Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Ordonnancement de tâches parallèles sur plates-formes hétérogènes partagées

THÈSE

présentée et soutenue publiquement le 22 janvier 2009

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Tchimou N'takpé

Composition du jury

Président :	Denis Trystram	Professeur INPG
Rapporteurs :	Olivier Beaumont	Directeur de recherche INRIA
	Jean-Louis Pazat	Professeur INSA
Examinateurs :	Marie-Claude Portmann	Professeur INPL
	Thomas Rauber	Professeur Université de Bayreuth
	André Schaff	Professeur UHP Nancy 1
Directeurs de thèse :	Jens Gustedt	Directeur de recherche INRIA
	Frédéric Suter	Chargé de recherche CNRS

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

Tout d'abord, j'adresse toute ma gratitude à Frédéric Suter et à Jens Gusted d'avoir accepté de diriger cette thèse, et d'avoir été toujours disponibles tout au long de ces travaux.

Je remercie également, pour l'aide qu'elle nous a apportée, toute l'équipe de maintenance de la bibliothèque logicielle *SimGrid* utilisée pour la mise en œuvre de notre simulateur. Il s'agit notamment d'Arnaud Legrand qui a intégré un modèle de tâches parallèles dans *SimGrid* à notre demande, de Martin Quinson et Malek Cherier que j'ai souvent sollicités pour mettre à disposition une version récente de cette bibliothèque.

Un grand merci aussi à Henri Casanova et à Pierre-François Dutot avec qui nous avons collaborer durant cette thèse.

Je voudrais également remercier les membres de mon jury qui ont été tous présents lors de la soutenance et qui sont venus de loin pour la plupart d'entre eux. Merci à Denis Trystram, pour l'honneur qu'il m'a fait d'être président de ce jury. Merci à Olivier Beaumont et Jean-Louis Pazat d'avoir accepté d'être les rapporteurs de cette thèse et, pour leurs critiques pertinentes. Merci aux examinateurs Marie-Claude Portmann, Thomas Rauber et André Schaff.

Je ne saurais oublier tous les autres membres de l'équipe AlGorille avec qui j'ai eu d'excellents rapports, en particulier Emmanuel Jeannot, Xavier Delaruelle, Sylvain Contassot-Vivier, Stéphane Genaud, Louis-Claude Canon et Pierre-Nicolas Clauss.

À mes parents, Tchimou Pascal N'TAKPÉ et Kousso Pauline ADI
et à la mémoire de ma grand mère, Epi DOFFOU

Table des matières

Table des figures

Chapitre 1

Introduction

Aujourd'hui le calcul parallèle permet de résoudre des applications issues de divers domaines scientifiques en des temps relativement courts. Cela consiste à traiter simultanément différentes parties d'une même application sur plusieurs microprocesseurs ou unités de calcul. Il s'applique dans divers domaines parmi lesquels nous pouvons citer les simulations nucléaires, les prévisions météorologiques, la recherche de nouveaux médicaments dans l'industrie pharmaceutique, les calculs financiers et le commerce électronique.

Les applications parallèles exploitent deux types de parallélisme : le parallélisme de tâches et le parallélisme de données. Une application utilisant le paradigme du parallélisme de tâches se subdivise en plusieurs tâches qui peuvent éventuellement être dépendantes entre elles (dépendances temporelle ou dépendance de données). Les applications pouvant utiliser le parallélisme de données exhibent le parallélisme au niveau des boucles, c'est-à-dire des itérations qui peuvent être perçues comme des instructions de type SIMD (*Single Instruction Multiple Data*). Elles appliquent une même opération à des ensembles de données différents. Une méthode qui permet d'augmenter le degré de parallélisme de certaines applications et de passer ainsi à l'échelle et d'améliorer leurs temps d'exécution, est d'exploiter simultanément les parallélismes de données et de tâches. Les applications parallèles mixtes, conçues pour exploiter les deux types de parallélisme, peuvent être modélisées par des graphes acycliques orientés ou DAGs (*Directed Acyclic Graphs*) où chaque nœud représente une tâche parallèle, c'est-à-dire une tâche pouvant s'exécuter sur plusieurs processeurs, et chaque arc définit une relation de précédence (dépendance de flots ou de données) entre deux tâches.

Le calcul parallèle peut s'effectuer sur diverses plates-formes telles que des supercalculateurs, des grappes d'ordinateurs standards reliés entre eux au sein d'un réseau local ou sur des grilles de calcul qui sont des agrégations hétérogènes de nœuds de calcul (ordinateurs standards et/ou supercalculateurs) répartis sur des sites géographiquement distants et interconnectés via un réseau longue distance. Grâce au développement des intergiciels et des technologies des réseaux informatiques, le déploiement des grilles de calcul s'est accéléré au cours de la dernière décennie. L'objectif de la mise en œuvre de ces plates-formes est de fournir une infrastructure matérielle et logicielle qui permet le partage coordonné de ressources (calcul, stockage réseau) par un ensemble d'individus et d'institutions, dans un environnement flexible et sécurisé. En outre, ces agrégations de nœuds de calcul permettent d'accéder à moindre coût à de nombreuses ressources afin de résoudre des problèmes complexes.

L'équipe-projet INRIA « Algorithmes pour La Grille » (AlGorille) au sein de laquelle s'est déroulée cette thèse, s'intéresse aux aspects algorithmiques du calcul sur les grilles. En particulier, elle traite des problèmes liés à l'organisation efficace d'applications pouvant s'exécuter sur

les grilles de calcul au niveau du fournisseur de service d'une part et d'autre part, au niveau de l'application. Les principaux thèmes de recherche d'AlGorille sont au nombre de trois : la structuration des applications pour le passage à l'échelle, la gestion transparente des ressources et la validation expérimentale.

Un des problèmes fondamentaux dans le domaine du calcul parallèle, faisant partie de la gestion transparente des ressources, est l'ordonnancement dont l'objectif est de gérer les ressources de manière efficace. L'ordonnancement consiste à déterminer l'ordre dans lequel les différentes parties d'une ou plusieurs applications parallèles doivent s'exécuter et les ressources que chacune des parties doit utiliser. La conception d'un algorithme d'ordonnancement dépend aussi bien de l'architecture cible que de la structure de l'application. La plupart des problèmes d'ordonnancement d'applications parallèles sont NP-difficiles [33, 43, 52]. Diverses études ont donc été réalisées dans le domaine de l'ordonnancement pour proposer des algorithmes approchés et des heuristiques.

Le problème de l'ordonnancement des applications parallèles mixtes a été largement étudié dans le cadre des plates-formes homogènes. Mais ce problème reste ouvert en ce qui concerne les plates-formes hétérogènes et plus particulièrement, les agrégations hétérogènes de grappes de calcul homogènes. En outre, il n'existe aucun algorithme pour l'ordonnancement concurrent de DAGs de tâches parallèles sur les plates-formes hétérogènes. Or les grilles de calcul sont intrinsèquement partagées par de nombreux utilisateurs. Sur de telles plates-formes, il est nécessaire de tenir explicitement compte du caractère partagé afin de mieux gérer l'utilisation des ressources et de satisfaire globalement toutes les applications en compétition pour accéder aux ressources.

L'objectif de cette thèse est l'étude et la conception d'algorithmes d'ordonnancement d'applications constituées de tâches parallèles interdépendantes en tenant explicitement compte du caractère partagé des grilles de calcul. Nous adopterons une approche qui consistera à nous inspirer des algorithmes existants dans la littérature pour proposer des algorithmes implantables dans des environnements réels.

Cette thèse comporte quatre parties. Nous avons procédé de manière incrémentale en adaptant le principe d'un même algorithme d'ordonnancement à de nouvelles contraintes jusqu'à obtenir des algorithmes d'ordonnancement d'applications parallèles mixtes sur plates-formes hétérogènes partagées. La suite de cette thèse est organisée comme suit :

Plates-formes, applications parallèles et ordonnancement

Dans le chapitre 2, nous étudierons les plates-formes de calcul parallèle et les applications parallèles et nous ferons un état de l'art des algorithmes d'ordonnancement existants à ce jour. Ce chapitre permettra de présenter les choix opérés pour les plates-formes et les applications qui seront étudiées par la suite.

Les grappes homogènes constituant une solution bon marché de mise en œuvre d'une plate-forme parallèle, des grilles peu hétérogènes se répandent de plus en plus en agrégeant des grappes homogènes réparties soit au sein d'une seule institution, soit entre plusieurs institutions. C'est le cas de la grille expérimentale Grid'5000 [25, 55] répartie sur le territoire français et destinée aux expérimentations des chercheurs en informatique. Ces plates-formes sont devenues très attrayantes car elles peuvent permettre de déployer des applications parallèles à des échelles sans précédents. Nous choisirons donc de les étudier dans cette thèse.

En ce qui concerne les applications étudiées, nous choisirons d'ordonnancer des applications pour lesquelles les tâches sont modelables. Une tâche modelable est une tâche parallèle pouvant s'exécuter sur différents nombre de processeurs. Ce nombre de processeurs doit être fixé avant l'exécution de la tâche considérée et il ne peut être modifié par la suite.

Ordonnancement de graphes de tâches modelables sur grappes homogènes

De nombreux algorithmes ont été proposés pour l'ordonnancement de DAGs de tâches modelables dans le cas des plates-formes homogènes. Nous commencerons donc, dans le chapitre 3, par les étudier afin de déterminer les meilleurs algorithmes existants et voir s'ils peuvent être améliorés.

L'ordonnancement de DAGs de tâches modelables consiste dans un premier temps, lors de la *phase d'allocation*, à allouer à chaque tâche le nombre de processeurs qu'elle utilisera au cours de son exécution. Ensuite, lors de la *phase de placement*, il faut déterminer les processeurs sur lesquels chaque tâche doit être exécutée et l'ordre dans lequel les tâches débuteront leur exécution dans le respect des contraintes de précédence. Les algorithmes d'ordonnancement de tâches modelables peuvent être répartis en deux catégories. La première est constituée d'algorithmes où la phase d'allocation et la phase de placement sont indissociables. Ce sont les algorithmes d'*ordonnancement en une étape*. La seconde catégorie regroupe les algorithmes d'*ordonnancement en deux étapes* où l'allocation et le placement sont séparés en deux procédures distinctes.

Nous sélectionnerons les heuristiques les plus performantes de chaque catégorie que nous comparerons grâce à de nombreuses simulations. Nous verrons que l'une des heuristiques en deux étapes n'est pas adaptée aux plates-formes comprenant de nombreux processeurs. Nous proposerons alors des améliorations à cette heuristique et évaluerons les nouvelles heuristiques obtenues.

Ordonnancement de graphes de tâches modelables sur plates-formes multi-grappes

Comme nous l'avons mentionné précédemment, les plates-formes hétérogènes les plus répandues sont composées de plusieurs grappes homogènes interconnectées. Cela pose de nouveaux problèmes qui n'ont pas été abordés jusqu'à présent, que l'agrégation soit homogène ou non, d'ailleurs et aussi bien du point de vue théorique que pragmatique. Le chapitre 4 propose donc des algorithmes d'ordonnancement de DAGs de tâches modelables sur les agrégations de grappes homogènes.

Avant de proposer des heuristiques pour des plates-formes multi-grappes très hétérogènes, nous allons tout d'abord proposer et comparer expérimentalement deux algorithmes pour l'ordonnancement de DAGs de tâches modelables sur des plates-formes multi-grappes quasi homogènes. En effet, de nombreuses plates-formes multi-grappes existantes exhibent des sous-ensembles constitués de nombreuses ressources qui sont pratiquement homogènes sur le plan de la vitesse des processeurs. Le premier de ces deux algorithmes sera une heuristique pragmatique qui adapte aux plates-formes multi-grappes une de nos améliorations de l'heuristique choisie dans le chapitre 3. Le second sera un algorithme issu de la théorie qui garantira une performance au pire cas par rapport au meilleur ordonnancement possible. Il est intéressant de confronter des heuristiques pragmatiques à des algorithmes garantis car la garantie offerte par ces dernières n'implique pas nécessairement que leur performance moyenne soit bonne par rapport aux heuristiques pragmatiques. Cette étude constituera sans doute la première fois qu'un algorithme garanti est évalué de manière expérimentale.

Il existe deux approches orthogonales de conception d'algorithmes d'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes en partant des heuristiques présentes dans la littérature. La première approche, utilisée par l'heuristique MHEFT [26], consiste à adapter aux tâches parallèles des algorithmes conçus pour l'ordonnancement de DAGs de tâches

séquentielles (applications qui utilisent uniquement le paradigme du parallélisme de tâches) en milieu hétérogène. Nous proposons des améliorations à l'heuristique MHEFT en vue de comparer les deux approches. L'autre approche est d'introduire la gestion des plates-formes hétérogènes dans des heuristiques d'ordonnancement de DAGs de tâches parallèles sur grappes homogènes. Cette seconde approche étant jusqu'alors inexplorée, nous proposerons de nouvelles heuristiques qui adaptent aux grappes hétérogènes de grappes homogènes les nouvelles heuristiques en deux étapes proposées dans le chapitre 3.

Ordonnancement concurrent de graphes de tâches modelables sur plates-formes hétérogènes multi-grappes

Il reste à gérer le caractère partagé pour s'approcher des plates-formes cibles que sont les grilles de calcul. Le problème majeur dans ce type d'ordonnancement est de garantir l'équité entre les applications. La gestion de l'équité consiste à trouver des ordonnancements où les applications (ou plus précisément les utilisateurs qui soumettent leurs applications) sont globalement satisfaites, c'est-à-dire que lorsqu'on considère une application donnée, celle-ci ne sera pas exagérément retardée par les autres applications.

Dans le chapitre 5 nous nous inspirerons des heuristiques proposées dans les chapitres précédents pour proposer les premières heuristiques d'ordonnancement concurrent de DAGs de tâches modelables. Notre idée pour gérer l'équité entre les applications sera de restreindre leurs taux d'utilisation respectifs des ressources. En effet, avec des ordonnancements individualistes certaines applications peuvent avoir des allocations de processeurs élevées à tel point qu'elle peuvent retarder ou être retardées par les autres applications en compétition pour les ressources.

Nous conclurons enfin la présentation de ces travaux par diverses perspectives de recherche et de développement.

Chapitre 2

Plates-formes, applications parallèles et ordonnancement

Sommaire

2.1 Introduction

Le calcul parallèle consiste en l'exécution d'un traitement pouvant être partitionné en tâches élémentaires adaptées afin de pouvoir être réparti entre plusieurs processeurs opérant simultanément. L'objectif est de traiter des problèmes de grandes tailles plus rapidement que dans une exécution séquentielle. Les avantages du calcul parallèle sont nombreux. Du fait de contraintes physiques et technologiques, la puissance d'un ordinateur monoprocesseur reste limitée. En effet, il est difficile pour les fabricants de processeurs de produire des processeurs dont la fréquence dépasse 4 GHz. La quantité de mémoire présente dans les ordinateurs monoprocesseurs est également limitée et il est par conséquent impossible d'exécuter certaines applications de grande taille sur de telles machines. Le calcul parallèle peut en revanche permettre de résoudre ce genre de problème.

L'ordonnancement est un aspect fondamental dans la parallélisation des applications car il permet d'améliorer leurs performances et de mieux gérer les ressources disponibles. L'ordon-

nancement d'une application parallèle consiste à déterminer l'ordre dans lequel les tâches de cette application doivent être exécutées et les ressources à leur affecter. De ce fait, les algorithmes d'ordonnancement dépendent aussi bien de l'architecture cible que de la structure de l'application.

Après avoir étudié différents types d'infrastructures d'exécution parallèle, nous proposerons une classification des applications parallèles. Enfin, nous ferons un état de l'art des différentes techniques d'ordonnancement d'applications parallèles existantes.

2.2 Infrastructures d'exécution parallèle

De nos jours, les plates-formes d'exécution d'applications parallèles ont évolué vers les machines MIMD (*Multiple Instruction Multiple Data*) capables d'effectuer simultanément différentes opérations sur des données différentes. Nous pouvons classer ces plates-formes en quatre catégories. La première catégorie regroupe les machines à mémoire partagée dont les processeurs ont tous accès à une mémoire commune. Dans la seconde classe, nous avons les machines à mémoire distribuée où chaque processeur dispose de sa propre mémoire privée. La troisième classe concerne des plates-formes hiérarchiques qui sont une interconnexion via un réseau de plusieurs machines à mémoire partagée. Enfin, dans la dernière catégorie on distingue des plates-formes beaucoup plus hétérogènes que les trois premières, appelées grilles de calcul. Cette section décrit en détails ces quatre types de plates-formes.

2.2.1 Plates-formes à mémoire partagée

Les machines à mémoire partagée se caractérisent par le fait que tous les processeurs ont accès à une mémoire vive commune avec un espace d'adressage global. Les processeurs peuvent travailler indépendamment mais partagent les mêmes ressources mémoire. Ainsi tout changement opéré par un processeur à une adresse mémoire donnée peut être visible quasi instantanément par tous les autres processeurs. Ces machines peuvent être classées en trois catégories. Ce sont les machines UMA, les machines NUMA et les machines COMA.

Les machines UMA (*Uniform Memory Acces*) sont telles que l'accès à la mémoire partagée est uniforme, c'est-à-dire que tous les processeurs ont la même vitesse d'accès à la mémoire. L'architecture UMA peut être représentée par des machines multiprocesseurs symétriques ou machines SMPs (*Symetric Multiprocessors*) classiques ou par des machines symétriques avec des microprocesseurs multicœurs. Dans les machines à microprocesseurs multicœurs connues également sous le nom de machines CMP (*chip-level multiprocessor*), les unités de calcul gravées à l'intérieur d'une même puce peuvent être considérées indépendamment. Chacune de ces unités de calcul peut avoir sa propre mémoire cache. Toutefois elles ont accès de manière symétrique au bus de la mémoire partagée.

Les machines NUMA (*Non-Uniform Memory Acces*) sont issues de l'agencement hiérarchique de plusieurs machines UMA. Chaque nœud comprend un ou plusieurs processeurs avec des caches privés et une partie de la mémoire partagée. Différentes copies d'un bloc mémoire donné peuvent résider dans plusieurs caches au même instant. Une opération d'écriture sur ce bloc nécessite alors la mise en œuvre d'un mécanisme qui permettra d'éviter aux processeurs de lire les anciennes valeurs contenues dans leurs caches. Ce mécanisme est la gestion de la *cohérence de cache*. Du fait qu'il serait ardu de programmer ces machines sans une gestion transparente de la cohérence de cache, toutes les machines NUMA actuelles intègrent un protocole de cohérence de cache. On parle alors de machines ccNUMA (*cache coherent NUMA*). Un exemple de machine NUMA avec gestion de cohérence de cache est le *SGI Altix 3000*. Cette machine peut contenir jusqu'à 512

processeurs à ce jour. Sur une machine NUMA classique, pour un processeur donné, le temps d'accès à une partie distante de la mémoire partagée peut être de deux à dix fois supérieur à celui de l'accès à la partie locale de cette mémoire. Du fait de ces latences importantes, l'accès fréquent à des données distantes dégrade les performances des machines NUMA. D'autre part, il peut être difficile pour un programmeur ou un compilateur d'optimiser la localisation des données d'une application sur de telles machines.

Les machines COMA(*Cache only memory architecture*) [38] font face à ce problème en augmentant les chances que les données sollicitées par un processeur soit présentes localement. Elles permettent la gestion transparente de la réplication ou de la migration des données utilisées par un processeur vers sa mémoire locale. Ainsi les machines COMA ont la même architecture que les machines NUMA mais chaque module mémoire fonctionne comme un cache.

Le point fort de toutes ces machines à mémoire partagée est la rapidité du partage de données entre des processeurs qui exécutent une même application parallèle. Cela est dû au fait que la mémoire commune est relativement proche des différents processeurs mis en jeu. En contrepartie, ces plates-formes souffrent d'un problème de passage à l'échelle. En effet, plus on augmente le nombre de processeurs, plus le trafic est important sur les bus mémoire et l'accès est donc de moins en moins rapide. Il existe également un surcoût en temps d'exécution non négligeable lié aux protocoles de maintenance de la cohérence de cache. Ainsi, le développement de plates-formes à mémoire partagée devient difficile et très coûteux lorsque le nombre de processeurs augmente. D'autre part, sur ces plates-formes plutôt adaptées à la programmation *multithread* (POSIX Threads, OpenMP, ...), le programmeur est responsable de la synchronisation des *threads* de son application. Il doit donc lui même veiller à l'intégrité des données en mémoire.

2.2.2 Plates-formes à mémoire distribuée

Dans les machines multiprocesseurs à mémoire distribuée, chaque processeur dispose de sa propre mémoire privée. Les processeurs sont interconnectés à travers un réseau et se communiquent les données par échanges de messages via ce réseau. Ces plates-formes sont également connues sous le nom d'architectures à passage de messages. Comme exemple de machines multiprocesseurs à mémoire distribuée nous pouvons citer les grappes de calcul dans lesquelles chaque nœud est constitué d'un ou plusieurs processeurs monocœurs. Une grappe de calcul est l'interconnexion au sein d'un réseau local (typiquement à l'intérieur d'une même salle machine), de plusieurs ordinateurs constitués avec des composants standards du marché. Le supercalculateur IBM SP2 (*Scalable Power2 parallel*) beaucoup utilisé dans les années 1990 fait partie de cette catégorie de plates-formes parallèles. Les nœuds utilisent la puce *RISC system 6000* d'IBM et sont interconnectés grâce à un commutateur haute performance ou HPS (*High-Performance Switch*).

Le principal avantage de ces plates-formes est qu'elles passent très bien à l'échelle et ne connaissent pas de limite théorique en termes de nombre de processeurs. En outre, plus on augmente le nombre de processeurs, plus la mémoire disponible est importante. De ce fait, ces machines peuvent permettre de résoudre des problèmes de très grandes tailles. Ces plates-formes sont également devenues compétitives faces aux plates-formes à mémoire partagées grâce au développement de réseaux à fortes bandes passantes et à faibles latences. Contrairement aux plates-formes à mémoire partagée, chaque processeur a un accès rapide à sa mémoire locale et ce, sans interférence avec les autres processeurs. Elles ne nécessitent pas non plus de maintenir une éventuelle cohérence de cache, ce qui évite le surcoût en temps de calcul lié aux protocoles de cohérence de cache.

Un des inconvénients des plates-formes à mémoire distribuée est que la latence du réseau peut rallonger les temps d'exécution de certaines applications. Les performances de ces plates-

formes seront d'autant plus mauvaises si d'une part, on a un réseau où la latence entre les nœuds est relativement élevée et d'autre part, on y exécute des applications à grain fin pour lesquelles, par définition, les communications entre les tâches sont importantes. Un autre problème lié à ces plates-formes est que le programmeur doit tenir compte de détails associés au découpage du problème à résoudre et à l'échange de données entre les processeurs. C'est le cas par exemple si l'on programme une application destinée aux plates-formes à mémoire distribuée à l'aide de la bibliothèque de passage de messages MPI (*Message Passing Interface*).

2.2.3 Plates-formes hiérarchiques

Aujourd'hui, il est de plus en plus difficile pour les fabricants de processeurs d'augmenter la vitesse de leurs processeurs. D'autre part, plus la fréquence d'un processeur est élevée, plus il dissipe de la chaleur et consomme de l'énergie. En intégrant plusieurs unités de calcul avec des fréquences relativement basses dans une même puce, les microprocesseurs multicœurs permettent à la fois de traiter plusieurs tâches (ou plusieurs processus - d'une ou plusieurs applications) en parallèle et de réduire la consommation d'énergie par rapport à une puce monocœur de puissance équivalente. Ainsi, les microprocesseurs multicœurs se répandent de plus en plus afin de faire face à la demande croissante en puissance calcul de la part des nouvelles applications destinées aux ordinateurs grand public.

De nombreuses plates-formes parallèles suivent cette tendance et sont une mise en réseau de machines à mémoire partagée (des grappes de machines SMP par exemple). Chaque nœud est une machine à mémoire partagée, avec éventuellement un ou plusieurs microprocesseurs multicœurs. Le nombre de processeurs (ou d'unités de calcul) est en général très faible au niveau de chaque nœud par rapport au nombre total de processeurs de la plate-forme. Les différents nœuds sont reliés via un réseau et s'échangent des données par passage de messages comme sur les plates-formes multiprocesseurs à mémoire distribuée. L'architecture à passage de messages est donc prédominante sur ces plates-formes hiérarchiques et elles présentent les mêmes avantages et les mêmes inconvénients que les machines multiprocesseurs à mémoire distribuée. Par conséquent, ces plates-formes hiérarchiques sont souvent programmées en utilisant le paradigme de passage de messages. Afin de tirer le meilleur parti de certaines de ces plates-formes, certaines applications tentent d'éviter des communications superflues entre des processeurs d'un même nœud et profitent de la rapidité d'échanges de données offerte par le partage de la mémoire sur ce nœud. Elles utilisent alors des méthodes de programmation hybrides [42, 64, 110] qui allient la programmation *multithread* pour le partage d'un même espace d'adressage entre des processus qui s'exécutent sur un même nœud et, le paradigme de passage de messages pour gérer les échanges de données entre des nœuds distants. Toutefois, l'utilisation pure du paradigme de passage de messages reste compétitive au niveau des temps d'exécution par rapport à la programmation hybride [42].

Comme exemple de plate-forme hiérarchique, nous pouvons citer le supercalculateur Blue Gene [109] eServer du *Lawrence Livermore National Laboratory*. Il est actuellement classé comme étant le supercalculateur le plus puissant au monde sur le site du TOP500 [97]. Il comprend 212 992 cœurs de processeurs PowerPC 440 cadencés à 700 MHz. Chaque puce élémentaire est un microprocesseur bicœur. Les applications qui y sont exécutées utilisent principalement la bibliothèque de passage de messages MPI.

2.2.4 Grilles de calcul

L'idée qui a conduit au développement des grilles informatiques est analogue au fonctionnement des réseaux électriques où les générateurs sont distribués, mais où les utilisateurs ont un accès transparent à l'électricité sans se préoccuper de la source de cette énergie. Une grille informatique [51] est une infrastructure matérielle et logicielle qui permet le partage coordonné de ressources par un ensemble d'individus et d'institutions, dans un environnement flexible et sécurisé. Ces ressources sont de divers ordres et l'on peut avoir des ressources de calcul, des ressources de stockage, des logiciels, des données, etc. Les grilles informatiques visent à être des systèmes transparents, hautement disponibles, fiables et avec une bonne qualité de service tout en étant bon marché. Depuis quelques années, il s'est formé de nombreuses communautés pour développer de nouvelles technologies pour les grilles informatiques. L'une de ces communautés les plus connues est la *Globus Alliance* [9] qui maintient un ensemble d'outils logiciels dédiés aux grilles. Cet ensemble d'outils, le *Globus toolkit*, est devenu un standard *de facto* pour implanter des solutions de grilles informatiques. Le *Globus toolkit* comprend des logiciels applicatifs et des bibliothèques logicielles pour la sécurité, la gestion des ressources, la découverte des ressources, la manipulation des fichiers et le lancement et le suivi des applications. Tous les logiciels proposés dans cette boîte à outils sont ouverts afin de permettre le partage des technologies proposées et de favoriser un développement plus rapide de ces technologies. Des efforts de standardisation des technologies liées aux grilles informatiques sont effectués autour de l'*Open Grid Forum* (OGF) [80] dont la *Globus Alliance* est un des membres. L'OGF est une association qui regroupe des entreprises, des gouvernements, des scientifiques et des organisations académiques à travers le monde dans le but de développer et promouvoir des technologies standards pour les grilles de calcul.

Il existe plusieurs types de grilles informatiques dont les grilles de calcul qui fédèrent de nombreuses ressources afin d'effectuer du calcul haute performance. Les grilles de calcul sont des plates-formes potentiellement très hétérogènes et partagées par de nombreux utilisateurs. Différentes approches permettent d'implanter les grilles de calculs dont les grilles d'ordinateurs personnels, les plates-formes de *metacomputing* et les supercalculateurs virtuels.

Dans les grilles d'ordinateurs personnels (*desktop grids*), un très grand nombre d'ordinateurs reliés à Internet sont utilisés pour réaliser du calcul distribué. C'est le cas des applications qui utilisent la plate-forme logicielle BOINC [19]. BOINC repose sur le volontariat des internautes et permet de récupérer les cycles CPUs inutilisés de leurs ordinateurs via un économiseur d'écran. Les grilles d'ordinateurs personnels sont potentiellement très hétérogènes et la latence entre les nœuds est élevée. En effet les ordinateurs de ces grilles sont en général reliés via Internet grâce à de simples connexions ADSL ayant des capacités limitées. Non seulement les distances entre les nœuds peuvent être très grandes, mais aussi le réseau peut être perturbé par de nombreuses autres applications. Ces grilles de calcul ne peuvent donc pas servir à exécuter des applications qui nécessitent beaucoup de communications ou des synchronisation entre les processeurs car les coûts des communications seraient trop importants. Les applications pouvant être exécutées sur ces grilles de calcul sont donc les applications hautement distribuées qui nécessitent peu de communication comparativement aux volumes de calcul.

Les plates-formes de *metacomputing* permettent de gérer un ensembles de serveurs de calcul. Chaque serveur offre à la fois un ensemble de logiciels et des ressources de calcul. Il existe de nombreux intergiciels de *metacomputing* dont DIET [40], Ninf [84, 108] et NetSolve/GridSolve [11, 78]. Ces intergiciels utilisent l'appel de procédures à distance sur les grilles (GridRPC ou *Remote Procedure Call*) et permettent d'accéder de manière transparente aux ressources grâce à des logiciels clients à même d'« appeler » des bibliothèques logicielles sur des

ressources distantes. Les intergiciels de *metacomputing* fonctionnent en général selon le principe client-agents-serveurs. Un utilisateur, à travers un client, soumet sa requête de calcul à un ou plusieurs agents qui se chargent de trouver le ou les serveurs disponibles capables de satisfaire cette requête.

Les supercalculateurs virtuels sont une association en réseau de plusieurs supercalculateurs ou grappes de calcul répartis sur plusieurs sites géographiquement distants. Chaque nœud est une machine parallèle administrée indépendemment des autres et peut être contrôlée par un gestionnaire de ressources tel que OAR [24, 79], PBS [56] ou Maui [61]. Les gestionnaires de ressources également appelés gestionnaires de tâches, comme leur nom l'indique, permettent de gérer le lancement des tâches et les ressources sur une plate-forme parallèle. En général, ils admettent deux principales catégories de requêtes. La première concerne les demandes de réservations de nœuds. L'utilisateur doit dans ce cas spécifier la date de début de la réservation, le nombre de nœuds souhaités et la durée de la réservation. Si cette requête ne peut être satisfaite, alors la demande est rejetée par le gestionnaire. Le second type de requête est la demande de lancement d'une tâche sans réservation. Le gestionnaire introduit alors cette tâche dans une file d'attente et déterminera où et quand elle sera exécutée.

Le principal avantage des grilles de calcul réside dans l'agrégation des ressources (matérielles ou logicielles). Elles permettent ainsi d'accéder à moindre coût à de nombreuses ressources afin de résoudre des problèmes de grandes tailles. En outre, l'approche orientée services de certaines grilles de calcul telles que les plates-formes de *metacomputing* permet aux utilisateurs d'accéder aux ressources en utilisant des interfaces de programmation relativement simples. Cependant, la forte hétérogénéité des grilles de calcul et la distribution des ressources sur des sites géographiquement distants les rendent difficilement exploitables par certains types d'applications parallèles.

De nos jours, avec l'amélioration des performances des réseaux informatiques, de nombreuses institutions déploient des grappes homogènes puisque celles-ci représentent des supercalculateurs à faible coût. Des grilles se développent de plus en plus par l'interconnexion de ces grappes homogènes. C'est le cas de la grille expérimentale Grid'5000 [25, 55] répartie sur le territoire français et destinée aux expérimentations des chercheurs en informatique. Dans cette thèse nous nous intéresserons à cette forme particulière de supercalculateur virtuel peu hétérogène : les agrégations hétérogènes de grappes homogènes. Ce type de plate-forme peut être hétérogène entre les grappes (différents types et nombre de processeurs, etc.) mais est homogène au sein d'une grappe (processeurs de même vitesse, réseau local homogène, etc.). Aucune topologie spécifique n'est imposée, mais le réseau d'interconnexion est souvent hiérarchique et rapide. Ces plates-formes peu hétérogènes sont devenues très attrayantes car elles peuvent permettre de déployer des applications parallèles à des échelles sans précédent.

2.3 Classification des applications parallèles

Une application parallèle est composée soit d'une tâche pouvant s'exécuter sur plusieurs processeurs, soit de plusieurs tâches (par exemple des sous-routines ou des boucles) dont certaines peuvent s'exécuter en parallèle sur des ensembles de processeurs distincts. La modélisation d'une application parallèle dépend aussi bien des tâches qui la composent que des interactions entre ces tâches.

Il existe deux principales catégories de tâches : les tâches séquentielles et les tâches parallèles. Une tâche séquentielle est une tâche qui ne s'exécute que sur un seul processeur. À l'inverse, une tâche parallèle est telle que les données à traiter peuvent être réparties sur plusieurs proces-

seurs afin de s'exécuter concurremment. On parle alors de parallélisme de données. Les tâches parallèles se déclinent également en diverses classes. Pour l'ordonnancement d'applications de tâches parallèles, il est courant d'associer des modèles d'accélération aux tâches afin d'estimer leurs temps d'exécution en fonction des ressources allouées.

Dans les sections suivantes, nous présenterons différents types de tâches parallèles, des modèles d'accélérations de tâches parallèles avant de détailler diverses catégories d'applications parallèles.

2.3.1 Catégories de tâches parallèles

Il est possible de distinguer trois principales classes de tâches parallèles selon leur flexibilité [49]. En premier lieu, nous pouvons citer les *tâches rigides* où le nombre de processeurs exécutant la tâche parallèle est fixée à priori. Ensuite, il y a les *tâches modelables* pour lesquelles le nombre de processeurs n'est pas fixé mais est déterminé avant l'exécution. Après avoir été fixé, ce nombre de processeurs ne change pas jusqu'à la fin de l'exécution. Enfin, les *tâches malléables* peuvent voir le nombre de processeurs qui leur est alloué changer en cours d'exécution (par préemption des tâches ou par redistribution des données).

Pour des raisons historiques, la plupart des applications soumises aux plates-formes de calcul parallèle sont traitées comme étant des tâches rigides. Pourtant la plupart des tâches parallèles sont intrinsèquement modelables puisqu'un programmeur ne connaît pas nécessairement à l'avance le nombre de processeurs qui seront disponibles à l'exécution. Le caractère malléable d'une tâche est plus facilement utilisable du point de vue de l'ordonnancement mais requiert des fonctionnalités avancées (pouvoir préempter une tâche à tout moment de son exécution, continuer ultérieurement cette exécution sur un ensemble de processeurs quelconque, ...) de la part de l'environnement d'exécution qui sont rarement disponibles.

Dans cette thèse nous nous intéresserons à l'ordonnancement d'applications composées de tâches modelables. L'ordonnancement d'une application parallèle nécessite la prédiction du temps d'exécution de chaque tâche en fonction des ressources utilisées. Cette prédiction peut s'effectuer en utilisant des modèles d'accélération qui décrivent l'évolution du temps d'exécution de chaque tâche parallèle en fonction du nombre de processeurs alloués à l'exécution. La section suivante présente quelques uns de ces modèles d'accélération et discute du choix que nous allons opérer pour la suite de cette thèse.

2.3.2 Modèles d'accélération

De nombreux modèles d'accélération [10, 15, 41, 46, 88] ont été proposés pour la prédiction du temps d'exécution de tâches parallèles. Ces modèles supposent en général que l'ensemble de processeurs utilisé est homogène et sont une fonction qui dépend du nombre de processeurs (p) alloués à la tâche et de quelques paramètres liés à cette tâche. En pratique, les paramètres de ces modèles sont déterminés pour être adaptés à chaque tâche soit analytiquement soit expérimentalement. L'accélération S se définit comme étant le rapport entre le temps d'exécution d'une tâche parallèle sur un processeur et son temps d'exécution sur p processeurs.

L'un des modèles les plus utilisés dans la littérature est la loi d'Amdahl [10]. Ce modèle permet d'estimer le temps d'exécution d'une application parallèle sur une grappe homogène (en termes de réseaux et de caractéristiques des nœuds de calcul). Selon cette loi, le temps d'exécution d'une tâche parallèle t est donné par :

$$T(t,p) = \left(\alpha + \frac{1-\alpha}{p}\right) \cdot T(t,1), \tag{2.1}$$

p étant le nombre de processeurs alloués à t, $T(t,1)$ son temps d'exécution sur un seul processeur (temps d'exécution séquentiel) et α sa portion non parallélisable. Cette loi stipule donc qu'une tâche parallèle peut se décomposer en une partie parallélisable et une partie non parallélisable. Le temps d'exécution de la partie parallélisable est inversement proportionnel aux nombre de processeurs tandis que le temps d'exécution de la partie non parallélisable ne varie pas quelque soit le nombre de processeurs disponibles (elle ne s'exécute que sur un processeur). On peut remarquer que ce modèle ne tient pas explicitement compte d'éventuels coûts de communications entre les processeurs exécutant une même tâche. Ce modèle est donc plutôt adapté aux tâches pour lesquelles les communications internes sont négligeables ou celles dont les calculs recouvrent totalement les communications.

Un autre modèle d'accélération pour un ensemble homogène de processeurs est celui proposé par Downey [41]. Ce modèle est une fonction non linéaire dépendant de deux paramètres A et σ. A représente le parallélisme moyen de la tâche et σ est une approximation du coefficient de variation du parallélisme. Dans son étude, Downey distingue deux types de profils de tâches parallèles selon la valeur de la *variance du parallélisme* $V = \sigma(A-1)^2$. Pour les tâches parallèles avec une faible variance ($\sigma \leq 1$), l'accélération de Downey s'exprime de la manière suivante :

$$S(p) = \begin{cases} \frac{Ap}{A+\sigma/2(p-1)} & 1 \leq p \leq A \\ \frac{Ap}{\sigma(A-1/2)+n(1-\sigma/2)} & A \leq p \leq 2A-1 \\ A & p \geq 2A-1 \end{cases} \quad (2.2)$$

Pour les tâches parallèles avec une variance élevée ($\sigma \geq 1$), on a :

$$S(p) = \begin{cases} \frac{pA(\sigma+1)}{\sigma(p+A-1)+A} & 1 \leq p \leq A+A\sigma-\sigma \\ A & p \geq A+A\sigma-\sigma \end{cases} \quad (2.3)$$

Downey a expérimentalement vérifié que son modèle est adapté à de nombreuses tâches parallèles sur les grappes homogènes. Toutefois certaines tâches ne peuvent être approchées par ce modèle. En effet, de par sa conception, ce modèle ne prend pas en compte les tâches qui ont une accélération non monotone ou une accélération superlinéaire. Il en est de même pour le modèle d'Amdahl ne tient pas compte de ces deux types de tâches.

Dans cette thèse, nous utiliserons le modèle d'Amdahl pour représenter les tâches des applications qui permettront d'évaluer nos heuristiques. En effet, en plus d'être relativement simple, il permet de représenter une large gamme de tâches parallèles. Ce modèle d'accélération ayant été conçu pour les grappes homogènes, nous restreindrons l'exécution d'une tâche parallèle à l'intérieur d'une grappe. Ce choix est également motivé par le fait qu'en pratique les communications intergrappes durant l'exécution d'une même tâche peuvent être très coûteuses lorsque la latence entre les processeurs impliqués est élevée. Pour la validité de ce modèle sur les agrégations de grappes homogènes que nous utiliserons, nous admettons également que les processeurs sont uniformes. Contrairement aux ensembles de processeurs non corrélés dont le temps d'exécution n'est pas lié à la vitesse des processeurs (du fait qu'ils peuvent être de différentes natures et avoir des caractéristiques non proportionnelles), les processeurs uniformes sont tels que le temps d'exécution séquentiel d'une tâche est inversement proportionnel à la vitesse du processeur considéré. Dans cette étude, la vitesse des processeurs sera mesurée en milliards d'opérations en virgule flottante par seconde (Gflop/s).

Le modèle d'Amdahl ne tenant pas explicitement compte des communications, nous proposons un second modèle que nous allons également utiliser dans cette thèse. Celui-ci comprend la quantité de calcul à effectuer (en nombre d'opérations en virgule flottantes - Gflop) et le

volume de données à échanger (en octets) entre les processeurs qui exécuteront la tâche. Nous supposons dans ce modèle que les calculs et les communications se recouvrent. Ainsi le temps d'exécution d'une tâche t sur p processeurs identiques sera le maximum entre le temps mis pour effectuer les opérations de calcul ($T_{comp}(t,p)$) et le temps mis pour effectuer les communications ($T_{comm}(t,p)$) :

$$T(t,p) = \max\{T_{comp}(t,p), T_{comm}(t,p)\}, \tag{2.4}$$

Nous admettons que les calculs sont équitablement répartis entre les processeurs. Ainsi, T_{comp} est inversement proportionnel au nombre de processeurs mis en jeu. Si on note $V_{comp}(t)$ la quantité de calcul à réaliser (en Gflop) et v la vitesse d'un des processeurs (en Gflop/s), alors on a :

$$T_{comp}(t,p) = \frac{V_{comp}(t)}{v \times p}. \tag{2.5}$$

En ce qui concerne les échanges entre les processeurs, nous faisons l'hypothèse qu'ils s'effectuent au sein de communications collectives de type *All-to-All Broadcast* où chaque processeur diffuse ses données à tous les autres processeurs. T_{comm} dépend notamment de la topologie du réseau. Dans cette thèse, les processeurs d'une même grappe seront interconnectés à travers un commutateur (*switch*) avec des liens identiques. En l'absence de contentions dues à d'autres flux de données on a :

$$T_{comm}(t,p) = \frac{2 \times V_{comm}(t) \times (p-1)}{p^2 \times B}, \tag{2.6}$$

où $V_{comm}(t)$ représente le volume total de donnée à échanger entre les p processeurs impliqués dans l'exécution de la tâche t, et B est la bande passante des liens de la grappe sur laquelle la tâche est exécutée. Cela suppose que le commutateur a une bande passante suffisamment grande et ne subit aucune contention lorsque tous les processeurs s'échangent des données au même moment.

2.3.3 Catégories d'applications parallèles

Les applications parallèles peuvent se subdiviser en deux principales catégories, selon qu'il existe ou non des relations entre les tâches qui les composent. La première catégorie regroupe les applications où toutes les tâches peuvent s'exécuter indépendamment les unes des autres. La seconde catégorie concerne les applications dans lesquelles il existe des relations de dépendance (de flots de données ou de précédences) entre les tâches.

Parmi les applications composées de tâches indépendantes, on distingue les applications comprenant des tâches séquentielles et celles constituées de tâches parallèles. Comme exemple d'applications composées de tâches séquentielles indépendantes, nous pouvons mentionner les applications multiparamétriques (*parameter sweep applications*) [28]. Une application multiparamétrique est conçue de sorte qu'un même programme est lancé plusieurs fois en faisant varier les valeurs de ses paramètres d'entrée. De nombreuses applications telles que celles utilisant la méthode de simulation de Monte-Carlo font partie de cette catégorie. D'autres applications, notamment dans les domaines de l'algèbre linéaire [31] et de la bioinformatique, sont composées d'un très grand nombre de tâches séquentielles indépendantes de sorte que ces tâches peuvent être regroupées en un nombre arbitraire de tâches indépendantes. Ces applications peuvent alors être représentées par le modèle des *applications divisibles* [17, 113]. Une application divisible est une application dont le volume de calcul à effectuer peut être partitionné et distribué entre les processeurs de n'importe quelle façon.

Il existe deux principaux types d'applications composées de tâches dépendantes. Le premier concerne les applications dont les tâches interagissent (s'échangent des données, etc.) durant leurs exécutions. Ces applications peuvent être modélisées à l'aide de graphes non orientés appelés *Task interaction graphs* (TIGs). Les nœuds de ces graphes représentent des processus monoprocesseurs (tâches séquentielles) et les arrêtes représentent les communications entre ces processus. Le second type d'applications regroupe les applications avec des contraintes de précédence entre les tâches. Une tâche d'une telle application ne peut démarrer son exécution que si toutes ses contraintes de précédence sont satisfaites. Ces applications peuvent être modélisées par des graphes acycliques orientés ou DAGs (*Directed Acyclic Graphs*). Les nœuds représentent les tâches (séquentielles ou parallèles) et les arcs définissent les relations de précédence (dépendances de flots ou de données) entre les tâches. Pour les DAGs composés de tâches séquentielles, on parle d'applications qui exploitent purement le paradigme de parallélisme de tâches. Ces deux types d'applications peuvent se combiner et donner ainsi lieu à des applications hybrides représentées par des TTIGs (*Temporal Task Interaction Graphs*) [89].

Plusieurs études [30, 87, 105] ont montré qu'il existe de nombreux types d'applications à gros grain pour lesquelles il est intéressant d'exploiter simultanément le parallélisme de données et le parallélisme de tâches. La modélisation de ces applications par des DAGs de tâches parallèles (souvent modelables) permet d'améliorer leur temps d'exécution aussi bien par rapport à l'utilisation du parallélisme de données seul que par rapport à l'utilisation du parallélisme de tâches seul. On parle alors de parallélisme mixte. Dans cette thèse, nous étudierons l'ordonnancement de tels DAGs de tâches parallèles.

Cette section a montré qu'il existe plusieurs modèles de tâches et d'applications parallèles. L'ordonnancement de ces applications vise à améliorer leurs performances et à utiliser efficacement les ressources. De nombreux algorithmes d'ordonnancement ont été étudiés et divers objectifs sont visés selon les types d'applications et de plates-formes ciblés. Dans la section suivante, nous faisons un état de l'art des algorithmes d'ordonnancement d'applications parallèles.

2.4 État de l'art sur l'ordonnancement

La plupart des problèmes d'ordonnancement d'applications sur des machines parallèles sont NP-difficiles [33, 43, 52]. Diverses études ont donc été réalisées dans le domaine de l'ordonnancement pour proposer des algorithmes approchés et des heuristiques. Nous avons regroupé les algorithmes existants dans la littérature en trois catégories. Nous présenterons tout d'abord des algorithmes d'ordonnancement pour les applications composées de tâches indépendantes. Ensuite nous étudierons les algorithmes d'ordonnancement pour les applications composées de tâches dépendantes avant de traiter des algorithmes d'ordonnancement concurrent de plusieurs applications.

2.4.1 Ordonnancement de tâches indépendantes

De nombreuses études ont été effectuées pour l'ordonnancement de tâches séquentielles indépendantes [16, 17, 23, 29, 31, 69, 74, 102, 113, 118, 119] que ce soit en ciblant des plates-formes homogènes ou hétérogènes. En ce qui concerne l'ordonnancement de tâches parallèles indépendantes [12, 48, 62, 66, 76, 77, 99, 100, 101, 103, 112], les études ont été effectuées essentiellement pour des plates-formes homogènes.

Les algorithmes d'ordonnancement de tâches séquentielles indépendantes sur plates-formes hétérogènes visent en général à minimiser le temps de complétion (ou *makespan*) de l'application ordonnancée ou à maximiser le débit (*throughput*) de la plate-forme utilisée. Le *makespan* d'une

application représente le temps d'exécution total de cette application sur une plate-forme donnée. Les algorithmes visant à optimiser le *makespan* considèrent que toutes les tâches sont présentes initialement et que l'application à ordonnancer dispose à elle seule de toutes les ressources de la plate-forme. Dans ce cas on peut utiliser des algorithmes d'ordonnancement statiques (ou hors ligne) qui ordonnancent toute l'application avant le début de l'exécution de la première tâche. Les algorithmes qui ont pour objectif l'optimisation du débit sont couramment utilisés sur les plates-formes de production pour lesquelles les tâches arrivent continuellement dans le système. On a donc recours à des techniques d'ordonnancement dynamiques où les décisions sont prises au fur et à mesure que les tâches sont soumises au système.

Le problème général de l'ordonnancement de n tâches séquentielles indépendantes sur m processeurs non corrélés visant à minimiser le *makespan* est un problème NP-difficile [52]. Des algorithmes approchés ont donc été proposés [69, 102].

L'avantage des algorithmes approchés, issus de la théorie, est qu'ils garantissent la performance au pire cas par rapport à l'ordonnancement optimal. La garantie de performance est un ratio qui ne doit jamais être dépassé par le rapport du *makespan* obtenu sur le *makespan* optimal. L'inconvénient de ces algorithmes est qu'en pratique ils sont difficilement utilisables pour des applications réelles. En effet, ils sont en général trop complexes et le temps qu'ils mettent à calculer l'ordonnancement est trop important. D'autre part ils ne tiennent pas compte d'éventuelles communications (transferts des données vers les processeurs qui exécuteront les tâches) qui existeraient dans des problèmes pratiques. Pour faire face à des problèmes pratiques, de nombreuses heuristiques non garanties ont été proposées. Celles-ci présentent l'avantage d'être peu complexes en général et elles visent à obtenir de bonnes performances moyennes même si elles n'offrent aucune garantie sur leurs performances au pire cas.

Dans [74], plusieurs heuristiques dynamiques sont proposées pour l'ordonnancement de tâches séquentielles indépendantes, dont des heuristiques d'ordonnancement en ligne et des heuristiques d'ordonnancement par lot (*batch scheduling*). Dans le mode d'ordonnancement en ligne, une tâche est ordonnancée sur un des processeurs de la plate-forme dès qu'elle arrive dans le système. En ce qui concerne l'ordonnancement par lot, les tâches ne sont pas ordonnancées aussitôt qu'elles arrivent. Elles sont d'abord collectées dans un ensemble de tâches qui sera ordonnancé ultérieurement, à un instant prédéterminé.

Des heuristiques pratiques ont également été proposées pour l'ordonnancement d'applications multiparamétriques [23, 29] et pour l'ordonnancement d'applications divisibles [16, 17, 31, 113, 118, 119]. Il est à noter que les heuristiques d'ordonnancement d'applications divisibles tiennent compte du réseau d'interconnexion de la plate-forme afin de distribuer efficacement les tâches.

L'ordonnancement de tâches parallèles indépendantes a été largement étudié aussi bien dans le domaine théorique que dans des cas pratiques. Parmi les études effectuées, on distingue les algorithmes d'ordonnancement en espace partagé et les algorithmes d'ordonnancement en temps partagé. Pour les algorithmes d'ordonnancement en espace partagé, deux tâches distinctes ne peuvent pas être exécutées de manière concurrente sur un même processeur. À l'inverse, avec l'ordonnancement en temps partagé, plusieurs tâches peuvent se partager le temps CPU d'un même processeur durant leurs exécutions.

Au sein des algorithmes d'ordonnancement en espace partagé, il y a les algorithmes d'ordonnancement statique (hors ligne) qui visent à minimiser le *makespan*. Le problème le plus simple dans cette classe concerne l'ordonnancement hors ligne de tâches parallèles rigides sur des ensembles de processeurs homogènes. C'est le problème du *strip-packing* ou, comment empiler des rectangles dans un rectangle de largeur fixe en vue d'obtenir la hauteur minimale. Il s'agit également d'un problème NP-difficile qui a été largement étudié [66, 103, 112]. En ce qui concerne l'ordonnancement statique de tâches modelables indépendantes sur des processeurs identiques, il

existe aussi bien des heuristiques approchées [112] que des algorithmes approchés plus complexes utilisant la programmation linéaire [62].

De nombreuses heuristiques dynamiques ont été proposées pour l'ordonnancement de tâches parallèles indépendantes. Ces heuristiques sont notamment utilisées dans les gestionnaires de ressources (des grilles de calcul ou des supercalculateurs) tels que OAR [24, 79], PBS [56], Maui [61], etc. Leur objectif est d'augmenter le taux d'utilisation de la plate-forme ou de diminuer le temps moyen d'attente des tâches. L'une des heuristiques les plus simples est FCFS (*First Come First Served*) ou *Premier Arrivé Premier Servi*. Avec cette heuristique, les tâches débutent leur exécution dans l'ordre strict de leur arrivée. La tâche à la tête de la file d'attente commence son exécution dès qu'il y a suffisamment de processeurs libres. L'heuristique FCFS permet de garantir qu'il n'y aura pas de situation de famine, c'est-à-dire que toute tâche qui arrive dans le système est certaine de pouvoir s'exécuter. De plus, elle ne nécessite pas la prédiction des temps d'exécution des tâches. Il s'agit donc d'une heuristique simple *non clairvoyante*. L'inconvénient est qu'elle peut occasionner de nombreux trous dans l'ordonnancement et donc conduire à une mauvaise utilisation des ressources. Cette situation a conduit à l'introduction de techniques de *backfilling* [72, 77, 99] dans FCFS. Le *backfilling conservatif* [77, 99] consiste à avancer l'exécution des tâches nécessitant peu de processeurs et de remplir les trous dans l'ordonnancement avec ces tâches si celles-ci ne retardent pas des tâches plus prioritaires (les tâches arrivées plus tôt dans la file d'attente). Cette technique nécessite une prédiction précise des temps d'exécution des tâches. Tout comme FCFS, elle permet d'éviter les situations de famine. Des expériences ont montré que le *backfilling conservatif* augmente significativement l'utilisation des ressources par rapport à FCFS. Le *backfilling agressif* [72] permet d'augmenter encore un peu plus l'utilisation des ressources. Cette technique consiste également à remplir les trous dans l'ordonnancement avec des tâches plus petites mais des tâches en tête de file, en attente de ressources, peuvent être retardées. Le *backfilling agressif* peut donc causer une situation de famine pour des tâches nécessitant de nombreuses ressources, c'est-à-dire que ces tâches peuvent être retardées indéfiniment par des tâches nécessitant moins de ressources. De nombreuses autres techniques ont été proposées pour améliorer l'utilisation des ressources des grilles par rapport à FCFS. L'une de ces techniques est LWF (*least work first*) qui donne la priorité aux tâches qui effectueront un plus petit travail. Le travail d'une tâche est estimé en faisant le produit du nombre de processeurs nécessaires par une estimation de son temps d'exécution. Dans [99], une étude a montré que le *backfilling conservatif* et LWF ont des performances similaires en matière d'utilisation des plates-formes. Avec une charge importante, ces deux heuristiques permettent d'utiliser en moyenne 70% des plates-formes étudiées.

L'ordonnancement en temps partagé permet d'obtenir une meilleure utilisation des plates-formes. Certains de ces algorithmes tirent profit de la préemption des tâches et peuvent en outre les réordonnancer sur d'autres ressources [76]. L'ordonnancement en gang [48] autorise plusieurs tâches à partager un même processeur durant leurs exécutions. Chaque tâche utilise les processeurs qui lui sont alloués seulement pendant un quantum de temps puis un changement de contexte sur ces processeurs permet de les attribuer à d'autres tâches concurrentes. Un des inconvénients de cette technique est que l'attribution d'un même processeur à plusieurs tâches peut allonger considérablement les temps d'exécution respectifs de chacune des tâches. De plus, il existe un surcoût non négligeable lié à ces changements de contextes coordonnés. L'ordonnancement en gang est donc inadapté aux grappes de calcul où la latence de communication entre les processeurs (quelques dizaines de microsecondes) est du même ordre de grandeur que le quantum des CPUs. Dans ce cas, le surcoût lié aux changements de contextes serait trop important. Le co-ordonnancement [12, 100, 101] permet de palier ce problème en réduisant les communications. Une des techniques de co-ordonnancement connues à ce jour est le co-ordonnancement

implicite qui utilise les communications et les messages de synchronisation d'une tâche parallèle pour coordonner son ordonnancement en temps partagé. Deux types d'événements sont utilisés pour réaliser l'auto-ordonnancement des processus d'une même tâche parallèle :

1. Si un processus envoie un message à un autre processus et attend la réponse plus longtemps que prévu, alors il peut inférer que le processus distant n'est pas ordonnancé actuellement. Par conséquent, le processus qui a envoyé le message libère son processeur et s'endort.
2. Si un processus reçoit un message, alors il admet que les autres processus de la même tâche sont actuellement ordonnancés. Ce processus est donc réveillé et pourra lui aussi être ordonnancé localement.

2.4.2 Ordonnancement de tâches dépendantes

Les études sur l'ordonnancement d'applications composées de tâches dépendantes visent essentiellement à minimiser le *makespan* de ces applications. Il existe quelques travaux sur l'ordonnancement de graphes de tâches qui interagissent (TIGs), notamment sur des plates-formes hétérogènes [20, 58]. Mais la plupart des travaux concernent l'ordonnancement de graphes de tâches avec contraintes de précédence (DAGs), ce type d'applications étant plus répandu. Le problème de l'ordonnancement de DAGs quelconques sur plusieurs processeurs avec pour objectif de minimiser le *makespan* est un problème NP-difficile, même pour les DAGs de tâches séquentielles [52]. La plupart des solutions d'ordonnancement proposées sont donc des heuristiques.

L'ordonnancement de DAGs composés de tâches séquentielles à été largement étudié aussi bien dans le cas particulier des plates-formes homogènes que dans le cas général des ensembles de processeurs hétérogènes. Les algorithmes obtenus peuvent être classés en diverses catégories.

Les *heuristiques d'ordonnancement de liste* [47, 67, 68, 94, 111, 116] se déroulent en deux phases. D'abord, une liste de tâches est construite en assignant une priorité à chaque tâche. Les priorités sont calculées de sorte que les contraintes de précédence sont respectées. Pour une tâche donnée, la priorité de cette tâche est ainsi inférieure à celle des tâches dont elle dépend. Les tâches de cette liste sont ensuite sélectionnées dans l'ordre des priorités et chaque tâche sélectionnée est ordonnancée sur le processeur qui minimise une fonction de coût prédéfinie (par exemple sa date de fin d'exécution). Les heuristiques d'ordonnancement de liste ont l'avantage d'être en général peu complexes par rapport aux autres catégories tout en produisant des *makespan* comparables. L'une des heuristiques d'ordonnancement de liste les plus connues est HEFT (*Heterogeneous Earliest-Finish-Time*) [111] conçue pour l'ordonnancement sur des processeurs hétérogènes non uniformes entièrement connectés.

Les *heuristiques de clustering* [81, 116, 117] exploitent l'idée de rassembler les tâches en plusieurs groupes (*clusters*), les tâches d'un même groupe étant contraintes de s'exécuter sur un même processeur.

D'autres heuristiques sont fondées sur *la duplication de tâches* [13, 8, 34, 82]. En effet, l'exécution redondante de certaines tâches peut être un moyen de réduire le surcoût en temps lié aux communications de données entre des tâches dépendantes. L'inconvénient de ces heuristiques est qu'elles ont une complexité bien supérieure aux deux premières catégories qu'elles génèrent surtout plus de travail pour le même résultat.

Certains algorithmes d'ordonnancement de DAGs de tâches séquentielles utilisent quant à eux la *recherche aléatoire guidée*. Les algorithmes génétiques [57, 92, 93, 96] ont été les plus étudiés dans cette catégorie. Bien que ces algorithmes produisent une bonne qualité d'ordonnancement (de bons *makespans*), leurs temps d'exécution (complexités) sont très supérieurs à ceux des autres catégories.

Plusieurs heuristiques ont été développées pour l'ordonnancement de DAGs de tâches parallèles sur des grappes homogènes [14, 70, 71, 85, 86, 87, 88, 115]. La plupart de ces heuristiques fonctionnent en deux étapes en séparant le problème de l'*allocation* de celui du *placement*. La phase d'allocation détermine le nombre de processeurs à allouer à chaque tâche. Ensuite la phase de placement décide de l'ensemble de processeurs sur lequel chaque tâche doit s'exécuter et l'ordre dans lequel les tâches sont lancées. Puisque nous souhaitons adapter les heuristiques existantes à l'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes multi-grappes, nous effectuerons une étude plus poussée des heuristiques conçues pour ordonnancer les DAGs de tâches parallèles sur grappes homogènes dans le chapitre suivant.

Il existe peu de travaux concernant l'ordonnancement de DAGs de tâches parallèles sur des plates-formes hétérogènes et plus particulièrement sur des agrégations hétérogènes de grappes homogènes. Dans [44], un algorithme garanti à été proposé spécifiquement pour ordonnancer des DAGs en forme d'arbre sur des grappes de machines SMPs. Les auteurs de [15] présentent de leur côté une heuristique destinée à l'ordonnancement de DAGs de tâches parallèles sur des grappes hétérogènes. Ils supposent que les processeurs sont uniformes et que les tâches sont telles que la quantité de données à calculer peut être répartie proportionnellement à la vitesse des processeurs. Lorsque nous avons débuté cette thèse, MHEFT [26] inspirée de l'heuristique de liste HEFT [111], était la seule heuristique d'ordonnancement de DAGs de tâches parallèles sur des agrégations hétérogènes de grappes homogènes. Par rapport à l'adaptation aux plates-formes hétérogènes des algorithmes d'ordonnancement de DAGs de tâches parallèles conçus pour les plates-formes homogènes, la méthode de mise en œuvre de MHEFT représente une approche orthogonale pour la conception d'algorithmes d'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes. Cette approche consiste à adapter aux tâches parallèles des algorithmes conçus pour l'ordonnancement de DAGs de tâches séquentielles en milieu hétérogène. Les deux approches seront comparées dans le chapitre 4. Enfin, dans une étude récente, Hunold *et al.* présentent une nouvelle heuristique vouée à l'ordonnancement de DAGs de tâches parallèles générés dynamiquement [59], sur des ensembles hétérogènes de grappes homogènes. Par exemple, il peut être question d'un processus récursif qui crée une nouvelle tâche si une certaine condition est satisfaite. Ainsi, la structure du DAG correspondant à leur application n'est entièrement connue qu'à la fin de l'exécution de celle-ci.

2.4.3 Ordonnancement multi applications

L'ordonnancement concurrent de plusieurs applications parallèles est un problème difficile du fait qu'il faut gérer l'équité entre les différentes applications en compétition pour les ressources tout en cherchant à réduire leurs *makespans*. Un ordonnancement multi applications peut être considéré comme étant équitable si les applications en compétition (où les utilisateurs qui soumettent ces applications) sont globalement satisfaites. Si un algorithme est non équitable, il peut entraîner au pire cas des situations de famine, c'est-à-dire que certaines tâches seront retardées indéfiniment. Dans le domaine de l'ordonnancement, cette notion d'équité qui est assez vague a été interprétée sous différentes formes [61, 75, 90, 91, 120]. Dans cette thèse nous utiliserons la définition formelle introduite par Zhao et Sakellariou dans [120], celle-ci étant adaptée aux ordonnancements multiDAGs. Ces derniers considèrent qu'un ordonnancement multiDAGs est équitable si les dégradations du *makespan* de tous les DAGs en concurrence, par rapport à leurs *makespans* respectifs si chaque DAG disposait à lui seul de la plate-forme, sont proches.

Aujourd'hui, il n'existe aucun algorithme pour réaliser l'ordonnancement concurrent de DAGs de tâches parallèles. Les études effectuées concernent seulement l'ordonnancement de DAGs de tâches séquentielles [32, 60, 120]. Les auteurs de [120] supposent que les DAGs arrivent

au même instant dans le système et proposent de les combiner avant d'ordonnancer le DAG obtenu à l'aide d'heuristiques existantes. Dans [32], un algorithme dynamique et distribué est présenté pour des processeurs interconnectés via un réseau hiérarchique. Le premier niveau est un réseau à large échelle (WAN) qui relie des réseaux locaux (LAN) de processeurs. L'algorithme proposé restreint l'exécution d'un DAG à l'intérieur d'un LAN et fonctionne en deux phases. Dans la première phase, l'ordonnanceur externe sélectionne le LAN sur lequel un DAG doit s'exécuter. Lors de la seconde phase, l'ordonnanceur interne du LAN alloue des ressources aux tâches de ce DAG. En ce qui concerne l'algorithme proposé dans [60], chaque application est ordonnancée indépendamment des autres. Elle récupère indirectement les informations nécessaires aux placement des tâches en estimant la charge des processeurs.

À notre connaissance, tous les algorithmes d'ordonnancement de DAGs de tâches parallèles qui existent aujourd'hui supposent que l'application ordonnancée dispose de l'intégralité des ressources. Autrement dit, cette application est seule à s'exécuter sur la plate-forme. Or les grilles de calcul en général, et les agrégations de grappes homogènes en particulier sont intrinsèquement partagées et elles permettent l'exécution concurrente de nombreuses applications. Il est donc nécessaire de concevoir des heuristiques d'ordonnancement concurrent de DAGs de tâches parallèles sur les plates-formes hétérogènes afin de mieux gérer le partage des ressources. Cela justifie l'intérêt des travaux de cette thèse dont l'objectif est la conception d'algorithmes d'ordonnancement d'applications constituées de tâches parallèles interdépendantes en tenant explicitement compte du caractère partagé et hétérogène des plates-formes.

2.5 Conclusion

Dans ce chapitre, nous avons vu qu'il existe divers types de plates-formes parallèles. Cette diversité des plates-formes offre alors diverses possibilités de déploiement pour des applications parallèles. Parmi ces plates-formes, les plus économiques à mettre en œuvre sont les architectures à passage de messages telles que les grappes de calcul. De nos jours, de nombreuses institutions mettent en place des grappes de calcul homogènes. De plus, il se développe de plus en plus de grilles de calcul résultant de l'interconnexion de ces grappes homogènes. Nous avons retenu ces agrégations hétérogènes de grappes homogènes en vue de les étudier dans la suite de cette thèse. En effet, ces plates-formes ont été peu étudiées pour l'ordonnancement d'applications parallèles mais offrent néanmoins une très grande puissance de calcul qui les rend de plus en plus attractives pour y exécuter des applications à large échelle.

La seconde partie nous a permis de présenter les applications que nous étudierons dans les chapitres suivants. Il s'agit des applications parallèles mixtes qui sont représentatives de nombreuses applications aujourd'hui. Ces applications qui peuvent être représentées par des DAGs de tâches modelables sont généralement conçues pour être exécutées sur des plates-formes homogènes. Toutefois, elles peuvent tirer profit de la puissance disponible sur les agrégations hétérogènes de grappes homogènes. Pour ce faire nous proposons de restreindre l'exécution d'une tâche à l'intérieur d'une grappe.

Dans la dernière partie de ce chapitre, nous avons constaté que de nombreuses études ont été réalisées pour l'ordonnancement de tâches indépendantes et pour l'ordonnancement de DAGs de tâches séquentielles. Les études concernant l'ordonnancement de DAGs de tâches parallèles ont été effectuées principalement pour les milieux homogènes. Mais le problème de l'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes reste ouvert. D'autre part, l'ordonnancement concurrent de plusieurs applications n'a été étudié que pour des DAGs de tâches séquentielles. Cela nous motive à concevoir des algorithmes d'ordonnancement de DAGs

de tâches parallèles destinées aux plates-formes hétérogènes partagées.

Le but de cette thèse étant de mettre en œuvre des heuristiques d'ordonnancement de DAGs de tâches parallèles sur des grappes hétérogènes de grappes homogènes en nous inspirant des heuristiques existantes, nous étudierons préalablement dans le chapitre suivant, des heuristiques d'ordonnancement de tâches parallèles sur grappes homogènes proposées dans la littératures. Nous apporterons des améliorations à l'une de ces heuristiques que nous aurons choisie avant de l'adapter aux plates-formes hétérogènes.

Chapitre 3

Ordonnancement de graphes de tâches modelables sur grappes homogènes

Sommaire

3.1 Introduction

Dans le chapitre précédent, nous avons vu que pour de nombreuses applications parallèles l'exploitation simultanée du parallélisme de données et du parallélisme de tâches permet d'obtenir de meilleurs temps d'exécution pour ces applications, comparé à l'utilisation du parallélisme de données seul ou du parallélisme de tâches seul [30, 87, 105]. Ces applications parallèles mixtes peuvent être modélisées par des graphes orientés acycliques ou DAGs (*Directed Acyclic Graphs*) dont les nœuds représentent des tâches modelables et les arcs, les relations de précédence entre les tâches.

L'ordonnancement de ces DAGs de tâches modelables est un problème critique qui vise à améliorer les performances des applications considérées et à mieux gérer les ressources disponibles. L'objectif visé est en général de minimiser le *makespan*, ou temps de complétion de l'application, c'est-à-dire le temps mis entre sa date de début d'exécution et sa date de fin d'exécution. Le problème de l'ordonnancement de DAGs de tâches modelables consiste dans un premier temps, lors de la *phase d'allocation*, à allouer à chaque tâche le nombre de processeurs qu'elle utilisera au cours de son exécution. Ensuite, lors de la *phase de placement*, il faut déterminer les processeurs sur lesquels chaque tâche doit être exécutée et l'ordre dans lequel les tâches débuteront leurs exécutions dans le respect des contraintes de précédence. Les algorithmes d'ordonnancement de tâches modelables peuvent être répartis en deux catégories. La première est constituée d'algorithmes où la phase d'allocation et la phase de placement sont indissociables. Ce sont les algorithmes d'*ordonnancement en une étape*. La seconde catégorie regroupe les algorithmes d'*ordonnancement en deux étapes* où l'allocation et le placement sont séparés en deux procédures distinctes.

De nos jours, les grappes homogènes constituent l'une des catégories de plates-formes les plus répandues dans le domaine du calcul parallèle. Du fait de leur homogénéité, ces plates-formes sont adaptées à l'exécution d'applications composées de tâches parallèles. En effet, les processeurs des plates-formes homogènes sont intervertibles et il est plus facile de prédire les performances de tâches modelables sur ces plates-formes. Sur une plate-forme homogène, l'accélération d'une tâche modelable peut être approchée par un simple modèle empirique tel que la loi d'Amdahl où le temps d'exécution dépend seulement du nombre de processeurs utilisés et des caractéristiques de la tâche. Ainsi, de nombreuses études ont été réalisées pour l'ordonnancement de DAGs de tâches modelables dans le cas des grappes homogènes [14, 22, 63, 70, 71, 85, 86, 87, 88, 115]. La figure 3.1 présente un exemple d'ordonnancement en deux étapes d'un DAG de tâches modelables. Notons dans cet exemple qu'à l'issue de la phase d'allocation, chaque tâche peut être représentée par une boîte dont la largeur correspond à l'allocation et la hauteur à une estimation de son temps d'exécution. Par exemple, 3 processeurs ont été alloués à la tâche 4, ce qui conduit à un temps d'exécution de 125 secondes pour cette tâche.

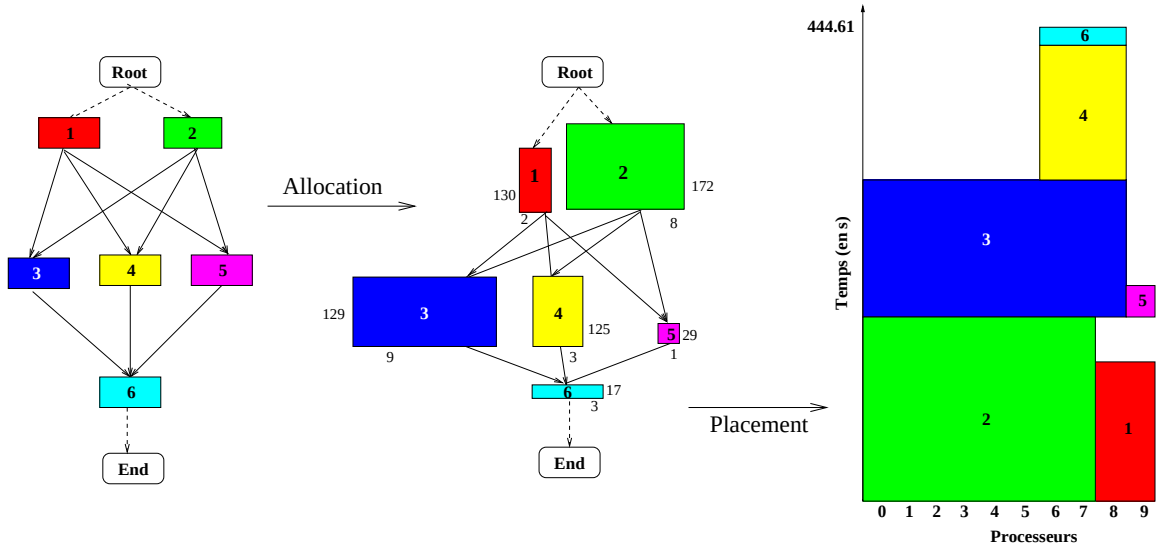

FIG. 3.1 – Exemple d'ordonnancement en deux étapes d'un DAG de tâches modelables sur une grappe homogène de 10 processeurs.

L'ordonnancement de DAGs de tâches modelables non préemptives sur des processeurs identiques avec comme objectif de minimiser le *makespan* est un problème NP-difficile au sens fort pour un nombre de processeurs supérieur ou égal à 2 [43]. Par conséquent, des algorithmes approchés [63, 70, 71] ou des heuristiques [14, 22, 85, 86, 87, 88, 115] sont utilisés pour ordonnancer des DAGs de tâches modelables.

Dans ce chapitre, nous étudierons plusieurs heuristiques d'ordonnancement de DAGs de tâches modelables sur grappes homogènes proposées dans la littérature. Nous choisirons ensuite une de ces heuristiques, qui obtient un bon compromis entre sa complexité et les temps d'exécution des applications ordonnancées, à laquelle nous apporterons différentes améliorations. Enfin, nous comparerons expérimentalement les performances des heuristiques ainsi obtenues à des heuristiques concurrentes.

Dans la section suivante, nous présentons les modèles de plates-formes et d'applications utilisés. Ensuite, après avoir étudié les travaux reliés dans la section 3.3, nous présenterons les améliorations proposées à l'heuristique choisie dans la section 3.4. Enfin, avant de conclure ce chapitre, nous présenterons comment les heuristiques sont évaluées ainsi que les résultats des évaluations dans la section 3.5.

3.2 Modèles de plates-formes et d'applications

3.2.1 Modèle de plates-formes

Dans ce chapitre, nous considérons des grappes homogènes de processeurs. Pour nos expérimentations, les processeurs des grappes étudiées seront reliés entre eux à travers un commutateur (*switch*). Dans cette topologie en étoile, nous ferons l'hypothèse que le commutateur ne subit pas de contentions. En effet, nous supposons que le commutateur est non bloquant et que la bande passante de chacun de ces ports est suffisamment grande par rapport à celle du lien local qui y est connecté. Ainsi le commutateur peut gérer simultanément tous les flux entre les liens qui le connectent aux processeurs sans contention. En revanche les liens reliant chaque processeur au commutateur ont une bande passante limitée et peuvent subir des contentions. Le modèle de communication de ces liens locaux est du N-ports borné. Une machine peut émettre et recevoir plusieurs flux en même temps mais en partageant la bande passante dans la limite de ce qui est disponible. Par la suite, nous noterons P, le nombre total de processeurs de la grappe.

3.2.2 Modèle d'applications

Nous choisissons de représenter une application parallèle mixte par un DAG $G = (\mathcal{V}, \mathcal{E})$ où $\mathcal{V} = \{t_i \,|\, i = 1, \cdots, V\}$ est un ensemble de V nœuds et $\mathcal{E} \subset \{e_{i,j} \,|\, (i,j) \in \{1, \cdots, V\} \times \{1, \cdots, V\}, i < j\}$ est un ensemble de E arcs. Les nœuds représentent les tâches modelables de l'application et les arcs définissent les relations de précédence (dépendances de flots ou de données) entre les tâches. Nous admettons que les tâches des applications étudiées sont modelables et que chaque tâche peut s'exécuter sur un nombre quelconque de processeurs, de manière non préemptive. On associe à chaque arc $e_{i,j}$, la quantité de données (en octets) que la tâche t_i doit transférer à la tâche t_j.

Par définition, une *tâche d'entrée* du DAG n'a aucun prédécesseur et une *tâche de sortie* est sans successeur. Dans cette thèse nous utiliserons deux nœuds fictifs *Root* et *End* pour représenter respectivement les tâches d'entrée et de sortie d'un DAG. Ces deux nœuds fictifs sans coût (de calcul ou de communication) sont ajoutés à chaque DAG, sans perte de généralité, en vue de faciliter leur manipulation par les heuristiques. Une tâche est dite *prête* lorsque

tous ses prédécesseurs ont terminé leur exécution. Le *bottom-level* d'une tâche t, noté $T_b(t)$, représente la longueur du chemin le plus long depuis la tâche t jusqu'à la tâche de sortie, incluant son propre temps d'exécution. $T_b(t)$ est calculé en faisant la somme des temps d'exécution des tâches présentes sur ce chemin. De manière analogue, le *top-level* d'une tâche t, noté $T_t(t)$, est la longueur du chemin le plus long depuis la tâche d'entrée jusqu'à cette tâche, excluant son propre temps d'exécution. Le *chemin critique* est le plus long chemin de DAG (entre les nœuds *Root* et *End*), en termes de temps d'exécution des taches. Une tâche du chemin critique est donc une tâche t telle que la somme $T_b(t)+T_t(t)$ est maximale. Enfin, le niveau de précédence $plev(t)$ d'une tâche t du DAG est un nombre entier positif ($plev(t) \geq 0$) tel que pour tout prédécesseur t' de t, $plev(t') < plev(t)$ et au moins un des prédécesseurs de t a le niveau de précédence $plev(t) - 1$. Nous fixerons le niveau de précédence de la tâche d'entrée à 0 ($plev(Root) = 0$).

Nous noterons $T(t, p(t))$, le temps d'exécution d'une tâche t à laquelle sont alloués $p(t)$ processeurs. La date à laquelle une tâche t est prête sera notée $EST(t)$ (*Earliest Start Time*). Enfin, $T_s(t)$ et $T_f(t)$ représenterons respectivement la date de début d'exécution et la date de fin d'exécution d'une tâche t.

3.3 Travaux reliés

Plusieurs algorithmes garantis ont été proposés pour l'ordonnancement de DAGs de tâches modelables sur plates-formes homogènes. Dans [71], Lepère *et al.* proposent pour des DAGs quelconques, un algorithme dont le *makespan* au pire cas est de $3 + \sqrt{5} \approx 5,236$ par rapport à l'ordonnancement optimal. Jansen et Zhang améliorent ce résultat en proposant un algorithme $100/43 + 100(\sqrt{4349} - 7)/2451 \approx 4,731$ approché [63]. La faiblesse de ces algorithmes approchés issus de la théorie avec une garantie de performance est qu'ils ne tiennent pas explicitement compte des communications (temps de redistributions de données) entre les tâches interdépendantes. Or les tâches des applications parallèles mixtes produisent des données qu'il est nécessaire de redistribuer. Pour certaines applications, les temps de communications dûs à ces redistributions de données sont non négligeables et ils doivent pris en compte. Toutefois ces algorithmes présentent un intérêt majeur du fait qu'ils garantissent une performance au pire cas. En outre, ces algorithmes sont adaptés à des applications composées de tâches à gros grains et pour lesquelles les temps de communications inter-tâches sont négligeables devant les temps de calcul des tâches.

Nous verrons dans le chapitre suivant qu'il est intéressant de comparer les performances de ces algorithmes approchés à celles d'heuristiques pragmatiques non garanties. L'objectif de ce chapitre étant la recherche d'une heuristique pragmatique peu complexe en vue de l'adapter aux plates-formes hétérogènes, nous n'étudierons que des heuristiques non garanties.

Dans [88], Rauber et Rünger s'intéressent à l'ordonnancement de graphes série-parallèle [50] construits par compositions séries et/ou parallèles des tâches de leurs applications. Dans le premier cas, une séquence d'opérations présentant des dépendances de données est placée sur l'ensemble des processeurs. Les tâches de cette séquence sont alors exécutées séquentiellement (les unes après les autres). Dans le second cas, l'ensemble des processeurs est divisé en un nombre optimal de sous-ensembles, déterminé par un algorithme glouton. Lepère *et al.* ont quant à eux mis en œuvre un algorithme approché avec une garantie de performance au pire cas de $4(1 + \varepsilon)$ pour des DAGs de tâches modelables présentant une structure d'arbre [70].

Plusieurs heuristiques en deux étapes, non garanties, ont été proposées pour ordonnancer des DAGs de tâches modelables quelconques sur des ensembles homogènes de processeurs [86, 87]. Dans [87], Ramaswamy *et al.* extraient des DAGs de tâches modelables à partir de codes

séquentiels puis y appliquent leur heuristique d'ordonnancement TSAS (*Two Step Allocation and Scheduling*). TSAS utilise la programmation convexe, rendue possible grâce à la propriété de *posynomialité* des modèles de coût choisis (pour l'estimation des temps d'exécution des tâches et des temps de redistribution des données entre des tâches interdépendantes), ainsi que par certaines propriétés de leur structure de DAGs. Les fonctions posynomiales sont semblables au fonctions polynomiales mais elles n'ont que des coefficients positifs et les exposants sont des nombres réels. Une fonction posynomiale peut être transformée en une fonction convexe par un simple changement de variable. Ainsi, la programmation convexe permet aux auteurs d'obtenir en temps polynomial des allocations dans l'espace des réels qu'ils arrondissent ensuite à des nombres entiers.

Parmi les heuristiques en deux étapes de la littérature, nous nous sommes intéressés plus particulièrement à CPA [86] du fait que d'une part, elle a une faible complexité par rapport aux heuristiques concurrentes et que d'autre part, elle permet d'obtenir des *makespans* relativement bons. En effet dans des environnements de production, en plus d'avoir des heuristiques qui fournissent de bons *makespans*, il est important que ces heuristiques soient peu complexes afin que les temps de calculs des ordonnancements aient un faible impact sur le déploiement d'une application parallèle. Certaines infrastructures, afin d'être plus réactives, peuvent même exiger des heuristiques d'ordonnancement ayant des temps d'exécution négligeables. Nous détaillerons les caractéristiques de l'heuristique CPA dans la section suivante avant de lui proposer différentes améliorations dans la section 3.4.

Il existe également des heuristiques en une étape pour l'ordonnancement des DAGs quelconques de tâches modelables [22, 85, 115]. Dans [22], Boudet *et al.* proposent un algorithme qui interdit la réplication des données. Ils associent à chaque tâche, une liste de configurations de processeurs homogènes qui représente ses allocations possibles. iCASLB [115] et CPR [85] sont des heuristiques itératives où, à chaque itération, les allocations sont mises à jour et les placements sont recalculés afin d'estimer le *makespan*. Le principe de ces deux heuristiques consiste à continuer les itérations tant que le *makespan* peut être réduit. Ces heuristiques sont en général plus complexes que les heuristiques en deux étapes mais elles permettent d'obtenir de meilleurs *makespans*. Nous nous intéresserons à iCASLB récemment proposée par Vydyanathan *et al.*. Nous décrirons cette heuristique dans la section 3.3.3. Elle sera ensuite comparée aux nouvelles heuristiques que nous proposerons.

3.3.1 L'heuristique CPA (*Critical Path and Area-based scheduling*)

Dans leur étude [86], Rădulescu *et al.* remarquent que le temps d'exécution total peut être approché par deux quantités. Tout d'abord, la longueur du chemin critique, qui est déterminée par le plus grand des *bottom-levels* :

$$T_{CP} = \max_{t \in \mathcal{V}} T_b(t). \tag{3.1}$$

Le temps d'exécution total peut aussi être exprimé en fonction de l'aire moyenne qui représente le temps moyen d'utilisation des processeurs :

$$T_A = \frac{1}{P} \sum_{t \in \mathcal{V}} \left(T\left(t, p(t)\right) \times p(t) \right). \tag{3.2}$$

Une borne inférieure du temps de complétion correspond à la plus grande de ces deux quantités, soit $T_p^e = \max\{T_{CP}, T_A\}$.

Le but de l'heuristique CPA est de minimiser T_p^e au terme de sa phase d'allocation. Sachant que T_{CP} diminue alors que T_A croît lorsque le nombre de processeurs alloués aux tâches augmente, on initialise les allocations en partant du cas où T_{CP} est maximal en allouant un processeur à chaque tâche. Ensuite, à chaque itération de la procédure d'allocation, on alloue un processeur de plus à la tâche la plus prioritaire jusqu'à ce que l'on obtienne $T_{CP} \leq T_A$. Cette tâche la plus prioritaire est celle appartenant au chemin critique et dont le ratio $T(t, p(t))/p(t)$ diminue le plus significativement si un processeur supplémentaire lui est attribué. Ce ratio correspond à un compromis entre le choix des tâches dont les temps d'exécution diminuent le plus et le choix des tâches qui ont les plus faibles allocations. Ainsi, si deux tâches présentent la même réduction de leurs temps d'exécution respectifs, alors la tâche ayant la plus faible allocation sera prioritaire. À chaque itération, on choisit d'augmenter l'allocation d'une tâche du chemin critique car le choix d'une tâche n'appartenant pas à ce chemin ne permettra pas de réduire T_{CP}. Le chemin critique peut alors varier au cours de la procédure d'allocation. Dès qu'elle est vérifiée, la condition d'arrêt de cette procédure d'allocation ($T_{CP} \leq T_A$) traduit le fait que T_p^e est proche d'un minimum local ($T_p^e \approx T_{CP} \approx T_A$). La phase d'allocation de CPA est présentée dans l'algorithme 1.

Algorithme 1 Phase d'allocation de CPA

1: **pour tout** $t \in \mathcal{V}$ **faire**
2: $\quad p(t) \leftarrow 1$
3: **fin pour**
4: **tant que** $T_{CP} > T_A$ **faire**
5: $\quad t \leftarrow$ tâche du chemin critique $\mid p(t) < P$ et $\left(\frac{T(t,p(t))}{p(t)} - \frac{T(t,p(t)+1)}{p(t)+1}\right)$ est maximum
6: $\quad p(t) \leftarrow p(t) + 1$
7: $\quad$ Mettre à jour le chemin critique
8: **fin tant que**

Une fois les allocations déterminées, la phase de placement a pour objectif de trouver l'ensemble de processeurs qui doit être utilisé par chaque tâche. Pour cela, la plupart des algorithmes utilisent une heuristique de liste lors de cette seconde phase. Les tâches sont d'abord triées selon une priorité avant d'être placées l'une après l'autre. Dans le cas de CPA, les tâches sont triées dans l'ordre décroissant de leurs *bottom-levels* car plus le *bottom-level* d'une tâche est élevé, plus celle-ci est éloignée de la tâche de sortie qui marque la fin d'exécution de l'application.

Algorithme 2 Phase de placement de CPA

1: **tant que** toutes les tâches ne sont pas placées **faire**
2: $\quad t \leftarrow$ tâche prête $t \mid T_b(t)$ est maximum
3: $\quad$ Placer t sur une configuration de $p(t)$ processeurs qui sera libre au plus tôt
4: **fin tant que**

L'algorithme 2 présente la procédure de placement de CPA. À chaque itération, la tâche prête t ayant le plus grand *bottom-level* est choisie et est placée sur l'ensemble de processeurs comprenant $p(t)$ processeurs qui sera libre au plus tôt. La tâche t étant prête (tous ses prédécesseurs ont terminé leur exécution), l'emplacement des données nécessaires à son exécution est connu. Il est alors possible de tenir compte des temps de redistributions pour estimer sa date de début d'exécution et sa date de fin d'exécution. La date de début d'exécution d'une tâche dépend de la date d'arrivée de la dernière donnée nécessaire à son exécution et de la date de

disponibilité des processeurs choisis. Une fois que la tâche t est placée, sa date de fin d'exécution représente la nouvelle date de disponibilité des processeurs choisis.

Il a été montré que la complexité de CPA est de l'ordre de $O\left(V(V+E)P\right)$. Nous rappelons que V, E et P représentent respectivement le nombre de nœuds du DAG, le nombre d'arcs du DAG et le nombre de processeurs de la plate-forme. À titre de comparaison, la complexité de TSAS [87] est de l'ordre de $O(V^{2.5}PlogP)$ et celle de CPR [85], de l'ordre de $O\left(EV^2P + V^3Plog(V) + V^3P^2\right)$. Bien que TSAS ait une complexité plus élevée par rapport à CPA, les *makespans* obtenus par CPA sont meilleurs que ceux de TSAS. En revanche, au prix d'une complexité très élevée par rapport à CPA, CPR ne fournit que des *makespans* légèrement meilleurs.

CPA permet donc de déterminer de bons ordonnancements avec une faible complexité. C'est une heuristique particulièrement adaptée à l'ordonnancement d'applications parallèles mixtes sur les plates-formes de production où la somme du temps de calcul de l'ordonnancement et du temps d'exécution de l'application doit être la plus faible possible.

3.3.2 L'heuristique MCPA (*Modified Critical Path and Area-based algorithm*)

Les phases d'allocation et de placement étant entièrement découplées dans l'heuristique CPA, celle-ci conduit parfois à des allocations telles que des tâches indépendantes prêtes au même moment seront contraintes de s'exécuter les unes après les autres. En effet, CPA peut conduire à des allocations impliquant un grand nombre de processeurs pour certaines tâches et l'exécution de ces tâches peut être retardée ou elles peuvent retarder des tâches concurrentes. On perd ainsi le bénéfice de l'exécution simultanée de plusieurs tâches. Cela peut conduire à de mauvais *makespans* par rapport à des situations ou l'on favorise l'exécution en parallèle de tâches concurrentes indépendantes.

MCPA [14] est une heuristique d'ordonnancement en deux étapes dérivée de CPA qui tente de préserver au mieux le parallélisme de tâches en restreignant les allocations de certaines tâches. À chaque étape de la phase d'allocation, la tâche t la plus critique qui sera choisie afin d'incrémenter son allocation doit respecter deux conditions. Comme dans CPA, la première condition est que t doit appartenir à l'un des chemins critiques du DAG. Le second critère à respecter est que le nombre total de processeurs alloué à des tâches critiques ayant le même niveau de précédence que t doit être inférieur au nombre total de processeurs de la plate-forme. La procédure d'allocation continue tant que la longueur du chemin critique est supérieure à l'aire moyenne et que l'on trouve à chaque itération, une tâche critique vérifiant les deux critères que nous venons d'énoncer. L'algorithme 3 présente la phase d'allocation de MCPA. On remarque qu'en plus des tâches des chemins critiques qui sont marquées pour participer à la restriction des allocations, la dernière tâche la plus critique précédemment sélectionnée reste également marquée et participe à la restriction des allocations sur son niveau de précédence.

Pour sa phase de placement, MCPA utilise la même heuristique de liste que CPA.

Les auteurs de MCPA ont expérimentalement montré que leur heuristique améliore CPA, avec une complexité du même ordre de grandeur. Mais nous verrons dans nos évaluations que cette heuristique n'améliore que très peu le *makespan* des applications par rapport à CPA. Le critère supplémentaire de choix de la tâche la plus critique semble plutôt adapté certains DAGs réguliers pour lesquels les tâches d'un même niveau de précédence ont les mêmes caractéristiques. Dans le cas de DAGs quelconques, il est peu probable d'obtenir, à une itération donnée de la phase d'allocation, plusieurs tâches critiques appartenant à un même niveau de précédence. En effet, il n'existe en général qu'un seul chemin critique. Ainsi, le second critère qui distingue

MCPA de CPA sera presque toujours vérifié. Dans ce cas, ce critère n'aura aucun effet et les deux heuristiques MCPA et CPA conduiront aux mêmes allocations.

Algorithme 3 Phase d'allocation de MCPA

1: **pour tout** $t \in \mathcal{V}$ **faire**
2: $p(t) \leftarrow 1$
3: Marquer t comme non visitée
4: **fin pour**
5: **tant que** $T_{CP} > T_A$ **faire**
6: $\Psi \leftarrow$ ensemble des tâches critiques
7: Marquer toutes les tâches $t \in \Psi$ comme visitées
8: **pour tout** $t \in \Psi$ **faire**
9: $plev\_Alloc(t) \leftarrow$ nombre total de processeurs alloué aux tâches visitées du niveau de précédence de t
10: **fin pour**
11: $t \leftarrow$ tâche critique $\mid (plev\_Alloc(t) < P)$ et $\left(\frac{T(t,p(t))}{p(t)} - \frac{T(t,p(t)+1)}{p(t)+1}\right)$ est maximum
12: $p(t) \leftarrow p(t) + 1$
13: Marquer toutes les tâches excepté t comme non visitées
14: Mettre à jour les T_t et T_b des tâches
15: **fin tant que**

3.3.3 L'heuristique iCASLB (*iterative Coupled processor Allocation and Scheduling algorithm with Look-ahead and Backfill*)

Contrairement à CPA et MCPA ,l'heuristique iCASLB [115] ordonnance des DAGs de tâches parallèles sur un ensemble homogène de processeurs en une seule étape. Elle suppose que les temps de communications inter-tâches sont négligeables par rapport aux temps d'exécutions des tâches. En effet, les auteurs de iCASLB considèrent que les applications parallèles mixtes sont en général constituées de tâches à gros grain.

À partir d'une allocation initiale, iCASLB réduit itérativement le *makespan* d'un DAG en augmentant à chaque étape le nombre de processeurs alloués à une tâche choisie sur le chemin critique. À chaque itération, l'heuristique recalcule le placement de chaque tâche du DAG et donne une estimation du temps de complétion de l'application. La technique de placement des tâches intègre le *backfilling* conservatif afin de réduire les trous dans l'ordonnancement. iCASLB utilise un mécanisme d'anticipation en effectuant plusieurs itérations successives qui évitent d'arrêter immédiatement la procédure d'ordonnancement lorsqu'un minimum local est atteint pour le *makespan*.

L'allocation initiale de chaque tâche t est déterminée en supposant qu'on allouerait à toute tâche t' potentiellement concurrente à t, sa meilleure allocation $P_{best}(t')$, c'est-à-dire le nombre de processeurs qui minimise le temps d'exécution de t'. On détermine pour cela, l'ensemble $c_G(t)$ composé des tâches potentiellement concurrentes de t. Autrement dit on considère les tâches qui n'ont aucune relation de précédence à t. Cet ensemble est construit en effectuant un parcours en profondeur de G, suivi d'un second parcours sur G^T, le graphe obtenu en inversant le sens des arcs. Si la somme des meilleures allocations des tâches potentiellement concurrentes à la tâche t ($\sum_{t' \in c_G(t)} P_{best}(t')$) est strictement inférieure au nombre de processeurs de la plate-forme, alors l'allocation initiale de la tâche t est donnée par la différence entre le nombre total de processeurs

de la plate-forme et cette somme $(P - \sum_{t' \in c_G(t)} P_{best}(t'))$. Sinon, on alloue initialement un processeur à la tâche t.

Après cette initialisation, tout comme après chaque itération, le placement des tâches est recalculé grâce à l'heuristique de liste $PrBS$ (*Priority-Based Backfill Scheduling*) présentée dans l'algorithme 4. Tant que toutes les tâches ne sont pas placées, $PrBS$ choisie la tâche t non placée ayant le *bottom-level* le plus élevé et cette dernière est placée sur les processeurs qui lui permettront de terminer son exécution au plus tôt sans retarder les tâches plus prioritaires déjà placées (étapes 3-4 de l'algorithme). $PrBS$ détecte donc les trous dans l'ordonnancement en cherchant la date supérieure ou égale à $EST(t)$ à laquelle $p(t)$ processeurs seront disponibles pour une durée nécessaire à l'exécution de t $(T(t, p(t)))$. S'il est prévu que la tâche t ne démarrera pas son exécution aussitôt qu'elle sera prête $(T_s(t) > EST(t))$, alors des pseudo-arcs sont rajoutés entre les tâches plus prioritaires qui retarderont son exécution et t. Ces pseudo-arcs signifient qu'il y a une dépendance entre les tâches due à une limitation des ressources. À l'issue de l'exécution de $PrBS$, on obtient une estimation du *makespan* de l'application pour les allocations courantes et un DAG G' résultant de l'ajout de pseudo-arcs au DAG G. Le *makespan* estimé sl est la date de fin d'exécution de la dernière tâche à terminer son exécution.

Algorithme 4 $PrBS(G, \{(t, p(t)) \mid t \in \mathcal{V}\})$

1: $G' \leftarrow G$
2: **tant que** toutes les tâches ne sont pas placées **faire**
3: $t \leftarrow$ tâche non placée ayant le *bottom-level* le plus élevé
4: $T_s(t) \leftarrow$ date supérieure ou égale à $EST(t)$ à laquelle $p(t)$ processeurs seront disponibles pour une durée $T(t, p(t)$
5: **si** $T_s(t) > EST(t)$ **alors**
6: Sélectionner un ensemble de tâches $t' \in \mathcal{V}$, telles que $T_f(t') = T_s(t)$ et $\sum p(t') \geq p(t)$
7: Ajouter des pseudo-arcs entre ces tâches et t
8: **fin si**
9: **fin tant que**
10: Retourner le *makespan* prédit sl, et G'

Après avoir déterminé les allocations initiales, iCASLB affine de manière itérative les allocation des tâches en choisissant, à chaque étape, la « meilleure » tâche et en augmentant de 1 le nombre de processeurs qui lui est alloué. Cette tâche est une tâche du chemin critique du DAG G' obtenu à l'aide de $PrBS$ à la suite de la dernière itération.

Si on impose qu'à chaque itération, l'augmentation de l'allocation de la meilleure tâche doit réduire le *makespan*, alors on risque d'arrêter les itérations de l'heuristique lorsqu'un minimum local est atteint. iCASLB évite ce problème en effectuant plusieurs itérations successives. Après ces itérations, l'état des allocations qui fournit le *makespan* minimum est retenu comme étant la meilleure solution courante et constitue le point de départ des prochaines itérations. Le nombre des itérations successives est fixé à : $LookAheadDepth = 2 \times \max_{t \in \mathcal{V}}(P - p(t))$.

L'algorithme 5 décrit iCASLB. L'allocation initiale s'effectue de l'étape 1 à l'étape 7. Dans la boucle principale de la procédure (étapes 11 à 36), en partant de la meilleure solution courante, l'algorithme réalise plusieurs itérations successives (16-30) et tente de trouver une nouvelle meilleure solution (25-28). Si ce mécanisme ne conduit pas à une meilleure solution, alors la tâche du chemin critique ayant servi de point de départ est marquée comme étant un mauvais point de départ pour les itérations qui vont suivre. Si un meilleur *makespan* est obtenu, alors toutes les tâches marquées sont démarquées ; cette meilleure allocation est retenue et la recherche d'une meilleure allocation continue à partir de cet état. La procédure est répétée jusqu'à ce que, pour

Algorithme 5 iCASLB

pour tout $t \in \mathcal{V}$ **faire**
  $p \leftarrow P - \sum_{t' \in c_G(t)} P_{best(t')}$ ▷ nombre de processeurs disponibles si on alloue son meilleur nombre de processeur à chaque tâche concurrente
  si $p > 1$ **alors**
    $p(t) \leftarrow \min(P_{best}(t), p)$
  sinon
    $p(t) \leftarrow 1$
  fin si
  $best\_Alloc \leftarrow \{(t, p(t)) \,|\, t \in \mathcal{V}\}$ ▷ au départ, la meilleure allocation est l'allocation initiale
  $(best\_sl, G') \leftarrow PrBS(G, best\_Alloc)$
fin pour
répéter
  $\{(t, p(t)) \,|\, t \in \mathcal{V}\} \leftarrow best\_Alloc$ ▷ partir de la meilleure allocation
  $old\_sl \leftarrow best\_sl$ ▷ et du meilleur placement
  $LookAheadDepth \leftarrow 2 \times \max_{t \in \mathcal{V}}(P - p(t))$
  $iter\_cnt \leftarrow 0$
  tant que $iter\_cnt \leq LookAheadDepth$ **faire**
    $CP \leftarrow$ Chemin critique de G'
    $t_{best} \leftarrow$ Meilleure tâche t de CP avec $p(t) < \min(P, P_{best}(t))$ et t est non marqué si $iter\_cnt = 0$
    si $iter\_cnt = 0$ **alors**
      $t_{entry} \leftarrow t_{best}$ ▷ t_{entry} est le point de départ des itérations successives
    fin si
    $p(t_{best}) \leftarrow p(t_{best}) + 1$
    $A' \leftarrow \{(t, p(t)) \,|\, t \in \mathcal{V}\}$
    $(cur\_sl, G') \leftarrow PrBS(G, A')$
    si $cur\_sl < best\_sl$ **alors**
      $best\_Alloc \leftarrow \{(t, p(t)) \,|\, t \in \mathcal{V}\}$
      $(best\_sl, G') \leftarrow PrBS(G, best\_Alloc)$
    fin si
    $iter\_cnt \leftarrow iter\_cnt + 1$
  fin tant que
  si $best\_sl \geq old\_sl$ **alors**
    Marquer t_{entry} comme mauvais point de départ pour les recherches futures
  sinon
    Retenir cette allocation et démarquer toutes les tâches marquées
  fin si
jusqu'à pour toute tâche $t \in CP$, soit t est marquée soit $p(t) = \min(P, P_{best}(t))$

toute tâche du chemin critique, soit celle-ci est marquée, soit elle a comme allocation son meilleur nombre de processeurs.

La complexité au pire cas de iCASLB est $O(V^3P^2 + VP^2E')$, où $E' \geq E$ est le nombre d'arcs dans G'. Dans leurs évaluations, les auteurs de iCASLB obtiennent que leur heuristique donne de meilleurs *makespans* que CPA et CPR. Mais nous pouvons remarquer que cette heuristique est plus complexe que CPA.

Dans cette thèse, nous avons choisi de partir de l'heuristique CPA et de lui apporter différentes améliorations car elle a un bon compromis entre sa complexité et les *makespans* obtenus. Avant que l'heuristique MCPA ne soit proposée dans la littérature, nous avions proposé nos propres améliorations à CPA. Nous présentons ces améliorations que nous avons apportées à CPA dans la section suivante.

3.4 Améliorations de l'heuristique CPA

Dans cette section, nous proposons deux améliorations à l'heuristique CPA de Rădulescu *et al.* [86]. La première porte sur la phase d'allocation et la seconde sur la phase de placement.

3.4.1 Nouveau critère d'arrêt de la procédure d'allocation

Nous avons constaté expérimentalement que le calcul de l'aire moyenne de CPA est moins pertinent lorsque le nombre de processeurs (P) de la plate-forme est beaucoup plus grand que le nombre de tâches (V) et qu'il existe un surcoût lié à la parallélisation des tâches (par exemple α non nul dans le modèle d'Amdahl). En effet, lorsque ces deux conditions sont réunies, T_A converge très lentement vers T_{CP}. Cela débouche sur des allocations contenant un très grand nombre de processeurs. Or plus on alloue de processeurs à chacune des tâches, plus le risque de ne plus pouvoir exécuter en parallèle certaines tâches concurrentes augmente. En outre, du fait du surcoût lié à la parallélisation, plus on alloue de processeurs à une tâche, plus son efficacité par rapport à l'utilisation des processeurs diminue. Il peut donc s'avérer préférable d'arrêter le processus d'allocation plus tôt et donc de déterminer des allocations plus petites, afin de profiter au mieux du parallélisme de tâches et d'utiliser plus efficacement les ressources. Nous proposons de trouver un compromis qui permet d'arrêter plus vite la procédure d'allocation dans le cas où le nombre de ressources est très élevé en prenant $\min(P, \sqrt{P \times V})$ au lieu de P dans le calcul de l'aire moyenne. Cela nous conduit à redéfinir la notion d'aire moyenne de la façon suivante :

$$T'_A = \frac{1}{\min(P, \sqrt{P \times V})} \sum_{t \in \mathcal{V}} (T(t, p(t)) \times p(t)). \tag{3.3}$$

De manière empirique, nous avons choisi la moyenne géométrique entre le nombre de processeurs et le nombre de tâches, $\sqrt{P \times V}$, car elle obtient un bon compromis entre l'utilisation des ressources et le *makespan* des applications lorsque P devient très grand devant V. Le fait de prendre la valeur minimale entre P et $\sqrt{P \times V}$ entraîne que pour $P \leq V$, les allocations sont les même qu'avec CPA (T'_A est égale à l'aire moyenne T_A utilisée dans CPA). Pour $P \gg V$, cette nouvelle définition augmente la pente de croissance de l'aire moyenne. La relation $T_p^e \approx T_{CP}$ reste toujours valable à la fin de la procédure d'allocation.

Nous verrons dans le chapitre 5 qu'une estimation du taux d'utilisation de la plate-forme (rapport de la puissance moyenne utilisée sur la puissance totale de la plate-forme) est $\beta' = T_A/T_{CP}$. Puisqu'on a $T'_A = \sqrt{P/V} \times T_A$ lorsque le nombre de processeurs est supérieur au nombre de tâches, la condition d'arrêt de notre nouvelle procédure d'allocation $T_{CP} \approx T'_A$ équivaut alors à réduire le taux d'utilisation de la plate-forme en le fixant à $\sqrt{V/P}$ quand $P > V$.

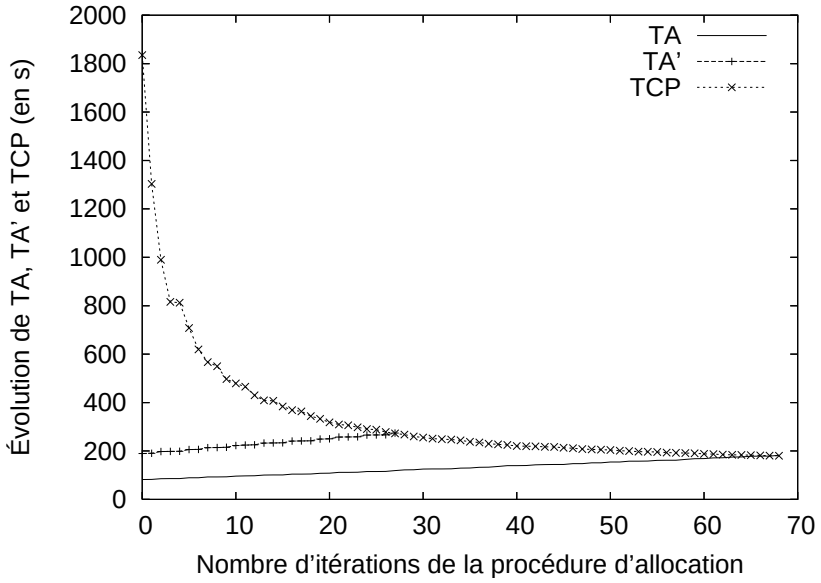

FIG. 3.2 – Exemple d'évolution de T_A, T'_A et de T_{CP} dans la procédure d'allocation de CPA et dans la nouvelle procédure d'allocation pour un DAG de 6 tâches sur une grappe de 30 processeurs.

La figure 3.2 montre un exemple d'évolution de T_{CP} et de T_A en utilisant CPA ainsi que celle de T'_A dans la nouvelle procédure d'allocation lors de l'ordonnancement d'un DAG de 6 tâches (voir DAG de la figure 3.1) sur une grappe contenant 30 processeurs. Les temps d'exécution des tâches sont modélisés par la loi d'Amdahl et leurs portions non parallélisables respectives ont été tiré aléatoirement entre 0% et 20%. Dans cet exemple, le nombre de processeurs ($P = 30$) est beaucoup plus grand que le nombre de tâches ($V = 6$). On note une importante réduction du nombre total de processeurs alloués lorsqu'on utilise la nouvelle procédure d'allocation (33 processeurs = 6 processeurs alloués dès l'initialisation + 27 processeurs supplémentaires) par rapport au nombre total de processeurs alloués avec CPA ($74 = 6 + 68$). Le tableau 3.1 compare, pour le même DAG et la même plate-forme, les allocations des différentes tâches à l'issue des deux procédures. Cette réduction des allocations fait passer la longueur du chemin critique de $180,10$ secondes à $272,58$ secondes mais elle permet de mieux profiter du parallélisme de tâches existant dans le DAG. Ainsi, dans la figure 3.3 qui présente le résultat de l'ordonnancement, nous pouvons observer que l'exécution en parallèle des tâches 1 et 2, puis des tâches 3, 4 et 5 permet d'obtenir un meilleur temps de complétion par rapport à l'ordonnancement de CPA en plus d'une faible consommation de ressources.

Tâche	1	2	3	4	5	6
Allocations avec T_A	3	29	24	8	1	9
Allocations avec T'_A	2	11	11	4	1	4

TAB. 3.1 – Comparaison des allocations obtenues pour un DAG de 6 tâches sur une grappe de 30 processeurs.

Le compromis que nous venons de définir n'est pas toujours optimal du point de vue du temps de complétion des applications mais nous l'avons choisi car il permet surtout de mieux

gérer l'utilisation des ressources. Ainsi, dans la figure 3.2 on observe qu'après 30 itérations lors l'utilisation de la procédure de CPA (avec T_A), les ajouts supplémentaires de processeurs ne permettent pas de réduire significativement T_{CP}.

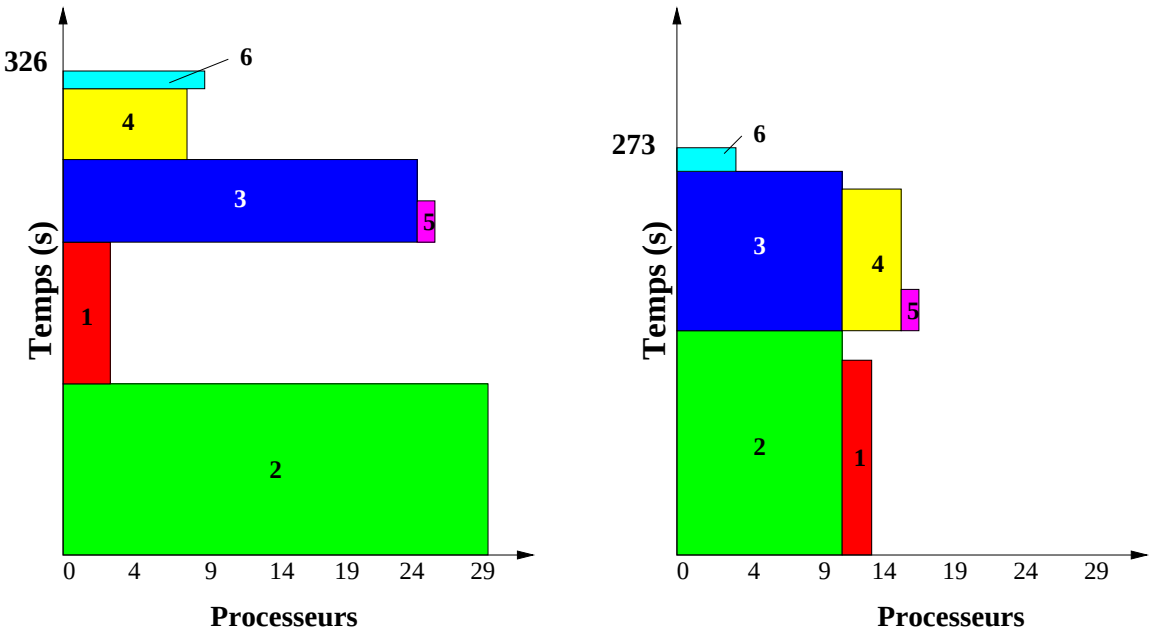

FIG. 3.3 – Ordonnancement du DAG de la figure 3.1 avec CPA (à gauche) et en utilisant la nouvelle allocation (à droite) sur une grappe homogène de 30 processeurs.

3.4.2 Tassage lors du placement

L'objectif premier de cette thèse étant l'ordonnancement de DAGs de tâches modelables sur des plates-formes partagées, nous utiliserons des techniques de placement dynamiques, c'est-à-dire que la décision de placement n'est effectuée qu'une fois la tâche concernée prête.

Lorsqu'on utilise CPA, il peut arriver qu'une tâche prête se mette à attendre qu'une partie des processeurs qui lui sont alloués soient disponibles alors que la majorité des processeurs dont elle aurait besoin le sont déjà. Cette tâche pourrait donc avoir une meilleure date de fin d'exécution si l'on réduisait son allocation de sorte qu'elle puisse démarrer son exécution dès la date où elle est prête.

La technique de *tassage* que nous proposons permet d'éviter cette situation. Elle permet d'améliorer si possible la date de fin d'exécution d'une tâche prête en réduisant le nombre de processeurs qui lui sont alloués. À l'instant où la tâche prête la plus prioritaire est choisie pour être placée, on regarde d'abord s'il est possible de débuter son exécution immédiatement avec son allocation initiale, à savoir si le nombre de processeurs disponibles à cette date est supérieur ou égal à cette allocation. Si cela est possible, alors la tâche est placée sur le nombre de processeurs disponibles dont elle à besoin. En revanche, si le nombre de processeurs disponibles est inférieur à l'allocation déterminée pour cette tâche prête, il faut vérifier si elle pourrait terminer son exécution plus tôt en utilisant seulement les processeurs déjà disponibles plutôt qu'en se mettant en attente que tous les processeurs qui lui sont alloués soient libres avant de démarrer son exécution. Si elle peut effectivement améliorer sa date de fin d'exécution en utilisant seulement les processeurs disponibles, alors une nouvelle allocation est attribuée à la tâche prête

courante et elle est placée sur ces processeurs. Sinon elle est placée selon son allocation initiale et s'exécutera dès que tous les processeurs sur lesquels elle est placée seront libres.

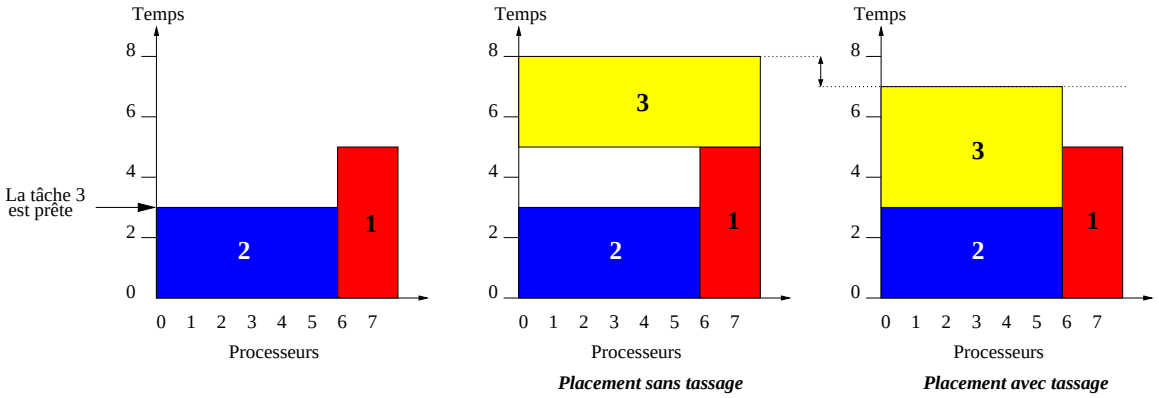

FIG. 3.4 – Illustration du placement avec tassage.

La figure 3.4 illustre un exemple de tassage pour une tâche prête. Le fait de réduire l'allocation de la tâche 3 de 8 à 6 processeurs permet d'obtenir une meilleure date de fin d'exécution pour cette tâche et d'éviter un trou dans l'ordonnancement. Ainsi, tout en menant à une utilisation plus efficace des ressources par la réduction des trous dans l'ordonnancement, le tassage peut permettre d'améliorer le *makespan* des applications. Nous n'avons pas envisagé le cas où l'augmentation de l'allocation d'une tâche pourrait permettre d'améliorer le *makespan*. En effet, les décisions de placement étant prises au fur et à mesure que les tâches deviennent prêtes, nous ne pouvons pas savoir à l'avance si allouer plus de processeurs qu'il n'en était prévu à une tâche risque de retarder ou non l'exécution d'une tâche à venir (le placement de cette dernière n'étant pas encore décidé).

L'utilisation seule de la nouvelle condition d'arrêt (T'_A au lieu de T_A) dans la phase d'allocation, l'application seule de la technique de tassage dans la procédure de placement et la combinaison de ces deux améliorations forment ainsi trois nouvelles heuristiques dérivées de CPA dont nous comparerons les performances face aux heuristiques CPA, MCPA et iCASLB dans la section suivante. Les heuristiques seront également comparées au cas où l'on n'exploite que le parallélisme de données, c'est-à-dire que chaque tâche s'exécute sur la totalité des processeurs de la plate-forme. L'heuristique gloutonne qui servira à exécuter les tâches l'une après l'autre, chaque tâche utilisant la totalité des processeurs, sera nommée DATA.

3.5 Évaluation des heuristiques

Pour l'évaluation de nos heuristiques, nous aurons recours dans cette thèse à des simulations afin d'explorer une large variété de plates-formes et d'applications. L'utilisation de simulations nous permet également de garantir la reproductibilité des expériences et des conditions expérimentales. Pour cela nous utilisons *SimGrid* [27, 95], une boîte à outils conçue pour la simulation de grilles de calcul et la mise en œuvre d'applications distribuées.

Les simulations sont effectuées sur des plates-formes et des DAGs générés aléatoirement en faisant varier plusieurs paramètres permettant de fixer certaines propriétés.

3.5.1 Plates-formes simulées

Le nombre total de processeurs de chaque plate-forme est tiré aléatoirement entre 16 et 512. Ces bornes ont été choisies de manière à étudier plusieurs situations allant de cas où le nombre de processeurs de la plate-forme est inférieur aux nombre de tâches du DAG jusqu'aux cas où le nombre de processeurs est très élevé par rapport au nombre de tâches. Les liens entre les processeurs et le commutateur sont de type Giga Ethernet (bande passante = $1Gb/s$ et latence = $100\mu s$). La vitesse des processeurs (en $Gflop/s$) est fixée à 0,25, 0,5, 0,75 ou 1. Ces valeurs ont été choisies car elles sont proches des caractéristiques des composants standards de certains parcs informatiques existants à ce jour. Pour chaque vitesse de processeurs, nous générons 10 échantillons de plates-formes, soit un total de 40 grappes homogènes.

Dans notre simulateur, nous utilisons le modèle de tâches parallèles *ptask_L07* de *Simgrid* qui tient compte du comportement du protocole TCP/IP dans les communications. Dans *SimGrid*, pour un flux de communication donné entre deux hôtes (processeurs) distants, la bande passante atteignable sur un lien est le minimum entre la bande passante de ce lien et $\gamma/(2 \times \lambda)$, γ étant la taille maximale de la fenêtre TCP (en octets) et λ la latence totale (en secondes) entre les deux hôtes. Pour nos expérimentations, la valeur maximale de la fenêtre TCP est fixée à 4Mo comme elle l'est actuellement sur les grappes de la plate-forme expérimentale Grid'5000 [25, 55].

3.5.2 Applications simulées

Nous supposons que les tâches modelables considérées traitent des nombres réels double précision et que les processeurs ont chacun une mémoire de 1 Go. Les graphes de tâches sont générés en tirant une taille de données M, multiple de 1024 (1 Ko), entre les valeurs limites 2048 et 11268, ce qui correspond au plus à 1 Go d'occupation mémoire par tâche. Ensuite, la complexité de toutes les tâches d'un même DAG est de la forme $a \cdot M$ (ce qui correspond par exemple à la complexité du traitement d'une image de taille $\sqrt{M} \times \sqrt{M}$), $a \cdot MlogM$ (tri d'un tableau de M éléments), ou $M^{3/2}$ (multiplication de deux matrices de tailles $\sqrt{M} \times \sqrt{M}$), ou est tirée au sort parmi ces trois complexités où a est un nombre tiré aléatoirement entre 2^6 et 2^9 pour tenir compte du fait qu'un algorithme effectue généralement plusieurs opérations. Le volume des transferts de données entre deux tâches est égal à M, où M est relatif à la tâche qui génère le transfert.

Les DAGs générés seront composés soit entièrement de tâches représentées par le modèle d'accélération d'Amdahl (voir l'équation 2.1 du chapitre 2), soit entièrement de tâches modélisées par notre modèle complémentaire de tâches avec communications intra-tâches (voir l'équation 2.4 du chapitre 2). Pour les DAGs où les tâches sont représentées par le modèle d'Amdahl, la portion non parallélisable de chaque tâche (α) est un nombre aléatoire pris entre 0 et 0,25. En ce qui concerne les autres DAGs, le volume de données à échanger à l'intérieur d'une tâche sera M.

Nous évaluerons les performances des heuristiques pour deux principales classes de DAGs. La première classe regroupe des DAGs « réguliers » pour lesquels toutes les tâches d'un même niveau de précédence ont des coûts de calcul identiques et où tous les arcs entre les tâches de deux niveaux de précédence consécutifs sont pondérés par le même volume de données à redistribuer. La propriété principale de ces DAGs est que tout chemin entre le nœud d'entrée et le nœud de sortie est un chemin critique.

Parmi les DAGs réguliers, nous avons généré des DAGs aléatoires comprenant 10, 20 ou 50 tâches. Trois paramètres permettent de faire varier leur forme. Le premier de ces paramètres est la largeur des DAG qui sera fixé à 0,2, 0,5 ou 0,8. Une largeur élevée (proche de 1) correspond à

un graphe compact avec beaucoup de parallélisme de tâches. À l'inverse, une largeur proche de 0 donnera un DAG filiforme. Le second paramètre est la régularité entre niveaux de précédence qui sera égale à 0,2 ou 0,8. Dans un graphe dont la régularité est faible, la différence entre le nombre de tâches sur deux niveaux consécutifs peut être très importante. Notons que ce paramètre n'a aucun lien avec le terme « régulier » que nous employons pour désigner la catégorie des DAGs. Le troisième paramètre est la densité qui caractérise le fait que l'on a plus ou moins de relations de précédences entre les tâches de l'application (plus ou moins d'arcs entre deux niveaux). Ce paramètre prendra les valeurs 0,2 ou 0,8.

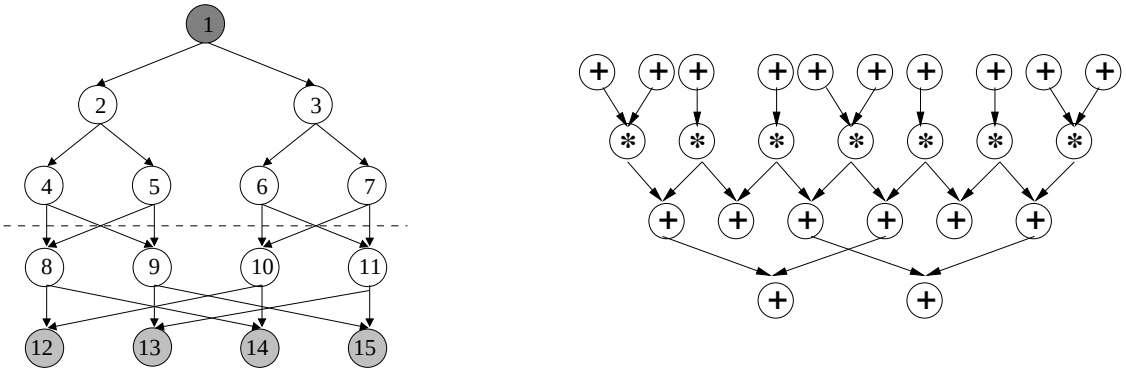

FIG. 3.5 – Exemples de DAGs pour les algorithmes FFT (à gauche) et de Strassen (à droite).

Dans cette même catégorie, nous considérons également deux types de DAGs ayant des formes inspirées d'applications réelles : la transformation de Fourier rapide ou FFT (*Fast Fourier Transformation*) [35] et la multiplication de matrice à l'aide de l'algorithme de Strassen [104]. Le graphe de gauche de la figure 3.5 présente la forme de l'algorithme FFT à 4 points. Ce DAG peut se diviser en deux parties. La première correspond à des appels récursifs et la seconde représente les opérations sous forme de papillon de l'algorithme. Pour n points, il y a $2 \times n - 1$ tâches pour les appels récursifs et $n \times \log_2 n$ tâches pour les opérations en papillon. Le nombre de points n sera égal à 2, 4, 8 ou 16. Les DAGs de type *Strassen* comprennent toujours 25 tâches et ont la forme du graphe de droite de la figure 3.5. En ce qui concerne la génération des quantités de calcul et des volumes de données à communiquer pour les DAGs de types FFT et pour les DAGs de type *Strassen*, nous utiliserons les complexités présentées dans les deux premiers paragraphes de cette section. Ces DAGs ne correspondent donc pas exactement aux applications originales.

La seconde classe comprend des DAGs « irréguliers » de forme quelconque dont les coûts de calcul des tâches sont générés indépendamment. Ces DAGs contiennent 10, 20 ou 50 tâches. En plus de la largeur, la régularité et la densité, nous utilisons un quatrième paramètre pour faire varier la forme de ces DAGs. Il s'agit de la longueur maximale des sauts entre niveaux qui sera fixé à 1, 2 ou 4. Ce paramètre sert à générer des graphes contenant des chemins de longueurs différentes (en termes nombre de nœuds) entre les tâches d'entrée et de sortie.

Pour chaque combinaison des paramètres de génération des DAGs (forme des DAGs, complexité et nombre de tâches), nous générons plusieurs échantillons différents (3 pour les DAGs quelconques et les DAGs aléatoires de la seconde catégorie, 10 pour les DAGs de type FFT et 25 pour les DAGs de type *Strassen*). Au total nous avons généré 1296 DAGs irréguliers, 432 DAGs réguliers quelconques, 160 DAGs de type FFT et 100 DAGs de type *Strassen*. Notons que nous avons généré les DAGs de telle sorte qu'un même DAG peut soit être utilisé avec le modèle d'Amdahl, soit avec le modèle comprenant communications internes.

Pour chaque couple {plate-forme, DAG}, nous lancerons une instance pour chacune des sept

heuristiques à évaluer. Ces sept heuristiques sont DATA, CPA, MCPA, iCASLB, l'amélioration de CPA grâce à la technique de tassage seule, l'amélioration de CPA grâce à l'utilisation de T'_A seule et la combinaison de ces deux techniques.

3.5.3 Métriques

Les simulations étant effectuées sur une large variété de plates-formes et d'applications, il est nécessaire d'utiliser des métriques normalisées pour l'évaluation des performances des heuristiques afin d'obtenir des mesures moyennes pertinentes. Nous utiliserons donc trois métriques normalisées dans nos évaluations.

Nous utiliserons en premier lieu un *makespan relatif* qui s'obtient en faisant le rapport du *makespan* obtenu par l'heuristique à évaluer par le meilleur *makespan* obtenu par l'une des heuristiques concurrentes pour la même instance (même plate-forme et même application), incluant le *makespan* de l'heuristique considérée. Notons que ce *makespan relatif* est toujours supérieur ou égal à 1. Il est égal à 1 si l'heuristique concernée donne le meilleur *makespan* pour cette instance.

La seconde métrique normalisée est l'*accélération* par rapport au meilleur algorithme d'ordonnancement séquentiel (SEQ). Elle mesure le gain en temps d'exécution par rapport à l'ordonnancement des tâches les unes après les autres sur un processeur de la plate-forme considérée.

La troisième métrique normalisée est l'*efficacité* de l'ordonnancement que nous définirons de la manière suivante. Le travail total W effectué durant l'exécution d'une application est la somme des travaux effectués pour l'exécution de chaque tâche, c'est-à-dire le produit du temps d'exécution de cette tâche (en s) par la puissance de calcul utilisée (en $Gflop/s$). SEQ étant l'heuristique qui utilise le plus rationnellement les ressources de calcul, pour une application et une plate-forme données, nous calculons l'efficacité d'une heuristique en faisant le rapport du travail total effectué (par les ressources de calcul) lorsqu'on utilise SEQ par le travail total effectué lorsqu'on utilise l'heuristique à évaluer (*cf.* équation 3.4). Une efficacité élevée (proche de 1) traduit le fait que les ressources de calcul sont rationnellement utilisées dans l'exécution parallèle de l'application. Notons que cette définition de l'efficacité ne tient pas compte des éventuels trous pouvant se produire dans l'ordonnancement. Notre choix est fondé sur le fait que l'ordonnancement peut être implanté afin d'épargner à l'utilisateur le « paiement » des processeurs inutilisés. Sur les plates-formes actuelles, contrôlées par des gestionnaires de ressources, cela peut être réalisé en effectuant plusieurs soumissions de tâches ou réservations de ressources. L'approche brutale qui consiste à réserver le maximum de processeurs nécessaires pour toute la durée de l'exécution d'une application parallèle mixte est très peu économe en ressources pour des DAGs de tâches parallèles.

$$E_{ALGO} = \frac{W_{SEQ}}{W_{ALGO}} \tag{3.4}$$

En plus de ces trois métriques, nous mesurerons le temps d'exécution des heuristiques lorsqu'elles calculent l'ordonnancement d'un DAG.

3.5.4 Résultats des simulations

DAGs réguliers / loi d'Amdahl

Dans un premier temps, nous nous intéressons au comportement des heuristiques pour des DAGs réguliers pour lesquels l'accélération des tâches est modélisée par la loi d'Amdahl.

La figure 3.6 montre les performances moyennes obtenues par les heuristiques CPA, MCPA, iCASLB, l'amélioration de CPA grâce à la technique de tassage seule, l'amélioration de CPA grâce à l'utilisation de T'_A seule et la combinaison de ces deux techniques pour les DAGs aléatoires de cette catégorie d'applications. Le graphique de gauche représente les *makespans* relatifs moyens des six heuristiques et celui de droite présente leurs efficacités moyennes. Certaines des grappes cibles comprenant un grand nombre de processeurs, l'heuristique CPA détermine des allocations qui empêchent dans certains cas l'exécution simultanée de tâches concurrentes, entraînant par conséquent un allongement du *makespan* moyen. Cela est confirmé par les deux améliorations visant à augmenter l'exploitation de la concurrence, MCPA et utilisation de T'_A, l'amélioration la plus significative étant obtenue par notre proposition. Un autre inconvénient des larges allocations de CPA est l'introduction de périodes d'inactivité dans l'ordonnancement, qui peuvent être réduites par l'utilisation de la technique de tassage. En revanche la combinaison de nos deux techniques conduit à de meilleurs résultats que l'utilisation seule de la nouvelle définition de l'aire moyenne mais moins bons que ceux obtenus en utilisant le tassage seul. Nous pouvons en conclure que dans le cas de DAGs réguliers, il est plus intéressant de réduire les allocations bloquantes que de déterminer des allocations plus petites pour l'ensemble des tâches. Enfin, il est à noter que l'heuristique en une étape iCASLB produit les meilleurs ordonnancements (dans $92,8\%$ des cas) mais au prix d'une plus grande complexité.

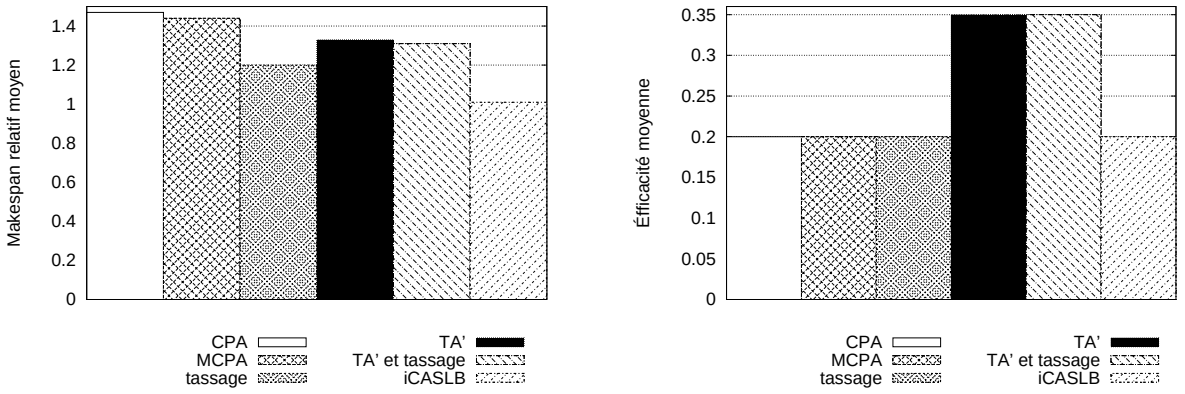

FIG. 3.6 – Performances moyennes des heuristiques pour les DAGs réguliers de forme quelconque.

Le graphique de droite de la figure 3.6 confirme que la diminution des allocations entraînée par l'utilisation de T'_A a également pour effet d'améliorer l'efficacité des ordonnancements. L'efficacité moyenne de nos deux nouvelles heuristiques qui utilisent T'_A lors de leur phase d'allocation est nettement meilleure par rapport à CPA, MCPA, iCASLB et notre heuristique dérivée de CPA qui utilise le tassage seul. Ces quatre heuristiques utilisent toutes de nombreuses ressources et leurs efficacités moyennes sont relativement faibles (environ $0,2$) car chacune d'elles essaie d'utiliser au maximum les ressources disponibles. La réduction des allocations avec T'_A permet d'avoir une efficacité presque deux fois meilleure que celle des autres heuristiques tandis que les *makespans* des heuristiques utilisant T'_A sont en moyenne 30% moins bons que la meilleure valeur obtenue par les différentes heuristiques.

Les figures 3.7 et 3.8 présentent les performances moyennes des heuristiques pour les DAGs de type FFT et *Strassen*. Nous observons les mêmes tendances que pour les DAGs réguliers de forme quelconque. Les DAGs ayant la même forme, les écarts sont moins importants.

La figure 3.9 regroupe les performances de CPA, MCPA et de notre heuristique utilisant T'_A seule en fonction du nombre de processeurs composant les plates-formes. Pour un nombre de

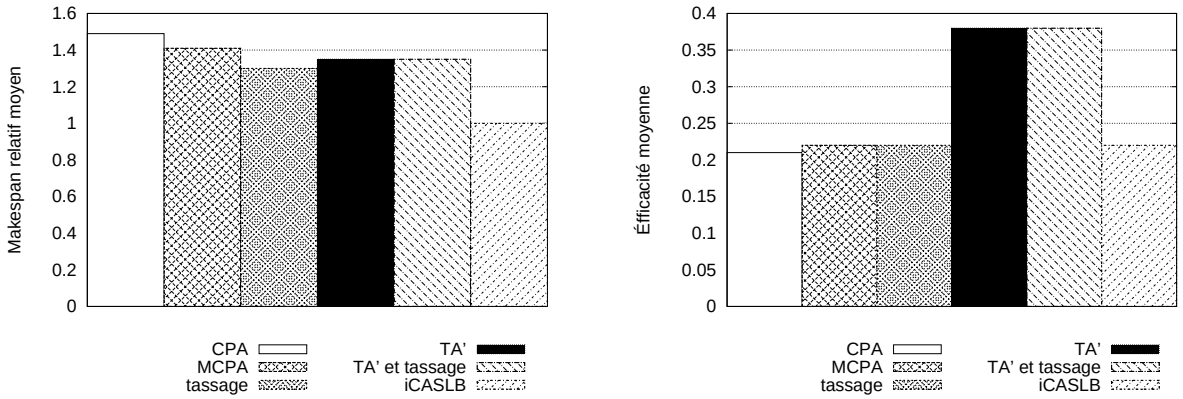

FIG. 3.7 – Performances moyennes des heuristiques pour les DAGs de type FFT.

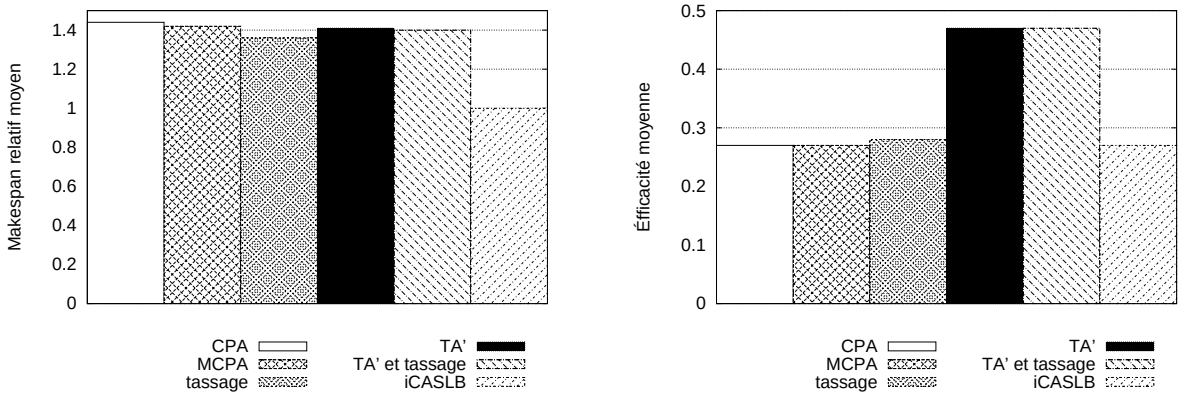

FIG. 3.8 – Performances moyennes des heuristiques pour les DAGs de type *Strassen*.

processeurs compris entre 16 et 50, ce nombre est du même ordre que le nombre de tâches et l'utilisation de T'_A seule donne presque toujours les mêmes allocations que CPA. En effet, l'aire moyenne T_A utilisée par CPA est égale à T'_A si le nombre de processeurs est inférieur ou égal au nombre de tâches et T'_A reste proche de T_A si le nombre de tâches est proche du nombre de processeurs. Cela conduit à des *makespans* relatifs similaires pour CPA et pour notre heuristique utilisant T'_A seule (voir le graphique de gauche). Lorsque le nombre de processeurs de la plate-forme augmente, MCPA et CPA allouent de nombreux processeurs à certaines tâches car la longueur du chemin critique et l'aire moyenne convergent lentement lors de la procédure d'allocation. Or lorsque l'allocation d'une tâche est élevée, son temps d'exécution ne diminue pas significativement quand on lui alloue des processeurs supplémentaires. En revanche l'augmentation de l'allocation d'une telle tâche augmente le risque d'avoir des périodes d'inactivité dans l'ordonnancement. Cela entraîne une dégradation des *makespans* relatifs moyens de CPA et MCPA par rapport à ceux de notre heuristique lorsque le nombre de processeurs augmente, cette dernière permettant d'exécuter plus de tâches en parallèle. L'écart entre les *makespans* relatifs de CPA et MCPA reste constant tandis que celui entre CPA et notre heuristique augmente.

Dans le graphique de droite de la figure 3.9, on observe que plus le nombre de processeurs compris dans la plate-forme augmente, plus l'efficacité des heuristiques diminue. En effet, elles tendent globalement à allouer plus de processeurs aux tâches lorsque le nombre de processeurs de la plate-forme augmente. Il est intéressant de noter que, par rapport à CPA et MCPA, l'efficacité

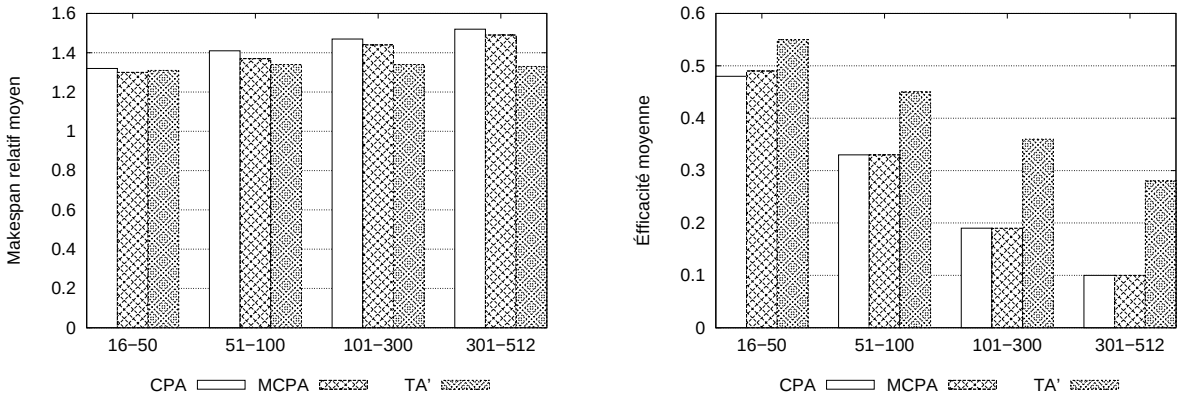

FIG. 3.9 – Performances moyennes des heuristiques en fonctions du nombre de processeurs des plates-formes pour les DAGs réguliers de forme quelconque.

de notre heuristique utilisant T'_A diminue moins vite en fonction du nombre de processeurs de plate-forme, celle-ci ayant été conçue dans l'optique de trouver un bon compromis entre l'utilisation des ressources et le *makespan* des applications. On passe alors d'une efficacité 16% meilleure pour notre heuristique lorsque le nombre de processeurs est compris entre 16 et 50 à une efficacité 180% meilleure quand le nombre de processeurs est compris entre 301 et 512.

Dans la figure 3.10, nous avons regroupé le temps de calcul des ordonnancements par les différentes heuristiques en fonction du nombre de tâches qui composent les applications. L'heuristique iCASLB étant la plus complexe, son temps de calcul des ordonnancements est, comme attendu, plus élevé que celui des autres heuristiques et ce temps croît très rapidement avec le nombre de tâches. Le temps d'exécution de iCASLB étant plus long, cette heuristique peut s'avérer peu pratique pour ordonnancer certains DAGs de grande taille bien qu'elle fournisse de meilleurs *makespans*. En effet il est nécessaire de tenir compte du temps de calcul des ordonnancements en plus de la qualité des ordonnancements sur certaines plates-formes de production. Certains ordonnanceurs se doivent d'être réactifs en ayant des temps de calcul négligeables car ils sont sollicités en permanence. En outre, si un ordonnancement ne sera pas réutilisé à plusieurs reprises, il est préférable de minimiser la somme du temps de calcul de l'ordonnancement et du *makespan* de l'application ordonnancée plutôt que de minimiser seulement le *makespan*. Il est important de noter que le temps de calcul d'un ordonnancement ne dépend pas du temps d'exécution de l'application mais il dépend seulement de la structure de l'application (nombre de tâches et dépendances entre les tâches). Cette somme peut donc être en faveur de nos heuristiques en deux étapes pour des applications comprenant un grand nombre de tâches et pour lesquelles le temps d'exécution est relativement faible.

DAGs irréguliers / loi d'Amdahl

Nous considérons maintenant les DAGs irréguliers pour lesquels l'accélération des tâches est modélisée par la loi d'Amdahl.

La figure 3.11 (gauche) présente les *makespans* relatifs moyens des différentes heuristiques dans le cas de DAGs irréguliers. Nous pouvons constater que ce type d'applications, représentatif des *workflows* scientifiques actuels, le gain en termes d'amélioration du *makespan* obtenu par nos optimisations de CPA est plus important que pour les DAGs réguliers. De plus nos propositions sont désormais compétitives vis-à-vis de iCASLB en termes de réduction du temps de complétion,

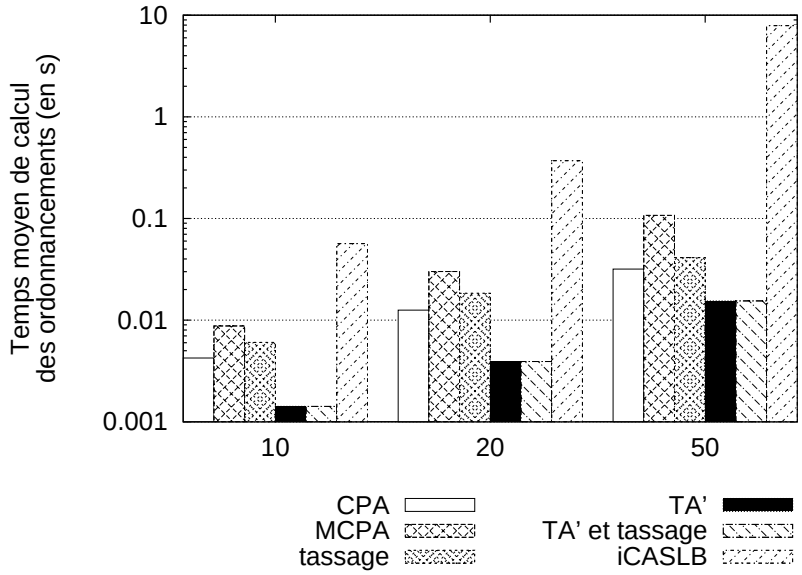

FIG. 3.10 – Temps moyens de calcul des ordonnancements par les heuristiques en fonction du nombre de tâches pour les DAGs réguliers de forme quelconque.

mais bien plus efficaces dans l'utilisation des ressources comme le montre la figure 3.11 (droite). Nos heuristiques modifiant le calcul de l'aire moyenne sont en particulier près de deux fois plus efficaces que iCASLB.

Ces résultats encourageants s'expliquent par la présence d'un seul chemin critique dans ce type DAGs, alors que plusieurs chemins équivalents étaient présents dans les DAGs réguliers. Les tâches de ce chemin critique se voient allouer beaucoup de processeurs et peuvent bloquer l'exécution d'une tâche concurrente qui va à son tour retarder la suite du chemin critique. Cela entraîne l'apparition de périodes d'inactivité importantes dans l'ordonnancement comme le montre la figure 3.12.

Nos optimisations permettent de réduire ce temps d'inactivité, soit en « tassant » les allocations qui sont trop importantes pour être placées au plus tôt, soit en déterminant des allocations plus petites par le biais de la modification de l'aire moyenne. La figure 3.13 montre l'ordon-

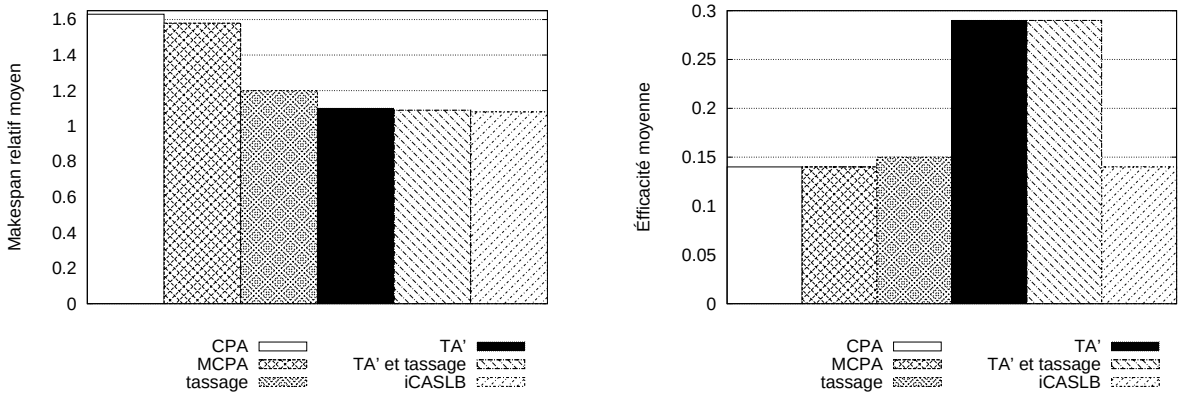

FIG. 3.11 – Performances moyennes des heuristiques pour les DAGs irréguliers.

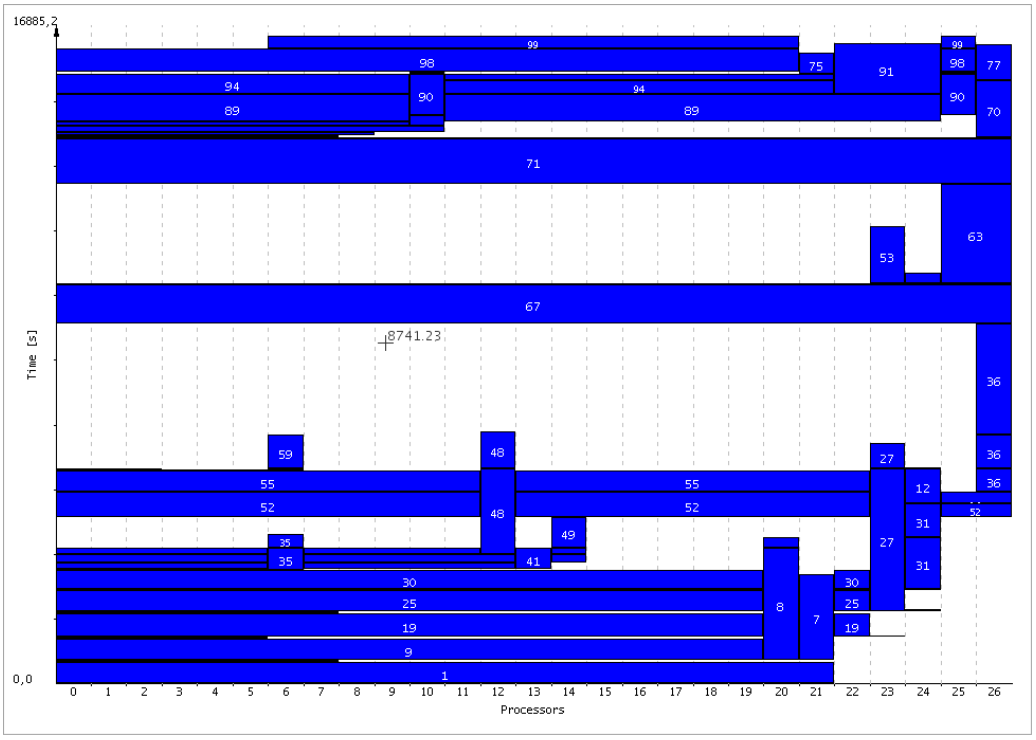

FIG. 3.12 – Exemple d'ordonnancement d'un DAG irrégulier avec CPA.

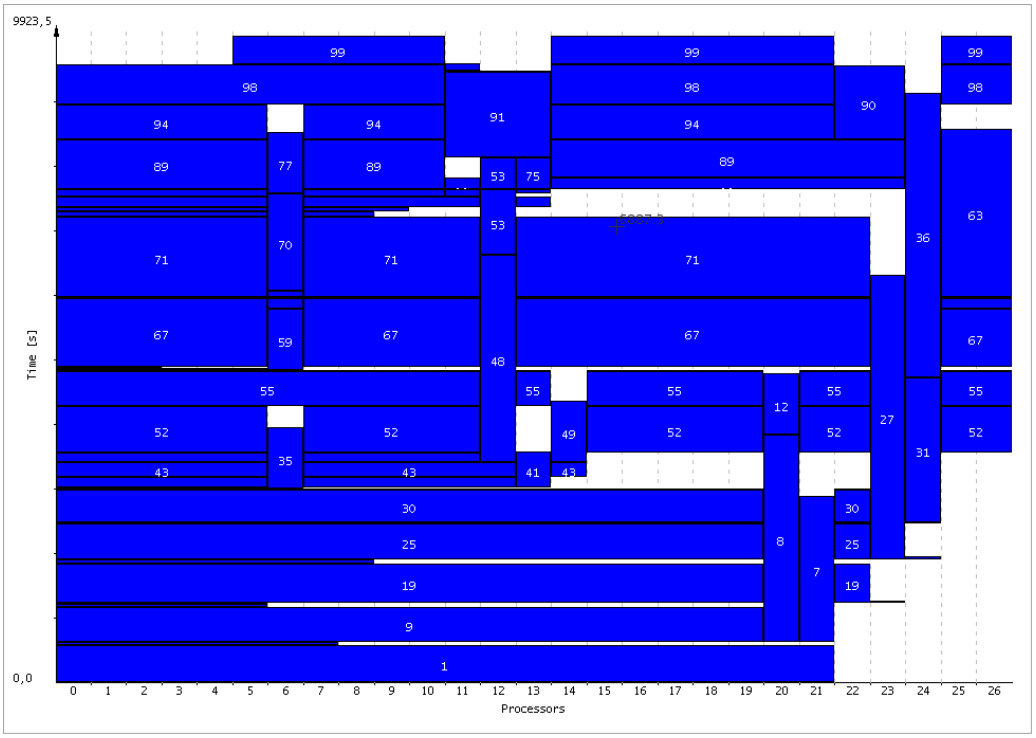

FIG. 3.13 – Exemple d'ordonnancement d'un DAG irrégulier avec notre heuristique utilisant la technique de tassage seule.

nancement obtenu pour le même DAG et la même plate-forme que dans la figure 3.12 mais en appliquant cette fois ci le tassage lors de la phase de placement. Nous pouvons constater que les périodes d'inactivités ont presque totalement disparu et que le *makespan* a été réduit de plus de 40%.

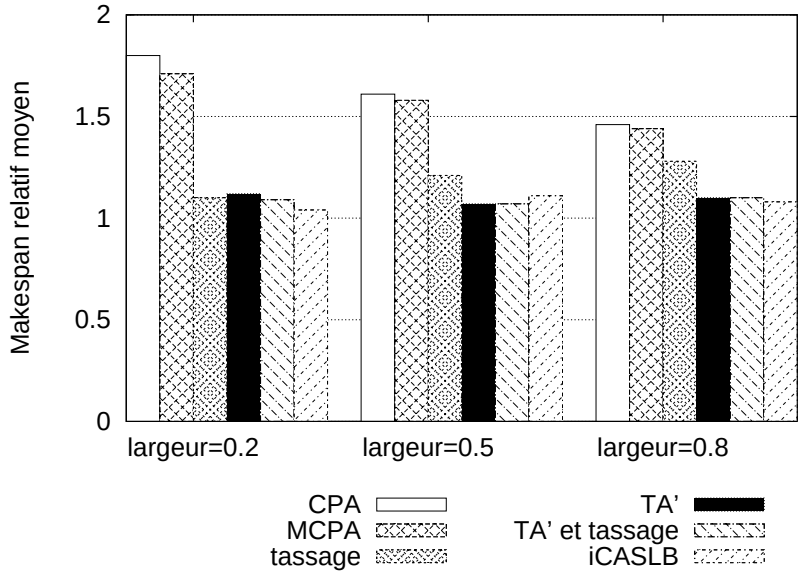

FIG. 3.14 – *Makespans* relatifs moyens des heuristiques regroupés selon la largeur des graphes.

Dans la figure 3.14, nous avons regroupé les *makespans* relatifs des heuristiques selon la largeur des DAGs. Nous observons que les *makespans* relatifs de CPA et MCPA deviennent compétitifs lorsque les applications exhibent plus de parallélisme de tâches. Cette amélioration des *makespans* relatifs est due au fait que l'aire moyenne et la longueur du chemin critique convergent plus rapidement pour des DAGs larges. En effet, à l'initialisation des allocations avec CPA (ou MCPA), la longueur du chemin critique T_{CP} d'un DAG large est en général plus proche de l'aire moyenne T_A par rapport à un DAG peu large. Par conséquent, la condition d'arrêt de la procédure d'allocation $T_{CP} \leq T_A$ est plus rapidement atteinte pour des DAGs larges. Les allocations obtenues sont donc plus petites lorsque les DAGs sont larges et il existe ainsi plus de tâches pouvant être exécutées en parallèle. Il existe alors plus de tâches candidates pour combler les périodes d'inactivité dans l'ordonnancement.

Nous observons également dans la figure 3.14 que le tassage est de moins en moins performant lorsque les DAGs deviennent larges. En effet, les tâches bloquantes ont une plus grande influence dans les DAGs filiformes et le tassage permet d'améliorer plus significativement le *makespan* dans ce cas.

Il nous a semblé intéressant de comparer les performances de nos heuristiques utilisant le parallélisme mixte à celles obtenues en utilisant uniquement le parallélisme de données. Dans la figure 3.15, nous comparons les *makespans* relatifs moyens de l'heuristique DATA à ceux de notre heuristique utilisant T'_A et le tassage. Plus les DAGs sont larges, plus y a de parallélisme de tâches à exploiter, ce que permettent nos heuristiques. Cela se traduit par une augmentation du gain par rapport à l'utilisation seule du parallélisme de données pour atteindre un facteur 3.

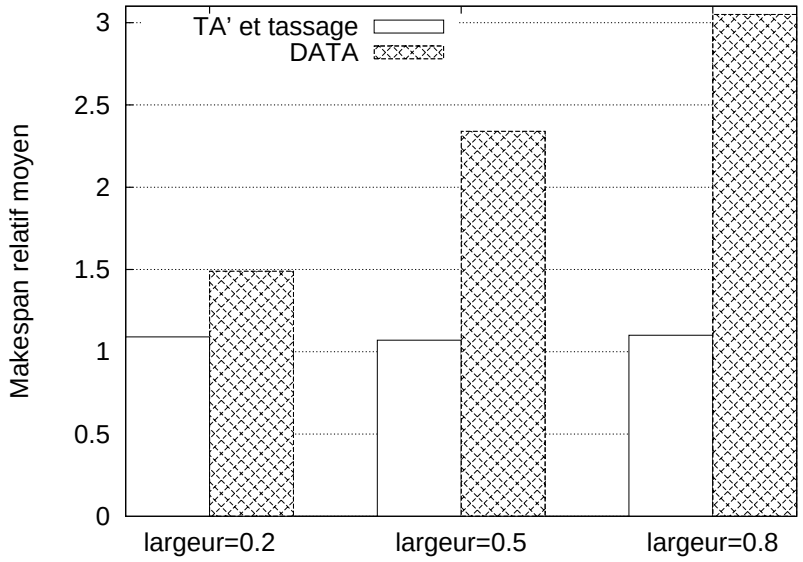

FIG. 3.15 – Comparaison des *makespans* relatifs moyens de DATA à ceux de notre heuristique utilisant T'_A et le tassage en fonction de la largeur des graphes.

Modèle d'accélération avec des communications intra-tâches

Nous allons enfin analyser les résultats obtenus par les heuristiques pour des tâches utilisant le modèle d'accélération avec des communications intra-tâches.

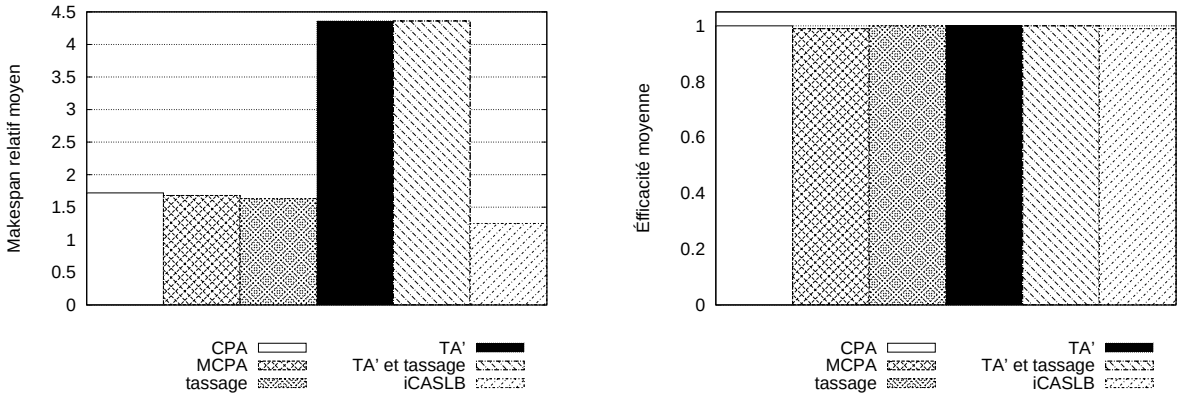

FIG. 3.16 – Performances moyennes des heuristiques pour les DAGs irréguliers.

La figure 3.16 présente les performances moyennes des heuristiques pour les DAGs irréguliers. On peut constater que le *makespan* relatif moyen de DATA est égal à 1, c'est-à-dire que cette heuristique donne toujours le meilleur *makespan* pour les applications considérées. Le modèle utilisé par *SimGrid* permet de recouvrir calculs et communications intra-tâches presque parfaitement. Le surcoût de parallélisation est donc quasi nul. Par conséquent, la diminution du temps d'exécution d'une tâche modelable est donc inversement proportionnelle à l'augmentation du nombre de processeurs alloués. L'allocation la plus efficace pour une tâche comprend alors l'ensemble des processeurs de la grappe. De plus, si toutes les tâches ont la même allocation, notre modèle d'applications suppose qu'il n'y a aucun coût associé aux communications inter-tâches.

DATA produit donc des ordonnancements optimaux comme le souligne le lemme 1.

Lemme 1. *En l'absence de communications entre les tâches, si toutes les tâches modelables t du DAG à ordonnancer sur une plate-forme homogène de P processeurs sont telles que $T(t, p(t)) = T(t, 1)/p(t)$, alors un ordonnancement optimal pour la minimisation du makespan consiste à allouer à chaque tâche P processeurs ($p(t) = P \,\forall\, t$).*

Démonstration. Le temps d'exécution des tâches étant inversement proportionnel aux nombres de processeurs qui leurs sont alloués, on a :

$$T_A \;=\; \frac{1}{P}\sum_{t\in\mathcal{V}}\left(T\left(t,p(t)\right)\times p(t)\right) \;=\; \frac{1}{P}\sum_{t\in\mathcal{V}}T\left(t,1\right).$$

L'aire moyenne qui représente une borne inférieure du *makespan* reste donc constante quelles que soient les allocations des tâches. Cette aire moyenne est donc une borne inférieur du *makespan* pour n'importe quel ordonnancement. Si on alloue P processeurs à chaque tâche, le *makespan* du DAG sera :

$$sl \;=\; \sum_{t\in\mathcal{V}}T\left(t,P\right) \;=\; \sum_{t\in\mathcal{V}}\frac{T\left(t,1\right)}{P} \;=\; T_A.$$

Le *makespan* obtenu étant égal à la borne inférieure de tous les *makespans* possibles, cet ordonnancement est optimal. □

En outre, la solution donnée par DATA a une complexité proche de 0, puisqu'il n'y a pas de prise de décision évoluée. Il n'y a donc aucun intérêt à essayer de réaliser des ordonnancements avec des heuristiques plus sophistiquées, pour des applications pour lesquelles il n'existe pas de surcoût lié à la parallélisation des tâches. Dans la suite de cette thèse, nous n'utiliserons plus ce modèle où il existe des communications internes. Nous nous restreindrons à l'utilisation du modèle d'Amdahl car il exhibe un surcoût lié à la parallélisation des tâches, grâce à l'utilisation de la notion de portion non parallélisable.

Toutefois, il faut remarquer que le modèle utilisé par *SimGrid* n'est pas représentatif de toutes les applications, le recouvrement parfait entre calculs et communications n'étant que rarement possible. Il serait donc intéressant d'étudier les heuristiques dans un contexte où les communications internes ont un impact. Mais cette étude n'a pas été réalisée dans cette thèse.

Bien qu'il ne soit pas nécessaire d'utiliser des heuristiques aussi évoluées que iCASLB ou CPA pour ordonnancer des DAGs pour lesquels le surcoût lié à la parallélisation des tâches est négligeable, il est intéressant d'analyser les résultats obtenus par les heuristiques étudiées. On remarque que les *makespans* relatifs de nos heuristiques utilisant T'_A sont élevés par rapport à ceux de CPA et MCPA. La raison de cette mauvaise performance est que la longueur du chemin critique T_{CP} diminue plus rapidement et converge très vite vers l'aire moyenne modifiée T'_A lorsque le surcoût lié à la parallélisation est faible. En effet, l'allocation d'un processeur supplémentaire à une telle tâche diminue plus significativement son temps d'exécution. Cela débouche à des allocations plus petites par rapport à des tâches où le surcoût lié à la parallélisation est non négligeable. La nouvelle condition d'arrêt des allocations ne permettant pas d'utiliser ici suffisamment de ressources, les *makespans* obtenus sont les plus élevés. En revanche, nous observons que l'application de notre technique de tassage seule améliore pour ces DAGs également, le *makespan* des applications par rapport à CPA et MCPA. De plus, malgré la présence de communications entre les tâches, le tassage seul obtient en moyenne des *makespans* seulement $1,6$ fois supérieurs aux solutions optimales données par DATA.

3.6 Conclusion

Dans ce chapitre, nous avons étudié différentes heuristiques de la littérature qui permettent d'ordonnancer des DAGs de tâches modelables sur des grappes homogènes. Parmi celles-ci nous avons retenue CPA, une heuristique en deux étapes car elle allie un bon compromis entre sa complexité et les *makespans* obtenues pour les applications ordonnancées et nous avons apporté deux améliorations à cette dernière. La première amélioration porte sur la phase d'allocation de CPA et la seconde concerne la phase de placement. Dans la phase d'allocation, nous avons redéfini la notion d'aire moyenne afin d'arrêter plus tôt les allocations de processeurs et de profiter au mieux de l'exécution en parallèle de tâches concurrentes. Cette amélioration a pour but de trouver un compromis entre l'utilisation des ressources et le *makespan* des applications pour lesquelles il existe un surcoût lié à la parallélisation des tâches. Dans la phase de placement, nous avons mis en place une technique de tassage dont le but est de réduire le temps d'attente d'une tâche prête en diminuant son allocation si cela est nécessaire. Nous avons ainsi obtenu trois nouvelles heuristiques qui utilisent respectivement le tassage seul, la réduction des allocations seule ou la combinaison de ces deux techniques.

Nous avons comparé grâce à de nombreuses simulations les performances de nos nouvelles heuristiques avec trois heuristiques de la littérature. Ces heuristiques sont CPA, MCPA qui représente une amélioration concurrente apportée à CPA, et l'heuristique iCASLB qui est une heuristique itérative en une seule étape. Les évaluations antérieures à cette étude ont montré que CPA et MCPA sont, parmi les heuristiques en deux étapes, celles qui fournissent les meilleurs temps de complétion en plus d'avoir des complexités relativement faibles. L'heuristique iCASLB obtient quant à elle de meilleurs *makespans* par rapport à toutes les heuristiques auxquelles elle a été comparée parmi lesquelles figure CPA. Mais elle reste relativement complexe et peut s'avérer moins pratique que les heuristiques en deux étapes pour l'ordonnancement sur des plates-formes de production.

Les évaluations des heuristiques ont montré que malgré sa faible complexité, l'application de la technique de tassage seule à CPA fournit toujours de meilleurs temps de complétion que CPA et MCPA et les *makespans* obtenus sont relativement proches de ceux de iCASLB. iCASLB améliore légèrement le *makespan* des applications par rapport au tassage seul mais nous avons mesuré que le temps d'exécution de cette heuristique est relativement élevé et qu'il croît très rapidement avec le nombre de tâches modelables contenues dans l'application à ordonnancer. Pour tous les DAGs où il existe un surcoût lié à la parallélisation des tâches, la réduction des allocations lors de la phase d'allocation aboutit à un meilleur compromis entre l'utilisation des ressources et le *makespan* des applications. Tout en ayant de biens meilleures efficacités par rapport à toutes les autres heuristiques, nos deux heuristiques qui exploitent la réduction des allocations fournissent de meilleurs *makespans* que CPA et MCPA pour ces DAGs. Nous obtenons même, avec toutes nos heuristiques, des *makespans* quasiment identiques à ceux de iCASLB pour des DAGs irréguliers où le surcoût lié à la parallélisation des tâches est non négligeable. Toutefois pour des DAGs où le surcoût lié à la parallélisation des tâches est faible, comme dans le modèle avec des communications internes aux tâches, la réduction des allocations donne de mauvais *makespans* par rapport aux autres heuristiques. Pour ces applications, il est préférable d'exploiter uniquement le parallélisme de données en exécutant chaque tâche sur la totalité des processeurs de la plate-forme. Le type d'application associé à ce modèle n'est pas forcément le plus courant. Par exemple, même les applications asynchrones utilisant la bibliothèque de passage de messages MPI ne recouvrent pas totalement les communications par les calculs. Elles sont souvent obligées de bloquer sur des barrières de synchronisation.

Avant de proposer des heuristiques d'ordonnancement concurrent de DAGs de tâches mo-

delables sur des agrégations hétérogènes de grappes homogènes, nous adaptons dans le chapitre suivant, les heuristiques que nous venons de concevoir à ce type de plates-formes. Tout comme dans ce chapitre, nous supposerons que chaque DAG ordonnancé peut disposer exclusivement de toutes les ressources de la plate-forme considérée. Nous proposerons également un algorithme garanti pour l'ordonnancement de DAGs de tâches modelables sur ces plates-formes multi-grappes auquel nous comparerons une de nos heuristiques non garantie.

Chapitre 4

Ordonnancement de graphes de tâches modelables sur plates-formes multi-grappes

Sommaire

4.1 Introduction

Grâce à l'amélioration des technologies des réseaux informatiques et des intergiciels, de nombreuses grilles de calcul se sont développées en agrégeant plusieurs grappes de calcul au sein d'une institution ou entre plusieurs institutions. Ces agrégations de grappes homogènes offrent de nombreuses ressources et permettent ainsi de déployer des applications parallèles à des échelles sans précédent.

La latence entre deux nœuds d'un réseau est en partie liée à la distance entre ces nœuds. Cela implique que la latence entre des nœuds géographiquement distants est relativement élevée. Or, les tâches qui composent les applications parallèles mixtes sont en général plutôt adaptées à un ensemble de processeurs regroupés au sein d'un réseau à faible latence, ces tâches pouvant nécessiter de nombreuses communications ou opérations de synchronisation entre les processeurs utilisés. Le besoin de limiter ces temps de communications ou de synchronisations est sans doute l'une des raisons pour lesquelles la plupart des études sur l'ordonnancement d'applications parallèles mixtes n'ont été effectuées que pour des grappes homogènes [14, 70, 71, 85, 86, 87, 88, 115]. En outre, du fait que les ressources peuvent être intervertibles sur une grappe homogène, il est relativement aisé d'estimer, sur une telle plate-forme, le temps d'exécution d'une tâche parallèle en fonction du nombre de processeurs utilisés.

Les applications parallèles mixtes originellement destinés aux plates-formes homogènes peuvent exploiter la puissance disponible sur les agrégations hétérogènes de grappes homogènes. Pour ce faire nous proposons de restreindre l'exécution d'une tâche à l'intérieur d'une grappe. Non seulement cette restriction limitera les temps de communications intra-tâches (si celles-ci existent) car la latence du réseau à l'intérieur d'une grappe est relativement faible, mais aussi elle permettra d'estimer plus facilement les temps d'exécution des tâches, chaque tâche utilisant un ensemble homogène de processeurs.

Les agrégations hétérogènes de grappes homogènes sont donc devenues attractives pour exécuter des DAGs de tâches parallèles. Cependant, l'état de l'art présenté dans la section 2.4.2 nous a révélé que ces plates-formes hétérogènes ont été peu étudiées pour l'ordonnancement des applications parallèles mixtes. Les études effectuées pour l'ordonnancement des applications de tâches dépendantes pouvant être représentées par des DAGs peuvent être classées en deux principales catégories. D'une part, l'ordonnancement des DAGs a été largement étudié dans le cas des tâches séquentielles, aussi bien pour des plates-formes homogènes que pour des plates-formes hétérogènes. De nombreux algorithmes d'ordonnancement de liste ont été proposés pour ces DAGs de tâches séquentielles [47, 67, 68, 94, 111, 116]. D'autre part, plusieurs algorithmes ont été proposés pour ordonnancer des DAGs de tâches parallèles sur des ensembles homogènes de processeurs [14, 70, 71, 85, 86, 87, 88, 115]. À l'instar des heuristiques que nous avons proposées dans le chapitre 3, la majorité de ces algorithmes se déroulent en deux étapes [14, 63, 70, 71, 86, 87]. Le problème de l'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes reste néanmoins un problème ouvert.

		Plates-formes	
		Homogènes	*Hétérogènes*
Type de Parallélisme	*Tâche*	Algorithmes d'ordonnancement de liste	⇒ Algorithmes d'ordonnancement de liste
	Mixte	⇓ Algorithmes en deux étapes	⇒ ⇓ **Problème ouvert**

Tab. 4.1 – Deux approches pour concevoir des algorithmes d'ordonnancement de tâches parallèles en milieu hétérogène.

Deux approches orthogonales présentées dans le tableau 4.1 peuvent servir à mettre en œuvre des algorithmes d'ordonnancement de DAGs de tâches parallèles sur les plates-formes hétérogènes à partir des travaux existants. La première approche, utilisée par l'heuristique MHEFT [26]

inspirée de l'heuristique de liste HEFT [111], consiste à adapter aux tâches parallèles des algorithmes conçus pour l'ordonnancement de DAGs de tâches séquentielles en milieu hétérogène. L'autre approche complémentaire est de gérer l'hétérogénéité dans des algorithmes en deux étapes d'ordonnancement de tâches parallèles en milieu homogène. La seconde approche étant jusqu'alors inexplorée, nous proposons dans ce chapitre de nouvelles heuristiques qui adaptent aux grappes hétérogènes de grappes homogènes les heuristiques en deux étapes que nous avons présentées dans le chapitre précédent. Nous proposons également des améliorations à l'heuristique MHEFT en vue de comparer les deux approches.

Avant de considérer des plates-formes très hétérogènes, nous allons tout d'abord proposer et comparer deux algorithmes pour l'ordonnancement de DAGs de tâches parallèles sur des plates-formes multi-grappes quasi homogènes. En effet, de nombreuses plates-formes multi-grappes existantes exhibent des sous-ensembles constitués de nombreuses ressources qui sont pratiquement homogènes sur le plan de la vitesse des processeurs. Le premier de ces deux algorithmes est une heuristique pragmatique qui adapte aux plates-formes multi-grappes une de nos améliorations de l'heuristique CPA proposée dans le chapitre précédent. Le second est un algorithme avec une garantie de performance. Plusieurs algorithmes avec une garantie de performance ont été développées pour ordonnancer des tâches parallèles sur des grappes homogènes [63, 71, 73, 112] ou sur des grappes hiérarchiques de machines SMPs [44, 45]. Mais aucune étude n'a été effectuée en vue de concevoir des algorithmes garantis pour l'ordonnancement de DAGs de tâches parallèles sur plates-formes multi-grappes.

Il est intéressant de confronter des heuristiques pragmatiques à des algorithmes garantis issus de la théorie. En général, les heuristiques pragmatiques s'implantent plus facilement et ont une faible complexité mais elles n'offrent aucune garantie sur leur performance au pire cas par rapport au meilleur ordonnancement possible. Les algorithmes issus de la théorie fournissent quant à eux une telle garantie mais cela n'implique pas nécessairement que la performance moyenne soit bonne par rapport aux heuristiques pragmatiques. Nous allons donc réaliser une étude comparative qui constituera sans doute la première fois qu'un algorithme garanti est évalué de manière expérimentale.

Dans la section suivante, nous présentons nos modèles de plates-formes et d'applications. Ensuite, nous proposons et évaluons deux algorithmes d'ordonnancement de DAGs de tâches parallèles pour les plates-formes multi-grappes quasi homogènes dont l'un est un algorithme garanti. La section 4.4 montre comment nous adaptons nos heuristiques proposées dans le chapitre précédent aux agrégations hétérogènes de grappes homogènes et la section 4.5 propose quelques améliorations pour MHEFT. Enfin, avant de conclure ce chapitre, nous évaluons les heuristiques obtenues grâce à de nombreuses simulations.

4.2 Modèles de plates-formes et d'applications

Pour nos applications, nous utiliserons le modèle de DAG de tâches modelables et les notations introduits dans le chapitre précédent (voir la sous section 3.2.2). Nous rappelons que $\mathcal{V}$ représente l'ensemble des tâches du DAG et V est le nombre de tâches du DAG. On notera également $\mathcal{E}$ l'ensemble d'arcs du DAG et E le nombre d'arcs. Enfin $T_b(t)$ désignera le *bottom-level* d'une tâche t du DAG.

En ce qui concerne les plates-formes, nous considérons maintenant des agrégations hétérogènes de grappes homogènes. Ces plates-formes sont représentatives de certaines infrastructures de grilles de calcul réparties entre différentes institutions, qui disposent généralement chacune de grappes homogènes. Des exemples réels d'agrégations hétérogènes de grappes homogènes sont

la grille expérimentale Grid'5000 [21, 55] et la plate-forme néerlandaise DAS-3 [39]. Le projet Grid'5000 a été fondé par l'ACI GRID (Action Concertée Incitative Globalisation des Ressources Informatiques et des Données) du ministère français de la recherche et de l'éducation. Son objectif est de construire une plate-forme expérimentale à large échelle hautement reconfigurable et contrôlable pour la recherche en calcul parallèle et en systèmes distribués. Il vise a agréger 5000 CPUs. La plate-forme Grid'5000 comprenait à l'époque où nous avons démarré cette étude, un total de 20 grappes reparties sur 9 sites en France : Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia Antipolis et Toulouse. Chaque site hébergeait de 1 à 3 grappes de calcul, les processeurs à l'intérieur de chaque grappe étant homogènes. Le nombre de processeurs par grappe est compris environ entre 100 et 1000. Les architectures des processeurs sont AMD Opteron, Intel Xeon, Intel Itanium ou PowerPC. Chaque grappe utilise des liens d'interconnexion Gigabit (Giga Ethernet ou Myrinet). La liaison entre les sites est effectuée par le réseau très haut débit RENATER (Réseau National de télécommunications pour la Technologie l'Enseignement et la Recherche) avec des liens de $10Gb/s$. Les caractéristiques des grappes de Grid'5000 sont représentées dans le tableaux 4.2. Le vitesse des processeurs (en Gfop/s) a été mesurée grâce à la suite de test HPLinpack. Notons que la vitesse des processeurs du site de Grenoble ont été omis car il était impossible d'exécuter HPLinpack sur ces processeurs au moment où les mesures ont été réalisées.

site	grappe	processeurs	nb. proc.	Gflop/s
Bordeaux	Bordemer (C1)	Opteron 248	48	3,542
	Bordeplage (C2)	Xeon 3GHz	51	3,464
Grenoble	IDPOT	Xeon 2,4GHz	32	N/A
	Icluster2	Itanium 2	103	N/A
Lille	Chuque (C3)	Opteron 248	53	3,647
	Chti (C4)	Opteron 252	20	4,311
	Chicon (C5)	Opteron 285	26	4,384
Lyon	Capricorne (C6)	Opteron 246	56	3,254
	Sagittaire (C7)	Opteron 250	70	3,865
Nancy	Grelon (C8)	Xeon 5110	120	3,185
	Grillon (C9)	Opteron 246	47	3,379
Orsay	GDX (C10)	Opteron 246	216	3,388
	GDX2 (C11)	Opteron 250	126	4,040
Rennes	Paravent (C12)	Opteron 246	99	3,364
	Parasol (C13)	Opteron 248	64	3,573
	Paraquad (14)	Xeon 5148LV	66	4,603
Sophia	Azur (C15)	Opteron 246	74	3,258
	Helios (C16)	Opteron 275	56	3,675
	Sol (C17)	Opteron 2218	50	4,389
Toulouse	Sun (C18)	Opteron 248	58	3,586
			1435	

Tab. 4.2 – Caractéristiques des grappes de Grid'5000.

Le nombre de grappes de chaque plate-forme sera noté C, C étant un nombre entier supérieur ou égal à 1. Chaque grappe C_k contiendra P_k processeurs identiques pour un total de P processeurs dans la plate-forme. Les vitesses des processeurs et les caractéristiques des réseaux

locaux ne sont pas nécessairement les mêmes entre les différentes grappes. Les processeurs d'une même grappe sont reliés entre eux à travers un commutateur (*switch*). Ce commutateur est à son tour connecté aux autres grappes via une passerelle. Aucune topologie spécifique n'est imposée pour l'interconnexion des grappes. Pour nos expérimentations, les graphes d'interconnexion des plates-formes seront tels que pour deux processeurs distincts de la plate-forme, il n'existe qu'une et une seule route pour relier ces processeurs.

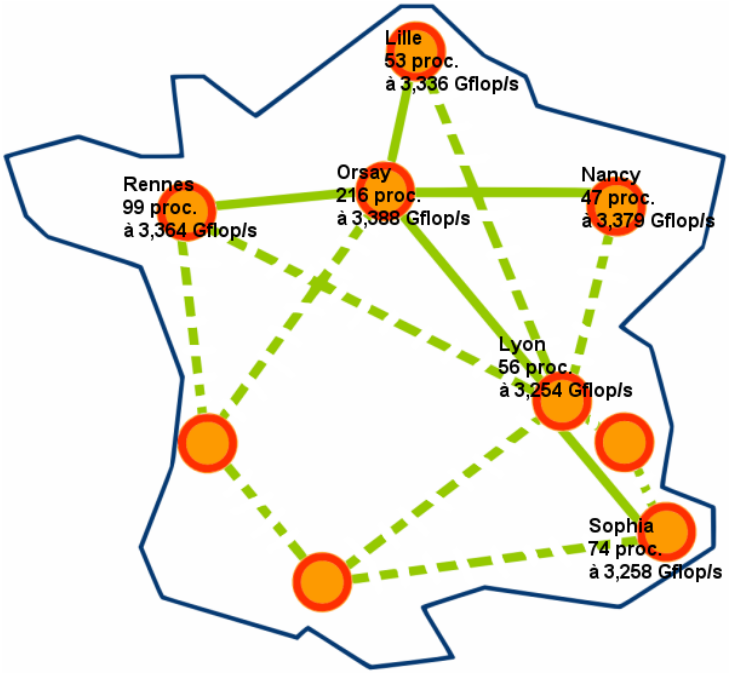

FIG. 4.1 – Exemple de plate-forme extraite de Grid'5000 avec des routes uniques entre les grappes retenues.

Il existe par ailleurs diverses approches pour générer des plates-formes synthétiques afin de réaliser des simulations pour l'évaluation d'algorithmes d'ordonnancement. Une des méthodes consiste à générer aléatoirement des plates-formes en utilisant des distributions uniformes pour certains paramètres (par exemple pour le nombre de grappes, pour le nombre de processeurs par grappes ou pour la vitesse des processeurs). Bien que cette approche soit simple et qu'elle permet de générer un grand nombre de plates-formes différentes, il n'est pas assuré que les plates-formes obtenues seront représentatives des plates-formes réelles. Une autre approche consiste à construire des modèles statistiques des grappes de calcul existantes et à utiliser ces modèles pour générer des plates-formes synthétiques en se fondant sur un ensemble de configurations réelles [65]. Bien que ces modèles statistiques conduisent à générer des plates-formes plus représentatives, beaucoup de chercheurs optent pour l'utilisation de quelques configurations construites à la main. L'inconvénient de cette dernière approche est que seulement quelques configurations sont construites. Dans ce chapitre, nous allons générer de nombreuses configurations qui correspondent à des sous-ensembles d'une plate-forme multi-grappes réelle : Grid'5000. Afin de respecter la contrainte de non multiplicité des routes entre deux processeurs, certaines dorsales de Grid'5000 ne seront pas utilisées. La figure 4.1 présente un exemple de plate-forme extraite de Grid'5000. Dans cet exemple, nous avons retenu six grappes respectivement sur les sites de Lille, Lyon, Nancy, Sophia Antipolis, Orsay et Rennes. Ce sous-ensemble comprend au total 545 processeurs et 5 dorsales à $10Gb/s$. Nous décrirons nos plates-formes synthétiques extraites de

Grid'5000 et leurs caractéristiques dans la section 4.6.1.

4.3 Ordonnancement sur plates-formes multi-grappes quasi homogènes

Dans cette section, nous proposons deux nouveaux algorithmes pour l'ordonnancement de DAGs de tâches parallèles sur des plates-formes multi-grappes quasi homogènes. Le premier de ces deux algorithmes est une heuristique pragmatique qui adapte aux plates-formes multi-grappes une de nos améliorations de l'heuristique CPA proposées dans le chapitre 3. Le second est un algorithme avec une garantie de performance que nous nommerons MCGAS (*Multi-Cluster Guaranted Allocation Scheduling*). Cette étude sera une des premières dans laquelle une heuristique garantie est expérimentalement comparée à heuristique pragmatique non garantie.

L'évaluation expérimentale de ces algorithmes se fera sur le sous-ensemble de Grid'5000 présenté dans la figure 4.1. Ce sous ensemble comprend 545 processeurs distribués sur 6 grappes localisées sur 6 sites différents. Le tableau 4.3 présente le nombre de processeurs de chaque grappe utilisée ainsi que la vitesse des processeurs de ces grappes. Il est possible pour les utilisateurs de réduire la vitesse de certains processeurs de grid'5000. La vitesse des processeurs de la grappe de Lille a alors été légèrement dégradée spécialement pour réduire l'hétérogénéité de ce sous-ensemble de grid'5000. Les autres grappes de ce sous-ensemble ont conservé leurs vitesses de processeurs originales. Bien que ce sous-ensemble représente seulement 10% de Grid'5000, il constitue avec ses 545 processeurs, une importante plate-forme quasi homogène.

Site	Lille	Lyon	Nancy	Orsay	Rennes	Sophia
nb. proc.	53	56	47	216	99	74
Gflop/s	3,336	3,254	3,379	3,388	3,364	3,258

TAB. 4.3 – Un sous-ensemble quasi homogène de la plate-forme Grid'5000.

Par la suite, nous admettrons sans perte de généralité que les grappes sont triées dans l'ordre décroissant de leur nombre de processeurs ($P_1 \geq P_2 \geq \ldots \geq P_C$). Ainsi, P_1 est le nombre de processeur de la plus grande grappe.

4.3.1 Adaptation d'une heuristique en deux étapes aux plates-formes multi-grappes quasi homogènes

Dans le chapitre précédent, nous avons montré que l'heuristique en deux étapes qui consiste à modifier la condition d'arrêt de la phase d'allocation de l'heuristique CPA [86] en utilisant T'_A au lieu de T_A permet d'améliorer les performances de CPA pour des DAGs quelconques pour lesquels il existe un surcoût lié à la parallélisation des tâches. En effet, non seulement elle permet d'utiliser plus efficacement les ressources, mais aussi elle réduit le makespan des applications. Nous adapterons donc cette heuristique pragmatique aux plates-formes multi-grappes quasi homogènes dans l'optique de la comparer à l'algorithme garanti que nous proposerons dans la section suivante.

Nous rappelons que la phase d'allocation de cette heuristique consiste à réduire le chemin critique de l'application ordonnancée tant sa longueur T_{CP} est supérieure à l'aire moyenne modifiée T'_A, en augmentant à chaque itération l'allocation de la tâche la plus critique (voir section 3.4.1).

Sur des plates-formes multi-grappes quasi homogènes, lors de la phase d'allocation, nous procédons comme si nous avions affaire à une grappe homogène de P processeurs, P étant le nombre total de processeurs de la plate-forme réelle. Pendant cette étape, la vitesse de chaque processeur est supposée égale à la vitesse minimale des processeurs de la plate-forme réelle. Puisque l'exécution d'une tâche parallèle doit être restreinte à l'intérieur d'une grappe, nous introduisons une contrainte supplémentaire dans la procédure d'allocation. Cette contrainte consiste à allouer au maximum à chaque tâche, le nombre de processeurs de la plus grande grappe (P_1). Par exemple, dans le cas de notre plate-forme expérimentale, le nombre maximum de processeurs qui peut être alloué à une tâche est $P_1 = 216$. Si la procédure d'allocation aboutit à un état où l'allocation de toute tâche du chemin critique est égale à P_1, alors nous arrêtons cette procédure car il n'est plus possible de réduire la longueur du chemin critique qui est une borne inférieure du *makespan* de l'application. Nous noterons $p(t)$, l'allocation d'une tâche modelable t obtenue à l'issue de cette nouvelle procédure d'allocation.

Pour la phase de placement, nous proposons deux heuristiques de liste. La première technique de placement, que nous appellerons FF (*First Fit*), est typiquement le genre d'heuristique utilisé dans les travaux théoriques sur l'ordonnancement. Elle place chaque tâche prête t sur la première grappe à avoir $p(t)$ processeurs libres. FF ne tient donc compte que de la date de disponibilité des processeurs et ignore ainsi la légère hétérogénéité de la plate-forme et les temps de redistribution des données entre les tâches. Cette procédure semble assez naïve mais ce genre d'heuristique de liste suffit pour préserver la garantie de performance des algorithmes issus de la théorie. En effet, la plupart des algorithmes garantis précédemment proposés [44, 45, 63, 71, 73, 112], ne tiennent pas compte des communications entre les tâches.

Les heuristiques pragmatiques utilisent des techniques de placement plus sophistiquées dans l'optique d'améliorer le *makespan*. Nous utiliserons EFT (*Earliest Finish Time*), une des techniques les plus connues dans les procédures de placement des heuristiques pragmatiques. Cette seconde technique est une adaptation de la procédure de placement de CPA aux plates-formes multi-grappes. Chaque fois que de nouvelles tâches deviennent prêtes, elles sont triées dans l'ordre décroissant de leur *bottom-level*. Ensuite toutes les tâches prêtes sont placées les unes après les autres en respectant cet ordre. Chaque tâche prête est placée sur la grappe qui lui confère la plus petite date de fin d'exécution. Cette technique tient compte de la vitesse des processeurs et des temps de redistribution de données entre les tâches pour déterminer l'ensemble de processeurs sur lequel chaque tâche doit être exécutée.

4.3.2 Résultats fondamentaux

Avant de présenter notre nouvel algorithme garanti, nous rappelons deux résultats fondamentaux qui fondent les bases de notre approche. Le premier résultat, obtenu par Skutella [98], donne un programme linéaire qui permet de trouver des allocations qui aboutissent au meilleur compromis possible entre la longueur du chemin critique T_{CP} et l'aire moyenne T_A. Le second résultat, obtenu par Lepère *et al.* [71], introduit le principe qui consiste à borner les allocations des tâches en vue d'améliorer la garantie de performance.

Motivés par le comportement usuel des programmes parallèles [36], nous ferons les hypothèses suivantes sur le temps d'exécution de nos tâches parallèles :

Hypothèse 1. *Le temps d'exécution $T(t, p(t))$ d'une tâche modelable t est décroissant en fonction du nombre de processeurs $p(t)$ qui lui est alloué.*

Hypothèse 2. *Le travail $w(t, p(t)) = p(t) \times T(t, p(t))$ d'une tâche modelable t est croissant en fonction du nombre de processeurs $p(t)$ qui lui est alloué.*

Blayo *et al.* ont montré que ces hypothèses sont réalistes [18]. L'hypothèse 2 reflète le fait que le surcoût lié à la parallélisation est de plus en plus perceptible lorsque le nombre de processeurs augmente.

Le problème discret de recherche de compromis durée / coût

Le problème qui consiste à déterminer les allocations des tâches d'une application parallèle mixte est analogue au problème discret de recherche de compromis entre la durée et le coût d'un projet (voir [114]). Tout comme dans le problème de recherche de compromis durée / coût, deux bornes inférieures doivent être optimisées. Dans la cas du problème d'allocation, il s'agit de la longueur du chemin critique et de l'aire moyenne. Notons que l'aire moyenne est égale au rapport du travail total effectué par les tâches sur le nombre total de processeurs de la plate-forme, c'est-à-dire le travail moyen effectué par les processeurs. Réduire le nombre de processeurs alloués à une tâche affecte ce compromis dans le sens où, d'après les hypothèses 1 et 2 l'aire moyenne diminue tandis que la longueur du chemin critique croît. À l'inverse, l'augmentation de l'allocation d'une tâche entraînera la réduction du chemin critique et l'accroissement de l'aire moyenne. L'objectif est donc de trouver le meilleur compromis possible entre les deux bornes inférieures, c'est-à-dire minimiser leur maximum. L'approche proposée dans [98] permet de résoudre ce problème dans le cadre d'une grappe homogène composée de m processeurs. Il y a m allocations possibles pour chaque tâche v, dont les temps d'exécution et les travaux correspondants sont respectivement $T(v, p(v))$ et $p(v) \times T(v, p(v))$, $p = 1, \ldots, m$. Il faut donc résoudre ce problème discret d'optimisation, c'est-à-dire trouver le meilleur compromis en choisissant pour chaque tâche, son allocation dans un ensemble fini de valeurs possibles.

Comme dans de nombreux problèmes, la résolution de ce problème discret est NP-difficile au sens fort tandis qu'il est plus facile de trouver des solutions pour la version continue. L'idée est donc de résoudre, dans un premier temps, un problème continu dans un espace de solutions plus large dans lequel chaque tâche v est remplacée par un ensemble de $m-1$ « activités ». Chaque activité a une fonction de coût linéaire fondée sur le temps d'exécution. À chaque activité v_i correspond une variable $x_{v,i}$ qui vérifie les inégalités $T(v,m) \leq x_{v,i} \leq T(v,i)$. Le coût de l'activité v_i est noté $\omega_{v,i}$ et est fixé à :

$$\omega_{v,i} = \frac{T(v,i) - x_{v,i}}{T(v,i) - T(v,m)}((i+1)T(v,i+1) - iT(v,i)).$$

Pour prendre en compte les contraintes de précédence, nous introduisons la variable s_v qui assurera qu'aucune tâche ne peut démarrer son exécution si tous ces prédécesseurs n'ont pas fini de s'exécuter. On doit avoir, pour tout $(u,v) \in \mathcal{E}$, $s_v \geq \max_i(s_u + x_{u,i})$.

En résumé, pour toute longueur de chemin critique donnée $\bar{T}_{CP}$, le coût minimal permettant d'obtenir cette longueur est déterminé en résolvant le programme linéaire définit par les contraintes suivantes :

$$\forall v, i, \quad T(v,m) \leq x_{v,i} \tag{4.1}$$
$$\forall v, i, \quad x_{v,i} \leq T(v,i) \tag{4.2}$$
$$\forall (u,v) \in \mathcal{E}, \quad \max_i(s_u + x_{u,i}) \leq s_v \tag{4.3}$$
$$\forall v, i, \quad s_v + x_{v,i} \leq \bar{T}_{CP} \tag{4.4}$$

et en minimisant la fonction de coût :

$$\sum_{v \in \mathcal{V}} \sum_{i=1}^{m-1} \omega_{v,i} \ . \tag{4.5}$$

Il faut noter qu'un coût fixe correspondant à la plus petite valeur possible du travail total, doit être ajouté à la fonction de coût ci-dessus afin de calculer le travail total que nous noterons $\bar{W}$. Pour plus de détails sur la construction du programme linéaire, le lecteur peut se référer à [98].

À ce stade, nous pouvons fixer $\bar{T}_{CP}$ et calculer la valeur de $\bar{W}$ correspondante, dans l'optique de trouver le meilleur compromis. Les valeurs de $\bar{T}_{CP}$ et $\bar{W}$ sont continues et pas nécessairement des nombres entiers. $\bar{T}_A = \frac{\bar{W}}{m}$ et $\bar{T}_{CP}$ ayant des comportement opposés, deux scénarios sont possibles. Si l'une des deux grandeurs est toujours supérieure l'autre, on utilisera les allocations extrêmes qui les rapprochent (chaque tâche utilisera un processeur si $\frac{\bar{W}}{m}$ est toujours plus élevé, ou tous les processeurs si $\bar{T}_{CP}$ est toujours plus grand). Sinon, le compromis optimal peut être approché par une recherche binaire. Dans le dernier scénario, les valeurs obtenues sont des bornes inférieures des valeurs optimales du problème discret. Nous noterons T^*_{CP} et W^* les valeurs optimales du problème discret.

Dans [98], une technique a été proposée pour arrondir la solution continue $(\bar{T}_{CP}, \frac{\bar{W}}{m})$ en une solution du problème discret dont les valeurs du temps et du coût $(T_{CP}, \frac{W}{m})$, sont respectivement inférieures à $\frac{1}{1-\mu}T^*_{CP}$ et $\frac{1}{\mu}\frac{W^*}{m}$ ou μ est un paramètre qui peut être arbitrairement choisi entre 0 et 1. Choisir une allocation de a processeurs pour une tâche v dans le problème d'origine revient à fixer toutes les valeurs de $x_{v,i}$ à $T(v,m)$ lorsque i est inférieur à a et à $T(v,i)$ pour i supérieur ou égal à a. Le coût induit par ces valeurs de $x_{v,i}$ dans le programme linéaire est alors

$$\sum_{i<a} (i+1)T(v,i+1) - iT(v,i) = aT(v,a) - T(v,1) \ ,$$

ce qui est le coût attendu moins le coût fixe mentionné précédemment qui correspond à la plus petite valeur du travail total. Lorsqu'on arrondi la solution optimale du programme linéaire on doit choisir quelles valeurs de $x_{v,i}$ doivent être fixées à $T(v,m)$ et celles qui doivent être fixées à $T(v,i)$. Cela est réalisé en utilisant le compromis suivant. Si $\frac{T(v,i)-x_{v,i}}{T(v,i)-T(v,m)}$ est inférieur ou égal à μ, alors $x_{v,i}$ est fixé à $T(v,i)$, réduisant ainsi la contribution du travail $\omega_{v,i}$ de l'activité correspondante à 0 pendant que le temps dédié à cette activité augmente :

$$T(v,i) - x_{v,i} \leq \mu(T(v,i) - T(v,m)) \leq \mu T(v,i) \ .$$

Par conséquent, $x_{v,i} = T(v,i) \leq \frac{1}{1-\mu}x_{v,i}$ et le temps utilisé par n'importe quelle activité est augmenté au plus d'un facteur $\frac{1}{1-\mu}$, d'où $T_{CP} \leq \frac{1}{1-\mu}T^*_{CP}$.

À l'inverse, si $\frac{T(v,i)-x_{v,i}}{T(v,i)-T(v,m)}$ est plus grand que μ alors $x_{v,i}$ est fixé à $T(v,m)$, réduisant ainsi le temps nécessaire pour l'activité concernée tandis que $\omega_{v,i}$ augmente à $(i+1)T(v,i+1) - iT(v,i)$. Puisque $T(v,i) - x_{v,i} > \mu(T(v,i) - T(v,m))$, $\omega_{v,i} > \mu\omega_{v,i}$, cela signifie que le travail total est au plus augmenté d'un facteur $\frac{1}{\mu}$. On a donc $\frac{W}{m} \leq \frac{1}{\mu}\frac{W^*}{m}$.

Nous avons ainsi déterminé des valeurs de T_{CP} et W pour le problème discret qui sont respectivement au plus $\frac{1}{1-\mu}$ et $\frac{1}{\mu}$ plus grand que les valeurs optimales.

Ordonnancement garanti sur une grappe

En s'appuyant sur le résultat de [98], l'étude effectuée dans [71] s'est focalisée sur comment ordonnancer efficacement des tâches modelables en préservant la plupart des allocations de sortes que l'on peut obtenir une garantie de performance dérivée des bornes inférieures du chemin critique et de l'aire moyenne. La difficulté découle du fait que les allocations sont construites de sorte qu'un nombre infini de processeurs peuvent être utilisés au même instant, car dans le programme linéaire il n'existe pas de contrainte concernant l'exécution simultanée des tâches. Avec

des tâches qui ont des temps d'exécution différents, il n'existe pas de transformation géométrique simple pour convertir un ordonnancement destiné à un nombre illimité de processeurs en un ordonnancement dédié à une plate-forme comprenant un nombre fixe de processeurs. L'ordonnancement doit donc être totalement reconstruit en conservant uniquement les informations sur les allocations.

L'algorithme proposé dans [71] est fondé sur les algorithmes d'ordonnancement de liste classiques. Cependant, un algorithme de liste ne peut être utilisé directement car le résultat peut être arbitrairement éloigné de celui de l'ordonnancement optimal. Considérons par exemple m paires de tâches dont la première tâche peut être exécutée sur tous les processeurs pendant un temps très court tandis que la seconde s'exécutera après la première sur un processeur pendant un temps très long. Le pire cas pour un ordonnancement de liste est d'exécuter toutes les paires l'une après l'autre. Le meilleur ordonnancement exécutera toutes les premières tâches, puis toutes les secondes tâches en parallèle évitant ainsi les périodes d'inactivité.

Afin d'éviter d'avoir un grand nombre de tâches prêtes en attente de s'exécuter sur un grand nombre de processeurs, la solution proposée dans [71] consiste à limiter le nombre de processeurs utilisés par tâche. Cette limite qui est notée $\mu(m)$ dans l'article original, où m est le nombre total de processeurs de la plate-forme, sera notée b dans la suite de ce chapitre afin d'éviter toute confusion avec le compromis du programme linéaire.

L'algorithme consiste tout simplement à insérer une étape d'arrondissement des allocations entre la phase d'allocation (dérivée de la résolution du problème durée / coût où μ est pris égale à $1/2$) et la phase de placement qui utilise un algorithme d'ordonnancement de liste. Cette étape d'arrondissement des allocations assure que la garantie de performance sera de $3 + \sqrt{5}$. L'analyse de ce résultat peut être résumé comme suit. Il y a trois types d'intervalles de temps dans le résultat de l'ordonnancement :

- T_1 : intervalles de temps tels qu'au plus $b - 1$ processeurs sont utilisés ;
- T_2 : intervalles de temps tels qu'au moins b et au plus $m - b$ processeurs sont utilisés ; et
- T_3 : intervalles de temps tels qu'au moins $m - b + 1$ processeurs sont utilisés.

Comme dans la preuve classique de Graham [54] qui proposait une garantie de performance égale à $2 - \frac{1}{m}$, le chemin critique et le travail total peuvent être bornés en utilisant ces trois intervalles. En effet :

- Pendant les intervalles de temps du premier type, aucune tâche n'a vu son allocation réduite à exactement b processeurs.
- Pendant les intervalles de temps du second type, aucune tâche prête n'est en attente de s'exécuter car il existe au moins b processeurs libres et la phase de placement est un algorithme d'ordonnancement de liste.
- Pour chaque type d'intervalle, on peut donner une borne inférieure du nombre de processeurs utilisés : 1 pour T_1, b pour T_2 et $m - b + 1$ pour T_3.

Un calcul relativement simple aboutit à la valeur optimale de b qui dépend du nombre total de processeurs et, à la garantie de performance (voir [71] pour plus de détails).

4.3.3 L'algorithme MCGAS

Nous nous inspirons du travail effectué dans [44] pour concevoir un nouvel algorithme garanti destiné aux plates-formes multi-grappes quasi homogènes. Il existe deux principales différences entre le modèle de plates-formes étudié dans [44] et celui que nous considérons dans cette étude. La première différence est que dans les plates-formes hiérarchiques de machines SMPs utilisées dans [44], tous les nœuds SMPs (pouvant être considérés comme des grappes) ont le même nombre de processeurs. Or ici, les grappes que nous étudions peuvent avoir des nombres de

processeurs différents. La seconde différence est que dans [44] les tâches parallèles sont autorisées à s'exécuter sur plusieurs nœuds. À l'inverse, dans cette étude nous restreignons l'exécution d'une tâche parallèle à l'intérieur d'une grappe. Tandis que la première différence engendre une difficulté supplémentaire, la seconde simplifie le problème. Ainsi la règle de placement définie dans [44] ne s'avère plus nécessaire. Cette règle permettait de limiter le nombre de grappes utilisé par chaque tâche en imposant qu'au moins $q-1$ grappes doivent être entièrement utilisées, si la tâche considérée doit s'exécuter sur q grappes avec q supérieur ou égal à 2.

Notre nouvel algorithme MCGAS est conçu en adoptant la même méthode que celle présentée dans [71] et permet ainsi de garantir une performance au pire cas par rapport au meilleur ordonnancement possible. Nous détaillerons les caractéristiques de cet algorithme par la suite. Nous utilisons la même technique fondée sur la programmation linéaire pour déterminer les allocations dans MCGAS. Afin d'obtenir plus de flexibilité, nous laissons la possibilité de faire varier le paramètre μ pour guider la technique d'arrondissement des allocations. Il est alors possible de biaiser le compromis en faveur de l'un des minima du *makespan*, T_{CP} ou $\frac{W}{P} = T_A$. D'après [98], T_{CP} et T_A seront au plus $1/(1-\mu)$ et $1/\mu$ fois plus grands que T^*_{CP} et T^*_A respectivement. T^*_{CP} et T^*_A représentent la longueur du chemin critique et l'aire moyenne de l'ordonnancement optimal.

Après avoir arrondi les solutions du programme linéaire à des nombres entiers grâce au paramètre μ, le paramètre b permet de borner les allocations des tâches afin de favoriser l'exécution en parallèle de plusieurs tâches et d'améliorer la garantie de performance. b est un nombre entier plus grand ou égal à 1 tel que l'allocation de chaque tâche doit être au plus égale à b. Une valeur de b élevée favorisera le parallélisme de données tandis qu'une faible valeur de b favorisera le parallélisme de tâches. b doit donc être fixé à une valeur inférieure ou égale au nombre total de processeurs de la grappe la plus grande (P_1) car l'exécution d'une tâche est restreinte à l'intérieur d'une grappe. L'objectif de la détermination de b, est d'éviter que l'exécution des tâches du chemin critique soient retardées. En effet, bien que l'algorithme MCGAS conduise à une garantie de performance, il ne tente pas de trouver un bon compromis entre le parallélisme de données et le parallélisme de tâches lors de la phase d'allocation. L'efficacité de l'ordonnancement et la garantie de la performance sont donc liées au choix de la valeur de ce paramètre.

Nous verrons la relation entre les paramètres μ et b et la garantie de performance dans la section suivante. Nous montrerons également comment ces paramètres peuvent être choisis pour que la garantie de performance soit la meilleure possible. Cette garantie de performance est déterminée en admettant qu'on utilisera, lors de la phase de placement, l'une des deux heuristiques de liste (FF ou EFT) présentées dans la section précédente. Il est important de noter que tout comme pour la plupart des algorithmes garantis précédemment proposés [44, 45, 63, 71, 73, 112], la garantie de performance de MCGAS que nous détaillerons par la suite ne tiendra pas compte des communications entre les tâches.

4.3.4 Recherche des meilleures valeurs des paramètres μ et b

Soit S le nombre minimum de processeurs pouvant être utilisés à un instant tel qu'aucune grappe ne dispose pas de plus de b processeurs libres :

$$S = \sum_{i=1}^{C} \max(0, P_i - b + 1),$$

Durant l'exécution d'un DAG de tâches parallèles, ordonnancé à l'aide de MCGAS, il est possible de dégager trois catégories d'intervalles de temps :

- T_1 : les intervalles de temps tels qu'au moins 1 et au plus $b-1$ processeurs sont utilisés ;
- T_2 : les intervalles de temps tels qu'au moins b et au plus $S-1$ processeurs sont utilisés ; et
- T_3 : les intervalles de temps tels qu'au moins S processeurs sont utilisés.

Nous noterons L_i la somme des longueurs des intervalles de type T_i ($i = 1, 2, 3$).

L'objectif de cette classification est de borner la contribution de chaque type d'intervalle de temps dans la longueur du chemin critique T_{CP} et l'aire moyenne T_A. Nous rappelons qu'au terme de l'arrondi des allocations déterminées par le programme linéaire, T_{CP} est au plus $1/(1-\mu)$ fois supérieur à T^*_{CP}, et T_A est au plus $1/\mu$ fois plus grand que T^*_A. D'après l'hypothèse 2, lorsqu'on réduit à b le nombre de processeurs alloué aux tâches dont l'allocation calculée était supérieure à b, la valeur de T_A n'augmente pas. De même, grâce aux hypothèses 1 et 2, la réduction de certaines allocations à b, entraîne l'augmentation de T_{CP}, d'au plus un facteur P_1/b.

Durant les intervalles de type T_1, aucune des tâches en cours d'exécution n'a vu son allocation réduite à b processeurs. On peut déduire que pour chacun de ces intervalles, une tâche du chemin critique est en cours d'exécution et que l'allocation de cette tâche critique n'a pas été réduite à b. Pour les intervalles de type T_2, il existe au moins une grappe sur laquelle b processeurs sont libres. Cela signifie qu'il existe suffisamment de processeurs libres pendant ces intervalles de temps et qu'aucune tâche prête n'est en attente d'être exécutée. Cela implique également que, durant ces intervalles, il y a une tâche du chemin critique en cours d'exécution. Cependant, dans ce cas, la tâche critique pourrait avoir vu son allocation réduite à b processeurs. Avec cette réduction probable des allocations à b, la contribution de ce type d'intervalles à la longueur du chemin critique est au moins d'un facteur b/P_1. En ce qui concerne les intervalles de type T_3, il n'y a aucune grappe qui dispose d'au moins b processeurs libres. Il peut donc avoir une tâche prête appartenant au chemin qui ne dispose pas d'assez de processeurs pour s'exécuter.

Par conséquent, d'une part T_{CP} ne peut pas être strictement inférieur à $T_1 + b/P_1 \times T_2$, et d'autre part la valeur de T_A est nécessairement supérieure ou égale à $T_1 + b \times T_2 + S \times T_3$. Puisque le *makespan*, C_{max} de l'application s'obtient en faisant la somme $L_1 + L_2 + L_3$ (car on ignore les temps des communications entre les tâches), nous pouvons maintenant écrire les trois équations suivantes qui conduiront à la détermination de la garantie de performance :

$$\begin{aligned} C_{max} &= L_1 + L_2 + L_3, \\ \frac{C^*_{max}}{1-\mu} &\geq T_{CP} \geq L_1 + \frac{b}{P_1}\, t_2, \\ \frac{C^*_{max}}{\mu}\, P &\geq W = T_A\, P \geq L_1 + b\, L_2 + S\, L_3. \end{aligned}$$

C^*_{max} représente le *makespan* optimal. Nous rappelons que $P = \sum_{i=1}^{C} P_i$ est le nombre total de processeurs de la plate-forme. Soit $\delta \in [0, 1]$ un nouveau paramètre. Ce paramètre n'a aucune interprétation concrète. Il s'agit d'une valeur algébrique que nous utiliserons pour combiner les inégalités précédentes. En multipliant la première des deux inégalités précédentes par δ, et la seconde inégalité par $1-\delta$ puis en additionnant les résultats, on obtient

$$C^*_{max} \geq \left(\delta(1-\mu) + \frac{(1-\delta)\mu}{P}\right) L_1 + b\left(\frac{\delta(1-\mu)}{P_1} + \frac{(1-\delta)\mu}{P}\right) L_2 + \frac{(1-\delta)\mu S}{P} L_3.$$

Nous simplifions cette équation en utilisant les quantités β_1, β_2 et β_3 telles que

$$C^*_{max} \geq \beta_1 L_1 + \beta_2 L_2 + \beta_3 L_3.$$

avec

$$\begin{aligned}\beta_1(\delta,\mu,b) &= \delta(1-\mu)+\frac{(1-\delta)\mu}{P},\\ \beta_2(\delta,\mu,b) &= b\left(\frac{\delta(1-\mu)}{P_1}+\frac{(1-\delta)\mu}{P}\right),\\ \beta_3(\delta,\mu,b) &= \frac{(1-\delta)\mu S}{P}.\end{aligned}$$

Soit β, le minimum des trois quantités :

$$\beta = \min_{i=1,2,3}\beta_i.$$

En utilisant le fait que $C_{max} = L_1 + L_2 + L_3$, nous obtenons

$$C^*_{max} \geq \beta C_{max}.$$

Le taux de performance garanti par MCGAS est donc $1/\beta$. Cette garantie de performance est minimale lorsque β prend sa plus grande valeur possible. Étant donné $(P_1,\ldots,P_C)$, trouver les valeurs de δ, b et μ qui maximisent β semble à priori complexe. Mais il s'avère qu'il est possible de trouver une bonne valeur approchée de la solution.

On remarque que les trois valeurs β_1, β_2 et β_3 sont de la forme $AX + BY$ avec A égal à $\delta(1-\mu)$, B égale à $(1-\delta)\mu$, et X et Y des nombres positifs. D'après le lemme 2, le couple (δ,μ) qui maximise le minimum de ces trois valeurs est tel que $\delta = 1-\mu$. Nous pouvons donc éliminer le paramètre δ des équations :

$$\begin{aligned}\beta &= \min(\beta_1(\mu,b),\beta_2(\mu,b),\beta_3(\mu,b))\\ &= \min\left((1-\mu)^2+\frac{\mu^2}{P}, b\left(\frac{(1-\mu)^2}{P_1}+\frac{\mu^2}{P}\right), \frac{\mu^2 S}{P}\right).\end{aligned}$$

Lemme 2. *Soit la fonction f définie sur $[0;1]\times[0;1]$ par $f(\delta,\mu) = \delta(1-\mu)X + (1-\delta)\mu Y$, X et Y étant deux nombres positifs ou nuls. $\forall(\delta,\mu)\in[0;1]\times[0;1]$, $\exists\, y\in[0;1] \mid f(y,1-y)\geq f(\delta,\mu)$.*

Démonstration. Soit $y = \sqrt{\delta(1-\mu)}$. Nous démontrons ci-dessous que $(1-y)^2$ est supérieur ou égale à $(1-\delta)\mu$:

$$\begin{aligned}(1-y)^2 &= 1+y^2-2y = 1+\delta(1-\mu)-2\sqrt{\delta(1-\mu)}\\ &= 1+\delta-\delta\mu-2\sqrt{\delta(1-\mu)}\\ &= 1-\mu+\delta+(1-\delta)\mu-2\sqrt{\delta(1-\mu)}\\ &= (1-\delta)\mu+(\sqrt{\delta}-\sqrt{1-\mu})^2 \geq (1-\delta)\mu.\end{aligned}$$

Puisque $y^2 = \delta(1-\mu)$ et X et Y sont positifs ou nuls, nous avons $f(y,1-y)\geq f(\delta,\mu)$. □

Nous allons maintenant calculer les valeurs de b et μ qui maximisent β. Rappelons que P_1 et P sont des caractéristiques fixées de la plate-forme tandis que S est une fonction affine par morceaux de b. On remarque que $\beta_1(\mu,b)$ dépend uniquement de μ. De plus, lorsque μ croît de 0 à 1, β_1 décroît de 1 à $1/P$ et β_3 croît de 0 à S/P pour b fixé. Par conséquent, la valeur la plus élevée du minimum de β_1 et β_3 s'obtient quand $\beta_1 = \beta_3$. Notons $\beta_{1.3}$ cette valeur de β_1 pour laquelle $\beta_1 = \beta_3$. Lorsque $\beta_1 = \beta_3 = \beta_{1.3}$, on a $(1-\mu)^2 = \mu^2(S-1)/P$. β est ensuite maximum lorsque $\beta_2(\mu,b)$ est égal à $\beta_{1.3}$, soit lorsque $b(S-1+P_1) = SP_1$. Nous obtenons la meilleure valeur

de b en arrondissant à un nombre entier la solution de cette équation simple à résoudre. Nous pouvons ensuite calculer la meilleure valeur possible pour μ. Nous sous-entendons par meilleures valeurs, des valeurs qui conduisent à la plus petite garantie de performance possible.

Dans le cas particulier de la plate-forme que nous utiliserons pour évaluer MCGAS (voir tableau 4.3), la meilleure valeur de β est atteinte pour $b = 87$. Cette valeur est le plus petit entier tel que $b(S + P_1 - 1) > SP_1$. Dans ce cas, la meilleure valeur pour μ est $1/(1 + \sqrt{\frac{S-1}{P}})$ soit $0,662$. La valeur de β est donc $S/(\sqrt{S-1} + \sqrt{P})^2$ soit $0,115$, ce qui correspond à un taux de performance $1/\beta$ proche de $8,695$.

Pour des tailles de grappes données, on peut aisément obtenir les différentes garanties de performances en fonction de b et μ Notons que pour les valeurs non optimales, le meilleur δ peut être différent de $1 - \mu$.

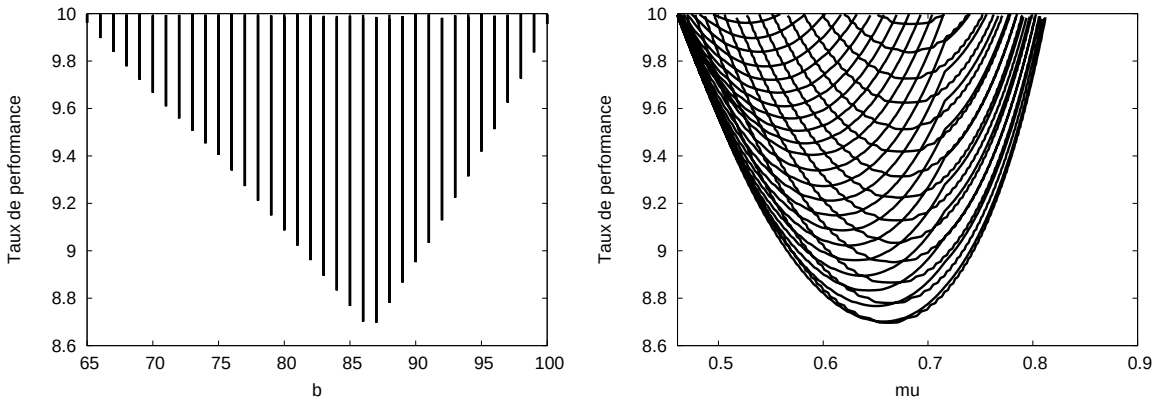

FIG. 4.2 – Projections 2D des valeurs du triplet $(\mu, b, 1/\beta)$ pour lequel le taux de performance $1/\beta$ est inférieur à 10, dans le cas particulier du sous-ensemble de Grid'5000 considéré.

La figure 4.2 montre deux projections de tous les triplets (b, μ, $1/\beta$) en fonction de b et μ pour la plate-forme étudiée. Nous observons que la garantie de performance croît plus rapidement quand on augmente b à partir de la valeur optimale calculée (87). Cette dégradation de la garantie de performance est plus modérée lorsque b devient plus petit que sa valeur optimale. Nous observons le même comportement quand μ varie autour de sa valeur optimale. La figure 4.3 présente le domaine de b et μ dans lequel la garantie de performance est inférieur à 10.

Ces graphes, ou des graphes similaires que l'on peut obtenir pour d'autres plates-formes donnent une bonne orientation pour paramétrer les valeurs de b et μ. En effet, les valeurs de b et μ qui conduisent à la meilleure garantie de performance pourraient ne pas conduire à la meilleure performance moyenne en pratique pour des application réelles. Par conséquent, on pourrait avoir besoin, dans des cas pratiques, de régler b et μ à des valeurs qui conduisent à une bonne performance moyenne pour un ensemble pertinent d'applications tout en conservant une garantie de performance raisonnable.

4.3.5 Évaluation expérimentale de MCGAS

Pour évaluer les deux algorithmes que nous venons de concevoir nous avons généré des DAGs quelconques selon la méthode présentée dans le chapitre précédent. La différence entre les DAGs utilisés dans cette section et les DAGs du chapitre précédent est qu'ils ne contiennent que 10, 20, ou 30 tâches au lieu de 10, 20, ou 50 tâches. Nous avons limité le nombre maximum de tâches à 30 car, pour déterminer les allocations, MCGAS résout un programme linéaire et cette

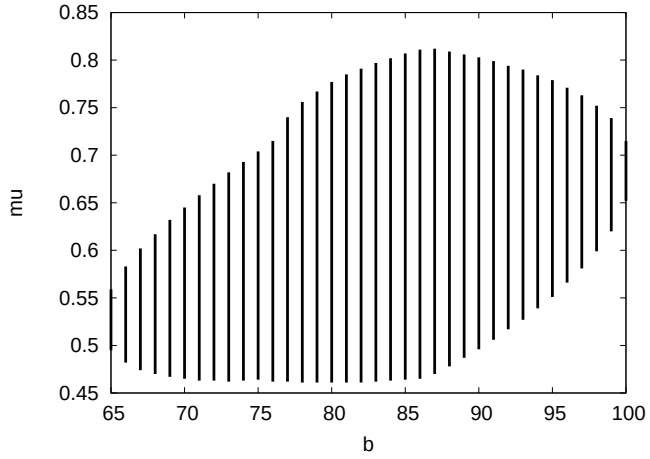

FIG. 4.3 – Domaine des valeurs de b et μ pour lesquelles le taux de performance garanti par MCGAS est inférieur à 10, dans le cas particulier du sous-ensemble de Grid'5000 considéré.

résolution met beaucoup de temps lorsque le nombre de tâches est élevé. En outre, pour des DAGs de grande taille l'ordinateur utilisé pour les simulations ne dispose pas d'une mémoire suffisante pour résoudre le programme linéaire.

Comparaison des allocations de processeurs

Dans un premier temps, nous allons comparer la qualité des allocations déterminées par les deux algorithmes. Pour cela, nous avons mesuré, pour chaque DAG, la longueur du chemin critique et le travail total à l'issue de la phase d'allocation. La longueur du chemin critique étant une borne inférieure du *makespan*, une faible valeur signifie que potentiellement, l'algorithme peut conduire à un bon *makespan*. En ce qui concerne le travail, il est lié à l'efficacité et une plus faible valeur correspond à une meilleure utilisation des ressources.

La figure 4.4 présente la longueur relative du chemin critique (à gauche) et le travail total relatif de MCGAS par rapport à l'heuristique pragmatique à l'issue de la phase d'allocation. Nous avons utilisé les valeurs optimales théoriques $\mu = 0,66$ et $b = 87$ pour MCGAS. Pour obtenir chaque courbe, nous avons trié les valeurs obtenues croissant. On observe dans le graphique de gauche que la longueur du chemin critique obtenu par MCGAS correspond dans le pire des cas à $95,28\%$ de la valeur donnée par les allocations de l'heuristique pragmatique. De plus cette longueur relative croît lorsque le nombre de tâches augmente. En moyenne, MCGAS conduit à des longueurs de chemin critique 12%, 9% et 8% plus petites que l'heuristique pragmatique respectivement pour 10, 20 et 30 tâches.

Dans le graphique de droite, nous observons que MCGAS consomme jusqu'à $3,5$ fois plus de ressources que l'heuristique pragmatique. Cela découle du fait l'heuristique pragmatique a été conçue pour limiter l'utilisation des ressources en utilisant T'_A au lieu de T_A dans la condition d'arrêt de sa procédure d'allocation. Par conséquent, MCGAS consomme en moyenne 110%, 57% et 35% plus de ressources que l'heuristique pragmatique respectivement pour 10, 20 et 30 tâches.

Enfin, nous avons voulu savoir si les allocations produites par MCGAS sont intrinsèquement meilleures que celles de l'heuristique pragmatique en mesurant le maximum des deux bornes inférieures du *makespan* que sont la longueur du chemin critique T_{CP} et l'aire moyenne T_A (Il s'agit bien de T_A même si pour sa condition d'arrêt des allocations, l'heuristique pragmatique

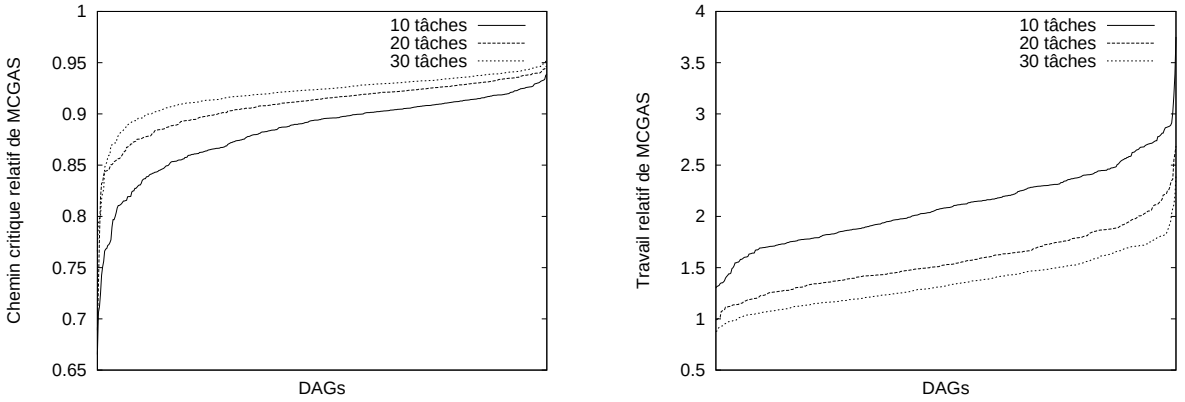

FIG. 4.4 – Longueur du chemin critique et travail total relatifs de MCGAS ($\mu = 0,66$ et $b = 87$) par rapport à l'heuristique pragmatique, pour des DAGs comprenant 10, 20, et 30 tâches.

utilise T'_A). Une plus petite valeur pour ce maximum indique une plus grande chance d'obtenir le meilleur *makespan*. Nous avons obtenu que la valeur donnée par MCGAS est en moyenne $9,63\%$ et au maximum $33,47\%$ plus petite que celle de l'heuristique pragmatique. MCGAS a donc plus de chance de fournir de meilleurs *makespans* que l'heuristique non garantie au prix d'une plus grande consommation de ressources.

En plus des DAGs irréguliers aléatoires, nous avons comparé les allocations de MCGAS et l'heuristique pragmatique pour des DAGs de type *Strassen* et FFT inspirés d'applications réelles. Ces deux types de DAGs représentent des cas de test classiques pour les algorithmes d'ordonnancement [83, 106, 120]. Ces deux types de DAGs sont beaucoup plus réguliers que nos DAGs synthétiques irréguliers de forme aléatoire, ces derniers étant représentatifs des *workflows* actuels qui sont une composition arbitraire de différents opérateurs. Pour chaque type d'application, nous avons utilisé 10 DAGs différents. Nous avons obtenu que les valeurs du maximum de T_{CP} et de T_A produites par MCGAS sont en moyenne 39% et 21% plus petites que celles de l'heuristique pragmatique respectivement pour les applications de type *Strassen* et FFT. L'avantage de MCGAS s'étend donc à un ensemble d'applications plus large que les DAGs irréguliers que nous avons générés.

Comparaison des deux techniques de placement

Nous avons mesuré les performances de MCGAS pour les deux techniques de placement FF et EFT. La phase d'allocation est réalisée en fixant μ à $0,66$ et b à 87. Comme attendu, nous avons obtenu que EFT donne en moyenne de meilleurs *makespans* que FF puisque cette technique ne tente pas d'optimiser le placement des tâches. Lorsque nous avons observé les performances des deux heuristiques en fonction de la largeurs des DAGs, nous avons obtenu que les *makespans* de EFT sont $3,7\%$, $20,5\%$ et $34,2\%$ meilleurs que ceux de FF respectivement pour des largeurs égales à $0,2$, $0,5$ et $0,8$. Il est donc nécessaire, pour la phase de placement, d'utiliser des heuristiques sophistiquées si l'on souhaite mieux gérer le parallélisme de tâches et améliorer en moyenne le *makespan* des applications, même si cela ne conduit pas améliorer la garantie de performance au pire cas. Par la suite, nous utiliserons donc EFT pour placer les tâches dans nos deux algorithmes.

Comme dans la section précédente, nous avons évalué MCGAS et l'heuristique pragmatique pour des DAGs de type *Strassen* et FFT. Pour les DAGs de type *Strassen*, MCGAS produit des

makespans en moyenne 22% plus petits que ceux de l'heuristique non garantie. Mais pour les DAGs de type FFT, l'heuristique non garantie produit de meilleurs résultats et ses *makespans* sont en moyenne 13% plus petits que ceux de MCGAS, bien que MCGAS produise des allocations intrinsèquement meilleures comme nous l'avons vu dans la section précédente. Ce phénomène révèle une faiblesse de la technique de placement EFT dans le cas particulier des DAGs de type FFT. Ces DAGs sont bien équilibrés en termes de volumes de communication et de calcul. Une caractéristique importante de ces DAGs est que certaines tâches produisent un volume important de données à redistribuer. Cependant, EFT s'intéresse seulement à la minimisation du temps de complétion de chaque tâche prête sans se « projeter en avant » afin de prendre en compte les volumes des données qui seront redistribuées après l'exécution de cette tâche. Par conséquent, EFT peut placer des tâches sur différentes grappes et ces décisions entraîneront de nombreuses communications inter-grappes pouvant être très coûteuses. Quand on utilise l'heuristique pragmatique, ce phénomène est moins visible car les tâches utilisent moins de processeurs. Plusieurs tâches peuvent donc être placées sur un plus petit nombre de grappes. Les communications inter-grappes sont ainsi réduites. Cette observation montre que des heuristiques de placement plus sophistiquées doivent être développées pour améliorer le *makespan* moyen des applications, même si on a de bonnes décisions d'allocation.

Compromis entre la garantie de performance et la performance moyenne

Nous étudions ici comment il est possible de trouver un compromis entre la garantie de performance de MCGAS et les valeurs moyennes du *makespan* si un utilisateur a pour cible un ensemble particulier d'applications. En effet celui-ci peut souhaiter obtenir de bonnes performances moyennes tout en s'assurant que la garantie de performance au pire cas n'est pas trop mauvaise.

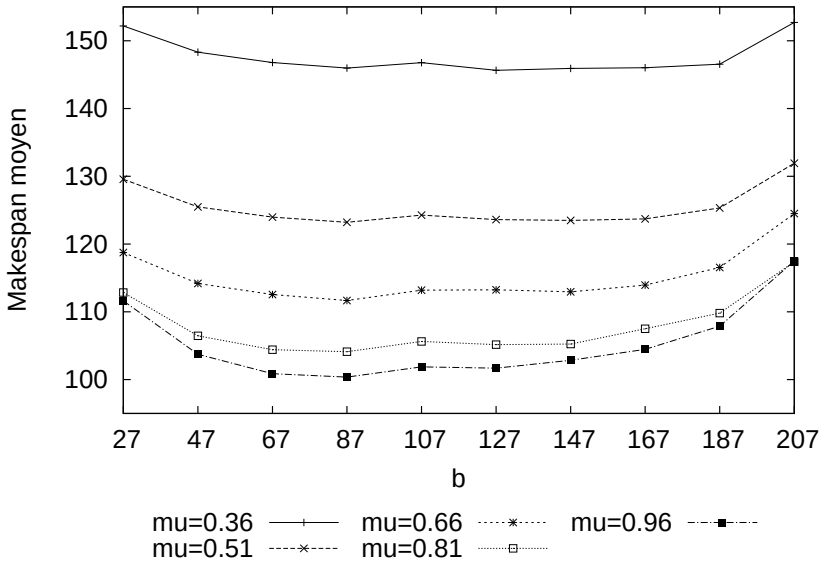

FIG. 4.5 – Évolution du *makespan* moyen obtenu par MCGAS lorsque b varie. Chaque courbe correspond à une valeur différente de μ.

La figure 4.5 présente les *makespans* moyens pour différentes valeurs de μ lorsque b varie. Nous sommes frappés par le fait que les *makespans* s'améliorent pour des valeurs de μ supérieures à

$0,66$. Une des raisons qui justifieraient ces valeurs inattendues est que l'ensemble d'applications généré pourrait biaiser les résultats. Pour la forme des DAGs, nous avons balayé une large gamme d'applications. Cependant, en ce qui concerne l'efficacité de la parallélisation des tâches, nous avons choisi une valeur maximale raisonnable pour la portion non parallélisable des tâches ($\alpha < 0,25$). Ces valeurs de α sont typiques pour des applications réelles mais cela pourrait biaiser les résultats. Pour cette raison nous avons également effectué les simulations pour des applications peu efficaces ($\alpha > 0,5$). Mais nous avons obtenu des résultats similaires.

On peut conclure que si la valeur optimale de μ déterminée analytiquement mène à la meilleure garantie de performance, en pratique, des valeurs plus élevées conduisent à de bonnes performances moyennes pour l'ensemble d'applications que nous avons générées. Toutefois il faut être vigilant sur la consommation de ressources car nous avons observé que le travail total augmente lorsque μ augmente. En effet, chaque fois que nous augmentons μ de $0,15$, le travail total moyen augmente entre 10 et 20%. Des valeurs élevées de μ conduisent donc à une utilisation moins efficace de la plate-forme. Il peut être intéressant de choisir pour un ensemble particulier d'applications, une valeur de μ plus élevée que sa valeur théorique dans le but de trouver un compromis entre la garantie de performance et la performance moyenne des applications. Par la suite nous utiliserons la plus grande valeur de μ telle que la garantie de performance est inférieure à 10, c'est-à-dire $\mu = 0,81$

Contrairement à μ, la valeur optimale calculée pour b fournit les meilleurs *makespans* moyens. En effet, une faible valeur rallonge le chemin critique et une valeur élevée force de nombreuses tâches à être exécutées sur la plus grande grappe, réduisant ainsi l'exploitation du parallélisme de tâches.

Comparaison des deux algorithmes

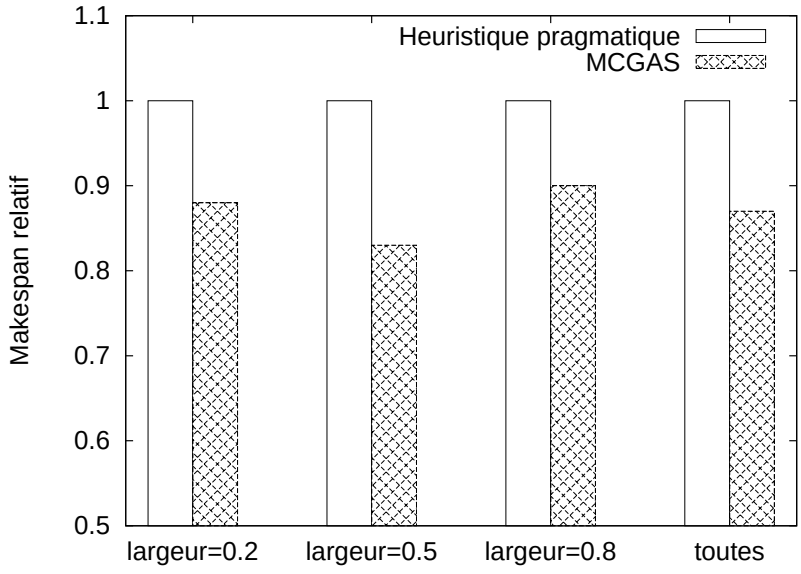

FIG. 4.6 – *Makespans* relatifs moyens obtenus par MCGAS ($\mu = 0,81$ et $b = 87$) par rapport à l'heuristique pragmatique.

La figure 4.6 montre le *makespan* relatif moyen de MCGAS par rapport à l'heuristique non garantie. Pour chaque instance ce *makespan* relatif de MCGAS est calculé en faisant le

rapport du *makespan* obtenu par MCGAS sur celui de l'autre heuristique. Les moyennes des *makespans* relatifs ont été ensuite calculées lorsque la largeur des DAGs est égale à $0,2$, $0,5$ ou $0,8$ et pour la totalité des simulations. Nous observons que MCGAS est plus performant en termes de *makespan*. Nous en déduisons que bien qu'une garantie de performance n'implique pas nécessairement une bonne performance moyenne, dans le cas de MCGAS, le choix de la valeur optimale pour b et d'une valeur supérieure à l'optimale pour μ de sorte que la garantie de performance soit inférieure à une borne que nous avons fixée, conduit à de bons *makespans* comparativement à une heuristique pragmatique.

Comparaison des temps de calcul des ordonnancements

Si les résultats précédents sont en faveur de MCGAS, il faut tenir compte des temps de calcul des ordonnancements dans la décision de choix d'un algorithme. MCGAS nécessite la résolution d'un programme linéaire qui peut mettre beaucoup de temps. Les deux algorithmes ont été exécutés sur un processeur Intel Xeon de fréquence $1,86$ GHz et nous avons mesuré le temps d'exécution de chaque instance. Le tableau 4.4 montre le temps d'exécution moyen des algorithmes en fonction du nombre de tâches des applications.

On observe que le temps d'exécution de MCGAS est beaucoup plus élevé que celui de l'heuristique non garantie. Par exemple, pour les DAGs contenant 30 tâches, MCGAS met 1 600 fois plus de temps que l'euristique non garantie pour calculer les ordonnancements. L'essentiel du temps de calcul des ordonnancements par MCGAS est dû au programme linéaire utilisé lors de sa phase d'allocation. Nous avons implanté MCGAS en utilisant la bibliothèque de programmation linéaire GLPK (*GNU Linear Programming Kit*) [53], fondée sur l'algorithme du simplexe. Cet algorithme a une complexité exponentielle au pire cas mais il est connu que dans certains cas pratiques, sa complexité est polynomiale. Il nous est difficile de déterminer la complexité du programme linéaire dans le cas de nos simulations mais celle-ci parait être quadratique au regard des temps moyens d'exécution de MCGAS en fonction du nombre de tâches. Le programme linéaire définit par les équations 4.1 à 4.5 donne une idée de la taille du problème. Le nombre de variables et de contraintes de ce programme linéaire sont respectivement $V \times (P+1)$ et $P \times (3V + E)$. Rappelons que P, V et E sont respectivement le nombre total de processeurs, le nombre de tâches du DAG et le nombre de relations de précédence du DAG.

nb. de tâches	*makespan* séquentiel	MCGAS		Heuristique non garantie	
		temps d'exéc.	*makespan*	temps d'exéc.	*makespan*
10	682,00	24,87	57,83	0,02	68,08
20	1378,00	99,91	103,62	0,07	119,36
30	2100,00	237,05	139,64	0,15	156,26

Tab. 4.4 – Comparaison des temps moyens d'exécution des algorithmes et *makespans* moyens obtenus (en secondes).

Bien que ces résultats semblent indiquer que MCGAS est difficilement applicable à des applications de grande taille, il existe trois raisons simples qui permettent de réduire ou de tolérer ces temps d'ordonnancement. Premièrement certaines bibliothèques commerciales de résolution de programmes linéaires, telles que CPLEX [37], sont présentées comme étant plus rapides que GLPK de plusieurs ordres de grandeurs. Leur utilisation peut donc considérablement réduire le temps d'exécution de MCGAS. La seconde raison est que l'utilisation de l'heuristique non garantie pourrait causer une perte de performance de l'ordre de 13% en moyenne. Or, il est

important de noter que le temps de calcul d'un ordonnancement ne dépend pas des temps d'exécution des tâches de l'application. Ce temps de calcul dépend uniquement de la structure du DAG et du nombre de tâches. Par conséquent, plus le *makespan* d'une application est élevé du fait que les temps d'exécution des tâches sont grands, plus il devient intéressant d'utiliser MCGAS. Pour certaines applications, le temps d'exécution de MCGAS peut être négligeable devant leur *makespan*. L'utilisation de MCGAS peut alors être bénéfique par rapport à l'heuristique pragmatique. Par exemple si on considère la dernière ligne du tableau 4.4 et on suppose que toutes les tâches mettent 100 fois plus de temps à s'exécuter, alors la somme du temps de calcul de l'ordonnancement et du *makespan* sera $237,05 + 139,64 \times 100 \simeq 14199$ pour MCGAS et $0,15 + 156,26 \times 100 \simeq 15626$ pour l'heuristique pragmatique. La dernière raison est qu'il est aussi important de noter que sur certaines plates-formes de production, un (bon) ordonnancement peut être réutilisé pour exécuter à plusieurs reprises une même application. Cela permet ainsi d'amortir le temps de calcul de l'ordonnancement.

Dans la section suivante, nous considérerons des plates-formes plus hétérogènes. nous adapterons nos heuristiques pragmatiques proposées dans le chapitre précédant aux agrégations hétérogènes de grappes homogènes.

4.4 Adaptation d'une heuristique en deux étapes aux plates-formes hétérogènes multi-grappes

Nous nous intéressons désormais au cas d'une plate-forme composée de C grappes véritablement hétérogènes. La plate-forme cible étant constituée de C grappes, notre idée consiste à attribuer, lors de la première phase, une *allocation de référence* à chaque tâche qui représentera ses C allocations potentielles sur les différentes grappes. Nous définirons dans la section suivante comment déterminer le nombre de processeurs à allouer à une tâche sur une grappe en fonction de son allocation de référence. Comme dans le chapitre précédent, cette première étape sera fondée sur la réduction du chemin critique. Les nouvelles heuristiques que nous allons présenter peuvent utiliser ou non le nouveau critère d'arrêt de la procédure d'allocation proposé dans la section 3.4.1 (utilisation de T'_A au lieu de T_A). En effet, pour les plates-formes homogènes, nous avons vu que la réduction des allocations avec T'_A permet de trouver un bon compromis entre l'utilisation des ressources et le *makespan* mais elle ne permet pas toujours de réduire le *makespan* par rapport à l'utilisation du tassage seul. Il serait donc intéressant de comparer les heuristiques avec et sans utilisation de T'_A. En revanche, le tassage étant toujours bénéfique pour le *makespan*, les heuristiques en deux étapes que nous proposerons dans cette section intégreront cette technique. Lors de la seconde phase, nous utiliserons des techniques de placement fondées sur le principe de EFT.

4.4.1 Allocation de processeurs

Étant donné qu'il existe désormais plusieurs allocations possibles pour une même tâche, il convient de redéfinir formellement les notions de chemin critique et d'aire moyenne qui déterminent la condition d'arrêt de la phase d'allocation. Pour cela nous introduisons la notion de *grappe de référence* sur laquelle nous ferons évoluer l'allocation des processeurs. Cette grappe de référence est une plate-forme homogène virtuelle ayant une puissance de calcul cumulée équivalente à celle de l'ensemble de la plate-forme réelle et dont les processeurs ont la plus petite vitesse de la plate-forme initiale. Le nombre total de processeurs contenus dans la grappe de

référence est donc :

$$P_{ref} = \left\lceil \sum_{k=1}^{C} \frac{P_k}{r_k} \right\rceil \tag{4.6}$$

où r_k est le rapport de la vitesse d'un processeur de la grappe de référence sur celle d'un processeur de la grappe C_k. L'utilisation de cette grappe homogène virtuelle permet de maintenir une complexité du même ordre que celle de CPA pour le nouvel algorithme. Notons $p^{ref}(t)$, l'allocation de référence pour la tâche t, c'est-à-dire le nombre de processeurs qui lui seraient attribués sur la grappe de référence. L'allocation de référence est définie de sorte que le temps d'exécution effectif de chaque tâche soit relativement proche du temps qu'elle mettrait sur la grappe de référence : $T^{ref}(t, p^{ref}(t))$. La longueur du chemin critique T_{CP} et l'aire moyenne modifiée T'_A se définissent maintenant par rapport aux allocations de référence :

$$T_{CP} = \max_{t \in \mathcal{V}} T_b^{ref}(t), \tag{4.7}$$

$$T'_A = \frac{1}{\min\left(P_{ref}, \sqrt{P_{ref} \times V}\right)} \sum_{i=1}^{V} \left(T^{ref}\left(t_i, p^{ref}(t_i)\right) \times p^{ref}(t_i) \right), \tag{4.8}$$

où $T_b^{ref}(t)$ est le *bottom-level* calculé à partir des allocations de référence des tâches. T_A est définie de manière analogue si l'heuristique utilise plutôt T_A au lieu de T'_A.

Comme dans CPA, à chaque itération de la procédure d'allocation, la tâche du chemin critique qui en bénéficie le plus se voit attribuer un processeur supplémentaire. Ce processeur est ajouté à son allocation de référence. Pour déterminer le nombre de processeurs réel à allouer à une tâche sur une grappe donnée, d'après son allocation de référence, nous utilisons la loi d'évolution de son temps d'exécution (son modèle d'accélération). Par exemple pour des tâches qui suivent la loi d'Amdahl [1] (voir l'équation 2.1 de la page 11), l'égalité :

$$T^k(t, p^k(t)) = T^{ref}(t, p^{ref}(t))$$

pour k fixé, conduit à :

$$f(p^{ref}(t), t, k) = \frac{(1-\alpha) \cdot T^k(t,1)}{T^{ref}(t, p^{ref}(t)) - \alpha \cdot T^k(t,1))}, \tag{4.9}$$

f étant la fonction qui permet de déduire $p^k(t)$, l'allocation de la tâche t sur la grappe C_k. Puisque sur une grappe donnée, on ne peut allouer à une tâche plus de processeurs que le nombre total de processeurs de cette grappe et que le nombre de processeurs alloués doit être un nombre entier, on en déduit :

$$p^k(t) = \min\{P_k, \lceil f(p^{ref}(t), t, k) \rceil\} \tag{4.10}$$

Ainsi, connaissant la fonction d'accélération d'une tâche t et son allocation de référence $p^{ref}(t)$, on peut aisément déduire sont allocation $p^k(t)$ sur une grappe C_k.

Lorsque la procédure d'allocation aboutit à un état où les allocations de référence des tâches sont telles que le nombre de processeurs à allouer à chacune des tâches du chemin critique sur toute grappe C_k est le nombre total de processeurs de cette grappe (P_k), alors les temps

[1] La loi d'Amdahl reste valide sur nos plates-formes multi-grappes car nous admettons que les processeurs sont uniformes.

d'exécution des tâches du chemin critique ne peuvent plus être réduits en augmentant leurs allocations de référence. La longueur du chemin critique T_{CP} a donc atteint sa valeur minimale. Nous dirons que *le chemin critique est saturé* quand cet état est atteint. Puisque la longueur du chemin critique ne peut plus être réduite, nous arrêtons la phase procédure d'allocation dès que le chemin critique est saturé.

L'algorithme 6 présente notre nouvelle phase d'allocation destinée aux plates-formes multi-grappes hétérogènes dans le cas où l'on utilise T'_A. Il suffit simplement de remplacer T'_A par T_A à la ligne 4 pour obtenir la version où l'on ne restreint pas les allocations.

Algorithme 6 Allocation de processeurs

1: **pour tout** $t \in \mathcal{V}$ **faire**
2: $p^{ref}(t) \leftarrow 1$
3: **fin pour**
4: **tant que** $T_{CP} > T'_A$ et chemin critique non saturé **faire**
5: $t \leftarrow$ tâche du chemin critique $\mid (\exists C_k \mid \lceil f(p^{ref}(t), t, k) \rceil < P_k)$ et $\left(\frac{T^{ref}(t,p^{ref}(t))}{p^{ref}(t)} - \frac{T^{ref}(t,p^{ref}(t)+1)}{p^{ref}(t)+1}\right)$ est maximum
6: $p^{ref}(t) \leftarrow p^{ref}(t) + 1$
7: Mettre à jour les T_t^{ref} et les T_b^{ref} des tâches
8: **fin tant que**

4.4.2 Placement des tâches

Comme dans le chapitre précédent, nous utiliserons des techniques de placement en ligne car celles-ci sont robustes du fait qu'elles ne décident des processeurs qu'une tâche doit utiliser seulement lorsque celle-ci est prête. Tout comme EFT, nos procédures de placement consisteront à attribuer à chaque tâche prête choisie t, les processeurs qui lui garantissent la plus petite échéance $T_f(t)$. Pour ce faire, ces procédures de placement tiendront compte des communications.

Soit $T_r^k(t', t)$ le temps nécessaire à la redistribution des données entre une tâche t' qui vient de s'exécuter (sur un ensemble de processeurs connu) et une tâche t qui lui succède, si on considère que t sera placée sur la grappe C_k. Ce temps dépend notamment des caractéristiques du réseau, de la quantité des données à transférer et des nombres de processeurs alloués respectivement aux tâches t' et t. Soit $T_m^k(t)$ la date d'arrivée de la dernière donnée d'une tâche prête t sur la grappe C_k. On a :

$$T_m^k(t) = \max_{t' \in Pred(t)} \left(T_f(t') + T_r^k(t', t) \right), \tag{4.11}$$

où $Pred(t)$ est l'ensemble des prédécesseurs de t. La date à laquelle la tâche t peut effectivement démarrer son exécution est donc :

$$T_s^k(t) = \max\{dispo(p^k(t)), T_m^k(t)\}, \tag{4.12}$$

où $dispo(p^k(t))$ est la date à laquelle la grappe C_k aura au moins $p^k(t)$ processeurs libres.

Une fois les allocations potentielles des tâches définies, nous avons étudié deux politiques différentes pour le placement des tâches.

Dans la première, nous adaptons l'heuristique d'ordonnancement de liste de CPA au cas où l'on dispose de plusieurs allocations possibles pour chaque tâche. Comme dans CPA, la tâche

prête la plus prioritaire est celle qui a le *bottom-level* le plus élevé. Ici nous utilisons le *bottom-level* calculé sur la grappe de référence T_b^{ref} à l'issue de la phase d'allocation. À chaque fois que la tâche t la plus prioritaire est choisie, nous déterminons l'allocation qui minimise sa date de fin d'exécution :

$$T_f(t) = \min_k \left(T_s^k(t) + T^k(t, p^k(t)) \right). \tag{4.13}$$

Nous en déduisons C_k, la grappe sur laquelle son exécution est prévue ainsi que sa date de début d'exécution : $T_s(t) = T_s^k(t)$. La combinaison de la procédure d'allocation et de cette heuristique de placement donne lieu à une nouvelle heuristique que nous nommerons HCPA (*Heterogeneous Critical Path and Area-based*).

Par la suite, nous utiliserons le mot clé OPT dans le nom de nos heuristiques en deux étapes qui utilisent T'_A au lieu de T_A. La combinaison des phases d'allocation et de cette technique de placement aboutit alors à deux heuristiques : HCPA et HCPA-OPT.

Dans notre seconde politique de placement, la tâche la plus prioritaire est déterminée selon le principe de l'heuristique *sufferage* [74]. Ce principe consiste à choisir parmi les tâches prêtes, celle qui serait la plus pénalisée s'il lui était attribué sa deuxième meilleure allocation au lieu de la première. Cette tâche est donc celle qui accuse la plus grande différence entre les deux dates de fin d'exécution prédites. Cette heuristique ne tient donc pas compte du chemin critique. Le but de sa mise en œuvre est de voir s'il peut être parfois intéressant d'utiliser des heuristiques de placement autres que celles fondées sur le chemin critique. Nous obtenons ainsi l'heuristique SHCPA (*Sufferage-based Heterogeneous Critical Path and Area-based*) ou plus précisément les heuristiques SHCPA et SHCPA-OPT. Ici également, le placement de la tâche prête la plus prioritaire vise à minimiser sa date de fin d'exécution en lui attribuant sa meilleure allocation.

4.4.3 Complexité des nouvelles heuristiques

Dans cette section, nous allons évaluer la complexité des nouvelles heuristiques en deux étapes et montrer que cette complexité du même ordre que celle de CPA.

Soit K, le nombre de processeurs maximum qu'on pourrait attribuer à une tâche sur la grappe de référence :

$$K = \max_{(k,t)} \left\lceil f^{-1}\left(P_k, t, k\right) \right\rceil \tag{4.14}$$

Dans le pire des cas, il faudrait K itérations pour déterminer l'allocation de chaque tâche dans la première phase. D'où un total de $K \times V$ itérations de la procédure d'allocation. À chaque itération de la boucle principale, la détermination du chemin critique est de l'ordre de $O(V+E)$. Le calcul T_{CP}, T_A ou T'_A, des T_t^{ref} et T_b^{ref} est également de l'ordre de $O(V+E)$. Lors de la détermination de la tâche la plus critique, il faudrait au pire cas déterminer les C allocations potentielles de chaque tâche du chemin critique pour savoir si un processeur supplémentaire peut lui être alloué. La complexité du corps de la boucle principale (choix de la tâche la plus critique, calcul de T_{CP}, T_A ou T'_A, T_t^{ref} et T_b^{ref}) est donc de l'ordre de $O((V+E)C)$. Nous en déduisons la complexité de la phase d'allocation : $O\left(V(V+E)C \times K\right)$.

En ce qui concerne la phase de placement, dans le cas des heuristiques fondées sur le chemin critique, les tâches prêtes sont triées par ordre de priorité à l'aide d'une procédure de l'ordre de $O(V^2)$ dans le pire des cas (cas où toutes les tâches du DAG sont prêtes en même temps du premier coup). Pour chaque tâche prête à placer, on teste les C allocations possibles. Au total, la procédure de placement des heuristiques de type HCPA est de l'ordre de $O(V(V+C))$.

Cette complexité est négligeable par rapport à celle de la phase d'allocation. Pour la phase de placement s'appuyant sur la technique de *Sufferage*, dans le pire des cas, les V tâches sont prêtes en même temps et la détermination de la i^{eme} tâche la plus prioritaire a un coût de l'ordre de $C \times (V - i)$. En effet, on examine les C allocations possibles pour chacune des $V - i$ tâches prêtes pas encore placées. La complexité de cette procédure de placement est donc de l'ordre de $C \times V^2$. Cette complexité peut également être négligée par rapport à celle de la phase d'allocation.

Il en découle que la complexité de nos nouvelles heuristiques est de l'ordre de :

$$O\left(V(V+E)C \times K\right). \tag{4.15}$$

Dans le cas d'une plate-forme composée d'une unique grappe homogène, la complexité de nos heuristiques est identique à celle de CPA puisque $C = 1$ et $K = P$. Dans le cas d'une plate-forme hétérogène comprenant plusieurs grappes, il convient d'exprimer le facteur $C \times K$ en fonction du nombre total de processeurs P. Soit H le degré hétérogénéité de la plate-forme qui correspond au rapport entre la vitesse du processeur le plus rapide et celle du processeur le plus lent. D'après l'équation 4.6 le nombre de processeurs de la grappe de référence est inférieur à $H \times P$, puisque $1/r_k \leq H$, $\forall k$ et que $P = \sum_{k=0}^{C-1} P_k$.

D'autre part, $K \leq \max_k \lceil \frac{P_k}{r_k} \rceil$ et est donc inférieur ou égal à P_{ref}. Par conséquent, $K \leq H \times P$. La complexité de nos heuristiques dans le cas d'une plate-forme hétérogène composée de plusieurs grappes est donc au plus supérieure à celle de CPA d'un facteur dépendant du nombre de grappes et du degré d'hétérogénéité de la plate-forme ($C \times H$).

Nous venons de mettre en œuvre, à partir d'heuristiques en deux étapes destinées aux plates-formes homogènes, quatre heuristiques (HCPA, HCPA-OPT, SHCPA et SHCPA-OPT) pour l'ordonnancement de DAGs de tâches parallèles sur des grappes hétérogènes de grappes homogènes. Dans la section suivante, nous proposons des améliorations à l'heuristique MHEFT, une approche orthogonale à la technique précédente dans la mise en œuvre d'algorithmes d'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes.

4.5 Améliorations de l'heuristique MHEFT

Les concepteurs de l'heuristique MHEFT [26] se sont inspirés de l'heuristique de liste HEFT [111] qui permet d'ordonnancer des DAGs de tâches séquentielles sur un ensemble hétérogène de processeurs. HEFT utilise les *bottom-levels* respectifs des tâches comme critère de priorité. Ces *bottom-levels* sont obtenus à partir du temps d'exécution moyen de chaque tâche et du temps de communication moyen relatif à chaque arc du DAG, ces moyennes étant calculées sur tous les processeurs et tous les liens réseaux de la plate-forme. Après ce calcul, les tâches sont triées dans l'ordre décroissant de leurs *bottom-levels*, puis chaque tâche est placée sur le processeur qui minimise sa date de fin d'exécution. Les décisions de placement de HEFT tiennent compte des temps de communications.

MHEFT étend HEFT au cas de l'ordonnancement de DAGs de tâches parallèles sur des agrégations hétérogènes de grappes homogènes de la manière suivante. Les temps moyens d'exécution et les temps moyens de communication sont calculés de la même façon que HEFT, en supposant que chaque tâche se verra allouer un seul processeur. Ensuite, après avoir déterminé les priorités des tâches, on alloue à chaque tâche l'ensemble de processeurs qui minimise sa date de fin d'exécution et la tâche est immédiatement ordonnancée sur cet ensemble. MHEFT tient également compte des temps de redistribution des données pour placer les tâches et restreint l'exécution d'une tâche à l'intérieure d'une grappe.

Le problème de l'heuristique MHEFT est que, contrairement à nos heuristiques en deux étapes qui tentent de trouver un compromis entre la longueur du chemin critique et l'aire moyenne, elle à tendance à allouer de nombreux processeurs à chaque tâche. En effet, cette heuristique ne cherche qu'à minimiser la date de fin d'exécution de la tâche courante sans se soucier de l'impact de son allocation sur les autres tâches. En plus d'utiliser de manière peu économe les ressources sur une plate-forme comprenant peu de grappes, cela peut entraîner une sous exploitation du parallélisme de tâches et conduire à de mauvais *makespans*. Nous proposons donc trois méthodes simples qui consistent à borner les allocations en vue de trouver un meilleur compromis entre l'utilisation des ressources et le *makespan* des applications.

La première amélioration que nous appellerons MHEFT-IMP permet d'augmenter l'allocation d'une tâche d'un processeur si et seulement si son temps d'exécution s'améliorera de plus d'un pourcentage fixé. Ce pourcentage représente un compromis qui peut être fixé par l'utilisateur qui ordonnance son application.

Avec la seconde proposition nommée MHEFT-EFF, on ne peut incrémenter l'allocation d'une tâche que si son efficacité vis-à-vis des ressources utilisées reste supérieure à un pourcentage donné. Nous rappelons que l'efficacité d'une tâche est le rapport entre son accélération et le nombre de processeurs utilisés et que cette grandeur diminue avec le nombre de processeurs s'il existe un surcoût lié à la parallélisation des tâches. Cette modification de MHEFT permettra alors de garantir que l'efficacité de l'heuristique à l'issue de l'ordonnancement sera au moins proche du pourcentage fixé.

La dernière amélioration, nommée MHEFT-MAX, consiste à allouer à chaque tâche au plus une fraction donnée des processeurs de la grappe qu'elle utilisera pour s'exécuter.

Notons que ces techniques, qui permettent de réduire les allocations des tâches par rapport à MHEFT, peuvent parfois mener à un *makespan* éloigné de celui de l'ordonnancement optimal. Par exemple, pour un DAG filiforme, le meilleur *makespan* peut être obtenu en utilisant le maximum de ressources possible pour chaque tâche. Le fait de réduire les allocations des tâches pour ce type de DAG conduira donc à une augmentation du *makespan*.

Dans la section suivante, nous comparons les deux techniques de conception d'algorithmes d'ordonnancement de DAGs de tâches parallèles sur des agrégations hétérogènes de grappes homogènes.

4.6 Évaluation des heuristiques

4.6.1 Méthodologie d'évaluation

Pour les simulations, nous utiliserons plates-formes synthétiques extraites de Grid'5000 dont les caractéristiques des grappes figurent dans le tableau 4.2. Un des paramètres qui caractérisera nos plates-formes extraites de Grid5000 est leur facteur d'hétérogénéité (en termes de vitesse des processeurs) défini par :

$$h = 100 \times (\frac{s_{max}}{s_{min}} - 1) \tag{4.16}$$

où s_{max} (respectivement s_{min}) est la vitesse (en Gflop/s) du processeur le plus rapide (respectivement le plus lent) de la plate-forme. Cela nous permet de distinguer les plates-formes quasi homogènes des plates-formes réellement hétérogènes. Nous considérerons qu'une plate-forme est quasi homogène lorsque $h \leq 10$.

Le tableau 4.5 résume les caractéristiques des plates-formes utilisées. $h = 0$ correspond à une plate-forme parfaitement homogène. Nous avons au total 355 plates-formes comprenant 1, 2, 4

	nb. grappes	nb. configurations	nb. total de processeurs	valeur de h
Hom.	1	18	20-216	0
	2	10	76-315	0,11-2,25
	4	10	221-445	2,85-4,82
	8	10	476-655	8,41-9,80
Hét,	2	82	67-342	10,07-44,52
	4	133	143-463	10,07-44,52
	8	92	438-729	20,1-44,52

Tab. 4.5 – Résumé des caractéristiques des plates-formes extraites de Grid'5000.

ou 8 grappes. Les configurations de ces plates-formes sont détaillées dans [107].

En ce qui concerne les applications, nous utiliserons tous les DAGs présentés dans le chapitre précédent pour lesquels le temps d'exécution des tâches est modélisé par la loi d'Amdahl. Nous aurons donc les DAGs réguliers quelconques, de type *Strassen*, de type FFT et les DAGs irréguliers.

Les heuristiques seront évaluées en utilisant les métriques normalisées définies dans la section 3.5.3.

Pour les heuristiques qui tentent d'améliorer MHEFT, nous avons décidé de fixer les paramètres de limitation des allocations des tâches à 5%, 50% et 50% respectivement pour MHEFT-IMP, MHEFT-EFF et MHEFT-MAX. En pratique, le choix de ces valeurs dépend de l'utilisateur et du compromis qu'il souhaite. Par exemple ce dernier peut avoir un budget plus ou moins limité et si l'utilisation ressources est payante, il peut décider de privilégier un ordonnancement efficace. Pour nos simulations, le choix de ces paramètres s'est fait de manière empirique en tenant compte du fait qu'il existe de plateau lié à la loi d'Amdhal. Nous avons par exemple remarqué que des valeurs supérieures à 5% pour MHEFT-IMP réduisent très fortement les allocations et elles doivent être évitées. Nous appellerons MHEFT-IMP5, MHEFT-EFF50 et MHEFT-MAX50 les trois heuristiques ainsi obtenues. MHEFT-IMP5 représentera donc la version modifiée de MHEFT qui autorise d'augmenter l'allocation d'une tâche seulement si cela permet d'améliorer son temps d'exécution de 5%. MHEFT-EFF50 est la modification de MHEFT telle que l'efficacité de chaque tâche doit être supérieur à 50%. Enfin avec MHEFT-MAX50, chaque tâche utilise au plus 50% des processeurs de la grappe sur laquelle elle est ordonnancée. La figure 4.7 présente un exemple de limitation des allocations avec MHEFT-IMP5, MHEFT-EFF50 et MHEFT-MAX-50. On y observe que les trois heuristiques limitent les allocations à des valeurs au delà desquelles le gain en temps d'exécution devient faible.

4.6.2 Résultats des simulations

Évaluations de l'adaptation de nos heuristiques en deux étapes aux plates-formes hétérogènes multi-grappes

Avant de comparer les deux approches de mise en œuvre d'algorithmes d'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes multi-grappes, nous évaluons l'adaptation aux plates-formes multi-grappes de nos heuristiques en deux étapes.

CPA n'étant pas adaptée aux plates-formes hétérogènes, chaque simulation de cette heuristique est effectuée sur une plate-forme équivalente à la plate-forme d'origine. La plate-forme

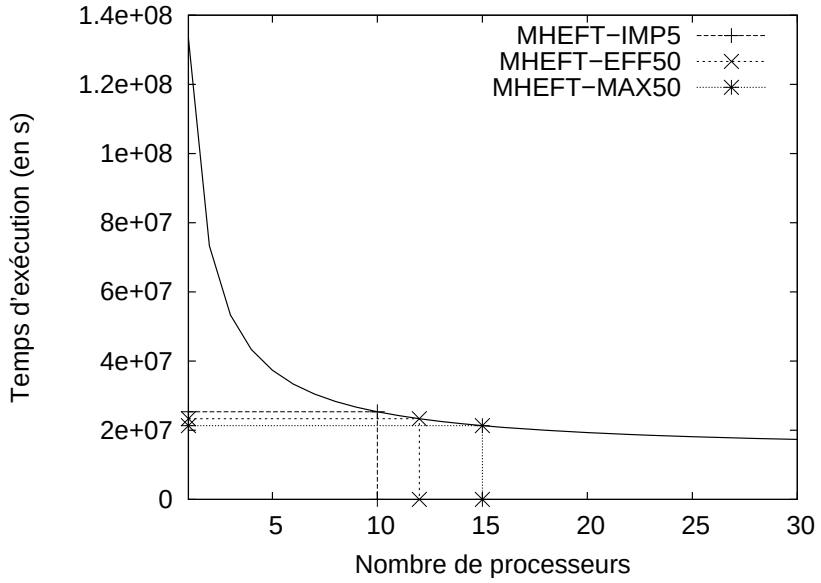

FIG. 4.7 – Exemple de limitation des allocations avec MHEFT-IMP5, MHEFT-EFF50 et MHEFT-MAX50 pour une tâche dont la portion non parallélisable est de 10% sur une plate-forme homogène comportant 30 processeurs.

utilisée par CPA a la même topologie que la plate-forme d'origine mais la vitesse des processeurs est la vitesse moyenne des processeurs de la plate-forme d'origine. Cette plate-forme a donc la même puissance totale que la plate-forme d'origine. CPA est ensuite lancée sans contraindre chaque tâche à s'exécuter à l'intérieur d'une grappe.

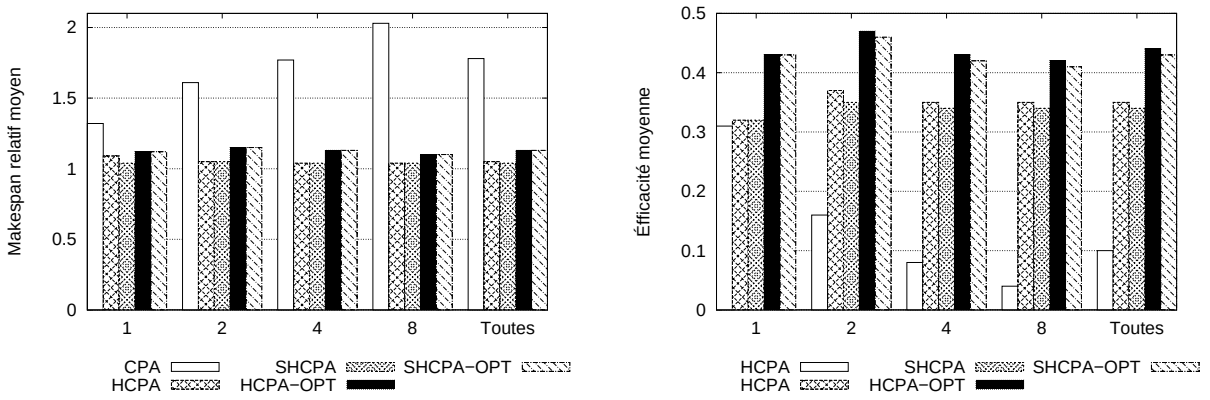

FIG. 4.8 – Performances moyennes des heuristiques en fonction du nombre de grappes pour les DAGs irréguliers.

La figure 4.8 présente les *makespans* relatifs moyens (gauche) et les efficacités relatives moyennes (droite) des heuristiques CPA, HCPA, SHCPA, HCPA-OPT, et SHCPA-OPT en fonction du nombre de grappes composant la plate-forme et sur l'ensemble des plates-formes (« Toutes »). Nous constatons tout d'abord que sur l'ensemble des simulations, CPA produit des *makespans* 59% plus élevés que la moins bonne de nos heuristiques. De plus, si l'on considère l'évolution du *makespan* relatif de CPA en fonction du nombre de grappes, on remarque que

celui-ci se dégrade jusqu'à être en moyenne à plus d'un facteur 2 de la meilleure solution obtenue par les différentes heuristiques. Cela s'explique, comme nous l'avons déjà souligné dans le chapitre précédent, par les allocations impliquant un grand nombre de processeurs déterminées par CPA lorsque la plate-forme comprend de nombreuses ressources. De telles allocations limitent l'exploitation du parallélisme de tâches tout en ne réduisant que très peu le temps d'exécution des tâches. Or, de façon assez logique, plus il y a de grappes dans la plate-forme, plus le nombre de processeurs est important. Cela est de plus confirmé par l'efficacité des ordonnancements produits par CPA qui décroît de façon significative lorsque le nombre de grappes augmente.

Nous pouvons également voir que nos quatre heuristiques ne sont pas impactées par l'évolution du nombre de grappes et obtiennent des *makespans* relatifs assez similaires. Comme dans le chapitre précédent, nous constatons que le tassage permet une plus grande minimisation du *makespan* tandis que l'utilisation combinée du tassage et de la nouvelle définition de l'aire moyenne conduit à des ordonnancements plus efficaces.

Enfin, si l'on compare les deux techniques de placement proposées (priorité relative au *bottom-level* et de type Sufferage), nous observons que les performances obtenues sont quasi identiques du point de vue du *makespan* et légèrement en faveur de HCPA du point de vue de l'efficacité. Une observation plus fine montre que le *makespan* obtenu par HCPA est en moyenne $0,3\%$ plus petit que celui de SHCPA pour les plates-formes hétérogènes. Pour ces plates-formes, HCPA donne un meilleur *makespan* que SHCPA dans $33,5\%$ des cas, HCPA donne un plus grand *makespan* dans 25% des cas et les *makespans* des deux heuristiques sont égaux dans $41,5\%$ des cas. À l'inverse, sur les plates-formes homogènes le *makespan* obtenu par SHCPA est en moyenne $0,2\%$ plus petit que celui de HCPA et le *makespan* de SHCPA est meilleur que celui de HCPA dans $31,9\%$ des cas tandis que celui de HCPA est meilleur dans $25,6\%$ des cas. Ces observations montrent que sur les plates-formes hétérogènes, le fait de donner la priorité aux tâches du chemin critique permet effectivement de réduire la longueur de ce chemin et par conséquent, de réduire le *makespan* de l'application ordonnancée. Nos heuristiques sont telles qu'une tâche prête est préférentiellement placée sur un petit nombre de processeurs rapides plutôt que sur un grand nombre de processeurs lents, à cause des erreurs de discrétisation dans la détermination d'une allocation réelle à partir de l'allocation de référence. En effet, la valeur obtenue par la fonction f de l'équation 4.9 est arrondie au plus petit nombre entier supérieur ou égal à cette valeur. Pour une même tâche parallèle, cela débouche sur des temps d'exécution plus petits sur les grappes les plus rapides. Puisque les tâches du chemin critique utilisent plus de ressources, le fait de les exécuter prioritairement sur les grappes les plus rapides avec HCPA réduit l'utilisation des ressources et augmente ainsi l'efficacité par rapport à SHCPA (voir graphique de droite de la figure 4.8). Il en est de même pour HCPA-OPT et SHCPA-OPT. Sur les plates-formes homogènes, les temps d'exécution des tâches ne varient pas entre les différentes grappes. Il est alors plus intéressant de favoriser les tâches qui souffriraient le plus. Pour ces plates-formes, l'utilisation du principe de *Sufferage* permet de réduire les périodes d'inactivité dans l'ordonnancement.

Il est intéressant de noter que CPA n'est pourtant pas défavorisé par la possibilité qui lui est offerte de déterminer des allocations utilisant des processeurs de plusieurs grappes. En effet le modèle de tâches parallèles utilisé (la loi d'Amdhal) ne répercute pas de surcoût lié à la présence de communications internes qui pourraient être impactées par les liens à grande latence connectant les grappes. Cependant la prise en compte de l'hétérogénéité de la plate-forme et la restriction de l'exécution d'une tâche à l'intérieur d'une grappe permet à nos heuristiques d'obtenir de meilleurs *makespans*. En effet la contrainte d'exécuter une tâche à l'intérieur d'une grappe peut réduire les temps de communications si certaines tâches interdépendantes s'exécutent à l'intérieur d'une même grappe. La figure 4.9 confirme cette hypothèse. On y observe

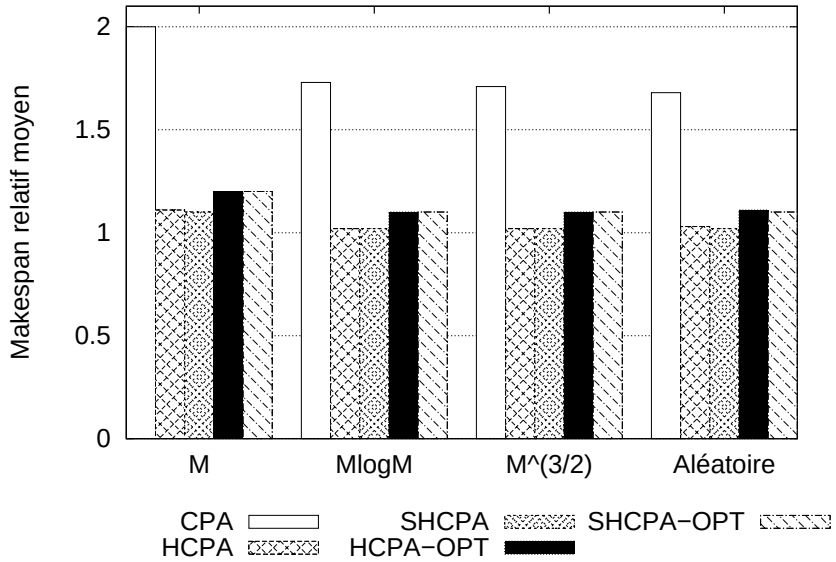

FIG. 4.9 – *Makespans* relatifs moyens des heuristiques en fonction de la complexité des tâches pour les DAGs irréguliers.

que pour les DAGs dont les coûts de redistribution des données sont du même ordre de grandeur que les coûts de calcul (complexité = M), CPA obtient des *makespans* relatifs en moyenne 2 fois plus grands que ceux de la meilleure solution contre environ $1,7$ pour les autres complexités.

Nous avons menés les mêmes expériences en considérant le cas de DAGs réguliers et avons obtenu des résultats similaires.

Impact des améliorations de MHEFT

Algorithme	*Makespan*	Efficacité
MHEFT	$121,88$	$0,126$
MHEFT-IMP5	$+72,5\%$	$+447,7\%$
MHEFT-EFF50	$+22,3\%$	$+324\%$
MHEFT-MAX50	$-3,9\%$	$+64,9\%$

TAB. 4.6 – Amélioration moyenne du *makespan* et de l'efficacité des heuristiques par rapport à MHEFT pour les DAGs irréguliers.

Le tableau 4.6 montre l'impact des différentes modifications de MHEFT. Nous observons plusieurs compromis entre le *makespan* et l'efficacité. Par exemple, MHEFT-IMP5 augmente en moyenne le *makespan* de 72% mais l'efficacité est en moyenne multipliée par 5, ce qui représente une importante économie de ressources. La modification la plus simple, c'est-à-dire MHEFT-MAX50 donne des performances strictement supérieures à MHEFT. En moyenne MHEFT-MAX50 réduit les *makespans* en allouant moins de processeurs aux tâches et en favorisant ainsi l'exécution en parallèle des tâches. La réduction des allocations par MHEFT-MAX améliore également l'efficacité.

La figure 4.10 présente les *makespans* relatifs (gauche) et les efficacités (droite) obtenus par les différentes variantes de l'heuristique MHEFT en fonction du nombre de grappes com-

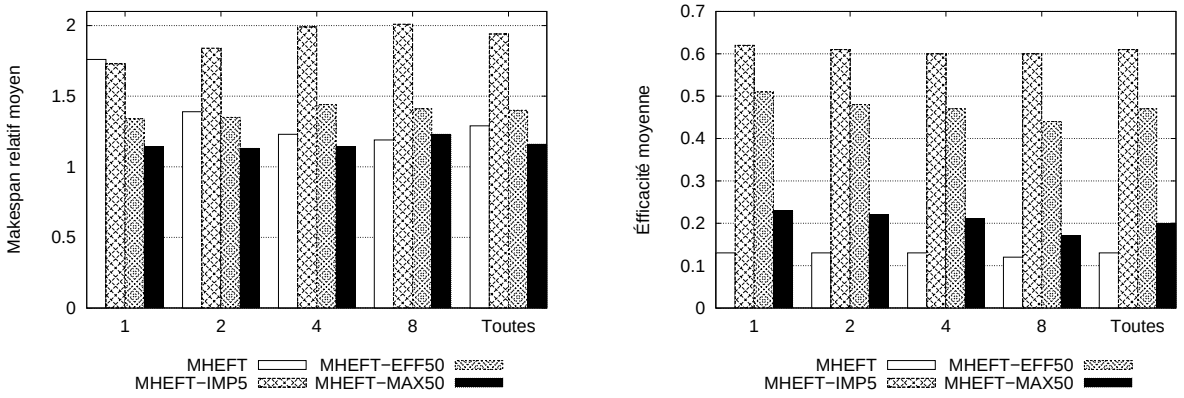

FIG. 4.10 – Performances moyennes des heuristiques en fonction du nombre de grappes pour les DAGs irréguliers.

posant la plate-forme. Le premier constat que nous pouvons effectuer est que, sur l'ensemble des simulations, le classement des différentes heuristiques est presque totalement inversé suivant que l'on s'intéresse au critère de la réduction du temps de complétion ou à celui de l'augmentation de l'efficacité. En effet, vis-à-vis du *makespan*, l'ordre est {MHEFT-MAX50, MHEFT, MHEFT-EFF50 et MHEFT-IMP5} alors que vis-à-vis de l'efficacité nous avons {MHEFT-IMP5, MHEFT-EFF50, MHEFT-MAX50, MHEFT}. Cela souligne la capacité de l'heuristique d'être paramétrable et de s'adapter aux exigences de l'utilisateur. Si l'utilisation des ressources est facturée à l'utilisateur, celui-ci préférera une variante plus économe mais néanmoins performantes telle que MHEFT-EFF50. Cette variante a en effet la deuxième meilleure efficacité $(0,47)$ et le troisième meilleur *makespan* relatif (à un facteur 1.4 en moyenne de la meilleure solution obtenue). En revanche, si l'utilisateur a un accès illimité et gratuit à la plate-forme, sa préférence pourra aller à MHEFT-MAX50 qui obtient les meilleurs *makespans* relatifs tout en étant légèrement plus efficace que l'heuristique originale. Cela confirme les informations du tableau 4.6.

Il faut cependant noter que les performances relatives des différentes heuristiques dépendent également de la configuration de la plate-forme. En effet, les performances relatives en termes de *makespan* de MHEFT s'améliorent lorsque l'on augmente le nombre de grappes (passage d'un facteur $1,8$ à un facteur $1,3$). À l'inverse, les performances de MHEFT-IMP5 se dégradent avec l'augmentation du nombre de grappes (de $1,7$ à 2). Dans le cas de MHEFT-IMP5 cela s'explique par le fait que cette variante n'utilise qu'assez peu de ressources (comme le montre par ailleurs son efficacité) et ce quelque soit la quantité disponible. Rappelons que l'augmentation de l'allocation d'une tâche s'arrête dès que le gain en termes de réduction du temps d'exécution devient trop faible (inférieur à 5%). En revanche, les concurrents vont quant à eux adapter leurs allocations en fonction des ressources disponibles et donc réduire leur *makespan*. Dans le cas de MHEFT, l'allocation d'une tâche est augmentée tant qu'il y a une amélioration du temps d'exécution, c'est-à-dire tout le temps du fait du modèle d'accélération choisi, comme on pouvait le voir sur la figure 4.7. Dans le cas d'une plate-forme composée d'une seule grappe, l'exploitation du parallélisme de tâches par MHEFT est ainsi nulle alors que lorsque plusieurs grappes sont disponibles, MHEFT peut placer autant de tâches en parallèle qu'il y a de grappes. Les performances de MHEFT seront par conséquent meilleures pour des applications exhibant peu de parallélisme de tâches, ce qui est confirmé par la figure 4.11. Sur cette figure il est également intéressant de noter le comportement inverse pour MHEFT-IMP5 qui produit de « petites » allocations. Cela implique que pour chaque tâche le temps d'exécution est assez long

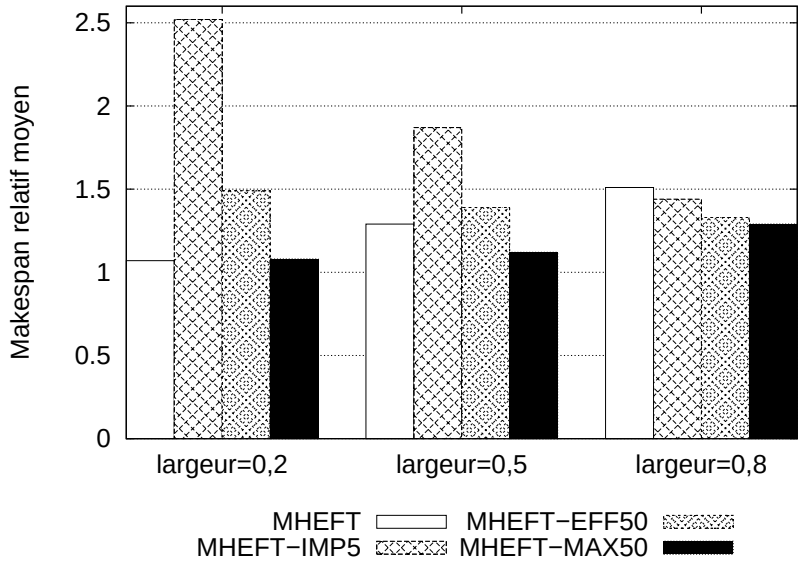

FIG. 4.11 – *Makespans* relatifs moyens des heuristiques en fonction de la largeur des DAGs pour les DAGs irréguliers.

(en comparaison de celui obtenu avec les « grandes » allocations de MHEFT par exemple) mais également que la possibilité d'exécuter des tâches en parallèle est maximale. MHEFT-IMP5 obtient donc assez logiquement de meilleurs résultats pour des applications présentant plus de parallélisme de tâches à exploiter.

Comparaison des deux approches

Nous allons maintenant comparer les performances des deux approches de conception d'algorithmes d'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes multi-grappes. Rappelons que l'une de ces approches, utilisée par MHEFT, introduit la gestion des tâches modelables dans des heuristiques conçues pour des tâches séquentielles, tandis que l'approche suivie par HCPA introduit la gestion de l'hétérogénéité dans des heuristiques d'ordonnancement de tâches modelables sur plates-formes homogènes.

La figure 4.12 présente les *makespans* relatifs moyens (gauche) et les efficacités moyennes (droite) obtenus par HCPA, HCPA-OPT et les différentes variantes de MHEFT pour les DAGs réguliers quelconques en fonction du nombre de grappes et sur la totalité des simulations. On y observe que sur la totalité des simulations, les *makespans* relatifs moyens des heuristiques sont supérieurs à $1,3$. Cela veut dire qu'aucune des heuristiques n'est toujours la plus performante sur le plan du *makespan*. Le tableau 4.7 confirme cette hypothèse. L'heuristique qui obtient le plus grand nombre de fois le meilleur *makespan* n'obtient ce résultat que dans $42,7\%$ des cas. Celle-ci est donc loin d'être tout le temps la meilleure heuristique en termes de *makespan*. Nous observons dans le graphique de gauche de la figure 4.12 que le *makespan* relatif des heuristiques dépend du nombre de grappes. Lorsque le nombre de grappes est inférieur ou égal à 4, HCPA obtient les meilleurs *makespans* relatifs suivie par MHEFT-MAX50. Mais lorsque le nombre de grappes est plus grand (8), MHEFT est en moyenne l'heuristique la plus performante en termes de *makespan* car tout en utilisant de nombreuses ressources, MHEFT peut désormais exécuter plus de tâches en parallèle par rapport aux plates-formes comprenant un plus petit nombre de

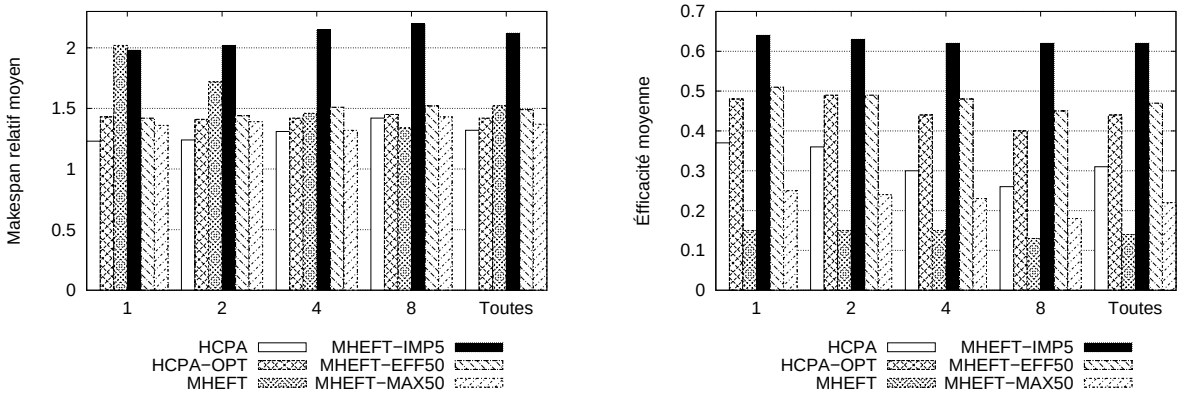

FIG. 4.12 – Performances moyennes des heuristiques en fonction du nombre de grappes pour les DAGs réguliers quelconques.

grappes.

Sur l'ensemble des simulations, HCPA-OPT et MHEFT-EFF50 obtiennent les meilleurs compromis entre le *makespan* et l'efficacité. Les *makespans* relatifs de ces deux heuristiques sont respectivement $7,6\%$ et $12,9\%$ moins bons que ceux de HCPA, l'heuristique la plus performante sur le plan du *makespan*, tandis que leurs efficacités sont respectivement $41,9\%$ et $51,6\%$ supérieurs à ceux de HCPA. En revanche, en plus d'obtenir en moyenne de moins bons *makespans* relatifs que HCPA, les efficacités moyennes données par MHEFT et MHEFT-MAX50 sont respectivement 45% et 30% plus faibles que celle de HCPA. MHEFT-IMP5 est l'heuristique la plus efficace mais son *makespan* relatif moyen est de loin le plus élevé.

Algorithme	Pourcentage
HCPA	$20,6\%$
HCPA-OPT	$7,8\%$
MHEFT	$42,7\%$
MHEFT-IMP5	$3,1\%$
MHEFT-EFF50	$7,3\%$
MHEFT-MAX50	$21,1\%$

TAB. 4.7 – Pourcentage des instances où chaque algorithme obtient le meilleur *makespan* pour les DAGs réguliers quelconques.

Les *makespans* relatifs des heuristiques dépendent également de la largeur des DAGs (voir figure 4.13). Pour des DAGs filiformes (largeur $= 0,2$), MHEFT et MHEFT-MAX50 qui allouent plus de processeurs aux tâches obtiennent les meilleurs *makespans* relatifs. En effet, il existe peu de tâches concurrentes durant l'exécution d'un tel DAG. La réduction du chemin critique a alors un fort impact sur le *makespan*. À l'inverse, lorsque les DAGs sont larges (largeur $= 0,8$), HCPA, HCPA-OPT, MHEFT-IMP5 et MHEFT-EFF50 qui allouent moins de processeurs aux tâches par rapport à MHEFT et MHEFT-MAX50 permettent d'exploiter plus de parallélisme de tâches et aboutissent ainsi à de meilleurs *makespans* relatifs moyens.

Les figures 4.14 et 4.15 présentent les performances moyennes des heuristiques en fonction du nombre de grappes et sur la totalité des simulations respectivement pour les DAGs de type FFT et *Strassen*. Nous observons également pour ces DAGs que le *makespan* relatif moyen d'aucune des heuristiques n'est proche de 1. Comme pour les DAGs réguliers quelconques, MHEFT et

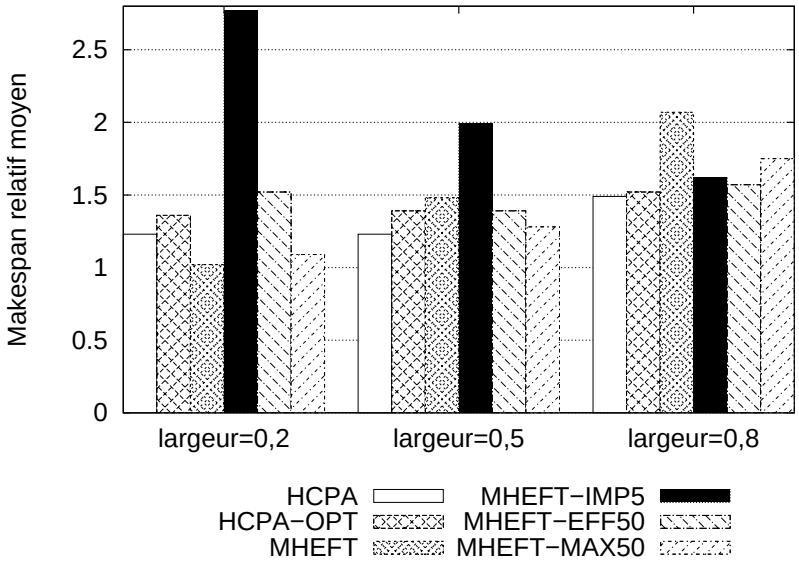

FIG. 4.13 – *Makespan* relatif moyen des heuristiques en fonction de la largeur des DAGs pour les DAGs réguliers quelconques.

MHEFT-MAX-50 deviennent performantes lorsque le nombre de grappes augmente. Ces DAGs étant larges, MHEFT n'obtient pas les meilleurs *makespans* relatifs lorsque le nombre de grappes est égal à 8. Sur l'ensemble des simulations, HCPA obtient le meilleur *makespan* relatif moyen pour les DAGs de type FFT alors que MHEFT-EFF50 est l'heuristique la plus performante en termes de *makespan* pour les DAGs de type *Strassen*. Pour ces deux types de DAGs, HCPA-OPT et MHEFT-EFF50 obtiennent également les meilleurs compromis entre le *makespan* et l'efficacité.

Lorsqu'on observe les résultats pour les DAGs irréguliers (voir figure 4.16) où il n'existe qu'un seul chemin critique, MHEFT-MAX50 fournit les meilleurs *makespans* relatifs moyens quand le nombre de grappes est inférieur ou égal à 4. En effet, la réduction de la longueur du chemin critique à un impact plus important sur le *makespan* pour ces DAGs et, au vue de son

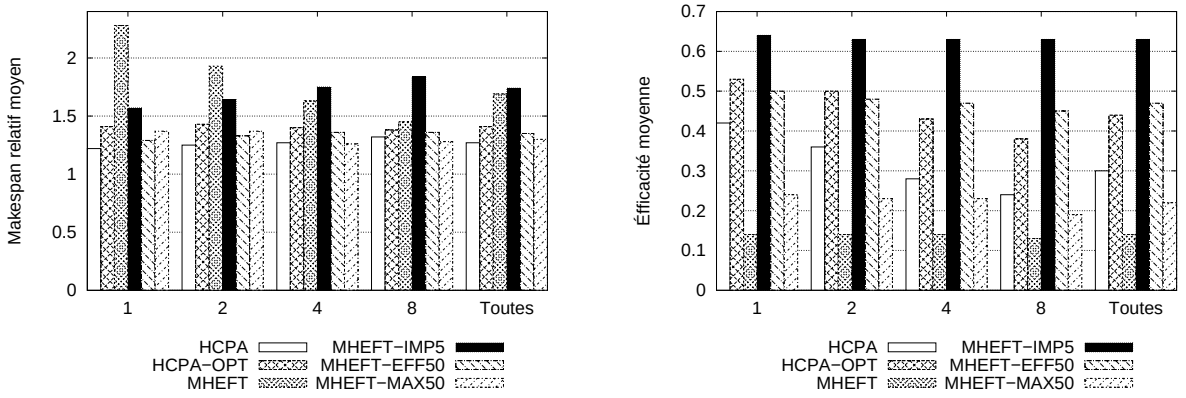

FIG. 4.14 – Performances moyennes des heuristiques en fonction du nombre de grappes pour les DAGs de type FFT.

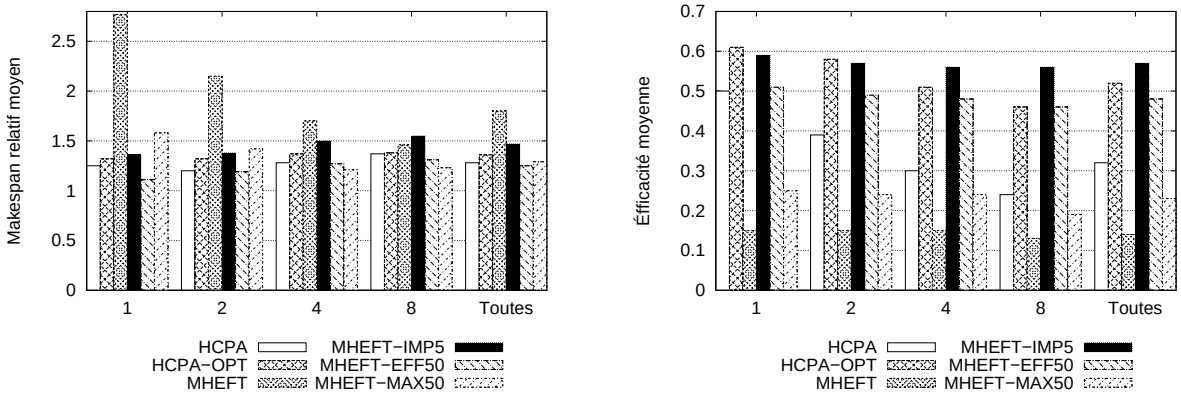

FIG. 4.15 – Performances moyennes des heuristiques en fonction du nombre de grappes pour les DAGs de type *Strassen*.

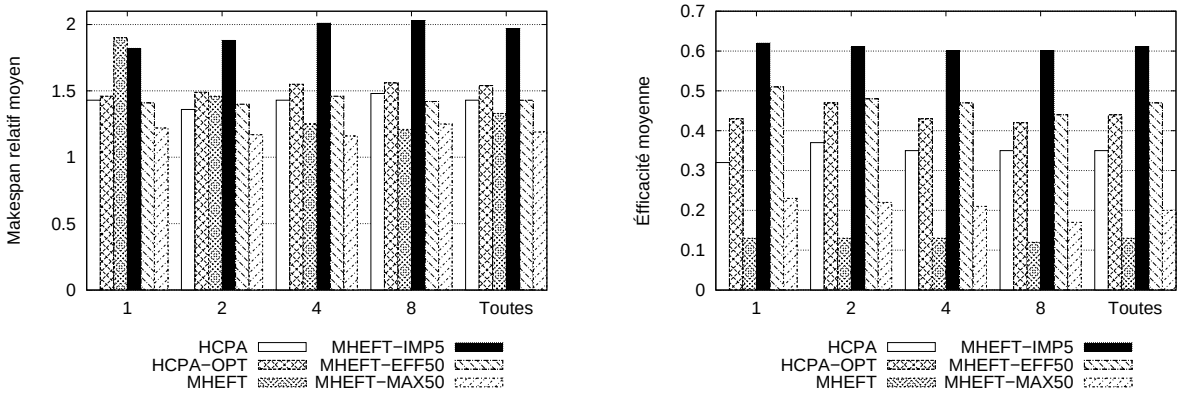

FIG. 4.16 – Performances moyennes des heuristiques en fonction du nombre de grappes pour les DAGs irréguliers.

efficacité, MHEFT-MAX50 est l'heuristique qui utilise le plus de ressources à part MHEFT. Elle permet ainsi de réduire considérablement le chemin critique tout en autorisant, comparativement à MHEFT, plus de tâches à s'exécuter en parallèle. Lorsque le nombre de grappes est égal à 8, l'heuristique MHEFT obtient le meilleur *makespan* relatif moyen car elle permet d'exécuter suffisamment de tâches en parallèle. Toutefois il faut noter que HCPA, HCPA-OPT et MHEFT-EFF50 donnent un meilleur compromis entre le *makespan* et l'efficacité vis-à-vis de MHEFT-MAX50 qui obtient le meilleur *makespan* relatif moyen. En effet, les *makespans* relatifs moyens de ces heuristiques sont respectivement $20,2\%$, $29,4\%$ et $20,2\%$ moins bons tandis que leurs efficacités sont respectivement 75%, 120% et 135% meilleures.

À partir de ces résultats, nous pouvons conclure qu'aucune des deux approches de conception d'algorithmes d'ordonnancement de DAGs de tâches parallèles sur plates-formes multi-grappes n'est meilleure vis-à-vis de l'autre. Les heuristiques en deux étapes qui tentent de trouver un bon compromis entre le *makespan* des applications et l'efficacité peuvent être préférées par de nombreux utilisateurs. L'avantage de nos améliorations de MHEFT est qu'elles peuvent être paramétrées afin de trouver le compromis souhaité entre le *makespan* et l'efficacité. Cette approche peut donc être adoptée par certains utilisateurs avancés.

4.7 Conclusion

Dans ce chapitre, nous avons étudié l'ordonnancement de DAGs de tâches parallèles sur les agrégations de grappes homogènes.

Nous avons dans un premier temps proposé deux algorithmes pour les plates-formes multi-grappes quasi homogènes. L'un de ces algorithmes est une heuristique pragmatique non garantie tandis le second, que nous avons nommé MCGAS, est un algorithme offrant sur le plan du *makespan* une garantie de performance au pire cas par rapport au meilleur ordonnancement. Cela nous a permis de réaliser la première étude comparative entre un algorithme garanti issu de la théorie et une heuristique pragmatique dans le cas de l'ordonnancement de DAGs de tâches modelables. Nous avons obtenu que, non seulement les performances au pire cas sont garanties pour l'algorithme MCGAS, mais aussi les simulations montrent qu'il donne en moyenne de meilleurs *makespans* par rapport à l'heuristique pragmatique. L'inconvénient de l'algorithme MCGAS est qu'il utilise un programme linéaire qui peut mettre beaucoup de temps pour calculer les allocations. Ce temps d'exécution de MCGAS croît de manière quadratique avec le nombre de tâches des DAGs. Mais cet algorithme peut être bénéfique si l'application ordonnancée est telle que le temps de calcul de l'ordonnancement est négligeable devant le *makespan* ou si l'ordonnancement calculé, peut être réutilisé à plusieurs reprises par une même application sur une plate-forme de production.

Ensuite, nous avons proposé des heuristiques pour des agrégations hétérogènes de grappes homogènes en vue de comparer deux approches orthogonales de conception d'heuristiques d'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes à partir des travaux existants. L'une de ces approches consiste à adapter aux plates-formes hétérogènes des algorithmes d'ordonnancement de DAGs de tâches parallèles en milieux homogènes. Cette approche étant jusqu'alors inexplorée, nous avons adapté les heuristiques en deux étapes proposées dans le chapitre précédent aux plates-formes hétérogènes multi-grappes. L'autre approche, utilisée par l'heuristique MHEFT [26], consiste à adapter aux tâches parallèles des algorithmes conçus pour l'ordonnancement de DAGs de tâches séquentielles en milieu hétérogène. Ayant détecté des faiblesses dans l'algorithme MHEFT, nous lui avons proposé des améliorations avant de comparer les performances des heuristiques issues des deux approches. Les résultats des simulations ont montré que l'une de ces améliorations donne en moyenne des performances strictement supérieures à MHEFT. Par rapport à MHEFT qui consomme trop de ressources, les autres améliorations trouvent un meilleur compromis entre le *makespan* et l'efficacité.

La comparaison des heuristiques issues des deux approches nous a permis de conclure qu'aucune de ces deux approches n'est strictement plus performante que l'autre sur le plan de la réduction du *makespan* des applications et sur le plan de l'efficacité dans l'utilisation des ressources. La tendance globale est que les heuristiques en deux étapes fournissent un bon compromis entre l'utilisation des ressources et le *makespan*. L'avantage des améliorations que nous avons proposées pour MHEFT est qu'elles sont paramétrables et qu'elles peuvent servir à des utilisateurs avancés qui souhaitent gérer eux mêmes le compromis entre l'utilisation des ressources et le *makespan*. Par exemple, sur une plate-forme où l'utilisation des ressources est « payante », un utilisateur à « budget limité » peut tenter d'obtenir la meilleure efficacité possible qui lui permettra d'exécuter son application dans le respect du délai qu'il s'est fixé.

Dans le chapitre suivant, nous nous inspirerons des heuristiques en deux étapes que nous venons de proposer et leurs procédures de placement dynamiques pour mettre en œuvre des heuristiques d'ordonnancement concurrent de plusieurs DAGs de tâches parallèles sur les agrégations hétérogènes de grappes homogènes. Ces nouvelles heuristiques devront être également peu complexes et permettre une utilisation efficace des ressources pour chaque DAG. Nous met-

trons en place de nouvelles procédures qui permettrons de gérer l'équité entre les applications en milieu partagé.

Chapitre 5

Ordonnancement concurrent de graphes de tâches modelables sur plates-formes hétérogènes multi-grappes

Sommaire

5.1 Introduction

Dans le chapitre précédent, nous avons proposé des heuristiques en deux étapes qui permettent d'ordonnancer les DAGs de tâches parallèles sur ces plates-formes multi-grappes en supposant que le DAG ordonnancé puisse disposer sans partage de toutes les ressources de la plate-forme. Or les plates-formes multi-grappes telles que Grid'5000 [25, 55] sont intrinsèquement partagées par de nombreuses applications. Les heuristiques qui considèrent que chaque application dispose intégralement des ressources conduisent à des ordonnancements « individualistes » pouvant nécessiter de nombreuses ressources. Sur une plate-forme partagée, ces ordonnancements peuvent avoir un impact négatif sur le temps de complétion des applications car l'exécution d'une application qui nécessite de nombreuses ressources peut être retardée ou peut retarder des applications concurrentes. Il est donc nécessaire de tenir compte de ce caractère partagé afin de gérer l'équité entre des applications concurrentes tout en essayant de réduire leurs *makespans*. Dans cette étude, nous utilisons la notion d'équité telle qu'elle est définie par Zhao

et Sakellariou [120]. Ceux-ci considèrent qu'un ordonnancement multi-DAGs est équitable si les dégradations du *makespan* de chacun des DAGs en concurrence, par rapport à leurs *makespans* respectifs si chaque DAG disposait à lui seul de la plate-forme, sont égales.

Une des méthodes qui permettent de gérer les plates-formes partagées est d'utiliser des algorithmes d'ordonnancement concurrents de plusieurs applications. Mais, nous avons vu dans le chapitre 2 que les travaux d'ordonnancement concurrent d'applications modélisées par des DAGs n'ont été réalisés que pour les tâches séquentielles [32, 60, 120]. Nous proposons donc, dans ce chapitre, les premières heuristiques d'ordonnancement concurrent de DAGs de tâches parallèles. Nous utiliserons les modèles de plates-formes et d'applications décrits dans la section 4.2.

Nos nouvelles heuristiques seront inspirées des heuristiques en deux étapes que nous avons proposées dans les chapitres précédents car ces dernières ont une faible complexité et grâce à leurs procédures de placement dynamiques, elles souffriront moins des écarts entre la valeur prédite et la valeur réelle de la date de début d'exécution des tâches, ces erreurs de prédiction étant dues à la non prise en compte des contentions par les heuristiques utilisées. En effet, ces procédures ne décidant du placement d'une tâche qu'une fois celle-ci est prête, les erreurs de prédiction se propageront moins entre les tâches comparativement à des techniques de placement statiques (où toutes les tâches sont placées avant le lancement de la première tâche). Afin de gérer l'équité entre des applications sur une plate-forme partagée, il peut être intéressant de limiter l'utilisation des ressources par chaque application. Par conséquent, nous proposons deux procédures d'allocation qui permettent à un utilisateur qui soumet son application ou à un méta-ordonnanceur qui gère l'accès entre différentes grappes de fixer le taux d'utilisation de la plate-forme à ne pas excéder. Ces deux procédures d'allocation se distinguent par la manière dont la contrainte sur l'utilisation des ressources est définie. Pour la phase de placement, nous proposons trois heuristiques de liste qui se différencient par leur technique de prioritisation des tâches.

Dans la section suivante, nous décrirons et évaluerons nos nouvelles procédures d'allocation sous contrainte de ressources. Ensuite nous présenterons trois procédures de placement dynamiques pour le placement concurrent de plusieurs DAGs de tâches parallèles déjà alloués. Enfin, les différentes heuristiques d'ordonnancement multi-DAGs obtenues à partir des nouvelles procédures d'allocation et de placement seront évaluées en supposant que tous les DAGs ont été soumis au même moment.

5.2 Procédures d'allocations sous contrainte de ressources

Nous proposons dans cette section de limiter les ressources utilisées par chaque application dans un environnement partagé en vue d'exécuter simultanément et de manière efficace plusieurs DAGs de tâches parallèles. Pour ce faire, nous nous focalisons sur la phase d'allocation d'heuristiques en deux étapes telles que celles proposées dans les chapitre précédents.

La responsabilité de fixer la contrainte d'utilisation des ressources peut être laissée à l'utilisateur qui soumet son application ou à un méta-ordonnanceur. Si les utilisateurs sont libres de fixer le taux d'utilisation des ressources, on a alors affaire à une gestion plus souple de la plate-forme. Mais cette gestion peut avoir comme inconvénient des comportements individualistes de certains utilisateurs pouvant entraîner une mauvaise utilisation des ressources. En revanche, un méta-ordonnanceur peut adapter la contrainte de ressources à chaque application en fonction de la charge de la plate-forme.

Plusieurs possibilités se présentent pour définir la contrainte d'utilisation des ressources. Cette contrainte pourrait être un nombre de processeurs à ne pas dépasser durant l'exécution

d'une application. Ce nombre de processeurs peut être soit une valeur maximale, soit une valeur moyenne à ne pas franchir à l'issue de l'exécution de l'application. Mais sur des plates-formes hétérogènes, ce raisonnement qui ne tient compte que du nombre de processeurs utilisés n'est pas pertinent. En effet, ordonnancer un DAG sur 100 processeurs ayant chacun une vitesse de traitement de 1 Gflop/s n'équivaut pas à ordonnancer le même DAG sur 100 processeurs à 4 Gflop/s. La contrainte de ressources peut également s'exprimer en termes de portion de la puissance de calcul de la plate-forme à utiliser (au maximum ou en moyenne). Nous utiliserons cette seconde forme de contrainte qui est plus adaptée aux agrégations hétérogènes de grappes homogènes. Nous noterons β ($0 < \beta \leq 1$), ce taux d'utilisation des ressources :

$$\beta = \frac{\textit{Puissance utilisée}}{\textit{Puissance totale}}.$$

La question qui se pose maintenant est de savoir comment répartir cette portion de puissance entre les tâches de l'application à ordonnancer en vue de respecter la contrainte fixée. Nous proposons pour cela deux stratégies fondées sur la réduction du chemin critique du DAG et sur la notion de grappe de référence définie dans la section 4.4.1. Nous noterons P_{ref} le nombre total de processeurs de la grappe de référence, s_{ref} la vitesse (en Gfop/s) des processeurs de la grappe de référence et $p^{ref}(t)$ l'allocation d'une tâche sur la grappe de référence. Afin d'éviter le dépassement de la contrainte imposée, l'allocation effective d'une tâche sur une grappe C_k donnée sera déterminée en arrondissant le nombre trouvé par la fonction f de l'équation 4.9 au nombre entier inférieur ou égal à $f(p^{ref}(t), t, k)$ si ce nombre est supérieur à 1. Si $f(p^{ref}(t), t, k)$ est inférieur ou égal à 1, alors on alloue évidemment un processeur à la tâche t afin qu'elle puisse être exécutée.

5.2.1 SCRAP (*Self-Constrained Resource Allocation Procédure*)

Le but de notre première procédure d'allocation nommée SCRAP est de déterminer les allocations des tâches dans le respect de la contrainte β. Comme dans toutes nos heuristiques en deux étapes proposées dans les chapitres précédents, cette procédure initialise les allocations en allouant un processeur à chaque tâche. Ensuite, tant que la contrainte est respectée, à chaque itération, on alloue un processeur supplémentaire à l'allocation de référence de la tâche du chemin critique qui en bénéficie le plus. À l'instar de nos autres heuristiques en deux étapes, cette tâche critique est celle dont le ratio $T^{ref}(t, p^{ref}(t))/p^{ref}(t)$ diminue le plus si un processeur supplémentaire lui est attribué.

La violation de la contrainte β est détectée de la manière suivante. Soit $w(t,(t))$ une estimation du travail qu'effectuerait une tâche t avec son allocation courante, c'est-à-dire le produit de son temps d'exécution par la puissance de calcul utilisée. On note $W = \sum_{t \in \mathcal{V}} w(t,(t))$ le travail total qui serait effectué par l'application si les tâches sont placées avec leurs allocations courantes. Afin d'estimer la quantité moyenne de ressources consommée durant l'exécution de l'application, nous divisons W par la longueur du chemin critique du DAG T_{CP}. En effet cette longueur est une borne inférieure du *makespan* du DAG et elle représente l'estimation courante de ce *makespan*. Ainsi, durant la phase d'allocation, nous pouvons estimer le taux d'utilisation des ressources β' en faisant le rapport de W/T_{CP} sur la puissance totale de la plate-forme. Lorsque β' devient supérieur à β, cela signifie que la contrainte sur l'utilisation des ressources sera violée. Si β' est supérieur à β dès l'initialisation des allocations, alors on a affaire à une contrainte trop forte et SCRAP n'alloue pas de processeurs supplémentaires aux tâches.

Tout comme dans nos autres heuristiques utilisant la notion de grappe de référence, nous arrêtons la procédure d'allocation dès que le chemin critique est saturé, c'est-à-dire si les allocations

de référence sont telles qu'il n'est plus possible de réduire le chemin critique. L'algorithme 7 résume SCRAP. Notons que lorsque β est égal à 1, la condition $\beta' < \beta$ est équivalente à la condition $T_{CP} > T_A$ utilisée dans nos heuristiques HCPA et SHCPA présentées dans le chapitre précédent. En effet, on a :

$$\begin{aligned}
\beta' &= \frac{W}{T_{CP} \times P_{ref} \times s_{ref}} \\
&= \frac{\sum_{t \in \mathcal{V}} w(t, p(t))}{T_{CP} \times P_{ref} \times s_{ref}} \\
&= \frac{\sum_{t \in \mathcal{V}} \left(T^{ref}(t, p^{ref}(t)) \times p^{ref}(t) \times s_{ref} \right)}{T_{CP} \times P_{ref} \times s_{ref}} \\
&= \frac{T_A}{T_{CP}}
\end{aligned}$$

Algorithme 7 SCRAP

1: **pour tout** $t \in \mathcal{V}$ **faire**
2: $p^{ref}(t) \leftarrow 1$
3: **fin pour**
4: **tant que** $\beta' < \beta$ et chemin critique non saturé **faire**
5: $t \leftarrow$ {tâche critique $|(\exists C_k| \ \lfloor f(p^{ref}(t), t, k) \rfloor < P_k)$ et $\left(\frac{T^{ref}(t, p^{ref}(t))}{p^{ref}(t)} - \frac{T^{ref}(t, p^{ref}(t)+1)}{p^{ref}(t)+1} \right)$ est maximum}
6: $p^{ref}(t) \leftarrow p^{ref}(t) + 1$
7: Mettre à jour β'
8: **fin tant que**

5.2.2 SCRAP-MAX

Dans notre seconde procédure d'allocation sous contrainte de ressources, nous tenons compte du niveau de précédence des tâches. Nous rappelons que le niveau de précédence d'une tâche t est un entier a tel que tous les prédécesseurs de t ont un niveau de précédence strictement inférieur à a et qu'au moins un des prédécesseurs de t a le niveau de précédence $a-1$.

Nous avons remarqué que lors de la phase de placement des heuristiques d'ordonnancement en deux étapes, les tâches prêtes concurrentes appartiennent généralement à un même niveau de précédence. Ainsi, afin d'exécuter plus de tâches en parallèle, nous décidons de restreindre les allocations à chaque niveau de précédence. Nous ajoutons alors une contrainte supplémentaire à SCRAP de sorte que la puissance maximale utilisée par toutes les tâches d'un même niveau de précédence doit être inférieure ou égale à $\beta \times P_{ref} \times s_{ref}$.

Cette contrainte supplémentaire influe sur le choix de la tâche la plus critique. Une tâche t du chemin critique peut être sélectionnée si d'une part la somme des puissances allouées à toutes les tâches de son niveau de précédence t incluse, $(plev\_Alloc(t) \times s_{ref})$ est inférieure à $\beta \times P_{ref} \times s_{ref}$ et si d'autre part, elle représente la tâche qui bénéficiera le plus d'un processeur supplémentaire. L'algorithme 8 présente notre seconde procédure d'allocation que nous nommerons SCRAP-MAX.

Avant d'aller plus loin et de proposer d'ordonnancer plusieurs DAGs en concurrence sur les plates-formes hétérogènes multi-DAGs, nous allons évaluer dans la section suivante si l'utilisation des nouvelles procédures d'allocation permet de respecter la contrainte de ressources fixée initialement.

Algorithme 8 SCRAP-MAX

pour tout $t \in \mathcal{V}$ **faire**
  $p^{ref}(t) \leftarrow 1$
fin pour
tant que $\beta' < \beta$ et chemin critique non saturé **faire**
  $t \leftarrow \{$ tâche critique $\mid (\exists C_k \mid \lfloor f(p^{ref}(t), t, k) \rfloor < P_k)$ et $(plev\_Alloc(t) \times s_{ref}) < (\beta \times P_{ref} \times s_{ref})$ et $\left(\frac{T^{ref}(t,p^{ref}(t))}{p^{ref}(t)} - \frac{T^{ref}(t,p^{ref}(t)+1)}{p^{ref}(t)+1}\right)$ est maximum$\}$
  $p^{ref}(t) \leftarrow p^{ref}(t) + 1$
  Mettre à jour β'
fin tant que

5.2.3 Évaluation du respect de la contrainte

L'évaluation du respect de la contrainte β s'effectue en mesurant le taux moyen d'utilisation des ressources durant l'exécution de chaque DAG. Ce taux est donné par le rapport du travail total effectif sur le *makespan*, le tout divisé par la puissance totale de la plate-forme. Nous considérons ici que l'application dispose à elle seule de la totalité des ressource de la plate-forme. Pour la phase de placement, après avoir déterminé les allocations avec SCRAP ou SCRAP-MAX, nous utilisons notre heuristique de liste fondée sur le chemin critique, présentée dans le chapitre précédent. Dans cette section, nous n'utiliserons pas la technique de tassage pour ces évaluations afin de ne pas influer sur le taux d'utilisation de la plate-forme en diminuant les allocations de certaines tâches.

Les simulations ont été effectuées avec les mêmes DAGs et plates-formes que ceux utilisés dans le chapitre précédent.

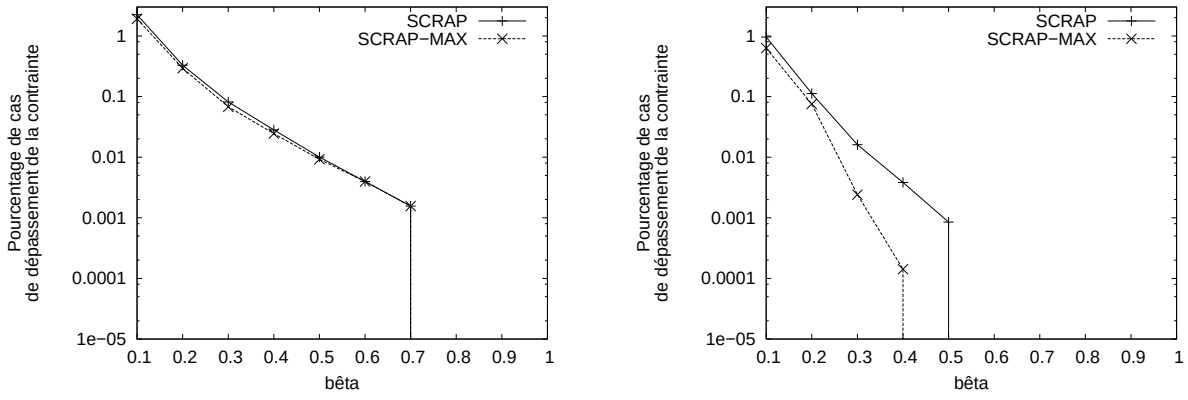

FIG. 5.1 – Pourcentage des cas de violation de la contrainte par SCRAP et SCRAP-MAX pour toutes les simulations (à gauche) et lorsqu'on exclue les cas où la contrainte est violée dès l'initialisation des allocations (à droite).

La figure 5.1 présente les pourcentages des cas où la contrainte β n'est pas respectée à l'issue de l'ordonnancement. Nous avons fait varier β de $0,1$ à 1 avec un pas de $0,1$. Le graphique de gauche présente les cas de violation lorsque nous prenons en compte toutes les simulations. Dans le graphique de droite, les pourcentages sont calculés en excluant les cas où la contrainte est violée dès l'initialisation des allocations. On remarque qu'il y a sensiblement moins de cas de violation de β lorsque nous excluons les cas où la contrainte est violée dès l'initialisation des allocations. Globalement, le nombre de cas de violation diminue très rapidement lorsque la contrainte devient

moins forte (β augmente). On observe même qu'il n'y a aucun cas de dépassement à partir de $beta = 0,7$. Pour β supérieur ou égale à $0,2$, la contrainte est respectée dans plus de $99,6\%$ des cas aussi bien avec SCRAP qu'avec SCRAP-MAX. Pour une contrainte très forte ($\beta = 0,1$), nous obtenons également un nombre important de cas où celle-ci est respectée (plus de $97,7\%$ des cas). D'autre part on constate que SCRAP-MAX obtient un nombre de cas de dépassement légèrement inférieur à SCRAP. En effet, en plus de tenir compte de la contrainte de SCRAP qui limite le taux d'utilisation moyen des ressource, SCRAP-MAX intègre une contrainte supplémentaire en restreignant l'utilisation des ressources de chaque niveau de précédence d'un DAG.

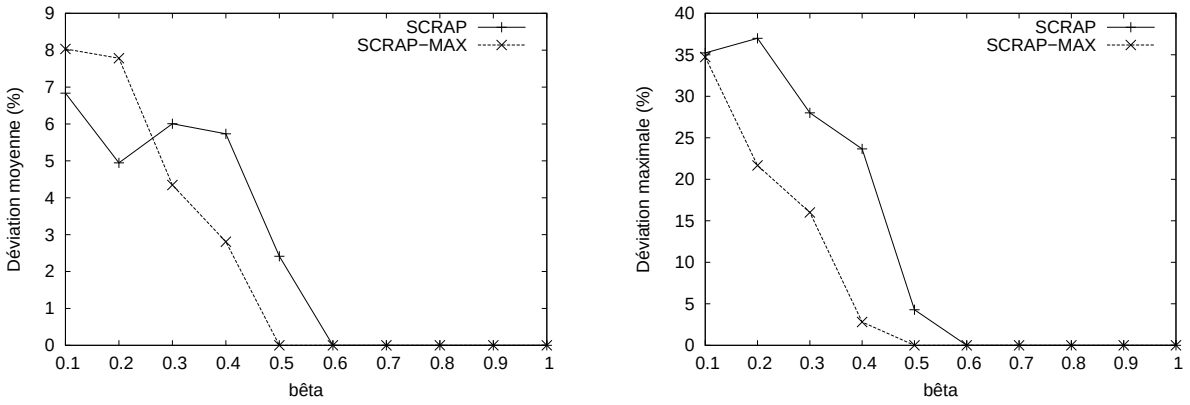

FIG. 5.2 – Déviations moyennes (à gauche) et maximales (à droite) par rapport à β pour les cas où la contrainte est violée, excluant les cas où la contrainte est violée dès l'initialisation des allocations.

Dans la figure 5.2, nous avons mesuré les déviations moyennes et maximales par rapport à la contrainte fixée lorsque celle-ci n'est pas respectée, en excluant les cas où β est violée dès l'initialisation des allocations. On constate que les déviations moyennes par rapport à β sont relativement faibles. Pour une contrainte forte ($\beta = 0,1$), cette déviation moyenne est au plus de 8%. Cela signifie qu'en général, le taux d'utilisation de la plate-forme s'écarte peu de la contrainte fixée en cas de dépassement de cette contrainte. Nous observons également que les déviations moyennes diminuent rapidement lorsque la contrainte devient moins forte.

Le graphique de droite de la figure 5.2 montre que pour des contraintes fortes, bien que nous ayons écarté les cas où la contrainte est violée dès l'initialisation des allocations sur la grappe de référence, la déviation au pire cas par rapport à β est relativement élevée. Pour $\beta = 0,1$, cette déviation maximale tend vers le degré d'hétérogénéité maximum des plates-formes utilisées qui est de 40%. Nous pouvons expliquer ce comportement par le fait que pour des cas extrêmes, certaines tâches qui ont une allocation de référence égale à 1 sur le processeur le plus lent, s'exécutent préférentiellement sur 1 processeur d'une grappe plus rapide. Cette erreur de discrétisation entraîne une diminution du *makespan* par rapport à la valeur estimée et cela a pour effet de d'augmenter le taux d'utilisation moyen de la plate-forme durant l'exécution de l'application. Lorsque β (ou le nombre de ressources de la plate-forme) est relativement élevé, ces erreurs sur le temps d'exécution des tâches sont moins importantes car une tâche peut se voir allouer sur chaque grappe un nombre de processeurs tel que son temps d'exécution est proche du temps d'exécution prédit sur la grappe de référence. Par conséquent, la déviation maximale diminue très rapidement lorsque β augmente.

Pour des plates-formes quasi homogènes, les erreurs concernant la prédiction des temps d'exécution des tâches sont plus faibles car les caractéristiques de tous les processeurs sont

proches de celles de la grappe de référence. C'est la raison pour laquelle l'on peut observer sur la figure 5.3 que la déviation maximale pour ces DAGs est inférieure à 2%.

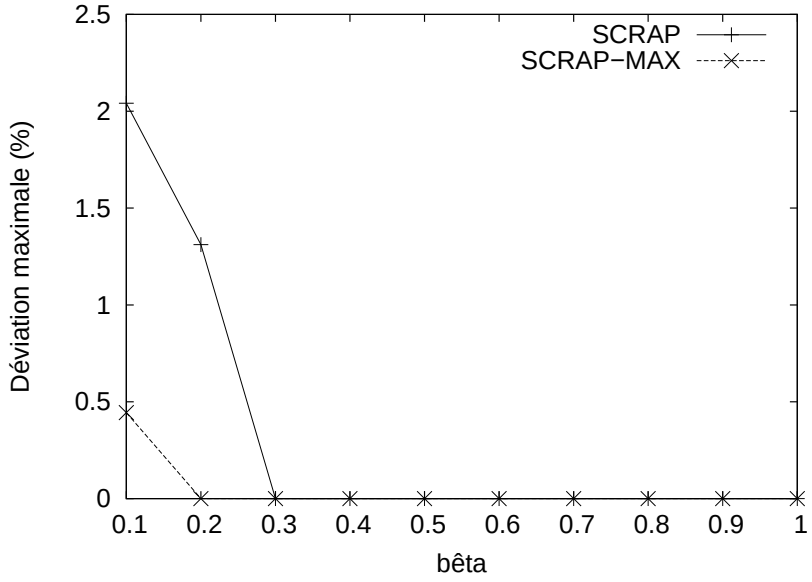

FIG. 5.3 – Déviations maximales par rapport à β pour les cas où la contrainte est violée dans le cas des plates-formes (quasi) homogènes ($h < 10$), excluant les cas où la contrainte est violée dès l'initialisation des allocations.

Dans le chapitre précédent, nous avions proposé l'heuristique MHEFT-MAX pour laquelle, sur une grappe donnée, aucune tâche ne peut utiliser un nombre de processeur supérieur à une certaine fraction (fixée) du nombre total de processeurs présents sur cette grappe. Cette manière de restreindre localement l'utilisation des ressources et de raisonner en termes de nombre de processeurs n'offre aucune garantie sur la puissance de calcul utilisée. Nous avons vérifié cela dans la figure 5.4 qui montre les cas de violation du taux d'utilisation des ressources par MHEFT-MAX. Pour β fixé, nous représentons le pourcentage des cas de violation de la contrainte globale par MHEFT-MAX-β. Nous observons que les cas de violation sont très élevés par rapport à SCRAP et SCRAP-MAX. Ce résultat conforte notre choix de l'expression de la contrainte en tant que ratio de la puissance totale de la plate-forme.

La figure 5.4 montre également le nombre de cas de violation par l'heuristique HCPA-OPT, conçue dans le chapitre précédent sans prendre en compte du caractère partagée de plate-forme. Bien que cette heuristique ait été proposée pour utiliser relativement peu de ressources, celle-ci ne respecte pas la contrainte β dans de nombreux cas. Cela confirme une fois de plus l'intérêt de proposer des heuristiques telles que SCRAP et SCRAP-MAX avec lesquelles on peut fixer le taux d'utilisation des ressources.

Les bons pourcentages des cas de respect de la contrainte de ressources par SCRAP et SCRAP-MAX nous encouragent à poursuivre notre étude pour l'ordonnancement concurrent de DAGs grâces à des heuristiques en deux étapes. Par la suite, nous utiliserons l'heuristique SCRAP-MAX pour effectuer les allocations car celle-ci obtient moins de cas de violation de la contrainte d'utilisation des ressources. Dans la section qui suit, nous proposons des procédures dynamiques pour le placement concurrent de DAGs de tâches parallèles déjà alloués.

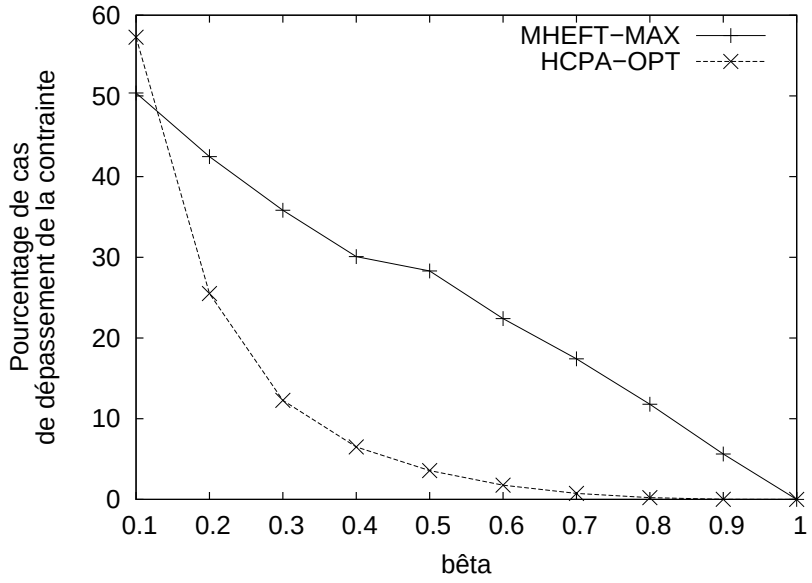

FIG. 5.4 – Pourcentage des cas de violation de la contrainte par MHEFT-MAX et HCPA-OPT.

5.3 Procédures pour le placement concurrent de DAGs

Pour exécuter concurremment plusieurs DAGs de tâches parallèles déjà alloués, nous proposons trois heuristiques de placement dynamiques s'appuyant sur la maintenance d'une liste de tâches prêtes. À chaque fois que de nouvelles tâches deviennent prêtes, elles sont triées relativement à une certaine priorité puis elles sont toutes placées en respectant cet ordre. Cette technique qui consiste à placer toutes les tâches prêtes immédiatement permet de moins retarder l'exécution des tâches de certains DAGs et évite des situations de famine. En effet, cela assure que toute tâche prête finira par s'exécuter sur la plate-forme, par conséquent tout DAG soumis au système finira par terminer son exécution. Afin de mieux utiliser les ressources en réduisant les trous dans les ordonnancements, ces procédures de placement intégreront la technique de tassage proposée dans le chapitre 3.

La première technique, FIFO (*First in First Out*), consiste à placer les tâches prêtes l'une après l'autre selon l'ordre de leur arrivée.

Avec la seconde technique que nous appellerons PRIO, à chaque instant où de nouvelles tâches sont prêtes, elles sont toutes placées l'une après l'autre dans l'ordre de leur *bottom-levels* décroissant. Les tâches dont le début d'exécution est le plus éloigné de la fin d'exécution du DAG correspondant sont donc les plus prioritaires parmi les tâches prêtes.

Notre dernière technique de placement ,SUFF, est inspirée de l'heuristique *sufferage* [74]. Parmi les tâches prêtes qu'il reste à placer, SUFF choisie celle qui serait la plus pénalisée s'il lui était attribué sa deuxième meilleure allocation au lieu de la première. Cette tâche qui « souffrirait » le plus est alors placée et retirée de la liste.

La combinaison de SCRAP-MAX et de ces procédures de placement dynamiques permet d'obtenir plusieurs heuristiques en deux étapes pour l'ordonnancement multi-DAGs. La question qui se pose désormais est comment limiter les ressources utilisées par chaque DAG. Dans la section suivante, nous proposons différentes stratégies en vue de limiter l'utilisation des ressources et nous évaluons les heuristiques d'ordonnancement multi-DAGs obtenues à partir de l'application de ces stratégies à SCRAP-MAX et des procédures de placement FIFO, PRIO et

SUFF.

5.4 Évaluation des heuristiques d'ordonnancement multi-DAGs

5.4.1 Méthodologie d'évaluation

Pour que l'impact des communications entre les tâches soit limité et qu'on appréhende mieux le comportement des heuristiques, nous nous restreignons à des plates-formes pour lesquelles la latence entre les différentes grappes est relativement faible. Nous utiliserons pour cela quatre plates-formes multi-grappes extraites de Grid'5000 [21, 55] pour lesquelles toutes les grappes sont localisées à l'intérieur d'un même site. Ces plates-formes multi-grappes mono-site sont présentées dans le tableau 5.1. Ce sont, l'ensemble des grappes de Lille, Nancy, Rennes et Sophia.

	Lille			Nancy		Rennes			Sophia				
	C3	C4	C5	C8	C9	C12	C13	C14	C15	C16	C17	Total	h
Lille	53	20	26									99	20, 2
Nancy				120	47							167	6, 1
Rennes						99	64	66				229	36, 8
Sophia									74	56	50	180	34, 7

Tab. 5.1 – Caractéristiques des plates-formes sélectionnées.

Plusieurs ensembles de DAGs ont été générés à partir des applications utilisées dans le chapitre précédent. Pour chacun type de DAGs (DAGs irréguliers, DAGs réguliers quelconques, DAGs réguliers de type FFT ou DAGs réguliers de type *Strassen*), nous avons généré aléatoirement des ensembles de DAGs où chaque ensemble $\mathcal{A}$ comprend $|\mathcal{A}|$ égal à 2, 4, 6, 8 ou 10 DAGs. Pour chaque type de DAGs et chaque cardinalité ($|\mathcal{A}|$) de ces ensembles, 25 combinaisons différentes de DAGs sont générées. Par exemple, nous avons généré 25 combinaisons de DAGs comprenant chacune 2 DAGs de type FFT.

Les heuristiques seront évaluées en mesurant pour chaque instance, le *makespan* global de l'ordonnancement et l'équité. Puisque dans ces évaluations, pour une instance donnée tous les DAGs sont soumis au même moment, le *makespan* global correspondra au *makespan* de la dernière application à terminer son exécution. Afin de ne pas biaiser les évaluations, nous proposons de normaliser les *makespans* globaux et de comparer plutôt les *makespans* globaux relatifs des heuristiques. Pour chaque instance, le *makespan* global relatif est calculé en faisant le rapport du *makespan* global obtenu sur la meilleure valeur obtenue par l'une des heuristiques, pour la même plate-forme et le même ensemble de DAGs.

Dans le domaine de l'ordonnancement, la notion d'équité a été considérée sous différentes formes [61, 75, 90, 91, 120]. Nous adopterons la définition utilisée par Zhao et Sakellariou dans [120] lorsqu'il étudient l'ordonnancement concurrent de DAGs de tâches séquentielles. Cette définition est fondée sur la dégradation du *makespan* de chaque DAG lorsqu'il est exécuté concurremment avec plusieurs autres DAGs, par rapport à son *makespan* si le DAG s'exécutait seul sur la plate-forme (sans contrainte d'utilisation des ressources). Un ordonnancement multi-DAGs sera considéré comme étant équitable si les dégradations des *makespans* de tous les DAGs sont proches. Pour une instance donnée, la dégradation d'un DAG a est donnée par :

$$\text{dégradation}(a) = \frac{M_{seul}(a)}{M_{multi}(a)}$$

où $M_{seul}(a)$ est le *makespan* que produirait le même algorithme si le DAG disposait à lui seul de toutes les ressources de la plate-forme, et $M_{multi}(a)$ est le *makespan* obtenu en présence d'autres DAGs.

Les valeurs attendues pour la dégradation doivent être normalement comprises entre 0 et 1, une faible valeur indiquant que le DAG considéré pâtit de la présence des autres DAGs. Toutefois, pour une faible proportion de nos simulations, il arrive que l'ordonnancement d'un DAG donne un *makespan* légèrement meilleur en présence d'autres DAGs par rapport au cas ou ce DAG utilise seul la plate-forme et est ordonnancé sans contrainte. Les cas de dégradations qui dépassent 1 sont essentiellement dûs à la non prise en charge des contentions dans nos heuristiques lors de la prédiction des temps de communications. Or la présence de contentions pendant l'exécution d'un DAG peut augmenter les temps de complétion de certaines tâches par rapport aux valeurs estimées. Alors, dans de rares situations où les volumes de données à échanger entre les tâches sont relativement élevés, nos heuristiques fournissent pour un DAG seul, un ordonnancement qui pourrait être amélioré si on réduisait les communications entre les grappes, sources de contentions. En présence de plusieurs DAGs, le DAG considéré peut être contraint de s'exécuter sur un nombre plus petit de grappes. Cela peut avoir pour effet bénéfique de réduire les temps de communications et aboutir à une dégradation supérieure à 1. L'autre cause de dégradations supérieures à 1 est le découplage entre l'allocation de processeurs aux tâches et leurs placements. Dans de très rares situations, la réduction des allocations avec l'application d'une contrainte de ressources permet de réduire le *makespan* par rapport à un ordonnancement sans contrainte en exploitant plus de parallélisme de tâches. Pour chaque heuristique, nous obtenons au plus $0,6\%$ de cas ou la dégradation est supérieure à 1. La dégradation maximale est de $1,7$ mais la dégradation moyenne en cas de dépassement de 1 est au plus égale à $1,08$ pour chaque heuristique. Cette dégradation moyenne prouve qu'en général, la dégradation ne s'écarte pas significativement de 1 en cas de dépassement. Au vue du faible pourcentage de cas de dégradations supérieures à 1 et de leur valeur moyenne proche de 1, nous pouvons conclure que ces valeurs inattendues n'influent pas sur les résultats de nos simulations.

À partir des dégradations des DAGs, la non équité d'un ordonnancement est définie par :

$$\text{non\_équité} = \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} |\text{dégradation} - \text{dégradation\_moyenne}|$$

où $\mathcal{A}$ est l'ensemble de DAGs ordonnancé et dégradation_moyenne est la dégradation moyenne des DAGs pour cette instance. Il faut noter que nous avons normalisé la mesure de non équité définit par Zhao et Sakellariou dans [120] en la ramenant aux nombre d'applications concurrentes ($|\mathcal{A}|$). Une faible valeur de la non équité signifie que la différence entre les dégradations est faible, autrement dit, que l'ordonnancement est équitable entre les DAGs en concurrence.

Quatre stratégies différentes seront employées pour limiter ou non les ressources utilisées par chaque DAG lors de l'ordonnancement concurrent de chaque ensemble de DAGs. La première stratégie que nous appellerons S1, comme dans des heuristiques d'ordonnancement mono-DAGs, consistera à allouer des ressources aux DAGs sans contrainte. Cela correspond à fixer la valeur de β à 1 pour chaque DAG. La seconde stratégie, S2, vise à être plus équitable entre les DAGs concurrents. Elle fixe la même contrainte $\beta = 1/|\mathcal{A}|$ à chaque DAG. Puisque les DAGs peuvent avoir des quantités de calcul relativement différentes, il peut s'avérer intéressant de leur appliquer des contraintes différentes si l'on souhaite minimiser le *makespan* global de chaque ensemble de DAGs. Pour cela nous proposons une troisième stratégie, S3 qui consiste à fixer pour chaque DAG une contrainte proportionnelle à la quantité totale de calcul à effectuer. la somme des β étant égale à 1 comme dans la seconde stratégie S2. La dernière stratégie S4 vise à trouver un

compromis entre l'équité et le *makespan* global. Avec cette dernière stratégie une portion des ressources ($x \,|\, 0 \leq x \leq 1$) est partagée équitablement entre les DAGs et l'autre portion $(1 - x)$ est partagée aux DAGs proportionnellement au volume de calcul de chaque DAG. Si V_i est le volume de calcul du i^{e} DAG de l'ensemble à ordonnancer, la contrainte imposée à ce DAG est :

$$\beta_i = \frac{x}{|\mathcal{A}|} + \frac{(1-x) \times V_i}{\sum_{k=1}^{|\mathcal{A}|} V_k},$$

où $|\mathcal{A}|$ est le nombre de DAGs de cet ensemble. Cette stratégie garantit ainsi que β_i sera au moins supérieur à $x/|\mathcal{A}|$.

5.4.2 Résultats des simulations

Avant de comparer les différentes heuristiques, nous avons mesuré l'impact du choix de la valeur de x dans la stratégie S4 sur l'équité et le *makespan* global pour les DAGs irréguliers en utilisant PRIO dans la phase de placement. La figure 5.5 montre l'évolution de la non équité et du *makespan* en fonction de x pour chaque nombre de DAGs concurrents. On observe que globalement la non équité diminue lorsque x croît. Ce comportement est celui que nous attendions car plus on fait croître x, plus la proportion des ressources qui est partagée équitablement est importante. Mais nous observons que l'augmentation de x a pour effet d'augmenter le *makespan* global. Cela est dû au fait que les DAGs qui ont un plus grand volume de calcul se voient allouer moins de ressources lorsque x augmente. Or ces DAGs ont un impact plus important sur le *makespan* global car ils mettent plus de temps à s'exécuter. La réduction des ressources utilisées par ces DAGs a donc pour inconvénient la dégradation du *makespan* global.

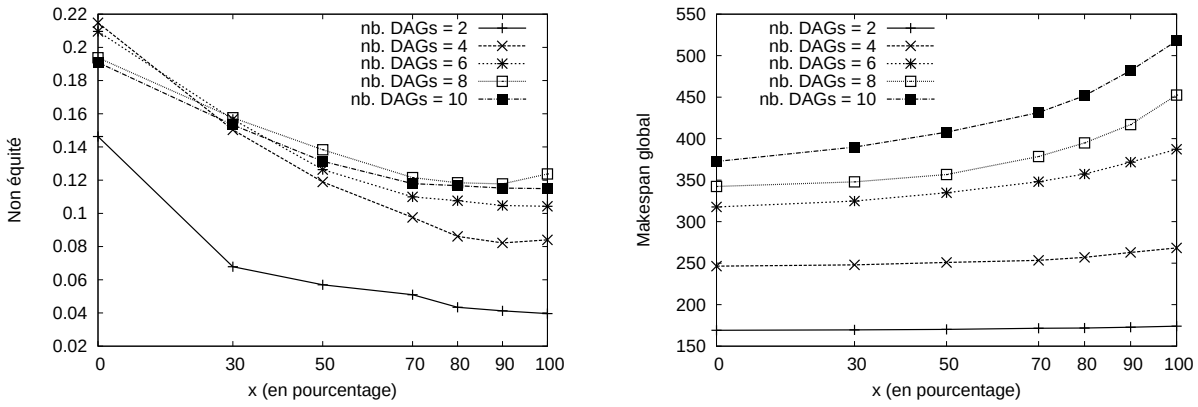

FIG. 5.5 – Évolution de la non équité moyenne (à gauche) et du *makespan* global moyen (à droite) en fonction de x pour les ensembles de DAGs irréguliers, pour chaque nombre de DAGs concurrents.

L'évolution de l'équité et du *makespan* global étant antagonistes, le choix de x est déterminant si l'on souhaite un bon compromis entre ces deux grandeurs. Nous observons dans la figure 5.5 que $x = 70\%$ représente un bon compromis pour les DAGs étudiés. En effet l'équité s'améliore très peu à partir de $x = 70\%$ tandis que le *makespan* global croît plus rapidement lorsque x devient plus grand que 70%. Nous utiliserons donc, dans la suite de ces évaluations, $x = 70\%$ pour la stratégie S4 de limitation des ressources.

La figure 5.6 présente la non équité moyenne et le *makespan* global moyen des heuristiques pour les ensembles de DAGs irréguliers, la technique de placement utilisée étant PRIO. Nous

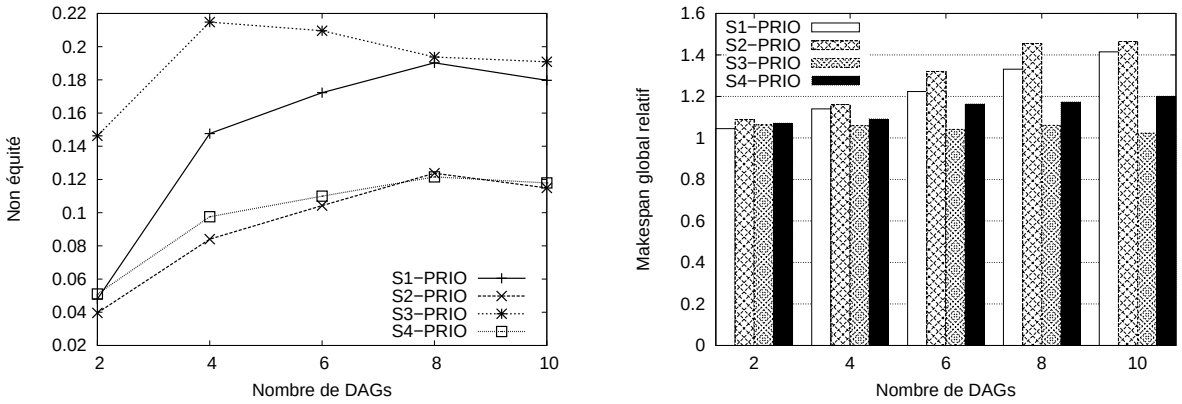

FIG. 5.6 – Non équité moyenne (à gauche) et *makespan* global moyen (à droite) des heuristiques en fonction du nombre de DAGs pour les ensembles de DAGs irréguliers.

observons que les stratégies S2 et S4 sont les plus compétitives du point de vue de l'équité. En effet, ces deux stratégies restreignent les allocations et les taux de ressources utilisées par les différentes applications sont relativement proches. En outre, nous observons que l'équité obtenue par ces deux stratégies atteignent un plateau lorsque le nombre de DAGs augmente. Nous pouvons donc conclure que S2 et S4 conservent une bonne équité quel que soit le nombre de DAGs.

Pour une équité équivalente entre S2 et S4, nous observons que le *makespan* relatif de S4 est jusqu'à 17% meilleur, cette dernière allouant plus de ressources aux DAGs qui ont un plus grand volume de calcul à réaliser. Avec la stratégie S1, chaque DAG tente d'occuper toutes les ressources à chaque instant. Les DAGs qui nécessitent peu de calcul peuvent donc se voir retardés considérablement par des DAGs qui ont des volumes de calcul importants. S1 est alors moins équitable que S2 et S4. La stratégie S3 est la moins équitable car certains DAGs (ceux qui ont moins de calcul à effectuer) ont des restrictions plus fortes que les autres sur l'utilisation des ressources. L'importante réduction des ressources pour ces DAGs leur confère une nette dégradation de leur *makespan* par rapports aux DAGs volumineux. Mais du fait qu'elle privilégie les DAGs les plus volumineux, cette stratégie obtient les meilleurs *makespans* globaux.

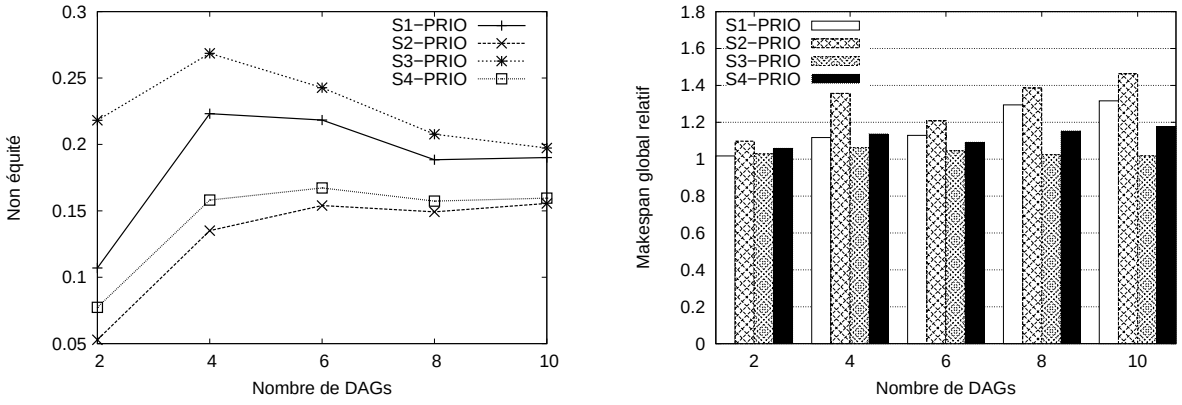

FIG. 5.7 – Non équité moyenne (à gauche) et *makespan* global moyen (à droite) des heuristiques en fonction du nombre de DAGs pour les ensembles de DAGs réguliers aléatoires.

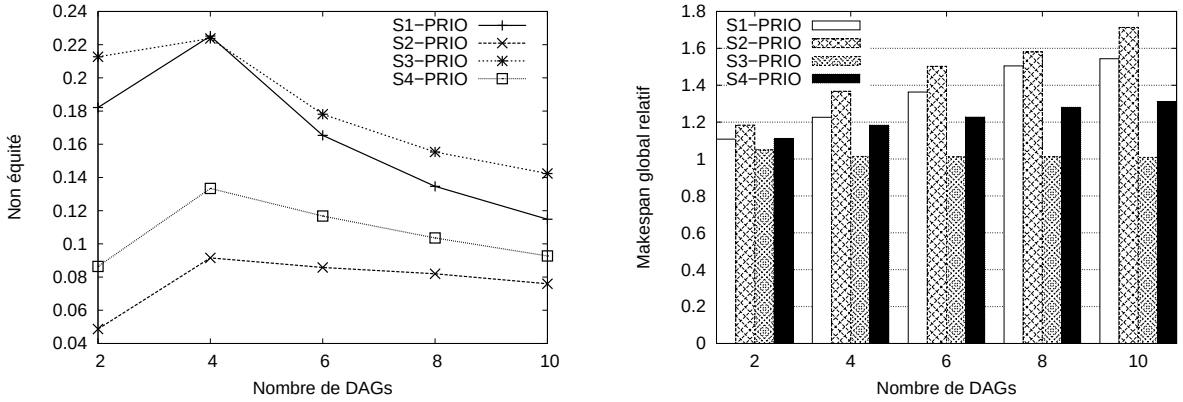

FIG. 5.8 – Non équité moyenne (à gauche) et *makespan* global moyen (à droite) des heuristiques en fonction du nombre de DAGs pour les ensembles de DAGs de type Strassen.

Dans les figures 5.7 et 5.8, nous observons que S3 est également la stratégie la moins équitable pour les DAGs réguliers aléatoires et de type *Strassen* mais elle obtient les meilleurs *makespans* globaux. S4 est la seconde stratégie la plus équitable, de peu derrière S2 et est également la seconde stratégie la plus performante au niveau du *makespan* global.

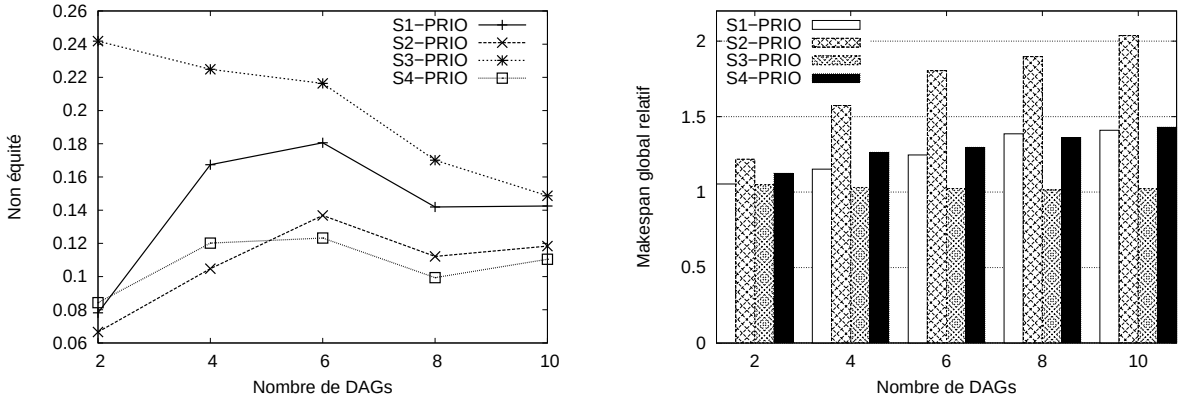

FIG. 5.9 – Non équité moyenne (à gauche) et *makespan* global moyen (à droite) des heuristiques en fonction du nombre de DAGs pour les ensembles de DAGs de type FFT.

Dans la figure 5.9 où nous avons représenté les performances des heuristiques pour les DAGs de type FFT, la stratégie S4 est encore meilleure que S2 car elle est globalement la plus équitable et elle donne de meilleurs *makespans* globaux. Les *makespans* globaux relatifs obtenus avec ces DAGs confirment le fait que les DAGs ayant un volume de calcul plus important sont limitant pour le *makespan* global. Pour les ensembles de DAGs de type FFT, le nombre de tâches dans une DAG peut varier fortement, de 5 (FFT 2 points) à 95 (FFT 16 points). Cela fait que certains DAGs peuvent avoir des volumes de calcul de plusieurs ordres de grandeurs supérieures à ceux des autres DAGs du même ensemble. La limitation des ressources avec S2 rallonge alors considérablement les *makespans* globaux par rapport aux autres stratégies où les DAGs les plus volumineux utilisent plus de ressources.

On observe également que pour les ensembles de DAGs de type FFT, la non équité obtenue avec la stratégie S3 diminue avec le nombre de DAGs. Cette non équité tant à s'améliorer car plus

il y a de DAGs, plus les DAGs dont les *makespans* sont fortement dégradés comparativement au DAG ayant le plus grand volume de calcul sont nombreux. Ces DAGs seront retardés presque de la même manière car au moins un processeur doit être alloué à chacune de leur tâches et il peut exister de nombreux DAGs pour lesquels les tâches ne s'exécutent que sur un processeur. Ces DAGs dont les dégradations des *makespans* sont similaires contribuent alors à l'amélioration de l'équité de S3 quand le nombre de DAGs augmente.

Nous avons réalisé les simulations avec les techniques de placement FIFO et SUFF et nous avons obtenu des résultats similaires à ceux de PRIO. Les techniques de placement ont donc un impact moindre que les différentes stratégies de restriction des ressources sur l'équité et le *makespan* global. Avec ces résultats, nous pouvons conclure que ne pas restreindre les ressources utilisées par les applications (stratégie S1) n'est ni bénéfique pour l'équité entre les applications, ni profitable du point de vue du *makespan* global. La stratégie S3 peut être préconisée si l'on cherche à exécuter rapidement un ensemble d'applications sans se soucier de l'équité. Si l'on cherche un ordonnancement équitable sans que l'ensemble des applications ne soit retardé, alors la stratégie S4 peut être choisie.

5.5 Conclusion

Les travaux réalisés dans ce chapitre constituent l'une des premières études concernant l'ordonnancement concurrent de DAGs de tâches parallèles. À notre connaissance, l'ordonnancement multi-DAGs n'a été effectué pour des tâches séquentielles. Nous avons proposé des heuristiques en deux étapes pour tenter de résoudre ce problème sur les plates-formes hétérogènes multi-grappes. Ces nouvelles heuristiques ont été inspirées des études effectuées dans le chapitre précédent.

Notre idée pour gérer la concurrence entre les applications a été de proposer deux procédures d'allocation qui permettent de restreindre les ressources utilisées par chaque application : SCRAP et SCRAP-MAX. L'évaluation de ces procédures a montré qu'elles permettent de respecter, à l'issue de l'ordonnancement, la contrainte de ressources initialement imposée. Il existe très peu de cas de violation de la contrainte, généralement observés lorsque la restriction des ressources est forte.

Ensuite, pour la seconde étape, nous avons proposé des heuristiques de listes dynamiques. Ces heuristiques, du fait qu'elles placent les tâches dès que celles-ci sont prêtes, ont la particularité de ne pas causer de situations de famine si elles sont utilisées dans un environnement de production.

Nous avons proposé différentes stratégies de restriction des ressources utilisées par les différents DAGs. SCRAP-MAX ayant moins de cas de violation de la contrainte, nous avons évalué les heuristiques d'ordonnancement multi-DAGs obtenues avec la combinaison de ces différentes stratégies appliquées à SCRAP-MAX et les techniques de placement proposées, en supposant que tous les DAGs sont soumis au même moment. Les résultats de nos simulations ont révélé que ne pas restreindre les allocations des applications dans un environnement partagé n'est ni bénéfique en ce qui concerne l'équité entre les applications en concurrence, ni optimal si l'on souhaite réduire le *makespan* global d'un ensemble de DAGs. Une des stratégies proposées pour les évaluations, consistant à imposer un taux d'utilisation de la plate-forme proportionnel aux nombre de DAGs, s'avère être plus équitable. Une autre stratégie conduisant aux meilleurs *makespans* globaux est d'imposer à chaque DAG d'utiliser un taux de ressources proportionnel à son volume de calcul. Si l'on souhaite trouver un compromis entre l'équité et le *makespan* global, une stratégie intéressante consiste à allouer une portion des ressources équitablement entre les différentes applications et l'autre portion, proportionnellement au volume de calcul de chaque application.

Cette étude a donc montré qu'il est possible de jouer sur la contrainte de ressources pour rechercher un compromis entre l'équité et les performances d'une plate-forme partagée. Elle constitue un premier pas dans l'ordonnancement concurrent de plusieurs DAGs de tâches parallèles. De nombreux autres problèmes restent ouverts tels que l'ordonnancement en ligne de DAGs de tâches parallèles. En effet, sur les plates-formes de production, des applications peuvent être soumises à tout moment.

Chapitre 6

Conclusion et perspectives

Dans cette thèse, nous nous sommes intéressés à l'ordonnancement d'applications parallèles représentées par des DAGs de tâches modelables sur les plates-formes hétérogènes formées par l'agrégation de plusieurs grappes de calcul homogènes. En effet, dans le chapitre 2, où nous avons étudié les plates-formes de calcul parallèle ainsi que les applications parallèles et fait un état de l'art des algorithmes d'ordonnancement, nous avons constaté que ces plates-formes ont été peu étudiées alors qu'elles se répandent de plus en plus et sont très attrayantes pour exécuter des applications parallèles.

L'objectif principal de ces travaux à été de concevoir des algorithmes d'ordonnancement qui tiennent explicitement compte du caractère partagé des grilles de calcul. Nous avons adopté une approche incrémentale en adaptant le principe d'un même algorithme d'ordonnancement à de nouvelles contraintes jusqu'à obtenir des algorithmes d'ordonnancement concurrent de plusieurs DAGs de tâches modelables sur des plates-formes hétérogènes multi-grappes.

Nous avons tout d'abord, dans le chapitre 3, étudié des algorithmes d'ordonnancement de DAGs de tâches modelables sur grappes homogènes déjà existants. Ces algorithmes peuvent être classés en deux catégories : les algorithmes en une étape dont la phase d'allocation de processeurs aux tâches et la phase de placement des tâches sont couplées, et les algorithmes en deux étapes dont les deux phases s'effectuent séparément. Les meilleurs algorithmes des deux catégories ont été évalués grâce à de nombreuses simulations. Des améliorations ont été apportées à l'une des heuristiques en deux étapes afin qu'elle soit adaptée aux agrégations de grappes homogènes par la suite. Nous avons choisi cette heuristique car les études antérieures ont montré qu'elle présentait un bon compromis entre sa complexité et les *makespans* obtenus. La première amélioration porte sur sa phase d'allocation et la seconde concerne sa phase de placement. Ces deux améliorations peuvent donc être combinées. Les évaluations ont montré que les trois nouvelles heuristiques que nous avons ainsi obtenues sont plus performantes que les heuristiques en deux étapes concurrentes étudiées. L'heuristique en une étape étudiée fournit en général de meilleurs *makespans* que nos nouvelles heuristiques mais celle-ci est plus complexe. Cependant nos heuristiques obtiennent un meilleur compromis entre l'efficacité et le *makespan* tout en s'avérant moins complexes.

Dans le chapitre 4, nous avons conçu des algorithmes d'ordonnancement de DAGs de tâches parallèles sur des plates-formes désormais formées de plusieurs grappes homogènes. Nous avons dans un premier temps proposé deux algorithmes pour les plates-formes quasi homogènes en termes de vitesse des processeurs. Le premier est une heuristique pragmatique non garantie tandis le second est un algorithme offrant sur le plan du *makespan*, une garantie de performance au pire cas par rapport au meilleur ordonnancement. Cela nous a permis de réaliser la première étude

comparative entre un algorithme garanti issu de la théorie et une heuristique pragmatique dans le cas de l'ordonnancement de DAGs de tâches modelables. Nous avons obtenu que, non seulement les performances au pire cas sont garanties pour le second algorithme, mais les simulations montrent également que cet algorithme donne en moyenne de meilleurs *makespans* par rapport à l'heuristique pragmatique. L'inconvénient de l'algorithme garanti est qu'il utilise un programme linéaire qui peut mettre beaucoup de temps pour calculer les allocations. Ce temps d'exécution croît de manière quadratique avec le nombre de tâches compris dans les DAGs étudiés. Mais cet algorithme peut être bénéfique si l'application ordonnancée est telle que le temps de calcul de l'ordonnancement est négligeable devant le *makespan* ou si l'ordonnancement calculé, peut être réutilisé à plusieurs reprises par une même application sur une plate-forme de production.

Ensuite, nous avons proposé des heuristiques pour des agrégations hétérogènes de grappes homogènes en vue de comparer deux approches orthogonales de conception d'heuristiques d'ordonnancement de DAGs de tâches parallèles sur plates-formes hétérogènes à partir des travaux existants. L'une de ces approches consiste à adapter aux plates-formes hétérogènes des algorithmes d'ordonnancement de DAGs de tâches parallèles en milieux homogènes. Cette approche étant jusqu'alors inexplorée, nous avons adapté les heuristiques en deux étapes proposées dans le chapitre 3 aux plates-formes hétérogènes multi-grappes. L'autre approche, utilisée par l'heuristique MHEFT [26], adapte aux tâches parallèles des algorithmes conçus pour l'ordonnancement de DAGs de tâches séquentielles en milieu hétérogène. Ayant détecté des faiblesses dans l'algorithme MHEFT, nous lui avons proposé des améliorations avant de comparer les performances des heuristiques issues des deux approches. Cette comparaison nous a permis de conclure qu'aucune de ces deux approches n'est strictement plus performante que l'autre sur le plan de la réduction du *makespan* des applications et sur le plan de l'efficacité dans l'utilisation des ressources. La tendance globale est que les heuristiques en deux étapes fournissent un bon compromis entre l'utilisation des ressources et le *makespan*. L'avantage des améliorations que nous avons proposées pour MHEFT est qu'elles sont paramétrables et qu'elles peuvent servir à des utilisateurs avancés qui souhaitent gérer eux mêmes le compromis entre l'utilisation des ressources et le *makespan*.

Enfin, dans le chapitre 5, nous avons réalisé l'une des premières études concernant l'ordonnancement concurrent de plusieurs DAGs de tâches modelables. Les algorithmes proposés jusqu'alors supposaient que l'application ordonnancée dispose à elle seule des ressources de la plate-forme. Les nouvelles heuristiques proposées dans ce chapitre ont été inspirées de nos heuristiques en deux étapes présentées dans le chapitre 4. Notre idée pour gérer la concurrence entre les applications a été de proposer au niveau de la phase d'allocation, deux heuristiques qui permettent de restreindre les ressources utilisées par chaque application : SCRAP et SCRAP-MAX. L'évaluation de ces procédures d'allocation a montré qu'elles permettent de respecter, à l'issue de l'ordonnancement, la contrainte de ressources initialement imposée. Nous avons alors poursuivi notre étude en proposant, pour la phase de placement, des heuristiques de listes dynamiques. Ces heuristiques, du fait qu'elles placent les tâches dès que celles-ci sont prêtes, ont la particularité de ne pas causer de situations de famine si elles sont utilisées dans un environnement de production. Différentes stratégies de restriction des ressources utilisées par les différents DAGs ont également été proposées. Puis, nous avons choisi d'évaluer les heuristiques d'ordonnancement multi-DAGs obtenues avec la combinaison de ces différentes stratégies appliquées à SCRAP-MAX et les techniques de placement proposées, en supposant que tous les DAGs sont soumis au même moment. Les résultats de nos simulations ont révélé que ne pas restreindre les allocations des applications dans un environnement partagé n'est ni bénéfique en ce qui concerne l'équité entre les applications en concurrence, ni optimal si l'on souhaite réduire le *makespan* global d'un ensemble de DAGs. Une des stratégies proposées pour les évaluations, consistant à

imposer un taux d'utilisation de la plate-forme proportionnel aux nombre de DAGs, s'avère être plus équitable. Une autre stratégie conduisant aux meilleurs *makespans* globaux est d'imposer à chaque DAG d'utiliser un taux de ressources proportionnel à son volume de calcul. Si l'on souhaite trouver un compromis entre l'équité et le *makespan* global, une stratégie intéressante consiste à allouer une portion des ressources équitablement entre les différentes applications et l'autre portion, proportionnellement au volume de calcul de chaque application.

Les perspectives de cette thèse sont nombreuses. En effet, elle ne constitue qu'un premier pas concernant l'ordonnancement concurrent de plusieurs DAGs de tâches modelables. À court terme, nous projetons d'étudier l'ordonnancement en ligne de DAGs de tâches modelables. Ce problème est plus réaliste car sur les plates-formes de production, les applications sont soumises continuellement. De nombreux défis seront à relever, en l'occurrence comment adapter la contrainte d'utilisation des ressources à l'état de la plate-forme, au « poids » de l'application ou encore à sa priorité.

À long terme, nous projetons d'intégrer nos algorithmes d'ordonnancement de DAGs de tâches modelables à des méta-ordonnanceurs ou à des intergiciels tels que DIET [40].

Annexe A

Publications personnelles

Revues internationales avec comité de lecture

[1] P.-F. Dutot, T. N'takpé, F. Suter and H. Casanova. Scheduling Parallel Task Graphs on (Almost) Homogeneous Multi-cluster Platforms. To appear in *IEEE Transactions on Parallel and Distributed Systems*, 2009.

Revues nationales avec comité de lecture

[2] T. N'takpé. Heuristiques d'ordonnancement en deux étapes de graphes de tâches parallèles. *Technique et Science Informatiques (TSI)*, 28(1/2009) :75-79, January 2009.

Conférences internationales avec comité de lecture

[3] T. N'takpé and F. Suter. Concurrent Scheduling of Parallel Task Graphs on Multi-Clusters Using Constrained Resource Allocations. To appear in *Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-09)*, Rome, Italy, May , 2009.

[4] T. N'takpé and F. Suter. Self-Constrained Resource Allocation Procedures for Parallel Task Graph Scheduling on Shared Computing Grids. In *Proceedings of 19th IASTED Parallel and Distributed Computing and Systems (PDCS 2007)*, Cambridge, USA, November, 2007.

[5] T. N'takpé and F. Suter and H. Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In *Proceedings of the 6th International Symposium on Parallel and Distributed Computing*, Hagenberg, Austria, July, 2007.

[6] T. N'takpé and F. Suter. Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms. In *Proceedings of the Twelfth International Conference on Parallel and Distributed Systems (ICPADS)*, Minneapolis, MN, USA, July, 2006.

Conférences nationales avec comité de lecture

[7] Tchimou N'takpé. Algorithmes d'ordonnancement de graphes de tâches parallèles sur plates-formes hétérogènes. In *17ième Rencontres francophones du Parallélisme (RenPar'17)*, Perpignan, October, 2006.

Annexe B

Bibliographie

Références

[8] I. Ahmad and Y. Kwok. A New Approach to Scheduling Parallel Programs Using Task Duplication. In *Proceedings of the 1994 International Conference on Parallel Processing (ICPP '94)*, pages 47–51, Washington, DC, USA, 1994. IEEE Computer Society.

[9] The Globus Alliance. `http://www.globus.org`.

[10] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS 1967 Spring Joint Computer Conference*, volume 30, pages 483–485, April 1967.

[11] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Department Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[12] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling With Implicit Information in Distributed Systems. *SIGMETRICS Performance Evaluation Review*, 26(1) :233–243, 1998.

[13] R. Bajaj and D. P. Agrawal. Improving Scheduling of Tasks in a Heterogeneous Environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2) :107–118, 2004.

[14] S. Bansal, P. Kumar, and K. Singh. An Improved Two-step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines. *Parallel Computing*, 32(10) :759–774, 2006.

[15] J. Barbosa, C. Morais, R. Nobrega, and A. P. Monteiro. Static Scheduling of Dependant Parallel Tasks on Heterogeneous Clusters. In *Proceedings of the 4th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 1–8, Boston, MA, USA, September 2005. IEEE Computer Society Press.

[16] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling Divisible Loads on Star and Tree Networks : Results and Open Problems. *IEEE Transactions on Parallel and Distributed Systems*, 16(3) :207–218, March 2005.

[17] V. Bharadwaj, T. G. Robertazzi, and D. Ghose. *Scheduling Divisible Loads in Parallel and Distributed Systems*. Wiley-IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[18] E. Blayo, L. Debreu, G. Mounié, and D. Trystram. Dynamic Load Balancing for Ocean Circulation Model with Adaptive Meshing. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing(Euro-Par'99)*, pages 303–312, London, UK, 1999. Springer-Verlag.

[19] BOINC. `http://boinc.berkeley.edu`.

[20] S. H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, 30(3) :207–214, 1981.

[21] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touché. Grid'5000 : a Large Scale and Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, November 2006.

[22] V. Boudet, F. Desprez, and F. Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003.

[23] R. Buyya, M. Murshed, D. Abramson, and S. Venugopal. Scheduling Parameter Sweep Applications on Global Grids : a Deadline and Budget Constrained Cost-Time Optimization Algorithm. *Software - Practice and Experience*, 35(5) :491–512, 2005.

[24] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A Batch Scheduler With High Level Components. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, Cardiff, UK, May 2005.

[25] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000 : a Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *Proceedings of the 6th IEEE/ACM International Conference on Grid Computing (GRID 2005)*, pages 99–106, Seattle, USA, November 2005.

[26] H. Casanova, F. Desprez, and F. Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In *Proceedings of the 10th International Euro-Par Conference (Euro-Par 2004)*, volume 3149 of *Lecture Notes in Computer Science*, pages 230–237, Pisa, Italy, August 2004.

[27] H. Casanova, A. Legrand, and M. Quinson. SimGrid : a Generic Framework for Large-Scale Distributed Experimentations. In *Proceedings of the 10th IEEE International Conference on Computer Modelling and Simulation (UKSIM/EUROSIM'08)*, Cambridge, U.K., April 2008.

[28] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template : User-level middleware for the Grid. *Scientific Programming*, 8(3) :111–126, 2000.

[29] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*, page 349, Cancun, Mexico, 2000. IEEE Computer Society.

[30] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the Benefits of Mixed Data and Task Parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 74–83, Santa Barbara, California, USA, July 1995.

[31] S. K. Chan, V. Bharadway, and D. Ghose. Large Matrix-Vector Products on Distributed bus Networks With Communication Delays Using the Divisible Load Paradigm : Performance Analysis and Simulaton. *Mathematics and Computers in Simulation*, 58(1) :71–92, 2001.

[32] H. Chen and M. Maheswaran. Distributed Dynamic Scheduling of Composite Tasks on Grid Systems. In *Proceedings of the 12th Heterogeneous Computing Workshop (HCW'02)*, Fort Lauderdale, FL, April 2002.

[33] P. Chrétienne. Task Scheduling Over Distributed Memory Machines. In *Proceedings of Parallel and Distributed Algorithms*, pages 165–176, North Holland, 1988.

[34] Y.-C. Chung and S. Ranka. Applications and Performance Analysis of a Compile-time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing(Supercomputing'92)*, pages 512–521, Minneapolis, MN, USA, 1992. IEEE Computer Society Press.

[35] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[36] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures*. International Thomson Computer Press, 1995.

[37] CPLEX. `http://www.ilog.com/products/cplex/`.

[38] F. Dahlgren and J. Torrellas. Cache-Only Memory Architectures. *IEEE Computer*, 32(6) :72–79, 1999.

[39] DAS-3. `http://www.cs.vu.nl/das3`.

[40] DIET. `http://graal.ens-lyon.fr/~diet`.

[41] A. B. Downey. A Model For Speedup of Parallel Programs. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1997.

[42] N. Drosinos and N. Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, USA, April 2004. EEE Computer Society.

[43] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4) :473–487, 1989.

[44] P.-F. Dutot. Hierarchical Scheduling for Moldable Tasks. In *Proceedings of the 11th International Euro-Par Conference (Euro-Par 2005)*, pages 302–311, Lisbon, Portugal, September 2005.

[45] P.-F. Dutot and D. Trystram. Scheduling on Hierarchical Clusters Using Malleable Tasks. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2001)*, pages 199–208, Heraklion, Crete Island, Greece, July 2001. ACM Press.

[46] R. A. Dutton and W. Mao. Online Scheduling of Malleable Parallel Jobs. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, Cambridge, Massachusetts, November 2007. ACTA Press.

[47] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2) :138–153, 1990.

[48] D. G. Feitelson and M. A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP 1997)*, pages 238–261, Geneva, Switerland, April 1997. Springer-Verlag.

[49] D. G. Feitelson and L. Rudolph. Towards Convergence in Job Schedulers for Parallel Supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 1996)*, pages 1–26, Honolulu, Hawaii, April 1996. Springer-Verlag.

[50] L. Finta, Z. Liu, I. Milis, and E. Bampis. Scheduling UET-UCT Series-Parallel Graphs on Two Processors. *Theoretical Computer Science*, 162(2) :323–340, 1996.

[51] I. Foster and C. Kesselman. *The Grid 2 : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[52] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, January 1979.

[53] GNU Linear Programming Kit (GLPK). `http://www.gnu.org/software/glpk/`.

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

[55] Grid'5000. `https://www.grid5000.fr`.

[56] R. L. Henderson. Job Scheduling Under the Portable Batch System. In *Processing of the Job Scheduling Strategies for Parallel Processing(JSSPP 1995)*, pages 279–294, Santa Barbara, CA, USA, April 1995.

[57] E. S. H. Hou, N. Ansari, and H. Ren. A Genetic Algorithm for Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 05(2) :113–120, 1994.

[58] C.-C. Hui and S. T. Chanson. Allocating Task Interaction Graphs to Processors in Heterogeneous Networks. *IEEE Transactions on Parallel and Distributed Systems*, 08(9) :908–925, 1997.

[59] S. Hunold, T. Rauber, and G. Rünger. Dynamic Scheduling of Multi-Processor Tasks on Clusters of Clusters. In *Proceedings of the Sixth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (Heteropar'07)*, Austin, TX, USA, November 2007. IEEE Computer Society Press.

[60] M. A. Iverson and F. Özgüner. Hierarchical, Competitive Scheduling of Multiple DAGs in a Dynamic Heterogeneous Environment. *Distributed System Engeneering*, 6(3) :112–, 1999.

[61] D. B. Jackson, Q. Snell, and M. J. Clement. Core Algorithms of the Maui Scheduler. In *Proceedings of the 7th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2001)*, pages 87–102, Cambridge, MA, USA, June 2001.

[62] K. Jansen and L. Porkolab. Linear-Time Approximation Schemes for Scheduling Malleable Parallel Tasks. *Algorithmica*, 32(3) :507–520, 2002.

[63] K. Jansen and H. Zhang. An Approximation Algorithm for Scheduling Malleable Tasks Under General Precedence Constraints. *ACM Transactions on Algorithms*, 2(3) :416–434, 2006.

[64] M. D. Jones, R. Yao, and C. P. Bhole. Hybrid MPI-OpenMP Programming for Parallel OSEM PET Reconstruction. *IEEE Transactions on Nuclear Science*, 53(5) :2752–2758, 2006.

[65] Y.-S. Kee, H. Casanova, and A. Chien. Realistic Modeling and Synthesis of Resources for Computational Grids. In *Proceedings of the Conference on High Performance Networking and Computing (SC'04)*, Pittsburgh, PA, USA, November 2004.

[66] C. Kenyon and E. Rémila. Approximate Strip Packing. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*, pages 31–36, Burlington, Vermont, USA, 1996. IEEE Computer Society.

[67] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, 05(1) :23–32, 1988.

[68] Y.-K. Kwok and I. Ahmad. Dynamic Critical-Path Scheduling : An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5) :506–521, 1996.

[69] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(3) :259–271, 1990.

[70] R. Lepère, G. Mounié, and D. Trystram. An approximation algorithm for scheduling trees of malleable tasks. *European Journal of Operational Research*, 142(2) :242–249, 2002.

[71] R. Lepère, D. Trystram, and G. J. Woeginger. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. *International Journal on Foundations of Computer Science*, 13(4) :613–627, 2002.

[72] D. A. Lifka. The ANL/IBM SP Scheduling System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'95)*, pages 295–303, Santa Barbara, CA, USA, April 1995. Springer-Verlag.

[73] W. Ludwig and P. Tiwari. Scheduling Malleable and Nonmalleable Tasks. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 167–176, Arlington, Virginia, January 1994.

[74] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *Proceedings of the Eighth Heterogeneous Computing Workshop (HCW'99)*, pages 30–, San Juan, Puerto Rico, April 1999. IEEE Computer Society.

[75] L. Marchal, Y. Yang, H. Casanova, and Y. Robert. A Realistic Network/Application Model for Scheduling Divisible Loads on Large-Scale Platforms. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 48.2, Denver, CA, USA, April 2005. IEEE Computer Society.

[76] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 11(2) :146–178, 1993.

[77] A. W. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6) :529–543, 2001.

[78] NetSolve/GridSolve. `http://icl.cs.utk.edu/netsolve`.

[79] OAR. Resource Management System for High Performance Computing. `http://oar.imag.fr`.

[80] The Open Grid Forum (OGF). `http://www.ogf.org`.

[81] M. A. Palis, J.-C. Liou, and D. S. L. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Transactions on Parallel Distributed Systems*, 7(1) :46–55, 1996.

[82] G.-L. Park, B. Shirazi, and J. Marquis. DFRN : A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems. In *Proceedings of the 11th International Symposium on Parallel Processing(IPPS'97)*, pages 157–166, Geneva, Switzerland, April 1997. IEEE Computer Society.

[83] G. N. S. Prasanna and B. R. Musicus. The Optimal Control Approach to Generalized Multiprocessor Scheduling. *Algorithmica*, 15(1) :17–49, 1996.

[84] Ninf. The Ninf Project. `http://ninf.apgrid.org`.

[85] A. Radulescu, C. Nicolescu, A. J. C. van Gemund, and P. P. Jonker. CPR : Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, pages 39–, San Francisco, CA, USA, April 2001. IEEE Computer Society.

[86] A. Radulescu and A. van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *Proceedings of the 15th International Conference on Parallel Processing (ICPP 2001)*, Valencia, Spain, September 2001.

[87] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11) :1098–1116, 1997.

[88] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45 :483–503, 1998.

[89] C. Roig, A. Ripoll, and F. Guirado. A New Task Graph Model for Mapping Message Passing Applications. *IEEE Transactions on Parallel and Distributed Systems*, 18(12) :1740–

1753, December 2007.

[90] G. Sabin, G. Kochhar, and P. Sadayappan. Job Fairness in Non-Preemptive Job Scheduling. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 186–194, Washington, DC, USA, 2004. IEEE Computer Society.

[91] U. Schwiegelshohn and R. Yahyapour. Fairness in Parallel job Jcheduling. *Journal of Scheduling*, 3(5) :297–320, 2000.

[92] P. Shroff, D. W. Watson, N. Flann, and R. Freund. Genetic Simulated Annealing for Scheduling Datadependent Tasks in Heterogeneous Environments. In *Proceedings of the 5th IEEE Heterogeneous Computing Workshop (HCW'96)*, pages 98–117, Honolulu, Hawaii, April 1996.

[93] H. J. Siegel, V. P. Roychowdhury, and L. Wang. A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Computing Environment. In *Proceedings of the 5th IEEE Heterogeneous Computing Workshop (HCW'96)*, pages 72–85, Honolulu, Hawaii, April 1996.

[94] G. C. Sih and E. A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 04(2) :175–187, 1993.

[95] SimGrid. `http://simgrid.gforge.inria.fr`.

[96] H. Singh and A. Youssef. Mapping and Scheduling Heterogeneous Task Graphs Using Genetic Algorithms. In *Proceedings of the 5th IEEE Heterogeneous Computing Workshop (HCW'96)*, pages 86–97, Honolulu, Hawaii, April 1996.

[97] TOP500 : TOP500 Supercomputing Sites. `http://www.top500.org`.

[98] M. Skutella. Approximation Algorithms for the Discrete Time-Cost Tradeoff Problem. *Mathematics of Operations Research*, 23(4) :909–929, 1998.

[99] W. Smith, V. E. Taylor, and I. T. Foster. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In *Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP'99)*, pages 202–219, San Juan, Puerto Rico, April 1999. Springer-Verlag.

[100] P. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'98)*, pages 231–256, Orlando, Florida, USA, March 1998. Springer-Verlag.

[101] A. C. Sodan. Loosely Coordinated Coscheduling in the Context of Other Approaches for Dynamic job Scheduling : a Survey. *Concurrency and Computation : Practice and Experience*, 17(15) :1725–1781, 2005.

[102] F. Sourd. Scheduling Tasks on Unrelated Machines : Large Neighborhood Improvement Procedures. *Journal of Heuristics*, 7(6) :519–531, 2001.

[103] A. Steinberg. A Strip-Packing Algorithm with Absolute Performance Bound 2. *SIAM Journal on Computing*, 26(2) :401–409, 1997.

[104] V. Strassen. Gaussian Elimination Is Not Optimal. *Numerische Mathematik*, 14(3) :354–356, 1969.

[105] F. Suter. *Parallélisme mixte et prédiction de performances sur réseaux hétérogènes de machines parallèles*. PhD thesis, École Normale Supérieure de Lyon, November 2002.

[106] F. Suter. Scheduling Δ-Critical Tasks in Mixed-Parallel Applications on a National Grid. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (Grid 2007)*, Austin, TX, USA, September 2007.

[107] F. Suter and H. Casanova. Extracting Synthetic Multi-Cluster Platform Configurations from Grid'5000 for Driving Simulation Experiments. Technical report, Institut National

de Recherche en Informatique et en Automatique (INRIA), August 2007.

[108] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G : A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1) :41–51, 2003.

[109] The BlueGene/L Team. An overview of the BlueGene/L Supercomputer. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (Supercomputing'02)*, pages 1–22, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[110] G. Tessier, J. Roman, and G. Latu. Hybrid MPI-Thread Implementation on a Cluster of SMP Nodes of a Parallel Simulator for the Propagation of Powdery Mildew in a Vineyard. In *Proceedings of the Second International Conference on High Performance Computing and Communications (HPCC 2006)*, pages 833–842, Munich, Germany, September 2006.

[111] H. Topcuouglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3) :260–274, 2002.

[112] J. Turek, J. Wolf, and P. Yu. Approximate Algorithms for Scheduling Parallelizable Tasks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'92)*, pages 323–332, San Diego, CA, USA, July 1992.

[113] K. van der Raadt, Y. Yang, and H. Casanova. Practical Divisible Load Scheduling on Grid Platforms with APST-DV. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 29.2, Denver, CA, USA, April 2005. IEEE Computer Society.

[114] M. Vanhoucke and D. Debels. The Discrete Time/Cost Trade-off Problem : Extensions and Heuristic Procedures. *Journal of Scheduling*, 10(4-5), 2007.

[115] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications. In *Proceedings of the International Conference on Parallel Processing (ICPP'06)*, Columbus, Ohio, USA, August 2006.

[116] M.Y. Wu and D. D. Gajski. Hypertool : A Programming Aid for Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 01(3) :330–343, 1990.

[117] T. Yang and A. Gerasoulis. DSC : Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9) :951–967, 1994.

[118] Y. Yang and H. Casanova. RUMR : Robust Scheduling for Divisible Workloads. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, pages 114–125, Seattle, WA, USA, June 2003. IEEE Computer Society.

[119] Y. Yang and H. Casanova. UMR : A Multi-Round Algorithm for Scheduling Divisible Workloads. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03)*, pages 24–32, Nice, France, April 2003. IEEE Computer Society.

[120] H. Zhao and R. Sakellariou. Scheduling Multiple DAGs onto Heterogeneous Systems. In *Proceedings of the 15th Heterogeneous Computing Workshop (HCW'06)*, Island of Rhodes, Greece, April 2006.

Glossaire

T_A : Aire moyenne (occupée dans le diagramme de Gantt représentant l'ordonnancement).
T'_A : Aire moyenne modifiée.
T_{CP} : Longueur du chemin critique.
***Bottom-level* d'une tâche** : Longueur du chemin le plus long depuis la tâche considérée jusqu'à la tâche de sortie (qui représente la fin d'exécution du DAG) lorsque l'on fait la somme des temps d'exécution des tâches présentes sur ce chemin, incluant le temps d'exécution de la tâche considérée.

Aire moyenne : Temps moyen d'utilisation des processeurs.

Calcul parallèle : Exécution d'un traitement pouvant être partitionné en tâches élémentaires adaptées afin de pouvoir être réparti entre plusieurs processeurs opérant simultanément.
Chemin critique : Chemin le plus long du DAG en termes de temps d'exécution des tâches présentes sur ce chemin.

DAG : Graphe acyclic orienté (*directed acyclic graph*). Dans la théorie des graphes, un DAG identifie un graphe qui ne possède pas de cycle, et dont les arcs sont orientés. Les DAGs permettent de modéliser des applications parallèles dans le domaine de l'ordonnancement. Les nœuds représentent les tâches de l'application et chaque arc définit une relation de précédence (dépendance de flots ou de données) entre deux tâches.

Grappe de calcul : Ensemble d'ordinateurs (appelés noeuds) reliés entre eux au sein d'un réseau local (typiquement à l'intérieur d'une même salle machine).
Grappe homogène : Grappe de calcul dont les processeurs ont les mêmes caractéristiques (vitesse de calcul, mémoire, etc.).
Grid'5000 : Grille expérimentale répartie sur le territoire français et destinée aux expérimentations des chercheurs en informatique.
Grille de calcul : Agrégation hétérogène de ressources informatiques reparties sur des sites géographiquement distants, partagées par de nombreux utilisateurs et administrées de manière décentralisée en vue d'effectuer du calcul haute performance.

Heuristique : Algorithme qui fournit rapidement (en temps polynomial) une solution réalisable, pas nécessairement optimale, pour un problème d'optimisation NP-difficile.

Longueur du chemin critique : Somme des temps d'exécution des tâches présentes sur le chemin critique.

Makespan : Temps d'exécution total d'une application (temps de complétion).

Ordonnancement : L'ordonnancement d'une ou plusieurs applications parallèles consiste à déterminer l'ordre dans lequel les différentes parties des applications considérées doivent s'exécuter et les ressources que chacune des parties doit utiliser.

Tâche modelable : Tâche parallèle susceptible de s'exécuter sur différents nombres de processeurs, ce nombre de processeurs ne pouvant pas être modifié durant l'exécution de cette tâche.

Tâche parallèle : Tâche telle que les données à traiter peuvent être réparties sur plusieurs processeurs afin de s'exécuter concurremment.

Tâche prête : Tâche dont tous les prédécesseurs ont terminé leur exécution.

Tâche séquentielle : Tâche qui ne s'exécute que sur un seul processeur.

Résumé

Aujourd'hui, les plates-formes hétérogènes et partagées que sont les grilles de calcul sont omniprésentes. De plus, le besoin d'exécuter des applications parallèles complexes est croissant. Cette thèse vise à ordonnancer des applications représentées par des graphes de tâches modelables (dont le nombre de processeurs est fixé par l'ordonnanceur) sur des grilles de calcul en exploitant le maximum de parallélisme, utilisant efficacement les ressources, gérant l'hétérogénéité et le partage des ressources. Nous avons pour cela opté pour des heuristiques pragmatiques car, bien qu'elles n'offrent pas de garantie de performance, elles peuvent néanmoins conduire à de bonnes performances moyennes tout en construisant des ordonnancements en des temps relativement courts. La plupart des heuristiques existantes n'ordonnancent les applications parallèles mixtes qu'en milieu homogène et utilisent parfois inefficacement les ressources. Nous avons donc tout d'abord étudié différentes heuristiques dans le cas de plates-formes homogènes et proposé des améliorations visant à améliorer le compromis entre réduction du temps de complétion et efficacité. Nous avons ensuite introduit la gestion de l'hétérogénéité dans l'heuristique proposée et comparé ses performances à celles d'un algorithme garanti. Enfin, nous avons tenu compte du caractère partagé des grilles en gérant la concurrence entre applications. L'approche retenue consiste à limiter la quantité de ressources que chaque application peut utiliser pour construire son ordonnancement. Nous avons également proposé plusieurs stratégies de détermination de cette contrainte de ressources.

Mots-clés: ordonnancement, ordonnancement multi applications, applications parallèles mixtes, DAGs, plates-formes multi-grappes.

Abstract

Today, computing grids, that are shared and heterogeneous platforms, are ubiquitous. Furthermore, the need to execute complex parallel applications is growing. The aim of this thesis is to schedule applications modeled by moldable task graphs (the number of processors allocated to each task is fixed by the scheduler) onto computing grids to exploit maximum parallelism, use efficiently the resources and manage the sharing of resources. We chose to design pragmatic heuristics because, although they do not guarantee performance, they can lead to good average performance while computing schedules in relatively short times. Almost all existing heuristics only schedule mixed parallel applications onto homogeneous platforms and they sometimes use inefficiently the resources. Thus, we first studied different scheduling heuristics in the case of homogeneous platforms and propose improvements to have a good compromise between the completion time and the efficiency. We then introduced management of heterogeneity in the proposed heuristic and compared its performances with those of a guaranteed algorithm. Finally, we have taken into account the fact that grids are shared by managing competition between applications. Our approach is to limit the amount of resources that each application can use to build its schedule. We also proposed different strategies to determine that resource constraint.

Keywords: scheduling, concurrent scheduling, mixed parallel applications, DAGs, multi-cluster platforms.