RENPAR'14 / ASF / SYMPA
Hamamet, Tunisie, 10 – 13 avril 2002

Ordonnacement pour la grille : une extension de MCT

Yves Caniou* Emmanuel Jeannot

LORIA,
Université Henri Poincaré, campus scientifique
BP 239, 54506 Vandœuvre lès Nancy
{caniou,ejeannot}@loria.fr

Résumé
Dans cet article nous proposons une extension de l'algorithme MCT (Minimum Completion Time) utilisé pour exécuter sur la << *grille* >> un ensemble de tâches dans le modèle agent-client-serveur. Ce nouvel algorithme est basé sur l'historique des tâches déjà ordonnancées. Les performances de cet algorithme sont étudiées et montrent une diminution du temps total d'exécution des tâches pour un coût négligeable devant la durée d'exécution des applications standard.

Mots-clés : Ordonnancement dynamique, ressources partagées, métacomputing, NetSolve, MCT

---

1. Introduction

L'accroissement des performances des réseaux qui relient les centres de calcul et les laboratoires universitaires ainsi que la popularisation croissante du parallélisme permet d'envisager de nouveaux types d'applications utilisant à la fois la puissance des machines et celle du réseau. L'accès transparent à des ressources distribuées, couramment appelé *métacomputing* [5], est un des enjeux majeurs des technologies de l'information. Cependant, cela nécessite de mettre en œuvre des techniques pour faire communiquer les centres, inter-opérer les applications, allouer les ressources, améliorer la qualité de service et la sécurité des transactions...Dans ce cadre un certain nombre d'outils permettent d'accéder à des ressources distantes et d'y exécuter des applications (NetSolve [2], Ninf [7], Globus [4], AppleS [1]). Pour que l'exécution d'une application sur un environnement distribué se déroule le plus efficacement possible, il est important d'optimiser le placement de ses différents composants.
Les algorithmes de Ninf et de NetSolve utilisent des superviseurs pour déterminer l'état du réseau et la charge des serveurs. Cependant ces données ne donnent pas d'informations sur leurs évolutions futures et peut conduire à de mauvaises décisions de placement. Dans cet article, nous utilisons en plus l'historique des problèmes soumis pour simuler l'environnement de manière fiable et prendre des décisions de placement efficaces. Pour cela nous avons mis au point un simulateur d'environnement de calcul distribué qui prend en compte le fait que le réseau et les serveurs peuvent partager leurs ressources entre plusieurs applications.
Nous avons comparé l'algorithme que nous proposons à celui de NetSolve dans le cas de tâches indépendantes exécutées sur des serveurs dédiés.
Le plan de cet article est le suivant : la section 2 décrit NetSolve et MCT ; la modélisation de la grille de calcul est réalisée section 3 ; le simulateur est décrit section 4 ; le nouvel algorithme d'ordonnancement est proposé section 5 ; les résultats expérimentaux sont présentés section 6 ; et la section 7 conclut cet article.

2. NetSolve

Nous présentons ici NetSolve, un *NES* (Network Enable Server) qui utilise MCT pour ordonnancer les tâches qui lui sont soumises.

---

* Ce travail est partiellement financé par la Région Lorraine

2.1. Généralité

NetSolve [2] est un environnement développé à l'université du Tennessee, qui permet de déporter des calculs sur un ensemble de serveurs dédiés. Il est composé de trois entités :

1. un (ou plusieurs) *client(s)* qui a une application à exécuter. Cette application est décomposée en un ensemble de tâches (appelées aussi des *problèmes*) ;
2. un ensemble de *serveurs*, répartis sur le réseau, capables chacun d'exécuter un ou plusieurs problèmes. Les serveurs peuvent être des machines séquentielles ou parallèles ;
3. un *agent* chargé de superviser l'environnement et de répartir l'exécution des problèmes sur les serveurs.

NetSolve fonctionne comme suit : un agent est lancé sur une machine. Chaque serveur s'inscrit auprès de l'agent en lui transmettant la liste des problèmes qu'il sait résoudre, ainsi que son état et la charge du réseau entre lui et l'agent. Quand un client veut faire résoudre un problème par un serveur, il demande à l'agent de lui attribuer un serveur. L'agent détermine le « *meilleur* » serveur et le client s'y connecte en lui envoyant les paramètres du problème. Le serveur effectue les calculs puis retourne les résultats au client. NetSolve permet au client d'envoyer des requêtes bloquantes et non-bloquantes, afin de pouvoir dégager du parallélisme. Par exemple, un client qui souhaite faire exécuter deux problèmes indépendants, peut faire ses deux requêtes avant de commencer à recevoir les résultats de l'une d'elles.

2.2. Algorithme de sélection du serveur

Pour choisir le serveur sur lequel le problème d'un client va être exécuté, l'agent dispose des informations suivantes :

1. la performance crête du serveur exprimée en MFlops ;
2. la charge du serveur. Régulièrement, le serveur communique à l'agent sa charge actuelle. Il s'agit d'un nombre compris entre 0 et 1 donné par l'équivalent de la commande Unix `uptime` ;
3. le nombre d'opérations à exécuter. Ce nombre est calculé en fonction du type du problème à résoudre et de la taille de ses paramètres.
4. la taille des paramètres. Il s'agit de la quantité de données à transmettre du client au serveur pour effectuer les calculs ;
5. l'état du réseau. Il s'agit de la dernière bande passante connue entre l'agent et le serveur ;
6. la taille des résultats. Il s'agit des données résultant du calcul effectué par le serveur et retournées au client.

Pour chaque serveur S
    $D_1(S) =$ durée estimée de transmission des données.
    $D_2(S) =$ durée estimée du calcul du problème.
Choisir le serveur S qui minimise $D_1(S) + D_2(S)$.

FIG. 1 – *Algorithme MCT de NetSolve pour sélectionner les serveurs*

L'algorithme d'ordonnancement de NetSolve est décrit figure 1. La durée estimée de transmission est calculée en divisant la quantité de donnée par la bande passante restante. La durée estimée du calcul est déterminée en fonction de la fraction de la puissance crête disponible sur la machine et du nombre d'opérations à effectuer. Il utilise MCT (minimum completion time) pour allouer un serveur à une tâche : le serveur choisi minimise la date de terminaison de la tâche.

Cet algorithme a le double avantage d'être rapide et robuste. Cependant, la qualité des décisions qu'il prend n'est pas optimale. En effet, la fréquence avec laquelle les informations sur l'état de l'environnement (environ toutes les deux-trois minutes) sont mises à jour conduit à des décisions erronées si par exemple un agent reçoit beaucoup de requêtes entre deux mises à jour. Pour palier à ce problème, NetSolve incrémente la variable représentant la charge de la machine sur laquelle il vient d'attribuer un problème afin de prendre en compte la surcharge occasionnée par l'arrivée d'un problème. Or, la charge d'une machine peut varier très brutalement, ce qui pose un second problème. Par exemple, si le serveur envoie sa charge à l'agent alors qu'il exécute plusieurs problèmes en même temps, elle sera très élevée. Mais si ces problèmes terminent juste après, la charge réelle de la machine va diminuer d'un coup sans que l'agent en soit informé, ce qui pourra le conduire, à tort, à limiter l'utilisation de cette machine. Le travail que nous proposons ici tente d'apporter une réponse à ce dernier problème. Pour ce faire, l'agent conserve en mémoire l'ensemble des problèmes qu'il a attribué aux serveurs et simule leur exécution. Il peut ainsi savoir quand une machine va devenir peu chargée et lui allouer, en conséquence, un voire plusieurs problèmes en cas d'arrivées simultanées.

3. Modèle

Pour modéliser l'environnement et les programmes que nous voulons y exécuter, nous utilisons un modèle proche de celui de NetSolve.
Chaque tâche/problème i à exécuter sur la grille est définie par les paramètres suivants :

1. c_i le nombre d'opérations à effectuer ;
2. D_{p_i} la quantité de données en paramètre ;
3. D_{r_i} la quantité de données en résultat.

Un serveur est défini par les paramètres suivants :

1. t son temps de cycle (le temps qu'il faut pour exécuter une opération) ;
2. b la bande passante de son interface réseau.

Un certain nombre de paramètres dynamiques sont aussi définis pour chaque serveur. Ainsi à chaque instant, on peut connaître :

1. P le nombre de problèmes que le serveur est en train d'exécuter ;
2. c_e le nombre de connexions en entrée avec ses clients et c_s le nombre de connexions en sortie avec ses clients.

Dans cet article nous supposons que l'on utilise la grille de manière exclusive : les serveurs n'exécutent que les problèmes attribués par l'agent. En effet, notre but est de comparer l'algorithme que nous proposons avec celui de NetSolve. Nous pensons que s'il n'est pas meilleur en environnement exclusif alors il ne le sera pas en environnement partagé.
Nous supposons que chaque serveur exécute de manière équitable les P problèmes qui lui sont soumis. Ainsi le temps de cycle réel d'un serveur est à tout instant égal à t/P. Nous faisons l'hypothèse que l'interface réseau fonctionne en mode « full duplex » et partage sa bande passante entre ses différentes connexions. Ainsi à tout instant, la bande passante en entrée (resp. en sortie) est égale à b/c_e (resp. b/c_s). Comme l'application déporte ses tâches à exécuter sur des serveurs de calculs distants via l'agent de NetSolve, nous nous intéresserons ici à des problèmes à *gros grain*. C'est à dire que pour chaque problème, sa durée de communication sur l'interface réseau la plus rapide est inférieure à sa durée de calcul sur le serveur le plus lent, soit :

$$\forall i \max_{\text{serveurs}}((D_{p_i} + D_{r_i})/b) < \min_{\text{serveurs}}(c_i.t)$$

4. Simulation de la grille

Pour simuler la grille, nous avons implanté une fonction qui gère, pour un serveur donné, l'ajout de tâches et les mises à jour sur l'ensemble des états des tâches présentes et non encore terminées. Pour une tâche, nous définissons quatre états, quelque soit le serveur considéré :
– *paramètre*. La partie communication des données en entrée de la tâche est en train d'être exécutée ;
– *calcul*. La partie traitement de la tâche est en train d'être exécutée ;
– *résultat*. La partie communication des données en sortie est en train d'être exécutée ;
– *terminée*, lorsque la tâche est finie.

t=0

debut	0
duree	20
duree prevue	20
terminaison	20
duree restante	20

t=10

debut	0
duree	20
duree prevue	20
terminaison	30
duree restante	10
debut	10
duree	20
duree prevue	30
terminaison	40
duree restante	20

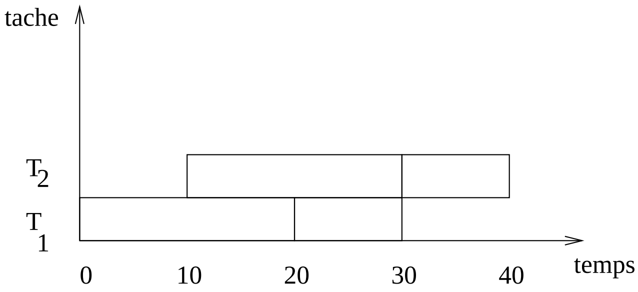

FIG. 2 – Exemple d'évolution des informations des tâches

Chaque état est caractérisé par les informations suivantes :
- le *début de la tâche*, i.e. sa date d'arrivée dans l'état ;
- la *durée de la tâche* dans cet état, sur le serveur considéré lorsqu'il est dédié uniquement à la tâche ;
- la *durée prévue*. C'est la durée réelle de la tâche dans l'état si aucune autre tâche n'est ajoutée après la date considérée (c'est à dire que la tâche change d'état avant l'arrivée d'une autre tâche dans cet état). Dans ce cas, la *date de terminaison* est la somme du *début* et de la *durée prévue* ;
- la *date de terminaison*. C'est la date à laquelle la tâche change d'état ;
- la *durée restante*, i.e. la durée d'exécution de la tâche par rapport à la date d'arrivée de la dernière tâche dans l'état, si elle pouvait disposer de la totalité de la ressource où elle a été allouée.

Certaines de ces données n'évolueront pas au cours du temps. Ainsi, les informations *début*, *durée* et *durée prévue* ne changeront pas une fois données. Mais la *durée prévue* nécessite de simuler l'exécution de la tâche en temps partagé avec les autres tâches. La *durée restante* est une information temporaire qui nous permet de déterminer la *date de terminaison* de la tâche considérée en fonction de l'historique du serveur où elle est allouée. La figure 2 nous montre l'évolution de ces informations pour l'arrivée dans le même état sur le même serveur de deux tâches d'une durée de 20 secondes, à la date $t = 0$ et $t = 10$. Au temps $t = 10$ l'arrivéede la tâche T_2 a pour effet d'accroître la date de terminaison de la tâche T_1.
Dans notre contexte, l'agent reçoit des requêtes les unes après les autres, au fur et à mesure des besoins de l'application. Ces requêtes arrivent donc selon un temps croissant, que l'on retrouve dans l'état paramètre sur les différents serveurs. Les durées des tâches (communication ou calcul) étant variables, on peut obtenir un tout autre ordre dans l'état *calcul*, ainsi que dans l'état *résultat*. Les algorithmes s'en trouvent donc légèrement modifiés.

4.1. Simulation de la transmission des messages en entrée

Il s'agit d'ajouter une tâche T dans son état paramètre à l'historique H du serveur s, T arrivant à la date *début*. Cet état a ceci de particulier que les temps d'arrivée sont ordonnés par ordre de réception des requêtes par l'agent, donc par ordre croissant sur leur champ *début*. On réalise la mise à jour de l'historique des tâches grâce à l'algorithme de la figure 3.

4.2. Simulation des exécutions dans les états calcul et résultat

La date calculée dans le champ *fin de la tâche* pour chaque tâche est la date d'entrée dans le nouvel état. L'arrivée d'une tâche en état *calcul* ou *résultat* ne se fait pas forcément après celles arrivées précé-

1 On simule sur la ressource l'exécution des tâches déjà présentes sur le serveur jusqu'à la date d'arrivée de la nouvelle tâche, par la mise à jour de leurs informations *date de terminaison* et *durée restante*.
2 Les tâches dont la *durée restante* s'annule quittent l'état.
3 On ajoute dans l'historique la tâche dont on veut simuler l'exécution.
4 On regarde ce que donne la simulation des tâches restantes dans l'état *paramètre* en allouant à chacune la même quantité d'accès à la ressource.
5 On met à jour les informations *fin de la tâche* des tâches restantes.
6 On sauvegarde l'information *durée finale prévue* pour la tâche nouvellement arrivée.

FIG. 3 – *Algorithme de simulation de l'état paramètre*

demment dans l'état antérieur, i.e. dans le même ordre observé dans l'état antérieur. En effet, à cause des tailles variables des données en entrée ou du nombre d'opérations à effectuer, donc de la durée du traitement, les tâches ne changent pas forcément d'état dans leur ordre d'arrivée.

La correspondance entre l'ordre d'arrivée des tâches dans l'état et l'ordre d'apparition des requêtes pour l'accès aux informations des tâches se fera donc au travers d'un tableau d'indices, ordonné selon l'arrivée des tâches dans l'état considéré. Il s'agit ensuite de répercuter sur les tâches arrivées plus tôt, l'arrivée des tâches ultérieures, une par une, en utilisant une variante de l'algorithme proposé figure 3.

En conclusion, notre simulation de la grille permet non seulement de connaître à chaque instant quel est l'état de l'environnement mais aussi de prévoir le temps que va prendre l'exécution d'un problème.

5. Nouvel algorithme d'ordonnancement

L'algorithme que nous proposons figure 4, s'appuie sur l'historique des tâches qui sont fournies aux serveurs et étend la stratégie MCT utilisée par NetSolve. Un agent est contacté par plusieurs clients, chacun demandant le meilleur serveur possible pour exécuter ses tâches le plus rapidement possible. La philosophie de NetSolve va le conduire à considérer à l'instant d'arrivée de la tâche les informations sur le réseau afin d'évaluer la date de terminaison de cette nouvelle tâche. Mais il peut choisir un serveur moins chargé qu'un se libérant aussitôt après, ce qui aura pour conséquence une perte possible de temps par non utilisation optimale de la ressource. Notre idée est donc de limiter ces pertes dues à ces variations subites de charge. Pour cela, l'agent va conserver dans sa mémoire les informations des tâches allouées aux différents serveurs dont il dispose, pour répartir ses problèmes. Construisant l'historique de chacun au fur et à mesure qu'il répartit les tâches, il lui est facile de simuler l'exécution de la tâche (figure 3).

1 **Pour chaque** serveur S
2 Simuler l'ajout de la nouvelle tâche sur S
3 Allouer la nouvelle tâche sur le serveur ayant offert la date de terminaison minimum

FIG. 4 – *Nouvel algorithme d'ordonnancement proposé*

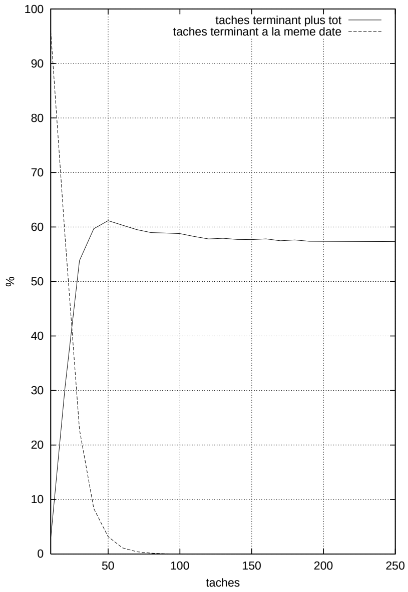

FIG. 5 – nombre de tâches moyen, pour 1000 simulations sur 25 serveurs, pour $\mu = 0$

6. Expériences de simulation

Nous avons généré à l'aide de la bibliothèque GSL [6], différents réseaux complets et applications aléatoires. Si on note $U(x, y)$ la loi uniforme, le nombre d'instructions d'un serveur, en millier, est donné par $U(150, 500)$ et les quantités de données des tâches, indépendantes pour l'instant, correspondent à $U(1Ko, 300Mo)$, le temps de calcul étant donné de telle sorte qu'il satisfasse l'hypothèse du modèle, et que la tâche soit au maximum d'une durée de 1000 sur le serveur le plus rapide. L'arrivée d'une tâche obéit à une loi de Poisson, de paramètre μ. De plus, l'algorithme d'ordonnancement de NetSolve ayant besoin de connaître l'état de l'environnement, et cet état lui étant normalement transmis de manière régulière par les serveurs, il lui est fourni dans le cadre de nos expériences par le simulateur lorsqu'il en a besoin. La précision de ces connaissances du réseau en est donc meilleure.

Chaque expérience a été conduite en faisant 1000 itérations et en regardant la moyenne des résultats obtenus. Les résultats nous montrent que, pour μ variant de 0 à 60, le makespan de l'application est en général 10% plus petit pour notre algorithme (figure 6). Par contre, comme on peut le voir en comparant les figures 5 et 7, le nombre de tâches où nous terminons plus tôt décroît en fonction de μ. Ceci est dû au choix que fait notre algorithme : lorsqu'il place une tâche, il prends le serveur qui minimise le temps de terminaison de cette tâche, et ceci, au détriment des tâches déjà présentes sur le serveur. MCT utilisé dans NetSolve est basé entre autre sur la charge des serveurs. Il y aura donc un équilibrage naturel. Notre algorithme exploite au maximum les capacités des serveurs dont il connait l'existence.

Dans la poursuite de nos travaux, nous souhaitons évoluer vers un modèle plus proche de la réalité, où les serveurs ne sont pas dédiés exclusivement à l'agent, comme c'est le cas actuellement. Nous utiliserons certainement Simgrid [3], afin de béneficier de l'implantation de l'utilisation de traces NWS [8].

7. Conclusion

Cet article décrit un nouvel algorithme d'ordonnancement sur la grille, basé sur la prise en compte de l'historique des problèmes déjà affectés aux serveurs ainsi que sur la simulation des communications et des calculs en temps partagé. Cet algorithme est particulièrement adapté à NetSolve où un agent centralisé doit allouer des problèmes à des serveurs à partir des requêtes de clients.

Nos expériences montrent que cet algorithme est plus performant que celui de NetSolve pour un coût négligeable devant la durée des problèmes à exécuter.

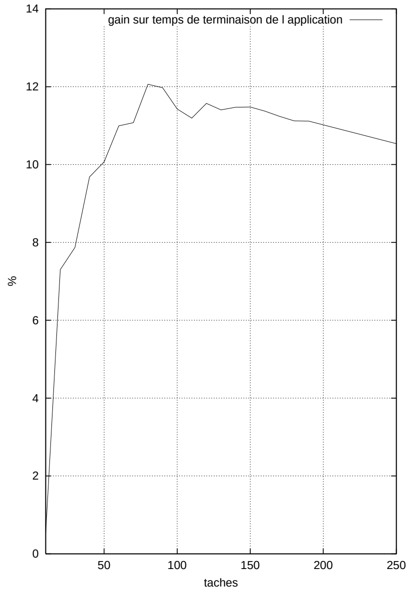

FIG. 6 – date de terminaison moyenne, pour 1000 simulations sur 25 serveurs, pour $\mu = 0$

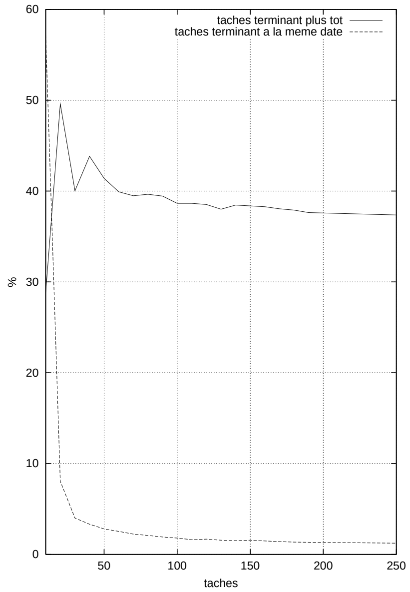

FIG. 7 – nombre de tâches moyen, pour 1000 simulations sur 25 serveurs, pour $\mu = 40$

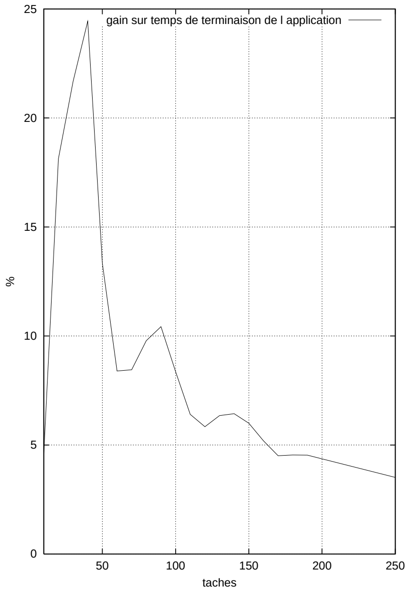

FIG. 8 – date de terminaison moyenne, pour 1000 simulations sur 25 serveurs, pour $\mu = 40$

Dans nos futurs travaux nous allons implanter cet algorithme dans le code de NetSolve et nous réaliserons des expériences en vraie grandeur. Nous souhaitons prendre en compte les informations de superviseurs sur l'état réel du réseau et de la charge des serveurs pour accroître la qualité de nos prises de décision lorsque l'on se trouve dans un environnent perturbé.

Bibliographie

1. AppLeS.- *Application Scheduling on the Information Power Grid*. 1998.
2. Casanova(H.) et J.Dongarra. -*NetSolve : A network server for solving computational science problems.* In : Proceedings of Super-Computing'96.-Pittsburg.
3. Casanova(H.). -*Simgrid : a Toolkit for the Simulation of Application Scheduling* - CCGrid01.
4. Foster(I.) et Kesselman(C.). -*The Globus project : A progress report.* In : Heteregeneous Computing Workshop.- mars 1998.
5. Foster(I.) et Kesselman(C.). -*The Grid : Blueprint for a New Computing Infrastructure.* - Morgan - Kaufmann, 1998.
6. GSL -*The Gnu Standard Library, http ://www.gnu.org/software/gsl/gsl.html*
7. Hidemoto Nakada, Mitsuhisa Sato, Satoshi Sekiguchi.- *Design and Implementations of Ninf : towards a Global Computing Infrastructure* - Future Generation Computing Systems, Metacomputing Issue, Vol.15, Issues 5-6, pp.649-658, 1999.
8. R. Wolski, Neil T. Spring, Jim Hayes. -*The Network Service : A Distributed Resource Performance Forecasting Service for Metacomputing* - Journal of Future Generation Computing Systems,Volume 15, Numbers 5-6, pp. 757-768, October, 1999.