comparons notre approche avec l'algorithme de liste classique LTF (*Largest Task First*). Nous avons conçu différents algorithmes utilisant les approximations linéaires successives du modèle creux. Cette approche donne de bons résultats. La contribution de ces travaux tient en la conception de ces algorithmes et l'intégration de la méthode mathématique dans le monde de l'ordonnancement.

5.1 Le problème

Dans cette section nous positionnons et définissons le problème de manière formelle. Nous considérons le problème d'ordonnancement d'une collection de n jobs parallèles indépendants. Nous nous attaquons aux cas des jobs rigides et moldables.

Les jobs sont exécutés dans un cluster homogène de nœuds distribués ou sur un ordinateur à mémoire partagée ou distribuée. Nous notons m le nombre de PEs disponibles dans la plate-forme sur laquelle les jobs parallèles s'exécutent.

Les jobs rigides se caractérisent par un temps d'exécution fixé ainsi qu'un nombre donné de PEs requis, c'est-à-dire, que le job ne peut s'exécuter avec plus ou moins de PEs prévu au départ. Chaque job rigide i est défini par son nombre de PEs requis $reqproc_i$ et sa durée d'exécution $reqtime_i$.

Un job moldable peut être exécuté sur différents nombres de PEs, mais ce nombre est fixé une fois pour toute dès l'exécution du job et ne peut être changé durant son exécution. Les jobs moldables considérés respectent le modèle défini de chapitre précédent et possèdent chacun un tableau d'allocation comme celui donné par le tableau 4.1. Soit $reqtime_i$ la durée du job i qui demande au plus $reqproc_i$ PEs. Soit $t_i(m)$ la durée du job i si m PEs sont alloués au job i. Nous rappelons que la relation entre la durée d'un job i et son nombre de PEs alloués est :

$$\forall i,\ \forall m \leq reqproc_i,\ t_i(m) = \left\lceil \frac{reqproc_i}{m} \right\rceil reqtime_i$$

Notre objectif est de minimiser le temps de complétion du dernier job (makespan) de l'ordonnancement. Selon la classification pour les problèmes d'ordonnancement $\alpha|\beta|\gamma$ (plate-forme | application | critère à optimiser) donné par Graham dans [53], ce problème se note $P|size_i|C_{max}$.

5.2 Approche creuse pour promouvoir les pénalisations avec des linéarisations successives

La méthode d'optimisation du makespan d'un ordonnancement d'une collection de jobs parallèles que nous proposons repose sur deux étapes principales. D'abord nous formulons le problème sous forme d'un programme linéaire en nombres entiers, puis nous relaxons ce programme et nous mettons en œuvre une approche creuse pour promouvoir la pénalisation, afin de se rapprocher de solutions à valeurs presque entières.

Expliquons le principe de la pénalisation dans notre cas. Dans notre programme linéaire en nombres entiers, toutes les variables sont binaires. En relaxant ce programme linéaire en nombres entiers, toutes les variables sont rationnelles et sont comprises entre 0 et 1. L'objectif

de la pénalisation est de contraindre les variables à être proche de 0 ou de 1. Ainsi la fonction objectif dans le programme linéaire contient l'ensemble des pénalités des variables. L'objectif est de minimiser la pénalité totale. Si une variable n'est ni proche de 0 ni proche de 1, alors la valeur de sa pénalité est grande. En revanche, si une variable est proche de 0 ou de 1, la valeur de sa pénalité est faible. Comme nous allons le voir, cette fonction objectif qui réalise la pénalisation n'est pas linéaire.

Nous désignons par le terme *sparsité* le caractère creux du problème. Dans ce problème le vecteur d'inconnues est très creux[1] Comme cette sparsité implique la minimisation d'une fonction objectif non linéaire nous utilisons des approximations par linéarisations successives afin de linéariser la fonction objectif. Dans cette section nous détaillons les principales étapes de la méthode.

5.2.1 Sparsité pour promouvoir les pénalisations

Des travaux récents sur le problème de récupération creuse (an anglais *sparse recovery problem*) en statistiques et dans le traitement du signal ont attiré l'attention générale sur le fait qu'utiliser des pénalisations non différentiables telles que la norme ℓ_p peut être un outil pour résoudre des problèmes combinatoires. Cette approche qui vise à construire des relaxations continues aux problèmes combinatoires difficiles est un ingrédient clé pour résoudre des problèmes combinatoires, par exemple en 2006, le nouveau domaine appelé *acquisition éparse* (en anglais *Compressed Sensing*) dont les auteurs sont Candès, Romberg and Tao [18] et Donoho [34] qui montrent que trouver une solution éparse d'un système sous-déterminé d'équations linéaires peut parfois être équivalent à trouver une solution avec la plus petite norme ℓ_1. Cette découverte a engendré une activité de recherche intense ces dernières années. Des travaux se sont concentrées sur la recherche de conditions suffisantes plus faibles sur les systèmes linéaires pour lesquelles il est possible de prouver cette équivalence. En particulier, des chercheurs ont trouvé que pour les matrices satisfaisant certaines conditions d'incohérence (impliquant que les colonnes de la matrice associée sont presque orthogonales), l'équivalence entre trouver la solution avec la plus petite norme ℓ_p et trouver la solution la plus creuse est vraie pour les systèmes avec un nombre d'inconnues de l'ordre de l'exponentielle du nombre équations. D'autres pénalisations non-différentiables ont été proposées afin d'augmenter les performances des procédures de récupération creuse. Candès Wakin et Boyd proposent un algorithme itératif ℓ_1 dans [19]. Chartrand [26] propose la procédure suivante, avec x comme solution du problème :

$$\min_x \|x\|_p \tag{5.1}$$

La norme ℓ_p n'est pas linéaire et donc le problème ne peut être résolu par un solveur linéaire. Nous proposons d'utiliser l'astuce des programmes linéaires successifs pour linéariser le problème. Cette astuce repose sur l'injection de solutions trouvées à chaque itération dans la fonction objectif linéarisé du programme linéaire à l'itération suivante. La linéarisation de la fonction objectif peut passer par la méthode du gradient ou par d'autres méthodes comme le *Reweighted Linear Program.*

[1] un vecteur est creux s'il contient "beaucoup" de zéros

5.2.2 Approximation linéaire

En physique et mathématiques on parle de développement limité d'ordre 1 ou d'approximation linéaire d'une fonction f, quand on souhaite approximer f au voisinage d'un point x_0 par une fonction linéaire. Si f est dérivable en x_0 une approximation linéaire de f s'exprime de la manière suivante :

$$f(x_0 + h) = f(x_0) + f'(x_0)h + o(h)$$

Les mathématiciens généralisent l'approximation linéaire pour les fonctions définies sur $\mathbb{R}^n$ avec l'utilisation du gradient. Ils appellent gradient d'une fonction f à plusieurs paramètres ($f : \mathbb{R}^n \to \mathbb{R}$), noté ∇f le vecteur dont chaque composante est égale à la dérivée par rapport à chacune des variables. Il indique comment f évolue selon les différentes variables. Par exemple, si $f(x, y, z)$ est définie sur $\mathbb{R}^3$, nous avons :

$$\nabla f = \begin{pmatrix} \frac{\partial f(x,y,z)}{\partial x} \\ \frac{\partial f(x,y,z)}{\partial y} \\ \frac{\partial f(x,y,z)}{\partial z} \end{pmatrix}$$

Comme pour les fonctions à une variable on peut approximer f de la manière suivante :

$$f(x + h) = f(x) + \langle \nabla f(x), h \rangle + o(h)$$

avec les chevrons qui représentent le produit scalaire.

5.3 Application de la méthode sur le problème d'ordonnancement

Dans ce paragraphe nous appliquons la méthode sur le problème d'ordonnancement d'une collection de jobs parallèles. D'abord cela implique de définir une représentation creuse du problème. Nous appliquons ensuite les deux étapes de la méthode d'approximation linéaire sur le modèle creux ainsi définis.

5.3.1 Formulation du problème comme un programme linéaire en nombres entiers

Comme il était indiqué dans la formulation du problème d'ordonnancement, pour les jobs rigides, une des solutions du programme linéaire doit fournir uniquement le début d'exécution de chaque job car ceux-ci ont un nombre de PEs et des durées constants. Pour les jobs moldables, les durées d'exécutions dépendent du nombre de PEs qui leur sont alloués. Donc l'ordonnancement des jobs moldables est complètement déterminé si nous connaissons leur date de début d'exécution et le nombre de PEs qui leur sont alloués. Nous appelons *configuration* d'un job le nombre de PEs alloués à ce job. Nous appelons *position* d'un job, sa position donnée directement par sa date de début d'exécution dans une échelle temporelle discrétisée. Enfin, nous appelons *slot* le couple (*configuration*, *position*).

Créons une liste de slots (configurations, positions) pour chaque job. L'idée consiste à créer un vecteur x_i pour chaque job i. Chaque composante $x_{i,s}$ du vecteur x_i est une variable binaire qui indique si le slot s du job i est choisi ou non. Ensuite nous fixons un horizon de temps T

et nous laissons un programme linéaire trouver une solution. Nous réduisons progressivement l'horizon de temps T jusqu'à ce que le programme linéaire ne trouve plus de solutions. Nous définissons le vecteur x comme la concaténation du vecteur x_i de chaque job i.

5.3.2 Notations

Les constantes sont les suivantes :

- n : le nombre de jobs à ordonnancer ;
- m : le nombre de PEs disponibles ;
- T : l'horizon du temps ;
- $proc_{i,j}$: le nombre de PEs pour la configuration j du job i ;
- $nconf_i$: le nombre de configurations possibles pour le job i ;
- $nslot_i$: le nombre de slots pour le job i ;
- $C_{i,s}$: la configuration du job i donnée par le slot s ;
- $run_{i,s,t}$ indique si le job i avec le slot s tourne à l'instant t ;
- $reqproc_i$: le nombre de PEs demandé au départ par le job i ;
- $reqtime_i$: la durée requise au départ par le job i.

Il n'y a qu'une seule variable : $x_{i,s}$. Il s'agit d'une variable binaire qui indique si le slot s du job i est utilisé.

La figure 5.2 illustre une matrice $run_{A,s,t}$ sur un exemple de job moldable A (Figure 5.1) qui requiert 2 PEs et qui dure 1 unité de temps. Cette matrice est de dimension $T \times nslot_A$: ici l'horizon de temps T est fixé à 4 et le job A possède $nslot_A = 4 + 3 = 7$ slots possibles. La première représente les slots possibles du job A avec 2 PEs et la deuxième représente ses slots possibles avec 1 PE. En fait la matrice $run_{A,s,t}$ indique à chaque instant (suivant la verticale) si le job A s'exécute pour un slot donné (suivant l'horizontale). La matrice $run_{A,s,t}$ est composée de 2 sous matrice. Par exemple la colonne 1 de la première sous matrice, indique quand le job est en cours d'exécution avec 2 PEs ; pour les autres colonnes, le job est translaté dans le temps si c'est possible. La colonne 1 de la deuxième sous matrice, indique quand le job A est en cours d'exécution avec 1 seul PEs ; il faut noter que le temps d'exécution du job A est doublé. De plus la matrice $run_{A,s,t}$ exprime le fait que le job A doit s'executer sans s'interrompre avant la fin.

a Formulation en programme linéaire en nombres entiers

Le problème s'exprime comme un programme linéaire mais sans fonction objectif car il suffit de trouver une solution. Comme nous l'avons dit précédemment, on réduit T et on lance le programme linéaire dont le principe a été donné précédemment, jusqu'à ce que celui-ci ne trouve plus de solutions.

Les contraintes sont les suivantes :

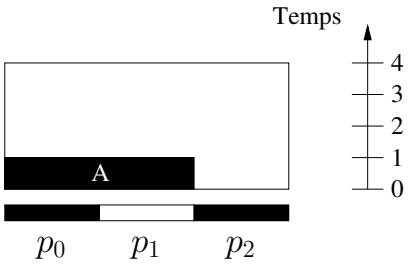


Figure 5.1 – exemple d'un job moldable

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ \underbrace{0 & 0 & 0 & 1}_{\text{2 PEs}} & \underbrace{0 & 0 & 1}_{\text{1 PE}} \end{pmatrix}$$

Figure 5.2 – La matrice $run_{A,s,t}$ associée

$$\forall 1 \leq i \leq n, \quad \sum_{s=1}^{s=nslot_i} x_{i,s} \quad = 1 \tag{5.2}$$

$$\forall 1 \leq t \leq T, \sum_{i=1}^{i=n} \sum_{s=1}^{s=nslot_i} x_{i,s} \times run_{i,s,t} \times proc_{i,C_{i,s}} \quad \leq m \tag{5.3}$$

La contrainte (5.2) impose qu'un seul couple c (configuration, translation) soit choisi pour un job i. La contrainte (5.3) impose qu'à chaque instant t l'ensemble des jobs qui tournent ne consomment pas plus que ce qui est disponible.

5.3.3 Relaxation du programme linéaire

Ce programme linéaire en nombres entiers est exponentiel en temps d'exécution. C'est pourquoi nous effectuons une relaxation. Nous transformons la variable binaire $x_{i,s}$ en une variable rationnelle avec $0 \leq x_{i,s} \leq 1$. Prenons maintenant une représentation matricielle et vectorielle des données et des variables. Chaque job i a un vecteur x_i dont chaque composante indique si le slot s correspondant est choisi. En concaténant chaque vecteur x_i, on obtient un vecteur x. Nous souhaitons que le vecteur x soit très creux : c'est à dire qu'il y a beaucoup de zéro et peu de valeurs différentes de zéro. Ceci revient à minimiser la norme ℓ_p du vecteur x qui se note

$$\|x\|_p = (\sum_k x_k^p)^{\frac{1}{p}}$$

.

Notons que la norme ℓ_1 d'un vecteur x est égale à la somme de ses composantes. Nous prenons $0 < p < 1$ car avec ces valeurs plus un vecteur est creux est plus sa norme ℓ_p est petite.

Dans notre problème le vecteur x_i a toutes ces composantes comprises entre 0 et 1 et la somme de ces composantes doit être égale à 1. De plus le vecteur x est la concaténation des vecteur x_i associé à chaque job i. Ainsi La somme des composantes du vecteur x est égale au nombre de jobs n : $\|x\|_1 = n$.

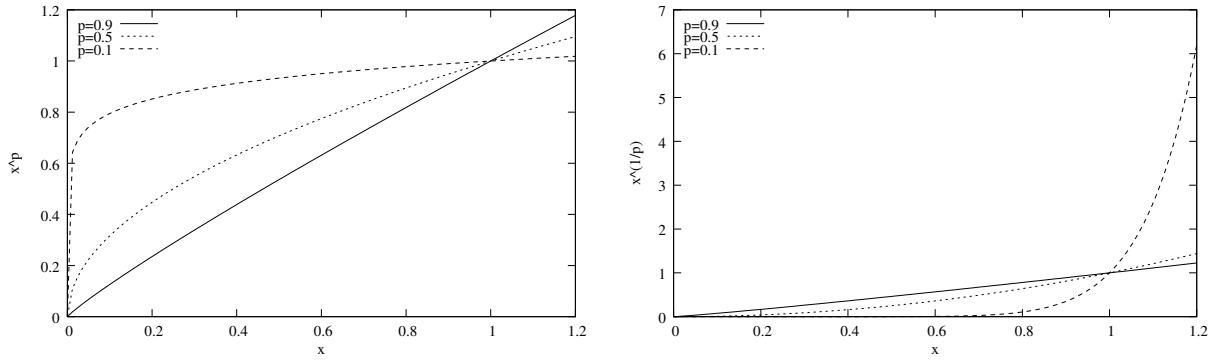
Prenons comme exemple deux vecteurs x et x' dont les normes ℓ_1 sont égales à 2 dans le cas où le problème consiste à ordonnancer 2 jobs. Comme nous pouvons le constater, comme le vecteur x est moins creux que le vecteur x' quelque soit la valeur $p \in]0,1[$ $\|x\|_p > \|x'\|_p$. Par

ailleurs, plus p est proche de 0 et plus le vecteur x a une forte pénalisation.

$$x = \begin{pmatrix} 0.8 \\ 0.2 \\ 0.9 \\ 0.1 \end{pmatrix}, \ \|x\|_{0.1} \approx 379161, \ \|x\|_{0.5} \approx 6.8, \ \|x\|_{0.9} = 2.27$$

et

$$x' = \begin{pmatrix} 0.99999 \\ 0.00001 \\ 1 \\ 0 \end{pmatrix}, \ \|x'\|_{0.1} \approx 15.6, \ \|x'\|_{0.5} \approx 1.006, \ \|x'\|_{0.9} = 1.00003$$



(a) la fonction x^p

(b) la fonction $x^{\frac{1}{p}}$

Figure 5.3 – Illustration de la norme ℓ_p

La courbe $x^{0.1}$ sur la figure 5.3a montre que la somme des composantes d'un vecteur x élevée à une puissance 0.1 est très élevée, si elles ne sont pas proches de 0. De plus la courbe $x^{\frac{1}{0.1}}$ sur la figure 5.3b croît très vite. Ces deux courbes explique la forte pénalisation par la norme ℓ_p d'un vecteur x qui n'est pas creux avec $p = 0.1$.

Nous nous servons de cette pénalisation pour faire tendre les valeurs des solutions du programme linéaire vers des nombres entiers. Ainsi nous prendrons une valeur $p = 0.1$ par la suite afin de forcer les solutions à tendre vers 0 ou 1.

5.3.4 Choix de la méthode de linéarisation de la fonction objectif

D'abord nous avons réfléchi sur l'expression de la fonction objectif ainsi que la manière de la linéariser. Nous avons pensé à la norme ℓ_p élevée à puissance p et sa linéarisation peut être réalisée au moyen de *Reweighted Linear Program*. Puis nous avons pensé à la norme ℓ_p qui peut être linéarisée grâce à son gradient. Dans les deux cas nous utilisons les solutions du programme linéaire de l'itération précédente pour résoudre celui de l'itération courante.

a Méthode avec *Reweighted Linear Program*

Comme la fonction exponentielle pour $p > 0$ est croissante, on peut aussi minimiser

$$\|x\|_p^p = \sum_k x_k^p$$

.

Comme à l'intérieur de la fonction objectif nous élevons x_k^p, la somme n'est pas linéaire. C'est pourquoi nous linéarisons la fonction x^p de la manière suivante en utilisant le principe du *Reweighted Linear Program* que Chartrand présente dans [26].

$$\begin{aligned} x^p &= x \times x^{p-1} \\ &= x \times \frac{1}{x^{1-p}} \end{aligned} \tag{5.4}$$

Le principe du *Reweighted Linear Program* repose sur le calcul(5.4). L'idée est de réutiliser l'ancienne valeur de la variable calculée par le programme linéaire à l'itération précédente.

Nous notons $x_k^{(iter)}$ la valeur de la composante k du vecteur x à l'itération précédente, ainsi x_k^p se calcule de la manière suivante :

$$x_k^p \leftarrow x_k \times \frac{1}{x_k^{(iter)(1-p)}}$$

La fonction objectif du programme linéaire à minimiser est maintenant :

$$\sum_k x_k \times \frac{1}{x_k^{(iter)(1-p)}}$$

Nous avons réalisé quelques essais, et nous nous sommes aperçus qu'au bout de quelques itérations le programme linéaire converge vers une solution non satisfaisante, c'est à dire qu'il ne donne pas un vecteur x creux. En effet pour quelques jobs i, le programme linéaire fait un choix décisif quant au couple (configuration, translation), cela signifie que x_i contient seulement un 1 et des 0. Mais pour d'autres jobs i, x_i contient des 0 et des valeurs rationnelles supérieures à 0, ni proches de 1 ni proches de 0.

b Méthode avec le gradient

La méthode de linéarisation précédente n'ayant pas donné de solutions satisfaisantes, nous avons donc pensé à minimiser non pas la norme ℓ_p du vecteur x, mais la norme ℓ_p de chaque vecteur x_i. Ainsi nous souhaitons minimiser :

$$\sum_i \|x_i\|_p$$

Mais cette expression n'est pas linéaire. Pour intégrer une linéarité dans la fonction objectif

du programme linéaire, nous avons utilisé la méthode faisant intervenir le gradient,

avec pour objectif de minimiser

$$\sum_i \|x_i\|_p \tag{5.5}$$

sous les contraintes (5.2) et (5.3).

5.4 Méthode avec le gradient

En remplaçant $\|x_i\|_p$ par l'expression de la norme ℓ_p et en tenant compte des notations données au paragraphe 5.3.2, nous avons :

$$\forall i,\ f(x_i) = \|x_i\|_p = \left(\sum_{s=1}^{s=nslot_i} x_{i,s}^p \right)^{\frac{1}{p}}$$

En utilisant le résultat $x_i^{(iter)}$ de l'itération précédente comme dans la section 5.3.4.a on calcule la norme ℓ_p en posant $h = x_i - x_i^{(iter)}$ avec une formule de Taylor :

$$f(x_i) = f(x_i^{(iter)}) + \langle \nabla f(x_i^{(iter)}), x_i - x_i^{(iter)} \rangle + o(x_i - x_i^{(iter)})$$

et pour calculer $\nabla f(x_i^{(iter)})$, nous calculons les dérivées par rapport à chaque composante :

$$\begin{aligned}
\forall i, s,\ \frac{\partial f(x_i^{(iter)})}{\partial x_{i,s}^{(iter)}} &= p \times x_{i,s}^{(iter)p-1} \times \frac{1}{p} \times \left(\sum_{e=1}^{e=nct_i} x_{i,e}^{(iter)p} \right)^{\frac{1}{p}-1} \\
&= x_{i,s}^{(iter)p-1} \times \left(\sum_{e=1}^{e=nct_i} x_{i,e}^{(iter)p} \right)^{\frac{1}{p}-1} \\
&= x_{i,s}^{(iter)p-1} \times f\left(x_i^{(iter)}\right)^{1-p}
\end{aligned}$$

Comme f n'est pas différentiable en 0, posons $g \in \partial f$ un des sous-gradients possibles de f

$$g_{i,s} = \begin{cases} x_{i,s}^{p-1} \times f(x_i)^{1-p} & \text{si } x_{i,s} \neq 0 \\ 0 & \text{sinon.} \end{cases} \tag{5.6}$$

Nous utilisons ce sous-gradient g à la place de ∇f.

Nous avons ainsi défini notre fonction objectif dont le but est de forcer la solution à contenir le maximum de 0 et le bon nombre de 1. Nous avons proposer une linéarisation de cette fonction afin de permettre une résolution par un solveur de programme linéaire.

Algorithme 11: Un algorithme d'approximation par linéarisations successives

```
1  lb ← borne inférieure du makespan
2  sched ← calculer un ordonnancement avec l'agorithme LTF
3  listMakespan ← makespan(sched)
4  T ← listMakespan
5  end ← faux
6  incT ← faux
7  tant que T > lb et non end faire
8      proc ← calculer les configurations des jobs 𝒥 ; run ← calculer tous les slots possibles avec (𝒥, m, T) ;
       iter ← 1 ; found ← faux
9      ∀i, k, x_{i,k}^{(iter)} ← 0
10     tant que iter < maxIter et non found faire
11         mettre la fonction objectif de 𝓛𝒫(𝒥, m, T, proc, run) à
           ∑_i^{|𝒥|} f(x_i^{(iter)}) + ⟨∇f(x_i^{(iter)}), x_i − x_i^{(iter)}⟩
12         x ← exécuter 𝓛𝒫(𝒥, m, T, proc, run)
13         si ∀i, x_i contient exactement un "1" alors
14             sched ← convertir en un ordonnancement (x, proc, run)
15             T ← makespan(sched)
16             T ← T − 1
17             found ← vrai
18             si incT = vrai alors
19                 end ← vrai
20         ∀i, k, x_{i,k}^{(iter)} ← x_{i,k}
21         iter ← iter + 1
22     si non found alors
23         si T = listMakespan alors
24             incT ← vrai
25         si incT = vrai alors
26             T ← T + 1
27         sinon
28             end ← vrai
29 retourner sched
```

c Algorithme

La méthode est implémentée dans l'algorithme 11. Il commence par initialiser un vecteur nul pour x. D'abord nous calculons une borne inférieure pour le makespan à la ligne 1, qui est égale au maximum entre la durée du job le plus long et $\frac{\sum_i reqproc_i \times reqtime_i}{m}$. L'horizon de temps T est initialisé au makespan de l'algorithme LTF. Si le programme linéaire $\mathcal{LP}$ trouve une solution satisfaisante (ligne 13), l'algorithme réduit l'horizon de temps (ligne 16) jusqu'à ce qu'il le ne puisse plus (line 28) avant $maxIter$ itérations. S'il ne trouve pas de solution satisfaisante avec $T = listmakespan$ avant $maxIter$ itérations (ligne 22), il augmente l'horizon de temps T (ligne 26). Pour un horizon de temps T donné il met à jour la fonction objectif du programme linéaire à chaque itération (ligne 11) selon la formule de Taylor avec le sous-gradient.

Il convient de noter, que la forme de l'algorithme ressemble à l'approximation duale de Shmoys et Hochbaum [56] pour résoudre le problème de minimisation du makespan de l'ordonnancement d'une collection de jobs séquentiels sur une machine à m machines identiques. En effet, ils supposent qu'ils possèdent une procédure résolvant le problème de bin packing à une dimension avec un nombre de boîtes fixé. Ils itèrent sur cette procédure par une dichotomie en changeant le nombre de boîtes jusqu'à ce qu'ils ne peuvent plus trouver de solutions satisfaisantes.

Durant la phase d'expérimentation un problème est apparu dans la résolution du programme linéaire. Des solutions satisfaisantes pour l'algorithme 11 sont uniquement détectées (à la ligne 13) si tous les jobs i ont leur vecteur x_i avec exactement un "1", car l'algorithme a été conçu pour obtenir des solutions exactes. En fait, pour un horizon de temps T donné, les approximations par les programmes linéaires successifs arrivent à trouver un ordonnancement pour la plupart des jobs de la collection mais ils laissent quelques jobs j de la collection dans un état d'ordonnancement flou. C'est-à-dire que les vecteurs x_i contiennent exactement un "1" alors que les vecteurs x_j contiennent des valeurs rationnelles entre 0 et 1. Dans ce cas, l'algorithme continue, même si une composante de x_j est proche de 1, jusqu'à ce que $maxIter$ itérations soient effectuées. En procédant ainsi, les temps de calcul étaient très longs et des solutions trouvées étaient inefficaces.

Par conséquent nous avons modifié l'algorithme 11 et son critère de détection de solution de la ligne 13 comme suit : quand le programme linéaire trouve un ordonnancement exacte pour les jobs i, dont les x_i contiennent exactement un "1", on garde ces jobs i dans l'ordonnancement et on ordonnance le reste des jobs pour lesquels le programme linéaire a trouvé un ordonnancement flou, grâce à l'algorithme LTF. Si une solution plus courte que l'horizon de temps T est trouvée, alors la variable $found$ est mise à $vrai$ et l'algorithme continue à itérer. Nous obtenons l'algorithme 12.

5.5 Résultats expérimentaux des algorithmes 11 et 12

Nous présentons dans ce paragraphe les résultats obtenus avec les deux versions de l'algorithme et nous comparons leurs performances avec le LTF. Nous évaluons les performances pour le cas à la fois rigide et moldable.

Nous avons développé un simulateur de cluster homogène fondé sur une architecture maître-esclave. Ce simulateur sert aussi à vérifier les ordonnancements obtenus par les différents algo-

Algorithme 12: L'algorithme d'approximation par linéarisations successives amélioré

```
1  lb ← borne inférieure du makespan
2  sched ← calculer un ordonnancement avec l'agorithme LTF
3  listMakespan ← makespan(sched)
4  T ← listMakespan
5  end ← faux
6  incT ← faux
7  tant que T > lb et non end faire
8      proc ← calculer les configurations des jobs 𝒥
9      run ← calculer tous les slots possibles avec (𝒥, m, T)
10     iter ← 1
11     found ← faux
12     ∀i, k, x_{i,k}^{(iter)} ← 0
13     tant que iter < maxIter et non found faire
14         mettre la fonction objectif de 𝓛𝒫(𝒥, m, T, proc, run) à
           ∑_i^{|𝒥|} f(x_i^{(iter)}) + ⟨∇f(x_i^{(iter)}), x_i − x_i^{(iter)}⟩
15         x ← exécuter 𝓛𝒫(𝒥, m, T, proc, run)
16         si ∀i, x_i contient exactement un "1" alors
17             bestSched ← convertir en un ordonnancement (x, proc, run)
18             T ← makespan(bestSched)
19             T ← T − 1
20             found ← vrai
21             if incT = vrai then
22                 end ← vrai
23         sinon
24             jobsAlreadySched ← garder les jobs i tel que x_i contient exactement un "1"
25             ordonnancer les jobs jobsAlreadySched
26             sched ← ordonnancer avec LTF le reste des jobs (𝒥 − jobsAlreadySched)
27             innerMakespan ← makespan(sched)
28             si innerMakespan < T alors
29                 bestSched ← sched
30                 T ← T − 1
31                 found ← vrai
32                 si incT = vrai alors
33                     end ← vrai
34         ∀i, k, x_{i,k}^{(iter)} ← x_{i,k}
35         iter ← iter + 1
36     si non found alors
37         si T = listMakespan alors
38             incT ← vrai
39         si incT = vrai alors
40             T ← T + 1
41         sinon
42             end ← vrai
43 retourner bestSched
```

$reqtime_i$ (en heures)	$lp\_reqtime_i$ (en unités de temps)
5	1
25	5
45	9
125	25
55	11

Table 5.1 – Exemple de durées d'exécution de jobs d'une collection à ordonnancer

rithmes. Le simulateur utilise SimGrid [25] et son API MSG. Il prend en entrée un workload et il sort un ordonnancement. Nous utilisons les bibliothèques de Gurobi [2] pour résoudre les programmes linéaires. Les calculs ont été réalisés sur les processeurs du Mésocentre[2] de Calcul de France-Comté équipé d'une machine à 32 cœurs Intel Xeon X7550 cadencés à 2.0 Ghz.

Pour simuler une collection de jobs parallèles, nous utilisons des workloads synthétiques générés avec des lois uniformes. Nous définissons la granularité d'un workload comme étant le rapport entre la durée du job le plus long sur la durée du job le plus court. Nous définissions le pas de discrétisation δ comme étant le temps réel égal à une unité de temps.

Afin de simplifier la compréhension des algorithmes nous supposons, dans un premier temps que le temps d'exécution requis $reqtime_i$ pour chaque job i est aligné avec le pas de discrétisation δ, Mathématiquement cela signifie que quel que soit le job i, δ est un diviseur de $reqtime_i$. Ainsi l'ensemble des temps requis $reqtime_i$ des jobs i ont un plus grand diviseur commun gcd et respectent les relations suivantes :

$$\begin{cases} \delta = gcd \\ reqtime_i = gcd \times lp\_reqtime_i \end{cases}$$

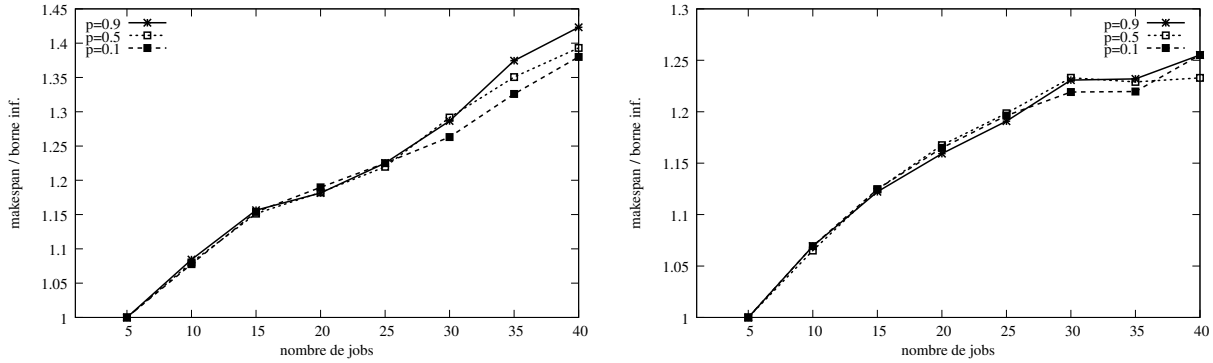
où $lp\_reqtime_i$ représente le temps requis (en unité de temps) du job i considéré par notre algorithme reposant sur des approximations linéaires successives. Le tableau 5.1 donne l'exemple d'une collection de 5 jobs dont les durées d'exécution respectives sont données dans la colonne $reqtime_i$ en heures et dans la colonne $lp\_reqtime_i$ en unités de temps. Dans cet exemple le pas de discrétisation de temps δ est de 5 heures.

De plus, comme nous résolvons un grand nombre de fois un programme linéaire au cours des itérations, nous limitons expressément la granularité pour obtenir des résultats des simulations dans un délai raisonnable. Bien que ces expériences ne sont pas très réalistes, nous montrons dans un premier temps que la méthode fondée sur les approximations linéaires successives fonctionne et donne de bons résultats. Dans un deuxième temps nous montrons que nous pouvons ordonnancer avec la méthode n'importe quelle collection de jobs dont la granularité est grande.

Avant de réaliser des expériences il est nécessaire de choisir les paramètres adéquats. En premier lieu, nous avons choisi de réaliser les expériences avec une norme ℓ_p où p est égale à 0.1. En effet, comme nous l'avons mentionné précédemment plus p est petit et plus sa pénalisation sur les vecteurs x_i est grande si les x_i ne sont pas suffisamment creux. Ceci est corroboré par la figure 5.4a. Les figures 5.4a et 5.4b montrent les performances de notre algorithme pour plusieurs valeurs de p. Nous remarquons globalement que pour les jobs rigides les meilleures valeurs de

[2] `http://meso.univ-fcomte.fr/`

makespan sont obtenues avec $p = 0.1$. En revanche pour les jobs moldables les différentes valeurs de p donnent des valeurs de makespan quasiment identiques. Comme nous nous sommes aperçus que l'algorithme mettait beaucoup de temps à converger, nous avons limité le nombre d'itérations $maxIter$ à 15000. De plus le nombre de variables $x_{i,s}$ est fonction de $T \times n$ où T est l'horizon du temps et n le nombre de jobs à ordonnancer. C'est pourquoi, pour cette simulation, nous limitons le nombre de jobs à 40 sur une machine de 32 PEs.



(a) jobs rigides

(b) jobs moldables

Figure 5.4 – Performances avec 32 PEs et différentes valeurs de p

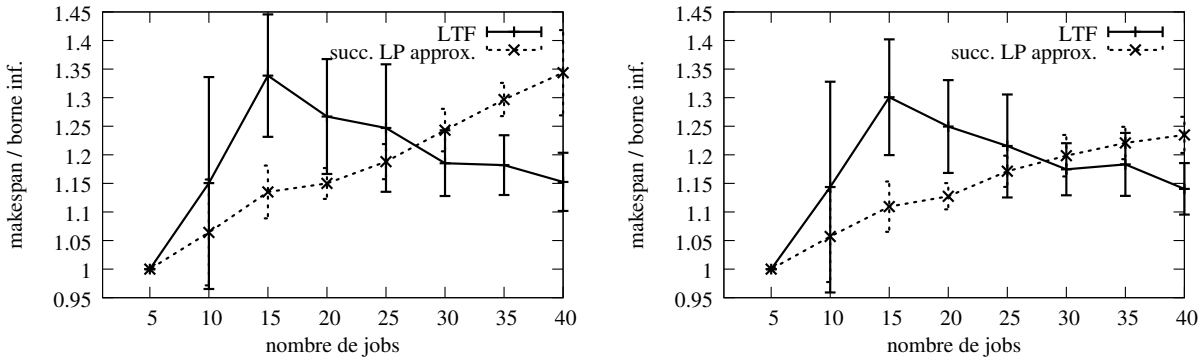
5.5.1 Performances de l'algorithme 11

Dans les premières expériences, nous avons lancé des simulations avec une norme ℓ_p où $p = 0.1$, $maxIter = 15000$ pour la première version de l'algorithme avec une machine simulée constituée de 64 PEs. Nous avons ordonnancé une collection d'au plus 60 jobs. Les résultats sont décevants : ils sont vraiment loin derrière LTF et les temps de calcul sont très longs.

Nous avons alors relancé les expériences avec moins de jobs. Nous avons utilisé $p = 0.1$, $maxIter = 15000$ comme paramètres. La plate-forme simulée est constituée de 32 PEs et le nombre de PEs requis pour chaque job est uniformément choisi entre 1 et 8. La granularité est fixée à 25. Pour chaque nombre de jobs dans la collection, nous exécutons 20 expériences. Ensuite nous retirons l'expérience qui a donné le meilleur résultat et celle qui a donné le pire résultat afin de réduire la variance statistique. Après ceci nous calculons la moyenne des 18 rapports du makespan à sa borne inférieure.

La figure 5.5a montre le rapport du makespan à la borne inférieure du makespan optimal en fonction du nombre de jobs rigides, alors que la figure 5.5b montre ce ratio pour les jobs moldables. Dans les figures les algorithmes sont noté *succ. LP approx* pour notre algorithme et *LTF* pour l'algorithme LTF. Les figures montrent également l'écart-type σ : la hauteur d'une ligne verticale est égale à 2σ.

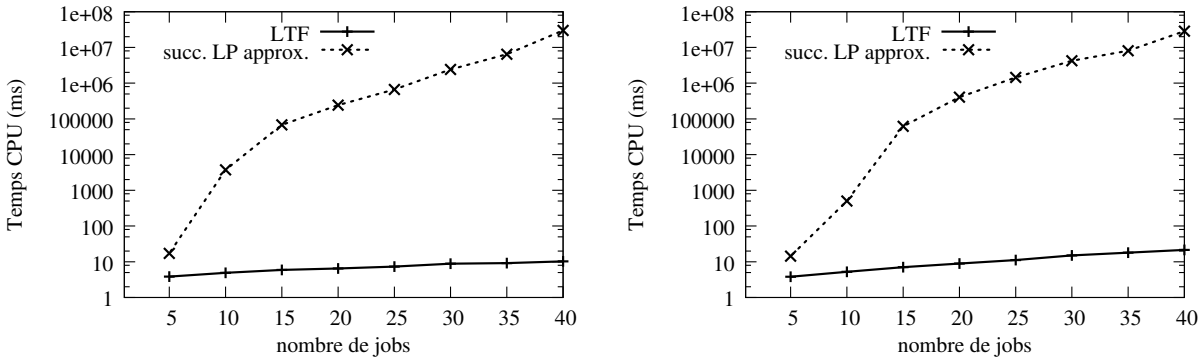
Nous remarquons que pour moins de 25 jobs l'algorithme d'approximations par linéarisations linéaires successives donne de meilleurs résultats que l'algorithme LTF, mais ce dernier est meilleur que notre algorithme à partir de 25 jobs. Il est à noter que LTF tend rapidement vers 1.1. Cela signifie, d'une part, que probablement il trouve dans la plupart des cas, une solution optimale ou proche de l'optimale, et d'autre part, qu'il est difficile de trouver de meilleures



(a) jobs rigides **(b)** jobs moldables

Figure 5.5 – Performances avec 32 PEs

solutions. De plus, avec plus de 25 jobs, le problème devient, si complexe que notre algorithme doit augmenter l'horizon de temps T pour trouver une solution. En dessous du seuil des 25 jobs, le gain maximal est d'environ 15% pour 16 jobs rigides ou moldables. Nous pouvons noter que l'écart-type des expériences pour notre nouvelle approche est plus faible que l'écart-type des expériences de l'algorithme LTF. Pour les deux algorithmes nous pouvons comprendre aisément que pour 5 jobs, l'optimal soit atteint. Cela s'explique par la mise en place des expériences : le nombre de PEs dont chaque job a besoin est choisi uniformément entre 1 et 8. Par conséquent, tous les jobs commencent au temps 0. Ceci explique aussi le pic avec 15 jobs qui ne commencent pas nécessairement au temps 0. Nous remarquons aussi que pour les jobs rigides ou moldables notre algorithme a le même comportement, c'est-à-dire, quand le nombre de jobs augmente, le rapport du makespan à la borne inférieure augmente.



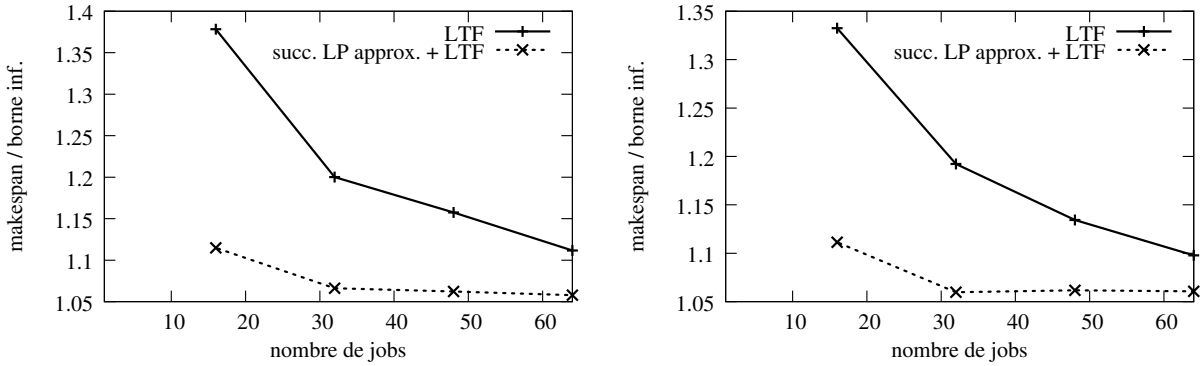
(a) jobs rigides **(b)** jobs moldables

Figure 5.6 – Temps de calcul avec 32 PEs

Comme nous le voyons à travers la figure 5.6a et la figure 5.6b, l'algorithme 11 met énormément de temps dans ses calculs à la fois pour les jobs moldables et les job rigides comparé à l'algorithme LTF qui ne prend que très peu de temps. Dans le paragraphe suivant nous évaluons les performances de la version améliorée de notre algorithme.

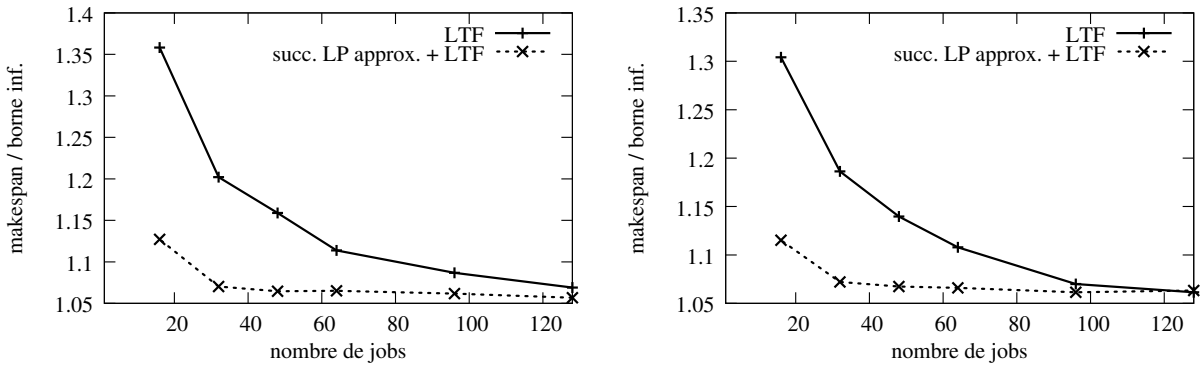
5.5.2 Performances de l'algorithme 12

Afin d'évaluer la version améliorée de l'algorithme, nous lançons des simulations sur deux machines simulées, une machine constituée de 64 PEs et une autre de 128 PEs. Le nombre de PEs requis pour chaque job est uniformément choisi entre 1 et 16 pour la machine à 64 PEs, et entre 1 et 32 pour la machine à 128 PEs. La granularité est de 25 pour la machine à 64 PEs et 10 pour la machine à 128 PEs. Nous utilisons $p = 0.1$ et $maxIter = 200$ comme paramètres. Pour chaque nombre de jobs de la collection, nous lançons 40 simulations.



(a) jobs rigides **(b)** jobs moldables

Figure 5.7 – Performances des algorithmes avec 64 PEs



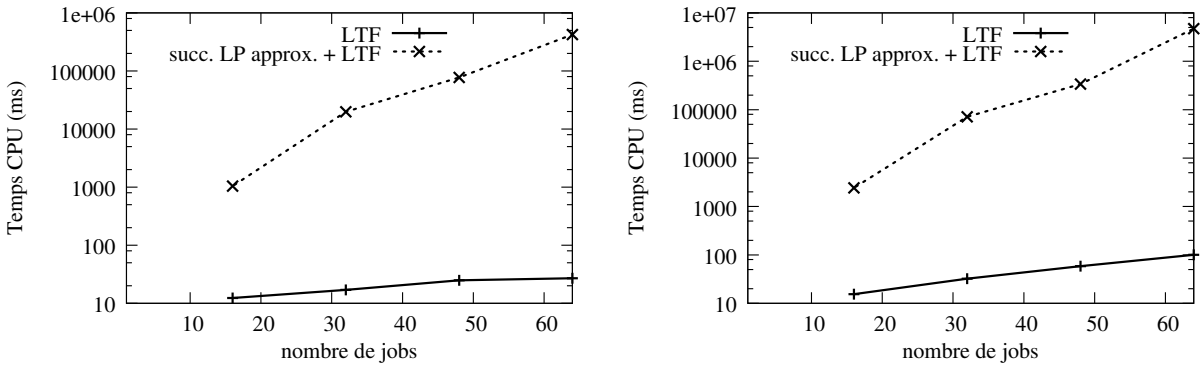
(a) jobs rigides **(b)** jobs moldables

Figure 5.8 – Performances des algorithmes avec 128 PEs

La figure 5.7a montre le ratio du makespan à l'optimal en fonction du nombre de jobs rigides dans un cluster de 64 PEs et dans figure 5.7b nous considérons l'ordonnancement de jobs moldables. Les performances de la nouvelle approche sont meilleures que l'algorithme LTF pour les jobs moldables et rigides. La Figure 5.8a et la figure 5.8b donnent les résultats pour une machine avec 128 PEs.

Nous remarquons que dans les quatre cas les gains apportés notre approche par rapport à l'algorithme LTF peuvent atteindre 20%. Avec une plate-forme de 128 PEs les figures montrent une amélioration des résultats obtenus avec l'algorithme LTF par rapport au cas où la plate-forme comporte 64 PEs. En revanche le comportement de notre algorithme reste le même avec

64 ou avec 128 PEs. Cela est probablement dû au fait que nos solutions sont très proches de l'optimal et que la marge de progression est très faible.



(a) jobs rigides

(b) jobs moldables

Figure 5.9 – Temps de calcul pour ordonnancer une collection de jobs dans une machine de 64 PEs

Les figures 5.9a et 5.9b montrent les temps d'exécution des deux algorithmes. Nous remarquons que l'algorithme hybride est plus rapide que la version originale, mais il est beaucoup plus lent que l'algorithme LTF.

5.6 Performances avec différentes lois de probabilité sur le nombre de PEs

Afin d'observer le comportement de notre algorithme et d'évaluer ses performances, nous mettons en place une suite d'expériences avec différentes lois de probabilité afin de générer aléatoirement le nombre de PEs requis pour l'ensemble des jobs du workload synthétique.

5.6.1 Loi gaussienne

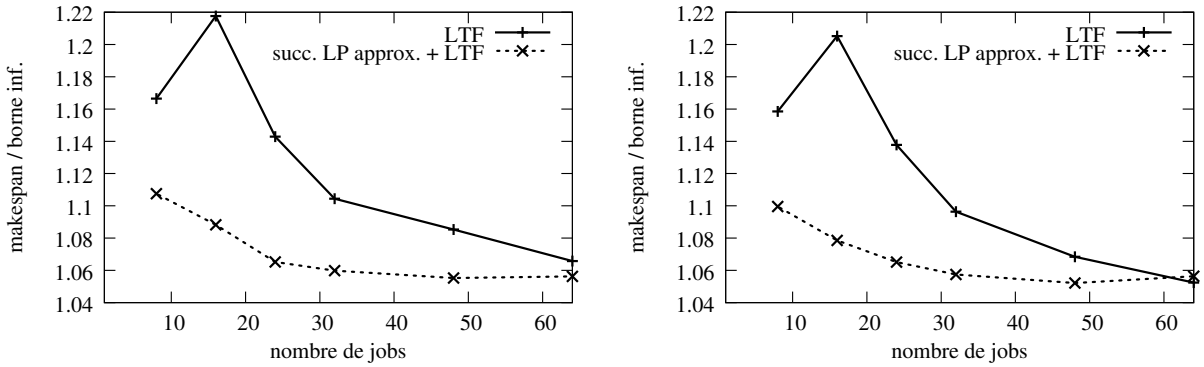
Pour les expériences décrites dans ce paragraphe, nous utilisons une loi gaussienne pour générer le nombre requis de PEs par les jobs. La loi gaussienne a une fonction de densité égale à :

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

où le paramètre μ est la moyenne et σ est l'écart-type. L'avantage d'une loi gaussienne est qu'elle favorise le mélange de jobs dont le niveau de parallélisme peut être contrôlé. Les jobs requièrent beaucoup de PEs en imposant que la moyenne μ soit importante ou bien peu de PEs en imposant que μ soit petite. L'inconvénient d'une loi gaussienne est que les valeurs de PEs générées sont centrées autour de cette moyenne μ, ce qui n'est pas assez réaliste.

a Avec une plate-forme de 64 PEs

Les paramètres des expériences sont les suivants. La granularité est de 25, $p = 0.1$ et $maxIter = 200$. Pour chaque taille de la collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 20 avec une loi gaussienne. Afin de simuler des collections de jobs avec un fort niveau de parallélisme, le paramètre μ (espérance) de la gaussienne est de 10. Le paramètre σ (écart-type) de la gaussienne est de 4 afin que les jobs aient des niveaux de parallélisme dispersés. Les figures 5.10a et 5.10b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. Notre algorithme est meilleur que LTF à la fois pour les jobs moldables et les job rigides. Néanmoins avec une collection de 64 jobs moldables, LTF rejoint les performances obtenues par notre algorithme.



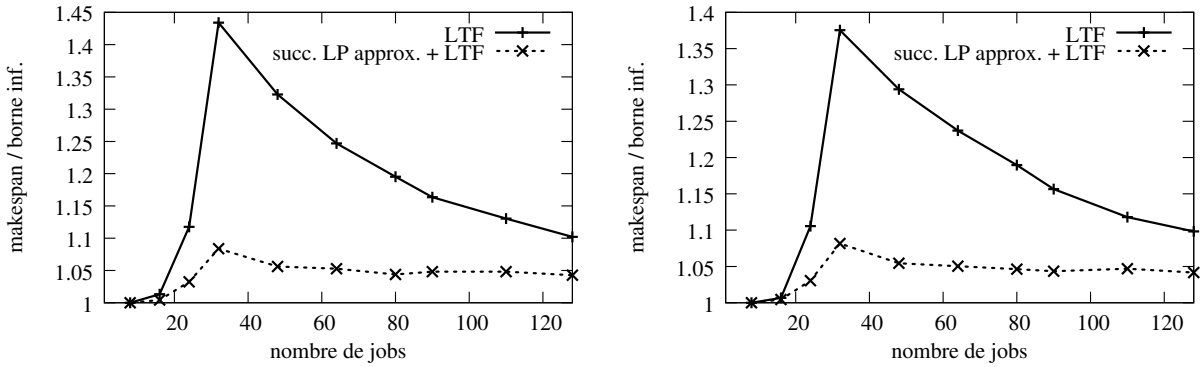
(a) rigid jobs **(b)** moldable jobs

Figure 5.10 – Performances de l'algorithme avec 64 PEs, loi gaussienne : $\mu = 10$ et $\sigma = 4$

Maintenant, un paramètre de la loi gaussienne change. Le paramètre σ de la loi gaussienne est toujours de 4. Le paramètre μ de la loi gaussienne est de 2. Donc il y a davantage de jobs qui requièrent un petit nombre de PEs que de jobs qui demandent beaucoup de PEs. Les figures 5.11a et 5.11b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. Notre algorithme est meilleur que LTF à la fois pour les jobs moldables et les jobs rigides pour chaque taille de collection. En considérant, la distribution gaussienne précédente ($\mu = 10$) et cette distribution ($\mu = 2$), on peut supposer que plus le nombre de PEs requis pour les jobs est petit et plus notre algorithme a des facilités pour trouver de bonnes solutions. Même pour ordonnancer un lot de 64 jobs avec cette distribution ($\mu = 2$), notre algorithme est loin devant LTF, alors qu'avec l'autre distribution ($\mu = 10$), LTF le rattrapait.

b Avec une plate-forme de 128 PEs

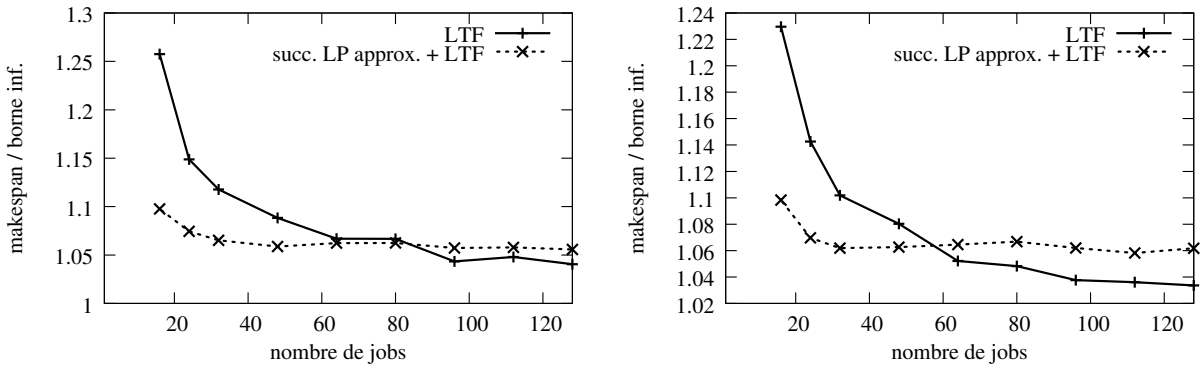
Les paramètres des expériences sont les suivants. La granularité est de 10, $p = 0.1$ et $maxIter = 200$. Pour chaque taille de la collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 40 avec une loi gaussienne. Le paramètre σ (écart-type) de la gaussienne est de 8. Le paramètre μ (moyenne) de la gaussienne est de 20. Les figures 5.12a et 5.12b montrent le ratio du makespan à l'optimal en fonction du nombre



(a) jobs rigides **(b)** jobs moldables

Figure 5.11 – Performances de l'algorithme avec 64 PEs, loi gaussienne : $\mu = 2$ et $\sigma = 4$

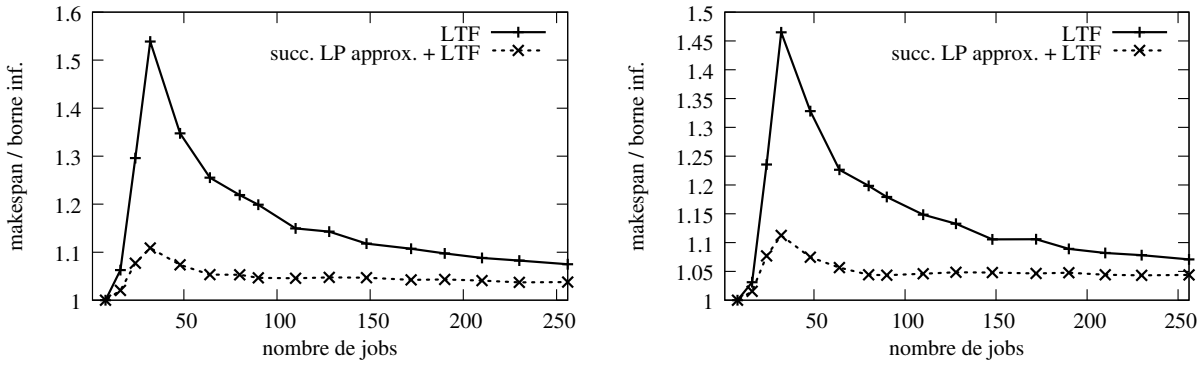
de jobs rigides et moldables pour cette distribution. On s'aperçoit qu'avec cette distribution, notre algorithme a des difficultés pour trouver des ordonnancements meilleurs que LTF pour des collections dont la taille est supérieure à 60 jobs. Mais il faut remarquer que la marge de manœuvre pour notre algorithme est très faible, car l'ordonnancement d'une collection de 128 jobs moldables par LTF est inférieur à 1.04, ce qui est très faible. De plus il convient de noter que l'on se compare à la borne inférieure du makespan et non au makespan optimal, ce qui veut dire que le LTF trouve probablement un ordonnancement très proche du makespan optimal.



(a) jobs rigides **(b)** jobs moldables

Figure 5.12 – Performances de l'algorithme avec 128 PEs, loi gaussienne : $\mu = 20$ et $\sigma = 8$

Dans une autre série d'expériences, un paramètre de la loi gaussienne change. Le paramètre σ de la loi gaussienne est toujours de 8. Le paramètre μ de la loi gaussienne est de 4. Donc il y a davantage de jobs qui requièrent un petit nombre de PEs que de jobs qui demandent beaucoup de PEs. Les figures 5.13a et 5.13b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. Nous remarquons que les performances de notre algorithme sont bien meilleures que celles de LTF, pour n'importe quelle taille de collection. Notre hypothèse se vérifie ici avec des workloads de jobs dont le nombre de PEs requis est faible ($\mu = 4$).



(a) jobs rigides **(b)** jobs moldables

Figure 5.13 – Performances de l'algorithme avec 128 PEs, loi gaussienne : $\mu = 4$ et $\sigma = 8$

5.6.2 Mélange de lois de probabilité

Dans une autre série d'expériences nous souhaitons observer le comportement des algorithmes testés en mélangeant les lois de probabilité du nombres de PEs requis par les jobs. La fonction de mélange de K densités de probabilité est définie comme suit :

$$f(x) = \sum_{k=1}^{K} \pi_k \Phi_k(x)$$

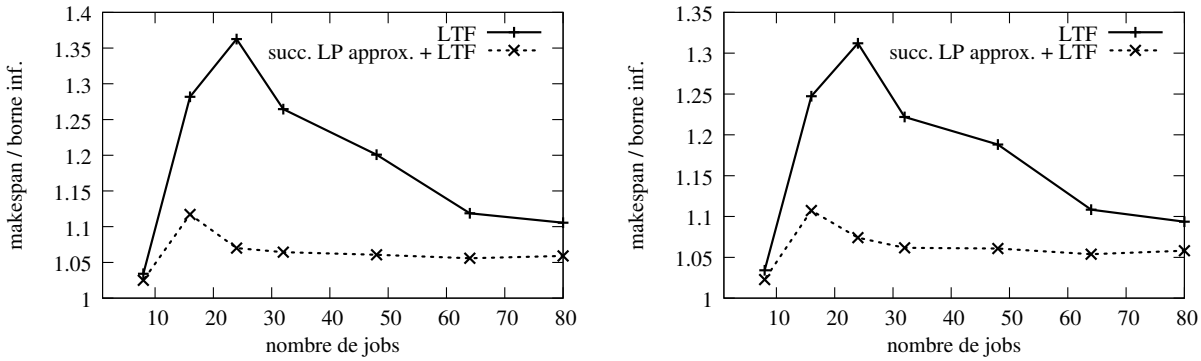
où $\Phi_k(x)$ est la fonction de densité de la loi k, π_k est le poids de la loi k et $\sum_{k=1}^{K} \pi_k = 1$

L'avantage d'utiliser un mélange de deux gaussiennes est que l'on peut générer des valeurs qui se répartissent autour de deux moyennes. Ainsi nous pouvons mieux contrôler les valeurs générées de PEs afin de simuler une distribution de jobs qui requièrent un petit nombre de PEs et d'autres qui requièrent un grand nombre de PEs. L'inconvénient d'un mélange de deux gaussiennes est que les valeurs se répartissent autour de deux moyennes. Ce qui rend les valeurs de PEs pas assez réalistes.

a Avec une plate-forme de 64 PEs

Les paramètres des expériences sont les suivants. La granularité est de 25, $p = 0.1$ et $maxIter = 200$. Pour chaque taille de la collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 20 avec un mélange de deux lois gaussiennes. Afin de simuler une distribution de jobs qui requièrent un petit nombre de PEs et d'autres qui requièrent un grand nombre de PEs, le paramètre μ_1 de la gaussienne 1 est de 2 et le paramètre μ_2 de la gaussienne 2 est de 10. Afin que les valeurs soit très diversifiée, le paramètre σ_1 de la gaussienne 1 est de 4 et le paramètre σ_2 de la gaussienne 2 est de 4. Les figures 5.14a et 5.14b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. On remarque que notre algorithme est bien meilleur que l'algorithme LTF. En effet, les valeurs pour les paramètres choisies des deux gaussiennes entraînent un mélange de jobs à la fois avec un faible de niveau de parallélisme et un niveau plus important. L'algorithme LTF fait des choix gloutons en commençant par les jobs les plus

gros alors que notre approche profite de l'hétérogénéité des jobs pour mieux les placer.



(a) jobs rigides

(b) jobs moldables

Figure 5.14 – Performances de l'algorithme avec 64 PEs, mélanges de deux gaussiennes : $\pi_1 = \pi_2 = 0.5$ $\mu_1 = 2$, $\mu_2 = 10$ et $\sigma_1 = \sigma_2 = 4$

b Avec une plate-forme de 128 PEs

Les paramètres des expériences sont les suivants. La granularité est de 10, $p = 0.1$ et $maxIter = 200$. Pour chaque taille de la collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 20 avec un mélange de deux lois gaussiennes. Afin de simuler une distribution de jobs qui requièrent un petit nombre de PEs et d'autres qui requièrent un grand nombre de PEs nous choisissons les valeurs suivantes pour les paramètres. Le paramètre σ_1 de la gaussienne 1 est de 8. Le paramètre μ_1 de la gaussienne 1 est de 4. Le paramètre σ_2 de la gaussienne 2 est de 8. Le paramètre μ_2 de la gaussienne 2 est de 20. Les figures 5.15a et 5.15b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. À partir de 150 jobs notre algorithme se fait rattraper par l'algorithme LTF. En fait notre algorithme ne donne pas de moins bons ordonnancements, mais c'est l'algorithme LTF qui profite du nombre de jobs importants pour combler les trous.

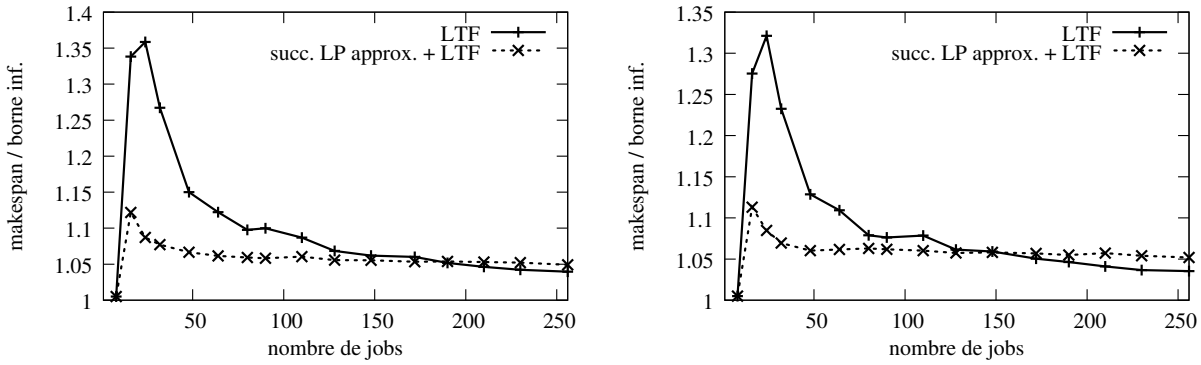
5.6.3 Loi de Cauchy

Nous utilisons une loi de Cauchy pour générer aléatoirement le nombre de PEs requis pour l'ensemble des jobs du workload. La loi de Cauchy a une fonction de densité égale à :

$$f(x, m, \gamma) = \frac{1}{\pi}\left(\frac{\gamma}{\gamma^2 + (x-m)^2}\right)$$

où le paramètre m est la médiane et γ un facteur d'échelle. Nous utilisons une loi de Cauchy car cette loi est spéciale : elle ne possède ni espérance ni variance. Du fait de cette singularité, nous souhaitons connaître le comportement de notre algorithme avec cette loi.

Le premier avantage d'une loi de Cauchy pour générer des valeurs de PEs requis pour les jobs est que les valeurs ne sont pas regroupées autour d'une moyenne contrairement à une loi gaussienne. Le second avantage est que nous pouvons tout de même contrôler la répartition du

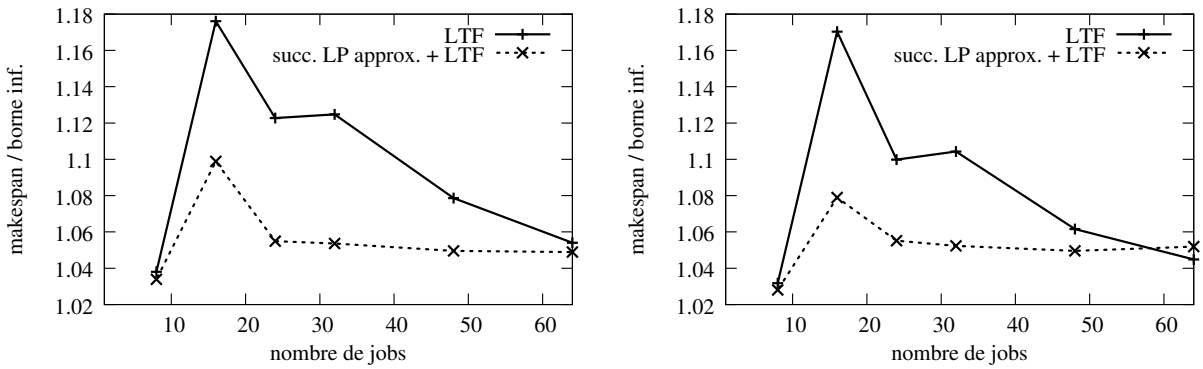


(a) jobs rigides **(b)** jobs moldables

Figure 5.15 – Performances de l'algorithme avec 128 PEs, mélanges de deux gaussiennes : $\pi_1 = \pi_2 = 0.5$ $\mu_1 = 4$, $\mu_2 = 20$ et $\sigma_1 = \sigma_2 = 8$

niveau de parallélisme des jobs grâce à la médiane. Ainsi les valeurs de PEs requis pour les jobs sont plus réalistes.

Avec une plate-forme de 64 PEs Les paramètres des expériences sont les suivants. La granularité est de 25, $p = 0.1$ et $maxIter = 200$. Pour chaque taille de collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 20 avec une loi de Cauchy. Le paramètre γ (facteur d'échelle) de la loi de Cauchy est de 1. Le paramètre m (médiane) de la loi de Cauchy est de 10. Les figures 5.16a et 5.16b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. Nous remarquons qu'en fonction des tailles de collections de jobs, le ratio évolue de manière moins monotone qu'avec une loi gaussienne aussi bien pour les jobs moldables que les jobs rigides. Cela peut s'expliquer par le caractère singulier de la loi de Cauchy. On remarque qu'à partir de 64 jobs notre algorithme se fait dépasser par LTF.

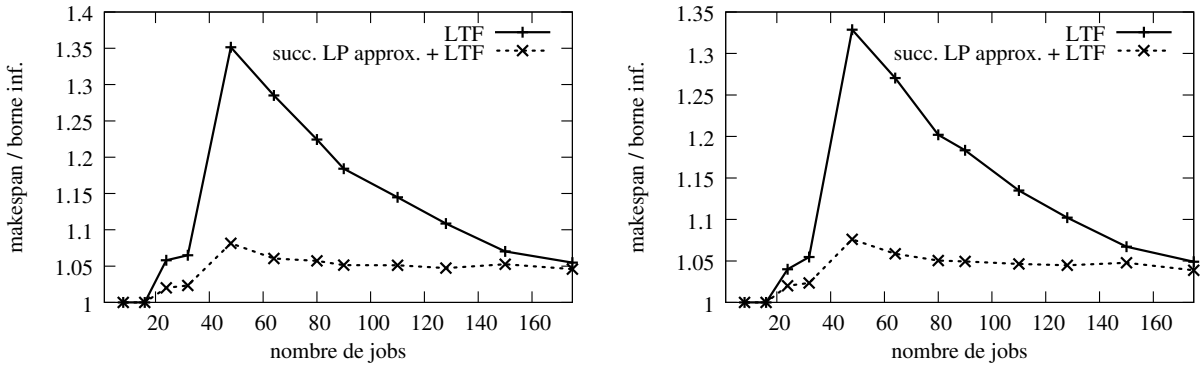


(a) jobs rigides **(b)** jobs moldables

Figure 5.16 – Performances de l'algorithme avec 64 PEs, loi de Cauchy : $m = 10$ et $\gamma = 1$

Maintenant, un paramètre de la loi de Cauchy change. Le paramètre γ de la loi de Cauchy est toujours de 1. Le paramètre m de la loi de Cauchy est de 2. Donc il y a plus de jobs qui requièrent

un petit nombre de PEs que de jobs qui demandent beaucoup de PEs. Les figures 5.17a et 5.17b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. Avec une médiane $m = 2$, les effets de la loi de Cauchy s'estompe, les deux courbes de ratio deviennent plus lisses et moins chaotiques. Cela s'explique par la médiane faible $m = 2$: la moitié des jobs requièrent un nombre de PEs inférieur ou égal à 2, par conséquent, ces petits jobs peuvent être ordonnancés facilement et donnent un ratio qui diminue progressivement avec le nombre de jobs de la collection. On remarque que pour une collection de 48 jobs, l'écart entre LTF et notre algorithme est d'environ 30%. Avec cette distribution, notre algorithme donne un ratio presque constant (≈ 1.05) à partir de 60 jobs.



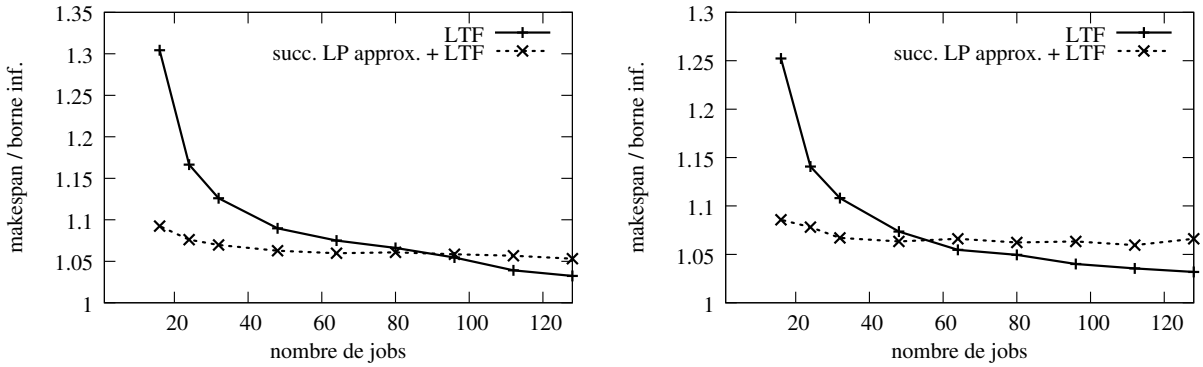
(a) jobs rigides **(b)** jobs moldables

Figure 5.17 – Performances de l'algorithme avec 64 PEs, loi de Cauchy : $m = 2$ et $\gamma = 1$

a Avec une plate-forme de 128 PEs

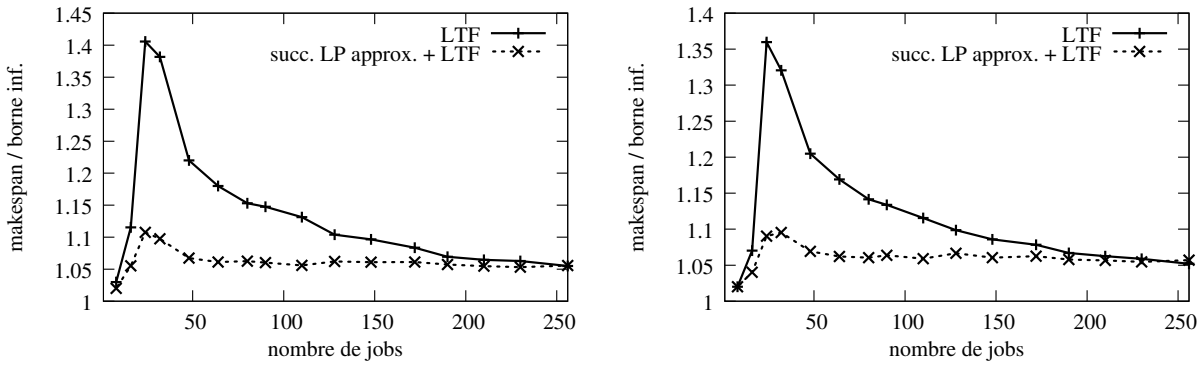
Les paramètres des expériences sont les suivants. La granularité est de 10. $p = 0.1$ et $maxIter = 200$. Pour chaque taille de la collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 40 avec une loi de Cauchy. Le paramètre γ (facteur d'échelle) de la loi est de 8. Le paramètre m (médiane) de la loi est de 20. Les figures 5.18a et 5.18b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. On remarque que dans cette situation, l'algorithme LTF est très performant et dépasse notre algorithme à partir de 80 jobs rigides et à partir de 60 jobs moldables.

Ici, un paramètre de la loi de Cauchy change. Le paramètre γ de la loi de Cauchy est toujours de 8. Le paramètre m de la loi de Cauchy est de 4. Donc il y a davantage de jobs qui requièrent un petit nombre de PEs que de jobs qui demandent beaucoup de PEs. Les figures 5.19a et 5.19b montrent le ratio du makespan à l'optimal en fonction du nombre de jobs rigides et moldables pour cette distribution. On remarque que l'algorithme donne de meilleurs résultats que l'algorithme LTF, néanmoins dès que le nombre de jobs à ordonnancer est important l'algorithme LTF rejoint notre algorithme.



(a) jobs rigides **(b)** jobs moldables

Figure 5.18 – Performances de l'algorithme avec 128 PEs, loi de Cauchy : $m = 20$ et $\gamma = 8$



(a) jobs rigides **(b)** jobs moldables

Figure 5.19 – Performances de l'algorithme avec 128 PEs, loi de Cauchy : $m = 4$ et $\gamma = 8$

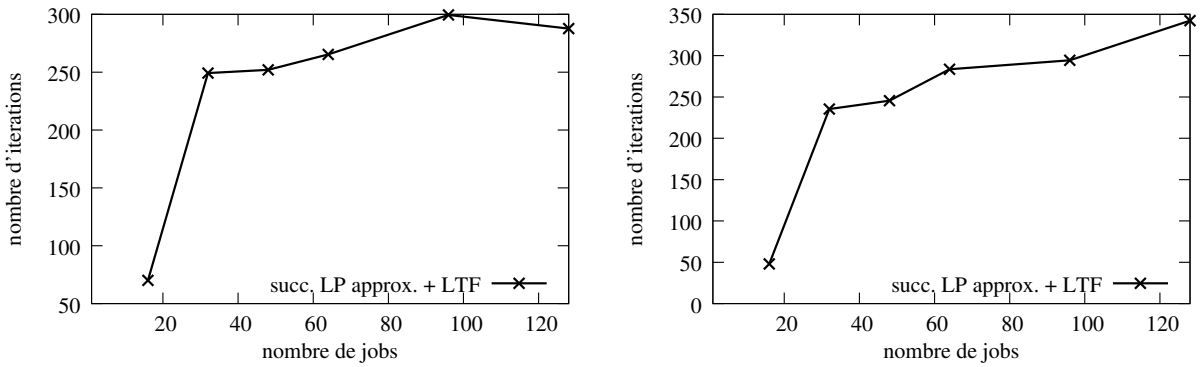
5.6.4 Conclusion sur les résultats expérimentaux

Nous pouvons conclure que les performances de notre algorithme ne dépend pas du type de la distribution que l'on utilise pour générer aléatoirement le nombre de PEs requis pour les jobs des workloads synthétiques. Néanmoins on s'aperçoit que plus ce nombre de PEs est grand, plus notre algorithme a des difficultés a faire mieux ou aussi bien que l'algorithme LTF. Cela se confirme pour la loi normale avec μ (moyenne) grand et pour la loi de Cauchy avec m (médiane) grand.

Quand le nombre de jobs à ordonnancer devient grand, l'algorithme LTF obtient des ordonnancements pour lesquels le ratio du makespan à la borne inférieure diminue. En revanche, pour notre algorithme, à partir d'un nombre donné de jobs à ordonnancer, ce ratio devient quasiment constant, et d'après les courbes, il avoisine environ 1.05.

5.7 Le nombre d'itérations

La figure 5.20b montre le nombre total lpx d'itérations pour trouver l'ordonnancement final, c'est-à-dire le nombre de fois où le programme linéaire est résolu. Si un ordonnancement trouvé par notre méthode a un makespan inférieur à celui de l'ordonnancement obtenu par LTF on a $lpx \leq (listMakespan - lb) * maxIter$. On remarque aussi bien pour les jobs rigides que les jobs moldables que plus le nombre de jobs est important et plus le nombre d'exécutions du programme linéaire est important. On s'aperçoit également que le nombre d'itérations est moins important pour les jobs rigides. Cela peut s'expliquer par le fait que le programme linéaire a moins de configurations à examiner.



(a) jobs rigides **(b)** jobs moldables

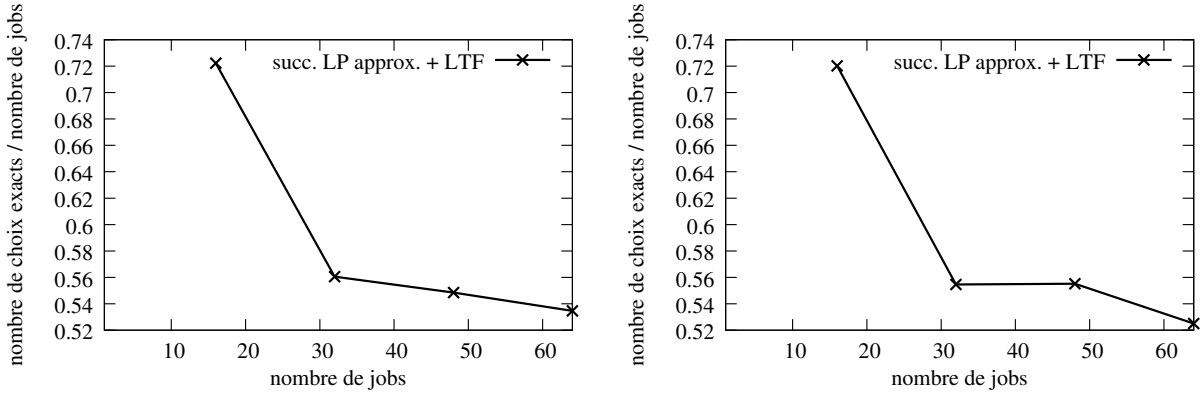
Figure 5.20 – Nombre total d'itérations pour les approximations linéaires successives avec 128 PEs et une loi uniforme

5.8 Le nombre d'ordonnancements exacts trouvés par le programme linéaire

Les figures 5.21a et 5.21b montrent le ratio moyen du nombre d'ordonnancements exacts trouvés à chaque exécution du programme linéaire par rapport au nombre de jobs à ordonnancer, pour une machine de 64 PEs et des workloads générés avec une loi uniforme (paragraphe 5.5.2). Pour ces expériences, en moyenne pour 16 jobs à ordonnancer presque 75% des jobs ont un ordonnancement exact, et les autres 25% ont un ordonnancement flou. Plus on augmente le nombre de jobs à ordonnancer et plus le ratio diminue, car le problème devient plus difficile : sur 64 jobs à ordonnancer seulement la moitié ont un ordonnancement exact. On remarque que le ratio est presque identique, que ce soit pour les jobs rigides ou pour les jobs moldables.

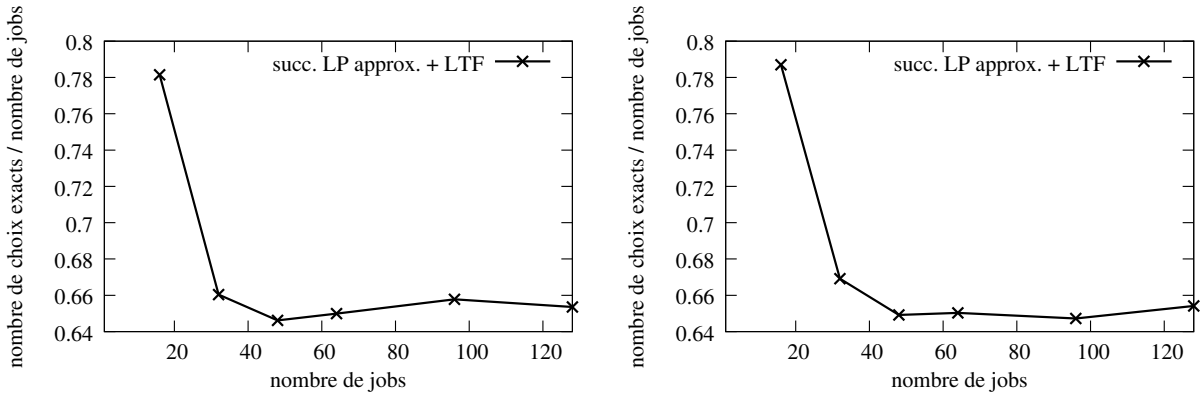
Les figures 5.22a et 5.22b montrent le ratio moyen du nombre d'ordonnancements exacts trouvés à chaque exécution du programme linéaire par rapport au nombre de jobs à ordonnancer, pour une machine de 128 PEs et des workloads générés avec une loi uniforme (paragraphe 5.5.2). Pour ces expériences, en moyenne pour 16 jobs à ordonnancer, presque 80% des jobs ont un ordonnancement exact, et les autres 20% ont un ordonnancement flou. Sur 128 jobs à ordonnancer presque les 2/3 ont un ordonnancement exact. On remarque que dans ces expériences, le ratio est plus élevé avec 128 PEs qu'avec 64 PEs, car dans les expériences avec 128 PEs la

granularité n'est que de 10 alors qu'avec 64 PEs la granularité est de 25, donc ici, le problème est plus simple pour le programme linéaire.



(a) jobs rigides **(b)** jobs moldables

Figure 5.21 – Ratio du nombre d'ordonnancements exacts trouvés par le programme linéaire par rapport au nombre de jobs avec 64 PEs (workloads générés avec une loi uniforme)



(a) jobs rigides **(b)** jobs moldables

Figure 5.22 – Ratio du nombre d'ordonnancements exacts trouvés par le programme linéaire par rapport au nombre de jobs avec 128 PEs (workloads générés avec une loi uniforme)

5.9 Visualisations de quelques ordonnancements obtenus

La figure 5.23 illustre un ordonnancement obtenu à partir de l'algorithme classique LTF (à gauche) et à partir de notre algorithme (à droite) avec 64 PEs et 32 jobs moldables. Du fait de ses priorités l'algorithme LTF ordonnance d'abord les gros jobs, puis s'occupe des petits jobs, alors que notre algorithme réarrange à sa façon de manière plus optimale. Nous pouvons remarquer que les jobs 3 et 28 ont été pliés sur moins de PEs qu'avec LTF. L'algorithme LTF

souffre du fait que les petits jobs mais longs sont ordonnancés à la fin, et donc font augmenter le makespan.

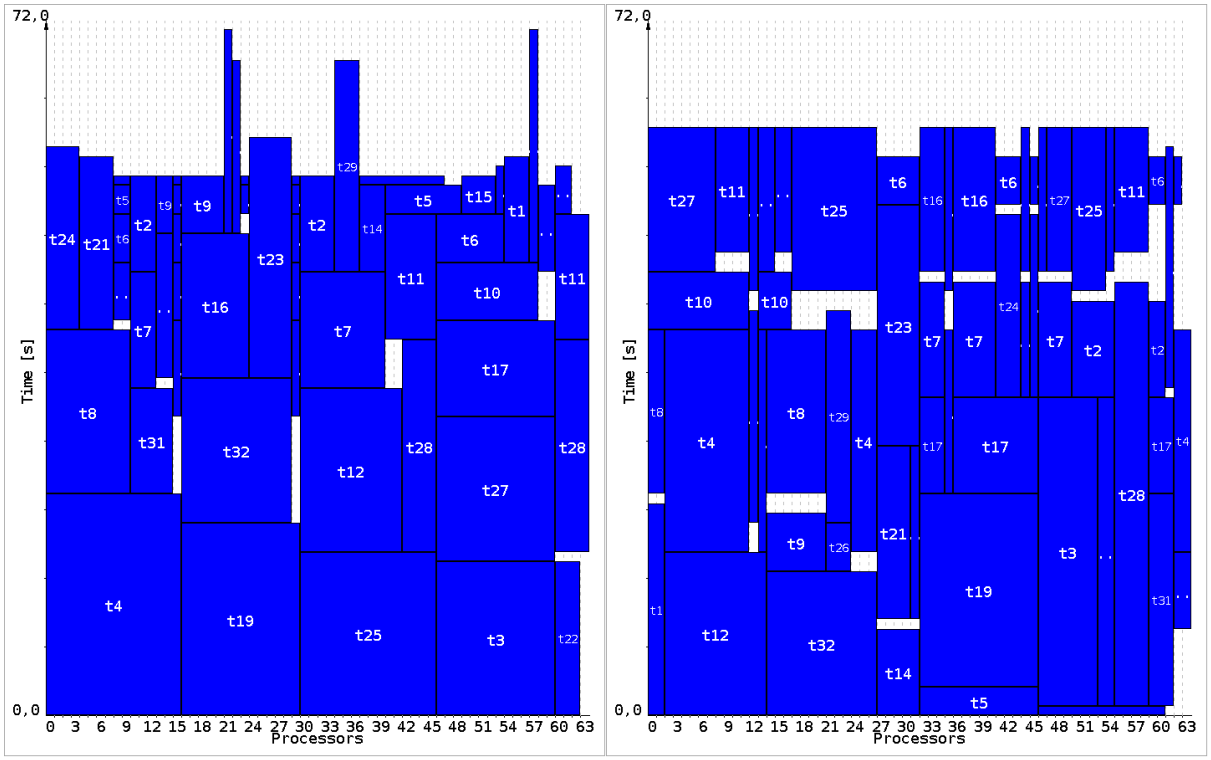


Figure 5.23 – Ordonnancement avec 64 PEs et 32 jobs moldables, LTF à gauche, approximations linéaires successives + LTF à droite avec une loi uniforme

5.10 Version améliorée de l'algorithme avec discrétisation automatique du temps

5.10.1 Motivation

Maintenant, nous considérons n'importe quels jobs avec des temps requis qui ne sont pas nécessairement alignés avec le pas de discrétisation δ (δ n'est pas nécessairement un diviseur de $reqtime_i$) si bien que nous pouvons réaliser des expériences plus réalistes avec des jobs de workloads dont la granularité peut être plus importante, par exemple 100 : le job le plus court peut avoir un temps requis de 1 heure et le job le plus long peut avoir un temps requis de 100 heures.

$lp\_reqtime_i$ est calculé comme suit avec δ comme paramètre des simulations :

$$lp\_reqtime_i = \left\lceil \frac{reqtime_i}{\delta} \right\rceil$$

Le tableau 5.1 donne un exemple d'une collection de 5 jobs dont les durées d'exécution respectives sont données dans la colonne $reqtime_i$ en heures et dans la colonne $lp\_reqtime_i$ en unités de temps. Dans cet exemple le pas de discrétisation de temps δ est de 4 heures. Les

$reqtime_i$ (en heures)	$lp\_reqtime_i$ (en unités de temps)
1	1
25	7
45	12
100	25
55	14

Table 5.2 – Exemple de durées d'exécution de jobs d'une collection à ordonnancer

temps requis en heures ne sont pas alignés avec δ.

Ainsi si on reprend des workloads générés avec une loi uniforme de la section 5.5.2 mais avec une granularité de 100 et $\delta = 4$, on peut se ramener à un problème de taille équivalente à celui des simulations en section 5.5.2 avec une granularité de 25 ($\frac{100}{4} = 25$).

5.10.2 L'algorithme avec discrétisation automatique du temps

Nous avons développé un algorithme qui prend en charge les jobs qui ne sont pas alignés avec le pas de discrétisation δ. Cet algorithme réalise la discrétisation du temps. Il s'agit de l'algorithme 13.

Soit $\mathcal{C}$ la collection de jobs i. Nous copions d'abord la collection $\mathcal{C}$ dans $\mathcal{C}'$. Nous ajustons les temps requis $reqtime_i$ des jobs i de $\mathcal{C}'$ en remplaçant $reqtime_i$ par $adjusted\_reqtime_i$ tels que $adjusted\_reqtime_i$ soit le plus petit multiple de δ immédiatement supérieur ou égal à $reqtime_i$. Maintenant les temps requis des jobs i de la collection $\mathcal{C}'$ sont alignés avec le pas de discrétisation δ. Nous pouvons alors diviser chaque $adjusted\_reqtime_i$ par δ pour réduire la taille du problème. Nous calculons ensuite un ordonnancement *sched* de la collection $\mathcal{C}'$ avec l'algorithme LTF. Le reste de l'algorithme est similaire à l'algorithme amélioré (algorithme 12) vu précédemment. Néanmoins cet algorithme avec discrétisation automatique du temps applique l'algorithme EST (en anglais *Earliest Start Time*) (lignes 29 à 30) pour les jobs avec un ordonnancement exact. Le reste des jobs qui ont un ordonnancement flou sont ordonnancés avec l'algorithme LTF comme dans l'algorithme 12.

Comme le temps requis de chaque job est réajusté au multiple le plus proche de δ et directement supérieur à δ, grâce à l'algorithme EST, les trous dus aux arrondis supérieurs des temps requis des jobs sont supprimés. C'est pourquoi nous utilisons l'algorithme EST.

5.10.3 Simulations et résultats

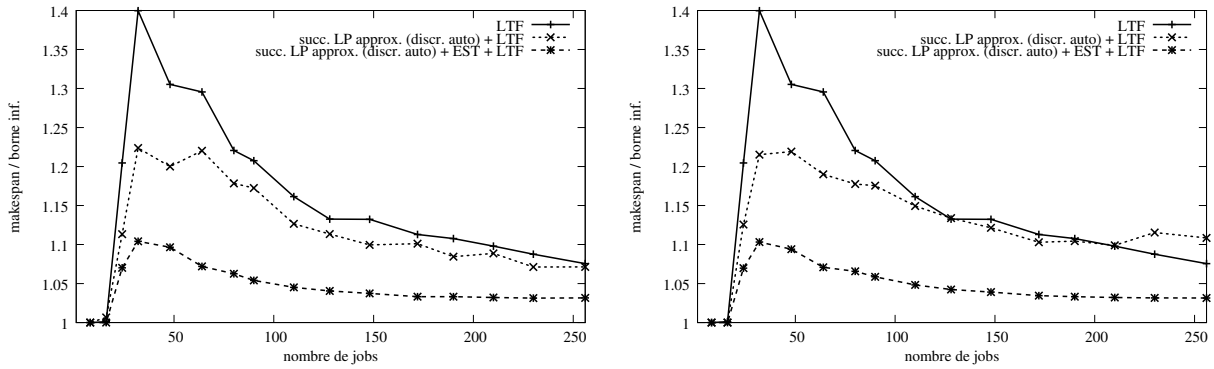
Les expériences sont réalisées avec les paramètres suivants. La granularité est de 100, $\delta = 4$, $p = 0.1$ et $maxIter = 200$. Pour chaque nombre de jobs à ordonnancer nous réalisons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 40 avec une loi normale. Le paramètre σ (écart-type) de la gaussienne est de 8. Le paramètre μ (moyenne) de la gaussienne est de 4.

Les figures 5.24a et 5.24b montrent les performances de l'algorithme avec discrétisation automatique + EST + LTF. Nous remarquons que pour n'importe quelle taille de collection de jobs, notre algorithme se comporte bien et donne de meilleurs résultats que le LTF. Nous

Algorithme 13: L'algorithme avec discrétisation automatique du temps + EST + LTF

$\mathcal{C}' \leftarrow$ copier $\mathcal{C}$
$\forall i \in \mathcal{C}'$ calculer $ajusted\_reqtime_i$ tel que $adjusted\_reqtime_i \geq reqtime_i$ et $adjusted\_reqtime_i$ est le plus petit multiple de δ
$\forall i \in \mathcal{C}'$ $adjusted\_reqtime_i \leftarrow \frac{adjusted\_reqtime_i}{\delta}$
$\forall i \in \mathcal{C}'$ $reqtime_i \leftarrow adjusted\_reqtime_i$
$lb \leftarrow$ borne inférieur du makespan avec la collection $\mathcal{C}$
$sched \leftarrow$ calculer un ordonnancement de la collection $\mathcal{C}'$ avec LTF
$listMakepsan \leftarrow makepsan(sched)$
$T \leftarrow listMakespan$
$end \leftarrow faux$
$incT \leftarrow faux$
tant que $T \times \delta > lb$ *et non* end **faire**
    $proc \leftarrow$ calculer les configurations des jobs de $(\mathcal{C}')$
    $run \leftarrow$ calculer tous les slots possibles$(\mathcal{C}', m, T)$
    $iter \leftarrow 1$
    $found \leftarrow faux$
    $\forall i, k,\ x_{i,k}^{(iter)} \leftarrow 0$
    tant que $iter < maxIter$ *et non* $found$ **faire**
        initialiser l'objectif de la fonction objectif de $\mathcal{LP}(\mathcal{C}', m, T, proc, run)$ à $\sum_i^{|\mathcal{C}'|} f(x_i^{(iter)}) + \langle \nabla f(x_i^{(iter)}), x_i - x_i^{(iter)} \rangle$
        $x \leftarrow$ exécuter $\mathcal{LP}(\mathcal{C}', m, T, proc, run)$
        si $\forall i,\ x_i$ *contient exactement un "1"* **alors**
            $bestSched \leftarrow$ convertir en un ordonnancement $(x, proc, run)$
            $T \leftarrow makespan(sched)$
            $T \leftarrow T - 1$
            $found \leftarrow vrai$
            if $incT = vrai$ **then**
                $end \leftarrow vrai$
        sinon
            garder dans $\mathcal{E}$ les jobs $i \in \mathcal{C}$ tels que x_i contient exactement un "1"
            trier les jobs i de $\mathcal{E}$ dans l'ordre croissant des dates de de début $starttime_i$
            $sched \leftarrow$ ordonnancer avec un algorithme classique de liste les jobs triés $i \in \mathcal{E}$
            $sched \leftarrow sched \cup$ ordonnancer avec l'algorithme LTF le reste des jobs $j \in \mathcal{C} - \mathcal{E}$
            $listMakepsan \leftarrow makespan(sched)$
            si $listMakespan \leq T \times \delta$ **alors**
                $bestSched \leftarrow sched$
                $T \leftarrow min\left(T - 1, round\left(\frac{listMakespan}{\delta}\right) - 1\right)$
                $found \leftarrow vrai$
        $\forall i, k,\ x_{i,k}^{(iter)} \leftarrow x_{i,k}$; $iter \leftarrow iter + 1$
    si $non\ found$ **alors**
        si $T = listMakespan$ **alors**
            $incT \leftarrow vrai$
        si $incT = vrai$ **alors**
            $T \leftarrow T + 1$
        sinon
            $end \leftarrow vrai$
retourner $bestSched$

avons ajouté une courbe qui montre l'intérêt d'utiliser EST dans le but de supprimer les trous. En effet cette courbe représente les performances du même algorithme sans l'algorithme EST : pour les jobs qui ont un ordonnancement exact, on récupère leur date de début et on les place en sachant que l'on a surévalué leur hauteur. Cela provoque des trous et un ordonnancement de qualité moyenne.



(a) jobs rigides **(b)** jobs moldables

Figure 5.24 – Performances des algorithmes avec 128 PEs, loi normale : $\mu = 4$, $\sigma = 8$, $granularite = 100$ et $\delta = 4$

5.11 Algorithmes fondés sur les approximations linéaires successives et tassement des jobs par l'algorithme EST

5.11.1 Algorithmes

Précédemment les deux algorithmes (avec et sans discrétisation automatique du temps) qui utilisent l'algorithme LTF pour compléter les ordonnancements exacts trouvés par le programme linéaire, donnent de bons résultats.

Typiquement dans les algorithmes utilisant un programme linéaire en nombres entiers relaxé, les valeurs maximales rationnelles sont conservées et sont arrondies aux valeurs entières les plus proches. L'algorithme effectue un post-traitement afin que les valeurs arrondies respectent les contraintes.

Pour les expériences décrites ici nous appliquons cette méthode. Nous observons ce qui se passe si nous remplaçons le LTF, par un simple algorithme EST de post-traitement : après chaque exécution du programme linéaire, pour chaque job i, on garde le slot s du job i tel que $x_{i,s} = \max_j(x_{i,j})$. Ce slot s indique une allocation et une position pour chaque job i. On range tous les jobs dans l'ordre croissant des dates de début et on applique sur ces jobs un simple algorithme de Liste (algorithme EST). On obtient alors un ordonnancement avec un makespan $innerMakespan$. Si $innerMakespan$ est inférieur ou égal à l'horizon T, on garde cet ordonnancement. Nous avons implémenté respectivement les algorithmes 15 et 14 respectivement avec et sans discrétisation automatique.

Algorithme 14: L'algorithme d'approximation par linéarisations successives + EST

```
1  lb ← borne inférieure du makespan
2  sched ← calculer un ordonnancement avec l'agorithme LTF
3  listMakespan ← makespan(sched)
4  T ← listMakespan
5  end ← faux
6  incT ← faux
7  tant que T > lb et non end faire
8      proc ← calculer les configurations des jobs J
9      run ← calculer tous les slots possibles avec (J, m, T)
10     iter ← 1
11     found ← faux
12     ∀i, k, x_{i,k}^{(iter)} ← 0
13     tant que iter < maxIter et non found faire
14         mettre la fonction objectif de LP(J, m, T, proc, run) à
           Σ_i^{|J|} f(x_i^{(iter)}) + ⟨∇f(x_i^{(iter)}), x_i − x_i^{(iter)}⟩
15         x ← exécuter LP(J, m, T, proc, run)
16         si ∀i, x_i contient exactement un "1" alors
17             bestSched ← convertir en un ordonnancement (x, proc, run)
18             T ← makespan(bestSched)
19             T ← T − 1
20             found ← vrai
21             if incT = vrai then
22                 end ← vrai
23         sinon
24             garder les slots s des jobs i ∈ C tels que x_{i,s} = max_j(x_{i,j})
25             ∀i, starttime_i ← convertir (run, x_{i,s}) en date de début
26             trier les jobs i de C dans l'ordre croissant des dates de de début starttime_i
27             sched ← ordonnancer avec un algorithme classique de liste les jobs triés i ∈ C
28             innerMakespan ← makespan(sched)
29             si innerMakespan < T alors
30                 bestSched ← sched
31                 T ← T − 1
32                 found ← vrai
33                 si incT = vrai alors
34                     end ← vrai
35         ∀i, k, x_{i,k}^{(iter)} ← x_{i,k}
36         iter ← iter + 1
37     si non found alors
38         si T = listMakespan alors
39             incT ← vrai
40         si incT = vrai alors
41             T ← T + 1
42         sinon
43             end ← vrai
44 retourner bestSched
```

Algorithme 15: L'algorithme avec discrétisation automatique du temps + EST

$\mathcal{C}' \leftarrow$ copier $\mathcal{C}$
$\forall i \in \mathcal{C}'$ calculer $ajusted\_reqtime_i$ tel que $adjusted\_reqtime_i \geq reqtime_i$ et $adjusted\_reqtime_i$ est le plus petit multiple de δ
$\forall i \in \mathcal{C}'$ $adjusted\_reqtime_i \leftarrow \frac{adjusted\_reqtime_i}{\delta}$
$\forall i \in \mathcal{C}'$ $reqtime_i \leftarrow adjusted\_reqtime_i$
$lb \leftarrow$ borne inférieur du makespan avec la collection $\mathcal{C}$
$sched \leftarrow$ calculer un ordonnancement de la collection $\mathcal{C}'$ avec LTF
$listMakepsan \leftarrow makepsan(sched)$
$T \leftarrow listMakespan$
$end \leftarrow faux$
$incT \leftarrow faux$
tant que $T \times \delta > lb$ *et non* end **faire**
    $proc \leftarrow$ calculer les configurations des jobs de $(\mathcal{C}')$
    $run \leftarrow$ calculer tous les slots possibles$(\mathcal{C}', m, T)$
    $iter \leftarrow 1$
    $found \leftarrow faux$
    $\forall i, k,\ x_{i,k}^{(iter)} \leftarrow 0$
    tant que $iter < maxIter$ *et non* $found$ **faire**
        initialiser l'objectif de la fonction objectif de $\mathcal{LP}(\mathcal{C}', m, T, proc, run)$ à $\sum_i^{|\mathcal{C}'|} f(x_i^{(iter)}) + \langle \nabla f(x_i^{(iter)}), x_i - x_i^{(iter)} \rangle$
        $x \leftarrow$ exécuter $\mathcal{LP}(\mathcal{C}', m, T, proc, run)$
        si $\forall i,\ x_i$ *contient exactement un "1"* **alors**
            $bestSched \leftarrow$ convertir en un ordonnancement $(x, proc, run)$
            $T \leftarrow makespan(sched)$
            $T \leftarrow T - 1$
            $found \leftarrow vrai$
            if $incT = vrai$ **then**
                $end \leftarrow vrai$
        sinon
            garder les slots s des jobs $i \in \mathcal{C}$ tels que $x_{i,s} = \max_j(x_{i,j})$
            $\forall i,\ starttime_i \leftarrow$ convertir $(run, x_{i,s})$ en date de début
            trier les jobs i de C dans l'ordre croissant des dates de de début $starttime_i$
            $sched \leftarrow$ ordonnancer avec un algorithme classique de liste les jobs triés $i \in \mathcal{C}$
            $listMakepsan \leftarrow makespan(sched)$
            si $listMakespan \leq T \times \delta$ **alors**
                $bestSched \leftarrow sched$
                $T \leftarrow min\left(T - 1, round\left(\frac{listMakespan}{\delta}\right) - 1\right)$
                $found \leftarrow vrai$
        $\forall i, k,\ x_{i,k}^{(iter)} \leftarrow x_{i,k}$; $iter \leftarrow iter + 1$
    si $non\ found$ **alors**
        si $T = listMakespan$ **alors**
            $incT \leftarrow vrai$
        si $incT = vrai$ **alors**
            $T \leftarrow T + 1$
        sinon
            $end \leftarrow vrai$
retourner $bestSched$

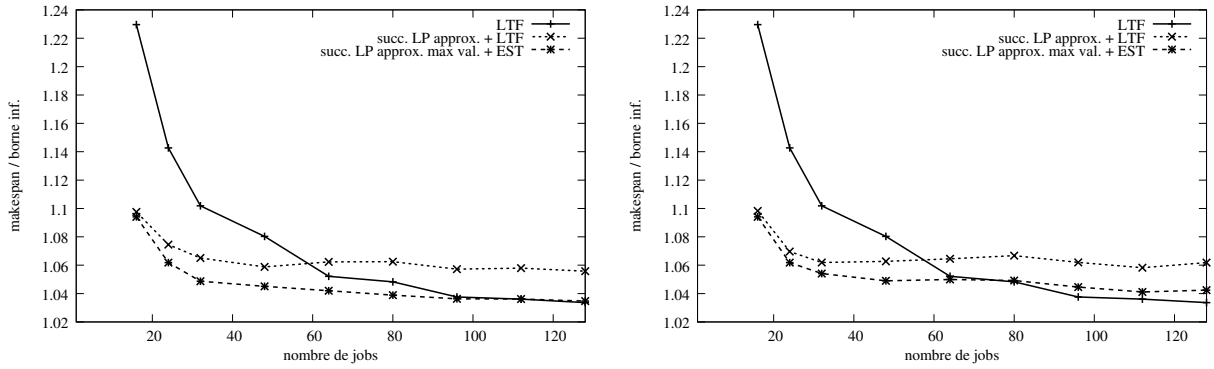
5.11.2 Simulations et résultats (sans discrétisation automatique du temps)

Dans ce paragraphe nous reprenons les workloads des expériences réalisées à la section 5.6 avec les différentes lois utilisées pour la génération des workloads mais nous réalisons les expériences avec l'algorithme 14 fondé sur les approximations linéaires successives et tassement des jobs par l'algorithme EST. Nous nous plaçons dans un contexte où les temps requis des jobs sont alignés avec le pas de discrétisation δ.

a Loi normale

Les paramètres des expériences sont les suivants. La granularité est de 10. $p = 0.1$ et $maxIter = 200$. Pour chaque taille de la collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 20 avec une loi gaussienne. Le paramètre σ (écart-type) de la gaussienne est de 8. Le paramètre μ (espérance) de la gaussienne est de 20.

Les figures 5.25a et 5.25b montrent les performances de l'algorithme 14 avec tassement des jobs par EST, les performances de l'algorithme 12 qui utilise LTF et celles de l'algorithme LTF seul avec 128 PEs. D'abord nous remarquons que le tassement des jobs par l'algorithme EST est efficace car cette technique améliore les résultats par rapport à l'algorithme utilisant LTF. Néanmoins, pour des tailles de collection de jobs moldables plus grandes, l'algorithme LTF est à peine meilleur que l'algorithme avec EST. Par ailleurs, pour les jobs rigides l'algorithme 14 trouve des ordonnancements meilleurs que le LTF, pour n'importe quelle taille de collections.



(a) jobs rigides **(b)** jobs moldables

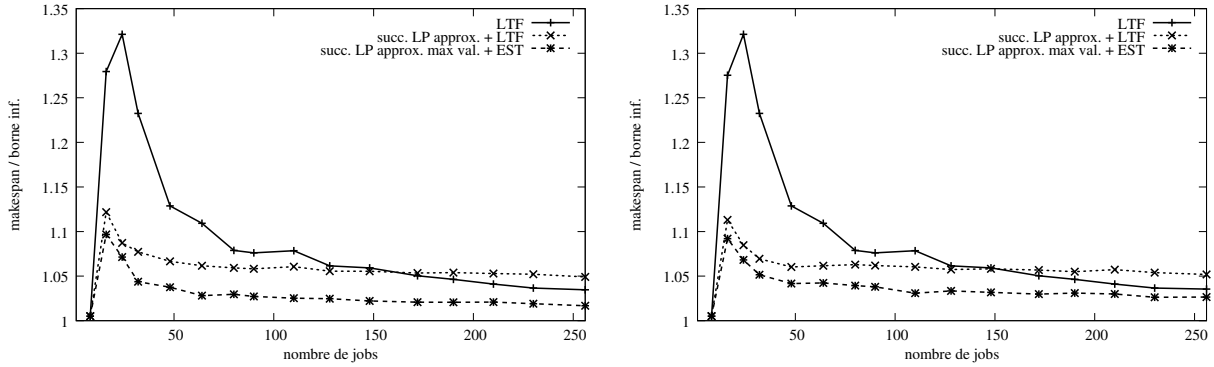
Figure 5.25 – Performances des algorithmes avec 128 PEs, loi normale : $\mu = 20$ et $\sigma = 8$

b Mélange de deux Lois normales

Les paramètres des expériences sont les suivants. La granularité est de 10, $p = 0.1$ et $maxIter = 200$. Pour chaque taille de la collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 20 avec un mélange de deux lois normales. Le paramètre σ_1 de la gaussienne 1 est de 8. Le paramètre μ_1 de la gaussienne 1 est de 4. Le paramètre σ_2 de la gaussienne 2 est de 8. Le paramètre μ_2 de la gaussienne 2 est de 20.

Les figures 5.26a et 5.26b montrent les performances de l'algorithme 14 avec tassement des

jobs par EST, les performances de l'algorithme 12 avec le LTF et celles de l'algorithme LTF seul avec 128 PEs. Nous remarquons que pour les jobs rigides et moldables, notre algorithme 14 trouve des ordonnancements meilleurs que l'algorithme LTF.



(a) jobs rigides **(b)** jobs moldables

Figure 5.26 – Performances des algorithmes avec 128 PEs, mélange de deux lois normales : $\pi_1 = \pi_2 = 0.5$, $\mu_1 = 4$, $\mu_2 = 20$ et $\sigma_1 = \sigma_2 = 8$

c Loi de Cauchy

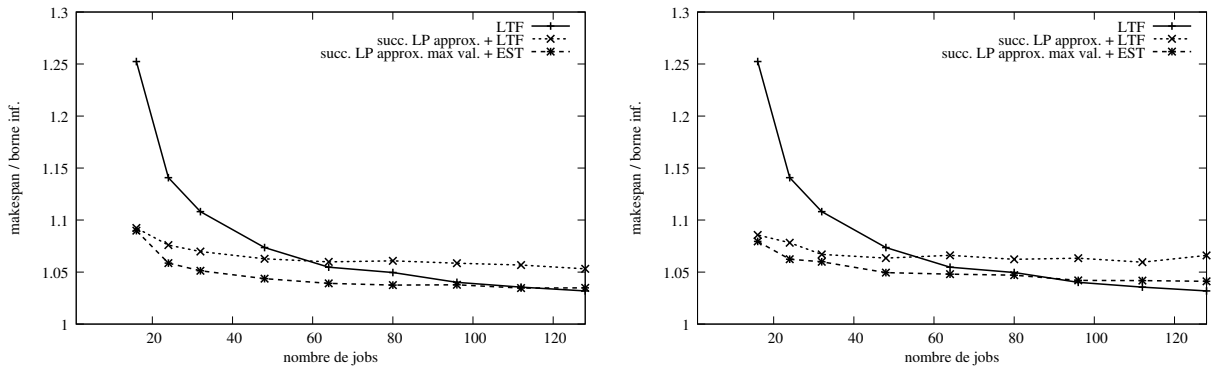
Les paramètres des expériences sont les suivants. La granularité est de 10, $p = 0.1$ et $maxIter = 200$. Pour chaque taille de la collection de jobs nous lançons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 20 avec une loi gaussienne. Le paramètre m (médiane) de la loi de Cauchy est de 20. Le paramètre γ (facteur d'échelle) de la gaussienne est de 8.

Les figures 5.27a et 5.27b montrent les performances de l'algorithme 14 avec tassement des jobs par EST, les performances de l'algorithme 12 avec le LTF et celles de l'algorithme LTF seul avec 128 PEs. Nous remarquons que le tassement des jobs est toujours efficace dans ces expériences avec une loi de Cauchy. Pour des grandes collections de jobs, on remarque un même comportement qu'avec la loi gaussienne de la section 5.11.2.a : pour des tailles de collection de plus de 100 jobs moldables, l'algorithme LTF est légèrement plus performant que l'algorithme 14 avec EST. Cependant pour n'importe quelle taille de collection de jobs rigides, notre algorithme 14 est toujours meilleur que l'algorithme LTF.

5.11.3 Simulations et résultats (avec discrétisation automatique du temps)

Nous nous plaçons dans un contexte où les temps requis des jobs ne sont pas alignés avec le pas de discrétisation δ afin de montrer l'intérêt de la méthode du tassement des jobs par l'algorithme EST.

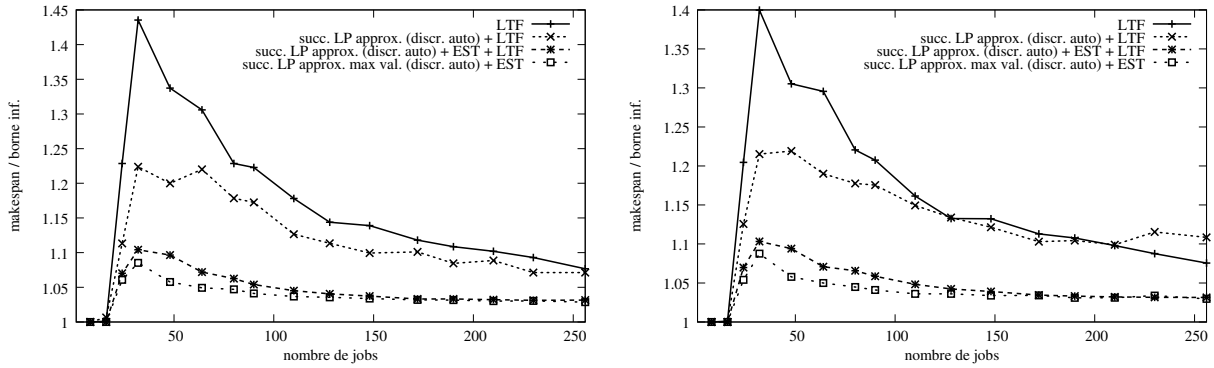
Les expériences sont réalisées avec les paramètres suivants. La granularité est de 100. $\delta = 4$, $p = 0.1$ et $maxIter = 200$. Pour chaque nombre de jobs à ordonnancer nous réalisons 20 expériences. Le nombre de PEs requis pour chaque job est choisi entre 1 et 40 avec une loi normale. Le paramètre σ (écart-type) de la gaussienne est de 8. Le paramètre μ (moyenne) de la gaussienne est de 4.



(a) jobs rigides **(b)** jobs moldables

Figure 5.27 – Performances des algorithmes avec 128 PEs, loi de Cauchy : $m = 20$ et $\gamma = 8$

Les figures 5.28b et 5.28b montrent les performances de l'algorithme avec discrétisation automatique + EST. Nous remarquons, que pour n'importe quel taille de collection de jobs, notre algorithme se comporte bien et donne les meilleurs résultats aussi bien pour les jobs rigides que les jobs moldables.

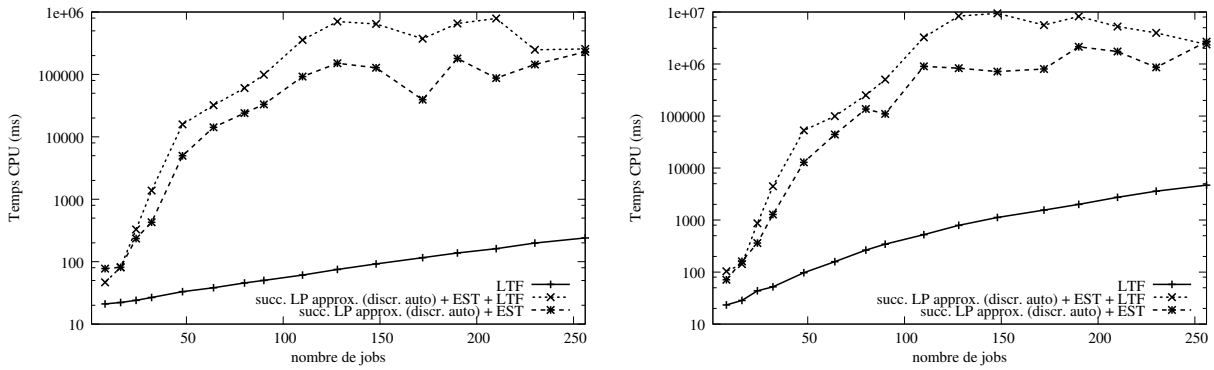


(a) jobs rigides **(b)** jobs moldables

Figure 5.28 – Performances des algorithmes avec 128 PEs, loi normale : $\mu = 4$, $\sigma = 8$, $granularite = 100$ et $\delta = 4$

a Temps de calcul

Les figures 5.29a et 5.29b montrent le temps nécessaire pour que les algorithmes trouvent un ordonnancement. Nous remarquons que l'algorithme avec tassement des jobs par EST est moins coûteux que l'algorithme avec LTF. Néanmoins, ils sont tous les deux beaucoup plus gourmands en temps de calcul que l'algorithme LTF. Il faut environ trois heures à nos algorithmes fondée sur les approximations linéaires successives pour trouver un ordonnancement d'une collection de 250 jobs moldables alors qu'il en faut 1 seconde à l'algorithme LTF. Par ailleurs, ordonnancer une cinquantaine de jobs avec notre méthode prend moins de 20 secondes et pour cette taille de collection, l'algorithme LTF offre des performances loin derrière nos algorithmes.



(a) jobs rigides **(b)** jobs moldables

Figure 5.29 – Temps de calcul avec 128 PEs et une loi normale : $\mu = 4$, $\sigma = 8$, $granularite = 100$ et $\delta = 4$

5.12 Synthèse

Dans ces travaux, les résultats expérimentaux montrent que l'algorithme qui utilise directement les approximations linéaires successives sans post-traitement donne de bons résultats en terme de makespan pour l'ordonnancement de quelques dizaines de jobs sur une machine de 32 PEs. En revanche, la première variante (algorithme 12) qui consiste à garder les jobs ordonnancés de manière exacte par le programme linéaire et à ordonnancer le reste des jobs *flous* avec le LTF donne de bons résultats. Nous proposons également une variante avec discrétisation automatique du temps quand les temps d'exécution des jobs ne s'alignent pas sur le pas de discrétisation du temps ou quand la granularité, c'est-à-dire, le rapport entre la durée du job le plus court et la durée du job le plus long est trop grande. Cette variante (algorithme 13) donne d'aussi bons résultats que la première variante (algorithme 12). Une autre variante est proposée (algorithme 14) : il s'agit de prendre pour chaque job le slot qui a une valeur maximale, d'enregistrer leur date de début et de les ordonnancer dans l'ordre de leur date de début au moyen d'un simple algorithme de liste. Cette variante est le meilleur algorithme. Nous proposons également une variante (algorithme 15) avec discrétisation automatique du temps qui donne d'aussi bons résultats.

Nous avons essayé plusieurs types de lois de probabilité sur le nombre de PEs requis pour chaque job mais ils n'ont pas d'incidence sur les performances de nos algorithmes. Cependant on constate que plus les jobs demandent de PEs et plus notre algorithme se fait rattrapé par le simple algorithme LTF quand le nombre de jobs augmente.

Nous constatons que les temps de calcul des ordonnancements par nos algorithmes sont beaucoup plus importants que ceux de l'algorithme LTF. Néanmoins, dans des travaux futurs, nous pourrions résoudre les programmes linéaires à l'aide d'algorithmes optimisés pour nos matrices creuses qui ont des structures particulières. Nous pourrions essayer de faire plusieurs petits lots de jobs et les ordonnancer en étages et de tasser l'ensemble par l'algorithme EST. De plus, comme les temps CPU pour ordonnancer un lot d'une cinquantaine de jobs sur une machine de 128 PEs est de l'ordre d'une dizaine de secondes, nous pourrions réaliser des expériences en mode online à l'aide de notre simulateur de cluster homogène et de son ordonnanceur de jobs. Ce dernier travaillerait avec des fenêtres de 50 jobs et ferait appel à notre algorithme fondé

sur les approximations linéaires successives et tassement des jobs par EST avec discrétisation automatique du temps (algorithme 15).

Il convient de noter que les possibilités d'optimisation d'une collection de jobs indépendants sont limitées quand le nombre de jobs devient très grand car, comme le montre l'algorithme LTF, un simple algorithme comme lui peut donner des ordonnancements proches de l'optimal. Les courbes qui figurent dans ce chapitre montrent que l'algorithme LTF donne des ordonnancements dont le makespan est à inférieur à 1.1 fois la borne inférieure du makespan. Cela signifie qu'il est déjà proche loin du makespan optimale. Ainsi, la marge de manœuvre qui nous reste est très faible.

5.13 Autres recherches

La recherche de méthodes mathématiques pour optimiser notre programme linéaire nous a conduit à tester d'autres outils. D'autres recherches sont présentées en annexe soit parce qu'elles n'ont pas abouti à un résultat significatif en terme de qualité de l'ordonnancement, soit parce que les méthodes utilisées ne rendent pas de résultats.

L'annexe A traite de la méthode *Split Bregman* [52] que nous avons essayée d'appliquer sur le problème d'ordonnancement d'une collection de jobs indépendants dans une machine multiprocesseur. Cette méthode sert en particulier à reconstruire un signal entier à partir d'une petite quantité d'informations. L'intérêt de cette méthode est qu'elle ne fait pas intervenir des programmes linéaires, mais seulement des produits de matrices et une résolution d'un système d'équations linéaires. Le principe de cette méthode repose sur des itérations (itérations de Bregman) successives qui visent à minimiser $J(x)$ sous la contrainte d'égalité de la forme $Ax = b$ où J est une fonction convexe sur $\mathbb{R}^n$ et $x \in \mathbb{R}^n$. Nous avons adapté cette méthode pour qu'elle minimise la norme ℓ_p d'un vecteur inconnu x. Nous avons essayé cette méthode car nous souhaitions améliorer les temps de calcul. Mais cette méthode n'a pas encore donné de résultats concluants.

L'annexe B traite d'une méthode utilisant un algorithme itératif [26], appelé *Iteratively Reweighted algorithm*. Nous avons essayé de mettre en œuvre cette méthode pour résoudre le problème d'ordonnancement d'une collection de jobs indépendants dans une machine multiprocesseur. L'idée générale de la méthode repose sur la minimisation de la norme ℓ_p d'un vecteur x élevée à la puissance p sujet à des contraintes d'inégalité. Ainsi nous souhaitions minimiser $\|x\|_p^p$. En résumé, dans cette méthode l'algorithme transforme l'expression $\|x\|_p^p$ afin d'être intégrée dans un programme quadratique. La transformation est fondée sur la formule suivante : $x^p = x^{p-2} \times x^2$. Ainsi, à chaque itération de l'algorithme nous exécutons un programme quadratique dont la fonction objectif est de la forme $x^{(iter)p-2} \times x^2$ où $x^{(iter)}$ représente la solution de l'itération précédente. Ici, nous modélisons l'occupation de chaque job sur les cœurs d'une machine multiprocesseur par des fonctions qui indiquent le nombre de cœurs utilisés en fonction du temps. Nous discrétisons le temps et à chaque instant t, nous attribuons à chaque job i le nombre de cœurs $x_{i,t}$ utilisé par le job i. Comme chaque job i utilise un nombre constant de cœurs durant son exécution, le vecteur δx_i formé de la différence $x_{i,t} - x_{i,t-1}$ doit être très creux : ce vecteur n'est composé que de 0 sauf à l'index correspondant au début du job et à l'index correspondant à la fin du job. Mais cette méthode n'a pas fonctionné non plus.

L'annexe C aborde un outil très célèbre du domaine des statistiques, l'Algorithme *Espérance-Maximisation* (EM). Il permet à partir de données statistiques relevées sur un phénomène, d'estimer avec la plus grande espérance, les paramètres d'une loi qui régit ce phénomène. Nous souhaitions utiliser cet outil statistique pour résoudre le problème d'ordonnancement d'une collection de jobs indépendants dans une machine multiprocesseur. Nous approximons le nombre de PEs utilisés par un job en fonction du temps, qui est plutôt une fonction indicatrice rectangulaire, par une fonction gaussienne. Donc chaque job est une gaussienne. Avec l'algorithme EM nous espérions que les mélanges des gaussiennes, donc des jobs, rentrent tous dans un rectangle dont les dimensions sont limitées horizontalement par un horizon de temps et verticalement par le nombre de PEs de la machine. Les résultats obtenus sont moins bons que ceux obtenus par les algorithmes fondés sur les linéarisations successives d'un modèle creux et un peu meilleur que l'algorithme LTF dans certains cas.

L'annexe D traite du problème d'ordonnancement online présenté dans la section 4, mais va plus loin, puisqu'il est question de résolution de programmes linéaires et de chaînes de Markov. L'annexe D présente les recherches pour optimiser les ordonnancements de jobs pour les clusters.

Conclusion

Les contributions apportées par nos différents travaux au monde de l'ordonnancement ont été motivées par le souhait de trouver des techniques innovantes et efficaces pour les environnements parallèles et distribués. Dans un premier temps nous nous sommes intéressés à dégager ce qui se fait déjà au niveau de l'ordonnancement de jobs pour les clusters, puis au niveau de l'ordonnancement de jobs complexes constituées de tâches munies de contraintes de précédence, pour les grilles de calcul. Dans un deuxième temps, nous présentons quelques outils généraux tels que les algorithmes génétiques et les programmes linéaires. Ces outils sont utilisés à travers nos différents travaux.

Pour les grilles de calcul, deux travaux de recherches ont été menés. Le premier travail était d'évaluer trois algorithmes d'ordonnancement d'une collection workflows identiques sur une plate-forme hétérogène : l'algorithme de liste EFT [88], l'algorithme génétique GATS [28] et l'algorithme steady-state [8] sans tenir compte des coûts de communication. Nous avons extrait les avantages et les limites de chacun d'eux et déterminé dans quels cas utiliser un algorithme plutôt qu'un autre. D'après les expériences menées par Sékou Diakité et nos analyses, nous recommandons d'utiliser l'algorithme génétique quand la taille du lot est inférieur à mille jobs et pour les couples de plates-formes/applications pas trop complexes, et d'utiliser l'algorithme du steady-state, quand la taille du lot dépasse quelques milliers de jobs. Ensuite, entre mille et quelques milliers de jobs, le choix dépend de l'hétérogénéité et de la complexité de la plate-forme. Dans le contexte des grilles et dans la continuité des travaux de Sékou Diakité, nous proposons un algorithme génétique pour l'ordonnancement d'un workflow sur une plate-forme hétérogène où les coûts de communication ne sont pas négligeables. Les résultats sont corrects, car meilleurs que ceux obtenus par l'algorithme de Liste. Cependant, les résultats ne sont pas exceptionnels car ordonnancer un workflow sur une plate-forme hétérogène où il faut tenir compte des coûts de communication reste un problème très difficile.

Pour les clusters homogènes, pour résoudre un problème d'ordonnancement de jobs online, nous proposons une nouvelle technique appelée pliage de jobs qui repose sur les techniques de virtualisation des cœurs pour optimiser l'utilisation du cluster. Le pliage de jobs rend un job parallèle qui est conçu pour tourner sur un certain nombre de cœurs, plus flexible, c'est à dire, qu'il peut tourner sur moins de cœurs. Bien que sa durée sera allongée et parce qu'il commence son exécution plus tôt il peut terminer plus tôt. Nous proposons des heuristiques qui utilisent cette technique. D'après nos expériences, les gains par rapport à un algorithme *FCFS* sont significatifs.

Par la suite nous nous sommes intéressés au cas off-line pour ordonnancer des jobs parallèles sur une machine multiprocesseur. Le critère à optimiser est le makespan. Nous considérons à

la fois des jobs rigides et des jobs moldables. Nous utilisons une nouvelle technique qui intègre des méthodes mathématiques telles que la norme ℓ_p, la linéarisation d'une fonction objectif non linéaire par un développement de Taylor et le gradient. Des expériences ont été réalisées en générant des workloads synthétiques à l'aide de plusieurs lois de probabilités. Les résultats ont montré que les performances de notre approche ne dépendent pas du type de loi de probabilité utilisé pour synthétiser le nombre de PEs requis par les jobs. En fait, les performances de nos heuristiques fondées sur des linéarisations successives d'une fonction objectif non linéaire dépendent du nombre de PEs requis par les jobs à ordonnancer : plus ce nombre est important et plus nos heuristiques ont du mal à battre l'algorithme LTF.

Ces derniers travaux de recherche ont donné globalement de très bons résultats et sont des plus prometteurs. Ces travaux sont le fruit d'une collaboration avec le laboratoire de Mathématiques de Besançon. D'autres voies ont été creusées afin d'optimiser le makespan, comme par exemple avec l'algorithme *EM* ou encore avec la méthode du *Split Bregman*. Tous ces outils ont un point commun, ils exploitent le caractère creux des problèmes étudiés.

Il y a quand même un bémol. Un travail important a été mené sur les programmes linéaires et les chaînes de Markov pour optimiser globalement l'utilisation d'un cluster homogène dans le contexte du pliage de jobs online. Ce travail n'a pas porté ses fruits. En dépit de leur complexité, les heuristiques mises en œuvre n'ont pas réussi à battre un simple algorithme online qui commence à exécuter le job qui a le plus petit deadline (*Earliest Deadline First*). Les résultats de ces travaux sont donc décevants.

Pour les travaux futurs, il restent à approfondir les recherches qui exploitent davantage encore le caractère creux des problèmes que nous posons , à améliorer la représentation des matrices creuses pour la résolution des programmes linéaires et ainsi réduire grandement les temps de calcul. De plus cela faciliterait le passage à l'échelle pour la prise en charge de grosses machines munies de milliers de cœurs et une plus grande granularité. Nous pourrions tenter de faire plusieurs petits lots de jobs et les ordonnancer en étages et de tasser l'ensemble par l'algorithme EST. De plus, comme les temps CPU pour ordonnancer un lot d'une cinquantaine de jobs sur une machine de 128 PEs est de l'ordre de la dizaine de secondes, pour des futurs travaux nous pourrions intégrer la méthode fondée sur les approximations linéaires successives dans notre simulateur de cluster homogène et de son ordonnanceur de jobs. Ce dernier travaillerait avec des fenêtres de 50 jobs et ferait appel à notre algorithme avec tassement des jobs par EST avec discrétisation automatique du temps. D'un point de vue théorique, il faudrait que nous démontrions mathématiquement que la méthode fonctionne quand le nombre de jobs tend vers l'infini. Pour l'algorithme *EM*, il faudrait remplacer les gaussiennes par des fonctions plus rectangulaires afin d'améliorer ses résultats.

Publications

- Stéphane Chrétien, Jean-Marc Nicod, Laurent Philippe, Veronika Rehn-Sonigo and Lamiel Toch. **Job Scheduling Using successive Linear Programming Approximations of a Sparse Model**. In *Euro-Par'12, 18th International European Conference on Parallel and Distributed Computing*, Rhodes Island, Greece, August 2012,

- Sékou Diakité, Jean-Marc Nicod, Laurent Philippe, and Lamiel Toch. **Assessing new approaches to schedule a batch of identical intree-shaped workflows on a heterogeneous platform**, *International Journal of Parallel, Emergent and Distributed Systems*, 27(1) :79–107, 2011,

- Laurent Philippe, Jean-Marc Nicod, and Lamiel Toch. **A Genetic Algorithm with Communication Costs to Schedule Workflows on a SOA-Grid**, In *HeteroPar'2011, 9-th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms, volume 7155 of LNCS*, Bordeaux, France, pages 419–428, 2011,

- Jean-Marc Nicod, Laurent Philippe, Veronika Rehn-Sonigo and Lamiel Toch. **Using Virtualization and Job Folding for Batch Scheduling**. In *ISPDC'2011, 10th Int. Symposium on Parallel and Distributed Computing*, Cluj-Napoca, Romania, pages 33–40, July 2011. IEEE Computer Society Press.

Annexes

Annexe A

Méthode *Split Bregman*

Nous avons appliqué la méthode *Split Bregman* [52] afin d'accélérer les temps de calcul de la méthode fondée sur les approximations linéaires successives du chapitre 5. Cette méthode sert en particulier à reconstruire un signal entier à partir d'une petite quantité d'informations. L'intérêt de cette méthode est qu'elle ne fait pas intervenir des programmes linéaires mais seulement des produits de matrices et une résolution d'un système d'équations linéaires. Ainsi avec cette méthode nous espérions nous affranchir des longs temps de calcul des programmes linéaires. Malheureusement cette méthode n'a pas pu converger vers un résultat. Nous présentons tout de même la méthode et son adaptation à notre problème d'optimisation.

A.1 Principe de la méthode

La méthode *Split Bregman* est une variante de la méthode de Bregman [16].

A.1.1 Méthode de Bregman

La méthode de Bregman sert à résoudre les problèmes de la forme :

$$\underset{x}{\operatorname{argmin}}\, J(x)$$

sous la contrainte d'égalité de la forme $Ax = b$ où J est une fonction convexe sur $\mathbb{R}^n$ et $x \in \mathbb{R}^n$.

D'abord nous commençons par aborder le concept de *distance de Bregman*. La distance de Bregman d'une fonction J en un point $x^{(iter)}$ est égale à

$$B(x, x^{(iter)}) = J(x) - J(x^{(iter)}) - \langle \nabla J(x^{(iter)}), x - x^{(iter)} \rangle$$

Elle représente l'écart entre la fonction et sa linéarisation par le développement de Taylor au premier ordre.

Le problème de minimisation de $J(x)$ sous la contrainte $Ax = b$ peut être résolu itérativement et s'écrit alors : trouver x qui minimise $B(x, x^{(iter)}) + \|Ax - b\|_2^2$ où $x^{(iter)}$ représente la valeur de x à l'itération précédente.

Afin de trouver x qui minimise l'expression, nous la dérivons par rapport à x et nous obtenons

l'équation :

$$\nabla J(x) - \nabla J(x^{(iter)}) + A^t(Ax - b) = 0$$

Au préalable l'expression de ∇J doit être calculée. À chaque itération nous devons résoudre cette équation pour trouver l'inconnue x qui est affecté à $x^{(iter+1)}$.

La convergence de la méthode a été démontrée. Quand *iter* tend vers $+\infty$: $x^{(iter)}$ tend x^* qui est la valeur de x qui minimise $J(x)$ sous la contrainte $Ax = b$. De plus, sous les conditions d'optimalité il a été démontré que :

$$\nabla \frac{1}{2}\|Ax^* - b\|_2^2 + \nabla J(x^*) = 0$$

a Cas particulier

Si le problème est de la forme :

$$\min_x \|x\|_1 + \lambda\|y - x\|_2^2$$

il a été démontré que la solution exacte à ce problème est $x = skrink(y, \lambda)$ avec

$$shrink(x, \lambda) = \begin{cases} 0 & \text{si } |x| \leq \frac{1}{\lambda} \\ x - \frac{1}{\lambda} & \text{si } x > \frac{1}{\lambda} \\ x + \frac{1}{\lambda} & \text{sinon.} \end{cases}$$

A.1.2 Méthode *Split Bregman*

La méthode *Split Bregman* s'appuie sur la méthode de Bregman. Elle sert à résoudre les problèmes de la forme :

$$\min_x \|\Phi(x)\|_1 + H(x)$$

où $\|\Phi(\cdot)\|_1$ et H sont des fonctions convexes sur $\mathbb{R}^n$ et $x \in \mathbb{R}^n$

L'algorithme sur lequel s'appuie la méthode est itératif. Chaque itération est constituée de 3 étapes :

1. $x^{iter+1} = \text{argmin}_x H(x^{iter}) + \frac{\lambda}{2}\|d - \phi(x^{iter}) - b^{iter}\|_2^2$
2. $d^{iter+1} = \text{argmin}_d \|d\|_1 + \frac{\lambda}{2}\|d - \phi(x^{iter}) - b^{iter}\|_2^2$
3. $b^{iter+1} = b^{iter} + \phi(x^{iter+1}) - d^{k+1}$

A.2 Adaptation de la méthode

Nous adaptons cette méthode pour qu'elle minimise la norme ℓ_p de notre vecteur x qui est la concaténation des vecteurs x_i.

Précédemment, au chapitre 5 nous avons formulé le problème sous la forme d'un programme linéaire en nombres entiers avec les contraintes suivantes :

$$\forall 1 \leq i \leq n, \quad \sum_{s=1}^{s=nslot_i} x_{i,s} = 1 \tag{A.1}$$

$$\forall 1 \leq t \leq T, \sum_{i=1}^{i=n} \sum_{s=1}^{s=nslot_i} x_{i,s} \times run_{i,s,t} \times proc_{i,C_{i,s}} \leq m \tag{A.2}$$

Les expressions A.1 et A.2 peuvent s'exprimer sous forme matricielle comme suit : $Gx = e$ où e est un vecteur contenant que des "1" et $Xx \leq m \times e$

A.3 Sans les contraintes d'utilisation des ressources (inéquation A.2)

Pour comprendre la méthode du *Split Bregman*, d'abord nous ne tenons pas compte des contraintes liées à la limitation des ressources et nous prenons pour l'instant $\phi(x) = x$. En revanche, afin de résoudre le problème du chapitre 5, dans le paragraphe suivant nous tenons compte des contraintes de ressources. Nous appliquons scrupuleusement la méthode *Split Bregman* :

Pour adapter la méthode à notre problème, nous posons

$$H(x) = \|Gx - e\|_2^2$$

et

$$\phi : x \to x$$

A.3.1 Étape 1

À chaque itération nous définissions x^{iter+1} de la manière suivante :

$$x^{iter+1} = \underset{x}{\operatorname{argmin}} \left(\|Gx - e\|_2^2 + \frac{\lambda}{2} \|d - x - b\|_2^2 \right) \tag{A.3}$$

cela revient à trouver x où le gradient s'annule :

$$\nabla \left(\|Gx - e\|_2^2 + \frac{\lambda}{2} \|d - x - b\|_2^2 \right) = 0$$

à l'aide des formules mathématiques matricielles, on obtient une équation équivalente :

$$G^T(Gx^{iter+1} - e) - \frac{\lambda}{2}(d - x^{iter+1} - b^{iter}) = 0 \tag{A.4}$$

$$\iff G^T G x^{iter+1} + \frac{\lambda}{2} x^{iter+1} - G^T e - \frac{\lambda}{2} d^{iter} + \frac{\lambda}{2} b^{iter} = 0 \tag{A.5}$$

$$\iff (G^T G + \frac{\lambda}{2} I) x^{iter+1} = G^T e + \frac{\lambda}{2} d^{iter} - \frac{\lambda}{2} b^{iter} \tag{A.6}$$

Avec I la matrice identité. Nous résolvons ce système linéaire afin de trouver x^{iter+1}. Si une composante de x^{iter+1} est négative alors nous la mettons à 0.

A.3.2 Étape 2

Nous suivons l'étape 2. Comme nous venons de le voir au paragraphe A.1.1.a, nous pouvons calculer d^{iter+1} à l'aide de la fonction *shrink*.

$$d^{iter+1} = shrink\left(x^{iter+1} + b^{iter}, \frac{1}{\lambda}\right)$$

A.3.3 Étape 3

Il suffit de suivre l'étape 3 et nous calculons b^{iter+1}.

$$b^{iter+1} = b^{iter} + x^{iter+1} - d^{iter+1}$$

A.4 Avec les contraintes d'utilisation des ressources(inéquation A.2)

Nous ajoutons maintenant à l'équation A.3 une pénalisation afin d'exprimer le fait que l'utilisation totale ne doit pas dépasser la quantité de PEs disponibles (*Processing Elements*) [43].

Cette pénalisation s'exprime comme suit :

$\langle u, (Xx - me)\rangle$ ou sous forme matricielle $u^T(Xx - me)$ avec un vecteur u tel que $u \geq 0$

De plus nous souhaitons que le vecteur x soit très creux, c'est pourquoi nous définissons la fonction ϕ qui associe à tout vecteur x le vecteur y tel que $y_i = x_i^p$. Ainsi $\|\phi(x)\|_1 = \|x\|_p^p$. Pour simplifier les calculs, nous linéarisons $\phi(x)$ avec l'astuce *Iteratively Reweighted Algorithm* présenté par Chartrand [26] et dont nous avons présenté le principe au chapitre 5 ($x^p = \frac{1}{x^{1-p}}x$). Nous exprimons la fonction ϕ linéarisée sous forme matricielle. $\phi(x) = D(w^{iter})x$ où $D(w^{iter})$ est une matrice diagonale où chaque élément de la diagonale est :

$$w_i^{iter} = \frac{1}{|x^{iter}|^{1-p} + \epsilon}$$

Le problème est présenté sous la forme adéquat pour être résolu par la méthode *Split Bregman* :

$$\underset{x}{\operatorname{argmin}} \|\Phi(x)\|_1 + H(x)$$

A.4.1 Étape 1

Nous procédons de la même façon que précédemment.

Nous cherchons à trouver x :

$$x^{iter+1} = \underset{x}{\operatorname{argmin}} \left(\|Gx - e\|_2^2 + \frac{\lambda}{2} \|d - x - b\|_2^2 + u^T(Xx - me) \right) \tag{A.7}$$

Nous obtenons alors l'équation suivante :

$$\left(G^T G + \frac{\lambda}{2} D(w^{iter}) \right) x = G^T e + \frac{\lambda}{2}(d^{iter} - b^{iter}) - X^T u^{iter} \tag{A.8}$$

Nous résolvons ce système linéaire puis nous calculons :

$$u^{iter+1} = u^{iter} + \gamma(Xx^{iter+1} - me)$$

avec γ une constante et si une composante de u^{iter+1} est négative alors nous mettons cette composante à 0. Les étapes 2 et 3 restent inchangées.

A.5 Synthèse

Nous avons programmé l'algorithme de la méthode *Split Bregmann* en C++ avec Gnu Scientific Library et en Matlab. Mais l'algorithme n'a pas convergé pour notre problème. À l'heure actuelle nous ne savons pas pourquoi cette méthode n'a pas fonctionné. Peut-être qu'il faudrait essayer d'appliquer la méthode sans linéariser ϕ.

Annexe B

Méthode utilisant *Iteratively Reweighted Algorithm*

Cette annexe traite d'une méthode utilisant un algorithme itératif [26], appelé *Iteratively Reweighted algorithm*. Nous avons essayé de mettre en œuvre cette méthode pour résoudre le problème d'ordonnancement d'une collection de jobs indépendants dans une machine multiprocesseur. L'idée générale de la méthode repose sur la minimisation de la norme ℓ_p d'un vecteur x élevée à la puissance p sujet à des contraintes d'inégalité. Ainsi nous souhaitons minimiser $\|y\|_p^p$. En résumé dans cette méthode l'algorithme transforme l'expression $\|y\|_p^p$ afin d'être intégrée dans un programme quadratique. La transformation est fondée sur la formule suivante : $y^p = y^{p-2} \times y^2$. Ainsi à chaque itération de l'algorithme nous exécutons un programme quadratique dont la fonction objectif est de la forme $y^{(iter)p-2} \times y^2$ où $y^{(iter)}$ représente la solution de l'itération précédente.

B.1 Idées avec les fonctions indicatrices

L'idée première de cette méthode est fondée sur les fonctions indicatrices f_i. En mathématique une fonction indicatrice $\chi(x)$ sur un ensemble F notée 1_F est égale à 1 si $x \in F$ et 0 sinon.

$f_i(t)$ représente le nombre de PEs alloués au job i en fonction du temps t. La collection contient n jobs à ordonnancer dans le cluster à m PEs.

Nous avons la contrainte suivante à respecter :

$$\forall t,\ G(t) = \sum_{i=1}^{i=n} f_i(t) \leq m$$

Afin de simplifier les calculs nous faisons l'hypothèse que la surface S_i de chaque job i est constante. S_i est égale au produit du nombre de PEs requis $proc_i$ et le temps réservé par l'utilisateur du job i.

Soit $y_i(t) = f_i(t)$

Ainsi,

$$S_i = \int_0^{+\infty} y_i(t)dt$$

Nous souhaitons minimiser le makespan

$$min\{t|G(t) = 0\}$$

B.2 Notations

Nous posons les notations suivantes :

- $y_i(t)$: le nombre de PEs utilisés par le job i à l'instant t ;
- $y_{i,j}$: le nombre de PEs utilisés par le job i à l'instant indexé par j ;
- $proc_i$: le nombre maximal de PEs requis par le job i ;
- $\overrightarrow{proc_i}$: un vecteur de taille T ne contenant que la valeur $proc_i$;
- $\overrightarrow{0_T}$: le vecteur nul de taille T

B.3 La formulation du problème

Ici, nous modélisons l'occupation de chaque job sur les PEs d'une machine multiprocesseur par des fonctions qui indiquent le nombre de PEs utilisés en fonction du temps. Nous discrétisons le temps et à chaque instant t nous attribuons à chaque job i le nombre de PEs $y_{i,t}$ utilisé par le job i. Comme chaque job i utilise un nombre constant de PEs durant son exécution, le vecteur δy_i formé de la différence $y_{i,t} - y_{i,t-1}$ doit être très creux : ce vecteur n'est composé que de 0 sauf à l'index correspondant au début du job et à l'index correspondant à la fin du job.

Nous prenons un horizon du temps T et nous vérifions si le problème donne une solution suffisamment creuse. Nous divisons le temps en plusieurs intervalles. D'abord nous prenons un intervalle de temps dont la longueur est égale à 1, donc nous avons T intervalles de temps. Nous représentons $y_i(t)$ comme un vecteur colonne de taille T.

$$y_i = \begin{pmatrix} y_{i,1} \\ \vdots \\ y_{i,T} \end{pmatrix}$$

Soit $\delta(y_i)_t$ la différence successive des composantes du vecteur y_i. $\delta(y_i)$ est un vecteur colonne de taille $T-1$. Concrètement $\delta(y_i)_t$ représente la différence entre le nombre de PEs utilisés par le job i à l'instant t et celui celui à l'instant $t-1$.

$$\delta(y_i) = \begin{pmatrix} y_{i,2} - y_{i,1} \\ \vdots \\ y_{i,T} - y_{i,T-1} \end{pmatrix}$$

Pour résoudre le problème nous définissons un ensemble de contraintes. Nous exprimons la contrainte B.1. Elle signifie qu'à chaque instant, chaque job i utilise au plus $proc_i$ PEs.

$$\forall 1 \leq i \leq n,\ \overrightarrow{0_T} \leq y_i \leq \overrightarrow{proc_i} \tag{B.1}$$

.

La contrainte B.2 signifie qu'à chaque instant, la somme du nombre de PEs utilisés par chaque job est inférieur au nombre m de PEs disponibles dans la machine.

$$\forall 0 \leq t \leq T,\ \sum_{i=1}^{i=n} y_{i,t} \leq m \tag{B.2}$$

La contrainte B.3 exprime que la surface du job i est égale à sa surface initiale S_i.

$$\forall 1 \leq i \leq n,\ \sum_{t=1}^{t=T} y_{i,t} = S_i \tag{B.3}$$

Sous les contraintes B.1 à B.3, nous définissons la fonction objectif B.4 qui permet de minimiser la norme ℓ_p de chaque vecteur $\delta(y_i)$:

$$\frac{1}{2}\|\sum_{i=1}^{i=n}\delta(y_i)\|_2^2 + \lambda_1\|\delta(y_1)\|_p^p + \ldots \lambda_n\|\delta(y_n)\|_p^p \tag{B.4}$$

La partie $\lambda_1\|\delta(y_1)\|_p^p + \ldots \lambda_n\|\delta(y_n)\|_p^p$ n'est pas quadratique, donc grâce au principe de *reweighted algorithm* décrit dans [26], nous transformons le programme quadratique en un algorithme iteratif qui utilise un programme quadratique.

Soit x un vecteur de taille T

$\|x\|_p^p$ avec la contrainte d'égalité $A \times x = b$

$$\Longleftrightarrow$$

$\sum_{t=1}^{t=T} |x_t|^p$ sous la contrainte $A \times x = b$

$$\Longleftrightarrow$$

$\sum_{t=1}^{t=T} |x_t|^{p-2}|x_t|^2$ sous la contrainte $A \times x = b$

Nous définissons itérativement :

$$x^{(l+1)} \longleftarrow \underset{x}{\operatorname{argmin}} \sum_{t=1}^{t=T} |x_t^{(l)}|^{p-2}|x_t|^2$$

sous la contrainte $A \times x = b$

Il est à noter que l'algorithme *reweighted* fonctionne avec une contrainte d'égalité ($A \times x = b$) et nous pensons qu'il fonctionne avec une contrainte d'inégalité ($A \times x \leq b$).

On note $\Delta y_i^{(l)}$, $\delta(y_i)^{(l)}$ à l'itération l. La fonction objectif de notre problème à l'itération $(l+1)$ s'écrit de la manière suivante :

$$\Delta y_i^{(l+1)} \longleftarrow \underset{\Delta y_i}{\operatorname{argmin}} \frac{1}{2} \| \sum_{i=1}^{i=n} \Delta y_i \|_2^2 + \sum_{i=1}^{i=n} \lambda_i \sum_{t=1}^{t=T} w_{t,i}^{(l)} (\Delta y_i)_t^2$$

avec

$$w_{t,i}^{(l)} = (\Delta y_i^{(l)})_t^{p-2}$$

ou mieux (voir l'article de Chartrand [26]) avec

$$w_{t,i}^{(l)} = (\Delta y_i^{(l)} + \epsilon)_t^{\frac{p}{2}-1}$$

Ainsi nous venons de définir un programme quadratique qui est utilisé dans l'algorithme itératif 16.

Algorithme 16: programme quadratique avec *iteratively reweighted algorithm*

Entrées : T : l'horizon du temps
$proc_i$: le nombre de PEs requis par le job i
S_i : la surface du job i
n : le nombre de jobs
m : le nombre de PEs dans le cluster
L : le nombre maximum d'itérations
Sorties : y_i : le vecteur du job i qui contient le nombre de PEs utilisés par le job i en fonction du temps
Données : $\Delta y_i^{(l)}$

```
1 l ← 0 tant que l < L faire
2 |   trouver et stocker les vecteurs y_i^(l) qui minimisent la fonction objectif du programme
  |   quadratique ci-dessus
3 |   l ← l + 1
4 retourner y_i^(l)
```

B.4 L'algorithme

Nous utilisons une dichotomie pour approcher l'optimal (algorithme 17). Il commence à fixer deux bornes d'horizon de temps a et b. Puis il exécute l'algorithme 16 pour trouver une solution entre ces deux bornes . Quand il trouve une solution, il met à jour les deux bornes a et b.

B.5 Résultats des essais expérimentaux

Nous avons implémenté cet algorithme en *python* en utilisant un solveur quadratique inclus dans le logiciel *OpenOPT* `openopt.org`. Malheureusement cela n'a pas donné de résultats probants. Pour chaque job le programme quadratique trouve $\delta(y_i) = \overrightarrow{0_T}$, ce qui constitue la solution la plus creuse possible. Cela signifie que nous obtenons une solution où chaque job

Algorithme 17: Dichotomie avec un programme quadratique

Entrées : T : l'horizon du temps
$proc_i$: le nombre de PEs requis par le job i
S_i : la surface du job i
n : le nombre de jobs
m : le nombre de PEs dans le cluster
ϵ_T : le seuil d'arrêt de la dichotomie
Sorties : y_i : le vecteur du job i qui contient le nombre de processeur utilisé par le job i en fonction du temps
Données : y'_i : copie de y_i
a : borne inférieure de T
b : borne supérieure de T
me : moyenne de a et b

1 $a \leftarrow 0$
2 $b \leftarrow T$
3 **répéter**
4 $\quad me \leftarrow \frac{a+b}{2}$
5 $\quad y_i \leftarrow$ exécuter l'algorithme 16 avec les paramètres $(proc_i, S_i, n, m)$ et l'horizon de temps me
6 $\quad$ **si** *(les vecteurs y_i sont "assez creux")* **alors**
7 $\quad\quad b \leftarrow m \times e$
8 $\quad$ **sinon**
9 $\quad\quad a \leftarrow m \times e$
10 $\quad$
11 **jusqu'à** $(b - a < \epsilon_T)$

utilise un nombre rationnel de PEs de la date 0 jusqu'à l'horizon du temps T. Nous n'avons pas pu exploiter ces nombres rationnels, car nous avons besoin d'un nombre de PEs entier pour ordonnancer les jobs. Nous avons donc mis cette voie de côté.

Annexe C

Algorithme Espérance-Maximisation (EM)

L'algorithme *EM* (*Expectation-Maximization*) expliqué dans [54] sert à trouver les paramètres d'un modèle probabiliste qui maximisent la probabilité d'une variable observée.

En fait nous utilisons cet algorithme pour approximer les fonctions indicatrices présentées dans la section B.1 par des gaussiennes. Nous souhaitions utiliser cet outil statistique pour résoudre le problème d'ordonnancement d'une collection de jobs moldables indépendants dans une machine multiprocesseur. Nous approximons le nombre de PEs utilisés par un job en fonction du temps, qui est plutôt une fonction indicatrice rectangulaire, par une fonction gaussienne. Avec l'algorithme EM nous espérions que les mélanges des gaussiennes, rentrent tous dans un rectangle dont les dimensions sont limitées horizontalement par un horizon de temps (utilisé précédemment dans la chapitre 5) et verticalement par le nombre de PEs de la machine.

La figure C.1 montre un résultat obtenu avec l'algorithme EM avec une instance du problème d'ordonnancement d'une collection de 6 jobs moldables indépendants dans une machine à 5 PEs.

C.1 Application de EM au problème d'ordonnancement

Soit T l'horizon du temps. A chaque job i est associée la fonction $\frac{1}{T}1_{[0,T]} = \sum_{i=1}^{n} \Pi_i \times \varphi_i$ avec $\Pi_i = \frac{S_i}{\sum_{i=1}^{n} S_i}$. Nous rappelons que S_i est la surface du job i.

On suppose que chaque job i a une surface constante S_i.

Les fonctions indicatrices de chaque job i sont approximées par des gaussiennes

$$\varphi_i(t) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{\frac{\frac{1}{2}(t-\mu_i)^2}{\sigma_i^2}}$$

Le paramètre à déterminer est $\theta^{(l)} = (\mu_1^{(l)}, \ldots, \mu_n^{(l)}, \sigma_1^{(l)}, \ldots, \sigma_n^{(l)})$. C'est à dire les paramètres moyenne et écart-type de chaque gaussienne.

Nous définissons la vraisemblance :

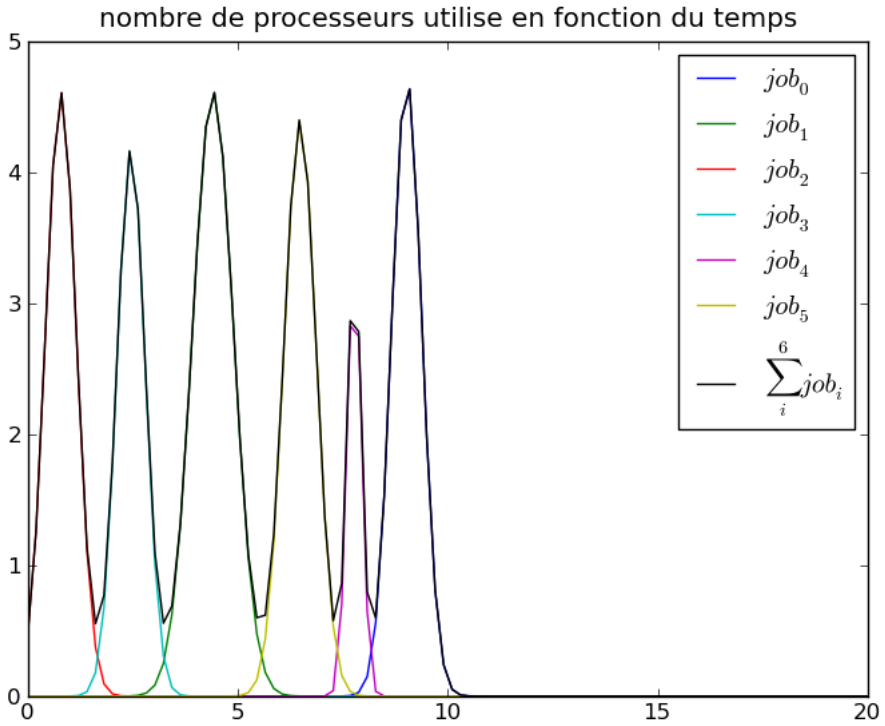


Figure C.1 – Exemple de gaussiennes pour 6 jobs moldables

$$L = \prod_{k=1}^{K} \sum_{i=1}^{n} \Pi_i \varphi_i(x_k)$$

Nous notons i_k le job qui tourne au temps x_k . Nous définissons la vraisemblance complète :

$$L^c(\theta) = \prod_{k=1}^{K} \Pi_{i_k} \varphi_{i_k}(x_k)$$

Nous calculons le log de l'expression précédente afin de supprimer le produit.

$$\log(L^c(\theta)) = l^c(\theta) = \sum_{k=1}^{k} (\log(\Pi_{i_k}) + \log(\varphi_{i_k}(x_k)))$$

Nous calculons l'espérance :

$$E\left[l^c(\theta)|x_1, \ldots, x_k\right] = \sum_{k=1}^{K} \sum_{i=1}^{n} \log(\Pi_i) P(i_k = i|x_k) + \sum_{i=1}^{n} \varphi_i(x_k) P(i_k = i|x_k)$$

Notons

$$\tau_{k,i} = P(i_k = i|x_k)$$

Avec le théorème de Bayes, nous avons :

$$P(i_k = i|x_k) = \frac{P(x|i_k = i)P(i_k = i)}{\sum_{i=1}^{n} P(x|i_k = i)P(i_k = i)}$$

On peut facilement calculer $P(i_k = i|x_k)$, en effet la probabilité $P(i_k = i)$ de tomber sur le job i au moment x_k est la fraction de la surface du job i sur la surface totale, autrement dit : Pi_i. La probabilité $P(x|i_k = i)$ de se situer à l'instant x sachant qu'on exécute le job i est $\varphi_i(x_k)$.

$$\varphi_i^{(l)}(t) = \frac{1}{\sqrt{2\pi\sigma_i^{(l)2}}} e^{\frac{\frac{1}{2}(t-\mu_i^{(l)})^2}{\sigma_i^{(l)2}}}$$

Nous obtenons :

$$E_{\theta^{(l+1)}}\left[l^c(\theta^{(l)})|x_1, \ldots, x_k\right] = \sum_{k=1}^{K}\sum_{i=1}^{n}\left(\log(\Pi_i) + \varphi_i^{(l)}(x_k)\tau_{k,i}\right)$$

Nous cherchons à maximiser l'espérance, donc nous cherchons $\theta^{(l+1)}$ qui maximise :

$$\sum_{k=1}^{K}\sum_{i=1}^{n}\varphi_i^{(l)}(x_k)\tau_{k,i}$$

En dérivant par rapport à μ_i et à σ_i nous obtenons les valeurs de ces paramètres à chaque itération :

$$\mu_i^{(l+1)} = \frac{\sum_{k=1}^{K}\tau_{k,i}x_k}{\sum_{k=1}^{K}\tau_{k,i}}$$

$$(\sigma_i^{(l+1)})^2 = \frac{\sum_{k=1}^{K}\tau_{k,i}(x_k - \mu_i^{(l)})^2}{\sum_{k=1}^{K}\tau_{k,i}}$$

Nous itérons ainsi plusieurs fois en maximisant l'espérance, l'algorithme de base converge et donne une valeur de θ.

C.2 Algorithmes

L'idée est d'utiliser l'algorithme EM pour déterminer un ordonnancement dont le makespan est inférieur ou égal à un horizon de temps T fixé. On diminue cet horizon T dès que l'algorithme EM trouve une solution. Dans le cas contraire on s'arrête. Tout d'abord on commence à calculer un horizon de temps T qui est égal au makespan de l'ordonnancement obtenu par l'algorithme LTF. On calcule les surfaces $surface_i$ des jobs i qui demandent $reqproc_i$ PEs et durent $reqtime_i$.

Normalement on effectue plusieurs itérations et nous laissons l'algorithme EM converger vers une solution pour T fixé. Cependant nous avons remarqué que EM converge après plusieurs ité-

rations vers une solution médiocre en terme de makespan. C'est pourquoi nous avons décidé qu'à chaque itération de EM nous vérifions si l'ordonnancement a un makespan ($innerMakespan$) inférieur à T.

Pour passer des gaussiennes à un ordonnancement, nous procédons comme suit. À chaque pas de l'algorithme EM, nous obtenons la moyenne de la gaussienne μ_i qui partage *le rectangle* caractéristique du job i en deux : ceci permet de savoir quand débute le job i. Nous obtenons également l'écart-type de la gaussienne σ_i : ceci permet de connaître la durée du job i. Nous prenons en première approximation $duration0_i = 2 \times sigma_i$ comme étant la durée du job i. À partir de cette durée de job, nous calculons le nombre approximatif de PEs $nbProcs0_i$ alloués au job i. Si besoin, on réduit le nombre de PEs alloués au job i jusqu'à $nbProc_i$ de telle sorte $\left\lceil \frac{reqproc_i}{nbProc_i} \right\rceil = \left\lceil \frac{reqproc_i}{nbProc0_i} \right\rceil$. Nous devons alors calculer la durée du job i $duration_i$ ainsi que la date de début du job i : $starttime_i$. Puis nous compactons les jobs à l'aide d'un ordonnancement de type EST (*Earliest Start Time*) afin de boucher les trous.

C.3 Résultats expérimentaux

Nous avons réalisé des expériences avec des lots de jobs moldables à ordonnancer sur une machine de 64 PEs et 128PEs, le nombre d'échantillons x_k est de $K = 200$, $maxIter = 200$. Pour chaque taille de lots nous réalisons 40 simulations.

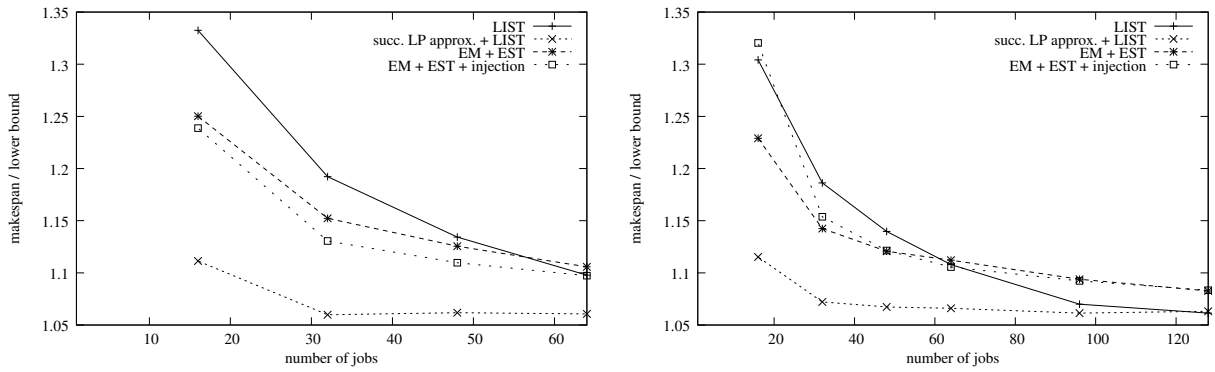
Les figures C.2a et C.2b représentent le rapport moyen du makespan de l'ordonnancement obtenu par l'algorithme EM à la borne inférieure du makespan des machines de 64 PEs et de 128 PEs. Dans les figures $LIST$ est l'algorithme LTF, succ. LP approx. + LIST représente l'algorithme du chapitre 5 fondé sur les approximations linéaires successives d'un modèle creux et l'algorithme LTF. EM+EST représente l'algorithme EM appliqué à notre problème d'ordonnancement (algorithme 18). EM+EST+injection représente une version de l'algorithme EM+EST pour lequel à chaque itération l'ordonnancement compacté par EST est injecté dans les paramètres μ_i et σ_i pour être utilisés par EM. Avec 64 PEs les deux versions de l'algorithmes EM trouvent de meilleurs ordonnancements que l'algorithme LTF. La version de l'algorithme EM+EST+injection trouve de meilleurs ordonnancements que la version EM+EST. Néanmoins, avec 128 PEs les deux versions de l'algorithme EM sont dépassées par l'algorithme LTF à partir de 64 jobs. Les temps CPU sont donnés à titre indicatifs. Nous remarquons que le temps utilisé par les deux versions de algorithme EM sont gourmands en temps de calcul. Il leur faut environ 20 minutes pour ordonnancer une collection de 128 jobs sur une machine de 128 PEs.

Nous avons adapté l'algorithme EM+EST pour qu'il tourne sur une carte graphique à l'aide de l'API Cuda[1]. Dans notre programme Cuda, nous associons à chaque bloc du GPU à un job i et à chaque thread de chaque bloc un échantillon k. À vu d'œil, la vitesse des temps de calcul est grandement réduite, mais nous n'avons pas pu augmenter le nombre d'échantions K à cause de problèmes probablement dûs aux erreurs inhérents au nombres flottants codés en simple précision.

[1] `http://developer.nvidia.com/cuda/what-cuda`

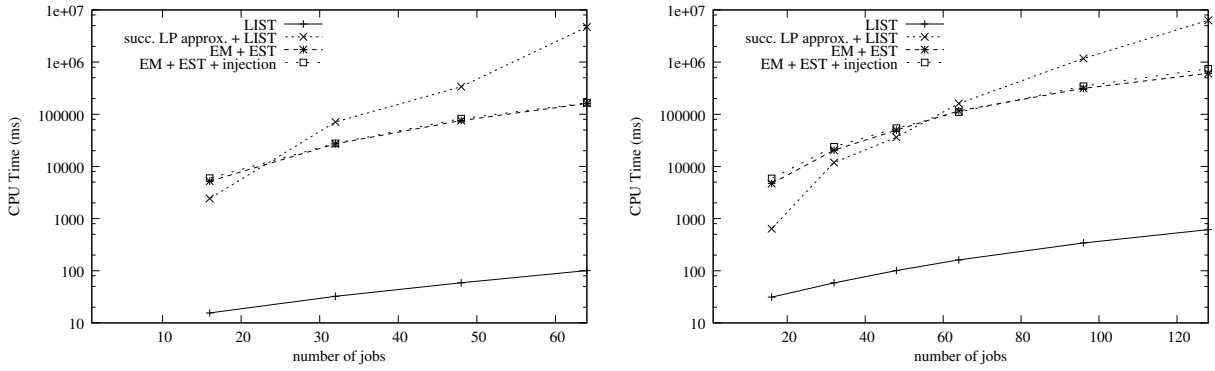
Algorithme 18: `ordonnanceurEM(plateforme,lotJobs)`

$lb \leftarrow$ borne inférieure du makespan
$sched \leftarrow$ calculer un ordonnancement avec l'algorithme LTF
$listMakespan \leftarrow makespan(sched)$
$T \leftarrow listMakespan$
$end \leftarrow false$
$incT \leftarrow false$
$\forall i \in lotJobs\ surface_i \leftarrow reqtime_i \times reqproc_i$
tant que $T > lb$ *et non* end **faire**
  $iter \leftarrow 1$; $found \leftarrow faux$
  $\forall i,\ \mu_i \leftarrow alea(0..T)$
  $\forall i,\ sigma_i \leftarrow alea(0..T)$
  tant que $iter < maxIter$ *et non* $found$ **faire**
    $\mu_i \leftarrow \frac{\sum_{k=1}^{K} \tau_{k,i} x_k}{\sum_{k=1}^{K} \tau_{k,i}}$
    $\sigma_i \leftarrow \sqrt{\frac{\sum_{k=1}^{K} \tau_{k,i} (x_k - \mu_i^{(iter)})^2}{\sum_{k=1}^{K} \tau_{k,i}}}$
    $\forall i \in lotJobs\ duration0_i \leftarrow 2 \times sigma_i$
    $\forall i \in lotJobs\ nbProcs0_i \leftarrow \frac{surface_i}{duration0_i}$
    $\forall i \in lotJobs$ choisir $nbProc_i$ tel qu'on utilise le moins de PEs possible et qui fait multiplier $reqtime_i$ par le même facteur que $nbProcs0_i$
    $\forall i \in lotJobs\ duration_i \leftarrow reqtime_i \times \left\lceil \frac{reqproc_i}{nbProcs_i} \right\rceil$
    $\forall i \in lotJobs\ starttime_i \leftarrow mu_i - \frac{duration_i}{2}$
    trier les jobs i de $lotJobs$ dans l'ordre croissant des dates de début $starttime_i$
    $sched \leftarrow$ ordonnancer avec un algorithme classique de LIST les jobs triés $i \in lotJobs$ avec $(nbProcs_i, duration_i)$
    $innerMakespan \leftarrow makespan(sched)$
    si $innerMakespan < T$ **alors**
      $bestSched \leftarrow sched$
      $T \leftarrow T - 1$
      $found \leftarrow vrai$
      si $incT = vrai$ **alors**
        $end \leftarrow vrai$
  $\forall i, k,\ \mu_i^{(iter)} \leftarrow \mu_i$
  $\forall i, k,\ \sigma_i^{(iter)} \leftarrow \sigma_i$
  $iter \leftarrow iter + 1$
  si $non\ found$ **alors**
    si $T = listMakespan$ **alors**
      $incT \leftarrow vrai$
    si $incT = vrai$ **alors**
      $T \leftarrow T + 1$
    sinon
      $end \leftarrow vrai$
retourner $bestSched$



(a) Avec 64 PEs **(b)** Avec 128 PEs

Figure C.2 – Performance



(a) Avec 64 PEs **(b)** Avec 128 PEs

Figure C.3 – Temps de calcul

C.4 Synthèse

Dans cette annexe nous avons proposé d'utiliser un outils provenant du domaine des statistiques pour ordonnancer une collection de jobs moldables sur une machine multiprocesseur : l'algorithme *Espérance-Maximisation*. Nous proposons deux algorithmes fondées sur cet outil statistique. Nous avons approximer l'utilisation des PEs (fonctions indicatrices rectangulaires du temps) par des gaussiennes. Les résultats obtenus sont moins bons que ceux obtenus avec les algorithmes fondés sur les linéarisations successives d'un modèle creux présentés dans le chapitre 5. Pour des travaux futurs il faudrait approximer les fonctions indicatrices par des fonctions plus rectangulaires.

Annexe D

Programmes linéaires pour l'ordonnancement *online*

Dans cette annexe nous nous intéressons à un problème d'ordonnancement de jobs moldables qui arrivent dans un cluster homogène. Ces travaux suivent ceux présentés au chapitre 4. Nous souhaitons toujours optimiser l'utilisation des ressources, pour ce faire, nous proposons divers algorithmes fondés sur la résolution de deux programmes linéaires. L'un cherche à minimiser le *makespan* d'une fenêtre de jobs et un autre qui cherche à minimiser le *stretch moyen* d'une fenêtre de jobs. Nous comparons les performances de ces algorithmes aux algorithmes plus classiques comme le *FCFS* ou l'algorithme de *backfilling*.

Dans cette annexe, pour de meilleures performances, nous mettons en place les heuristiques mentionnées précédemment en nous aidant de deadlines. Ces deadlines sont calculées fenêtre par fenêtre en exécutant un simple algorithme de liste dans l'ordre d'arrivée des jobs de la fenêtre concernée. Dans les programmes linéaires nous ajoutons la contrainte qui dit qu'un job doit terminer avant sa deadline.

D.1 Un programme linéaire comme heuristique

Nous discrétisons le temps en K intervalles de temps. Avec le pliage de jobs, un job présente plusieurs *configurations*. Suivant le nombre de processeurs utilisés dans une configuration son temps d'exécution change.

D.1.1 programme linéaire minimisant le makespan

a Notations

Nous posons les notations suivantes. Les constantes sont :

- K : le nombre d'intervalles de temps ;
- k : l'index de l'instant sur une échelle de temps discrétisé ;
- i : l'index du job de la fenêtre considérée ;

- req_i : le nombre maximal de processeurs requis par le job i ;
- $proc_{i,j}$: le nombre de processeurs utilisés par le job i avec la configuration j ;
- $nbk_{i,j}$: le temps d'exécution requis par le job i avec la configuration j ;
- $dead_i$: la deadline associée au job i ;
- $conf_i$: le nombre de configurations possibles que possède le job i.

Les variables sont :

- $NC_{i,k} \in \mathbb{N}$: le nombre de processeurs utilisés par le job i entre l'instant k et $k+1$;
- $n_i \in \mathbb{N}$: le nombre de processeurs utilisés par le job i durant toute son exécution ;
- $b_{i,j} \in \mathbb{B}$ indique si la configuration j du job i est choisie ;
- $c_{i,k} \in \mathbb{B}$ indique si le job i s'exécute entre l'instant k et $k+1$;
- $o_k \in \mathbb{B}$ indique si un job s'exécute entre l'instant k et $k+1$.

b Formulation du programme linéaire en nombres entiers

Afin de définir un programme linéaire qui minimise le makespan de l'ordonnancement des jobs d'une fenêtre, nous définissons un ensemble de contraintes linéaires.

La contrainte (D.1) limite l'utilisation des ressources, on dit qu'à chaque instant k la somme du nombre de processeurs utilisés est inférieure au nombre total m de processeurs dans le cluster.

$$\forall k,\ \sum_{i=1}^{i=J} NC_{i,k} \leq m \tag{D.1}$$

La contrainte (D.2) indique que le système ne doit pas allouer à un job i plus de processeurs que ce qu'elle requiert.

$$n_i \leq req_i \tag{D.2}$$

La contrainte (D.3) permet de choisir un nombre donné de processeurs pour le job i parmi $conf_i$ configurations.

$$\forall i,\ n_i = \sum_{j=1}^{conf_i} b_{i,j}.proc_{i,j} \tag{D.3}$$

Pour chaque job i la variable binaire $b_{i,j}$ doit être égale à 1 et il n'y a qu'un j pour lequel $b_{i,j}$ est égal à 1. Ceci est imposée par la contrainte D.4.

$$\forall i,\ \sum_{j=1}^{conf_i} b_{i,j} = 1 \tag{D.4}$$

La contrainte (D.5) signifie que si une configuration j' est choisie alors la durée (en nombre d'intervalles de temps) du job i est $nbk_{i,j'}$.

$$\forall i,\ \sum_{k=1}^{K} c_{i,k} = \sum_{j=1}^{conf_i} b_{i,j}.nbk_{i,j} \tag{D.5}$$

La contrainte (D.6) dit que si un job s'exécute au temps e et au temps f avec $e < f$ alors elle s'exécute à n'importe quel temps g avec $e < g < f$. Nous appelons cette contrainte *continuité temporelle*.

$$\forall e < g < f,\ (1 - c_{i,e}) + (1 - c_{i,f}) + c_{i,g} \geq 1 \tag{D.6}$$

Expliquons la contrainte D.6 *continuité temporelle*. Nous savons en logique propositionnelle que $(A \Rightarrow B)$ est équivalent à $(\bar{A} \vee B)$ ou encore $((1 - \sigma(A)) + \sigma(B) \geq 1)$ avec $\sigma(X) = \begin{cases} 1 & , \ X \text{ est vrai} \\ 0 & , \ X \text{ est fausse} \end{cases}$ et donc $1 - \sigma(X) = \begin{cases} 0 & , \ X \text{ est vrai} \\ 1 & , \ X \text{ est fausse} \end{cases}$. Soient $C_{i,e}$, $C_{i,f}$ et $C_{i,g}$ les variables propositionnelles qui sont vraies si et seulement si le job i s'exécute respectivement au temps e, f, g. Nous exprimons la continuité temporelle de la manière suivante : si la tache i s'exécute à deux dates différentes e et f avec $e < f$ alors quelque soit la date g entre e et f, le job i s'exécute à la date g. Cette contrainte s'exprime en logique : $C_{i,e} \wedge C_{i,f} \Rightarrow C_{i,g}$. Nous transformons l'implication et nous obtenons : $\overline{C_{i,e} \wedge C_{i,f}} \vee C_{i,g}$. Avec la loi de De Morgan, nous obtenons $\overline{C_{i,e}} \vee \overline{C_{i,f}} \vee C_{i,g}$. En utilisant la fonction σ, on a $(1-\sigma(C_{i,e}))+(1-\sigma(C_{i,f}))+\sigma(C_{i,g}) \geq 1$. En posant $c_{i,e} = \sigma(C_{i,e})$, $c_{i,f} = \sigma(C_{i,f})$ et $c_{i,g} = \sigma(C_{i,g})$, on obtient l'inégalité D.6.

La contrainte (D.7) $\forall k,\ \frac{1}{J}.\sum_{i=1}^{J} c_{i,k} \leq o_k$ définit une variable binaire o_k qui indique si un job s'exécute au temps k.

$$\forall k,\ \frac{1}{J}.\sum_{i=1}^{J} c_{i,k} \leq o_k \tag{D.7}$$

Les contraintes (D.8), (D.9), (D.10) constituent la linéarisation du produit $NC_{i,k} = n_i \times c_{i,k}$.

$$NC_{i,k} \leq c_{i,k}.req_i \tag{D.8}$$

$$NC_{i,k} \leq n_i \tag{D.9}$$

$$NC_{i,k} \geq n_i - (1 - c_{i,k}).req_i \tag{D.10}$$

Elles sont issues de la formule de la linéarisation d'un produit d'une variable quelconque x par une variable binaire b. Si on souhaite linéariser le produit $y = b \times x$, alors on peut utiliser le système de contraintes suivant où $Sup\{x\}$ est une valeur qui majore toutes les valeurs possibles de x :

$$\begin{cases} y \leq b \times Sup\{x\} \\ y \leq x \\ y \geq x - (1 - b) \times Sup\{x\} \end{cases}$$

La contrainte (D.11) force le job i à terminer avant sa deadline.

$$\forall i,\ \forall k \geq dead_i,\ c_{i,k} = 0 \tag{D.11}$$

Le programme linéaire en nombres entiers qui minimise le makespan est le suivant :

Minimiser

$$\sum_{k=1}^{K} o_k \tag{D.12}$$

sous les contraintes D.1 à D.11.

D.1.2 Programme linéaire minimisant le stretch moyen

a Notations

Nous reprenons les notations du paragraphe D.1.1.a.

Nous ajoutons les constantes suivantes :

- tu : la durée d'un intervalle de temps ;
- a_i : la date de soumission du job i ;
- $req1_i$: la durée d'exécution du job i s'il était tout seul dans la machine.

Nous ajoutons les variables suivantes :

- $sc_{i,k} \in \mathbb{B}$ est égale à 1 si et seulement si le job i a déjà commencé au moment q tel que $1 \leq q \leq k$;
- $bs_{i,k} \in \mathbb{B}$ est égale à $1 - sc_{i,k}$;
- $st_i \in \mathbb{Q}^+$: la date de début d'exécution du job i ;
- $cpl_i \in \mathbb{Q}^+$: la date de fin d'exécution du job i ;
- $reqt_i \in \mathbb{Q}^+$: la durée d'exécution du job i.

b Formulation du programme linéaire en nombres entiers

Afin de définir un programme linéaire qui minimise le *stretch moyen* de l'ordonnancement des jobs d'une fenêtre, nous définissons un ensemble de contraintes linéaires.

Les contraintes (D.1) à (D.11) restent valables et inchangées pour ce problème.

$sc_{i,k}$ qui est une variable binaire est égale à 1 si et seulement si le job i a déjà commencé au moment q tel que $1 \leq q \leq k$, notons cette contrainte $\mathcal{C}$. $\mathcal{C}$ est vérifiée grâce à l'inégalité (D.13).

$$\forall i,\ \forall k,\ \frac{1}{K}\sum_{q=1}^{k} c_{i,q} \leq sc_{i,k} \leq \frac{K+1}{K}\sum_{q=1}^{k} c_{i,q} \tag{D.13}$$

Vérifions que la contrainte D.13 est équivalente à la contrainte $\mathcal{C}$. Dire qu'un job i a déjà commencé au moment q tel que $1 \leq q \leq k$ revient à dire que $\sum_{q=1}^{k} c_{i,q} \neq 0$: $c_{i,q} = 1$ indique que le job i tourne au moment q. Nous supposons que $\sum_{q=1}^{k} c_{i,q} = 0$. Alors avec les inégalités (D.13), nous pouvons dire que $0 \leq sc_{i,k} \leq 0$. Donc $sc_{i,k} = 0$. Nous avons prouvé la formule :

$$\forall i, k,\ \sum_{q=1}^{k} c_{i,q} = 0 \Rightarrow sc_{i,k} = 0$$

. Puisque $sc_{i,k}$ est binaire, Nous pouvons en déduire sa contraposée :

$$\forall i, k,\ sc_{i,k} = 1 \Rightarrow \sum_{q=1}^{k} c_{i,q} \neq 0$$

Maintenant nous supposons que $\sum_{q=1}^{k} c_{i,q} \neq 0$, alors nous pouvons écrire $\frac{1}{K} \sum_{q=1}^{K} c_{i,q} > 0$. Avec les inégalités (D.13), nous pouvons dire que $0 < \frac{1}{K} \sum_{q=1}^{K} c_{i,q} \leq sc_{i,k}$. Nous savons que $sc_{i,k} \in \{0, 1\}$. Donc $sc_{i,k} = 1$. Puisque $1 \leq \frac{K+1}{K}$, l'inégalité $sc_{i,k} \leq \frac{K+1}{K} \sum_{q=1}^{K} c_{i,q}$ de (D.13) est toujours cohérente. Nous avons prouvé :

$$\forall i, k,\ \sum_{q=1}^{k} c_{i,q} \neq 0 \Rightarrow sc_{i,k} = 1$$

Finalement nous avons prouvé :

$$\forall i, k,\ \sum_{q=1}^{k} c_{i,q} \neq 0 \Leftrightarrow sc_{i,k} = 1$$

Donc l'inégalité (D.13) respecte la contrainte $\mathcal{C}$.

La contrainte D.14 permet d'indiquer si à un instant k le job i n'a pas déjà commencé. $sc_{i,k}$ qui est une variable binaire est égale à 0 si et seulement si le job i a déjà commencé au moment q tel que $1 \leq q \leq k$.

$$\forall i, k,\ bs_{i,k} = 1 - sc_{i,k} \tag{D.14}$$

La contrainte D.15 permet de calculer la date de début du job i.

$$\forall i,\ st_i = \sum_{k=1}^{K} bs_{i,k} tu \tag{D.15}$$

La contrainte D.16 permet de calculer la durée du job i.

$$\forall i,\ reqt_i = tu \sum_{j=1}^{conf_i} b_{i,j} nbk_{i,j} \tag{D.16}$$

La contrainte D.17 permet de calculer la date de fin d'exécution du job i.

$$\forall i,\ cpl_i = st_i + reqt_i \tag{D.17}$$

Le programme linéaire en nombres entiers qui minimise le *stretch moyen* est le suivant :

Minimiser :

$$\sum_{i=1}^{J} \frac{cpl_i - a_i}{req1_i} \tag{D.18}$$

sous les contraintes précédentes (D.1) à (D.11) et les nouvelles (D.13) à (D.17).

D.1.3 L'algorithme général

Nous utilisons le programme linéaire dans l'algorithme 19.

Algorithme 19: Choix du job à exécuter en utilisant un programme linéaire

```
1  Évènement : un job i arrive début
2      attribuer une deadline au job i en se basant sur un algorithme de liste
3  Évènement : Des ressources sont libérées, il y a n > 0 nœuds disponibles début
4      initialiser le programme linéaire PL avec les jobs en cours d'exécution
5      prendre comme horizon du temps Z le plus grand des deadlines
6      discrétiser le temps de la date courante jusqu'à Z en K parties
7      essayer de lancer PL avec les y ≤ z jobs en attente
8      si PL ne trouve aucune solution alors
9          trier les y jobs dans l'ordre croissant des deadlines
10         choisir le job dont la deadline est la plus petite et qui peut se plier
11         si aucun job n'a été choisi alors
12             revenir à l'étape ligne 3
13     si PL a une solution alors
14         dans la solution, choisir un job qui commence à la date courante
```

D.1.4 L'algorithme avec programme linéaire + chaîne de Markov

Nous relaxons soit le programme linéaire en nombres entiers qui vise à minimiser le makespan (paragraphe D.1.1) soit celui qui vise à minimiser le stretch moyen (paragraphe D.1.2).

Après avoir résolu l'un ou l'autre des programmes linéaires relaxés, nous obtenons des valeurs réelles de $c_{i,k}$ (le job i tourne à l'instant k), $NC_{i,k}$ (le nombre de processeurs alloués au job i à l'instant k) pour chaque job i à chaque instant k. Pour chaque k depuis 0 jusqu'à l'horizon K nous rangeons les jobs i dans l'ordre décroissant des $c_{i,k}$ et nous gardons en mémoire le nombre de processeurs alloués au job i à l'instant k. Nous construisons un ordonnancement initial à partir de cette liste de job avec un algorithme de liste classique.

Cet ordonnancement x sert à l'initialisation de la chaîne de Markov. Nous reprenons l'algorithme 1 présenté au paragraphe 2.4.2 et $F(x)$ représente le makespan ou le stretch moyen de l'ordonnancement x. La fonction $GENERER\_VOISIN$ à la ligne 5 construit aléatoirement un autre ordonnancement à partir de l'ordonnancement x_n. Cette construction choisit aléatoi-

rement un job i, puis elle modifie soit le nombre de processeurs alloués au job i soit elle échange deux jobs dans la liste, auquel cas elle applique l'algorithme de liste.

D.1.5 Résultats expérimentaux (sans les chaînes de Markov)

Dans cette section nous évaluons les performances de notre algorithme qui utilise deux versions de programme linéaires (makespan ou stretch moyen). Nous comparons ses performances avec les algorithmes classiques tels que *FCFS* et *backfilling*. Nous avons conçu un simulateur de clusters homogènes et un ordonnanceur de jobs en C++ fondé sur Simgrid [23] et son API MSG [25]. Nous utilisons GNU Linear Programming Kit (GLPK) pour résoudre les programmes linéaires.

Ici nous considérons différentes métriques pour évaluer l'efficacité des algorithmes de plusieurs points de vues : un point de vue *système* et un point de vue utilisateur. Nous considérons le makespan comme un point de vue centré *système* et le stretch moyen comme point de vue centré *utilisateur*.

a Mise en place des expériences

Dans ces expériences nous utilisons un cluster de 1024 processeurs. Il y a 1 000 jobs à ordonnancer. Les paramètres de la génération des workloads synthétiques sont les suivants :

- $timeInterval = 3600\,s$;
- $rtMin = 1000\,s$: le temps minimal requis par un job ;
- $rtMax = 200\,000\,s$: le temps maximal requis par un job ;
- $diffMax = 40$: le pourcentage maximum entre le temps requis par un job et son temps d'exécution réelle ;
- $procMin = 1$: le nombre minimum de processeurs requis par un job ;
- $procMax = 1024$: le nombre maximum de processeurs requis par un job ;
- $\lambda \in \{10, 50, 100, 200, 300, 500\}$: le nombre moyen de jobs arrivant par intervalle de temps $timeInterval = 3600\,s$.

Pour chaque valeur λ, 20 workloads synthétiques sont générés. Alors pour chaque valeur de λ, la mesure est calculée par une moyenne arithmétique de 20 valeurs.

La figure D.1 montre le pourcentage d'amélioration du makespan par rapport à *FCFS* apportée par les différents algorithmes :

- $H_1(5)$, $H_2(5)$ et $H_1(5) + backfilling(5)$ sont les algorithmes avec une fenêtre de 5 jobs (section 4.5.4).
- $backfilling(5)$ est le *backfilling* avec une fenêtre de 5 jobs.

- *ILP-Folding(5, 10)* : implémentation de l'algorithme 19 utilisant le programme linéaire en nombres entiers qui minimise le makespan (de la section D.1.1) avec une fenêtre de 5 jobs et une échelle de temps divisée en 10.

- *ILP-Mean-Stretch-Folding(5, 10)* : implémentation de l'algorithme 19 utilisant le programme linéaire en nombres entiers qui minimise le stretch moyen (de la section D.1.2) avec une fenêtre de 5 jobs et une échelle de temps divisée en 10 unités de temps.

- *ILP-wo-folding(5, 10)* : implémentation de l'algorithme 19 utilisant le programme linéaire en nombres entiers qui minimise le makespan (de la section D.1.1) avec une fenêtre de 5 jobs et une échelle de temps divisée en 10, mais on ne réalise pas de pliage des jobs.

A travers la figure D.1 on remarque que le meilleur algorithme est *ILP-Mean-Stretch-Folding(5, 10)* suivi de *ILP-Folding(5, 10)*. Ils apportent tout de même un gain de 10% par rapport à *backfilling(5)*. Nous remarquons également que *ILP-wo-folding(5, 10)* n'apporte aucune amélioration par rapport au *backfilling*. Cela montre encore une fois l'intérêt du pliage de jobs.

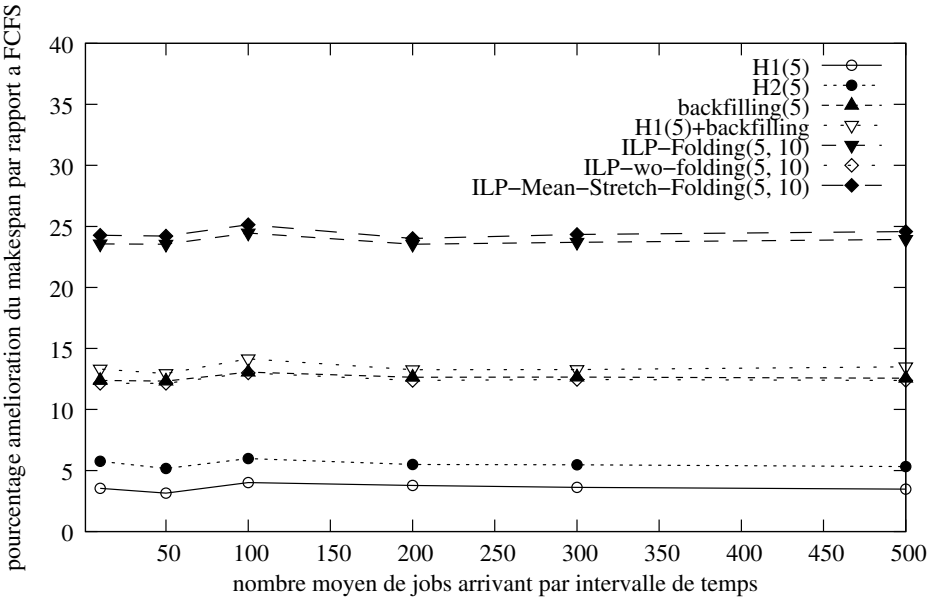


Figure D.1 – Amélioration du makespan par rapport à FCFS

La figure D.2 montre le stretch moyen des ordonnancements de 1000 jobs obtenus avec les différents algorithmes. Le meilleur algorithme reste *ILP-Mean-Stretch-Folding(5, 10)* suivi de *ILP-Folding(5, 10)*. Ces algorithmes sont 10% meilleurs que le *backfilling(5)*.

La figure D.3 montre le max stretch des ordonnancements de 1000 jobs obtenus avec les différents algorithmes. Le meilleur algorithme reste *ILP-Mean-Stretch-Folding(5, 10)* suivi de *ILP-Folding(5, 10)*.

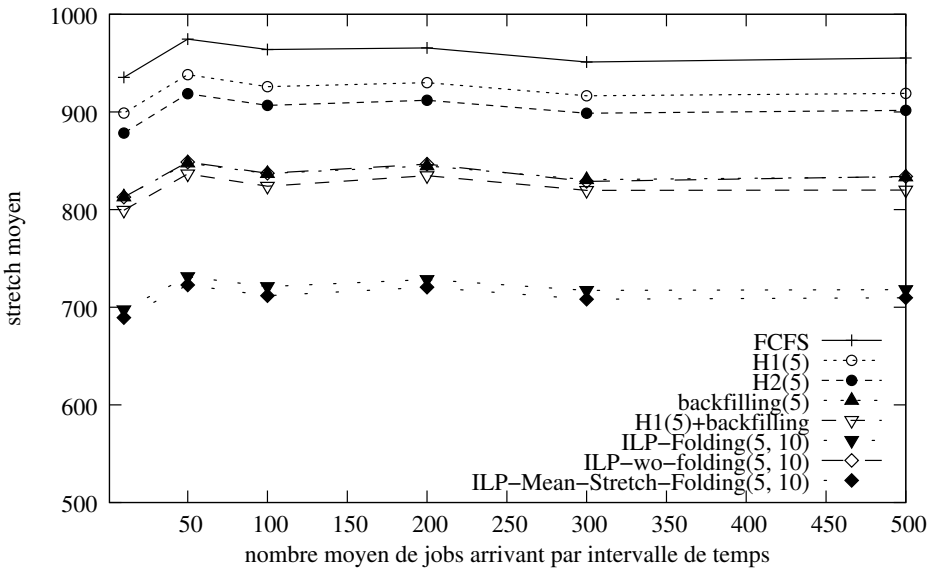


Figure D.2 – Mesure du stretch moyen

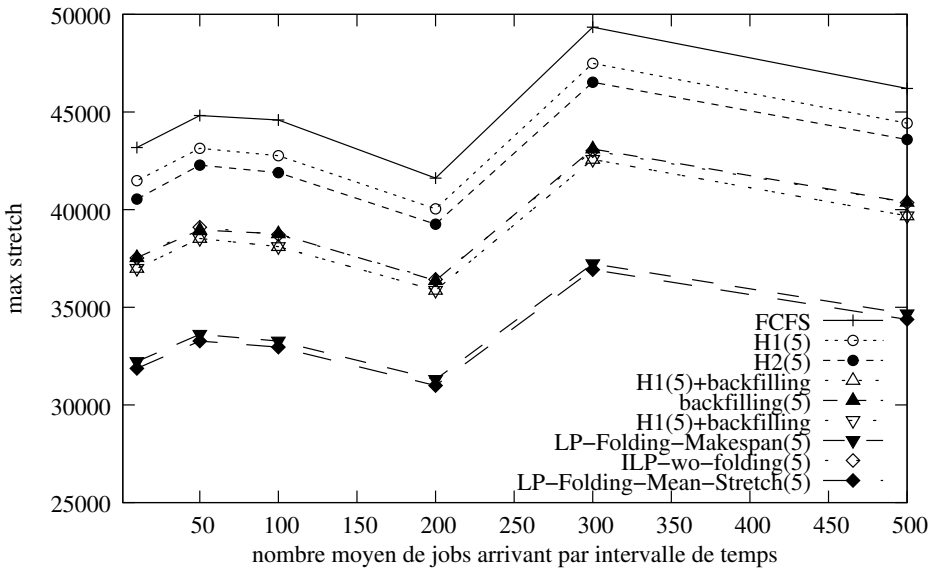


Figure D.3 – Mesure du max-stretch

D.1.6 Résultats expérimentaux (avec les chaînes de Markov)

Nous rajoutons parmi les courbes :

- *LP(30, 10) + Markov* : implémentation de l'algorithme 19 utilisant la relaxation du programme linéaire en nombres entiers qui minimise le makespan (de la section D.1.1) + une chaîne de Markov (algorithme 1) avec une fenêtre de 30 jobs et une échelle de temps divisée en 10 unités de temps ;
- *LP2(30, 10) + Markov* : implémentation de l'algorithme 19 utilisant la relaxation du programme linéaire en nombres entiers qui minimise le makespan (de la section D.1.1) + une chaîne de Markov (algorithme 1) avec une fenêtre de 30 jobs et une échelle de temps divisée en 10 unités de temps. La différence avec l'implémentation précédente est qu'elle utilise le pliage non entier.

a Mise en place des expériences

Dans ces expériences nous utilisons un cluster de 1024 processeurs. Il y a 1 000 jobs à ordonnancer. Les paramètres de la génération des workloads synthétiques sont les suivants :

- $timeInterval = 3600\,s$;
- $rtMin = 1000\,s$: le temps minimal requis par un job ;
- $rtMax = 200\,000\,s$: le temps maximal requis par un job ;
- $diffMax = 40$: le pourcentage maximum entre le temps requis par un job et son temps d'exécution réelle ;
- $procMin = 1$: le nombre minimum de processeurs requis par un job ;
- $procMax = 1024$: le nombre maximum de processeurs requis par un job ;
- $\lambda \in \{10, 25, 50, 75, 100, 150, 200, 250, 300, 500, 1000\}$: le nombre moyen de jobs arrivant par intervalle de temps $timeInterval = 3600\,s$.

Pour chaque valeur λ, 20 workloads synthétiques sont générés. Alors pour chaque valeur de λ, la mesure est calculée par une moyenne arithmétique de 20 valeurs.

b Résultats de simulations

La figure D.4 montre les résultats des expériences sur les performances des différents algorithmes quant au makespan selon différentes valeurs de λ.

Elle montre de combien chaque algorithme réduit le makespan par rapport à *FCFS*. D'abord nous pouvons dire que les meilleures algorithmes sont *ILP-Folding-Mean-Stretch(5)* et *ILP-Folding(5)*, l'amélioration du makespan atteint presque 25% par rapport à *FCFS* alors que l'algorithme classique *backfilling(5)* obtient une amélioration de seulement 13%. Ainsi nous remarquons que *LP-wo-folding(5)* et *backfilling(5)* ont presque les mêmes performances : *LP-wo-folding(5)* est pratiquement 1% au dessus *backfilling(5)*. C'est une observation importante

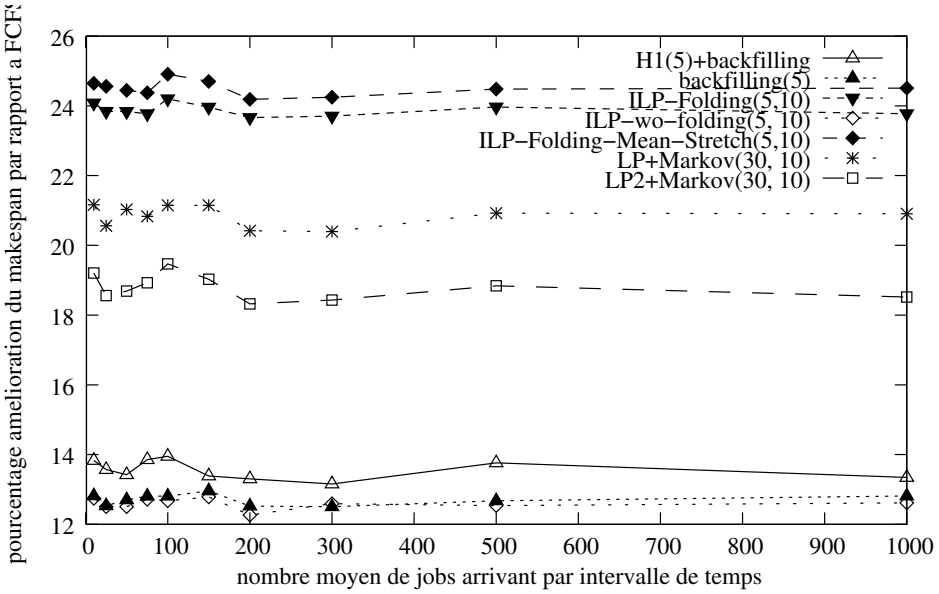


Figure D.4 – Amélioration du makespan par rapport à *FCFS* en fonction de λ

car elle met en avant l'intérêt du pliage de jobs. Nous remarquons que l'amélioration en terme de makespan de chaque algorithme ne dépend pas de λ. Nous remarquons également que *ILP-Folding-Mean-Stretch(5)* donne des résultats un peu meilleur que *ILP-Folding(5)* bien que l'algorithme *ILP-Folding-Mean-Stretch(5)* essaie d'optimiser le stretch moyen des jobs de la fenêtre au lieu du makespan.

Nous pouvons dire, qu'en terme de makespan, les chaînes de Markov, n'apporte aucune amélioration, par rapport à l'utilisation seule du programme linéaire en nombres entiers.

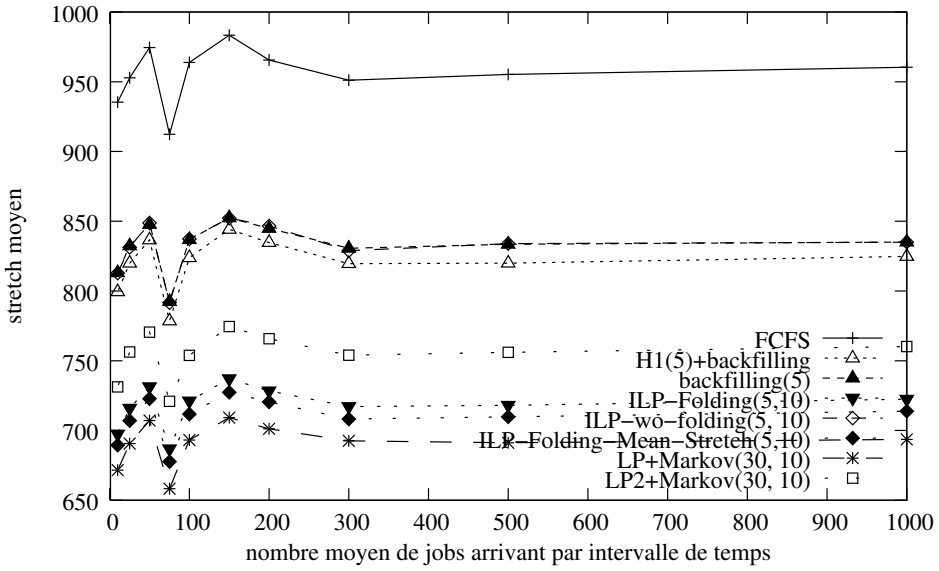


Figure D.5 – Mean stretch des ordonnancements avec différentes heuristiques en fonction de λ

Précédemment nous considérions le makespan. Maintenant nous considérons une métrique centré *utilisateur* : le stretch moyen. La figure D.5 montre le stretch moyen des ordonnancements

Algorithmes	Temps de calcul par job (ms)
FCFS	25
backfilling	28
H1(5)	2520
H2(5)	2342
H1(5)+backfilling	3958
backfilling(5)	50
ILP-Folding(5)	1108
ILP-wo-folding(5)	757
ILP-Folding-Mean-Stretch(5)	1884

Table D.1 – Temps de calcul moyens pour ordonnancer un job (ms) dans un cluster de 1024 processeurs

obtenus avec *FCFS*, *backfilling* et nos algorithmes selon différentes valeurs de λ. Pour cette métrique, le meilleur algorithme est *LP + Markov(30, 10)*. Le meilleur algorithme pour la métrique makespan, *ILP-Folding-Mean-Stretch(5, 10)*, devient deuxième. Les algorithmes les moins bons sont *FCFS* et *backfilling*.

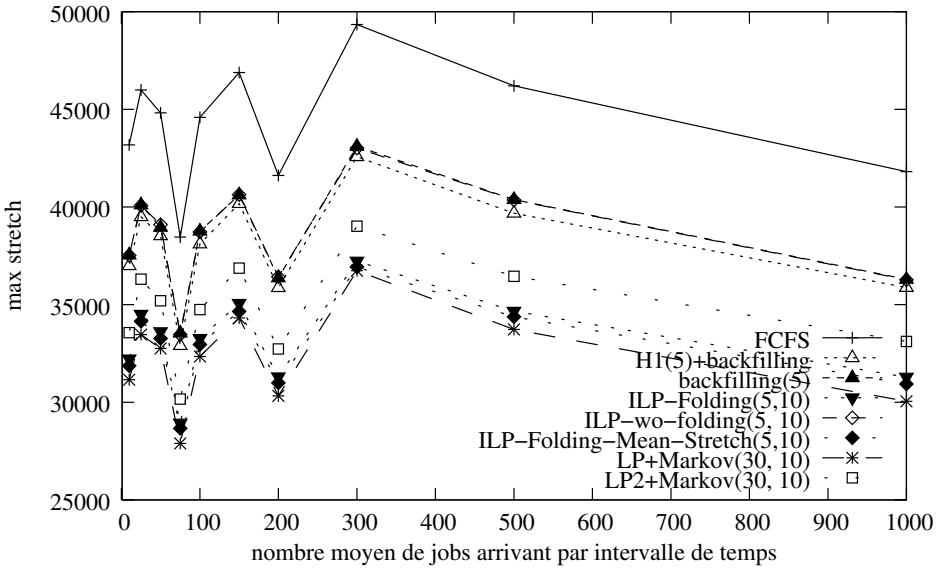


Figure D.6 – Max stretch des ordonnancements avec différentes heuristiques en fonction de λ

La figure D.6 montre le max stretch des ordonnancements obtenus avec *FCFS*, *backfilling* et nos algorithmes selon différentes valeurs de λ. *LP + Markov(30, 10)* reste le meilleur.

c Temps de calcul

Les simulations ont été exécutées au Mésocentre de calcul de Franche-Comté[1], sur des processeurs Xeon avec quatre cœurs cadencés à 2.8 Ghz. La table D.1 montre les différents temps de calcul moyens utilisés pour ordonnancer un job. Nous remarquons qu'il faut seulement une seconde à *ILP-Folding(5)* pour ordonnancer un job, et deux secondes pour *ILP-Folding-Mean-Stretch(5)*.

[1] `http://meso.univ-fcomte.fr/`

Algorithmes	amélioration du makespan (%/*FCFS*)
backfilling(5)	16
ILP-Folding(5)	24
LP-wo-folding(5)	12
ILP-Folding-Mean-Stretch(5)	24

Table D.2 – Amélioration du makespan par rapport à *FCFS* avec un cluster de 10000 processeurs

Algorithmes	stretch moyen
backfilling(5)	788.69707609015
ILP-Folding(5)	710.00962691072
LP-wo-folding(5)	826.35764163091
ILP-Folding-Mean-Stretch(5)	702.99543664146

Table D.3 – Stretch moyen avec un cluster de 10000 processeurs

D.1.7 Autres résultats

Ici, nous passons à l'échelle suivante : nous simulons un cluster de 10000 processeurs afin d'observer le comportement des différents algorithmes avec des programmes linéaires *ILP-Folding(5)*, *LP-wo-folding(5)* ,*ILP-Folding-Mean-Stretch(5)* et le classique algorithm *backfilling(5)*. Nous mettons en place les expériences avec les paramètres suivants :

- $timeInterval = 3600\,s$;
- $rtMin = 1000\,s$: le temps minimal requis par un job ;
- $rtMax = 200\,000\,s$: le temps maximal requis par un job ;
- $diffMax = 40$: le pourcentage maximum entre le temps requis par un job et son temps d'exécution réelle ;
- $procMin = 1$: le nombre minimum de processeurs requis par un job ;
- $procMax = 10000$: le nombre maximum de processeurs requis par un job ;
- $\lambda = 100$: le nombre moyen de jobs soumis par intervalle de temps.

Les paramètres sont pratiquement les mêmes que ceux de la section D.1.5.a, excepté le fait que nous avons uniquement changé le nombre de processeurs du cluster et fixé λ.

Quand nous augmentons la taille du cluster, nous remarquons à travers les tableaux D.2 et D.3 et D.4 que *ILP-Folding(5)* et *ILP-Folding-Mean-Stretch(5)* donnent une amélioration de 24% par rapport à *FCFS*, 700 pour le stretch moyen et 30000 pour le max stretch. Et précédemment dans la section D.1.5, les résultats étaient similaires. Donc nous pouvons conclure qu'augmenter la taille du cluster n'influe pas les performances de nos algorithmes fondés sur des programmes linéaires.

Le tableau D.5 montre le temps moyen nécessaire pour calculer un ordonnancement d'un job avec les différents algorithmes : *ILP-Folding(5)* et *ILP-Folding-Mean-Stretch(5)* prennent environ 4 secondes pour ordonnancer un job alors qu'avec un cluster de 1024 processeurs, il ne lui fallait qu'une seconde. On pouvait s'attendre à ce que le temps CPU soit de 10 secondes, cependant cela peut s'expliquer par le fait que la fonction diviseur $\sigma_0(n)$ d'un nombre entier n

Algorithmes	Max stretch
backfilling(5)	33806
ILP-Folding(5)	30078
ILP-wo-folding(5)	35034
ILP-Folding-Mean-Stretch(5)	29719

Table D.4 – Max stretch avec un cluster de 10000 processeurs

Algorithmes	Temps de calcul pour ordonnancer un job (ms)
backfilling(5)	1744
ILP-Folding(5)	4517
LP-wo-folding(5)	5866
ILP-Folding-Mean-Stretch(5)	4211

Table D.5 – Temps CPU moyen pour ordonnancer un job (ms) avec un cluster de 10000 processeurs

est plus petit que n, dans notre cas, le nombre $conf_i$ de pliages pour le job i qui requiert au départ n processeurs.

D.1.8 Résultats expérimentaux et comparaison avec un EDF

Dans cette section nous comparons nos algorithmes fondés sur des programmes linéaires et chaînes de Markov avec un simple algorithme qui choisit le job qui a le plus petit deadline (en anglais *Earlierst Deadline First* : EDF). Notre algorithme EDF avec pliage est équivalent à l'algorithme 19 sans l'exécution du programme linéaire en passant systématiquement à la ligne 9.

a Mise en place des expériences

Dans ces expériences nous utilisons un cluster de 128 processeurs. Il y a 1 000 jobs à ordonnancer. Les paramètres de la génération des workloads synthétiques sont fondés sur le modèle de Lublin [70].

Une charge l d'un workload synthétique de n jobs pour un cluster de m processeurs est calculée de la manière suivante :

$$l = \frac{\sum_i^n runtime_i * n_i}{m * max_i(a_i)} \tag{D.19}$$

avec

- $runtime_i$ le temps d'exécution du job i ;
- n_i le nombre de processeurs alloués au job i ;
- m le nombre de processeurs dans le cluster ;
- a_i la date d'arrivée du job i.

Pour chaque valeur de charge l, 40 workloads synthétiques sont générés. Alors pour chaque valeur de l, la mesure est calculée par une moyenne arithmétique de 40 valeurs.

b Résultats de simulations

Nous considérons le makespan, le stretch moyen et le max stretch.

La figure D.7 montre le pourcentage d'amélioration du makespan par rapport à *FCFS* apportée par les différents algorithmes :

- $H_1(5)$ + *backfilling* : implémentation de l'algorithme H_1 + *backfilling* (de la section 4.5.4) avec une fenêtre de 5 jobs ;
- *backfilling(5)* : implémentation de l'algorithme *backfilling* avec une fenêtre de 5 jobs ;
- *ILP-Folding(5, 10)* : implémentation de l'algorithme 19 utilisant le programme linéaire en nombres entiers qui minimise le makespan (de la section D.1.1) avec une fenêtre de 5 jobs et une échelle de temps divisée en 10 ;
- *ILP-Mean-Stretch-Folding(5, 10)* : implémentation de l'algorithme 19 utilisant le programme linéaire en nombres entiers qui minimise le stretch moyen (de la section D.1.2) avec une fenêtre de 5 jobs et une échelle de temps divisée en 10 unités de temps ;
- *LP(30, 10) + Markov* : implémentation de l'algorithme 19 utilisant la relaxation du programme linéaire en nombres entiers qui minimise le makespan (de la section D.1.1) + une chaîne de Markov (algorithme 1) avec une fenêtre de 30 jobs et une échelle de temps divisée en 10 unités de temps ;
- *LP2(30, 10) + Markov* : implémentation de l'algorithme 19 utilisant la relaxation du programme linéaire en nombres entiers qui minimise le makespan (de la section D.1.1) + une chaîne de Markov (algorithme 1) avec une fenêtre de 30 jobs et une échelle de temps divisée en 10 unités de temps ; La différence avec l'implémentation précédente est qu'elle utilise le pliage non entier ;
- *LP-Mean-Stretch(30, 10) + Markov* : implémentation de l'algorithme 19 utilisant la relaxation du programme linéaire en nombres entiers qui minimise le stretch moyen (de la section D.1.2) + une chaîne de Markov (algorithme 1) avec une fenêtre de 30 jobs et une échelle de temps divisée en 2 unités de temps ;
- *EDF-Folding(30)* : Earliest Deadline First avec pliage et une fenêtre de 30 jobs.

La figure D.7 montre le pourcentage d'amélioration du makespan par rapport à *FCFS* apportée par les différents algorithmes en fonction de la charge. On remarque que pour toute charge les meilleurs algorithmes sont $H_1(5)$ + *backfilling*, *ILP-Folding(5,10)* et *ILP-Folding-Mean-Stretch(5,10)*. Ils apportent tous les trois un gain de plus de 20% par rapport à *FCFS* pour une charge entre 90% et 100%. L'algorithme *backfilling(5)* apporte le moins de gain. Ensuite viennent l'algorithme *EDF-Folding(30)* qui apporte un gain de 15 à 20% avec une charge de plus de 70%. Par ailleurs pour une charge inférieure à 40% tous les algorithmes apportent le même gain. Concernant les trois implémentations qui associent un programme linéaire et une chaîne de Markov, on s'aperçoit, qu'en fait ceux-ci donnent de moins bons résultats qu'un simple algorithme *EDF-Folding(30)*.

La figure D.8 montre le stretch moyen en fonction de la charge. Le *FCFS* donne des ordonnancements avec le moins bon stretch moyen, puis viennent l'algorithme *backfilling(5)*, puis

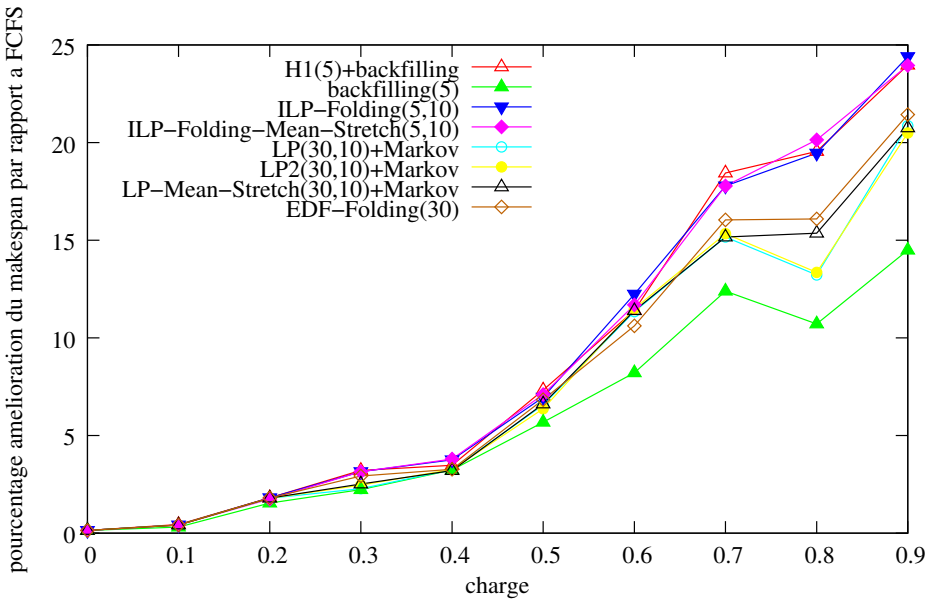


Figure D.7 – Amélioration du makespan par rapport à *FCFS* en fonction de la charge

$H_1(5)$ + *backfilling* et *ILP-Folding(5,10)*. La figure D.9 est le même graphique que D.8 auquel nous avons retiré ces quatre algorithmes. Nous nous apercevons également qu'en prenant le stretch moyen comme métrique, *LP2(30, 10)* + *Markov* donne des résultats presque identiques à ceux de l'algorithme *EDF-Folding(30)*. Pour certaines valeurs de charge par exemple entre 50% et 60% *LP2(30, 10)* + *Markov* est bien meilleur que *EDF-Folding(30)*. Néanmoins dans la plupart des cas *EDF-Folding(30)* reste le meilleur algorithme.

D.1.9 Un workload réel

Le workload a été pris depuis les archives de Dror G. Feitelson à l'adresse web [41]. Nous évaluons les performances de *ILP-Folding(5)* et *ILP-Folding-Mean-Stretch(5)* et nous les comparons avec *FCFS*, *backfilling(5)* et *H1(5)+backfilling*.

Le tableau D.6 montre les résultats concernant le stretch moyen, le max stretch et le makespan après avoir ordonnancé 28 490 jobs du workload du Swedish Royal Institute of Technology (KTH) . Cette machine est constituée de 100 processeurs. Nous remarquons que nos algorithmes n'améliorent pas le makespan du tout par rapport à *FCFS*, mais ils améliorent efficacement l'algorithme *FCFS* de manière drastique du point de vue du stretch moyen et du max stretch. *ILP-Folding(5)* est devant *H1(5)+backfilling* et le gain pour le strech moyen est d'environ 25%. *ILP-Folding-Mean-Stretch(5)* est même meilleur que *ILP-Folding(5)* du point de vue du stretch moyen et du max stretch.

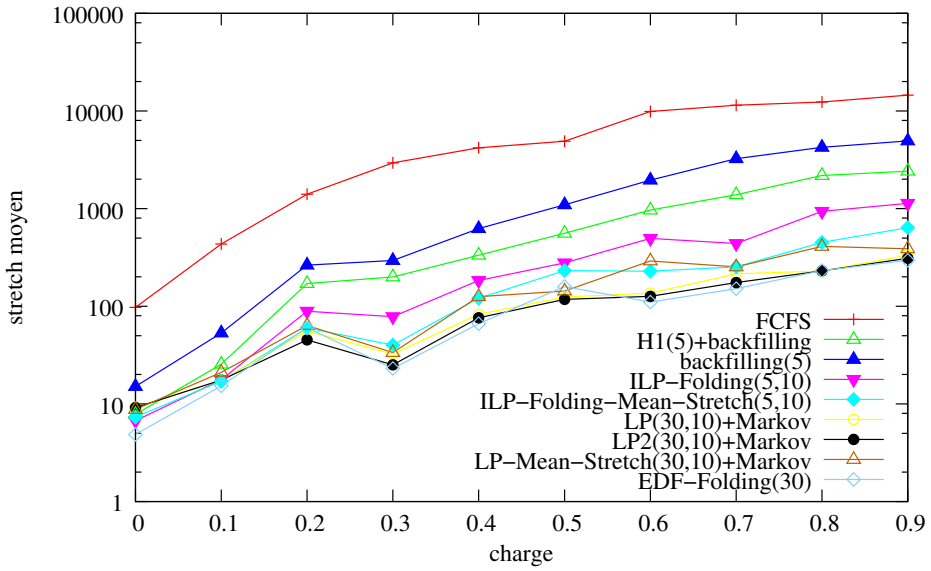


Figure D.8 – Stretch moyen en fonction de la charge

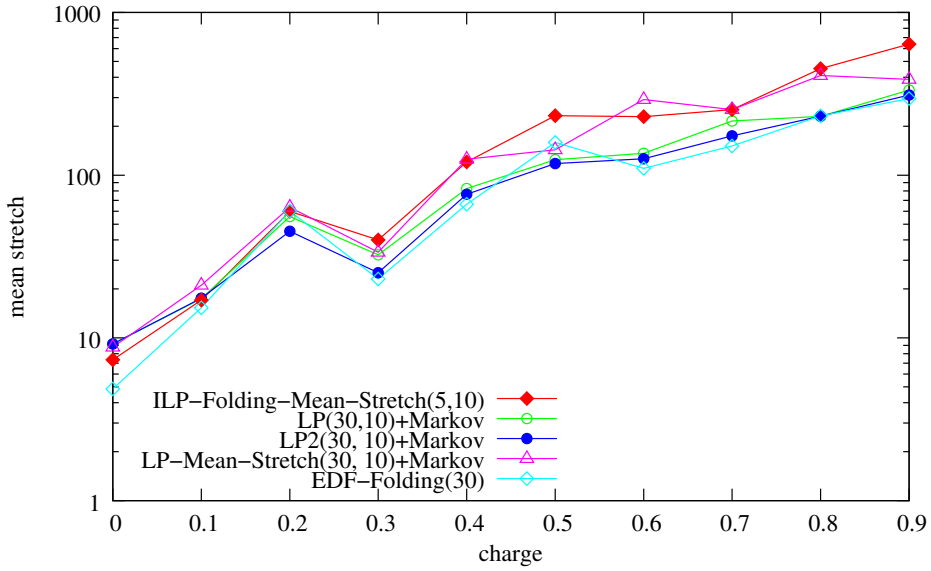


Figure D.9 – Stretch moyen en fonction de la charge (zoom)

Algorithmes	stretch moyen	Max stretch	Makespan
FCFS	22886	1453464	29381423
backfilling(5)	880	351971	29363626
H1(5)+backfilling	327	114423	29363626
ILP-Folding(5)	242	107563	29363626
ILP-Folding-Mean-Stretch(5)	139	51465	29363626
LP(30, 10)+Markov	641	184290	29363626
LP2(30, 10)+Markov	607	183343	29363626
LP2(30, 2)+Markov	86	51322	29371225
EDF(5)-Folding	167	97182	29363626
EDF(30)-Folding	107	119450	29363626

Table D.6 – Résultats avec un workload du KTH (100 processeurs)

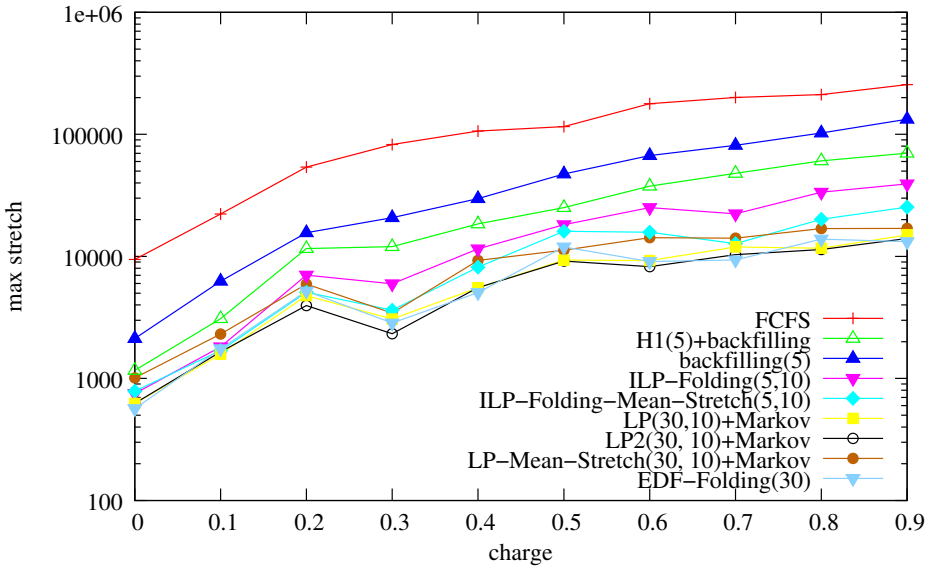


Figure D.10 – Max stretch en fonction de la charge

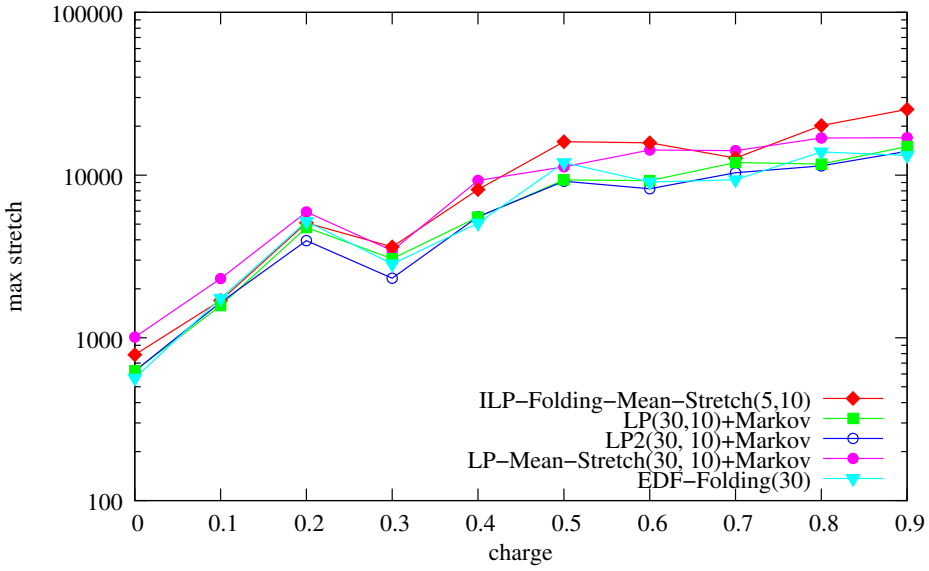


Figure D.11 – Max stretch en fonction de la charge (zoom)

D.2 Synthèse

Nous avons présenté ici un travail laborieux qui n'a pas apporté des résultats à sa hauteur. En effet, un simple algorithme EDF avec pliage fait mieux que nos heuristiques fondées sur la résolution de programmes linéaires et sur des chaînes de Markov.

Néanmoins nous pouvons affirmer grâce aux expériences que la résolution de nos programme linéaires en nombres entiers avec un grand nombres de variables peuvent se substituer à leur relaxation suivie d'une chaîne de Markov et que cette relaxation donne de bonnes performances. Ainsi, l'ordonnancement en terme de stretch moyen est meilleur avec la relaxation d'un programme linéaire en nombres entiers suivie d'une chaîne de Markov qu'avec un programme linéaire en nombres entiers avec moins de variables.

Bibliographie

[1] The community enterprise operating system, `http://www.centos.org` 17, 58

[2] Gurobi, `http://www.gurobi.com` 89

[3] Kernel-based virtual machine, `http://www.linux-kvm.org` 58

[4] Top 500 supercomputing sites, `http://www.top500.org/` 8

[5] van der Aalst, W.M.P. : The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998) 12

[6] Bansal, S., Kumar, P., Singh, K. : An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. IEEE Transactions on Parallel and Distributed Systems 14(6), 533–544 (2003), IEEE Computer Society, Los Alamitos, CA, USA 55

[7] Beaumont, O., Legrand, A., Marchal, L., Robert, Y. : Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms. In : Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks. pp. 296–302. ISPDC '04, Cork, Ireland, IEEE Computer Society, Washington, DC, USA (2004) 21, 24, 25, 28, 33

[8] Beaumont, O., Legrand, A., Marchal, L., Robert, Y. : Steady-state scheduling on heterogeneous clusters : why and how ? In : 6th Workshop on Advances in Parallel and Distributed Computational Models (APDCM 2004). IEEE Computer Society Press, Santa Fe, USA (2004) 26, 115

[9] Bender, M.A., Chakrabarti, S., Muthukrishnan, S. : Flow and stretch metrics for scheduling continuous job streams. In : Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms. pp. 270–279. SODA '98, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1998) 16

[10] Benoit, A., Gallet, M., Gaujal, B., Robert, Y. : Computing the throughput of replicated workflows on heterogeneous platforms. In : Proceedings of the 2009 International Conference on Parallel Processing. pp. 204–211. ICPP '09, IEEE Computer Society, Washington, DC, USA (2009) 24, 25

[11] Benoit, A., Marchal, L., Robert, Y. : Who needs a scheduler ? Tech. Rep. RR-2008-34, Laboratoire de l'Informatique du Parallélisme (LIP) (Oct 2008) 12

[12] Bertsimas, D., Gamarnik, D. : Asymptotically optimal algorithms for job shop scheduling and packet routing. Journal of Algorithms 33(2), 296–318 (1999), Academic Press, Inc., Duluth, MN, USA 28

[13] Bode, B., Halstead, D.M., Kendall, R., Lei, Z., Jackson, D. : The portable batch scheduler and the maui scheduler on linux clusters. In : Proceedings of the 4th annual Linux Showcase & Conference - Volume 4. USENIX Association, Berkeley, CA, USA (2000) 11

[14] Borgwardt, K.H. : The average number of pivot steps required by the simplex-method is polynomial. Mathematical Methods of Operations Research 26, 157–177 (1982), Physica Verlag, An Imprint of Springer-Verlag GmbH 21

[15] Braun, T.D., Siegel, H.J., Beck, N., Boloni, L.L., Maheswaran, M., Reuther, A.I., Robertson, J.P., Theys, M.D., Yao, B. : A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. Journal of Parallel and Distributed Computing (61), 810–837 (2001), Academic Press, Inc. 24

[16] Bregman, L. : The relaxation method of finding the common points of convex sets and its application to the solution of problems in convex optimization. USSR Computational Mathematics and Mathematical Physics 7, 200–217 (1967) 121

[17] Brent, R.P. : The parallel evaluation of general arithmetic expressions. J. ACM 21(2), 201–206 (Apr 1974), ACM, New York, NY, USA 58

[18] Candes, E.J., Romberg, J., Tao, T. : Robust uncertainty principles : exact signal reconstruction from highly incomplete frequency information. IEEE Transactions on Information Theory 52(2), 489–509 (Feb 2006), IEEE 79

[19] Candes, E.J., Wakin, M.B., Boyd, S.P. : Enhancing Sparsity by Reweighted L1 Minimization. Journal of Fourier Analysis and Applications 14(5), 877–905 (Dec 2008) 79

[20] Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounie, G., Neyron, P., Richard, O. : A batch scheduler with high level components. In : Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05). pp. 776–783. CCGRID '05, Cardiff, UK, IEEE Computer Society, Washington, DC, USA (2005) 11

[21] Caron, E., Desprez, F. : Diet : A scalable toolbox to build network enabled servers on the grid. International Journal of High Performance Computing Applications 20(3), 335–352 (2006) 12

[22] Casanova, H., Legrand, A., Marchal, L. : Scheduling distributed applications : the simgrid simulation framework. In : 3rd IEEE International Symposium on Cluster Computing and the Grid. pp. 138–. Tokyo, Japan, IEEE Computer Society, Washington, DC, USA (2003) 38, 49

[23] Casanova, H. : Simgrid : A toolkit for the simulation of application scheduling. In : Proceedings of the 1st International Symposium on Cluster Computing and the Grid. pp. 430–. CCGRID '01, Brisbane, Australia, IEEE Computer Society, Washington, DC, USA (2001) 38, 49, 145

[24] Casanova, H. : Modeling large-scale platforms for the analysis and the simulation of scheduling strategies. In : 18th International Parallel and Distributed Processing Symposium. vol. 8. IEEE Computer Society, Los Alamitos, CA, USA (2004) 33

[25] Casanova, H., Legrand, A., Quinson, M. : Simgrid : A generic framework for large-scale distributed experiments. In : Proceedings of the Tenth International Conference on Computer Modeling and Simulation. pp. 126–131. UKSIM '08, IEEE Computer Society, Washington, DC, USA (2008) 38, 49, 66, 89, 145

[26] Chartrand, R., Yin, W. : Iteratively reweighted algorithms for compressive sensing. In : 33rd International Conference on Acoustics, Speech, and Signal Processing (ICASSP). pp. 3869 – 3872. Las Vegas, USA, Institute of Electrical and Electronics Engineers (IEEE) (2008) 79, 84, 113, 124, 127, 129, 130

[27] Dandamudi, S.P. : Hierarchical Scheduling in Parallel and Cluster Systems. Kluwer Academic Publishers, Norwell, MA, USA (2003) 23

[28] Daoud, M., Kharma, N. : Gats 1.0 : A novel ga-based scheduling algorithm for task scheduling on heterogeneous processor nets. In : Proceedings of the 2005 conference on Genetic and evolutionary computation. GECCO '05, Washington, DC, USA, ACM, New York, NY, USA (2005) 25, 26, 35, 47, 115

[29] Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M.H., Vahi, K., Livny, M. : Pegasus : Mapping Scientific Workflows onto the Grid. In : Dikaiakos, M. (ed.) Grid Computing, Lecture Notes in Computer Science, vol. 3165, chap. 2, pp. 131–140. Springer Berlin / Heidelberg, Berlin, Heidelberg (2004) 12

[30] Deelman, E., Callaghan, S., Field, E., Francoeur, H., Graves, R., Gupta, N., Gupta, V., Jordan, T.H., Kesselman, C., Maechling, P., Mehringer, J., Mehta, G., Okaya, D., Vahi, K., Zhao, L. : Managing large-scale workflow execution from resource provisioning to provenance tracking : The cybershake example. In : E-SCIENCE '06 : Proceedings of the Second IEEE International Conference on e-Science and Grid Computing. p. 14. IEEE Computer Society, Washington, DC, USA (2006) 9

[31] Diakité, S., Marchal, L., Nicod, J.M., Philippe, L. : Steady-state for batches of identical task trees. In : Proceedings of the 15th International Euro-Par Conference on Parallel Processing. pp. 203–215. Euro-Par '09, Delft, The Netherlands, Springer-Verlag, Berlin, Heidelberg (2009) 28

[32] Diakité, S., Nicod, J.M., Philippe, L., Toch, L. : Assessing new approaches to schedule a batch of identical intree-shaped workflows on a heterogeneous platform. International Journal of Parallel, Emergent and Distributed Systems 27(1), 79–107 (2011), Taylor & Francis 25

[33] Diakité, S. : Contribution à l'ordonnancement de lots de graphes de tâches sur une plateforme hétérogène. Ph.D. thesis, Université de Franche-Comté (2010) 31

[34] Donoho, D.L. : Compressed sensing. IEEE Transactions on Information Theory 52, 1289–1306 (2006) 79

[35] Downey, A.B. : A parallel workload model and its implications for processor allocation. In : Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing. pp. 112–. HPDC '97, Portland, USA, IEEE Computer Society, Washington, DC, USA (1997) 67

[36] Du, J., Leung, J.Y.T. : Complexity of scheduling parallel task systems. SIAM J. Discret. Math. 2(4), 473–487 (Nov 1989), Society for Industrial and Applied Mathematics 62

[37] Dutot, P.F., Eyraud, L., Mounié, G., Trystram, D. : Bi-criteria algorithm for scheduling jobs on cluster platforms. In : Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. pp. 125–132. SPAA '04, Barcelona, Spain, ACM, New York, NY, USA (2004) 15, 23

[38] Dutot, P.F., Netto, M.A.S., Goldman, A., Kon, F. : Scheduling moldable bsp tasks. In : Proceedings of the 11th international conference on Job Scheduling Strategies for Parallel Processing. pp. 157–172. JSSPP'05, Cambridge, MA, USA, Springer-Verlag, Berlin, Heidelberg (2005) 24, 62

[39] Dutot, P.F., Trystram, D. : Scheduling on hierarchical clusters using malleable tasks. In : Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures. pp. 199–208. SPAA '01, Crete Island, Greece, ACM, New York, NY, USA (2001) 11

[40] Fallenbeck, N., Picht, H.J., Smith, M., Freisleben, B. : Xen and the art of cluster scheduling. In : Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing. pp. 4–. VTDC '06, Tampa, Florida, USA, IEEE Computer Society, Washington, DC, USA (2006) 17

[41] Feitelson, D.G. : Parallel workloads archive, `http://www.cs.huji.ac.il/labs/parallel/workload/` 66, 70, 154

[42] Feitelson, D.G. : Workload characterization and modeling book, `http://www.cs.huji.ac.il/~feit/wlmod/wlmod.pdf` 67

[43] Feitelson, D.G. : A Survey of Scheduling in Multiprogrammed Parallel Systems. International Business Machines Corporation (1994) 15, 16, 23, 57, 124

[44] Feitelson, D.G., Mu'alem, A.W. : On the definition of "on-line" in job scheduling problems. Tech. rep., SIGACT News (2000) 15

[45] Fidanova, S. : Simulated annealing for grid scheduling problem. In : Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing. pp. 41–45. JVA '06, Sofia, Bulgaria, IEEE Computer Society, Washington, DC, USA (2006) 18

[46] Fidanova, S., Durchova, M. : Ant algorithm for grid scheduling problem. In : Proceedings of the 5th international conference on Large-Scale Scientific Computing. pp. 405–412. LSSC'05, Sozopol, Bulgaria, Springer-Verlag, Berlin, Heidelberg (2006) 18

[47] Floudas, C.A., Lin, X. : Mixed integer linear programming in process scheduling : Modeling, algorithms, and applications. Annals of Operations Research 139(1), 131–162 (2005) 21

[48] Foster, I., Kesselman, C. : Globus : A metacomputing infrastructure toolkit. International Journal of Supercomputer Applications 11, 115–128 (1996) 2

[49] Garcia-Lorenzo, D., Prima, S., Parkes, L., Ferré, J.C., Morrissey, S., Barillot, C. : The impact of processing workflow in performance of automatic white matter lesion segmentation in multiple sclerosis. In : MICCAI workshop on "Medical Image Analysis on Multiple Sclerosis (MIAMS)". New York, USA (2008) 9

[50] Garey, M.R., Johnson, D.S. : Computers and Intractability ; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990) 15, 23, 62

[51] Goh, C., Teoh, E., Tan, K. : A hybrid evolutionary approach for heterogeneous multiprocessor scheduling. Soft Computing - A Fusion of Foundations, Methodologies and Applications 13, 833–846 (2009), Springer Berlin / Heidelberg 26

[52] Goldstein, T., Osher, S. : The split bregman method for l1-regularized problems. SIAM Journal on Imaging Sciences 2, 323–343 (April 2009), Society for Industrial and Applied Mathematics, Philadelphia, PA, USA 113, 121

[53] Graham, R., al. : Optimization and approximation in deterministic sequencing and scheduling : a survey. Annals of Discrete Mathematics pp. 287–326 (1979), Elsevier 34, 78

[54] Gupta, M.R., Chen, Y. : Theory and use of the em algorithm. Foundations and Trends in Signal Processing 4, 223–296 (March 2011), Now Publishers Inc., Hanover, MA, USA 133

[55] Henderson, R. : Job scheduling under the portable batch system. In : Feitelson, D., Rudolph, L. (eds.) Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 949, pp. 279–294. Springer Berlin / Heidelberg (1995) 11

[56] Hochbaum, D.S., Shmoys, D.B. : Using dual approximation algorithms for scheduling problems theoretical and practical results. Journal of the ACM 34(1), 144–162 (Jan 1987), ACM, New York, NY, USA 87

[57] Holland, J.H. : Adaptation in natural and artificial systems. MIT Press, Cambridge, MA, USA (1992) 18

[58] Hui, C.W., Gupta, A. : A bi-index continuous-time mixed-integer linear programming model for single-stage batch scheduling with parallel units. Industrial and Engineering Chemistry Research 40(25), 5960–5967 (2001) 21

[59] Iverson, M., Ozguner, F. : Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In : Proceedings of the Seventh Heterogeneous Computing Workshop. pp. 70–. HCW '98, Orlando, Florida, USA, IEEE Computer Society, Washington, DC, USA (1998) 24

[60] Jackson, D.B., Snell, Q., Clement, M.J. : Core algorithms of the maui scheduler. In : Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing. pp. 87–102. JSSPP '01, Cambridge, MA, USA, Springer-Verlag, London, UK (2001) 11

[61] Jansen, K., Porkolab, L. : Linear-time approximation schemes for scheduling malleable parallel tasks. In : Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms. pp. 490–498. SODA '99, Baltimore, Maryland, United States, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1999) 77

[62] Kannan, S., Roberts, M., Mayes, P., Brelsford, D., Skovira, J.F. : Workload management with loadleveler (2001) 11

[63] Kumar, V.S., Rutt, B., Kurc, T., Catalyurek, U., Saltz, J., Chow, S., Lamont, S., Martone, M. : Large image correction and warping in a cluster environment. In : SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing. p. 79. Tampa, Florida, USA, ACM, New York, NY, USA (2006) 9

[64] Kwok, Y.K., Ahmad, I. : Static Scheduling Algorithms for Allocating Task Graphs to Multiprocessors. ACM Computing Surveys 31(4), 406–471 (jan 1999), ACM Press, New York, NY, USA 24

[65] Kwok, Y.K., Ahmad, I. : Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. IEEE Transactions on Parallel and Distributed Systems 7(5), 506–521 (May 1996), IEEE Press, Piscataway, NJ, USA 24

[66] Lee, K., Paton, N.W., Sakellariou, R., Deelman, E., Fernandes, A.A., Mehta, G. : Adaptive workflow processing and execution in pegasus. In : International Conference on Grid and Pervasive Computing. pp. 99–106. Kunming, China, IEEE Computer Society, Los Alamitos, CA, USA (2008) 12

[67] Lenstra, J., Kan, A.R. : Complexity of scheduling under precedence constraints. Operations Research 26(1), 22–35 (Jan-Feb 1978), INFORMS 24, 31

[68] Litzkow, M.J., Livny, M., Mutka, M.W. : Condor - a hunter of idle workstations. In : Proceedings of the 8th International Conference of Distributed Computing Systems. pp. 104–111. San Jose, CA, USA, IEEE-CS Press (1988) 12

[69] Lodi, A., Martello, S., Monaci, M. : Two-dimensional packing problems : A survey. European Journal of Operational Research pp. 241–252 (2002) 77

[70] Lublin, U., Feitelson, D.G. : The workload on parallel supercomputers : Modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing 63(11), 2003 (2001), Academic Press, Inc., Orlando, FL, USA 11, 67, 152

[71] M. Grinstead, C., Snell, J.L. : Introduction to probability. American Mathematical Society (1997) 19

[72] Mandal, A., Kennedy, K., Koelbel, C., Marin, G., Mellor-Crummey, J., Liu, B., Johnsson, L. : Scheduling strategies for mapping application workflows onto the grid. In : Proceedings. 14th IEEE International Symposium on High Performance Distributed Computing, 2005. pp. 125–134. HPDC '05, Triangle Park, North Carolina, USA, IEEE Computer Society, Washington, DC, USA (jul 2005) 24

[73] Mandal, A., Kennedy, K., Koelbel, C., Marin, G., Mellor-Crummey, J., Liu, B., Johnsson, L. : Scheduling strategies for mapping application workflows onto the grid. In : Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium. pp. 125–134. HPDC '05, IEEE Computer Society, Washington, DC, USA (2005) 12

[74] Mau, B., A, L.M., Larget, B. : Bayesian phylogenetic inference via markov chain monte carlo methods. Biometrics 55, 1–12 (1999) 20

[75] Mounie, G., Rapine, C., Trystram, D. : Efficient approximation algorithms for scheduling malleable tasks. In : Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures. pp. 23–32. SPAA '99, Saint-Malo, France, ACM, New York, NY, USA (1999) 11, 60

[76] Parsons, E.W., Sevcik, K.C. : Implementing multiprocessor scheduling disciplines. In : Proceedings of the Job Scheduling Strategies for Parallel Processing. pp. 166–192. IPPS '97, Geneva, Switzerland, Springer-Verlag, London, UK (1997) 17

[77] Peng, L., Candan, K.S., Mayer, C., Chatha, K.S., Ryu, K.D. : Optimization of media processing workflows with adaptive operator behaviors. Multimedia Tools and Applications 33(3), 245–272 (jun 2007), Kluwer Academic Publishers, Hingham, MA, USA 9

[78] Schaaff, A., Petit, F., Prugniel, P.and Slezak, E., Surace, C. : Workflow in astronomy : the vo france workflow working group experience. In : Astronomical Data Analysis Software and Systems XVII. Kensington Town Hall, London, United Kingdom, Astronomical Society of the Pacific (2007) 9

[79] Sgall, J. : On-line scheduling. In : Developments from a June 1996 seminar on Online algorithms : the state of the art. pp. 196–231. Springer-Verlag, London, UK, UK (1998) 15

[80] Skovira, J., Chan, W., Zhou, H., Lifka, D.A. : The easy - loadleveler api project. In : Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing. pp. 41–47. IPPS '96, Honolulu, Hawaii, USA, Springer-Verlag, London, UK (1996) 23

[81] Skutella, M. : Convex quadratic and semidefinite programming relaxations in scheduling. Journal of the ACM 48(2), 206–242 (2001), ACM, New York, USA 22

[82] Sotomayor, B., Keahey, K., Foster, I. : Combining batch execution and leasing using virtual machines. In : Proceedings of the 17th international symposium on High performance distributed computing. pp. 87–96. HPDC '08, Boston, MA, USA, ACM, New York, NY, USA (2008) 17

[83] Stillwell, M., Vivien, F., Casanova, H. : Dynamic fractional resource scheduling for hpc workloads. In : In Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10). pp. 1–12. Atlanta, Georgia, USA, IEEE (2010) 17

[84] Suchard, M.A., Wang, Q., Chan, C., Frelinger, J., Cron, A., West, M. : Understanding gpu programming for statistical computation : Studies in massively parallel massive mixtures. Journal of computational and graphical statistics : a joint publication of American

Statistical Association, Institute of Mathematical Statistics, Interface Foundation of North America 19(2), 419–438 (Jun 2010) 2

[85] Tanaka, Y., Takemiya, H., Nakada, H., Sekiguchi, S. : Design, implementation and performance evaluation of gridrpc programming middleware for a large-scale computational grid. In : GRID '04 : Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing. pp. 298–305. Pittsburgh, PA, USA, IEEE Computer Society, Washington, DC, USA (2004) 12

[86] Taylor, I., Deelman, E., Gannon, D., Shields, M. : Workflows for e-Science. Springer Verlag (2007) 9, 12

[87] Thain, D., Tannenbaum, T., Livny, M. : Distributed computing in practice : the condor experience : Research articles. Concurrency Computation : Practice and Experience 17(2-4), 323–356 (Feb 2005), John Wiley and Sons Ltd., Chichester, UK 12

[88] Topcuouglu, H., Hariri, S., Wu, M.y. : Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (Mar 2002), IEEE Press, Piscataway, NJ, USA 24, 25, 115

[89] Turek, J., Wolf, J.L., Yu, P.S. : Approximate algorithms scheduling parallelizable tasks. In : Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures. pp. 323–332. SPAA '92, San Diego, California, USA, ACM, New York, NY, USA (1992) 11, 60

[90] Vydyanathan, N., Catalyurek, U.V., Kurc, T.M., Sadayappan1, P., Saltz, J.H. : Toward optimizing latency under throughput constraints for application workflows on clusters. In : Heidelberg, S.B.. (ed.) Euro-Par 2007 Parallel Processing. Lecture Notes in Computer Science, vol. 4641/2007, pp. 173–183. Rennes, France (2007) 24, 25

[91] Wieczorek, M., Prodan, R., Fahringer, T. : Scheduling of scientific workflows in the askalon grid environment. ACM SIGMOD Record Journal 34(3), 56–62 (2005), ACM, New York, USA 12

[92] Ye, D., Zhang, G. : On-line scheduling of parallel jobs in a list. J. of Scheduling 10(6), 407–413 (Dec 2007), Kluwer Academic Publishers, Hingham, MA, USA 15

SPIM

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex

■ tél. +33 (0)3 81 66 66 02 ■ **ed-spim@univ-fcomte.fr** ■ www.ed-spim.univ-fcomte.fr

UFC
UNIVERSITÉ
DE FRANCHE-COMTÉ