7e Conférence Francophone de MOdélisation et SIMulation - MOSIM'08 - du 31 mars au 2 avril 2008 - Paris - France
Modélisation, Optimisation et Simulation des Systèmes : Communication, Coopération et Coordination

ÉQUILIBRAGE DE CHARGE POUR LA GRILLE

M. PÉROTIN, P. MARTINEAU et C. ESSWEIN
Laboratoire d'Informatique de Tours
64 av J. Portalis 37200 Tours
{matthieu.perotin, patrick.martineau, carl.esswein}@univ-tours.fr

RÉSUMÉ : *La recherche en informatique et ses applications demandent une puissance de calcul croissante. Dans cet article, une formalisation du besoin est proposée afin de répondre à cette demande en supprimant les procédures de soumission de tâches contraignantes et peu conviviales. De plus, on souhaite utiliser les ordinateurs déjà en place dans les universités de la manière la plus conviviale possible pour les utilisateurs. Un modèle mathématique du problème est proposé afin d'apporter une solution au problème d'ordonnancement des processus. La complexité du problème est établie et diverses heuristiques sont proposées, puis implémentées au sein d'un simulateur réaliste.*

MOTS CLÉS : *équilibrage de charge, ordonnancement en ligne, Grid, calcul haute performance*

1. INTRODUCTION

1.1. Objectif

On répond généralement à la demande en puissance de calcul en faisant l'acquisition d'onéreux ordinateurs dédiés. Cette solution demande des experts pour l'administration de machines qui deviennent obsolètes rapidement. L'objectif est de produire une solution utilisant les centaines d'ordinateurs qui équipent déjà les universités, par exemple au sein de salles de travaux pratiques. Ces ordinateurs sont souvent récents, interconnectés par un réseau rapide et peu utilisés : ils ne sont que marginalement utilisés durant la nuit, et durant les enseignements nous avons mesuré une charge moyenne de 0,25.

Cet article propose une solution permettant d'utiliser cette puissance résiduelle en satisfaisant deux objectifs. Le premier est que les utilisateurs légitimes des ordinateurs, principalement les étudiants, ne doivent pas être pénalisés. En effet, on ne veut pas qu'un utilisateur voit son expérience se dégrader si sa machine croule subitement sous les tâches des autres utilisateurs. Le second objectif à atteindre est de produire un système utilisable facilement. Ainsi les utilisateurs ne doivent pas passer trop de temps à soumettre leurs tâches, et l'on peut préciser cet objectif en trois points :

1. La procédure de soumission de tâches parallèles doit être aisée
2. On ne peut pas demander à l'utilisateur d'estimer la durée de ses tâches
3. On ne veut pas imposer un langage de programmation ou une librairie particulière : l'utilisateur doit pouvoir utiliser le langage qu'il préfère.

1.2. État de l'art

Deux catégories de parallélisme sont généralement considérées :

1. Distribution des données
2. Distribution des processus

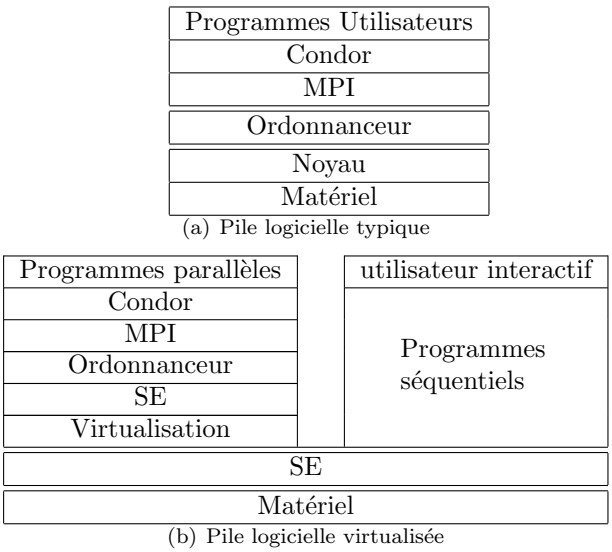
La première catégorie est fréquente. Elle repose sur l'hypothèse que chaque ordinateur du système fait tourner un processus en attente de données à traiter. Il est possible d'avoir un exécutable pour chaque architecture matérielle que l'on souhaite utiliser. Des projets comme SETI@Home (Anderson et al., 2002), et maintenant BOINC (Anderson, 2004) implémentent une telle méthode de parallélisme. Ces deux projets ont démontré toute leur pertinence. Malgré cela, ils sont limités à un nombre restreint d'applications où les tâches sont mutuellement indépendantes, et où les données peuvent être fragmentées en instances indépendantes.

La seconde catégorie contient des systèmes dans lesquels des processus sont migrés d'un ordinateur à un autre. Cette méthode repose sur l'hypothèse de la compatibilité binaire des programmes. Cela n'est pas toujours possible, un exécutable ne fonctionnant correctement que pour un couple (matériel, système d'exploitation) donné. Certains systèmes, tels Condor (*Condor*, 2008), résolvent ce problème en imposant la mise à disposition d'un exécutable spécifique pour chaque couple. ProActive (*ObjectWeb Consortium Proactive*, 2007) fonctionne au-dessus d'une machine virtuelle Java et résout ainsi le problème de façon plus élégante. Mais il impose l'utilisation d'un langage donné. Quant aux Single System Image (SSI), ils migrent des processus en cours d'exécution de façon transparente pour l'utilisateur,

et n'imposent ainsi ni langage ni bibliothèque particulière. OpenMosix (Hanquez et Bar, 2007) ou Kerrighed (*Kerrighed*, 2008) implémentent de tels systèmes. Pour que cette migration fonctionne, il faut un couple (matériel, système d'exploitation) fixé. Mais l'utilisateur a l'illusion d'être face à une machine unique multi-processeurs. Ainsi il n'a ni besoin d'utiliser des bibliothèques spécifiques, ni besoin de changer ses habitudes de programmation. La granularité de tels procédés est généralement le processus, parfois le thread. On se propose de déployer une telle solution en résolvant le problème de l'unicité du couple (matériel, système d'exploitation) par l'utilisation de la virtualisation.

La virtualisation est une technologie en plein essor qui permet de lancer un système d'exploitation dans un autre. Nous proposons d'utiliser la virtualisation afin de faire tourner notre environnement avec un coût d'administration réduit (*VMWare*, 2008).

Une pile logicielle typique est représentée sur la figure 1(a). La figure 1(b) montre une architecture virtualisée où un ordinateur fait tourner une machine virtuelle dans laquelle des applications parallèles peuvent être exécutées. Des processus séquentiels classiques peuvent être exécutés dans le système d'exploitation classique. Ceci lève la restriction sur le SE virtualisé. Déployer une machine virtuelle se fait à moindre coût, il s'agit d'un fichier à copier sur chaque ordinateur. Ainsi les problèmes d'administration et d'uniformité posés par le choix d'un SSI sont tous deux résolus.



(a) Pile logicielle typique

(b) Pile logicielle virtualisée

Figure 1. Piles logicielles

Comme les machines virtuelles peuvent être mises en réseau, nous pouvons les utiliser pour déployer un SSI au-dessus des systèmes existants. Le niveau d'abstraction offert par ces machines virtuelles résout le problème d'hétérogénéité du matériel. Les performances des outils de virtualisation vont en s'améliorant. Nous avons mesuré que ces performances sont acceptables. En effet le coût en terme de puissance est de l'ordre de 10% sur du matériel standard.

1.3. Contribution

On s'intéresse ici à un problème d'équilibrage de charge. La solution technique choisie est l'utilisation d'un SSI et d'une pile logicielle virtualisée. L'équilibrage de charge de processus a été largement étudiée dans la littérature ((Xu et Lau, 1997), (Berenbrink et al., 1999), (Rudolph et al., 1991), (Casavant et Kuhl, 1988)). Cependant, le modèle proposé ici est différent des modélisations existantes. Contrairement aux modélisations trouvées précédemment, le modèle est ici plus réaliste par l'ajout de contraintes liées à l'occupation mémoire, au modèle de réseau et à la métrique de charge. Les deux premiers points sont souvent négligés ((Berenbrink et al., 2001), (Rudolph et al., 1991)) mais ils sont pourtant d'une grande importance si l'on souhaite implémenter ces solutions grandeur nature. De plus, on fait l'hypothèse que l'on ne peut ni estimer les durées opératoires, ni prévoir la date d'arrivée et le lieu d'arrivée des processus. Les processus n'apparaissent pas dans une file globale que l'on vide lorsque les processeurs sont inactifs. De plus, on considère, comme c'est le cas sur un ordinateur classique, que l'exécution des processus se fait de façon préemptive.

Afin de comparer des heuristiques issues de la littérature, un simulateur a été écrit. Sa flexibilité le rend adaptable à de très nombreux cas de figure.

L'organisation de cet article est conforme à ces deux aspects. Dans une première partie le modèle est décrit, sa complexité est établie et des résultats théoriques sont donnés. En particulier on montrera qu'une approche jusque lors utilisée dans OpenMosix produit des résultats non garantis. Une deuxième partie décrit le simulateur. La génération d'instances est décrite avec précision, et des résultats expérimentaux sont donnés pour une heuristique classique.

2. MODÉLISATION

2.1. Modèle Mathématique

Un modèle d'étude est ici proposé afin de répondre à la question d'ordonnancement suivante : comment choisir quel processus migrer vers quelle machine ?

Soit M l'ensemble des ordinateurs de l'université. Soit $m = |M|$ le nombre total d'ordinateurs, et m_j l'ordinateur j, pour $j = 1..m$.

Chaque ordinateur m_j est caractérisé par sa quantité de mémoire vive totale ($mem(m_j)$), la vitesse de son processeur ($speed(m_j)$) et son état ($state(m_j)$). Celui-ci peut être *libre* ou *occupé* dépendamment de la présence d'un utilisateur physique travaillant sur la machine à un instant donné.

$$state(m_j) = \begin{cases} 0 & si\ m_j\ est\ libre \\ 1 & sinon \end{cases}$$

Le réseau est modélisé par une matrice carrée N de taille m, où $N_{i,j}$ est la vitesse du réseau entre les ordinateurs m_i et m_j. On considère que $\forall i, N_{i,i} = 0$. Cette modélisation est une simplification de l'architecture réelle : on ne considère que la connexion des ordinateurs les uns avec les autres, sans se préoccuper de l'enchaînement de switchs et de routeurs.

Soit $P = \{p_i\}_{i=1..p}$ l'ensemble des processus à l'état activable sur M.

$\forall i = 1..p, \forall j = 1..m$, let

$$P_{ij} = \begin{cases} 1 & si\ p_i\ s'ex\acute{e}cute\ sur\ m_j \\ 0 & sinon \end{cases}$$

Soit $mu(p_i)$ la mémoire utilisée par p_i

La première fonction objectif est l'équilibrage de charge : plus la charge sera équilibrée, plus vite les processus se finiront.

Traditionnellement, la charge d'un ordinateur donné est mesurée comme étant le nombre de processus activables à l'instant t. Comme cette valeur varie fréquemment, la moyenne sur les n précédentes minutes lui est généralement préférée. Seuls les processus à l'état activable sont considérés. Les processus interactifs sont négligés car ils passent le plus clair de leur temps à attendre des saisies de l'utilisateur, contribuant ainsi marginalement à la charge. D'où les définitions :

La charge instantanée d'un ordinateur m_j, $InstantLoad_j$ est définie par :

$$InstantLoad_j = \sum_{i=1}^{p} P_{ij}$$

Cette définition est inappropriée pour comparer la charge d'ordinateurs de puissances variables. Ainsi une autre métrique est proposée. Elle dépend de la vitesse des ordinateurs et de leur état.

Soit

$$load_j = (1 + K \times state(m_j)) \frac{InstantLoad_j}{speed(m_j)}$$

$$load_j = (1 + K \times state(m_j)) \frac{\sum_{i=1}^{p} P_{ij}}{speed(m_j)}$$

la charge de l'ordinateur m_j, où K est une constante entière positive qui permet de spécifier la charge maximale acceptable pour un ordinateur sur lequel un utilisateur est connecté.

Une seconde fonction objectif est le temps de communication réseau qui est à minimiser. Les migrations de processus peuvent être longues et empêcher le gain de performance ((Eager et al., 1988) and (Downey et Harchol-Balter, 1995)). Ainsi les migrations doivent être évitées, mais elles sont nécessaires pour équilibrer la charge : c'est un problème bi-critères.

2.2. Résolution du problème en-ligne

Le problème auquel on s'intéresse est en-ligne (Borodin et El-Yaniv, 1998), car plusieurs paramètres sont acquis en ligne.

- L'état des machines : à un instant donné il peut changer
- Les processus : ils peuvent apparaître sur n'importe quel ordinateur
- Les durées opératoires : ne pouvant être estimées, les processus peuvent disparaître à chaque instant.

Travailler en ligne serait pertinent s'il était possible de prendre des décisions d'ordonnancement quand les processus apparaissent ou disparaissent, ou quand les ordinateurs changent d'état. Il est impossible d'estimer les durées opératoires au moment de l'apparition d'un processus. Ainsi on ne peut savoir s'il restera actif suffisamment longtemps pour contribuer à la charge de façon significative, et bénéficier d'une migration.

On se propose donc de laisser la situation se déséquilibrer et de la rééquilibrer à intervalles réguliers. À un instant donné on considère l'état global du système

et on fait l'hypothèse qu'il n'évoluera plus. Des décisions d'ordonnancement sont alors prises en conséquence, en résolvant un problème d'ordonnancement hors ligne. À l'instant de décision suivant, soit l'hypothèse s'est avérée exacte, auquel cas il n'y a rien à faire, soit elle s'est avérée fausse et on réitère le processus. Ainsi on se retrouve face à la résolution d'une suite de problèmes hors ligne plutôt qu'à la résolution d'un problème en ligne.

Ce modèle est très proche du problème de rééquilibrage de charge défini par (Aggarwal et al., 2003) :

Définition 1 *Le problème de rééquilibrage de charge (Aggarwal et al., 2003) : Étant donnée une répartition de n tâches sur m processeurs, ainsi qu'un entier positif k, déplacer au plus k tâches pour minimiser la charge maximale sur un processeur. Plus généralement, soit c_i le coût de migration de la tâche i, on souhaite déplacer les tâches en ne dépensant pas plus qu'un budget B donné.*

Le modèle d'Aggarwal ne considère que des processeurs uniformes, et la charge est définie comme étant la somme de la *taille* des processus. Dans le modèle proposé dans cet article, les processus ont une taille constante au regard de la fonction de charge, conformément à la mesure de charge classique d'UNIX.

On peut modéliser le problème sous la forme d'un programme par contrainte.

Variables : $\forall i = 1..p, \forall j = 1..m$, soit

$$X_{ij} = \begin{cases} 1 & if\ p_i\ est\ migré\ vers\ m_j \\ 0 & sinon \end{cases}$$

Un processus ne peut s'exécuter que sur une machine à la fois :

$$\forall i = 1..p, \forall t \sum_{j=1}^{m} X_{ij} = 1$$

Une machine ne peut allouer plus de mémoire que ce dont elle dispose :

$$\forall j = 1..m, \sum_{i=1}^{p} X_{ij} \times mu(p_i) \leq mem(m_j)$$

Équilibrer la charge :

$$min \max_{j=1..m} load_j = (1 + K \times state(m_j)) \frac{\sum_{i=1}^{p} X_{ij}}{speed(m_j)}$$

Minimiser le coût des migrations :

$$min \sum_{i=1}^{p} N_{k,l} \times P_{ik} \times X_{il} \times mu(p_i)$$

Minimiser deux fonctions objectif à la fois est généralement difficile, et plusieurs approches sont possibles. L'approche ϵ-contrainte (T'kindt et Billaut, 2002) est appropriée, et vient naturellement comme la quantité de processus que l'on peut migrer entre deux instants de décision.

Ainsi la seconde fonction objectif devient

$$\sum_{i=1}^{p} N_{k,l} \times P_{ik} \times X_{il} \times mu(p_i) \leq Const$$

Seul l'objectif d'équilibrage de charge demeure.

2.3. Complexité

Étant donnée, la métrique de charge, chaque processus contribue à la fonction objectif à hauteur de :

$$\frac{(1 + K \times state(m_j))}{speed(m_j)}$$

En négligeant les contraintes de coût de migration et de mémoire, on peut réduire le problème au Multiprocessor Scheduling Problem (MPSP) où chaque processeur m_j a pour vitesse $\frac{speed(m_j)}{1+K}$ si $state(j) = 1$ et $speed(m_j)$ sinon. Les temps de traitement sont tous égaux à une constante donnée. Ainsi minimiser le makespan minimise la charge maximale de notre problème. Bien que le MPSP soit connu pour être NP-Complet (Garey et Johnson, 1979), le cas particulier où toutes les durées opératoires sont identiques est connu pour être polynomial. Ainsi, une telle relaxation du problème permet d'obtenir une borne inférieure notée OPT_L.

On s'intéresse maintenant au problème de décision consistant à savoir s'il existe une solution réalisable telle que $\max_j \{load_j\} = OPT_L$ au problème non relaxé. On montre que ce problème est NP-Complet par réduction au problème de 3-Partition.

3-Partition

Données : $A = \{a_1, \ldots, a_{3n}\}$ $s : A \to \chi$ tq. $s(A) = nB$ et $\forall i \in \{1 \ldots n\}$ $\frac{B}{4} \leq s(a_i) < \frac{B}{2}$

Question : Peut-on réaliser une partition de A en n classes de même poids B ?

Si la réponse est oui, chaque classe contient exactement trois éléments de A et est de taille B.

Load Balancing

Données : $M = \{m_1, \ldots, m_m\}$ un ensemble de machines dotées d'une mémoire $memoire(m_i)$. $T = \{T_1, \ldots, T_n\}$ un ensemble de processus, d'occupation mémoire $mem(T_i)$. On note $load_j$ la charge de la machine j, c'est-à-dire le nombre de processus qui lui sont affectés.

Question : Peut-on réaliser une affectation réalisable des n processus sur les m machines telle que le $max_j load_j - min_j load_j \leq 1$?

Énoncé de 3-Partition $I = (A, S)$ où :
- $Card(A) = 3n$
- $s(A) = nB$
- $\forall a \in A \ \frac{B}{4} \leq s(a) < \frac{B}{2}$

Énoncé f(I) de Load Balancing
- $3n + 3$ tâches T_i
- $n + 1$ machines m_j
- $mem(T_i) = s(a_i) \forall i > 3$, $r_i = 0$
- $mem(T_i) = B + 1 \forall i \leq 3$, $r_i = 0$
- $memoire(m_1) = HV$
- $memoire(m_j) = B \ \forall i = 4, \ldots 3n + 3$
- $mem(T_i) > memoire(m_j) \forall i = 1 \ldots 3, j = 2 \ldots n + 1$
- $mem(T_i) < memoire(m_j) \forall i = 4 \ldots 3n + 3, \forall j$

Formulation hors ligne de notre problème. On suppose que toutes les tâches T_i sont sur la machine 1 à $t = 0$.

Les trois premières tâches qui nécessitent beaucoup de mémoire restent sur la machine 1.

Le but est de répartir les $3n$ tâches sur les n autres machines.

$\Longrightarrow$ Supposons que l'énoncé (A, S) a pour réponse oui.

Alors soient $C_1, \ldots, C_n$ les sous-ensembles de trois tâches T_i associés aux n classes de A de poids B.

L'ordonnancement f(I) suivant est réalisable :

Chaque sous ensemble C_i contiendra exactement trois tâches T_i dont la somme des encombrements mémoire fera exactement B.

$\Longleftarrow$ Réciproquement, supposons qu'il existe un ordonnancement de l'énoncé f(I) de LoadBalancing. Les trois premières tâches sont bloquées sur la machine 1 car c'est la seule à avoir une mémoire suffisante pour les accueillir.

Les n machines qui restent sont libres et peuvent recevoir les 3n tâches de manière à équilibrer la charge, c'est-à-dire avoir exactement trois tâches par machine. On peut donc partitionner A en n classes de poids B.

Conclusion f est une réduction de 3-partition à LoadBalancing.

LoadBalancing est NP-Complet.

2.4. Algorithmes et taux de compétitivité

Un algorithme trivial, dont l'idée n'est pas très éloignée de celle de (Rudolph et al., 1991) est appelé SIMPLE. Il est par exemple utilisé par OpenMosix. Cet algorithme est complètement distribué et un agent tourne sur chaque ordinateur, prenant des décisions de migration pour les processus s'y trouvant. Il peut être écrit ainsi : tant qu'il existe un ordinateur moins chargé que moi, lui envoyer un processus (quelle que soit la façon de choisir un processus).

On peut montrer que cet algorithme n'est pas un approximateur du problème en considérant l'instance suivante :
- deux ordinateurs m_1 et m_2 avec
  - $speed(m_1) = k \times speed(m_2)$
  - $mem(m_2) \geq 2 \times mem(m_1)$
- $k + 1$ processus, 1 de taille $mem(m_1)$ et k de taille $\frac{mem(m_1)}{k}$
- Situation initiale :
  - le processus 1 est sur m_1
  - les autres sur m_2

Ici, la charge est k, et l'algorithme SIMPLE ne peut pas migrer de processus à cause de la contrainte de mémoire. La charge optimale est 1 et s'obtient en migrant le processus de m_1 vers m_2 et en migrant ensuite tous les processus de m_2 vers m_1. Le ratio de SIMPLE est k. Comme k peut être choisi arbitrairement grand, ce ratio ne peut être borné. SIMPLE n'est pas un approximateur.

3. SIMULATION

Afin de comparer différentes heuristiques de résolution du problème LoadBalancing, des tests ont été réalisés dans le simulateur SimGrid.

3.1. Génération d'Instances

Trois types de données sont à considérer afin de réaliser la simulation : la plateforme matérielle, les processus et les utilisateurs.

Plateforme Matérielle La génération stochastique de plateformes matérielles semble hors de propos ici, bien que rendue possible par l'utilisation d'outils tels Brite (Medina et al., 2001-003). En effet, on s'intéresse à l'architecture d'un réseau universitaire, souvent hierarchisé par une cascade de switchs. On considère la salle de TP au sein de laquelle les ordinateurs sont uniformes. Ils sont caractérisés par une taille mémoire et par la vitesse de leur processeur. Ils sont directement reliés à un switch. On considère aussi que les switchs sont reliés directement entre eux par un unique switch.

Le seul élément stochastique considéré sur les machines est le passage de l'état occupé à l'état libre ($state(m_j)$). La probabilité d'un changement d'état dépend de l'apparition d'utilisateurs sur le réseau de l'université, et du type d'utilisateur.

Processus Les processus sont modélisés comme ayant deux paramètres : une durée et une occupation mémoire. Par souci de simplification, l'occupation mémoire est considérée comme étant constante. On caractérise les processus plus en détail comme pouvant être courts (temps d'exécution de l'ordre de la seconde) ou longs (plusieurs dizaines de minutes), petits (quelques Mo) ou gros (plusieurs dizaines de Mo). On tire ces caractéristiques en suivant une loi normale de moyenne 1s et d'écart type 0,1 pour les processus courts, une moyenne de 10 minutes et un écart type de 2 minutes pour les processus longs. Pour les mémoires les lois normales respectives sont les suivantes : 1Mo de moyenne et 0,1 Mo d'écart type, 20 Mo de Moyenne et 2 Mo d'écart type. L'exécution des processus se fait de façon préemptive, comme c'est le cas sur un système réel.

Les Utilisateurs Le paramètre essentiel générateur d'évènements est l'utilisateur. On peut en considérer deux types :

– Utilisateur classique (type A) : il s'agit d'un utilisateur unique se connectant sur une machine afin d'effectuer un travail ne demandant pas de calcul intensif. Il créera des processus courts avec une probabilité de 0,9 et des processus longs avec une probabilité de 0,1. Cet utilisateur utilise une machine pour une durée suivant une loi normale de moyenne 1h30 et d'écart type 30 minutes.
– Utilisateur intensif (type B) : il s'agit d'un utilisateur demandeur de puissance de calcul. Il lance plusieurs dizaines de processus longs avec une probabilité de 0,6. Cet utilisateur utilise les ressources pour une durée suivant une loi normale de moyenne 6 heures et d'écart type 2 heures.

Pour pouvoir créer un processus sur une machine, il faut que la machine dispose d'une mémoire libre au moins égale à la taille en mémoire du processus. Si ce n'est pas le cas, le processus est refusé, comme c'est le cas dans un système d'exploitation classique. On ne considère ici que la mémoire vive et on néglige la mémoire dite *swap*. En effet, on formule l'hypothèse que son utilisation est contradictoire avec le calcul haute performance.

3.2. Algorithmes

Trois méthodes d'ordonnancement ont été considérées et implémentées dans le simulateur.

Simple+ L'algorithme SIMPLE+ ici proposé est une version améliorée de SIMPLE. Il est proposé afin d'éviter les problèmes d'oscillation que SIMPLE peut connaître : une tâche migrée vers une machine n'ayant pas assez de mémoire vive pour l'accueillir, est alors migrée en retour sur la machine initiale. Afin de tenter d'éviter ces situations, on propose l'algorithme suivant :

Algorithm 1 Simple+

```
1: if Je suis la machine la plus chargée then
2:   if La machine la moins chargée a suffisamment
     de mémoire pour accueillir mon plus petit pro-
     cessus then
3:     envoyer mon plus petit processus à la ma-
       chine la moins chargée
4:   else
5:     if Il existe une machine à même de recevoir
       mon plus petit processus then
6:       Le lui envoyer
7:     end if
8:   end if
9: end if
```

Random On considère un algorithme aléatoire, RANDOM, dont l'écriture est référencée par la figure : Algorithme 2.

Algorithm 2 Random

1: **if** Je suis la machine la plus chargée **then**
2: Envoyer un processus tiré au hasard à une machine tirée au hasard
3: **end if**

Null Enfin on se comparera au résultat de la simulation dans le cas où l'on n'effectue pas de rééquilibrage de charge.

3.3. Utilisation de SimGrid

Simgrid ((Casanova et al., 2003), (*SimGrid*, 2008)) est une bibliothèque permettant la simulation d'ordonnancements distibués sur plateforme hétérogène. Cette bibliothèque permet une simulation réaliste des topologies de grilles de calcul ainsi que des communications réseau, tout en fournissant divers degrés d'abstraction pour l'utilisateur final.

Le module MSG de Simgrid 3.2 a été utilisé ici pour réaliser une implémentation du modèle. Ce module fournit un niveau d'abstraction élevé en fournissant des primitives de communication entre agents, d'exécution de tâches ou encore de gestion de machines.

Dans l'implémentation proposée, un agent tourne sur chaque machine. Cet agent suit une boucle dont les étapes sont les suivantes :

1. Envoyer la charge de la machine et la mémoire libre dont elle dispose aux autres agents
2. Traiter les communications en attente
3. Exécuter les tâches de la file de Processus
4. Gérer l'utilisateur de la machine (apparition, disparition)
5. Gérer l'apparition de nouveaux processus
6. Prendre des décisions de migration

MSG souffre d'un certain nombre de limitations qu'il a fallu contourner afin de produire des résultats. Tout d'abord, le modèle de communication est synchrone. Il n'est possible de faire des communications asycnhrones qu'au prix de contorsions qui alourdissent la formalisation du modèle. L'utilisation de telles communications impose une certaine synchronisation des agents (les appels aux primitives d'envoi et de réception de tâches étant bloquants). Cela éloigne les résultats de ce que l'on pourrait obtenir sur une plateforme réelle. Cette limitation affecte les étapes 2 et 6 : par exemple un agent sur la machine B décidant d'envoyer un processus à une machine A ne passera pas à l'étape suivante tant que cette machine n'aura pas reçu le processus envoyé. Si la machine A, dans l'étape 1, envoie sa charge à la machine B au même moment, il y a interblocage. Des primitives d'envoi avec *timeout* existent. Elles ne règlent pas le problème de fond mais elles permettent d'éviter ces situations.

Une deuxième limitation de MSG est le modèle non préemptif d'exécution des tâches : si plusieurs tâches sont affectées à une machine, il faut les exécuter séquentiellement. Cela n'est pas conforme au modèle proposé dans cet article, et une solution a été trouvée en implémentant dans la simulation un algorithme d'ordonnancement préemptif sur chaque machine. Ainsi, les tâches arrivant sur une machine sont placées dans une file de processus locale. À chaque fois que l'ordonnanceur local est appelé, un morceau de quelques unes d'entre elles est exécuté.

Troisième limitation de MSG : l'absence de gestion de la mémoire des machines. Cette contrainte est complètement négligée par SimGrid, aussi faut-il la gérer dans son implémentation. Cependant, il est à noter que SimGrid est suffisamment flexible pour que l'ajout de données supplémentaires à une tâche soit aisé.

3.4. Résultats

L'instance de test est composée de six ordinateurs inter connectés par l'intermédiaire d'un switch. Quatre sont de puissance égale et disposent de 512 Mo de mémoire vive, deux sont deux fois plus rapides et disposent de deux fois plus de mémoire.

La simulation s'étend sur une période de 24 heures au cours desquelles des utilisateurs se connectent sur les ordinateurs. Les créations de processus ont lieu exclusivement sur les ordinateurs les plus rapides et sur l'un des autres. Les trois ordinateurs n'ont pas d'utilisateurs sur la période donnée.

On considère deux cas d'utilisation. Dans le cas A, les utilisateurs sont essentiellement de type A, et peu de processus longs sont créés, la mémoire n'est donc jamais saturée par les processus. Dans le Cas B, il y a plus d'utilisateurs de type B et donc plus de processus longs : la mémoire est une ressource critique.

Pour comparer les résultats donnés par les différents algorithmes on utilise deux indicateurs. Le premier est une traduction directe de la fonction objectif du problème, c'est-à-dire la charge maximale. Afin de pouvoir comparer des instances différentes, on utilise l'évolution au cours du temps de la fonction :

$$\delta(t) = \frac{\max_{j=1..m} load_{j,t} - \min_{j=1..m} load_{j,t}}{\max_{j=1..m} load_{j,t}}$$

Le deuxième indicateur est le nombre de tâches créées

par les utilisateurs ainsi que le nombre de tâches qui ont pû être terminées par le système sur l'intervalle de temps de 24 heures. En effet, on fait l'hypothèse que si la charge est correctement équilibrée alors le système travaille plus efficacement et est à même de réaliser plus de tâches dans le même intervalle de temps.

Étant donné le caractère stochastique de la simulation, les résultats donnés sont des moyennes sur cinq lancements.

Algorithme	#cr (m)	#cr (e.t)
SIMPLE+	382	8,09
RANDOM	363,5	6,66
NULL	251,6	13,13

Figure 2. Cas A : Nombre de créations de tâches (moyennes et écarts types)

Algorithme	#cons (m)	# cons (e.t.)
SIMPLE+	348	7,04
RANDOM	262,6	9,07
NULL	209,6	13,46

Figure 3. Cas A : Nombre de tâches menées à terme (moyennes et écarts types)

Algorithme	$\delta(t)$ (m)	$\delta(t)$ (e.t)
SIMPLE+	0,52	0,07
RANDOM	0,91	0,02
NULL	0,89	0,02

Figure 4. Cas A : Indicateur $\delta(t)$ (moyennes et écarts types)

Algorithme	#cr (m)	#cr (e.t)
SIMPLE+	360	13,73
RANDOM	255	25,17
NULL	194,6	24,13

Figure 5. Cas B : Nombre de créations de tâches (moyennes et écarts types)

Algorithme	#cons (m)	# cons (e.t.)
SIMPLE+	260,4	13,96
RANDOM	144,4	24,52
NULL	115,6	27,19

Figure 6. Cas B : Nombre de tâches menées à terme (moyennes et écarts types)

3.5 Analyse

Algorithme	$\delta(t)$ (m)	$\delta(t)$ (e.t)
SIMPLE+	0,3	0,03
RANDOM	0,88	0,05
NULL	0,89	0,04

Figure 7. Cas B : Indicateur $\delta(t)$ (moyennes et écarts types)

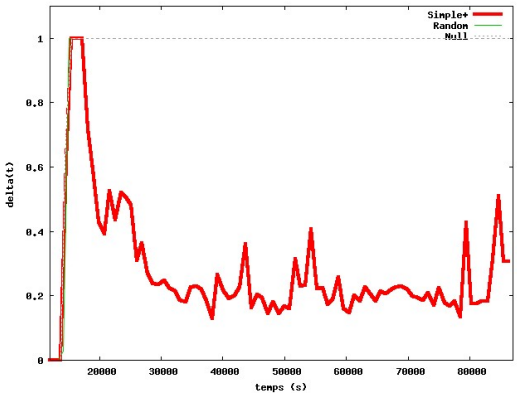


Figure 8. Évolution du critère delta

Les résultats obtenus confirment que le modèle proposé est pertinent. On remarque, d'une part, que l'utilisation d'une règle simple et irréfléchie (RANDOM) produit des résultats meilleurs que ceux obtenus en ne faisant rien. Cela se caractérise par le fait que plus de tâches sont menées à terme. D'autre part, on constate que l'utilisation d'une règle simple et un peu plus réfléchie (SIMPLE+) produit des résultats meilleurs que ceux de RANDOM.

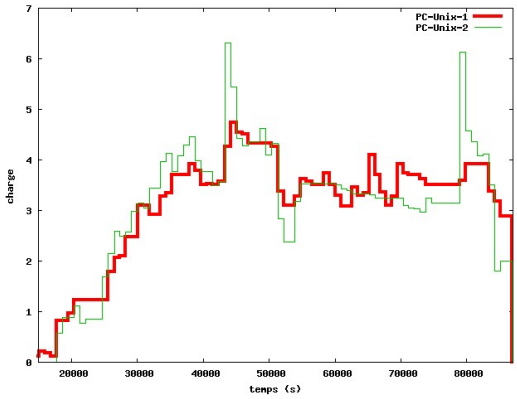


Figure 9. Évolution de la charge sur deux machines (SIMPLE+)

La figure 9 montre l'évolution de la charge sur deux machines du système. Des processus apparaissent sur la machine PC-Unix-2, et la charge de la machine PC-

Unix-1 (en gras sur la figure) n'augmente que grace à des migrations venant des autres ordinateurs. En effet aucun processus n'y apparait. On constate sur cet exemple que les charges de ces deux machines varient conjointement : la politique de migration de SIMPLE+ fonctionne. À titre de comparaison, les figures 10 et 11 montrent l'évolution de la charge de ces deux machines avec les politiques d'ordonnancement RANDOM et NULL.

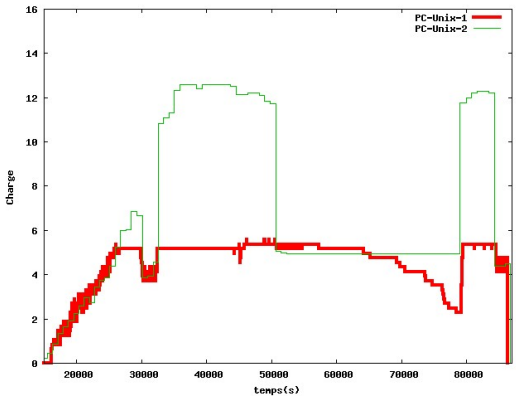


Figure 10. Évolution de la charge sur deux machines (RANDOM)

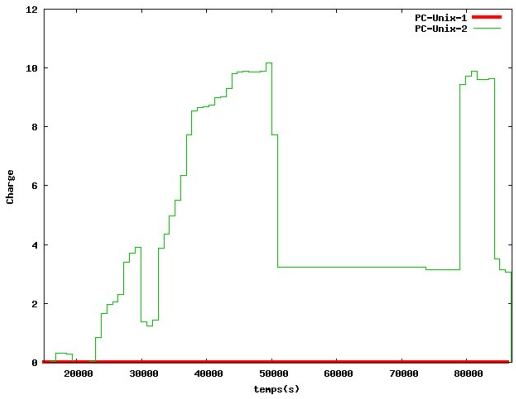


Figure 11. Évolution de la charge sur deux machines (NULL)

Par ailleurs, on remarque une très nette différence de résultat dépendamment du type d'instance. Dans une instance de type calcul intensif, l'écart se creuse considérablement entre SIMPLE+ et les autres algorithmes. Cela est en grande partie dû à la mémoire des machines sur lesquelles les processus sont lancés. Elle devient une ressource critique, d'où la faible performance de NULL.

Enfin, on remarque que l'indicateur $\delta(t)$ est plus faible dans le cas B que dans le cas A pour l'algorithme SIMPLE+. Cela s'explique par le fait que les tâches courtes bénéficient moins d'une migration que des tâches longues : le temps d'exécution d'une tâche longue domine généralement le temps de migration. De plus, de nombreuses tâches courtes font varier frequemment la charge d'une machine, ce qui peut entraîner des choix malheureux. Un agent verra sa machine surchargée à l'instant t et prendra des décisions de migration, puis les tâches restantes se finiront rapidement, entraînant une baisse de sa charge...La situation avec des tâches longues est plus stable.

4. CONCLUSION

Un modèle d'étude réaliste a été proposé. Un mécanisme d'équilibrage de charge distribué et en-ligne a été exposé puis implémenté au sein du simulateur SimGrid. Pour ce faire, l'ensemble du processus a été décomposé en trois sous-ensembles : les ressources, les utilisateurs et leurs tâches.

Des hypothèses fortes ont été effectuées : l'impossibilité d'estimer les durées opératoires, la prise en compte des contraintes de mémoire et l'exécution parallèle des tâches sur un processeur donné. Dans ce cadre, l'apport de règles d'ordonnancement heuristiques simples a été prouvé.

Les pistes d'études sont nombreuses : outre l'apport de nouvelles heuristiques, il convient maintenant de fixer un certain nombre de paramètres de façon expérimentale. En particulier, la variable K de la fonction objectif a une incidence directe sur le nombre de processus s'exécutant sur un ordinateur où un utilisateur est connecté.

Par ailleurs, le modèle peut être étendu au cas où la mémoire des processus varie au cours du temps. Une autre relaxation d'un cas pourtant bien réel a été faite et mérite d'être étudiée : l'hypothèse qu'un processus s'exécute aussi rapidement sur l'ordinateur sur lequel il a été créé que sur un autre ordinateur du réseau (à la vitesse du processeur près). De nombreux cas réels peuvent mettre en défaut cette hypothèse, en particulier dans le traitement des appels systèmes. Nous travaillons actuellement sur la formalisation et la modélisation des appels systèmes émis par un processus afin de les inclure à notre simulation. Parallèlement, le déploiement de la solution technique est en cours de planification.

REMERCIEMENTS

Merci à Erik SAULE (LIG-Moais, Grenoble) pour l'idée concernant la preuve de complexité et son aide quant à sa rédaction.

RÉFÉRENCES

Aggarwal, G., Motwani, R. et Zhu, A., 2003. The load rebalancing problem, *Proc. ACM SPAA, 2003* .

Anderson, D., 2004. Boinc : A system for public-ressource computing and storage, *5th IEEE/ACM International Workshop on Grid Computing* .

Anderson, D., Cob, J., Korpela, E., Lebofsky et Werthimer , 2002. Seti at home : An experiment in public ressource computing, *Communications of the ACM* **45 No. 11** : 56–61.

Bellard, F., 2005. QEMU CPU Emulator and virtualizer, *QEMU Website http ://fabrice.bellard.free.fr/qemu/* .

Berenbrink, P. et al., 1999. Randomized and adversarial load balancing, *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures* pp. 175 – 184.

Berenbrink, P., Friedetzky, T. et Goldberg, L. A., 2001. The natural work-stealing algorithm is stable, *IEEE Symposium on Foundations of Computer Science*, pp. 178–187.

Borodin, A. et El-Yaniv, R., 1998. *Online Computation and Competitive Analysis*, Cambridge University Press.

Casanova, H., Legrand, A. et Marchal, L., 2003. Scheduling distributed applications : the simgrid simulation framework, *Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)* .

Casavant, T. et Kuhl, J., 1988. A taxonomy of scheduling in general-purpose distributed computing systems, *IEEE Transactions on Software Engineering* pp. 141–154.

Choco constraint solver, 2005. http ://choco.sourceforge.net/.

Condor, 2008. http ://www.cs.wisc.edu/condor/.

Downey, A. B. et Harchol-Balter, M., 1995. A note on "the limited peroformance benefits of migrating active processes for load sharing", *Technical report*, Computer Science Division University of California Berkeley.

Eager, D. L., Lazowska, E. D. et Zahorjan, J., 1988. The limited performance benefits of migrating active processes for load sharing, *SIGMETRICS* pp. 662–675.

Garey, M. et Johnson, D. S., 1979. *Computer and Intractability, a guide to the theory of NP-Completeness*, WH Freeman and Compagny.

Hanquez, V. et Bar, M., 2007. Openmosix kernel patch website, http ://openmosix.sf.net.

Intel, 2007. Hardware virtualization technology project website, http ://www.intel.com/technology/computing/vptech/.

Kerrighed, 2008. http ://www.kerrighed.org/.

Legrand, A., Renard, H., Robert, Y. et Vivien, F., 2003. Load-balancing iterative computations on heterogeneous clusters with shared communication links, *PPAM-2003 : Fifth International Conference on Parallel Processing and Applied Mathematics*, LNCS 3019, Springer Verlag, pp. 930–937.

Medina, A., Lakhina, A., Matta, I. et Byers, J., 2001-003. Brite : Universal topology generation from a user's perspective., *Technical report*, Computer science department Boston University, 1 2001.

MPI base documentation, 2008. http ://www.mpi-forum.org/docs/.

ObjectWeb Consortium Proactive, 2007. http ://www-sop.inria.fr/oasis/ProActive/.

Rudolph, L., Slivkin-Allalouf, M. et Upfal, E., 1991. A simple load balancing scheme for task allocation in parallel machines, *ACM Symposium on Parallel Algorithms and Architectures*, pp. 237–245.

Schoch, J. et Hupp, H., 1982. Computing practices : the 'worm' programs - early expericences with a distributed computation, *Comm. ACM* **25** : 172–180.

SimGrid, 2008. http ://simgrid.gforge.inria.fr/.

T'kindt, V. et Billaut, J., 2002. *Multicriteria Scheduling*, Springer Verlag, Berlin.

VMWare, 2008. http ://www.vmware.com.

Xu, C. et Lau, F. C., 1997. *Load Balancing in Parallel Computers : Theory and Practice*, Kluwer Academic Publishers, Norwell, MA, USA.