Heuristiques d'ordonnancement en deux étapes de graphes de tâches parallèles

Tchimou N'Takpé

Heuristiques d'ordonnancement en deux étapes de graphes de tâches parallèles

Tchimou N'takpé [1]

Nancy Université / LORIA [2]
Campus Scientifique - BP 239
F-54506 Vandoeuvre-lès-Nancy Cedex
Tchimou.Ntakpe@loria.fr

RÉSUMÉ. *L'ordonnancement d'applications parallèles représentées par des graphes de tâches consiste à trouver l'ensemble de processeurs sur lesquels chaque tâche doit être exécutée afin de minimiser le temps d'exécution de ces applications tout en exploitant rationnellement les ressources. Alors que la plupart des algorithmes d'ordonnancement de graphes de tâches parallèles visent des grappes homogènes, cet article montre la nécessité d'avoir de tels algorithmes pour des agrégations de grappes de calcul qui sont de plus en plus répandues et qui peuvent permettre de déployer des applications parallèles à échelles sans précédents. Nous proposons des améliorations d'une heuristique d'ordonnancement de tâches parallèles en milieu homogène. Ensuite, nous l'adaptons au cas des plates-formes hétérogènes de type grappe hétérogène de grappes homogènes.*

ABSTRACT. *While most parallel task graph scheduling research has been done in the context of single homogeneous clusters, heterogeneous platforms have become prevalent and are extremely attractive for deploying applications at unprecedented scales. In this paper we address the need for scheduling techniques for parallel task applications for heterogeneous clusters of clusters by proposing a method to adapt existing parallel task graph scheduling heuristics that have proved to be efficient on homogeneous environments. Before adapting that heuristic to heterogeneous platforms, we propose some improvements for homogeneous platforms*

MOTS-CLÉS : *Heuristiques d'ordonnancement, tâches parallèles, DAGs, grappe de grappes*

KEYWORDS: *Heterogeneous scheduling, parallel tasks, DAGs, cluster of clusters*

1. Cette étude a été en partie soutenue par l'ARC INRIA OTaPHe, la Région Lorraine et le Gouvernement Ivoirien
2. UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1

e soumission à *Techniques et Sciences Informatiques*, le 17 juillet 2007

1. Introduction

Une approche récente permettant de pallier la demande croissante en mémoire et en ressources de calcul des applications parallèles consiste à agréger des grappes de calcul existantes soit au sein d'une seule institution, soit réparties entre plusieurs institutions. Il s'agit souvent de grappes de tailles variables dont les capacités peuvent être différentes en fonction des technologies présentes au moment de leur installation. Ces plates-formes composées de plusieurs *grappes de stations de travail* (Anderson *et al.*, 1995) sont à la fois attractives parce qu'elles offrent une importante puissance de calcul, et un défit pour les chercheurs du fait de leur hétérogénéité.

Une des méthodes qui permet d'exploiter la puissance de calcul ainsi disponible est de combiner les parallélismes de tâches et de données présents dans les applications scientifiques. Ces applications peuvent alors être modélisées par des graphes de tâches parallèles. De manière informelle, une tâche parallèle est une tâche qui contient des opérations élémentaires, typiquement une routine numérique ou des boucles imbriquées, qui contiennent suffisamment de parallélisme pour être exécutées par plus d'un processeur. Dans cet article, nous considérons un certain type de tâches parallèles : les *tâches modelables* (Turek *et al.*, 1992). Ce sont des tâches parallèles pouvant s'exécuter sur un nombre quelconque de processeurs. Ce nombre n'est pas fixé à priori mais est déterminé avant l'exécution et ne varie pas ultérieurement.

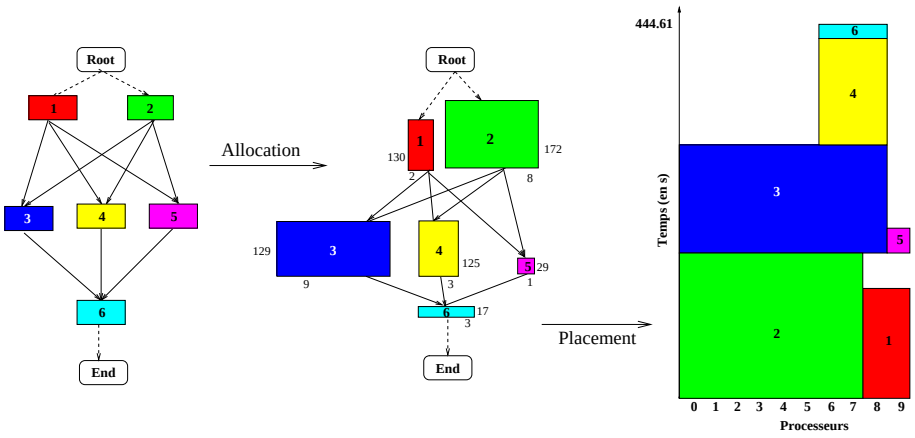

Figure 1. *Exemple d'ordonnancement en deux étapes d'un graphe de tâche synthétique sur une grappe homogène de* 10 *processeurs.*

La figure 1 illustre le problème de l'ordonnancement de tâches parallèles qui consiste dans un premier temps à trouver le bon nombre de processeurs à allouer à chaque tâche lors de la *phase d'allocation*. Ensuite il faut déterminer quels sont les processeurs sur lesquels exécuter les différentes tâches lors de la *phase de placement*. Les algorithmes d'ordonnancement de tâches parallèles peuvent être repartis 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* (Boudet *et al.*, 2003). La seconde catégorie regroupe les algorithmes d'*ordonnancement en deux étapes* (Rădulescu *et al.*, 2001a; Rădulescu *et al.*, 2001b; Ramaswamy *et al.*, 1997; Rauber *et al.*, 1998) où l'allocation et le placement sont séparés en deux procédures distinctes. 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 étés alloués à la tâche 4, ce qui conduit à un temps d'exécution de 125 secondes.

De nombreuses études ont été réalisées pour l'ordonnancement de graphes de tâches parallèles dans le cas de plates-formes homogènes (Lepère *et al.*, 2002; Rădulescu *et al.*, 2001a; Rădulescu *et al.*, 2001b; Ramaswamy *et al.*, 1997; Rauber *et al.*, 1998). Or les plates-formes hétérogènes sont de plus en plus répandues et très attrayantes car elles peuvent permettre de déployer des applications parallèles à échelles sans précédents. Il est donc nécessaire de développer des heuristiques d'ordonnancement pour de tels systèmes hétérogènes. Une première approche a consisté à adapter une heuristique d'ordonnancement de tâches séquentielles sur plates-formes hétérogènes au cas de tâches parallèles (Casanova *et al.*, 2004). Dans cet article nous nous intéressons à une approche complémentaire consistant à modifier un algorithme d'ordonnancement de tâches parallèles en milieu homogène (Rădulescu *et al.*, 2001b) et à prendre en compte l'hétérogénéité des ressources.

Les contributions de cet article sont : (i) apporter des améliorations à l'heuristique choisie dans le cas de plates-formes homogènes, aussi bien dans sa procédure d'allocation que dans sa procédure de placement ; (ii) utiliser un nouveau concept de virtualisation des plates-formes qui permet de gérer plus facilement les allocations de ressources en milieu hétérogène ; (iii) introduire une nouvelle méthode de placement fondée sur l'idée de l'heuristique *sufferage* (Maheswaran *et al.*, 1999) ; (iv) et, définir une méthode d'évaluation par simulation de nombreux scénarios.

Dans la section suivante, nous présentons les modèles de plates-formes et d'applications utilisés. Nous décrirons ensuite notre méthodologie d'évaluation (section 3) puis nous présenterons quelques travaux reliés qui déboucherons sur la formalisation du problème dans la section 4. Après avoir apporté quelques améliorations à CPA, une heuristique d'ordonnancement en milieu homogène issu de la littérature (section 5), nous adapterons l'une des heuristiques obtenues aux plates-formes hétérogènes dans la section 6. Enfin, la section 7 nous permettra de conclure cet article.

2. Modèles de plates-formes et d'applications

Dans cet article, nous considérons 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. Nous avons donc C grappes et chaque grappe C_k contient P_k

processeurs identiques pour un total de P processeurs sur la plate-forme. Les vitesses des processeurs et les caractéristiques des réseaux locaux ne sont pas nécessairement les mêmes entre les différentes grappes. La figure 2 montre la structure de nos plates-formes. Les processeurs d'une même grappe sont reliés entre eux à travers un commutateur (Switch). Les grappes sont reliées par le biais d'une passerelle à un lien réseau très haut débit (dorsale) commun.

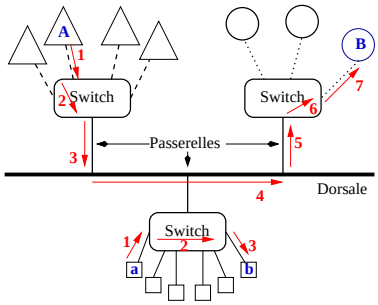

Figure 2. *Modèle de plate-forme avec les routes entre deux processeurs A et B localisés sur deux grappes différentes et entre deux processeurs a et b de la même grappe.*

Une application parallèle peut être modélisée par un graphe acyclique orienté (DAG) $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ où $\mathcal{N} = \{t_i \,/\, i = 1, \cdots, N\}$ est un ensemble de N nœuds (ou tâches) et $\mathcal{E} = \{e_{i,j} \,/\, (i,j) \subset \{1, \cdots, N\} \times \{1, \cdots, N\}\}$ est un ensemble de E arcs. Les nœuds représentent les tâches parallèles et les arcs définissent les relations de précédence (dépendances de flots ou de données) entre les tâches. On associe à chaque arc $e_{i,j}$, la quantité de données que la tâche t_i doit transférer à la tâche t_j. Par définition, une *tâche d'entrée* du graphe n'a aucun prédécesseur et une *tâche de sortie* est sans successeur. Dans cette étude nous utilisons les nœuds **Root** et **End** pour représenter respectivement les tâches d'entrée et de sortie des DAGs. Ce sont deux nœuds fictifs sans coût (de calcul ou de communication) qui facilitent la manipulation des graphes de tâches. Une tâche est dite *prête* lorsque tous ses prédécesseurs ont terminé leur exécution.

Dans une grappe homogène, le temps d'exécution d'une tâche parallèle $t \in \mathcal{N}$ peut être modélisé par un modèle d'accélération classique. Dans cet article, nous utilisons la loi d'Amdahl (Amdahl, 1967) où le temps d'exécution parallèle est donné par :

$$T(t, p(t)) \quad = \quad \left(\alpha + \frac{1-\alpha}{p(t)}\right) \cdot T(t, 1), \qquad [1]$$

$p(t)$ 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.

Ce modèle d'accélération ayant été conçu pour prédire le temps d'exécution d'une application parallèle sur une grappes homogène (en termes de réseaux et des caractéristiques des nœuds de calcul), nous restreignons 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 inter-grappes peuvent être très coûteuses. Nous admettons également dans cet article que les processeurs des plates-formes utilisées sont uniformes, *i.e.*, le temps d'exécution séquentiel d'une tâche est inversement proportionnel à la vitesse (en GFlops) des processeurs

On définit enfin $T_b(t)$, le *bottom level*, comme étant la longueur du chemin le plus long depuis la tâche t, incluant son propre temps d'exécution, jusqu'à une *tâche de sortie* quelconque. $T_b(t)$ est calculé en faisant la somme des temps d'exécution (prédits) des tâches présentes sur ce chemin.

Le tableau 1 présente les principales notations qui seront utilisées dans cet article.

$T_b(t)$	bottom level de la tâche t
T_{CP}	longueur du chemin critique du DAG
T_A	aire moyenne
$T_s(t)$	date de début d'exécution d'une tâche t
$T_f(t)$	date de fin d'exécution d'une tâche t
C	nombre de grappes
P_{ref}	nombre total de processeurs sur la grappe de référence
P_k	nombre total de processeurs sur la grappe C_k
$p^k(t)$	nombre de processeurs alloués à la tâche t sur la grappe C_k
$T^k(t, p^k(t))$	temps d'exécution de la tâche t sur $p^k(t)$ processeurs de la grappe C_k
r_k	rapport de la puissance d'un processeur de la grappe de référence sur celle d'un processeur de la grappe C_k

Tableau 1. *Liste des principales notations utilisées.*

3. Méthodologie d'évaluation

Pour l'évaluation de nos heuristiques, nous aurons recours 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[1] (Legrand *et al.*, 2003), une boîte à outils conçue pour la simulation de grilles de calcul et/ou la mise en œuvre d'applications distribuées. Les simulations seront effectuées sur des plates-formes et des DAGs générées aléatoirement en faisant varier plusieurs paramètres permettant de fixer certaines propriétés.

Les plates-formes générées sont constituées de 1, 2, 4 ou 8 grappes. Le nombre de processeurs de chaque grappe est tiré aléatoirement entre 16 et 128 pour les plates-formes dont le nombre de grappes est supérieur ou égal à 2. Pour les plates-formes à

1. http ://simgrid.gforge.inria.fr

une seule grappe, le nombre de processeurs est tiré entre 16 et 512. Dans une même plate-forme, les liens intra-grappes sont du Fast Ethernet (débit $= 100Mb/s$ et latence $= 100\mu s$) pour les grappes d'indice pair et du Giga Ethernet (débit $= 1Gb/s$ et latence $= 100\mu s$) pour les grappes d'indice impair. Ces liens peuvent subir des contentions. La dorsale reliant les grappes entre elles a un débit de $2{,}5Gb/s$ et une latence de $50ms$. Chaque grappe est connectée à la dorsale via une passerelle ayant une capacité de $1Gb/s$ avec une latence de $100\mu s$.

La borne inférieure des vitesses des processeurs (en $GFlops$) est de 0,25, 0,5, 0,75 ou 1. En ce qui concerne les plates-formes ayant plusieurs grappes, le rapport entre cette borne inférieure et la borne supérieure des vitesses des processeurs de la plate-forme, qui constitue le degré d'hétérogénéité, est pris parmi les valeurs 1, 2 ou 5. Nous choisissons ainsi la vitesse des processeurs des différentes grappes de la plate-forme en tirant aléatoirement des vitesses entre les bornes inférieure et supérieure. Par exemple si la borne inférieure des vitesses de processeurs est fixée à $0{,}5GFlops$ et le degré d'hétérogénéité à 5, alors les vitesses des processeurs sont tirées entre $0{,}5GFlops$ et $2{,}5GFlops$. Pour chaque combinaison de ces paramètres nous générons 10 échantillons dans le cas des plates-formes à une grappe et 5 échantillons pour les autres. Au total, nous avons généré 220 plates-formes dont 40 à une grappe et 180 (60×3) pour les plates-formes ayant plusieurs grappes.

Nous supposons que les tâches parallèles considérées traitent des nombres réels double précision et que les processeurs ont chacun une mémoire de 1 Go. Les graphes de tâches sont donc générés en tirant la taille des données m, un 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 $a \cdot M$ (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 $a \cdot M^{3/2}$ (multiplication de deux matrices de tailles $\sqrt{M} \times \sqrt{M}$), ou elle est tirée au sort parmi les trois valeurs précédentes. On a $M = m^2$ et a est un nombre tiré aléatoirement entre 2^6 et 2^9. La portion non parallélisable de chaque tâche (le α de la loi d'Amdahl) est un nombre aléatoire pris entre 0 et 0,25. Le coût des transferts est égal à M, où M est relatif à la tâche qui génère le transfert.

Quatre paramètres permettent de faire varier la forme des graphes : (i) la largeur (0,1, 0,2 ou 0,8). Une largeur de 0,8 correspond à un graphe compact avec beaucoup de parallélisme de tâches ; (ii) la régularité entre niveaux (0,2 ou 0,8). Dans un graphe peu régulier, la différence entre les nombres de tâches sur deux niveaux consécutifs peut être très importante ; (iii) la densité qui caractérise le fait que l'on a plus ou moins de relations de précédence entre les tâches de l'application (0,2 ou 0,8) ; et (iv) la longueur maximale des sauts entre niveaux (1, 2 ou 4). Ce dernier paramètre sert à générer des graphes contenant des chemins de longueur différentes (en nombre de nœuds) entre les tâches d'entrée et de sortie. Pour chaque combinaison de ces paramètres, nous générons 3 échantillons différents, soit 1296 DAGs.

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 de nos heuristiques afin d'obtenir des mesures moyennes pertinentes. Nous utiliserons donc les trois métriques suivantes :

Le temps de complétion normalisé. L'une des mesures les plus utilisées dans l'évaluation des heuristiques d'ordonnancement est le temps de complétion ou *makespan*. Le *makespan* est la différence entre la date de début et la date de fin d'exécution d'une application. Autrement dit, c'est le temps d'exécution total de l'application. Nous normalisons cette mesure en la divisant par un *makespan de référence*. Pour obtenir le *makespan* de référence, on extrait le chemin critique séquentiel du DAG, *i.e.*, le chemin le plus long lorsqu'un même processeur est alloué aux tâches. Ensuite, le DAG en forme de chaîne obtenu est exécuté par une heuristique de liste qui lui procure le *makespan* le plus petit possible sur la plate-forme considérée : le *makespan* de référence. Pour cela, nous utilisons l'heuristique gloutonne M-HEFT (Casanova *et al.*, 2004) qui prend chaque tâche prête et l'exécute sur la configuration de processeurs qui lui garantit la plus petite date de fin d'exécution. Le temps de complétion normalisé que nous venons de définir s'apparente au *SLR* (*Schedule Length Ratio*) utilisé dans (Topcuoglu *et al.*, 2002) où le *makespan* de référence est une borne inférieure des *makespan* possibles. Plus une heuristique est performante, plus le temps de complétion normalisé obtenu est faible.

L'accélération par rapport au meilleur algorithme séquentiel. Elle mesure le gain en temps d'exécution par rapport à un algorithme d'ordonnancement séquentiel (SEQ) qui ordonnance les unes après les autres, les tâches sur le processeur le plus rapide de la plate-forme considérée.

L'efficacité des algorithmes. Sur une grappe homogène, l'efficacité se définit comme étant le rapport de l'accélération par rapport à SEQ sur le nombre moyen de processeurs utilisés par l'application. Nous généralisons cette définition dans l'équation 2 : pour un DAG et une plate-forme donnés, l'efficacité peut se calculer en faisant le rapport du travail total effectué (par les ressources de calcul) en utilisant SEQ par le travail total effectué en utilisant l'algorithme à évaluer. Une efficacité élevée (proche de 1) traduit le fait que les ressources de calcul sont rationnellement utilisées dans l'exécution parallèle de l'application. Le travail total W effectué durant l'exécution d'une application est la somme des travaux effectués pour l'exécution de chaque tâche, *i.e.*, le produit du temps d'exécution de cette tâche (en s) par la puissance de calcul utilisée (en $GFlops$).

$$E_{ALGO} = \frac{W_{SEQ}}{W_{ALGO}} \quad [2]$$

4. Travaux précédents

Le problème de l'ordonnancement de tâches en prenant en compte le coût des communications, est NP-complet au sens fort, même dans le cas ou il existe un nombre infini de processeurs (Chrétienne, 1988). De nombreuses heuristiques ont

donc été conçues pour ordonnancer des tâches parallèles. Dans (Dutot, 2005), un algorithme d'ordonnancement de tâches modelables interdépendantes sur une hiérarchie de machines multiprocesseurs est présenté. Dans (Casanova *et al.*, 2004), les auteurs proposent l'une des rares heuristiques d'ordonnancement de tâches parallèles destinées aux plates-formes hétérogènes. Ils adaptent un algorithme d'ordonnancement de tâches séquentielles sur plates-formes hétérogènes au cas de DAGs de tâches parallèles. Notre approche complète cette dernière puisque nous partons d'un algorithme d'ordonnancement de graphes de tâches parallèles sur plates-formes homogènes pour l'adapter au cas de plates-formes hétérogènes.

Si certaines études théoriques existent (Lepère *et al.*, 2002), la plupart des algorithmes d'ordonnancement de tâches parallèles (Rădulescu *et al.*, 2001a; Rădulescu *et al.*, 2001b; Ramaswamy *et al.*, 1997; Rauber *et al.*, 1998) sont des algorithmes en deux étapes et ont été conçus pour des milieux homogènes. La première étape vise à déterminer un nombre de processeurs adéquat pour chaque tâche. Dans la seconde étape, les différents auteurs utilisent des heuristiques de liste pour ordonnancer les tâches.

Dans (Ramaswamy *et al.*, 1997), les auteurs extraient des DAGs à partir de codes séquentiels puis appliquent leur algorithme TSAS (*Two Step Allocation and Scheduling*). TSAS utilise la programmation convexe, rendue possible grâce à la propriété de *posynomialité* des modèles de coût choisis, ainsi que certaines propriétés de leur structure de DAGs. Les fonctions posynomiales sont semblables au fonctions polynomiales mais elles ont des coefficients positifs et les exposants sont des nombres réels. Une telle fonction peut être transformée en une fonction convexe par un simple changement de variable. Ainsi, la programmation convexe leur permet d'obtenir en temps polynomial les allocations dans l'espace des réels qu'ils arrondissent ensuite à des nombres entiers. Les auteurs de (Rauber *et al.*, 1998) limitent quant à eux leur étude à des graphes construits par compositions séries et/ou parallèles. 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. Dans le second cas, l'ensemble des processeurs est divisé en un nombre optimal de sous-ensembles, déterminé par un algorithme glouton. Le critère d'optimisation de cet algorithme est la minimisation du temps de complétion de l'ensemble des tâches.

Dans cet article, nous nous intéressons à l'algorithme CPA (Rădulescu *et al.*, 2001b) qui est à la fois peu coûteux en termes de complexité et relativement performant aussi bien pour les temps de complétion des applications que pour une meilleure gestion des ressources. CPA (*Critical Path and Area-based Scheduling*) vise à obtenir le meilleur compromis entre la longueur du chemin critique, *i.e.*, le plus long chemin du graphe en tenant uniquement compte des temps d'exécution des tâches, et l'aire moyenne du diagramme de Gantt. Cette étape ne tient pas compte des communications car les processeurs sur lesquels les tâches seront exécutées ne sont pas connus d'avance. Ils ne seront connus qu'au moment du placement de chaque tâche. Rădulescu *et al.* remarquent que le temps d'exécution d'une application parallèle peut

être approché par sa borne inférieure $T_p^e = \max\{T_{CP}, T_A\}$, où T_{CP} est la longueur du chemin critique et T_A, l'aire moyenne :

$$T_{CP} = \max_{t \in \mathcal{N}} T_b(t), \text{ et} \quad [3]$$

$$T_A = \frac{1}{P} \sum_{i=0}^{N} \left(T\left(t_i, p(t_i)\right) \times p(t_i) \right). \quad [4]$$

Notons que la notion d'aire moyenne définie par les auteurs de CPA a la dimension d'un temps. Elle peut être interprétée comme étant le temps moyen d'utilisation des processeurs.

Le but de CPA est de minimiser T_p^e au terme de la phase d'allocation. Sachant que T_{CP} diminue tandis 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, 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 rapport $T(t, p(t))/p(t)$ diminue le plus significativement si un processeur supplémentaire lui est attribué. 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 figure 3 présente l'évolution de T_{CP} et de T_A en fonction du nombre d'itérations de la procédure d'allocation de CPA appliquée à l'exemple de la figure 1.

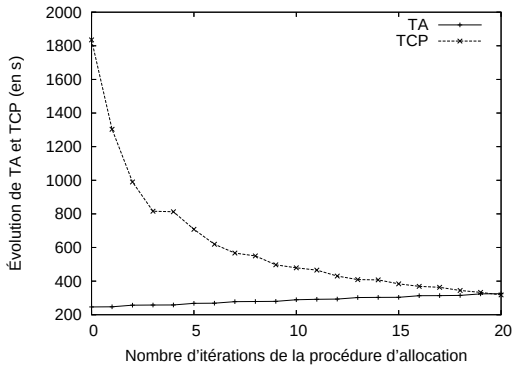

Figure 3. *Exemple d'évolution de T_{CP} et T_A lors de la procédure d'allocation.*

La tâche prête ayant le plus grand *bottom level* est choisie pour être placée à chaque itération de la procédure de placement. Lorsqu'une tâche est prête (tous ses prédécesseurs ont terminé leur exécution), l'emplacement des données nécessaires à son exécution est connu. On peut maintenant tenir compte des temps de redistributions de données et trouver les processeurs qui minimisent la date de fin d'exécution $T_f(t)$ de chaque tâche prête t. $T_f(t)$ dépend entre autres de la date d'arrivée de la dernière donnée nécessaire à l'exécution de t et de la date de disponibilité des processeurs choisis.

Dans (Rădulescu *et al.*, 2001b) il a été montré que la complexité de CPA est de l'ordre de $O\left(N(N+E)P\right)$.

5. Améliorations de CPA en milieu homogène

Dans cette section, nous nous plaçons dans le cas où les plates-formes sont constituées d'une seule grappe homogène et nous proposons deux améliorations de l'algorithme original : la première porte sur la phase d'allocation et la seconde sur la phase de placement.

5.1. *Nouveau critère d'arrêt dans la procédure d'allocation*

Par expérience, nous avons constaté que le calcul de l'aire moyenne de CPA n'était plus pertinent lorsque le nombre de processeurs (P) de la plate-forme est beaucoup plus grand que le nombre de tâches (N). En effet, T_A converge très lentement vers T_{CP} lorsque le nombre de processeurs de la plate-forme est très grand. Cela débouche sur des allocations contenant un très grand nombre de processeurs. Or, plus on alloue de processeurs aux tâches, plus le risque de ne plus pouvoir exécuter en parallèle certaines tâches concurrentes augmente. Il peut donc s'avérer préférable d'arrêter le processus d'allocation plus tôt afin de profiter au mieux du parallélisme de tâches. Nous proposons pour cela le compromis suivant qui permet d'arrêter plus vite la procédure d'allocation dans le cas où le nombre de ressources est très élevé en prenant $\min(P, \sqrt{P \times N})$ au lieu de P. Cela nous conduit à redéfinir la notion d'aire moyenne de la façon suivante :

$$T'_A = \frac{1}{\min(P, \sqrt{P \times N})} \sum_{i=0}^{N} \left(T\left(t_i, p(t_i)\right) \times p(t_i)\right). \quad [5]$$

Pour $P > N$, 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.

La figure 4 montre l'évolution de T_{CP} et de T_A en utilisant CPA ainsi que celle de T'_A dans la nouvelle procédure d'allocation sur le DAG de la figure 1. Dans cet exemple, le nombre de processeurs ($P = 30$) est supérieur au nombre de tâches ($N = 6$). D'où une convergence plus rapide de T_{CP} et T'_A dans le cas de la nouvelle procédure d'allocation. 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 ($74 = 6 + 68$). 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 5 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

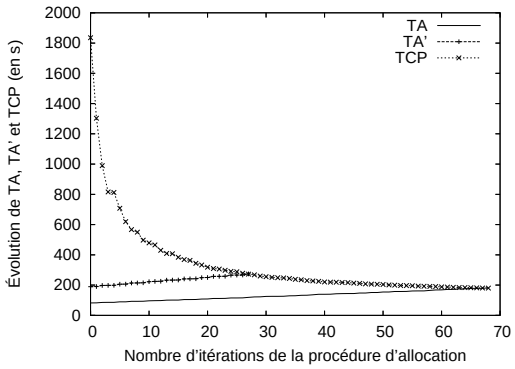

Figure 4. *Exemple d'évolution de T_A, T'_A et de T_{CP} dans la procédure d'allocation de CPA et la nouvelle procédure d'allocation.*

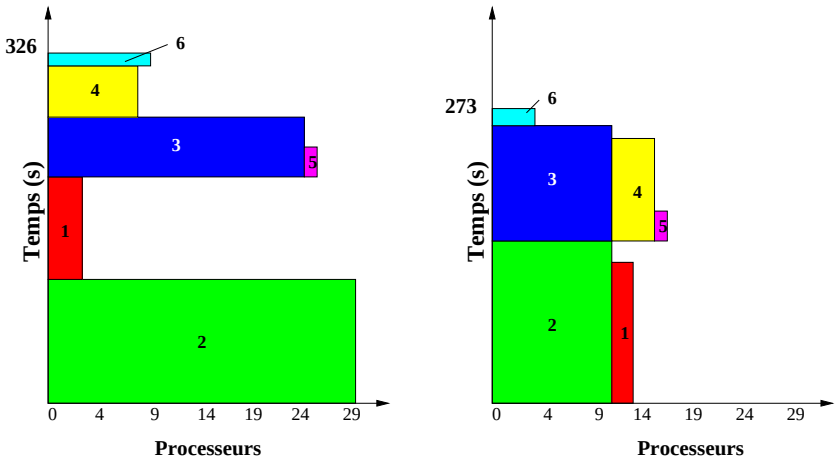

Figure 5. *Ordonnancement du DAG de la figure 1 avec CPA (à gauche) et en utilisant la nouvelle allocation (à droite) sur une grappe homogène de* 30 *processeurs.*

permet d'obtenir un meilleur temps de complétion par rapport à l'ordonnancement de CPA en plus d'une moindre consommation de ressources. Nous sommes conscients que le compromis que nous venons de définir n'est pas toujours optimal du point de vue du temps de complétion des applications mais il permet surtout de mieux gérer l'utilisation des ressources. En effet, hormis les tâches « entièrement parallélisables », plus l'on alloue de processeurs aux tâches, plus l'efficacité diminue. Nous pouvons également remarquer dans la figure 6 qu'appliquer cette modification à l'exemple de la figure 1 (où $N = 6$ et $P = 10$) permet également de réduire le temps de complétion par rapport à CPA.

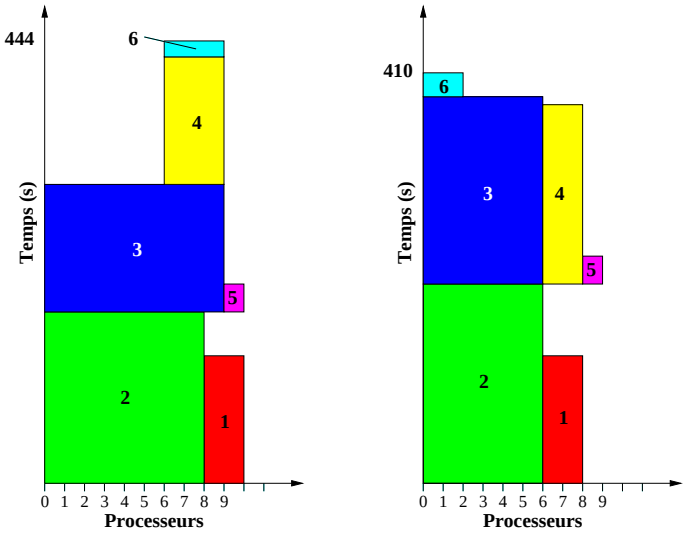

Figure 6. *Ordonnancement du DAG de la figure 1 avec CPA (à gauche) et en utilisant la nouvelle allocation (à droite) sur une grappe homogène de* 10 *processeurs.*

5.2. *Tassage lors du placement*

La phase d'allocation et la phase de placement des tâches étant découplées, 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 à la date où elle est prête.

Le *tassage* permet d'éviter ce genre de situation. Il permet dans certains cas d'améliorer la date de fin d'exécution d'une tâche prête tout en réduisant le nombre de processeurs qui lui sont alloués. La procédure de placement avec tassage se comporte de la manière suivante :

1) À l'instant où la tâche prête la plus prioritaire est choisie, on regarde s'il est possible de débuter son exécution immédiatement avec son allocation initiale, *i.e.*, le nombre de processeurs disponibles est supérieur ou égal à cette allocation. Dans ce cas, la tâche est placée sur les processeurs disponibles.

2) Si le nombre de processeurs disponibles est inférieur à l'allocation déterminée pour cette tâche, il faut vérifier si elle pourrait terminer son exécution plus tôt en utilisant seulement les processeurs déjà disponibles plutôt que si elle devait attendre que tous les processeurs qui lui sont alloués soient libres. Si c'est le cas, alors une nouvelle allocation lui est attribuée et la tâche est placée sur les processeurs disponibles. Sinon elle attend qu'il y ait suffisamment de processeurs libres en vue d'être placée selon son allocation initiale.

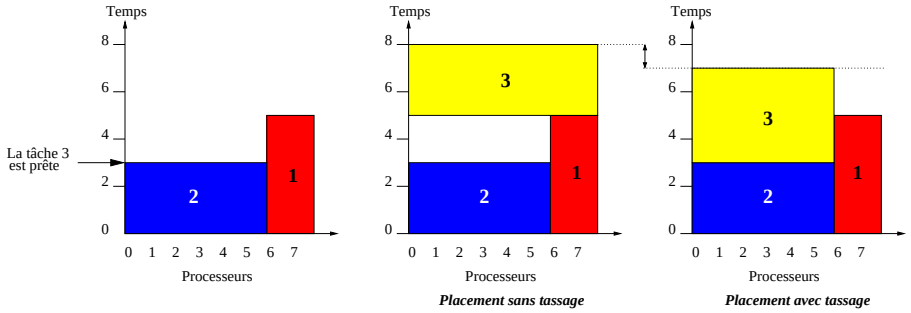

Figure 7. *Illustration du placement avec tassage.*

La figure 7 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'avoir une meilleure date de fin d'exécution pour cette tâche.

Dans la section suivante, nous allons comparer les performances obtenues en appliquant chacune des modifications proposées avec l'algorithme original CPA (Rădulescu *et al.*, 2001b) en milieu homogène. Nous regarderons également comment l'algorithme se comporte lorsque nous combinons les deux modifications que nous venons de décrire.

5.3. *Évaluation des algorithmes*

La figure 8 compare les moyennes du temps de complétion normalisé et de l'efficacité des algorithmes sur l'ensemble des DAGs et pour les plates-formes à une grappe dans les cas suivants :

– L'algorithme original (CPA).
– Utilisation du tassage pendant le placement.
– Utilisation de T'_A au lieu de T_A.
– Utilisation combinée de T'_A et du tassage.

Nous observons qu'en moyenne, le tassage permet de réduire le temps de complétion des applications et de gagner légèrement en consommation de ressources par rapport à CPA. En effet, cette amélioration de l'heuristique de placement permet de réduire les trous dans l'ordonnancement en diminuant légèrement l'allocation de certaines tâches. En ce qui concerne l'utilisation de T'_A, non seulement elle permet de gagner de manière significative sur le temps de complétion des applications, mais aussi elle utilise plus efficacement les ressources par rapport à CPA. Cela confirme le fait qu'il est important d'arrêter les allocations suffisamment tôt afin de pouvoir exécuter les tâches prêtes en parallèle. La combinaison des deux améliorations (T'_A et

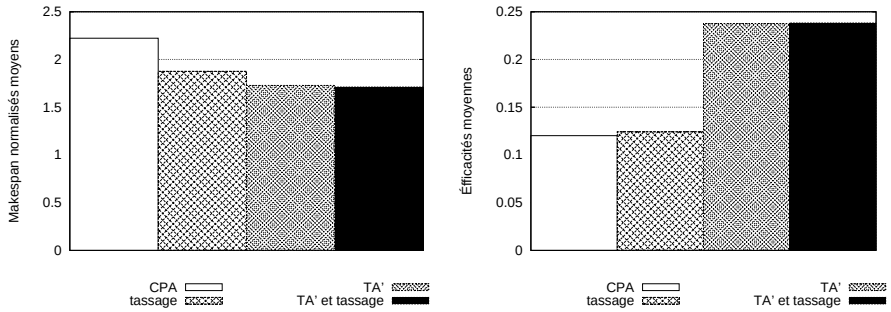

Figure 8. *Performances globales sur une grappe.*

tassage) améliore légèrement les performances par rapport à l'utilisation de T'_A seule, cette dernière ayant vraisemblablement un apport prédominant.

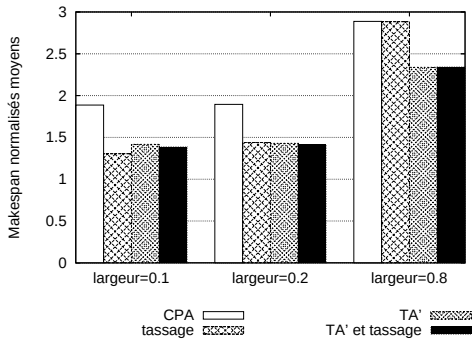

Figure 9. *Makespan moyens en fonction de la largeur des graphes.*

Lorsque nous regardons l'évolution du temps de complétion normalisé des algorithmes en fonction de la largeur des DAGs (figure 9), il apparaît que pour les DAGs peu larges ($largeur = 0{,}1$) le tassage fournit un meilleur temps de complétion par rapport à l'utilisation du nouveau critère d'arrêt. Ceci s'explique par le fait que pour les graphes quasi filiformes, il existe peu de tâches pouvant s'exécuter en parallèle. Par contre dès qu'il existe suffisamment de tâches concurrentes dans les applications ($largeur > 0{,}2$), le nouveau critère d'arrêt de la procédure d'allocation permet de gagner en temps de complétion par rapport au tassage qui améliore de moins en moins l'algorithme original.

Dans la section suivante, nous adaptons l'heuristique obtenue en utilisant T'_A au cas des plates-formes hétérogènes. Nous mettrons en place deux heuristiques qui diffèrent par leur phase de placement.

6. Adaptation aux plates-formes hétérogènes

La plate-forme cible étant maintenant constituée de C grappes, notre idée consiste à attribuer, lors de la première phase, une *allocation de référence* à chaque tâche qui représente ses C allocations potentielles. Nous définirons comment déterminer le nombre de processeurs à allouer à une tâche sur une grappe en fonction de son allocation de référence dans la section 6.1. Lors de son placement nous retiendrons parmi ses allocations potentielles celle qui minimise la date de fin d'exécution d'une tâche prête.

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 :

$$P_{ref} = \left\lceil \sum_{k=0}^{C-1} \frac{P_k}{r_k} \right\rceil \quad [6]$$

où r_k est le rapport de la puissance 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 conserver une faible complexité dans le nouvel algorithme. Notons $p^{ref}(t)$, l'allocation de référence pour la tâche t, *i.e.*, 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 (T'_A) se définissent maintenant par rapport aux allocations de référence :

$$T_{CP} = \max_{t \in \mathcal{N}} T_b^{ref}(t), \quad [7]$$

$$T'_A = \frac{1}{\min\{P_{ref}, \sqrt{P_{ref} \times N}\}} \sum_{i=0}^{N} \left(T^{ref}\left(t_i, p^{ref}(t_i)\right) \times p^{ref}(t_i) \right). \quad [8]$$

où $T_b^{ref}(t)$ est le *bottom level* calculé à partir des allocations de référence des tâches.

Les deux sections qui suivent décrivent les deux étapes des heuristiques que nous avons conçus.

6.1. *Allocation de processeurs*

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.

Algorithme 1 Allocation de processeurs

pour tout $t \in \mathcal{N}$ **faire**
  $p^{ref}(t) \leftarrow 1$;
fin pour
tant que $T_{CP} > T'_A$ et chemin critique non saturé **faire**
  $t \leftarrow$ tâche du chemin critique / $(\exists C_k / \lceil f(p^{ref}(t), t, k) \rceil < P_k)$ 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 ;
  $p^{ref}(t) \leftarrow p^{ref}(t) + 1$;
  Mettre à jour les T_b^{ref} ;
fin tant que

Dans nos algorithmes ce processeur est ajouté à son allocation de référence. Pour déterminer le nombre de processeurs à allouer à une tâche sur une grappe donnée, d'après son allocation de référence, nous utilisons la loi d'Amdahl. Du fait que nous utilisons un modèle de processeurs uniformes, l'égalité :

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

conduit à :

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

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

$$p^k(t) \quad = \quad \min\{P_k, \lceil f(p^{ref}(t), t, k) \rceil\} \qquad [9]$$

Pour éviter une boucle infinie dans cette procédure, nous définissons une condition d'arrêt supplémentaire qui est la notion de *chemin critique saturé*. Le chemin critique est dit saturé si les allocations de référence sont telles qu'il est impossible de rajouter des processeurs aux tâches qui le composent. Le nombre de processeurs à leur allouer sur toute grappe C_k est alors le nombre total de processeurs de cette grappe (P_k). Les temps d'exécution des tâches du chemin critique ne peuvent donc plus être réduits en augmentant leurs allocations de référence. T_{CP} est alors minimal et nous arrêtons la procédure d'allocation dès que cet état est atteint.

L'algorithme 1 présente notre version hétérogène de la phase d'allocation.

6.2. *Placement des tâches*

Le placement consiste à attribuer à chaque tâche prête choisie *t*, les processeurs qui lui garantissent la plus petite échéance $T_f(t)$. Soit $T_r^k(t_i, t_j)$ le temps nécessaire à la

redistribution des données entre une tâche t_i qui vient de s'exécuter (sur un ensemble de processeurs connu) et une tâche t_j qui lui succède, en considérant son placement 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 aux tâches t_i et t_j. 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 :

$$T_m^k(t) \quad = \quad \max_{t_i \in Pred(t)} \left(T_f(t_i) + T_r^k(t_i, t) \right), \qquad [10]$$

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

$$T_s^k(t) \quad = \quad \max\{dispo(p^k(t)), T_m^k(t)\} \qquad [11]$$

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'algorithme 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. Une fois que cette tâche t est déterminée, nous choisissons l'allocation qui minimise sa date de fin d'exécution :

$$T_f(t) \quad = \quad \min_k \left(T_s^k(t) + T^k(t, p^k(t)) \right) \qquad [12]$$

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 cet algorithme de placement donne lieu à l'heuristique HCPA (*Heterogeneous Critical Path and Area-based*).

Dans notre seconde heuristique d'ordonnancement, la tâche la plus prioritaire est déterminée selon le principe de l'heuristique *sufferage* (Maheswaran *et al.*, 1999). 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'algorithme S-HCPA (*Sufferage-based Heterogeneous Critical Path and Area-based*). 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.

6.3. *Complexité de HCPA et S-HCPA*

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

$$K = \max_{(k,t)} \left\lceil f^{-1}\left(P_k, t, k\right) \right\rceil \quad [13]$$

Dans le pire des cas, il faudrait K itérations pour chaque tâche dans la procédure d'allocation. D'où un total de $K \times N$ itérations de la procédure d'allocation. la complexité du corps de la boucle (choix du chemin critique, calcul de T_{CP}, T_b, et T_A) est de l'ordre de $O((N+E)C)$. Nous en déduisons la complexité de la phase d'allocation : $O\left(N(N+E)C \times K\right)$.

En ce qui concerne la phase de placement, dans le cas de HCPA, les tâches prêtes sont triées par ordre de priorité à l'aide d'une procédure de l'ordre de $O(N^2)$ 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, l'on teste les C allocations possibles. Au total, la procédure de placement de HCPA est de l'ordre de $O(N(N+C))$. Cette complexité est négligeable par rapport à celle de la phase d'allocation. Pour la phase de placement de S-HCPA, dans le pire des cas, toutes les N tâches sont prêtes en même temps et la détermination de la i^{eme} tâche la plus prioritaire a un coût de l'ordre de $C \times (N-i)$. En effet, on examine les C allocations possibles pour chacune des $N-i$ tâches prêtes pas encore placées. La complexité de la procédure de placement de S-HCPA est donc de l'ordre de $C \times N^2$. Cette complexité peut également être négligée par rapport à celle de la phase d'allocation.

Il en découle que la complexité de HCPA et S-HCPA est de l'ordre de

$$O\left(N(N+E)C \times K\right). \quad [14]$$

6.4. *Évaluation des algorithmes*

Dans cette section nous comparons les deux algorithmes obtenus (HCPA et S-HCPA) à CPA (Rădulescu *et al.*, 2001b) et M-HEFT (Casanova *et al.*, 2004), une heuristique qui adapte un algorithme d'ordonnancement de tâches séquentielles sur plates-formes hétérogènes au cas de DAGs de tâches parallèles. En vue de comparer CPA à nos algorithmes, nous générons une plate-forme homogène (en termes de vitesse des processeurs) équivalente à la plate-forme hétérogène considérée. Cette plate-forme équivalente possède la même structure que la plate-forme initiale mais tous les processeurs ont comme vitesse, la vitesse moyenne des processeurs de la plate-forme hétérogène.

La figure 10 montre les performances moyennes des algorithmes en fonction du nombre de grappes (1, 2, 4 ou 8) et sur la totalité des plates-formes générées (toutes). En regardant les moyennes sur l'ensemble des plates-formes, nous observons que nos deux heuristiques sont en moyenne plus performantes en termes de *makespan* par

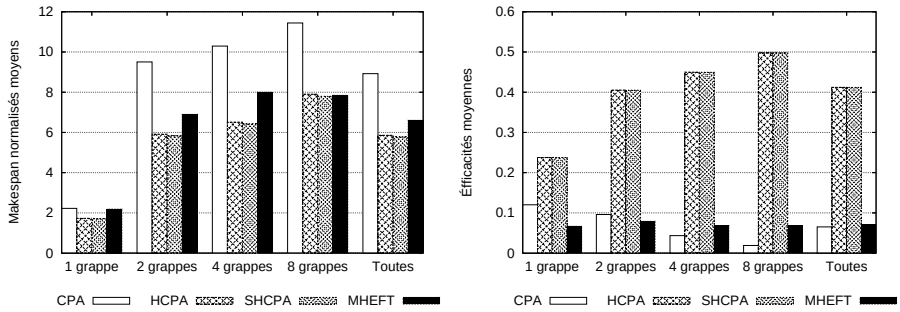

Figure 10. *Impact du nombre de grappes.*

rapport à M-HEFT et CPA bien que cette dernière heuristique, qui ne restreint pas l'exécution d'une tâche à l'intérieur d'une même grappe, soit favorisée par le modèle d'Amdhal, ce modèle ne tenant pas explicitement compte des communications intra-tâches. L'heuristique M-HEFT est telle que les tâches prêtes sont placées l'une après l'autre sur l'ensemble de processeurs qui minimisent leur date de fin d'exécution. Par conséquent, M-HEFT bénéficie pleinement du parallélisme de tâches que si le nombre de grappes de la plate-forme est supérieur au nombre de tâches concurrentes car chaque tâche utilise la totalité de la grappe choisie. La non exploitation du parallélisme de tâches par M-HEFT au sein d'une même grappe explique les mauvais *makespan* obtenus par rapport à nos heuristiques. Nous pouvons donc observer que HCPA et SHCPA fournissent en moyenne un meilleur *makespan* par rapport à M-HEFT sur la majorité des plates-formes sauf sur celles à 8 grappes. La figure 11 confirme le fait que le parallélisme de tâche est mal exploité par M-HEFT. Pour les DAGs peu larges ($largeur = 0{,}1$ ou $largeur = 0{,}2$), M-HEFT donne des *makespan* normalisés moyens meilleurs que ceux de HCPA et SHCPA. À l'inverse, pour les DAGs exhibant plus de parallélisme de tâches ($largeur = 0{,}8$), nos algorithmes donnent de meilleurs *makespan* normalisés.

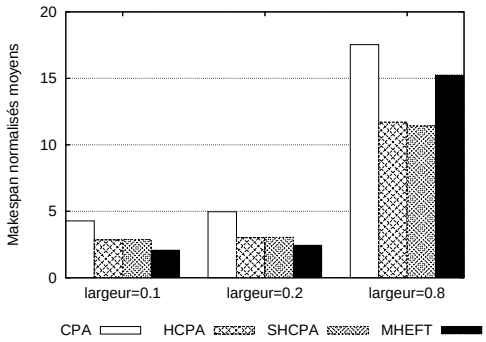

Figure 11. *Impact de la largeur des graphes.*

En outre, du fait que nous allouons des processeurs supplémentaires qu'aux tâches du chemin critique et que nous arrêtons les allocations plus tôt que CPA (utilisation de T'_A, restriction des allocations à l'intérieur d'une seule grappe), nos deux heuristiques utilisent plus rationnellement les ressources comparativement à CPA et M-HEFT quelque soit le nombre de grappes de la plate-forme. Cela se traduit par une efficacité plus importante pour nos deux heuristiques.

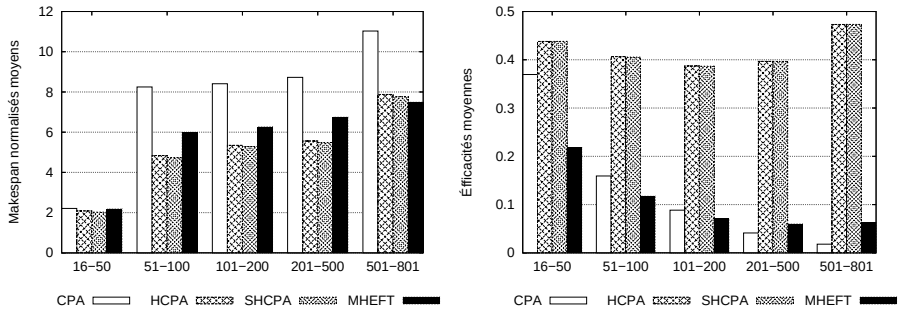

Figure 12. *Impact du nombre de nœuds des plates-formes.*

Dans la figure 12, nous avons regroupé les plates-formes en fonction du nombre de processeurs. Pour un nombre de processeurs P compris entre 16 et 50, Tous les DAGs ont leur nombre de tâches N proches de P. CPA et HCPA ont alors des performances assez proches aussi bien pour les *makespan* que pour les efficacités. En revanche, dès que P est très grand par rapport à N ($P \gg N$), nous pouvons observer l'intérêt de notre contribution en utilisant T'_A au lieu de T_A.

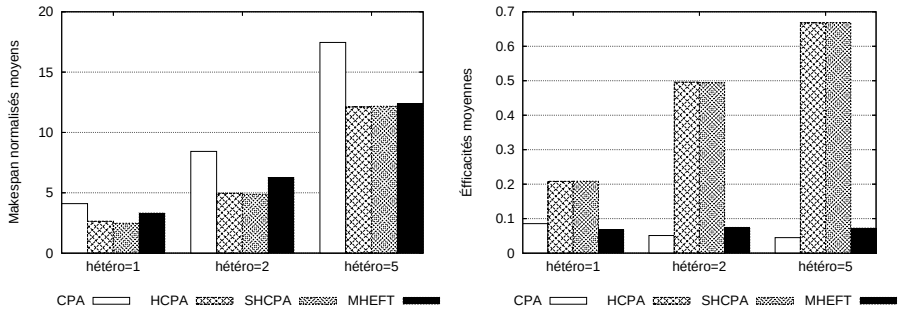

Figure 13. *Impact de l'hétérogénéité des plates-formes.*

Dans la figure 13, lorsque l'hétérogénéité des plates-formes est égale à 1, nous avons encore une illustration de l'impact de l'utilisation de T'_A. Ces plates-formes sont homogènes en termes de vitesse de processeurs mais peuvent être composées de plusieurs grappes et d'un grand nombre de processeurs. CPA pouvant utiliser des allocations multi-sites, elle obtient des performances moins bonnes. Lorsque l'hétéro-

généité augmente, l'écart se creuse entre nos heuristiques et CPA aussi bien vis à vis du *makespan* que sur l'utilisation des ressources.

Algorithme	Accélération % SEQ	Efficacité
CPA	$5,12$	$6,51\%$
HCPA	$7,28$	$41,21\%$
S-HCPA	$7,36$	$41,20\%$
MHEFT	$7,75$	$7,11\%$

Tableau 2. *Performances relatives par rapport à SEQ.*

Le tableau 2 illustre les performances relatives des algorithmes par rapport à un ordonnancement séquentiel (SEQ) où nous avons calculé les moyennes sur la totalité des simulations. On peut constater que HCPA et S-HCPA sont largement plus efficaces que CPA et M-HEFT tout en permettant d'obtenir les meilleurs accélérations. Nous rappelons au lecteur que l'efficacité est le ratio entre le gain en temps de complétion et l'utilisation des ressources par rapport au meilleur algorithme séquentiel.

Enfin pour ce qui est de la comparaison HCPA et S-HCPA, il apparaît que les deux heuristiques donnent des résultats sensiblement proches. Une observation plus fine des résultats de simulations révèle que S-HCPA produit un meilleur *makespan* que HCPA dans $21,65\%$ des cas et un *makespan* égal à celui de HCPA dans $54,76\%$ des cas, HCPA étant meilleur dans les $23,59\%$ de cas restants. Cela signifie que les heuristiques de liste fondées sur le chemin critique pour le placement des tâches prêtes ne minimisent pas toujours le temps de complétion des applications parallèles. D'autres stratégies peuvent être explorées.

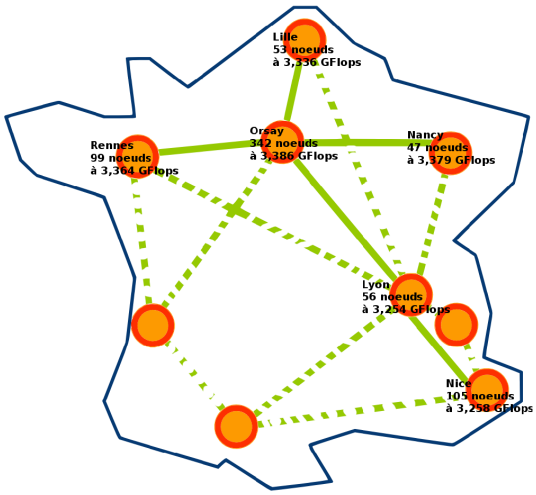

Figure 14. *Exemple de grappe de grappes extraite de grid'5000.*

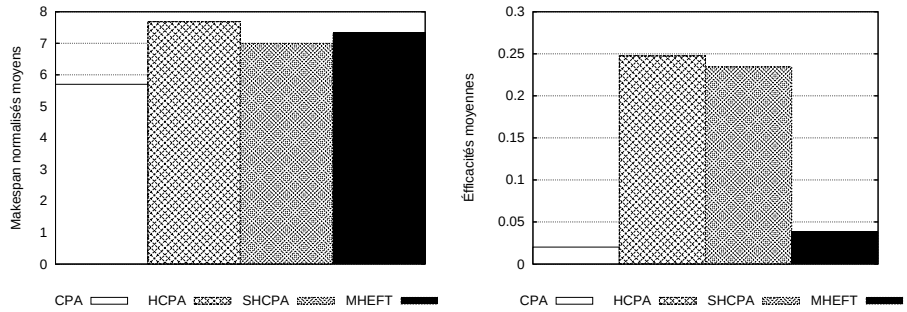

Figure 15. *Comparaison sur un sous-ensemble de grid'5000.*

La figure 15 montre les résultats de la simulation des algorithmes sur une grappe de grappes extraite d'une plate-forme réelle. Cette plate-forme est un sous-ensemble de la grille expérimentale *grid'5000*[2] (voir figure 14). Nous avons retenu 6 grappes sur les sites de Lille, Lyon, Nancy, Nice, Orsay et Rennes. Elle comprend au total 702 processeurs et un réseau relativement rapide dont 5 dorsales à $10Gb/s$. Sur cette plate-forme de faible hétérogénéité (tous les processeurs ont une vitesse environ égale à $3{,}3GFlops$) où le nombre de processeurs ($P = 702$) est très élevé par rapport aux nombre de tâches des applications ($N \leq 50$), nous constatons que les efficacités de M-HEFT et CPA sont très faibles par rapport à HCPA et S-HCPA. Or HCPA et SHCPA fournissent des temps de complétion comparables à ceux de CPA et M-HEFT. Cela confirme les bonnes performances de nos deux algorithmes qui fournissent un meilleur compromis entre l'utilisation des ressources et le temps de complétion des applications.

7. Conclusion

Dans cet article, nous avons adapté un algorithme d'ordonnancement de tâches parallèles sur plates-formes homogènes au cas des plates-formes hétérogènes. Pour cela nous avons choisi un algorithme d'ordonnancement en deux étapes de la littérature : CPA (Rădulescu *et al.*, 2001b) dont les performances sont relativement bonnes par rapport aux algorithmes concurrents si on tient compte à la fois de la complexité de ces algorithmes, des temps de complétion obtenus et des efficacités.

Dans un premier temps, nous avons proposé deux améliorations de CPA dans le cas des plates-formes homogènes. La première concerne la phase d'allocation et la seconde la phase de placement. Dans la phase d'allocation, nous avons redéfini la notion d'aire moyenne afin d'arrêter plus tôt la procédure d'allocation de processeurs et de profiter au mieux de l'exécution en parallèle des tâches concurrentes. Nous avons expérimentalement vérifié que cette modification permet en moyenne de réduire le temps

2. https ://www.grid5000.fr

de complétion des applications tout en utilisant peu de ressources par rapport à CPA. Dans la phase de placement, nous avons mis en place une technique de *tassage* dont le but est de réduire le temps d'attente et la date de fin d'exécution de la tâche prête à placer en diminuant son allocation si cela est nécessaire. Nous avons également vérifié que cette amélioration réduit le temps de complétion des applications par rapport à CPA. De plus, la combinaison de ces deux améliorations fournit un léger gain par rapport à chacune des améliorations prise individuellement, la première amélioration ayant plus d'impacts que la seconde.

Ensuite, nous avons adapté l'heuristique obtenue en modifiant la procédure d'allocation au cas de plates-formes hétérogènes de type grappe hétérogène de grappes homogènes. À cet effet, nous avons introduit la notion de *grappe de référence* qui nous a permis de mieux gérer l'hétérogénéité des plates-formes dans la procédure d'allocation. Le but de cette virtualisation des plates-formes était surtout de conserver une faible complexité pour nos heuristiques. Ainsi, nous avons mis en place deux heuristiques qui se distinguent par leur phase de placement : HCPA et S-HCPA. Dans HCPA, nous adaptons la technique de placement de CPA au cas où nous disposons de plusieurs grappes, et donc de plusieurs allocations possibles. La phase de placement de S-HCPA est quant à elle basée sur l'heuristique *sufferage* (Maheswaran *et al.*, 1999).

La complexité de nos deux heuristiques reste de l'ordre de celle de CPA. Leur évaluation en milieu hétérogène révèle qu'elles sont plus performantes que CPA et qu'elles consomment beaucoup moins de ressources par rapport à M-HEFT (Casanova *et al.*, 2004) tout en menant à des temps de complétion en moyenne comparables et même sensiblement meilleurs par rapport à ceux de M-HEFT.

Dans nos travaux futurs, nous essaierons de mettre en œuvre des heuristiques qui gèrent équitablement l'exécution concurrente de plusieurs DAGs de tâches parallèles sur des agrégations de grappes de calcul. À l'instar de HCPA et S-HCPA, ces heuristiques doivent fournir un bon compromis entre la gestion des ressources et les *makespan*. Pour ces études à venir, nous projetons d'utiliser des plates-formes plus réalistes. Il sera par exemple possible d'avoir des topologies réseau complexes avec des sous-graphes non couplés. Nous prévoyons également l'utilisation de modèles de tâches parallèles qui incluent les coûts des communications intra-tâches.

8. Bibliographie

Amdahl G., « Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities », *AFIPS 1967 Spring Joint Computer Conference*, vol. 30, p. 483-485, April, 1967.

Anderson T. E., Culler D. E., Patterson D. A., « A Case for NOW (Networks Of Workstations). », *IEEE Micro*, vol. 15, n° 1, p. 54-64, 1995.

Boudet V., Desprez F., Suter F., « One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication », *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, April, 2003.

Casanova H., Desprez F., Suter F., « From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling », *10th Int. Euro-Par Conference*, vol. 3149 of *LNCS*, Springer, Pisa, Italy, p. 230-237, August, 2004.

Chrétienne P., « Task Scheduling Over Distributed Memory Machines », *Parallel and Distributed Algorithms*, North Holland, p. 165-176, 1988.

Dutot P.-F., « Hierarchical Scheduling for Moldable Tasks », *11th Int. Euro-Par Conference*, vol. 3648 of *LNCS*, Springer, Lisbon, Portugal, p. 302-311, August, 2005.

Legrand A., Marchal L., Casanova H., « Scheduling Distributed Applications : The SimGrid Simulation Framework », *3rd IEEE Symposium on Cluster Computing and the Grid (CCGrid'03)*, Tokyo, p. 138-145, May, 2003.

Lepère R., Trystram D., Woeginger G., « Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints », *IJFCS*, vol. 13, n° 4, p. 613-627, 2002.

Maheswaran M., Ali S., Siegel H. J., Hensgen D., Freund R. F., « Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems », *HCW '99 : Proceedings of the Eighth Heterogeneous Computing Workshop*, IEEE Computer Society, Washington, DC, USA, p. 30, 1999.

Ramaswamy S., Sapatnekar S., Banerjee P., « A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers », *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, n° 11, p. 1098-1116, Nov, 1997.

Rauber T., Rünger G., « Compiler Support for Task Scheduling in Hierarchical Execution Models », *Journal of Systems Architecture*, vol. 45, p. 483-503, 1998.

Rădulescu A., Nicolescu C., van Gemund A., Jonker. P., « CPR : Mixed Task and Data Parallel Scheduling for Distributed Systems », *15-th International Parallel and Distributed Processing Symposium (IPDPS)*, apr, 2001a. Best Paper Award.

Rădulescu A., van Gemund A., « A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling », *15th International Conference on Parallel Processing (ICPP)*, Valencia, Spain, September, 2001b.

Topcuoglu H., Hariri S., Wu M., « Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing », *IEEE TPDS*, vol. 13, n° 3, p. 260-274, 2002.

Turek J., Wolf J., Yu P., « Approximate Algorihtms for Scheduling Parallelizable Tasks », *4th ACM Symp. on Parallel Algorithms and Architectures*, p. 323-332, 1992.