Algorithmes d'ordonnancement de graphes de tâches parallèles sur plates-formes hétérogènes en deux étapes

Tchimou N'Takpé

RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

Algorithmes d'ordonnancement de graphes de tâches parallèles sur plates-formes hétérogènes en deux étapes

Tchimou N'takpé*
Nancy Université / LORIA †
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. Ainsi, nous proposons d'adapter une heuristique d'ordonnancement de tâches parallèles en milieu homogène au cas d'une plate-forme hétérogène.

Mots-clés : Heuristiques d'ordonnancement, tâches parallèles, graphes de tâches, grappes de grappes

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 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 permettent 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* [11]. 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.
De nombreuses études ont été réalisées pour l'ordonnancement de graphes de tâches parallèles dans le cas de plates-formes homogènes [7, 8, 9, 10]. Or les plates-formes hétérogènes sont de plus en plus répandues et très attrayantes car elles peuvent permettre de déployer des applications parallèles avec des échelles sans précédents. Il est donc nécessaire de développer des heuristiques d'ordonnancement pour ces systèmes hétérogènes. Une première approche a consisté à adapter une heuristique d'ordonnancement de tâches séquentielles sur plates-formes hétérogènes au cas des tâches parallèles [1]. Ici, nous utilisons une approche complémentaire qui consiste à modifier un algorithme d'ordonnancement de tâches parallèles en milieu homogène [8, 9] et à prendre en compte l'hétérogénéité des ressources.
Les contributions de cet article sont : (i) un nouveau concept de virtualisation des plates-formes qui permet de gérer plus facilement les allocations de ressources ; (ii) une nouvelle manière de calculer l'aire

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

moyenne utilisée dans la procédure d'allocation ; (iii) une nouvelle méthode de placement fondée sur l'idée de *sufferage* [2] ; (iv) une méthode d'évaluation par simulation de nombreux scénarios.
Dans la section suivante, nous présentons les modèles de plates-formes et d'applications utilisés. La section 3 traite des travaux précédents et formalise le problème. La section 4 montre comment nous adaptons l'algorithme CPA aux plates-formes hétérogènes. La section 5 décrit notre méthode d'évaluation et la section 6 présente les résultats des simulations. Enfin, la section 7 nous permet de conclure et de faire une ouverture sur les travaux futurs.

2. Modèles de plates-formes et d'applications

Nous considérons des agrégations hétérogènes de grappes homogènes. Ces plates-formes sont représentatives de certaines infrastructures de grilles réparties entre des institutions, qui disposent généralement de grappes homogènes. Nous avons donc C grappes et chaque grappe C_i contient P_i 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 1 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 (Backbone) commun.

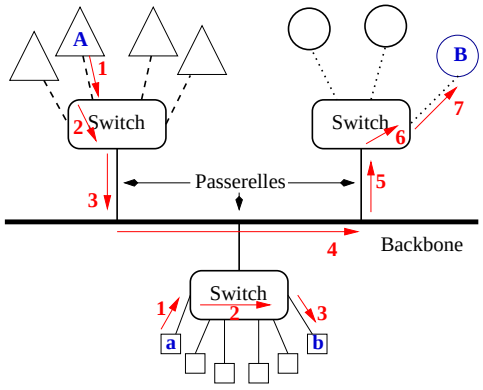

FIG. 1 – Modèle de plate-forme avec les routes entre deux processeurs A et B localisés sur deux grappes différentes et entre deux processeurs a et b de la même grappe.

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 arêtes. Les nœuds représentent les tâches parallèles et les arêtes définissent les relations de précédence (dépendances de flots ou de données) entre les tâches. On associe à chaque arête $E_{i,j}$, 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. Une tâche est *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 où le temps d'exécution parallèle est donné par :

$$T_\omega(t, N_p(t)) = \left(\alpha + \frac{1-\alpha}{N_p(t)}\right) \cdot T_\omega(t, 1),$$

$N_p(t)$ étant le nombre de processeurs alloués à t, $T_\omega(t,1)$ son temps d'exécution sur un seul processeur et α sa portion non parallélisable. Dans cette étude, nous restreignons l'exécution d'une tâche parallèle à l'intérieur d'une grappe, afin de garantir la validité de ce modèle. Ce choix est également motivé par le

fait qu'en pratique les communications inter-grappes peuvent être très coûteuses. On définit enfin $T_b(t)$, le *bottom level*, comme étant le chemin le plus long (en termes de temps d'exécution) depuis la tâche t, en incluant son propre temps d'exécution, jusqu'à une *tâche de sortie* quelconque.

3. 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 [3]. De nombreuses heuristiques ont donc été conçues pour ordonnancer des tâches parallèles. Dans [5], un algorithme d'ordonnancement de tâches modelables interdépendantes sur une hiérarchie de machines multiprocesseurs est présenté. Dans [1], 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.

3.1. Ordonnancement sur plates-formes homogènes

Si certaines études théoriques existent [7], la plupart des algorithmes d'ordonnancement de tâches parallèles [8, 9, 10] sont des algorithmes en deux étapes et ont été conçus pour des milieux homogènes. La première étape vise à déterminer un nombre de processeurs optimal pour chaque tâche. Dans la seconde étape, les auteurs utilisent des heuristiques de liste pour ordonnancer les tâches.
Dans [9], 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 auteurs de [10] 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 de tâches.

3.2. CPA

Dans cet article, nous nous intéressons à l'algorithme CPA [8] qui est le plus performant des algorithmes en deux étapes que nous venons de citer. 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 qui représente le temps moyen d'occupation des processeurs. Rădulescu *et al.* remarquent que le temps d'exécution d'une application parallèle peut ainsi être approché par $T_p^e = \max\{T_{CP}, T_A\}$, où T_{CP} est la longueur du chemin critique et T_A, l'aire moyenne :

$$T_{CP} = \max_{t \in \mathcal{N}} T_b(t), \text{ et } T_A = \frac{1}{P} \sum_{i=0}^{N} \left(T_\omega \left(t_i, N_p(t_i) \right) \times N_p(t_i) \right). \quad (1)$$

Le but de CPA est donc 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. On alloue donc un processeur à chaque tâche à l'instant initial. 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 est celle appartenant au chemin critique et dont le rapport $T_\omega(t, N_p(t))/N_p(t)$ 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 de sa valeur minimale ($T_p^e \approx T_{CP} \approx T_A$).
À chaque itération de la procédure de placement, la tâche prête ayant le plus grand *bottom level* est choisie pour être placée. Cette phase tient compte du coût des redistributions des données pour déterminer la date de début d'exécution, $T_s(t)$, et la date de fin d'exécution, $T_f(t)$, de chaque tâche t.

4. Adaptation de CPA aux plates-formes hétérogènes

La plate-forme cible étant constituée de C grappes, notre idée consiste à attribuer, lors de la première phase, une *allocation de référence* qui représente les C allocations potentielles à chaque tâche. Nous définirons plus loin comment déterminer le nombre de processeurs à allouer à une tâche sur une grappe en fonction de son allocation de référence. Lors de la phase de placement nous retiendrons parmi ces allocations potentielles celle qui minimise la date de fin d'exécution de la tâche.
Étant donné qu'il existe maintenant 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 de CPA. 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 : $P_{ref} = \lceil \sum_{i=0}^{C-1} P_i/r_i \rceil$ où r_i est le rapport de la puissance d'un processeur de la grappe de référence sur celle d'un processeur de la grappe C_i. L'utilisation de cette grappe homogène virtuelle permet de conserver une faible complexité dans le nouvel algorithme. Notons ${N_p}^{ref}(t)$, 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 : ${T_\omega}^{ref}(t, {N_p}^{ref}(t))$. La longueur du chemin critique est donc :

$$T_{CP} = \max_{t \in \mathcal{N}} T_b^{ref}(t), \tag{2}$$

où $T_b^{ref}(t)$ est le *bottom level* en utilisant les allocations de référence des tâches. 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 plus grand que le nombre de tâches (N) d'un ordre de grandeur. Nous proposons donc le compromis suivant qui permet d'arrêter plus vite la procédure d'allocation au cas ou le nombre de ressources est très élevé en prenant $\min(P_{ref}, \sqrt{P_{ref} \times N})$ au lieu de P_{ref} :

$$T_A = \frac{1}{\min\{P_{ref}, \sqrt{P_{ref} \times N}\}} \sum_{i=0}^{N} \left(T_\omega^{ref}\left(t_i, N_p^{ref}(t_i)\right) \times N_p^{ref}(t_i) \right). \tag{3}$$

Les deux sections qui suivent décrivent les deux étapes de nos algorithmes originaux.

4.1. Allocation de processeurs

Dans cette phase, nous ne tenons pas compte du coût des communications entre les différentes tâches du DAG. En effet la longueur du chemin critique de l'application dépend uniquement des temps d'exécution des tâches. Le coût des transferts de données sera pris en compte lors de la phase de placement pour déterminer les dates de début et de fin d'exécution de chaque tâche. 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. 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. L'égalité :

$$T_\omega^i(t, N_p^i(t)) = T_\omega^{ref}(t, N_p^{ref}(t)) \text{ conduit à : } f(N_p^{ref}(t), t, i) = \frac{(1-\alpha) \cdot T_\omega^i(t,1)}{T_\omega^{ref}(t, N_p^{ref}(t)) - \alpha \cdot T_\omega^i(t,1))},$$

f étant la fonction qui permet de déduire $N_p^i(t)$. Puisque l'on ne peut allouer plus de processeurs qu'il en existe sur une grappe et que le nombre de processeurs alloués doit être un nombre entier, on en déduit :

$$N_p^i(t) = \min\{P_i, \lceil f(N_p^{ref}(t), t, i) \rceil\} \tag{4}$$

Pour éviter une boucle infinie dans cette procédure, nous définissons une nouvelle condition d'arrêt qui est la notion de *chemin critique saturé*. Le chemin critique est saturé si les allocations de référence sont telles qu'il est impossible de rajouter des processeurs aux tâches qui le composent : le nombre de

Algorithme 1 Allocation de processeurs

pour tout $t \in \mathcal{N}$ **faire**
  ${N_p}^{ref}(t) \leftarrow 1$;
  $N_p^i(t) \leftarrow 1, \forall i \in [0, C-1]$;
fin pour
tant que $T_{CP} > T_A$ et chemin critique non saturé **faire**
  $t \leftarrow$ tâche du chemin critique / $(\exists i / \lceil f({N_p}^{ref}(t), t, i) \rceil < P_i)$ et
  $\left(\frac{{T_\omega}^{ref}(t,{N_p}^{ref}(t))}{{N_p}^{ref}(t)} - \frac{{T_\omega}^{ref}(t,{N_p}^{ref}(t)+1)}{{N_p}^{ref}(t)+1}\right)$ est maximum ;
  ${N_p}^{ref}(t) \leftarrow {N_p}^{ref}(t) + 1$;
  Mettre à jour les T_b^{ref} ;
fin tant que

processeurs à leur allouer sur toute grappe C_i est le nombre total de processeurs de cette grappe. 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 dès que cet état est atteint. L'algorithme 1 présente notre version hétérogène de la phase d'allocation.

4.2. Placement des tâches

L'ordonnancement va maintenant consister à attribuer à chaque tâche prête choisie t, l'allocation qui lui garantit la plus petite échéance $T_f(t)$. Notons $T_r^k(t_i, t_j)$ le coût de la redistribution des données lorsque l'on passe d'une tâche t_i qui vient de s'exécuter sur un ensemble de processeurs connus à l'exécution d'une de ses tâches filles t_j sur une grappe C_k. Ce coût dont nous tenons maintenant compte dépend entre autres 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 $T_m^i(t)$ la date d'arrivée du dernier message d'une tâche prête t si celle-ci devait s'exécuter sur la grappe C_i. L'on a :

$$T_m^i(t) = \max_{t_j \in Pred(t)} (T_f(t_j) + T_r^i(t_j, t)), \tag{5}$$

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

$$T_s^i(t) = \max\{dispo(N_p^i(t)), T_m^i(t)\} \tag{6}$$

où $dispo({N_p}^i(t))$ est la date à laquelle la grappe C_i aura au moins ${N_p}^i(t)$ processeurs libres.
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 :

$$T_f(t) = \min_i (T_s^i(t) + T_\omega^i(t, N_p^i(t))) \tag{7}$$

On en déduit C_i, la grappe sur laquelle son exécution est prévue ainsi que sa date de début d'exécution : $T_s(t) = T_s^i(t)$. 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* [2]. Ce principe consiste à choisir parmi les tâches prêtes celle qui serait la plus pénalisée si 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 qui sont fondées sur la réduction du 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.

5. Méthodologie d'évaluation

Pour l'évaluation de nos algorithmes, nous avons eu recours à des simulations afin d'explorer une large variété de plates-formes et de DAGs. Pour cela nous utilisons SIMGRID [6], une boîte à outils conçue pour la simulation de grilles de calcul et/ou la mise en œuvre d'applications distribuées.

On génère des plates-formes constituées de 1, 2, 4 ou 8 grappes. Le nombre de processeurs de chaque grappe est tiré aléatoirement entre 16 et 128. Les liens intra-grappes sont soit du Fast Ethernet (débit $=$ $100\mathrm{Mb/s}$ et latence $=$ $100\mu\mathrm{s}$) soit du Giga Ethernet (débit $=$ $1\mathrm{Gb/s}$ et latence $=$ $100\mu\mathrm{s}$). Ces liens peuvent subir des contentions. Le backbone a un débit de $2,5\mathrm{Gb/s}$ et une latence de $50\mathrm{ms}$. Chaque passerelle a une capacité de $1\mathrm{Gb/s}$ avec une latence de $100\mu\mathrm{s}$. La borne inférieure des vitesses des processeurs (en GFlops) est de $0,25$, $0,5$, $0,75$ ou 1. Le rapport entre cette borne inférieure et la borne supérieure des vitesses des processeurs de la plate-forme qui, constitue son degré d'hétérogénéité, est prise 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. Pour chaque combinaison de ces paramètres nous générons 5 échantillons, soit un total de 200 plates-formes.

Les graphes de tâches sont générés en tirant la taille des données n, un multiple de 1024 (1 ko), entre les valeurs 2048 et 11268, ce qui correspond au plus à 1 Go d'occupation mémoire. Ensuite, la complexité des tâches est fixée à $a \cdot N$, $a \cdot N\log N$, ou $a \cdot N^{3/2}$, ou tirée au sort parmi les trois valeurs précédentes. $N = n^2$ 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 $0,2$. Le coût des transferts est égal à N, où N est relatif à la tâche qui génère le transfert.

Quatre paramètres permettent de régler 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.

On soumet 5 algorithmes différents aux plates-formes et aux DAGs générés. Ces algorithmes sont CPA, HCPA et S-HCPA que nous venons de décrire ; M-HEFT [1], l'adaptation aux tâches parallèles d'une heuristique d'ordonnancement de tâches séquentielles en milieu hétérogène et SEQ qui ordonnance séquentiellement les tâches sur le processeur le plus rapide. Il en résulte un total de 1296000 simulations.

SEQ est utilisé pour évaluer les performances relatives des quatre autres algorithmes. Bien que CPA soit destiné aux grappes homogènes, il peut être utilisé sur des grappes de grappes. Pour cela, la simulation de CPA s'effectue sur des grappes de grappes équivalentes dont tous les processeurs ont la vitesse moyenne de la plate-forme initiale.

Pour comparer les quatre algorithmes, nous mesurons le temps de complétion des applications (*makespan*), *i.e.*, la différence entre leur date de début et leur date de fin d'exécution ainsi que la puissance moyenne utilisée durant l'exécution des applications (en Gflops).

6. Résultats des simulations

La figure 2 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 que CPA par rapport au *makespan* bien que ce dernier, qui ne restreint pas l'exécution d'une tâche à l'intérieur d'une même grappe, soit favorisé par le modèle d'Amdhal, ce modèle ne tenant pas compte des communications intra-tâches. M-HEFT est un peu meilleur mais n'utilise le parallélisme de tâches que s'il y a plusieurs grappes. En effet, le modèle utilisé fait que sur une seule grappe, toutes les tâches sont placées sur tous les processeurs l'une après l'autre. D'où les moins bonnes performances de M-HEFT sur une grappe. Nos deux heuristiques utilisent beaucoup moins de ressources que CPA et M-HEFT quelque soit le nombre de processeurs de la plate-forme.

Dans la figure 4, nous avons regroupé les plates-formes en fonction du nombre de processeurs. Pour un

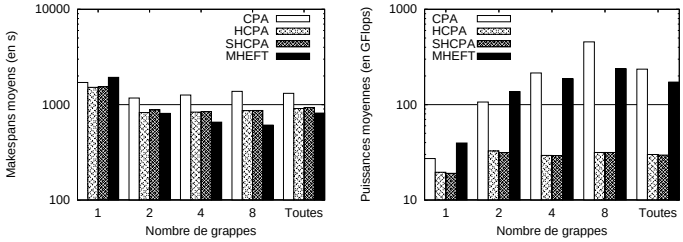

FIG. 2 – Impact du nombre de grappes.

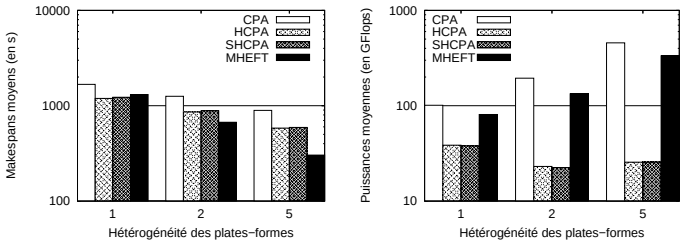

FIG. 3 – Impact de l'hétérogénéité des plates-formes.

nombre de processeurs P compris entre 16 et 50, Tous les DAGs ont leur nombre de tâches N proches de P. Donc CPA utilise les ressources de façon rationnelle. En revanche, dès que $P > N$, alors on note l'intérêt de notre contribution sur le calcul de T_A.
Dans la figure 3, lorsque l'hétérogénéité des plates-formes est égale à 1, on a encore une illustration de l'impact du changement du calcul de T_A, car 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. Or CPA pouvant gérer des allocations multi-sites, il obtient des performances moins bonnes. Lorsque l'hétérogénéité augmente, l'écart se creuse entre nos heuristique et CPA aussi bien vis à vis du *makespan*

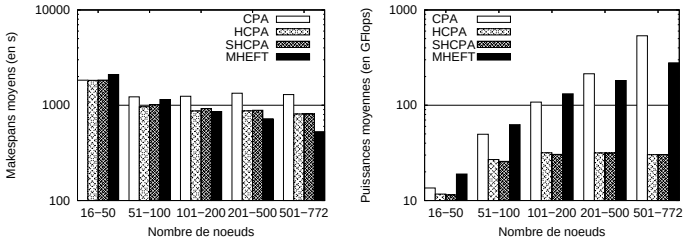

FIG. 4 – Impact du nombre de nœuds des plates-formes.

Algorithme	Makespan SEQ/Makespan algo	Puissance algo/Puissance SEQ	Efficacité
CPA	6,45	170,87	3,8%
HCPA	9,32	21,72	42,9%
S-HCPA	9,08	21,41	42,4%
MHEFT	10,39	124,93	8,3%

TAB. 1 – Performances relatives par rapport à SEQ

que sur l'utilisation des ressources.
Le tableau 1 illustre les performances relatives des algorithmes par rapport à un ordonnancement séquentiel où nous avons calculé les moyennes sur la totalité des simulations. On note clairement que HCPA et S-HCPA sont plus efficaces que CPA et M-HEFT, l'efficacité étant 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 HCPA est en moyenne légèrement meilleure au niveau des *makespan*. Mais une observation plus fine des résultats des simulations révèle que S-HCPA produit un résultat meilleur à celui de HCPA dans 21% des cas et, un *makespan* égale à celui de HCPA dans 36% des cas. Cela signifie que les heuristiques d'ordonnancement de liste fondées sur la réduction du chemin critique ne minimisent pas toujours le temps de complétion des applications parallèles et donc, d'autres stratégies peuvent être explorées.

7. Conclusion et travaux futurs

Dans cet article, nous avons proposé deux algorithmes d'ordonnancement d'applications composées de tâches parallèles sur des plates-formes hétérogènes : HCPA et S-HCPA. Ces algorithmes sont l'adaptation d'une heuristique en deux étapes pour milieux homogènes, CPA, au cas hétérogène. Nous avons introduit la notion de *grappe de référence* qui nous a permis de mieux gérer l'hétérogénéité des plates-formes et de conserver une complexité de l'ordre de celle de CPA. L'évaluation de nos heuristiques révèle qu'elles sont plus performantes que CPA et qu'elles gèrent mieux l'utilisation des ressources par rapport à M-HEFT [1].
Dans nos travaux futurs, nous projetons d'utiliser des plates-formes plus réalistes telles que *Grid'5000* (`www.grid5000.fr`). Il pourra être possible d'avoir des topologies plus 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. Il s'agira de modèles d'accélération plus réalistes [4] ou de profils dépendant à la fois des complexités en calcul et en communication.

Bibliographie

1. H. Casanova, F. Desprez, and F. Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In *10th Int. Euro-Par Conference*, volume 3149 of *LNCS*, pages 230–237, Pisa, Italy, August 2004. Springer.
2. H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *9th Heterogeneous Computing Workshop (HCW'00)*, pages 349–363, Cancun, Mexico, 2000.
3. P. Chretienne. Task Scheduling Over Distributed Memory Machines. In *Parallel and Distributed Algorithms*, pages 165–176, North Holland, 1988.
4. A. Downey. A Model For Speedup of Parallel Programs. Technical Report UCB/CSD-97-933, EECS Department, University of California, Berkeley, 1997.
5. P.-F. Dutot. Hierarchical Scheduling for Moldable Tasks. In *11th Int. Euro-Par Conference*, volume 3648 of *LNCS*, pages 302–311, Lisbon, Portugal, August 2005. Springer.
6. A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications : The SimGrid Simulation Framework. In *3rd IEEE Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 138–145, Tokyo, May 2003.
7. R. Lepère, D. Trystram, and G. Woeginger. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. *IJFCS*, 13(4) :613–627, 2002.
8. A. Radulescu, C. Nicolescu, A. van Gemund, and P. Jonker. Mixed Task and Data Parallel Scheduling for Distributed Systems. In *15th Int. Parallel and Distributed Processing Symp. (IPDPS)*, Apr 2001.
9. S. Ramaswany. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, Univ. of Illinois, Urbana-Champaign, 1996.
10. T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45 :483–503, 1998.
11. J. Turek, J. Wolf, and P. Yu. Approximate Algorihtms for Scheduling Parallelizable Tasks. In *4th ACM Symp. on Parallel Algorithms and Architectures*, pages 323–332, 1992.