RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Comparaisons entre le simulateur `SimGrid` et une application réelle pour évaluer les algorithmes d'équilibre de charge et de redistribution de données

Hélène Renard
UNS/CNRS, Modalis, 930 route des Colles, BP 145, 06903 Sophia Antipolis Cedex, France
Helene.Renard@polytech.unice.fr

Résumé

Cet article est centré autour de l'étude et l'implémentation de techniques d'équilibrage de charge et de redistribution (que nous avons proposés dans des travaux précédents) dans le contexte d'une application réelle. Cette étude a pour objectif de comparer le comportement de nos algorithmes sur de deux plates-formes différentes : la première est simulée - `SimGrid`- tandis que la seconde - `Grid'5000`- est réelle. Nous considérerons le cas de plates-formes hétérogènes (qu'elles soient simulées ou réelles) sur lesquelles des calculs indépendants sont effectués en parallèle en utilisant une organisation des processeurs en anneau virtuel. Dans cet article, nous insisterons tout particulièrement sur l'évaluation de nos algorithmes lorsque des variations de charge surviennent. Enfin, notre étude expérimentale illustrera le fait que le comportement simulé est voisin de celui observé lors d'une exécution réelle.

1. Introduction

Nous nous intéressons à la mise en œuvre d'algorithmes itératifs sur des grappes hétérogènes. Ces algorithmes fonctionnent avec un volume important de données, qui sera réparti sur l'ensemble des processeurs. À chaque itération, des calculs indépendants seront effectués en parallèle et certaines communications auront lieu. Ce schéma est très général, et couvre un large spectre de méthodes dans le domaine du calcul scientifique, partant de solvers basés sur des maillages (solveurs d'équations aux dérivées partielles), au traitement du signal (convolution recursive, etc ...) en passant par algorithmes de traitement d'images.
Nous énonçons le problème comme suit : l'algorithme itératif fonctionne répétitivement sur une matrice rectangulaire de données qui est divisée en tranches verticales allouées aux processeurs. À chaque étape de l'algorithme, les tranches sont mises à jour localement et les informations frontières sont échangées entre tranches consécutives. Cette contrainte géométrique implique que les processeurs soient organisés en anneau virtuel. Chaque processeur communiquera seulement deux fois, une fois avec son prédécesseur (virtuel) dans l'anneau et une fois avec son successeur. Il n'existe pas de raison *a priori* de réduire le partitionnement des données à une unique dimension et de ne l'appliquer que sur un anneau de processeurs unidimensionnel. Cependant, un tel partitionnement est très naturel et nous montrerons que trouver l'optimal est déjà très difficile.
Nous considérons une grappe totalement hétérogène, composée de processeurs de vitesses de calculs différentes, communicant par des liens de bandes passantes différentes. Du point de vue architectural, le problème se décompose en deux parties : (i) sélectionner les processeurs participant à la solution et décider de leur position dans l'anneau ; (ii) définir les routes pour aller d'un processeur à son successeur.
Une difficulté majeure est que certaines routes partageront des liens physiques, les réseaux de communication de grappes hétérogènes n'étant pas totalement connectés. Si plusieurs routes partagent le même lien physique, nous décidons quelle fraction de bande passante sera attribuée à chaque route. Nous avons proposé dans un travail précédent (voir [9]) un algorithme qui construit un anneau virtuel optimal de processeurs du point de vue d'équilibrage de charge.
Une fois que l'algorithme d'équilibrage de charge a été appliqué, nous nous intéressons au problème de redistribution de données. Du fait des variations de performances des ressources (puissance de calcul disponible des processeurs, capacité de la bande passante) et des demandes du système (tâches terminées, nouvelles tâches, etc.), les données doivent être redistribuées sur l'ensemble des processeurs participants afin de mieux équilibrer la charge.
Dans cet article, nous nous intéressons à l'équilibrage de charge, à la redistribution de données dans le

contexte d'une application qui, dans notre cas, est la propagation de la chaleur. Nous allons étudier le comportement de nos algorithmes sur des plates-formes hétérogènes. Pour ce faire, nous allons comparer les deux cadres expérimentaux. Le premier est ce qu'on appelle la simulation de plates-formes - `SimGrid` [5] - pour l'étude du comportement de nos algorithmes. Le second est l'implémentation en exécution réelle pour la mise en œuvre de notre application. Notez que l'objet de cet article est l'étude des algorithmes et la comparaison des comportements observés entre la exécution réelle et la simulation de `SimGrid`.

La section suivante (Section 2) est consacrée aux fondements théoriques de notre travail. Nous allons brièvement décrire notre équilibrage de charge avec les données correspondantes pour la redistribution de données. Puis, en Section 3, nous décrivons notre application (i.e. la propagation de la chaleur) et d'illustrer comment elle s'intègre dans notre modèle. Après, dans la section 4, nous montrons comment nous avons mis en place notre application de propagation de la chaleur afin qu'elle corresponde à notre modèle. Ensuite, nous présentons notre cadre expérimental, à savoir comment perturber la plate-forme et générer des plates-formes simulées. Puis, nous donnons, dans la section 5, des résultats expérimentaux où nous présentons une comparaison entre des exécutions simulées et des exécutions réelles illustrant l'intérêt de `SimGrid`. Nous donnons dans la section 6 un état de l'art des travaux déjà publiés. Enfin, nous concluons à la section 7.

2. Cadre de travail

2.1. Équilibrage de charge

Coûts de calcul. La plate-forme de calcul ciblée est représentée par un graphe orienté $G = (P, E)$. Chaque nœud P_i du graphe, $1 \leq i \leq |P| = p$, représente une ressource de calcul, et est pondéré par son temps de cycle w_i : P_i nécessite w_i unités de temps pour effectuer une tâche unitaire.

Coûts des communications. Les arêtes du graphe représentent les liens de communication et sont étiquetées avec les bandes passantes disponibles. Si $e \in E$ est un lien orienté de P_i à P_j, notons b_e la bande passante du lien. Nous aurons besoin de D_c/b_e unités de temps pour transférer un message de taille D_c de P_i à P_j en utilisant le lien e.

Routage. Supposons que nous pouvons décider comment les messages sont routés d'un processeur à un autre et que nous voulons router un message de taille D_c de P_i à P_j, en passant par k arêtes $e_1, e_2, \ldots, e_k$. Pour chaque arête e_m, le message aura une fraction f_m de la bande passante b_{e_m}. La vitesse globale de communication le long du chemin sera limitée par la plus petite bande passante disponible.

Paramètres de l'application : calculs. Soit D_w la taille totale du travail qui doit être accompli à chaque itération de l'algorithme. Le processeur P_i effectuera une quantité de travail $\alpha_i.D_w$ de ce travail, où $\alpha_i \geq 0$ pour $1 \leq i \leq p$ et $\sum_{i=1}^{p} \alpha_i = 1$.

Paramètres de l'application : communications. Les processeurs sont organisés le long d'un anneau (qui n'est pas encore déterminé). Après avoir mis à jour ses données de taille $\alpha_i D_w$, chaque processeur P_i envoie un message de longueur D_c fixée (typiquement, la taille des données frontières) à son successeur. Pour illustrer la relation entre D_w et D_c, nous pouvons voir la matrice de données comme un rectangle composé de D_w colonnes de hauteur D_c, ainsi une seule colonne est échangée entre paire de processeurs consécutifs dans l'anneau. Soit succ(i) et pred(i) le successeur et le prédécesseur de P_i dans l'anneau virtuel. Le temps nécessaire pour transférer un message de taille D_c de P_i à P_j est $D_c.c_{i,j}$ où $c_{i,j}$ est la capacité des liens de communication entre P_i à P_j. $c_{i,j}$ varie selon que l'on suppose que les liens sont partagés ou non. Nous avons proposé, dans un précédent travail [9], un algorithme qui construit un anneau optimal en respectant les hypothèses.

2.2. Les algorithmes de redistribution de données

Nous allons décrire dans cette section notre algorithme de redistribution dans le cas d'un anneau bidirectionnel hétérogène. (Le cas des anneaux homogènes est hors de propos dans cet article et a déjà été étudié dans [11].)

Nous ne connaissons pas d'algorithme optimal dans ce cas de figure. Cependant, si nous supposons que chaque processeur possède initialement plus de données (L_i) que ce qu'il doit envoyer pendant l'exécution de l'algorithme (ce que nous appelons une redistribution *légère*), alors nous réussissons à obtenir une solution optimale.

Dans toute cette section, nous supposons que nous sommes dans le cas de figure d'une redistribution *légère* : le nombre de données envoyées par n'importe quel processeur dans tout l'algorithme de redistribution est inférieur ou égal à sa charge originale. Nous appelons δ_i le *déséquilibre* de P_i. Il existe deux raisons pour qu'un processeur P_i envoie une donnée : (i) il est surchargé ($\delta_i > 0$) ; (ii) il propage des données à un autre processeur placé plus loin dans l'anneau. Si P_i possède au départ au moins autant de données qu'il devra en envoyer pendant l'exécution de l'algorithme, alors P_i peut envoyer immédiatement tout ce qu'il doit envoyer. Sinon, dans le cas général, certains processeurs doivent attendre de recevoir des données d'un voisin avant de pouvoir les retransmettre à un autre voisin.
Sous l'hypothèse de « redistribution légère », nous pouvons construire un programme linéaire en entiers pour résoudre notre problème (cf. système 1). Soit $\mathcal{S}$ une solution et $\mathcal{S}_{i,i+1}$ le nombre de données qu'un processeur P_i envoie au processeur P_{i+1}. De la même manière, $\mathcal{S}_{i,i-1}$ est le nombre de données que P_i envoie à P_{i-1}. Afin d'alléger l'écriture des équations, nous imposons dans les deux premières équations du système 1 que $\mathcal{S}_{i,i+1}$ et $\mathcal{S}_{i,i-1}$ sont positifs pour tout i, ce qui impose d'utiliser d'autres variables $\mathcal{S}_{i+1,i}$ et $\mathcal{S}_{i-1,i}$ pour les communications symétriques. La troisième équation établit qu'après redistribution, il n'y a plus de déséquilibre. Nous notons τ le temps d'exécution de la redistribution. Pour tout processeur P_i, du fait du modèle 1-port, τ doit être plus grand que le temps passé par P_i à envoyer des données (quatrième équation) ou passé par P_i à recevoir des données (cinquième équation). Notre but est de minimiser τ, d'où le système :

$$\text{Minimiser } \tau \text{ avec les contraintes suivantes}$$
$$\begin{cases} \mathcal{S}_{i,i+1} \geq 0 & 1 \leq i \leq n \\ \mathcal{S}_{i,i-1} \geq 0 & 1 \leq i \leq n \\ \mathcal{S}_{i,i+1} + \mathcal{S}_{i,i-1} - \mathcal{S}_{i+1,i} - \mathcal{S}_{i-1,i} = \delta_i & 1 \leq i \leq n \\ \mathcal{S}_{i,i+1} c_{i,i+1} + \mathcal{S}_{i,i-1} c_{i,i-1} \leq \tau & 1 \leq i \leq n \\ \mathcal{S}_{i+1,i} c_{i+1,i} + \mathcal{S}_{i-1,i} c_{i-1,i} \leq \tau & 1 \leq i \leq n \end{cases} \quad (1)$$

Nous utilisons le système 1 pour trouver une solution optimale au problème. Si, dans cette solution optimale, pour un processeur P_i donné, le nombre total d'éléments de données transmis est inférieur ou égal à la première charge ($\mathcal{S}_{i,i+1} + \mathcal{S}_{i,i-1} \leq L_i$), nous nous trouvons sous la « redistribution légère » et nous pouvons utiliser la solution du système 1. Une preuve de l'exactitude de ce qui a été montrée dans cette section est donnée dans [10].

3. La propagation de la chaleur

3.1. Description

Pour étudier le comportement de nos algorithmes, nous allons nous concentrer sur une simple application : la propagation de la chaleur. La propagation de la chaleur est une application parmi tant d'autres qui correspond à notre modèle. Nous l'avons choisie pour sa simplicité. Toutefois, les résultats présentés dans ce document restent valables pour des applications plus complexes (simulation de tourbillon, simulation de turbulence en aérodynamique, ...) à condition qu'elles respectent le modèle. Dans ce contexte, nous nous concentrerons sur des algorithmes d'équilibrage de charge et des algorithmes de redistribution de données.
Imaginez une plaque de métal (voir la figure 1) sur laquelle est appliquée une source de chaleur sur les bords. La chaleur se propage à l'intérieur de la plaque. La température sur les bords est maintenue constante, la distribution de la chaleur vers la plaque tend vers un état stable. Ce problème peut-être résolu en utilisant les méthodes de différences finies pour discrétiser la plaque à deux dimensions dans une grille à deux dimensions. Ensuite, une méthode itérative, comme Jacobi, est utilisée pour résoudre l'équation discrétisée. Pour des raisons de simplicité, nous ne considérerons que des communications associées à la métode en différences finies de Jacobi. Dans cette classe de méthodes numériques, une grille multidimensionnelle est mise à jour plusieurs fois par le remplacement de la valeur à chaque point à l'aide d'une opération sur les valeurs d'un nombre fixe de points voisins (voir la figure 3). Nous appelons l'ensemble des valeurs nécessaires à la mise à jour d'un point de la grille un stencil (voir figure 2).
Cette simple application correspond pleinement à notre modèle. En effet, vu que les communications sont dans un proche voisinage, si nous donnons à chaque processeur un bloc vertical de colonnes de la grille discrétisée, les communications entre les processeurs correspondront à notre anneau-hypothèse. Ainsi,

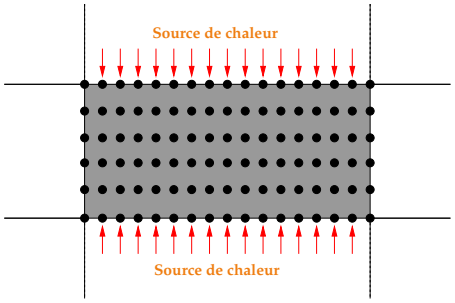


FIG. 1 – Exemple de propagation de la chaleur.

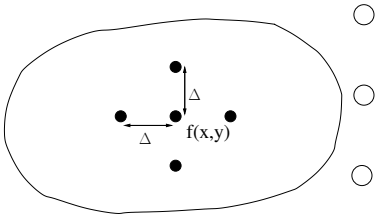


FIG. 2 – Un point de la grille mis à jour

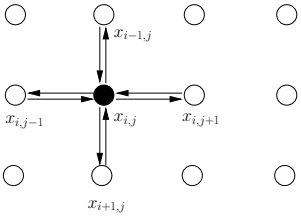


FIG. 3 – Schéma de communication

chaque processeur P_i communiquera seulement avec ses voisins P_{i-1} et P_{i+1} (voir Figure 3).
Nos algorithmes s'appliquent parfaitement à la discrétisation en deux dimensions où chaque processeur P_i a besoin d'échanger ses données avec ses voisins P_{i-1} et P_{i+1}.

4. De la théorie à la pratique

Ayant pour objectif d'évaluer expérimentalement l'impact de nos algorithmes de redistribution, nous avons conçu deux prototypes expérimentaux : le premier est basé sur le simulateur `SimGrid` [5] alors que le second est une implémentation réelle. L'idée derrière une telle approche est de comparer le comportement de l'application cible (i.e. la propagation de la chaleur) sur chacun des environnements.
`SimGrid` est un outil fournissant des fonctionnalités qui permettent de simuler l'exécution d'applications distribuées sur plates-formes hétérogènes. L'intêret principal de telles approches est de confronter les algorithmes à des plates-formes totalement contrôlable. De plus, le fait qu'une même expérience peut être reproductible représente un avantage certain. Ainsi nous avons utilisé cet outil pour mettre en œuvre une version simulée de notre application cible (nous entendons par simulée le fait que nous ne soyons intéressée que par les transferts de données et le temps passé dans les différentes étapes du calcul).
D'autre part, nous avons écrit un prototype illustrant l'intérêt des algorithmes de redistribution dans le contexte d'une application simple qui correspond totalement à notre modèle : la propagation de la chaleur. Le programme correspondant a été écrit en langage C en utilisant les *sockets* UNIX standards pour les communications ainsi que la couche `XDR` pour assurer l'intéropérabilité durant les communications entre machines hétérogènes. MPI n'est pas un bon candidat puisqu'il ne fonctionne principalement qu'avec un cluster tandis que les sockets et la couche XDR fonctionnent avec des multi-clusters ce qui correspond à nos plates-formes cibles. L'application est finalement régie par l'algorithme 1.

1: **while** fin non détectée **do**
2: échanger des données avec les voisins.
3: mise à jour des données locales et exécution de l'itération courante.
4: **if** *modulo* (numéro de l'itération courante, intervalle) == 0 **then**
5: effectuer des mesures pour découvrir les caractéristiques de la plate-forme.
6: faire, si nécessaire, une phase de redistribution.

Algorithme 1: Schéma itératif

Notre étude expérimentale suivra une approche dans laquelle nous tenterons (autant que possible) d'avoir les mêmes conditions expérimentales entre l'environnement réel et l'environnement simulé. Pour se faire, nous avons utilisé l'outil `wrekavoc` [4] pour contrôler l'hétérogénéité de la plate-forme cible. En effet, cet outil permet à l'utilisateur de dégrader à distance les performances du processeur et des liens réseau de

chacun des hôtes de la plate-forme. L'idée est donc dégrader périodiquement certaines caractéristiques de ressources de la plate-forme choisies aléatoirement et d'évaluer le comportement de nos algorithmes. Il est important de signaler qu'à chaque variation de plate-forme, une plate-forme simulée ayant les mêmes caractéristiques est générée. L'objectif est de comparer le comportement des algorithmes dans le cas d'une exécution réelle et dans le cas d'une exécution simulée. Chacune des variations appliquées à la plate-forme originale concerne un nombre aléatoire de ressources (processeurs ou liens réseau) et attribue à chacune d'elles un pourcentage aléatoire de sa capacité initiale (pour nos expérimentations, nous avons choisi des pourcentages dans l'intervalle $[40; 100]$). Il est à noter que la principale différence de comportement entre l'exécution réelle et l'exécution simulée est que dans le premier cas, nous laissons l'application découvrir le nouvel état de la plate-forme (par le biais de mesures), alors que dans le cas simulé nous fournissons au simulateur la plate-forme modifiée. Pour être plus précis, dans une exécution réelle, dès que l'application veut tenter de redistribuer des données, elle récupère les informations concernant l'état de la plate-forme de chaque nœud de la plate-forme (i.e. sa vitesse de processeur, sa latence réseau, sa bande passante entrante, ...). Dans ce cas, toutes les mesures sont faites dynamiquement en mesurant les caractéristiques de la plate-forme (c.f. ligne 5 de l'algorithme 1). Par opposition, pour une exécution simulée, même si les variations sont les mêmes que celles d'une exécution réelle, nous fournissons au simulateur l'état de la plate-forme simulée. Cette différence est motivée par le fait que d'une part nous ne voulons pas fournir à une exécution réelle une sorte d'oracle qui fournirait l'état de la plate-forme à notre algorithme et que d'autre part nous voulons conserver un schéma « simple » pour la version simulée.

5. Résultats expérimentaux

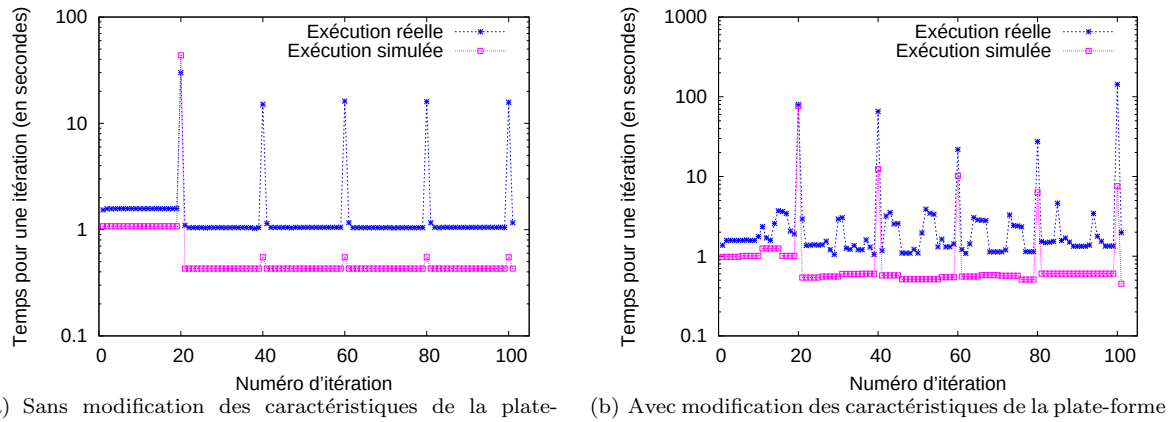
Dans cet article, nous basons notre modèle de plates-formes sur une plate-forme multi-cluster, `Grid'5000` [1] [1]. L'objectif du projet Grid'5000 est de construire une plate-forme hautement configurable, contrôlable pour l'évaluation d'algorithmes parallèles ou distribués. La plate-forme est composée de neuf sites géographiques distincts fournissant un total de 5000 unités de calcul. Chaque site héberge au moins un cluster dont la taille peut varier d'une centaine à un millier de processeurs. Le type des processeurs est soit de l'AMD Opteron, de l'intel Xeon, de l'intel Itanium 2, ou du PowerPC. Chaque cluster dispose d'un réseau d'interconnexion interne de type Gigabit (GigaEthernet, Myrinet, ou Infiniband). D'autre part, les sites sont reliés les uns aux autres par les réseaux RENATER dont la capacité peut atteindre 10Gb selon les sites. Pour nos expériences nous avons utilisé une plate-forme composée de 22 nœuds distribués sur 4 sites (5 à Lille, 6 à Nancy, 6 à Sophia et 5 à Toulouse). Il est à noter que nous n'avons pas réussi à réserver des plates-formes de taille plus importante pour une plus ou moins longue durée (pour effectuer toutes les mesures) à cause de la politique d'attribution des ressources de `Grid'5000`.

Nous avons fixé la taille totale de notre matrice rectangulaire à 40000 par 22000. D'autre part, nous n'avons pas réussi à utiliser des problèmes de plus grande taille vu que nous sommes limitée par la mémoire disponible sur chaque nœud. De plus, par défaut, les redistributions sont faites une fois toutes les 20 itérations. De plus, nous nous sommes limitée à 100 itérations même si pour autant le solveur n'a pas encore convergé. Ceci est motivé par le fait que le temps nécessaire à l'obtention de la convergence peut être plus ou moins important et par le fait que le comportement observé sur cent premières itérations est suffisant pour l'évaluation des algorithmes. Enfin, lorsqu'une phase de redistribution est effectuée, son temps est inclus dans celui de l'itération courante (la redistribution est faite à la fin d'une itération donnée).

Nous pouvons constater un écart presque constant entre le temps nécessaire pour une itération dans le cas réél et dans le cas simulé. Cet écart est principalement dû au côté intrusif du processus servant à modifier l'état de la plate-forme (i.e. surveiller un des processus participant à l'exécution pour savoir s'il faut effectuer une variation à un cout non-négligeable).

Nous donnons dans la figure 4, une comparaison du comportement de notre application aussi bien sur une plate-forme réelle que sur une plate-forme simulée. Nous pouvons observer tout d'abord que l'allure des deux courbes (exécution réelle et exécution simulée) est très proche. Nous pouvons voir à partir de la figure 4(a) que si nous ne modifions pas les caractéristiques de la plate-forme, les exécutions sont très similaires. De plus, nous pouvons observer que le temps nécessaire pour effectuer une itération est plus petit dans le cas d'une exécution simulée. Ceci n'est pas surprenant vu que le temps simulé représente

[1] http://www.grid5000.fr/



(a) Sans modification des caractéristiques de la plate-forme .

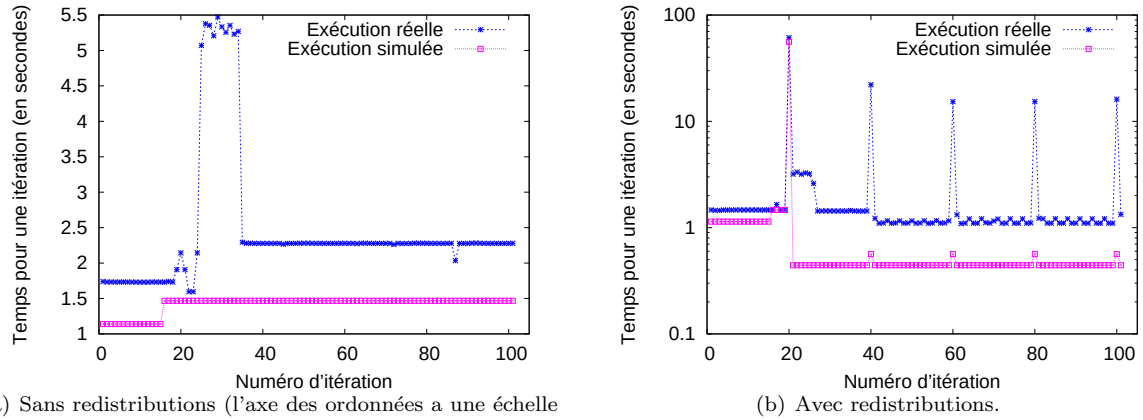
(b) Avec modification des caractéristiques de la plate-forme (1 modification toutes les 5 itérations) .

FIG. 4 – Temps nécessaire (en secondes) pour chaque itération pour une exécution réelle et une exécution simulée : Impact des modifications de la plate-forme.

un temps idéal qui n'inclut pas tous les détails d'implémentations de la version réelle. Une différence principale entre les deux exécutions dans ce contexte est le temps nécessaire à la redistribution (ces dernières sont faites une fois toutes les 20 itérations). Ceci est principalement dû au fait que le temps nécessaire à la redistribution inclut (dans le cas de l'exécution réelle) le temps nécessaire aux mesures de l'état de la plate-forme. Ces dernières sont en effet très coûteuses et nécessitent dans ce cas 20 secondes. Il est important de rappeler que ces mesures ne sont pas faites dans l'environnement simulé. Bien entendu, nous aurions pu fournir à l'application réelle les informations concernant l'état de la plate-forme mais nous pensons que découvrir l'état de la plate-forme est plus réaliste. Si nous considérons maintenant la figure 4(b), nous pouvons remarquer que même si l'allure des courbes est très proche, il y a plus de différences. Comme mentionné dans la légende de la figure, une modification de l'état de la plate-forme est faite toutes les 5 itérations. Nous pouvons voir que la modification de l'état de la plate-forme lors d'une exécution réelle peut être une opération intrusive qui peut perturber l'exécution pendant quelques itérations. Ce phénomène explique l'aspect irrégulier (après chaque redistribution) de la courbe correspondant à l'exécution réelle. La figure 4 illustre le fait que l'exécution simulée arrive à capturer les effets des choix algorithmiques dûs aux variations de la plate-forme tout en ignorant tous les détails techniques liés par exemple aux mesures des caractéristiques de la plate-forme. Cette dernière observation est très importante et reflète l'intérêt de l'utilisation du simulateur pour l'évaluation des algorithmes.

La figure 5 présente le comportement de notre application lorsqu'une seule modification de l'état de la plate-forme est effectuée à l'itération 15. Tout d'abord, dans le cas où nous n'utilisons pas notre mécanisme de redistribution, nous pouvons voir que l'aspect des deux courbes de la figure 5(a) est très proche sauf autour de l'itération 15. Ceci peut être expliqué par le phénomène décrit précédemment, à savoir que la modification de l'état de la plate-forme est une opération intrusive. De plus, nous pouvons constater que même si l'allure des deux courbes est très proche, les temps mesurés sont différents. Pour pallier ce problème, nous devrions mettre en œuvre une implémentation plus fine de la version simulée qui inclurait les mécanismes de mesure de l'état de la plate-forme. D'autre part, la figure 5(b) illustre l'intérêt de l'algorithme de redistribution. Nous pouvons constater que le temps nécessaire à une itération est réduit après redistribution (en comparant les figures 5(a) et 5(b)). Pour conclure, les résultats qui ont été présentés illustrent le fait que le simulateur est adapté à l'évaluation et à la comparaison d'algorithmes mais pas toujours adapté à la mesure de performance.

Pour une validation plus pointue, plutôt que de fournir nos plates-formes modifiées au simulateur, il serait intéressant de découvrir l'état de la plate-forme dans la version simulée de manière similaire à ce qui est fait dans une éxécution réelle.



(a) Sans redistributions (l'axe des ordonnées a une échelle linéaire).

(b) Avec redistributions.

FIG. 5 – Temps nécessaire (en secondes) pour chaque itération pour une exécution réelle et une exécution simulée : Impact des redistributions. Une seule modification de l'état de la plate-forme est effectuée à l'itération 15.

6. Travaux antérieurs

L'équilibrage de charge a été étudié de manière intense aussi bien pour des plates-formes homogènes (voir la collection d'articles [12]) que pour des plates-formes hétérogènes (voir le chapitre 25 de [3]). Distribuer le travail peut être effectué aussi bien en statique qu'en dynamique et même en mixant les deux.

Les algorithmes de redistribution ont fait l'objet d'intenses recherches. D'un point de vue théorique, dans le contexte de la compilation HPF [6], Kremer [7] a montré la NP-complétude d'un simple problème de redistribution. Ce résultat négatif illustre surtout le fait que des solutions optimales peuvent être conçues dans des cas particuliers tels que la plate-forme en anneau étudiée dans cet article.

En dernier lieu, nous mentionnons brièvement trois applications dont l'exécution peut directement tirer bénéfice des stratégies de redistribution conçues dans l'article. L'analyse des impulsions se propageant dans un milieu non-linéaire réclame des fenêtres de calcul, et la redistribution doit se produire fréquemment pendant que le calcul progresse [2]. Un procédé à deux niveaux de redistribution est préconisé dans [8] pour l'amélioration de maillage. Naturellement, cette courte liste pourrait être fortement allongée.

Nous avons présenté dans cet article une étude du comportement d'un algorithme de redistribution conçu pour équilibrer la charge de travail sur un ensemble de processeurs organisés en anneau virtuel. Pour se faire, nous avons mis en œuvre deux versions d'une même application : la propagation de la chaleur. La première implémentation a été faite au dessus du simulateur `SimGrid` et représente une mise en œuvre de haut niveau très adaptée à l'étude algorithmique. La seconde, quant à elle, est une implémentation réelle utilisant les *sockets* UNIX standards. Vu que nos algorithmes sont conçus pour gérer des plates-formes hétérogènes, nous avons utilisé l'outil `wrekavoc` pour contrôler de manière externe et dynamique les caractéristiques de la plate-forme. Les modifications apportées à la plate-forme sont stockées pour être par utilisée lors de l'exécution simulée. L'étude expérimentale a montré l'intérêt de notre algorithme de redistribution. De plus, nous avons pu mettre en évidence l'intérêt du simulateur. En effet, le comportement observé pendant une exécution simulée est très proche de celui d'une exécution réelle. Cependant, nous avons aussi observé qu'il existe dans certains cas quelques différences en terme de « performance » entre les deux approches. Pour comprendre les raisons de ces différences, il est nécessaire de concevoir plus finement la version simulée. Autant qu'on sache, cette étude est une des premières confrontant `SimGrid` à une exécution réelle dans le contexte d'une vraie application. Ce travail représente une première étape pour la validation à travers la comparaison avec des cas réels de `SimGrid` dans le contexte d'applications complexes.

7. Conclusion

Nous avons présenté dans cet article une étude du comportement d'un algorithme de redistribution conçu pour équilibrer la charge de travail sur un ensemble de processeurs organisés en anneau virtuel. Pour se faire, nous avons mis en œuvre deux versions d'une même application : la propagation de la chaleur. La première implémentation a été faite au dessus du simulateur `SimGrid` et représente une mise en œuvre de haut niveau très adaptée à l'étude algorithmique. La seconde, quant à elle, est une implémentation réelle utilisant les *sockets* UNIX standards. Vu que nos algorithmes sont conçus pour gérer des plates-formes hétérogènes, nous avons utilisé l'outil `wrekavoc` pour contrôler de manière externe et dynamique les caractéristiques de la plate-forme. Les modifications apportées à la plate-forme sont stockées pour être par la suite utilisées lors de l'exécution simulée. L'étude expérimentale a montré l'intérêt de notre algorithme de redistribution. De plus, nous avons pu mettre en évidence l'intérêt du simulateur. En effet, le comportement observé pendant une exécution simulée est très proche de celui d'une exécution réelle. Cependant, nous avons aussi observé qu'il existe dans certains cas quelques différences en terme de « performance » entre les deux approches. Pour comprendre les raisons de ces différences, il est nécessaire de concevoir plus finement la version simulée. Autant que nous sachions, cette étude est une des premières confrontant `SimGrid` à une exécution réelle dans le contexte d'une vraie application. Ce travail représente une première étape pour la validation à travers la comparaison avec des cas réels de `SimGrid` dans le contexte d'applications complexes.

Bibliographie

1. 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 T. Irena. Grid'5000 :a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20, 2006.
2. A. Bourgeade and B. Nkonga. Dynamic load balancing computation of pulses propagating in a nonlinear medium. *The Journal of Supercomputing*, 28(3) :279–294, 2004.
3. R. Buyya. *High Performance Cluster Computing. Volume* 1 *: Architecture and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.
4. L.-C. Canon and E. Jeannot. Wrekavoc : a tool for emulating heterogeneity. In *Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006.
5. H. Casanova, A. Legrand, and M. Quinson. SimGrid : a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, March 2008.
6. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.
7. U. Kremer. NP-Completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, 1993. Also available as Rice Technical Report CRPC-TR93330-S.
8. Z. Lan, V.E. Taylor, and G. Bryan. Dynamic load balancing of samr applications on distributed systems. In *Proceedings of the ACM/IEEE Symposium on Supercomputing (SC'01)*. IEEE Computer Society Press, 2001.
9. A. Legrand, H. Renard, Y. Robert, and F. Vivien. Mapping and load-balancing iterative computations. *EEE Transactions on Parallel and Distributed Systems*, 15(6) :546–558, 2004.
10. H. Renard, Y. Robert, and F. Vivien. Data redistribution algorithms for heterogeneous processor rings. Research Report RR-2004-28, LIP, ENS Lyon, France, May 2004. Also available as INRIA Research Report RR-5207.
11. H. Renard, Y. Robert, and F. Vivien. Data redistribution algorithms for homogeneous and heterogeneous processor rings. In *HiPC'04 11th International Conference On High Performance Computing*, Lecture Notes in Computer Science, 2004.
12. B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.