Département de formation doctorale en informatique
UFR STMIA

École doctorale IAEM Lorraine

Ordonnancement sur une plate-forme de métacomputing

THÈSE

présentée et soutenue publiquement le 16 décembre 2004

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Caniou Yves

Composition du jury

Rapporteurs : Laurent Philippe, Professeur, Université de Franche–Comté
Denis Trystram, Professeur, Institut National Polytechnique de Grenoble

Examinateurs : Alexander Bockmayr, Professeur, U.H.P. Nancy I
Christophe Cérin, Maître de conférences, Université de Picardie Jules Vernes, Amiens
Emmanuel Jeannot, Maître de conférences, U.H.P. Nancy I (Directeur de thèse)
Jens Gustedt, Directeur de recherche à l'INRIA–LORIA (Directeur de thèse)

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Résumé

L'exécution d'une application sur la grille de calcul se fait au moyen de couches logicielles intermédiaires. Parmi ces couches, nous trouvons les intergiciels de grille. L'objectif de la thèse consiste à proposer des algorithmes d'ordonnancement dynamiques efficaces dans le cadre des systèmes GridRPC.

Nous avons conçu un module de prédictions de performance (HTM). Il donne des estimations fiables des durées des tâches qui peuvent s'exécuter concurremment sur un serveur de calcul. Nous avons aussi élaboré des heuristiques d'ordonnancement dynamiques et multi–critères reposant sur le HTM. En plus du makespan, nos heuristiques tentent de donner une meilleure qualité de service à chaque requête, comme de réduire le temps de réponse moyen.

Les heuristiques et le HTM ont été implantés afin de les étudier à l'aide de simulations. Des expériences intensives et réelles ont permis d'étudier concrètement les performances du HTM au sein d'un intergiciel, d'affiner la précision de ses informations en introduisant des mécanismes de synchronisation et d'étudier l'efficacité de nos heuristiques.

Mots-clés: heuristiques d'ordonnancement dynamiques, serveurs de calcul hétérogènes et distribués, simulations de plates-formes de métacomputing, gestionnaire de l'historique des tâches, interférences

Abstract

Grid applications are typically built using grid middlewares. In our work, we study scheduling in client-agent-server based middlewares. Specifically, we focus on GridRPC systems that use dynamic scheduling algorithms. The objective of this dissertation is to propose efficient dynamic scheduling algorithms in this context.

In our work, we have developed a performance prediction module (HTM) that is able to give accurate estimations of the remaining duration of tasks, which can possibly execute concurrently on a server. We have elaborated multi-criteria dynamic scheduling heuristics. While most heuristics optimize only for the makespan, ours seeks to provide a better quality of service by, for example, also reducing the mean flow.

The HTM and our heuristics have been implemented and studied with simulations. We have also performed extensive real experiments to observe the performance of the HTM in a grid environment, to improve HTM's accuracy by introducing synchronization mechanisms and to study the efficacity of our heuristics.

Keywords: dynamic scheduling heuristics, heterogeneous distributed computing servers, simulations of metacomputing platforms, historical trace manager, contention

Remerciements

Tout d'abord, je souhaite remercier ma Famille. À mes parents, havre de paix toujours accueillant, qui m'ont toujours soutenu, cette thèse est pour vous. À mes petites familles du bout du monde et de la montagne, je pense bien à vous. Cela fait quelques temps maintenant que partout où je vais, j'emmène avec moi les rires enchanteurs de mes gamins...

Je remercie ensuite mes directeurs de thèse, Emmanuel et Jens : d'avoir accepté d'encadrer cette thèse naturellement, mais aussi pour leur pertinence, leurs remarques constructives, leur patience et pour l'autonomie qu'ils ont bien voulu m'accorder. Emmanuel, ton recul et ta vivacité d'esprit, en particulier lors de la rédaction des articles, continue encore de m'impressionner.

Je remercie aussi les membres du Jury qui m'ont fait l'honneur d'accepter d'examiner ce travail de thèse. Merci pour vos remarques constructives qui m'ont permis d'améliorer ce manuscrit.

Merci aux différents membres d'AlGorille : les assistantes de projet Josiane et Isabelle, pour leur gentillesse et leurs sourires, mes co-bureaux Mohammed et Fred. W. de m'avoir tant laissé gagner à Frozen ;). Fred, merci d'avoir facilité le passage à la debian. Qu'est-ce que je pourrais bien faire sans maintenant ?.. :P.
Algorille vient tout juste de s'aggrandir et je vous souhaite à tous tout plein de bonnes choses.

Merci aux personnes que j'ai pu rencontrer pendant cette thèse. Certaines via le monitorat (amis chanteurs, JB – je viendrai à ta soutenance, be strong ! ;) –).
Mais aussi au Loria (youp les dromadaires ! Tant de repas que nous avons partagés ensemble, Isa, Olivier(s), Benjamin, Daniel, Julien – 'falloir te mettre à danser ;) –, Nico, Yann, J.-C. et tous les autres...).
Je n'oublie pas non plus les autres, certains ayant même pour mon plus grand plaisir accepté d'être rapporteurs de cette thèse : Fred D., Fred S. (encore merci pour tes bons conseils !), Denis, Laurent, Fredo, Eddy, Arnaud, Martin, Henri, Yves, Jack...

Je suis actuellement ATER à l'École normale supérieure de Lyon, où j'ai terminé la rédaction de ce manuscrit. Elle s'est trouvée grandement facilitée par l'ambiance et l'amitié que j'ai pu y trouver. À ce titre, je tiens à remercier les gens du LIP (en particulier Éric, Alain..) et tout particulièrement mes amis de GRAAL.

Durant cette thèse, j'ai utilisé de très nombreux programmes qui m'ont été d'une utilité incroyable, au point que je n'utilise maintenant plus qu'eux. Je remercie donc, c'est la moindre des choses, tous les acteurs œuvrant pour les Logiciels Libres. De la même manière, je ne peux terminer mes remerciements sans référence aux artistes qui ont égayé mes journées et mes nuits par leurs mélodies si ennivrantes.

Enfin, parmi les relecteurs de ce manuscrit, je souhaite chaleureusement remercier ma douce Coline. En plus de m'avoir supporté au quotidien – et de continuer ;P –, tu t'es beaucoup investie dans ces nombreuses relectures au point qu'il n'y a presque plus de fautes, qu'il y a moins de redondances et de tournures maladroites. Merci pour ces demi-journées passées avec moi devant l'ordi, c'est grâce à toi si ce manuscrit est plus agréable à lire.

À ma famille

À ma Douce,
Ma Belle,
Mon Inédite

Table des matières

Table des figures

Chapitre 1

Introduction

Les applications scientifiques requièrent de plus en plus de calculs. L'augmentation de leurs besoins est bien plus rapide que celle, due aux progrès technologiques, de la puissance calculatoire des nouvelles machines. En effet, les applications fondées sur des simulations demandent plus d'itérations ou instancient des modèles plus complexes afin d'affiner leurs estimations (comme c'est le cas pour les estimations météorologiques par exemple[1]). Les domaines de la biologie et de la physique nécessitent le traitement d'un volume de plus en plus important de données[2] ou demandent simplement le maximum de ressources possible[3], voire plus généralement les deux avec l'imagerie médicale ou le projet LCG[4] de la grille de calcul du Grand Collisionneur de Hadrons (LHC)[5].

Auparavant, les seules possibilités offertes pour accélérer l'exécution d'une application étaient de la paralléliser *à la main* et de l'exécuter sur une machine parallèle. Depuis, des modèles de programmation et des solutions logicielles (bibliothèques de fonctions, compilation, langages de programmation visuels [BDH+94]) ont quelque peu simplifié le procédé. La technologie a aussi apporté les super-calculateurs dont le nombre de processeurs dépasse la centaine. Ils disposent d'une quantité de mémoire vive centralisée de plusieurs giga–octets et leur capacité disque se mesure en péta–octets. Certains des super-calculateurs les plus récents s'articulent autour d'une architecture NUMA (Non Uniform Memory Access) et comportent plus d'un millier de processeurs regroupés en nœuds qui sont interconnectés par du réseau très haut débit (par exemple Earth Simulator [SSYS04] dispose de 640 nœuds de 8 processeurs interconnectés à 12.3Go/sec). Cependant, les coûts importants de telles architectures les prédisposent *de-facto* à de gros centres de recherche et entreprises dont les besoins nécessitent de telles performances avec ce type d'architecture.

En parallèle le circuit de grande consommation a vu apparaître depuis une dizaine

[1] `http://www.eurogrid.org/wp2.html`
[2] `www.griphyn.org/`, `http://www.ppdg.net/` et `http://egee-intranet.web.cern.ch/`
[3] `http://setiathome.ssl.berkeley.edu/`
[4] `http://lcg.web.cern.ch/LCG/`
[5] `http://public.web.cern.ch/public/about/future/future.html`

d'années des ordinateurs personnels très performants à des prix très attractifs. Le courant actuel des universités, centres de recherche et entreprises tend donc plutôt vers l'acquisition de grappes d'ordinateurs composées de machines à bas prix reliées par des réseaux classiques. De plus, elles utilisent généralement le système d'exploitation libre Gnu/Linux[6] qui offre les fonctionnalités d'UNIX sans investissement supplémentaire. En plus d'avoir l'avantage d'un rapport performance/prix plus intéressant, une telle architecture dispose ainsi d'une capacité de calcul capable de rivaliser voire de dépasser celle des machines parallèles comme le montre le classement des machines de calcul haute performance parmi les 500 plus performantes[7]. Les centres de recherche disposent en général de plusieurs types de ressources, super-calculateurs et grappes d'ordinateurs, chacun étant complémentaire de l'autre et offrant des capacités bien distinctes.

Naturellement, agréger la puissance de ces ressources permettrait déjà de disposer au sein d'un établissement de plus de puissance. Mais la vulgarisation d'internet, avec l'arrivée massive d'offres du haut débit à bas prix, permet d'étendre encore le concept. En effet, d'après [GBG+03], les ordinateurs ont une moyenne de vie de 4 ans et, d'après la loi de Moore, sont renouvelés par des machines dont la capacité double tous les 18 mois. De plus, chaque année le nombre de réseaux connectés à Internet augmente de 50%. D'après les auteurs, la puissance de calcul sur Internet s'accroît en conséquence d'un facteur $2,7$ chaque année.

Le concept de la grille est expliqué par I. Foster et C. Kesselman dans [FK99]. Le raisonnement est fondé sur l'analogie à la grille électrique : il s'agit d'interconnecter l'ensemble de toutes les ressources disponibles et de faire profiter l'ensemble des utilisateurs des capacités de calcul et de stockage de l'agrégat ainsi constitué. Toutefois, les ressources sont hétérogènes et distribuées géographiquement et la difficulté de leur gestion ne doit pas incomber à l'utilisateur. Il doit pouvoir y avoir accès «aussi simplement» qu'en branchant un appareil à une prise électrique pour obtenir l'énergie nécessaire à son fonctionnement. La capacité d'abstraction des ressources et de leurs caractéristiques (par exemple d'ordre matériel, leur puissance ou encore leur dynamicité) est appelée le **grid computing**. Nous en discernons la partie **métacomputing** qui est plus proche des besoins applicatifs des utilisateurs et porte plus particulièrement sur l'utilisation transparente des ressources de calcul. Par induction, une *plate–forme de métacomputing* est une grille de calcul.

Actuellement, la grille en tant qu'interconnexion globale et unifiée des ressources, n'est encore qu'un concept. En revanche, des modèles de grilles ont été créés pour caractériser les nombreuses grilles existantes et leurs applications en fonction de leurs besoins relatifs prépondérants : les *Information Grid* concernent les grilles axées sur les ressources de stockage, où la rapidité et la gestion des communications sont déterminantes dans les performances des applications ; les *Desktop Grid* regroupent des ressources qui s'enregistrent auprès d'entités définies pour demander à effectuer des tâches de calcul ; les *Server Grid* désignent les grilles de calcul où l'aspect calculatoire des applications est prédominant, ou encore les *Global Grid* dont la capacité d'abstraction des ressources amène à considérer la

[6] `http://www.kernel.org/` et `http://www.gnu.org/`
[7] `http://www.top500.org/`

grille comme un *méta–ordinateur*.

L'exécution d'une application sur la grille se fait au moyen de couches logicielles intermédiaires. Parmi ces couches, nous trouvons les intergiciels de grille (en anglais «grid middleware») qui s'appuient sur la couche des services. La couche service regroupe entre autres l'ensemble des fonctionnalités nécessaires à la bonne exécution d'une application par l'intergiciel. Elle contient en particulier les services de gestion des ressources, de gestion des utilisateurs, de gestion des transferts de données, d'ordonnancement, de sécurité, de déploiement et de prédiction. L'intergiciel définit le modèle de grille et le modèle de programmation des applications et fournit la ou les API (*Application Programming Interface*) en conséquence. Il utilise les services pour distribuer l'exécution de l'application sur la grille et aide l'utilisateur à obtenir de bonnes performances en répartissant les travaux soumis sur les ressources disponibles.

Cette efficacité dépend toutefois du déploiement des composantes de l'intergiciel ainsi que de la performance offerte à chaque application qui lui est soumise. Notons que la notion de performance est intimement liée à la ou les métriques regardées (comme «terminer le plus tôt» ou encore «utiliser le moins de bande passante»). L'optimisation de l'utilisation des ressources de la grille incombe au service d'ordonnancement. C'est un service critique qui fait l'objet de cette thèse. En effet, l'ordonnancement des tâches de calcul sur les différentes ressources doit être spécifiquement conçu : il s'agit de déterminer les ressources utilisées et quand elles exécuteront le travail demandé afin d'optimiser la métrique souhaitée. Dans ce but, les stratégies d'ordonnancement nécessitent des prédictions de performance de qualité : par exemple des prédictions sur l'évolution de l'utilisation des ressources ou sur la durée des tâches.

Dans notre travail, nous considérons plus particulièrement le contexte GridRPC du modèle client–agent–serveur sur la grille que les NES (*Network Enable Server*) instancient. Le module d'ordonnancement appelé par ce type d'intergiciel prend ses décisions grâce à un algorithme dynamique. Généralement, c'est une variante de MCT (*Minimum Completion Time*) qui est utilisée. Son but est de minimiser le makespan, c'est–à–dire la date de terminaison des requêtes. L'objectif de la thèse consiste à proposer des algorithmes d'ordonnancement dynamiques efficaces dans le cadre des grilles de calcul et qui soient entre autres capables de remplacer avantageusement cette heuristique.

Parmi les contributions de notre travail, nous avons conçu un module de prédictions de performance, le gestionnaire de l'historique des tâches (HTM, *Historical Trace Manager*), capable de donner une estimation fiable de la durée des tâches. De nombreuses études existent sur la prédiction de la durée des tâches. A notre connaissance, aucune n'utilise les connaissances disponibles en couplant les décisions établies au fur et à mesure par le module d'ordonnancement avec les prédictions, pour des tâches capables de s'exécuter concurremment sur les serveurs de calcul. Le HTM est donc fondé sur les interférences que des tâches concurrentes pour une ressource de calcul ont les unes sur les autres et donne ainsi le retard occasionné pour chacune d'elles.

Nous avons conçu des heuristiques d'ordonnancement dynamiques et multi–critères.

Le makespan étant la métrique la plus couramment utilisée, les heuristiques tentent de plus, avec l'aide des informations délivrées par le HTM, de donner une meilleure qualité de service à chaque requête et par extension à chaque client, comme de réduire le temps de réponse moyen des requêtes. Nous avons programmé une plate–forme de lancement d'applications (tâches indépendantes dont la date d'arrivée est fixée ou applications composées de relations de précédence) reposant en partie sur Simgrid, un simulateur de plate–forme.

Les heuristiques et le HTM ont ensuite été implantés afin de les étudier et de les comparer sur des expériences de simulation. NetSolve est l'un des intergiciels de référence, librement téléchargeable sur internet. Nous avons utilisé plusieurs de ses versions (1.4.1 et 2.0) afin de réaliser des expériences réelles. Elles nous ont permis d'étudier concrètement l'emploi du HTM dans un environnement de résolution de problèmes, de travailler la précision de ses informations en introduisant des mécanismes de synchronisation et d'étudier l'efficacité réelle de nos heuristiques. Ceci a d'ailleurs donné lieu à une collaboration avec l'équipe de NetSolve : les mécanismes de synchronisation mis en place pour le HTM sont incorporés dans NetSolve depuis la version 2.0. Ils sont en particulier utilisés par visperf [LDR02], l'outil de surveillance temps–réel développé conjointement à NetSolve.

Ce manuscrit s'organise en trois premiers chapitres qui donnent un état de l'art sur les différents points principaux dont traite la thèse et de deux chapitres présentant le travail effectué. Son plan est le suivant : dans le chapitre 2, nous caractérisons les grilles et donnons les attentes majeures sur les différents outils et intergiciels nécessaires à leur exploitation. Le chapitre 3 décrit la problématique de la surveillance des capacités des ressources, de la prédiction de leur évolution et de l'estimation des performances des services démandés. Nous présentons au chapitre 4 une taxonomie de l'ordonnancement, les différentes métriques qu'il est possible d'optimiser et des travaux liés aux algorithmes d'ordonnancement pour ressources hétérogènes et distribuées. Le HTM, notre module de prédiction de performance, est décrit dans le chapitre 5. Nous y présentons les heuristiques que nous avons conçues et étudiées à l'aide d'expériences de simulation et grâce à des expériences réelles. Nous étudions dans le chapitre 6 la précision du module de prédiction de performance, avant et après l'ajout d'un mécanisme de synchronisation avec l'exécution réelle des tâches sur la plate–forme. Les performances observées sur des expériences réelles sont aussi présentées et montrent qu'il est possible de réaliser des gains conséquents pour chaque application soumise dans le système grâce aux informations délivrées par le HTM et aux heuristiques multi–critères proposées. Pour finir, nous concluons et nous présentons les perspectives de nos travaux de recherche au chapitre 7.

L'étude préliminaire des heuristiques HMCT, MP et MSF sur des expériences de simulation a été publiée dans les actes de *Renpar*'02. Les résultats suivants ont été en partie publiés dans les actes de l'école *Grid'02* [Can02] et font l'objet d'une publication dans les actes de la conférence *QOS and Dynamic System workshop*, QDS'04 [CJ04a]. La section 5.5.3 fait l'objet d'une publication dans les actes de la conférence *Heterogeneous Computing Workshop*'03 [CJ03]. Une partie des résultats des sections 5.5.4 et 6.2 a été publiée dans les actes de la conférence *Euro–Par*'04 [CJ04b].

Chapitre 2

Plates-formes de métacomputing et leur gestion

Table des matières

2.1 Introduction

Dans [FK99], C. Kesselman et I. Foster avancent le concept de la grille par analogie à la grille électrique. Premièrement, il faut interconnecter l'ensemble des ressources afin d'agréger leurs performances propres, calcul et stockage. Ensuite, l'accès à ces ressources et à leur puissance de traitement, clairement hétérogènes et distribuées géographiquement, doit se faire aussi simplement que de brancher une prise de courant pour profiter de la puissance électrique. Tout doit s'opérer en totale transparence pour l'utilisateur : le partage, la gestion des ressources, comment, pourquoi et où se fait l'affectation d'un travail demandé sont des questions dont l'utilisateur ne doit pas avoir à se soucier. Cette capacité d'abstraction des ressources, dont l'utilisation ne nécessite pas la connaissance de caractéristiques comme leur puissance ou leur dynamicité, est ce que nous nommons le **grid computing**. Nous définissons le **métacomputing** comme une partie du grid computing, plus proche de la notion applicative de l'utilisateur et plus particulièrement axée sur l'utilisation de grilles de calcul. Par induction, une plate-forme de métacomputing est une grille de calcul dont les ressources sont hétérogènes et distribuées.

Selon l'échelle de la grille considérée, on pourra aussi parler de *global computing*. Par nature, il y a un nombre variable d'utilisateurs, ce qui implique une puissance relative accordée à chacun. Dans le cas du *global computing*, la puissance de la plate–forme peut de plus énormément varier du fait de ses ressources particulièrement volatiles.

Nous utiliserons par la suite le terme *grille* pour désigner une plate–forme de ressources hétérogènes et distribuées. Nous parlerons de «la grille» pour dénommer le concept, l'agrégat de plate-formes dont le but à terme est de converger par leur interconnexion vers son unicité et son ubiquité. Contrairement à la définition des clusters rappelée dans [Ess04] où tous les nœuds sont censés être homogènes, nous considérons ici les *clusters* comme des grappes de machines, c'est–à–dire des grilles désignant un agrégat de ressources potentiellement hétérogènes reliées par un réseau local dans un même domaine administratif.

L'utilisation de ressources distribuées et homogènes n'est pas facile et l'hétérogénéité ajoute des problèmes supplémentaires. La gestion des ressources d'une grille concerne leur accès et le partage de leurs capacités entre les différents utilisateurs. Des outils ont été développés afin d'améliorer la qualité de la gestion tout en la simplifiant. Choisis en fonction de la nature de la grille considérée, ils peuvent assurer de simples services de test de présence jusqu'à proposer des environnements de travail complets et performants.

Dans ce chapitre, nous présentons tout d'abord une classification des grilles en fonction de leurs caractéristiques dans la section 2.2. Nous parlons ensuite de leur exploitation à l'aide d'outils et d'intergiciels qui leur sont adaptés dans la section 2.3. A la section 2.4, nous décrivons les services auxquels doit répondre un environnement de résolution de calculs ainsi que les problèmes inhérents au contexte de la grille et certaines solutions communément utilisées. Nous présentons plus en détail certains intergiciels en fonction des méthodes et concepts utilisés pour gérer les ressources de la grille dans la section 2.5. Enfin, nous concluons à la section 2.6.

2.2 Caractéristiques des grilles

Pour le moment, la grille peut être considérée comme une interconnexion dynamique et totalement hétérogène de réseaux locaux d'ordinateurs et de ressources de stockage affiliés à une ou plusieurs organisations indépendantes. Mais sa définition et ses caractéristiques sont encore à définir clairement [NS03]. Ainsi, dans [FKT01], la définition des grilles se traduit plutôt sur la façon dont on peut la construire et quels composants, couches, protocoles et interfaces doivent être fournis. On y trouve une description de leurs rôles

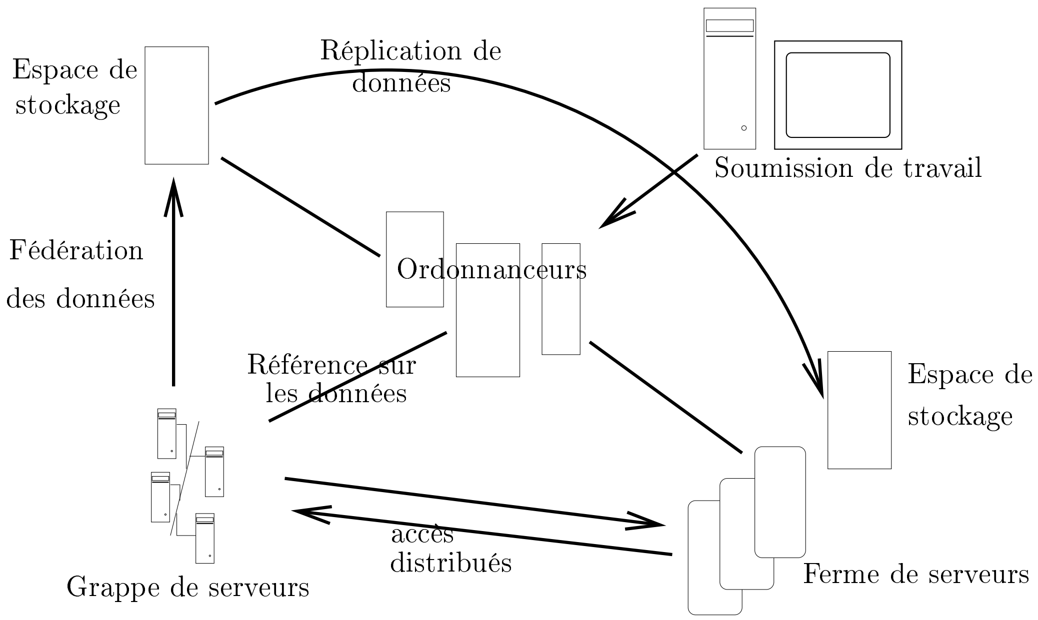


Fig. 2.1 – Information Grid

respectifs et de leurs interactions.

La notion de grille, si elle renvoie à l'image d'un agrégat de ressources, suggère souvent dans sa définition une idée sur le type des travaux et/ou le type de gestions qu'il va lui être appliqué. Ainsi, on parlera de :

– **Information Grid** pour une grille où les ressources de stockage et les échanges de données sont primordiaux dans l'efficacité qu'elle apportera à l'exécution des applications comme le montre par exemple la figure 2.1. La gestion de la distribution, la réplication et la fédération des données doivent entre autres être particulièrement bien conçues et optimisées. L'objectif ultime en terme de performance est de fournir la facilité et la rapidité d'accès aux données en donnant l'impression aux utilisateurs qu'ils manipulent des données locales,
– **Desktop Grid** pour une grille fonctionnant selon l'approche dite technique de *pull* : ce sont généralement les ressources de calcul qui demandent auprès des entités dont c'est la charge, le travail qu'elles doivent exécuter (s'il en reste). On peut donner comme exemple d'une telle utilisation des desktop grids le cas des applications `*@home`[8]. Typiquement une application doit faire un calcul sur un domaine de définition donné. Comme le montre la figure 2.2, ce domaine est découpé et l'application est finalement exécutée sur plusieurs nœuds, chacun calculant sur un sous-domaine propre et ne générant aucune communication entre chaque instance. Le parallélisme dégagé ici est majoritairement dû au nombre de ressources impliquées dans la résolution du problème. On peut noter que dans certains cas, le nombre de ressources à un instant donné pourra être très important mais ses capacités pourront beaucoup

[8] `http://gridcafe-f.web.cern.ch/gridcafe-f/gridprojects/athome.html`

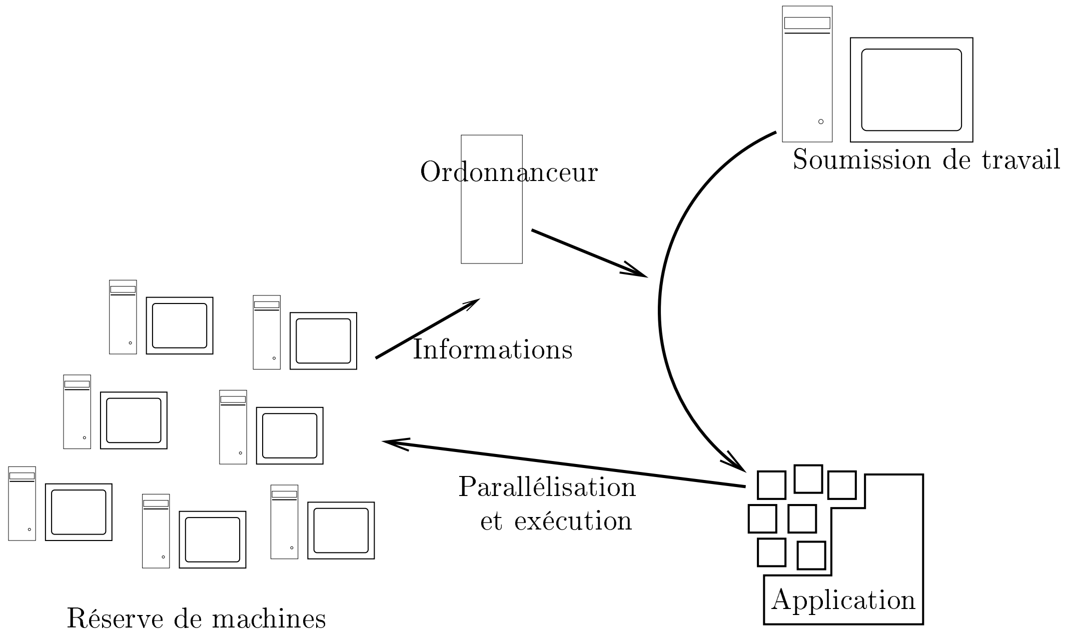


FIG. 2.2 – Desktop Grid

fluctuer au cours du temps,
- **Server Grid** pour une grille dont les ressources de calcul sont plus particulièrement mises en avant par rapport au réseau ou aux ressources de stockage. Ce type de grilles, dont la figure 2.3 donne un exemple, est donc particulièrement intéressant pour les applications gourmandes en temps de calcul et qui ne nécessitent pas ou peu de communication en terme de quantité de données à échanger. On classe généralement ce type d'application sous l'appellation *Calcul Haute Performance*. Dans cette approche, c'est normalement l'utilisateur qui contacte et demande aux ressources, selon l'avis des ordonnanceurs, d'effectuer du travail (par exemple une opération algébrique ou la résolution d'un problème donné) : c'est ce que nous appelons la technique de *push*. Dans ce cas précis, la topologie de la grille revêt une forme plus particulière puisqu'on suppose que c'est un réseau Très Haut Débit (voire le réseau Vraiment Très Haut Débit VTHD[9]) qui interconnecte les différentes grilles la composant. Les travaux effectués lors de cette thèse se focalisent plus particulièrement sur ce type de grilles,
- **Global Grid** pour désigner une grille dont le degré d'abstraction des ressources est encore au–dessus de ce que nous avons présenté jusqu'ici : l'ensemble des ressources constitue un super–calculateur virtuel sur lequel l'utilisateur soumet ses applications utilisant des bibliothèques de fonctions spécifiquement conçues. On peut citer par exemple l'emploi des outils de Globus [Pro] que nous présentons plus en détail dans la section 2.5, comme la bibliothèque MPICH–G [GLDS96], la version Globus de l'implantation MPICH [GLDS96] de MPI [MPI] (*Message Passing Interface*) qui permet l'exécution de programmes MPI non modifiés sur des architectures hétéro-

[9] `http://www.vthd.org/`

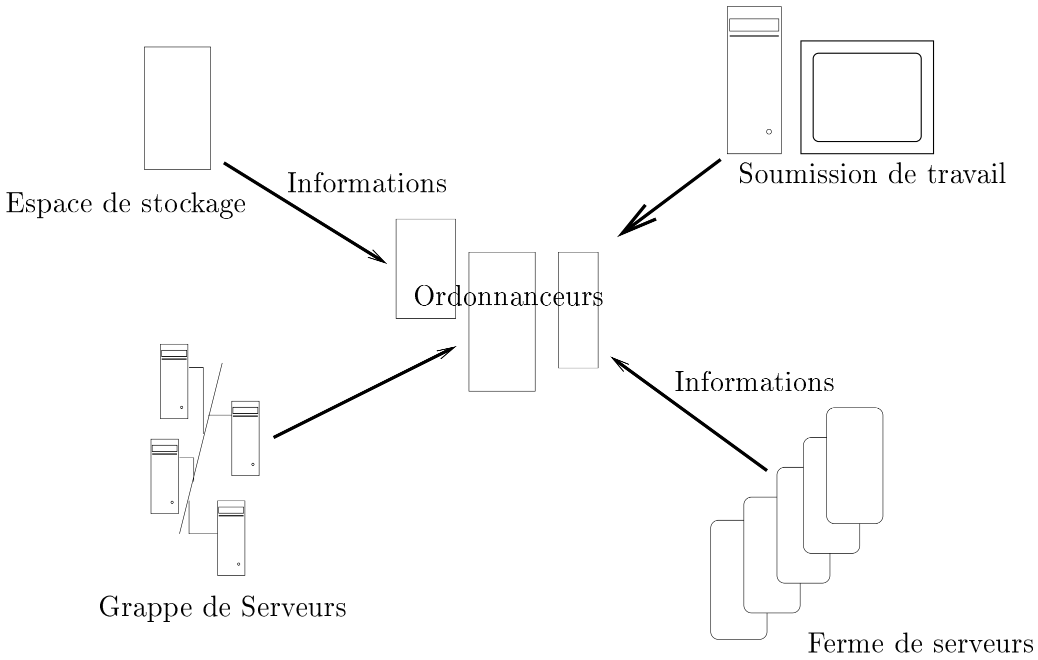


FIG. 2.3 – Server Grid

gènes.

On peut citer des exemples de plate-formes de grid–computing en production comme DOE Science Grid[10], NPACI Grid[11], TeraGrid[12] aux États-Unis, les projets européens EGEE[13] ou griphyn[14] ou encore le très récent projet Mammouth[15] au Canada.

2.3 Exploitation des grilles : présentation des outils et environnements

L'objectif de la grille est le partage des ressources entre les différents membres d'une ou plusieurs communautés afin de profiter de plus de matériel sans pour autant l'acquérir et de faciliter son usage distant. Elle permet l'introduction de plus de parallélisme dans les applications afin de minimiser le coût total en distribuant ses différentes composantes. En résumé, l'utilisateur doit avoir l'impression de disposer d'un méta-ordinateur.

Cependant, il faut encore établir des modèles de programmation pour la grille de la même manière qu'ont émergé des modèles pour programmer des applications paral-

[10] http://www.doesciencegrid.org/
[11] http://www.npaci.grid.edu/
[12] http://www.teragrid.org/
[13] http://egee-intranet.web.cern.ch/egee-intranet/gateway.html
[14] http://www.griphyn.org
[15] http://salle.css.usherbrooke.ca/

lèles lorsque seules les grappes ou les machines parallèles étaient disponibles (comme par exemple le modèle à passage de messages (ou *message passing*) avec MPI [MPI] et PVM [PVM]). La volatilité des ressources, c'est–à–dire la dynamicité de la grille, l'hétérogénéité des moyens impliqués, ou encore le passage à l'échelle de l'implantation des applications, font qu'il est difficile d'appliquer à la grille les modèles conçus jusqu'à présent tout en espérant conserver des performances acceptables. Par exemple, le fait que la grille soit composée de multiples parties qui ne sont pas nécessairement gérées par une même administration impose des contraintes de sécurité incontournables. Tout doit être repensé ou adapté.

Ainsi, un utilisateur pourra utiliser des langages orientés objets comme dans Légion [GW97, Hum03] ou l'approche orientée composants [Pro99]. De même, il pourra aussi réutiliser les programmes parallèles MPI existants, simplement en recompilant avec des bibliothèques de passage de messages comme MPICH-G2 [KTF03] ou PACX-MPI [KKM+03], mais ce sera alors souvent au détriment des performances. Des boîtes à outils comme le Globus Toolkit [FK97, FK98] existent encore, rendant possible l'implantation de son application de sorte qu'elle s'auto–adapte aux ressources disponibles comme il est expliqué dans [IO98].

Cependant, on ne peut concevoir à long terme que l'utilisateur doive continuellement fournir un travail supplémentaire de recherche, d'implantation, afin d'adapter son algorithme à la grille dont les ressources évoluent au cours du temps. Il se trouve que bon nombre d'utilisateurs potentiels de la grille ne sont pas des experts des applications distribuées, ni même informaticiens, mais plutôt des scientifiques qui ont besoin d'une grande capacité de calcul.

En conséquence, l'utilisateur préférera souvent être déchargé de la surveillance des ressources, ce qui sous–entend les connaître, et de placer au mieux les tâches composant son application sur les ressources de la grille afin de profiter d'un maximum de performances. Il faut dans ce cas une approche telle l'approche ASP (*Application Service Provider*) des services fondés sur le Web comme par exemple iMW [GG00], WebFlow [HAF00], Javelin [NPRC00, NBK+00] ou Punch [KJF98], où les requêtes d'un client concernent certains services hébergés chez un fournisseur : le client n'a aucune connaissance des moyens requis et mis en œuvre pour affecter et effectuer ses requêtes. Seul lui importe le résultat. On trouve une telle approche sous la forme des environnements de calcul global propres aux *Global Grids* ou des environnements de soumission de problèmes (PSE, *Problem Solving Environment*) ou des NES (Network Enable Server) plus adaptés aux *Server Grids*. Le modèle de calcul est alors le *parallélisme de tâches* (utilisant des appels asynchrones), c'est–à–dire que l'application initiale est découpée en tâches de communication et de calcul selon un modèle structurel (section 3.3.1). Cependant, les serveurs de calcul pouvant être parallèles, le modèle s'oriente alors vers un parallélisme mixte [Ram96, DS04], où une tâche de calcul peut s'exécuter sur plusieurs processeurs.

Le schéma de la figure 2.4 présente l'architecture en couches, telle qu'on peut la représenter, des composants et services donnés autour d'un intergiciel (ou *middleware*) de grille. Un intergiciel (à la couche (3)) est un *PSE* qui va se charger, grâce aux services présents dans la couche (2), de gérer les ressources de la couche (1), de les monitorer et de décider où les requêtes, formulées par l'application de la couche (4), devront être exécutées. Naturellement, ces décisions s'appuient partiellement sur les contraintes de capacités et de performances entre ce qui est demandé au niveau applicatif et ce qui peut être fourni par l'infrastructure. Ainsi, un intergiciel doit à moyen et long termes devenir l'outil de gestion de la grille et répondre au mieux à chaque utilisateur en fonction de ses besoins propres.

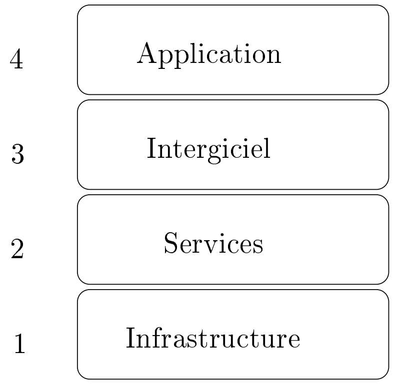


FIG. 2.4 – Une architecture en couches

Pour le moment, certains intergiciels conviennent mieux que d'autres en fonction du type des besoins des requêtes, de la composition et de la taille de la grille. Ainsi, on trouve des environnements complets multi–utilisateurs et multi–applications pour les *Global Grids* comme par exemple Condor–G [ELvD+96, TTL03], Nimrod–G [ASGH95, BAG00, BGA00]. D'autres projets plus récents comme Alchemy[16] (écrit pour le langage .NET et qui ne peut être utilisé que pour les applications .NET), OurGrid [Our] ou encore XtremWeb [CDF+04], sont encore plus adaptés à des grilles où la quantité instantanée de travail produite, qu'on exprime généralement en *flop* (unité indiquant le nombre d'opérations arithmétiques effectuées sur des nombres flottants) par seconde, peut devenir immense mais est susceptible de beaucoup fluctuer à cause de ressources particulièrement volatiles. On trouve aussi, dans le même contexte, avec le projet P3 (Personal Power Plant) [Plab], une tentative de profiter au mieux des technologies *Global Grid* et *Desktop Grid*. En ce qui concerne les *Server Grids*, des projets de NES tout aussi complets existent tels NetSolve [CD96, ACD02], Ninf [NSS99] ou DIET [CDF+02]. Ils permettent aux utilisateurs de se focaliser sur leurs applications en leur procurant des services distants invoqués par des appels à travers le réseau.

[16] `http://www.alchemi.net/`

2.4 Fonctionnalités et problématiques

Nous nous intéressons ici plus particulièrement aux services de base auxquels doit répondre un environnement à finalités calculatoires, c'est–à–dire du type global grid ou server grid. Nous introduisons les problèmes qu'il doit maîtriser lors de son déploiement et certains des moyens utilisés pour y parvenir.

L'un des premiers critères est **l'installation des différentes composantes de l'environnement** qui doit être évidemment la plus facile possible et ne doit préférablement pas nécessiter de droits particuliers comme ceux d'administrateur système, puisque le système peut être amené à gérer des ressources de domaines administratifs différents.

De même, les fonctions de soumission des requêtes qui composent **l'interface client**, doivent être claires et proposer différents modes d'appels (synchrones et asynchrones) dans des langages évolués comme C, Fortran voire Java ou avec des mnémoniques incorporées dans des logiciels de calculs mathématiques comme Scilab, Matlab ou Mathématica, ou encore par fichier de script : l'objectif est d'être le plus générique possible.

Les requêtes sont effectuées grâce à des invocations de routine à distance (RPC, *Remote Procedure Call*), XML–RPC (qui permet d'encapsuler les données à échanger dans un protocole normalisé reconnu) ou RMI (java Remote Method Invocation). Il n'existe pas de standard encore défini pour ce qui concerne la gestion des données avec les messages de type RPC. En revanche, les appels de type RMI permettent d'appeler des méthodes sur des objets avec la possibilité de passer les objets ou des références lors de l'appel. Aussi, le type XML–RPC est un moyen de fournir, de la même manière, des données en les encapsulant avec les appels et donc, intrinsèquement, de les gérer.

Afin d'assurer la «complète» transparence de l'accès aux ressources de la grille, l'environnement doit aussi s'occuper des services comme :

- **la gestion des ressources** qui concerne prioritairement l'enregistrement des ressources et de leurs capacités dans une base de données, la découverte ou le déploiement des services que pourront offrir les serveurs, l'ajout de nouveaux serveurs ou leur retrait et en conséquence l'ajout, le retrait ou la migration de nouveaux services,
- **le monitoring** des ressources via des senseurs comme ceux de NWS [WSH99], installés à différents endroits *stratégiques* sur la plate-forme afin d'avoir des informations les plus précises et pertinentes possibles sur le comportement de chaque ressource (charge instantanée, charge moyenne, espace mémoire restant, espace disque restant, etc.) sans les altérer par la prise proprement dite des mesures (sections 3.3 et 3.4),
- **la prédiction de performances** qui découle généralement des données enregistrées par le monitoring et de l'identification des requêtes reçues. Nous en parlerons plus en détail dans la section 3.5,
- **l'ordonnancement et le placement** des requêtes, c'est–à–dire le choix des ressources que l'environnement va mettre en œuvre pour un travail demandé. C'est ici que se situe l'objectif de notre travail, et nous introduirons ce domaine dans le

chapitre 4,
– **les mécanismes de soumission** consistant entre autres en l'identification des problèmes demandés et la gestion des paramètres d'entrée et de sortie correspondantes pour une cohésion totale.

Le client peut avoir besoin indirectement d'autres services. Ainsi, en fonction de la grille utilisée, l'environnement se doit de proposer **des mécanismes de tolérance aux pannes** : pour chacun des composants de l'environnement par leur réplication par exemple, mais cela peut vite devenir difficile à gérer ; pour chacun des services offerts grâce à des techniques de *checkpoints* incorporées dans les applications ou dans les messages (usage de l'un des protocoles de MPICH–V [ifVR] par exemple), grâce à de la réplication de donnée ou encore avec de la duplication d'exécution.

De plus, certaines utilisations nécessitent la confidentialité et les droits d'accès aux données et aux logiciels. **Des mécanismes de sécurité** doivent donc être pourvus. Le degré de sécurité pourra varier en fonction du type d'environnement et du domaine sur lequel il est utilisé. Un environnement de résolution de problèmes devra gérer l'authentification de tous les composants, des applications et des utilisateurs du système afin de savoir qui pourra profiter de l'environnement et quelles applications ont le droit d'être exécutées voire migrées. Ensuite, parce que tous les utilisateurs ne pourront bénéficier de tous les services ou de toutes les ressources, par exemple à cause de leur coût d'exploitation, des mécanismes doivent assurer les permissions accordées à chaque client par rapport aux ressources et aux applications, et à chaque application par rapport à chaque ressource. Enfin, les données transférées entre les différentes composantes du système doivent être chiffrées afin d'éviter qu'elles ne soient interceptées ou altérées. Il est en revanche plus difficile de garantir, surtout avec les approches pair à pair sur les Desktop Grid, qu'il n'y ait pas espionnage des données directement sur les ressources de calcul. Les mécanismes mis en œuvre peuvent par exemple utiliser le protocole de communication Kerberos[17], fondé sur un chiffrement symétrique dont la sécurité est modulable, ou la bibliothèque OpenSSL[18] capable d'utiliser des chiffrements symétriques et asymétriques pour les messages échangés.

Il existe enfin de nombreux moyens de communication entre les composantes du système parmi lesquels on peut citer l'utilisation des sockets UNIX qui est très courante, de CORBA (*Common Object Request Broker Architecture*) afin de faire communiquer les objets distribués ou encore de JXTA[19] pour une approche pair à pair ou de MPICH–V [ifVR] pour utiliser des protocoles avec tolérance aux pannes.

[17] `http://web.mit.edu/kerberos/www/`
[18] `http://www.openssl.org/` et `http://wp.netscape.com/eng/ssl3/` pour le draft IETF
[19] `http://www.jxta.org`

2.5 Quelques outils et environnements pour grilles de calcul

Dans cette section, nous présentons plus en détail certaines solutions utilisées actuellement en insistant sur les aspects et les mécanismes mis en jeu que nous venons de présenter. Puisque chaque solution est plus particulièrement adaptée à un type de grilles (section 2.2), nous donnons aussi les domaines applicatifs concernés. Remarquons qu'une synthèse couvrant d'autres systèmes que ceux que nous présentons ici peut être trouvée dans [KBM01].

2.5.1 Système d'exploitation distribué : openMosix

Tout comme le projet OpenSSI[20] ou le projet Kerrighed[21] développé à l'IRISA[22], OpenMosix[23] est une solution qui mérite d'être mentionnée puisque sans entrer directement dans les critères que nous avons présentés dans la section 2.4, elle est applicable à la notion la plus réduite possible d'une grille : la grappe.

OpenMosix est une extension du noyau Linux, pour une grappe dont les machines hétérogènes reposent sur la même version patchée du noyau (on parle alors de *single image*). Cette solution n'est donc adaptée que pour des machines dans une même organisation. OpenMosix se veut capable d'interfacer les ressources de la grappe comme si elles composaient un super–calculateur.

Il n'y a pas d'interface client puisque l'utilisateur exécute ses programmes comme sur sa propre station de travail. En effet, toutes les extensions openMosix sont directement incorporées aux noyaux. Les applications bénéficient automatiquement et de façon transparente de l'ensemble des ressources de la grappe. Le système gère l'ordonnancement des tâches exécutées et est capable de migrer un processus d'un nœud plus chargé qu'un autre pour faire de l'équilibrage de charge [HBD97]. Il cherche généralement à optimiser l'allocation des ressources. La solution embarque une technique de découverte des ressources : les capacités d'un nouveau nœud peuvent ainsi commencer à être exploitées dès son ajout à la plate–forme. La solution est extensible linéairement sur plus d'une centaine de nœuds qui peuvent être des machines SMP (*Symmetric Multi–Processors*).

2.5.2 Globus Toolkit

En inventant le concept de grille, I. Foster et C. Kesselman commençaient le projet Globus [FK97, FK98] pour montrer que l'exploitation de la grille était réalisable. Globus

[20] `http://www.openssi.org/`
[21] `http://www.kerrighed.org/`
[22] `http://www.irisa.fr/`
[23] `http://openmosix.sourceforge.net/`

n'est pas un environnement en soi. Ce précurseur de la grille constitue la référence dans le domaine et les environnements peuvent utiliser ses services ou lui les leurs. Il est livré sous forme d'une boîte à outils, d'où son nom : le *Globus Toolkit* [Pro]. D'autres projets visant les mêmes objectifs existent comme le projet européen GridLab[24].

Globus se focalise sur les problèmes d'architecture : il fournit des implantations de protocoles de grille et des APIs (*Interface de Programmation d'Application*) pour infrastructure de base. C'est un ensemble de services qui ne constitue pas une solution intégrée. L'utilisateur doit entre autres faire le travail de surveillance des ressources et de soumission des travaux en utilisant les composants dont il a besoin.

Nous présentons à la figure 2.5 l'architecture en couches de Globus dans sa version 2. Nous observons quatre couches principales : (4) les applications utilisateurs utilisant les différentes fonctionnalités de Globus, (3) les services de haut niveau qui viennent se greffer à la boîte à outils, (2) les services principaux que propose Globus et (1) les services locaux propres à chaque ressource.

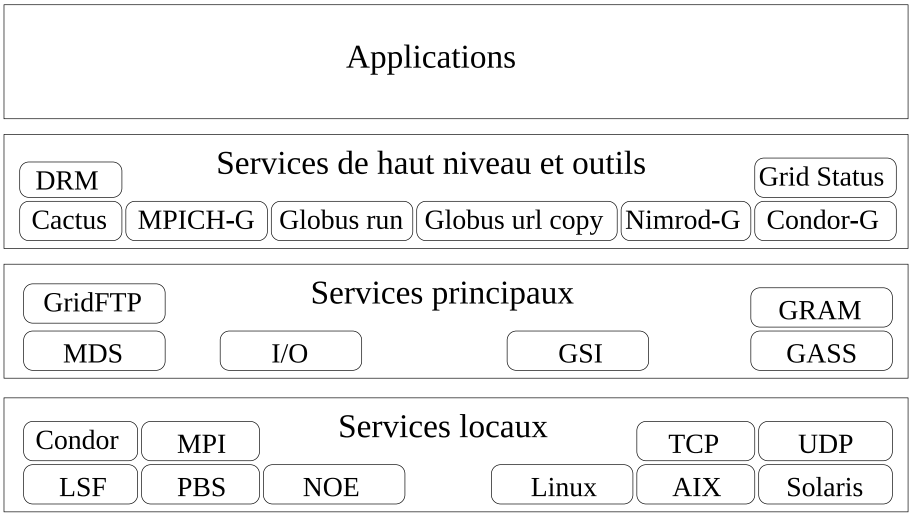


Fig. 2.5 – Architecture en couches de Globus (version 2)

Les services de haut niveau de la couche (3) sont généralement des bibliothèques ou des environnements modifiés pour la grille. Par exemple, MPICH–G est une implantation du standard MPI version 1.1 se différenciant de MPICH [GLDS96] car adaptée à la grille. Il utilise le composant Nexus de Globus qui gère toutes les communications (Nexus supporte plusieurs protocoles et la conversion automatique des données entre des machines d'architecture différente) ainsi que les services offerts pour l'exécution des tâches et la sécurité. De la même manière, Condor–G [TTL03] est une version modifiée de l'environnement Condor[25] dont nous parlerons plus loin (section 2.5.3). Il utilise les mécanismes de soumissions, de communication et sécurité de Globus. Cactus[26][ABG$^+$00] est un environ-

[24] `http://www.gridlab.org/`
[25] `http://www.cs.wisc.edu/condor/`
[26] `http://www.cactuscode.org/`

nement de soumission de problèmes (*PSE*, Problem Solving Environnement) open source. Il a été initialement conçu pour résoudre les équations à dérivées partielles d'Einstein puis étendu à d'autres problèmes de physique comme la mécanique des fluides (utilisation de la bibliothèque PETSC[27], *the Portable, Extensible Toolkit for Scientific Computation*). Pour finir, le service globusrun est une infrastructure de soumissions de job sous la forme de scripts exprimés en langage de spécification de ressource (*RSL*, Resource Specification Language).

Les services de la boîte à outils Globus offrent une interface unifiée des appels aux services locaux de la couche (1). Ainsi les protocoles de communications TCP et UDP, les caractéristiques des systèmes UNIX, les systèmes de soumission par queue tels que LSF, PBS ou les environnements comme Condor propres aux machines parallèles et grappes de PCs sont directement accessibles par les composants de Globus.

Parmi les composants de la couche (3), on trouve :

- **GRAM** : le composant *Grid Resource Access Manager* est chargé de vérifier l'identité du client et sa capacité à utiliser la ressource donnée. Dans ce cas, c'est lui qui gère la création et le contrôle d'un nouveau processus pour la tâche à exécuter ou il s'interface avec les ordonnanceurs locaux (PBS, etc.) des ressources,
- **GASS** : le composant *Globus Access to Secondary Storage* simplifie le portage, la réutilisation et l'exécution sur la plate–forme gérée par Globus des applications utilisant les entrées/sorties POSIX et standard de C. Ainsi, le client pourra par exemple avoir sur son écran les données générées par son programme sur la sortie standard de façon transparente,
- **MDS** : le composant *Metacomputing Directory Service* synthétise l'information sur la grille. Il renseigne sur les ressources disponibles de la grille en gérant leur découverte grâce au module GRIP. Il donne aussi leurs statuts grâce à l'annuaire d'informations statiques et dynamiques, le module GRIS (*Grid Resource Information Service*), et aux pages jaunes gérées par le module GIIS. Tous les serveurs utilisent le protocole LDAP permettant le passage à l'échelle,
- **GSI** : le composant *Grid Security Infrastructure* s'occupe de la sécurité sur l'ensemble de la grille. Il est fondé sur du chiffrement asymétrique, les certificats X.509 et le protocole de communication SSL (*Secure Socket Layer*). Cela permet d'avoir des communications sûres entre les différents éléments de la grille, authentifiés, en étant indépendant du niveau de sécurité des organisations locales,
- **I/O** : le composant d'entrée/sortie permet l'utilisation des sockets sur la grille tout en assurant la conversion dans le bon encodage des données entre des machines d'architecture différente,
- **GridFTP** : les transferts de fichiers sont transparents pour le client grâce à ce composant qui implante une extension de FTP (avec possibilité de transfert sur ou à partir d'une entité tiers).

Notons que Globus dans sa version 3 change légèrement d'architecture : les composants

[27] `http://www-unix.mcs.anl.gov/petsc/petsc-2/`

évoluent vers des services afin se rapprocher de la norme OGSA [FKNT02] (*Open Grid Services Architecture*) tentant de faire converger les *web services* et les *grid protocols*.

2.5.3 Environnements construits sur le modèle maître–esclave

Le **modèle maître–esclave** (aussi appelé *work queue*, *task farming* ou encore *self–scheduling*) dont nous présentons le schéma à la figure 2.6 fonctionne sur le principe suivant : une machine maître dispose de multiples tâches de travail à effectuer. Quand un des esclaves n'a plus de travail à faire, il contacte le maître pour en recevoir. La technique utilisée ici est le *pull* : ce sont les esclaves qui demandent à fournir du travail.

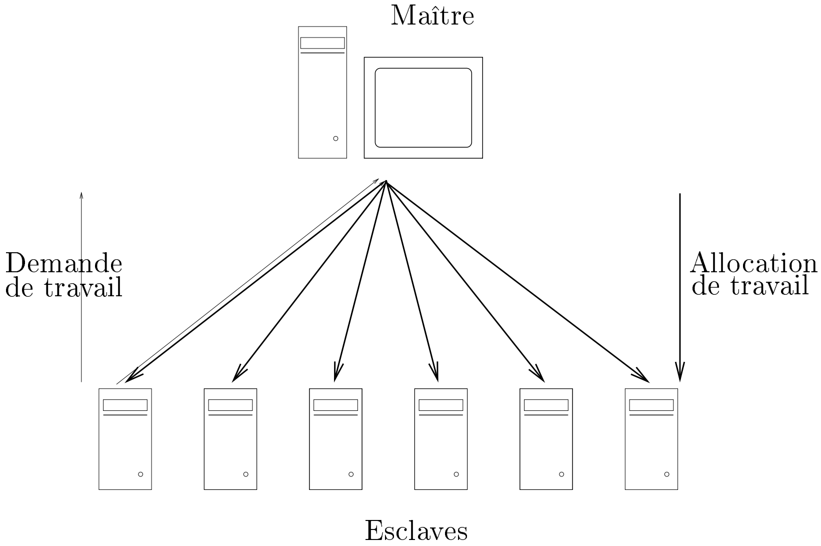


FIG. 2.6 – Modèle maître–esclave

Depuis `seti@home` qui a popularisé la grille en 1996, d'autres «environnements» sont nés tels : distributed.net[28] qui comporte majoritairement des résolutions de problèmes de cryptographie ; BOINC[29], qui permet de participer à la résolution de plusieurs projets dont `seti@home` qui a migré pour cela ses applications. Certains ne fonctionnent que sur un type de systèmes d'exploitation comme la barre de Google[30] pour les systèmes d'exploitation Microsoft et Xgrid[31] pour les machines Apple.

Condor

Le concept de DRM (*Distributed Resource Management*) fondé sur la notion de maître–esclave, est déjà présent depuis 1988 pour des ressources hétérogènes distribuées grâce au

[28] `http://www.distributed.net/`
[29] `http://boinc.berkeley.edu/`
[30] `http://toolbar.google.com/dc/offerdc.html`
[31] `http://www.apple.com/acg/xgrid/`

projet Condor [ELvD+96] développé à l'Université de Wisconsin–Madison. Il consiste à l'assignation de tâches de calcul soumises au préalable et stockées dans une queue, sur des ordinateurs inscrits pour faire profiter de leur capacité de calcul dès que leur inactivité dépasse un certain laps de temps. Le retour impromptu du propriétaire met un terme à l'exécution distante de la tâche. Pour éviter la perte de temps de calcul, des mécanismes de *préemption* fondés sur le *checkpointing* ont été mis en place. Ensuite, un travail change généralement de machine pour poursuivre son exécution.

Un travail Condor s'exécute sur une machine distante en ayant l'illusion d'être sur la machine de soumission : certains appels sont détournés (principalement ceux en relation avec les entrées/sorties). Pour cela, s'il n'y a pas de système de fichier accessible par l'ensemble des ressources (comme NFS, *Network File System*), un travail Condor doit être compilé avec la bibliothèque C modifiée livrée avec Condor.

Il existe une version adaptée[32] pour la grille qui se nomme Condor–G [TTL03]. Ses mécanismes de soumission inter–domaines sont fondés sur des outils Globus [Pro].

XtremWeb

XtremWeb est un projet développé au LRI à l'université Paris Sud. Il étudie la conception et le développement d'un environnement de calcul *global* et *desktop grid*. Son objectif est de fournir la gestion sécurisée d'une plate–forme de calcul intensif dont les ressources sont très volatiles et peuvent à un instant donné être très nombreuses.

Le système repose ici sur une architecture trois tiers : les *Workers* sont les entités qui exécutent les tâches, les *Clients* soumettent leurs requêtes à un *Coordinateur* qui peut être répliqué. Lors du déploiement de l'environnement, les programmes Client et Worker sont chargés à partir d'un serveur sur les machines hôtes. Une liste des Coordinateurs disponibles qu'ils peuvent contacter leur est alors fournie.

Tous les éléments du système peuvent communiquer, y compris derrière des pare–feux, puisque les communications sont toujours laissées à la charge des Clients ou des Workers en direction du Coordinateur. La résolution de problèmes proprement dite prend la forme d'appels RMI ou XML–RPC.

Les interfaces utilisateur comptent une interface de type système de traitement par lot (ligne de commande ou script) et une API Java.

Les **Coordinateurs** sont les responsables de la bonne **gestion des ressources** de l'environnement. Le Client fournit au Coordinateur le(s) programme(s) qu'il souhaite voir utiliser sur la grille et qui sera stocké dans son cache. Les Workers les renseignent sur leurs caractéristiques statiques et dynamiques comme la description matérielle, la charge processeur, etc. ainsi que sur les noms des applications dont ils disposent déjà du code localement.

[32] `http://www.cs.wisc.edu/condor/condorg/versusG.html`

L'ordonnancement consiste à placer les tâches sur les Workers, lorsque ceux–ci demandent du travail. Cela se passe en deux étapes : il faut tout d'abord sélectionner quelles tâches vont être exécutées puis les placer. La sélection des tâches s'effectue selon une priorité donnée aux applications en attribuant des proportions de tâches à accomplir par application. Puis, le Coordinateur sélectionne la première tâche capable d'y être exécutée.

La tolérance aux pannes est assurée pour tous les composants du système : Workers, Coordinateur et mobilité des Clients qui enregistrent localement les messages qu'ils émettent. Le Coordinateur est répliqué. Enfin, chacun dispose de bornes temporelles pour détecter d'éventuelles anomalies : un message est envoyé comme une pulsation des Clients et Workers vers le Coordinateur.

En ce qui concerne **la sécurité**, de nombreux mécanismes sont mis en œuvre pour protéger les Workers d'utilisateurs malicieux (et réciproquement) et les données d'espions internes ou externes au système. Ainsi, pour réaliser une action, tous les éléments impliqués doivent pouvoir authentifier chacun des autres grâce à des *certificats*. Les attaques d'un Worker par un Client avec une application authentifiée sont gérées en utilisant le confinement d'exécution (Sanbboxing), via l'outil *Ptrace*.

ConFiit

L'intergiciel ConFiit (*Computation Over a Network with Finite number of Irregular Independant Tasks*) est développé en Java à l'Université de Reims. Il utilise une approche pair à pair totalement distribuée pour résoudre des applications Fiit [Kra99], dont les tâches sont indépendantes et dont la durée n'est pas prévisible.

Chaque procédure RPC correspond à un appel ConFiit réalisé par des communications XML–RPC.

L'architecture retenue pour gérer la plate–forme est un anneau logique sur l'ensemble des machines participantes, aussi appelées *nœud*. On trouve sur chacun des nœuds trois *threads* principaux : un gestionnaire de topologie et de communication, un gestionnaire de tâches et un ou plusieurs solveurs dédiés à un problème. Chaque nœud connaît son prédécesseur et son successeur, et communique grâce au jeton qui contient entre autres la liste des tâches qu'il reste à effectuer.

Une machine désirant entrer dans un calcul existant le demande auprès d'un nœud ConFiit. Il reçoit l'ensemble des informations à disposition pour commencer un calcul immédiatement. Une machine choisit une tâche à exécuter de façon aléatoire. Dans ce contexte, il n'y a donc pas de prédiction de performances ni d'ordonnancement et il est donc possible que le calcul commencé soit déjà en court sur un autre nœud.

Des mécanismes de tolérance aux pannes assurent une k–tolérance pour reconstruire un anneau fonctionnel quand k nœuds successifs s'arrêtent dans un anneau de taille n ($k \leq n$). De plus, à chaque envoi de jeton, un délai de garde est lancé. Ainsi lorsque le nœud qui dispose du jeton tombe en panne, un nouveau jeton peut être réémis. Cependant, comme plusieurs machines peuvent détecter la panne, plusieurs jetons peuvent être émis. Un mécanisme établi sur l'algorithme des estampilles de Lamport [Lam78] permet d'éliminer les jetons redondants.

2.5.4 Environnements construits sur le modèle client–agent–serveur : le GridRPC

Dans un souci de normalisation de l'API des appels de procédures à distance (RPC) utilisés dans les Nes, le Global Grid Forum (GGF), une organisation créée en 1999 afin de promouvoir les travaux dans le domaine des grilles, a confié au groupe de travail GridRPC–WG le soin de définir des standards architecturaux et de soumissions dans le cadre du projet APME (*Applications, Programming Model and Environments*). L'approche résultante, expliquée dans les travaux [NMS+03, Phi04] et fonctionnant sur le mode *push*, est appelée **GridRPC**.

Elle propose pour architecture le modèle de référence illustré dans la figure 2.7 tirée de [Phi04] et fonctionne de la façon suivante : dans la phase initialisation de l'environnement, chaque serveur déclare ses services auprès de l'agent (appelé aussi *Registry* ou encore *RMS, Resource Management System*). Lorsqu'un utilisateur requiert un service, il contacte l'agent pour passer sa requête au système. Celui-ci lui communique en retour un identifiant donnant accès à l'application recherchée. Grâce à cet identifiant, le client contacte le serveur qui lui retournera les résultats du service exécuté.

Cette approche définit aussi le comportement de certains appels de fonction et leur contrôle, l'existence des modes synchrone et asynchrone, les retours d'erreur. En revanche, la manipulation des données comme *la persistance*, c'est–à–dire la capacité de laisser des résultats sur un serveur, et leur typage sont toujours en étude.

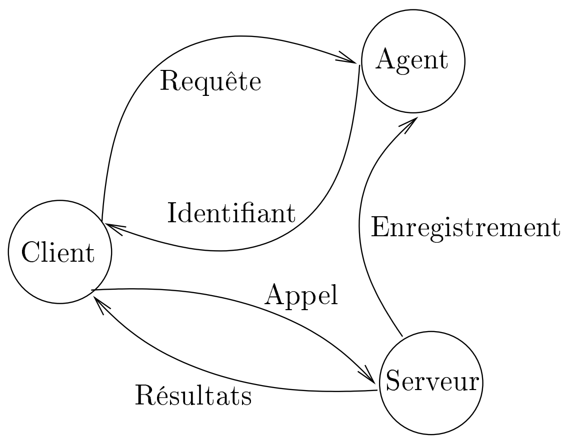


Fig. 2.7 – Modèle de base GridRPC

NetSolve

NetSolve [CD96] est un environnement Server Grid codé en C et développé à l'Université du Tennesse, Knoxville. Les membres de l'équipe de NetSolve participent activement à la formulation des standards du *grid computing* comme les GridRPC que l'intergiciel utilise. Notons que nous avons travaillé en collaboration avec l'équipe de NetSolve. Emmanuel Jeannot a apporté deux modifications importantes à NetSolve : la possibilité d'une compression à la volée et adaptative aux conditions du réseau grâce à l'utilisation de la bibliothèque AdOC[33] et la possibilité d'offrir la persistance des données aux utilisateurs [DJ01]. Nous avons utilisé NetSolve pour valider les travaux effectués au cours de cette thèse et les améliorations apportées sont incluses dans NetSolve depuis la version officielle 2.0. Les mécanismes implantés sont en particulier utilisés par Visperf [LDR02], un outil de monitoring développé pour administrer des plate–formes de calcul où NetSolve est déployé.

Dans l'approche NetSolve, le déploiement de l'environnement et les mécanismes de soumission des requêtes correspondent à la norme GridRPC présentée avant. Un aperçu des appels utilisés lors d'une session de soumission est présenté dans la section 6.3.4.

L'interface client propose des appels synchrones et asynchrones dans des langages variés comme C, Fortran, Matlab, Scilab et même Mathématica ou Octave[34].

Une seule entité coordonne l'ensemble de l'environnement : **l'agent**. Il s'occupe des informations sur le système afin d'assurer la **gestion des ressources**. Après l'enregistrement des serveurs de calcul, il contient (1) des informations statiques : les noms des serveurs enregistrés, leur performance de calcul crête (grâce à l'exécution des benchmarks LINPACK) et les services qu'ils proposent (ainsi que leur monôme de plus haut degré du polynôme de complexité pour la prédiction de performances) ; (2) des informations dynamiques : mesures de bandes passantes et de latences, charge processeur des serveurs de calcul. L'ajout de serveur se fait très simplement par exécution du code approprié sur la ressource. Les serveurs exécutent les bibliothèques BLAS, LAPACK, ScaLAPACK ou PETSc. En revanche, pour qu'un nouveau service soit reconnu par le serveur, il doit être d'abord décrit dans un fichier PDF (*Problem Description File*) contenant le nombre et le type de ses entrées et son polynôme de complexité. Si dans sa version 1.4.1, le serveur devait être stoppé puis relancé pour que le nouveau service soit pris en compte, il suffit maintenant d'opérer une mise à jour par l'intermédiaire d'une application fournie.

L'actualisation des informations dynamiques des différentes ressources de la plate–forme s'effectue grâce à des moniteurs installés sur chaque serveur de calcul. Ils reportent la moyenne sur la dernière minute de la charge constatée, valeur obtenue grâce à la commande UNIX `uptime`. NetSolve utilise par défaut ses propres senseurs mais peut aussi utiliser ceux de NWS[35] que nous présentons brièvement dans la section 3.4.2. **Le module**

[33] `http://www.loria.fr/~ejeannot/adoc/`
[34] `http://www.octave.org`
[35] `http://nws.cs.ucsb.edu/`

de prédiction de performances intégré à l'agent calcule à chaque requête une estimation du temps pris par chacun des serveurs disposant du problème à résoudre. Ce calcul prend en compte les informations statiques à disposition (taille des données, performances crêtes) et les informations dynamiques (quote part processeur du serveur qui sera *a priori* donnée à l'application pendant sa résolution, paramètres réseau).

Le module d'ordonnancement est chargé ensuite de classer la liste des serveurs selon les informations délivrées par le module de prédiction de performances. L'heuristique d'ordonnancement utilisée est MCT *Minimum Completion Time* [MAS+99] que nous présentons plus en détail dans la section 4.4.3. La liste triée est envoyée au client qui dispose alors de l'identité d'au moins un serveur pour appeler son service.

Envoyer une liste de serveur pour chaque requête est l'un des **mécanismes de tolérance aux pannes** mis en œuvre dans l'agent de NetSolve : si un serveur ne peut répondre (trop chargé, indisponibilité à cause d'une panne réseau ou machine), le client pourra contacter le serveur suivant jusqu'à en trouver un disponible. De plus, des messages sont envoyés périodiquement entre les serveurs et l'agent. Si le message ne parvient pas à l'agent, le serveur et ses services sont retirés de la base de données de l'agent. Dans le cas où l'agent s'écroule, des mécanismes de reprise de contacts sont intégrés aux serveurs qui font la démarche de s'enregistrer auprès d'un nouvel agent sur le même hôte dès qu'il est mis en place.

Depuis sa version 2.0, NetSolve dispose de **mécanismes de sécurité** établis sur Kerberos.

On notera qu'il est entre autres possible d'interfacer NetSolve sur Condor–G.

Diet

Le projet Diet (*Distributed Interactive Engineering Toolbox*) est développé à Lyon dans le cadre des projets GASP[36] du Réseau National des Technologies Logicielles (*RNTL*) et ASP[37] de l'ACI Grid[38]. L'équipe de Diet participe à l'élaboration des standards du *grid computing* comme la formulation des GridRPC ou des normes sur la gestion des données (communication, persistance, etc.). L'objectif de Diet est l'étude de nouvelles approches de types Nes.

Fondées sur l'observation des expériences des autres projets, de nouvelles architectures sont proposées afin d'assurer une meilleure scalabilité à l'ensemble de l'environnement. Ainsi, l'agent unique de NetSolve est remplacé par une hiérarchie d'agents afin d'en améliorer les performances en y ajoutant de surcroît une meilleure tolérance aux pannes. En effet, dans le cas de NetSolve, lorsque le nombre de requêtes envoyées à l'agent devient important, il devient un goulet d'étranglement. Le but est donc ici de répartir la charge

[36] `http://graal.ens--lyon.fr/~gasp/`
[37] `http://graal.ens--lyon.fr/~gridasp/`
[38] `http://www-sop.inria.fr/aci/grid/public/acigrid.htm`

entre les différents agents. De plus, si la machine hébergeant l'agent de NetSolve tombe en panne, le système tombe à son tour. Dans DIET, les autres agents s'organisent alors pour le remplacer. On remarquera aussi que si DIET instancie une variante du modèle GridRPC, il existe aussi un prototype DIET$_{JXTA}$ [CD04] disposant d'une approche pair à pair que nous ne détaillerons pas ici.

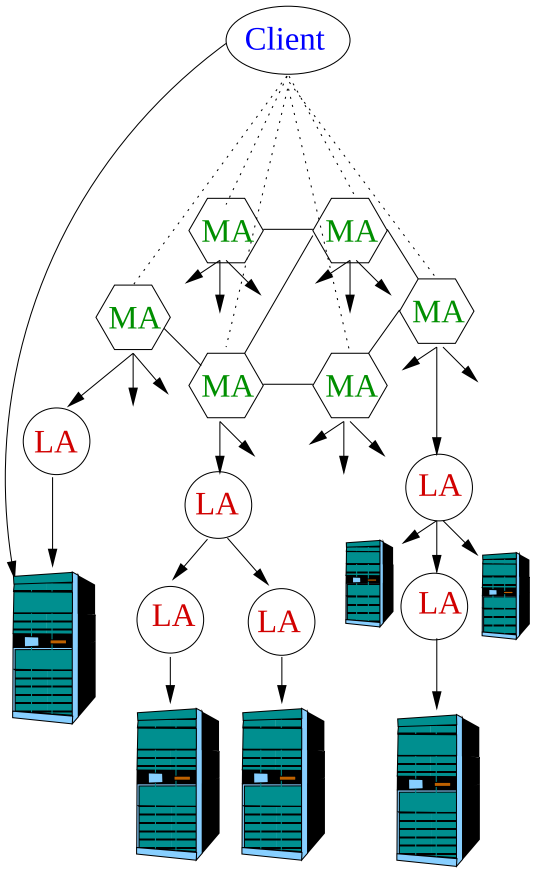


FIG. 2.8 – Vue générale de DIET

Les communications de l'environnement DIET reposent sur CORBA qui propose une interface de haut niveau pour la construction d'applications distribuées.

L'interface client propose des API contenant des appels synchrones et asynchrones dans les langages C, C++ et Scilab.

Une hiérarchie d'agents coordonne l'ensemble de l'environnement. Comme il est indiqué dans le schéma de la figure 2.8, il existe deux types d'agent : Les MA (*Master Agent*) et les LA (*Leader Agent*). Un MA est directement relié aux clients dont il reçoit les requêtes. Pour choisir les ressources à utiliser, il dispose des mêmes informations que les LA mais a une vue globale de tous les problèmes capables d'être résolus et de toutes les données distribuées sur la plate–forme. Un LA est le lien entre les MA et les ressources. Il diffuse les requêtes et les informations tout en tenant à jour une liste des requêtes en cours de traitement ainsi que les informations sur les données. De la même manière que

NetSolve dans sa version 1.4.1, il faut définir les services que sera capable de proposer chaque serveur, encapsulé dans un SeD (*Server Daemon*), avant de le lancer sur la ressource. Le SeD est le point d'entrée d'un serveur de calcul et tient à jour en particulier toutes les informations concernant les ressources dont il a la charge comme les problèmes pouvant être résolus mais aussi les charges processeur, mémoire, disque, etc.

La prédiction de performances est établie par l'outil FAST [Qui02, Qui03] que nous présenterons dans la section 3.5.2.

Dans DIET, l'utilisation d'une hiérarchie d'agents augmente le passage à l'échelle de l'environnement mais complexifie **l'ordonnancement**. Pour l'instant, une variante de MCT [MAS+99] est utilisée : la requête descend sur l'arborescence dont le MA a la charge. Chaque Sed renseigne son LA qui choisit ainsi les ressources les mieux adaptées et dont il envoie la liste à son père. D'autres stratégies sont en cours d'étude. Remarquons qu'actuellement pour des raisons de performance (coût de communication, gestion d'un délai de réponse), un MA ne contacte un autre MA pour lui transmettre le problème que s'il ne dispose pas lui–même de ressources capables de le gérer.

Comme nous l'avons vu, **les mécanismes de tolérance aux pannes** des agents sont inclus dans l'architecture même de l'environnement.

En ce qui concerne **les mécanismes de sécurité**, ils reposent sur l'implantation open source des services de sécurité de CORBA, MicoSec[39] [LS02].

2.6 Conclusion

Nous avons présenté dans ce chapitre la genèse du concept de la grille et son utilité face à des besoins de performance croissants. Nous l'avons définie comme l'agrégat unique de ressources hétérogènes de domaines administratifs différents. Cependant, nous avons vu que la grille reste pour l'instant un concept et que l'on parle en générale de grilles plus petites. Nous les avons classées en quatre grandes familles selon une caractérisation fondée sur leurs enjeux et leurs domaines d'applications respectifs. Nous avons donc proposé de regrouper les grilles sous les appellations : information grid, desktop grid, server grid et global grid, et nous les avons brièvement décrites et commentées.

Parce que l'utilisateur de la grille n'est pas nécessairement un informaticien et/ou parce qu'il est plus intéressant pour lui de déléguer la gestion de la soumission de ses applications (pour des raisons de dynamicité de la grille par exemple), nous avons vu que l'accès aux ressources de la grille doit être simple et transparent. Nous avons désigné cette capacité d'abstraction des ressources de la grille par le terme de métacomputing.

Nous avons ensuite présenté l'analogie entre les modèles de programmation utilisés pour simplifier la programmation des machines parallèles et des grappes de machines ho-

[39] `http://www.micosec.org/`

mogènes, puis l'émergence d'outils pour la grille, comme des extensions des bibliothèques à passage de messages, des approches de programmation orientées composants ou des bibliothèques de gestion dynamique de grilles.

Dans le contexte des grilles de calculs, nous avons introduit l'approche des environnements de résolution de problèmes reposant sur le parallélisme de tâche des applications à soumettre. Après avoir présenté l'architecture globale des mécanismes mis en jeu, nous avons détaillé les fonctionnalités dont devait disposer un environnement de soumission de travail, qu'elles aient trait à la communication des données ou à la gestion de la plate–forme. Nous avons souligné certains problèmes (gestion des données, des ressources, tolérance aux pannes ou sécurité) que posait le déploiement d'un tel système sur un ensemble de ressources hétérogènes distribuées géographiquement et nous avons indiqué quelques solutions utilisées.

Nous avons sélectionné les outils présentés dans ce chapitre pour leurs caractéristiques et leur pertinence dans le domaine (originalité, référence, etc.). Nous avons présenté les modèles utilisés (modes push et pull, modèles maître–esclave et GridRPC) et leurs fonctionnalités en établissant le parallèle entre les motivations du projet et les fonctionnalités théoriquement disponibles vues précédemment.

L'approche GridRPC donne un modèle de programmation simple et n'impose pas à l'utilisateur une quelconque action de surveillance sur les ressources. La soumission et l'exécution de ses requêtes se fait de façon transparente. Le GridRPC est donc un bon candidat pour élaborer des environnements de résolution de problèmes dans le cadre du métacomputing. Nous avons donc décidé d'orienter nos travaux vers l'ordonnancement d'une grille de calcul régie par un système fonctionnant sur le principe du modèle client–agent–serveur. Nous nous intéressons plus particulièrement ici à l'ordonnancement d'un Nes sur une grille de production du type *Server Grid*. L'environnement que nous utiliserons pour valider nos travaux sur des expérimentations réelles est NetSolve, que nous avons présenté à la section 2.5.4.

Chapitre 3

Prédiction de performances. Outils de simulation

Table des matières

3.1 Introduction

Pour utiliser les grilles efficacement, des techniques d'ordonnancement doivent être développées afin d'exploiter les caractéristiques et performances dynamiques des ressources partagées. Dans le cas contraire, la performance exploitée peut ne pas rendre compte des capacités de l'ensemble des ressources. Cependant, le comportement des algorithmes d'ordonnancement dépend de la qualité des prédictions établies par le module de prédiction de performances. En effet, pour qu'une politique de placement soit correctement appliquée, il faut que les informations du système sur lesquelles reposent les décisions soient les plus proches possible de ce qu'il se passe réellement sur la grille. Or les grilles sont par définition hétérogènes et distribuées. Il n'est donc pas aisé d'obtenir des informations valides et pertinentes du fait de leur dynamicité et des communications nécessaires à leur acheminement.

Comme nous le présentons plus loin, certaines informations sont particulièrement intéressantes et doivent être surveillées. Parmi elles, on trouve des informations statiques dont la qualité ne change pas au cours du temps et dont la mesure peut n'être effectuée qu'une unique fois. Elles concernent l'étalonnage des ressources de la plate–forme et des problèmes qui seront traités. En revanche, les informations dynamiques évoluent au cours du temps. Elles concernent l'évolution des applications et/ou des ressources de la plate–forme et doivent donc être communiquées périodiquement pour donner la possibilité d'évaluer leurs disponibilités. Connaître une approximation de l'état global de l'environnement a un coût. La mesure de ces informations et les mécanismes d'acheminement perturbent généralement l'environnement, on dit alors qu'ils sont intrusifs. C'est toutefois nécessaire pour prédire l'évolution des ressources, permettre une adéquation entre leurs capacités matérielles et les requis d'exécution et estimer les performances que l'on peut escompter sur chacune d'elles : des évaluations sur les durées des applications peuvent être ainsi effectuées et données à l'ordonnanceur pour la prise de décision. Les informations recueillies peuvent aussi être utilisées dans des émulateurs ou des simulateurs pour faire de la prédiction de performances mais aussi pour évaluer des algorithmes d'ordonnancement. Ils permettent de reproduire les mêmes conditions d'expérimentation que celles observées dans la réalité et de comparer les performances relatives des décisions prises sur la même base expérimentale.

Nous présentons dans ce chapitre, d'une part la problématique de la surveillance des capacités des ressources et d'autre part, celle de la prédiction de l'évolution des ressources et des performances escomptées sur l'exécution des tâches. Dans la section 3.2, nous introduisons certaines caractéristiques qui peuvent être observées sur les ressources de la plate–forme. Ensuite, nous exposons dans la section 3.3 et la section 3.4 les diverses informations statiques et dynamiques qu'il est possible de recueillir sur les applications et sur la plate–forme. Nous y décrivons différents moyens et outils communément utilisés pour les récolter et les acheminer vers l'ordonnanceur. La prédiction de performances est abordée dans la section 3.5 où des outils et des méthodes statistiques et probabilistes couramment employés sont étudiés. Puis nous évoquons la problématique de la simulation, la génération des plate–formes et des applications dans la section 3.6. Nous y décrivons un émulateur, MicroGrid, et le simulateur que nous avons utilisé pour une partie de la validation de nos travaux, Simgrid. Enfin, les informations et les techniques présentées n'étant pas formulées expressément pour le cadre GridRPC qui nous occupe, nous concluons et résumons la problématique au contexte client–agent–serveur d'un NES dans la section 3.7.

3.2 Caractéristiques des ressources d'une plate–forme

Puisqu'une plate–forme de calcul est dynamique, multi–utilisateurs et distribuée, la connaissance de l'état du système est importante pour optimiser sa gestion. Ceci se traduit naturellement par la synthèse des informations recueillies sur les ressources distribuées, afin d'obtenir un instantané proche de la situation de la plate–forme dans sa globalité. Nous présentons dans ce qui suit plusieurs informations permettant d'apprécier les performances et la disponibilité des ressources en indiquant leur pertinence pour un module de prédiction de performances ou pour l'ordonnanceur d'un environnement. Certaines d'entre elles n'évoluent pas au cours du temps et sont qualifiées de *statiques*, les autres sont *dynamiques*.

3.2.1 Processeur

Il est important de connaître la puissance brute de calcul d'une machine afin de les classer. Pourtant, l'opération est délicate puisque des considérations matérielles et logicielles (différence d'architecture, de système d'exploitation, etc.) sont à prendre en compte. Ainsi, la **vitesse du processeur** n'est pas une information suffisante (lorsqu'elle est disponible!) puisqu'elle ne reflète pas sa capacité de calcul. On peut utiliser des données comme les **Mflops** et les **Mips** qui sont le nombre de millions d'opérations arithmétiques que peuvent exécuter les processeurs en une seconde, sur des nombres flottants et des nombres entiers respectivement. Plus particulier à l'environnement Linux, le système calcule le nombre de **BogoMips**[40]. L'acronyme vient de la contraction de Bogus (pour «faux») et de Mips (millions d'instructions par seconde) car elle n'est qu'une *indication sur la rapidité du processeur* qui ne tient pas compte de la possible parallélisation d'exécution des instructions. Elle a cependant l'avantage d'être accessible sur tout système Linux. On utilise plus généralement la **performance crête** d'une machine, le plus souvent issue d'une mesure obtenue par l'exécution d'un benchmark (section 3.3.2).

Pour donner un certain confort à l'utilisateur d'une machine, le système qui la gère est *multitâches*, ce qui signifie qu'il donne l'illusion que plusieurs tâches sont exécutées à un instant donné alors qu'elles le sont séquentiellement à intervalles de temps très courts. Une tâche de calcul partageant la puissance d'un processeur avec un ou plusieurs autres processus nécessite donc plus de temps et se termine plus tard. Aussi, dans le cadre du métacomputing, le nombre de processeurs et leur charge sont des données capitales. **La**

[40] `http://www.tldp.org/HOWTO/BogoMips.html`

charge processeur est propre à chaque processeur[41] : elle exprime son degré d'activité en donnant le nombre de processus concurrents pour la ressource (c'est–à–dire dans l'état *running*, *runnable* ou *uninterruptable* sous Linux)[42].

Lors d'une prise de mesures, on peut ainsi observer **la charge instantanée** perçue au niveau du système. L'information est très sensible à la moindre activité comme celle des démons par exemple. Elle varie donc beaucoup et ne peut donner un aperçu réellement exploitable de la capacité de calcul de la ressource. L'information fréquemment utilisée est **la charge moyenne** constatée durant les 1, 5 ou 15 dernières minutes. Ceci se justifie par la durée des calculs généralement effectués sur ce type de plate–formes.

Pour apprécier l'état de la machine et savoir si celle–ci est sous–exploitée par exemple, d'autres informations peuvent être observées, notamment les pourcentages de temps CPU des processus passés en mode utilisateur, noyau, etc. (e.g., *user*, *nice*, *system*, *idle*). Les priorités des processus, qui influent sur leur capacité à prendre plus ou moins facilement la ressource lorsqu'ils sont en concurrence avec d'autres, sont peu ou pas utilisées car supposées identiques (la raison vient du modèle d'ordonnancement utilisé, semblable au round–robin mentionné à la section 4.2.2). Grâce à l'observation du lancement d'un processus, un capteur peut aussi fournir au module de prédiction de performances **la quote–part du temps processeur** dont disposerait un nouveau processus, s'il démarrait son exécution sur l'hôte considéré.

D'autres informations peuvent encore être exploitées comme le nombre total de processus, le nombre de ceux en train d'être exécutés, le temps passé en mode utilisateur ou mode noyau pour un processus donné ou encore le numéro du dernier processeur utilisé par la tâche dans le cas d'hôtes multiprocesseurs. Cependant, elles ne sont que très rarement exploitées, les modèles ne les prenant généralement pas en compte dans leurs estimations.

3.2.2 Mémoire

L'adéquation entre les besoins de stockage requis pour exécuter une tâche et les dispositions matérielles d'une ressource nécessite d'en prendre des mesures. Mais connaître les capacités maximales de stockage n'est pas suffisant. En effet, les programmes concurrents pour le processeur partagent aussi l'espace disponible qu'il convient donc de surveiller. Plusieurs caractéristiques peuvent être considérées : la taille de la mémoire sur les disques durs, la taille de l'espace de SWAP, la taille de la RAM ou encore la taille des caches des processeurs.

Pour une *information grid*, les capacités de stockage sur **disques** sont des informations importantes. En revanche, pour les *server grid*, la quantité de mémoire vive (la **RAM**, *Random Access Memory*) totale et disponible est une information majeure : pour espérer

[41] `http://www.samspublishing.com/articles/article.asp?p=101760&seqNum=2`
[42] http ://www.teamquest.com/resources/gunther/ldavg1.shtml

le maximum de gain sur la durée d'exécution qui est l'un des principaux critères de performance de ce type de grilles (section 4.3), les données sur lesquelles calcule une tâche doivent tenir en mémoire vive (celles en entrée, en sortie, en plus des temporaires nécessaires au calcul). Dans le cas contraire, la machine devra utiliser les mémoires disques (technique de *swap*) dont les accès, coûteux en temps, dégradent les performances. Dans un cas extrême, la tâche sera interrompue sans possibilité de reprise et aura ralenti le système en monopolisant une ressource.

D'autres informations sont encore accessibles comme **le pourcentage de mémoire utilisée**, **la taille de l'exécutable**, **la taille des données** en mémoire et de celles placées sur le disque. Eventuellement, les **défauts de pages** peuvent aussi être surveillés. En revanche, la taille des caches des processeurs est une information importante pour l'optimisation des programmes lors de leur compilation mais n'a que peu d'intérêt pour l'ordonnancement d'une grille.

3.2.3 Réseau

Puisque les ressources de la grille sont distribuées, l'exécution d'une tâche de calcul sur un serveur requiert généralement un transfert préalable des données en entrée et demande un transfert des données en sortie du problème à résoudre. Si les serveurs peuvent être dédiés à une application ou à un client, à l'aide par exemple de mécanismes de réservation, le réseau ne l'est que rarement et doit donc être supervisé.

Parmi les principales caractéristiques réseau, on trouve[43] le **nombre de paquets transitants** avec la distinction entre les paquets reçus ou émis, la **quantité de données** observée sur l'interface, le **taux d'erreur** rencontré, le **RTT** (*Round Trip Time*), c'est–à–dire le temps d'aller–retour d'un paquet ou encore le **MTU** (*Maximum transmission unit*) qui est la taille maximale d'un paquet pouvant transiter sur un réseau donné.

Cependant, aucune de ces informations n'a de réelle pertinence pour un module d'ordonnancement. Le module de prédiction de performances doit calculer et corréler l'information à un niveau applicatif. Dans notre cas, l'information utile mesurable concerne la **latence**, c'est–à–dire le temps de diffusion d'un paquet d'une machine à l'autre, et la **bande passante** qui est la quantité maximale de données pouvant transiter entre deux machines pendant un laps de temps sur un canal de communication. Nous verrons plus loin qu'avec des estimations sur les quantités de données à transmettre, des prédictions sur les durées des communications peuvent être réalisées.

[43] `http://www.gridlab.org/WorkPackages/wp-11/monitor-manual/metrics.html`

3.3 Mesures des informations statiques

Certaines caractéristiques que l'on peut observer sur les applications soumises ou sur les ressources d'une plate–forme ne changent pas au cours du temps. Ce que nous dénommons *informations statiques* n'a donc besoin d'être évalué qu'une unique fois. Nous présentons certains moyens de découverte de ces informations pour chaque application puis pour la plate–forme.

3.3.1 Applications : graphes de tâches et requis d'exécution

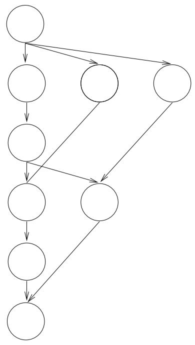
Pour pouvoir prédire un temps de terminaison ou choisir les ressources où exécuter les tâches d'une application, il est naturel de considérer le maximum d'informations disponibles sur cette application. L'objectif est d'en tirer ses besoins, qu'ils soient processeur, mémoire ou de communication jusqu'à éventuellement être capable de la modéliser. On discerne donc parmi les objectifs des techniques pour caractériser une application, celles qui s'attachent à la modélisation de l'application de celles qui s'attardent sur les besoins de chaque tâche la composant.

Après avoir introduit les modèles des applications, nous nous intéresserons aux besoins des tâches. Ils peuvent être obtenus grâce à des techniques d'analyse statique du code source ou d'analyse dynamique lors d'une exécution, à l'aide d'un profiler ou de senseurs. Nous verrons dans les sections suivantes comment ces informations sont utilisées pour déterminer la durée des tâches et des communications.

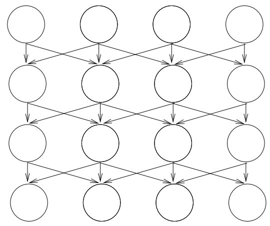
Modélisation des applications

Une application est un ensemble de tâches communicantes. Elle peut être représentée selon son *modèle structurel*, c'est–à–dire un graphe orienté acyclique (DAG, *Directed Acyclic Graph*) $G = (V, E, C, T)$, où $V = \{v_1, v_2, \dots v_n\}$ est l'ensemble des sommets, $E = \{e_{i,j}\}$ est l'ensemble des arêtes, $C : V \to \mathbb{R}$ et $T : E \to \mathbb{R}$ sont deux fonctions de coût. Chaque sommet de V modélise une tâche de calcul. Chacune de ses tâches v à un coût défini par son nombre total d'instructions donné par $C(v)$. Une arête $e_{i,j}$ de E correspond à une relation de précédence entre les tâches i et j. Elle a un coût de communication qui lui est associé et qui est donné par $T(e_{i,j})$, le volume de données qui passe de la tâche i à la tâche j. On donne à la figure 3.1 différents graphes de tâches d'applications courantes. Les nœuds représentent les tâches de calcul qui doivent être placées sur les serveurs. Les arêtes, orientées, modélisent les dépendances entre les tâches. Selon ce modèle, une tâche est une unité atomique de calcul ou de communication : hormis les communications nécessaires pour l'envoi des données en entrée et en sortie, une tâche ne communique donc pas avec d'autres pendant sa phase de calcul.

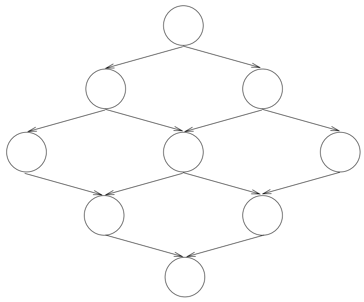
Remarquons que les besoins des tâches rencontrées sont souvent dépendants des res-



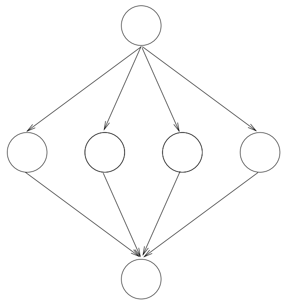
Graphe d'une décomposition LU



Graphe d'une application stencil



Graphe d'une application Laplace



Graphe d'une application fork–join

FIG. 3.1 – Exemple de différents graphes de tâches

sources où elles sont exécutées. L'implantation d'une routine est souvent différente selon l'architecture du serveur et peut même exister sous des formes différentes dans plusieurs bibliothèques : on rencontre des optimisations pour le cache, pour des machines multi–processeurs, vectorielles, etc. Une application peut donc disposer de plusieurs modèles propres à certaines architectures, voire à chaque hôte. Dans ce dernier cas, sous réserve d'avoir les informations correspondantes sur chaque hôte, le coût de calcul d'une tâche peut directement se définir par sa durée sur la machine dédiée.

Analyse du code source

Si on dispose du code source d'une application qui sera traitée sur l'environnement, par exemple lorsqu'on en est l'auteur, l'une des premières méthodes pour en tirer les informations relatives au travail demandé consiste à l'analyser. Il y a deux niveaux d'observation :

– Structure de l'application : l'analyse peut être automatisée et peut être considérablement simplifiée dès que le développeur dégage bien la structure grâce par exemple à un logiciel comme Hence [BDH$^+$94] ou s'il fournit une description de l'implantation, qui est généralement donnée à l'aide de logiciels de représentation graphique des graphes de dépendances comme Zoom [ASWB95].

– Structure d'une tâche : on dispose alors du nombre d'opérations élémentaires de calcul effectuées. Elles sont généralement dissociées entre les opérations effectuées sur les nombres flottants et sur les nombres entiers.

Ces informations renseignent très exactement sur la composition d'une application et des besoins des tâches. Ajoutées aux informations sur la capacité de traitement des serveurs de calcul, elles permettent la mise au point d'estimations en temps comme il a été fait dans [Fig97, Sch98, TWS03]. Réinjectées dans des simulateurs (voir la section 3.6), l'utilisateur peut tenter de mieux paramétrer son application et observer les résultats d'autres allocations afin d'obtenir de meilleures performances. Cependant, l'étude peut s'avérer fastidieuse à cause des optimisations possibles pour différents types d'architecture par exemple.

Analyse d'une exécution, profiling et monitoring

L'analyse du code source a un inconvénient majeur : la technique repose sur l'étude du source du programme pour déterminer ses besoins. Or, sa disponibilité n'est pas systématique et l'exécutable généré dépend du compilateur et des options qui lui sont données.

Une méthode simple pour obtenir la durée d'exécution d'une tâche consiste à la chronométrer. Ceci suppose un comportement constant de l'exécution, indépendant du contenu des données et une exécution sur une ressource dédiée. D'autres approches consistent à analyser le code exécutable, son comportement pendant son exécution ou *post–mortem* à l'aide de traces ou *fichiers de logs*.

Profiling.

Le profiling est généralement utilisé pour optimiser certaines parties d'une application qui sont les plus exécutées. Dans notre contexte, il permet de mesurer *les besoins mémoire*, *le graphe d'appel des fonctions*, *le temps d'exécution de chaque appel* sur l'architecture où a lieu le test, etc. Il n'implique pas nécessairement la disponibilité du code sauf pour la production de rapports annotés, et donc plus compréhensibles. En revanche, les calculs déportés ne peuvent être analysés et les volumes des données difficilement observés. La méthode est plus propice pour observer les besoins et le comportement d'une routine séquentielle. Elle est fastidieuse puisqu'elle demande une exécution complète de l'application et l'étude du rapport généré mais renseigne précisément dans le cas de routines ne dépendant pas du contenu des données. Nous présentons ici trois utilitaires :

Gprof [GKM82] est un utilitaire qui fournit, après analyse d'un exécutable d'un programme C, Pascal ou Fortran 77, *le graphe d'appel* des différentes fonctions le constituant et le temps passé dans chacune d'elles. Pour cela, une option particulière ("-pg" pour `gcc`) doit être entrée lors de la compilation. C'est l'exécution du code obtenu qui génère le profil d'appel dans un fichier qu'on donnera à gprof pour exploitation.

Des utilitaires comme **pixie** [Sys90] ne nécessitent pas d'options particulières supplémentaires lors de la compilation. Pixie analyse le code pour compter *le nombre de blocs d'opérations élémentaires.* Le post–processeur convertit ensuite ces nombres en temps idéal en faisant la somme du coût théorique de chaque instruction d'un bloc. Notons cependant que pixie n'est disponible que sur certaines machines MIPS (*Microprocessor without Interlocked Piped Stages*) et que les informations temporelles produites ne considèrent pas les effets de cache, ce qui peut les fausser.

Enfin, une solution peut–être plus «générale» pour établir les coûts relatifs à l'exécution d'une tâche de calcul est l'utilisation de **OProfile**[44]. Il ne nécessite pas d'option de compilation particulière et opère au niveau du noyau d'un système Linux[45] avec le chargement d'un module. Notons qu'OProfile est disponible sur plusieurs architectures mais que son utilisation est conditionnée par la version du noyau installé sur le système.

Monitoring.

Les capteurs distribués comme ceux de **NWS** (section 3.4.2) ou **Mercury** rendent possible la surveillance des données utilisées par les applications. Des caractéristiques comme les besoins de chaque tâche de calcul, le volume de données traitées (lors des transferts et en mémoire vive des serveurs de calcul) et certaines contraintes de précédences, peuvent ainsi être déterminées. Parmi les inconvénients de la méthode, on remarque qu'elle peut être dépendante de la plate–forme de test et qu'elle suppose un comportement d'exécution indépendant du contenu des données transférées pour recueillir des informations directement exploitables. Nous présentons ici trois moyens différents pour surveiller le comportement d'une application.

pMAP [GRRL04] (*predicting the best Mapping of parallel APplications*) est un outil qui se charge de répartir automatiquement les tâches d'une application à passage de messages sur des processeurs, à partir de la caractérisation de son comportement. Pour cela, il utilise les traces d'une exécution (obtenues ici avec VAMPIR–trace [Vam]) et détermine le modèle du graphe de l'application soumise, c'est–à–dire son DAG, qui comprend : le temps de calcul associé à chaque tâche, le volume de communication des données à transférer entre deux tâches de calcul et le pourcentage maximum d'exécution parallèle auquel des tâches adjacentes peuvent prétendre, calculé en fonction de leurs dépendances. Il synthétise l'ensemble des informations obtenues, coûts des tâches, appels bloquants ou non, etc. dans un *Temporal Flow Graph* dont il tire finalement le graphe d'allocation.

Les évènements eux–mêmes peuvent être monitorés et exploités pour un ordonnancement de l'application basé sur l'utilisation des ressources. Leur enregistrement dans **des fichiers de logs** permet une exploitation *post–mortem* et rend possible des comparaisons entre plusieurs exécutions. Ainsi, dans [GMC02], les appels système sont détournés et les durées des transitions entre les appels système sont mesurées. Cependant, la méthode employée pour la modélisation nécessite l'application d'un patch sur le noyau Linux de

[44] `http://oprofile.sourceforge.net/`
[45] `http://www.kernel.org/`

chaque ressource.

L'étude des variations des besoins des tâches exécutées est parfois utilisée pour générer **la signature de l'application** [VAMR01]. Elle introduit les caractéristiques intrinsèques de l'application, seules dépendantes de l'implantation et de ses besoins. La signature est une courbe paramétrée en n–dimensions selon le temps qui parcourt l'espace des besoins requis lors de l'exécution du programme : charge processeur, quantités en entrées et en sorties et le volume de communication dégagé. La signature d'une exécution est la trajectoire obtenue dans l'espace des caractéristiques d'exécution, qui reflètent la corrélation entre les besoins de l'application et les ressources proposées.

3.3.2 Puissance de calcul d'une ressource et benchmarking

Le **benchmarking** englobe plusieurs techniques sur l'évaluation des performances de l'exécution d'un code. Les benchmarks sont utilisés pour comparer les capacités des différentes composantes du système. Les caractéristiques détaillées des ordinateurs (architecture, gestion du cache, etc.) sont difficilement comparables pour déterminer quelle est la meilleure ressource pour un type de problèmes donné. Résoudre un ensemble bien défini de problèmes sur ces ressources permet en revanche de les étalonner, c'est–à–dire de pouvoir saisir une empreinte de leurs capacités sans entrer dans les détails architecturaux. Grâce à elle, on peut alors émettre un jugement de valeur pour les classifier. Un ensemble de benchmarks bien choisis (c'est–à–dire sensibles à divers paramètres architecturaux par exemple) permet d'évaluer la faiblesse et la puissance de certains systèmes. Il doit être exécuté sur chaque élément de la grille.

Il existe de nombreux benchmarks comme les NAS[46] (NPB, *NAS Parallel Benchmarks* et NGB, *NAS Grid Benchmarks*). Les NPB, sur lesquels sont fondés les NGB, ont été développés pour la NASA afin de mesurer les capacités matérielles et logicielles de systèmes utilisés pour la résolution de problèmes de dynamique des fluides. On trouve aussi le benchmark LINPACK [DLP03], communément utilisé dans le domaine du calcul matriciel. Développé par Jack Dongarra, il repose sur la résolution de systèmes denses d'équations linéaires. Le type et le nombre exact d'opérations élémentaires à exécuter sont fonction de la taille des entrées et sont donc très exactement connus. Grâce à ce benchmark de calcul, le temps retourné permet dans une certaine mesure de valider la capacité maximale de calcul de la ressource, i.e., sa **performance crête** (section 3.2.1). Grâce à cette valeur et aux informations disponibles sur le travail qu'on souhaite la voir exécuter, une estimation des performances possibles peut être calculée (section 3.5.1).

[46] `http://www.nas.nasa.gov/GGF/Benchmarks/NGBspec1.0.pdf`

3.3.3 Mesures du réseau

Le réseau est rarement dédié, les mesures prises sur la connectivité des machines, la bande passante ou la latence doivent donc être effectuées de façon périodique. En revanche, certains *éléments topologiques* d'un réseau peuvent être déterminés et ensuite réinjectés dans des simulateurs pour faire de la prédiction de performances ou des études sur la modélisation d'un support réel et non généré aléatoirement (section 3.6.1).

De plus, *le débit maximal* d'un réseau, en bits par seconde, peut aussi être une valeur intéressante. Une application peut utiliser cette information afin d'optimiser les échanges de données entre ses tâches de calcul, par exemple en ouvrant simultanément plusieurs canaux de communication (*sockets*). Elle permet aussi de savoir quel est le rendement obtenu (pourcentage de la bande passante utilisée par rapport à celle disponible) jusqu'à présent par rapport aux capacités maximales du réseau. Cependant, le réseau étant partagé, la mesure est délicate, en particulier pour des réseaux WAN (*Wide Area Network*) ou des réseaux universitaires disposant de grands débits.

Iperf[47], utilisé dans le projet TeraGrid[48], permet de mesurer les limites du réseau pour les protocoles TCP et UDP dont le débit maximal. Il est aussi capable de mesurer la bande passante disponible ainsi que d'autres données dynamiques. A l'inverse, connaissant les capacités théoriques d'un réseau, il permet aussi de mieux le configurer pour optimiser ses performances.

3.4 Mesure des informations dynamiques

Les informations dynamiques concernent l'ensemble des données qui peuvent être recueillies sur la progression de la soumission d'une application ou de l'exécution d'une tâche, et sur le suivi des disponibilités des ressources. Elles permettent de *déterminer l'état des ressources et l'état d'avancement d'exécution des tâches et des applications.* Nous présentons donc ici des moyens complémentaires d'obtention de ces informations : tout d'abord la technique d'instrumentalisation pour surveiller l'évolution de l'exécution d'une tâche (donc de l'application dont elle fait partie) puis les projets Gridmon et NWS dont les outils servent à monitorer les ressources de la plate–forme.

3.4.1 Instrumentalisation du code source

L'analyse d'un code peut être complétée par son instrumentalisation afin de récupérer après compilation et exécution, des informations plus détaillées sur le programme. La technique est peu ou non intrusive et consiste généralement à insérer des instructions dans

[47] `http://dast.nlanr.net/Projects/Iperf/`
[48] `http://www.teragrid.org/`

le code qui produiront un rapport sur le coût en temps ou en mémoire d'une partie ou de la totalité du programme. Elle suppose la disponibilité du code source et le travail d'analyse correspondant, ce qui limite *de facto* une utilisation systématique. Elle est utilisée par exemple dans [VSF02] sur le service de transfert de fichier Globus, GridFTP. Le service enregistre les informations des transferts effectués jusqu'à présent. Ensuite, la mesure passive de la bande passante est calculée en divisant les tailles des fichiers transférés par la durée du transfert. Cependant, l'instrumentalisation d'un code dans le but de générer ces rapports peut conduire à la perturbation des valeurs obtenues [MRW92, TWS03].

Notons que l'instrumentalisation est aussi utilisée pour sauvegarder l'état des données du programme afin de permettre, en cas de panne ou de préemption (section 4.2.6), de reprendre l'exécution de la tâche à la dernière sauvegarde effectuée.

3.4.2 Quelques projets de surveillance

L'instrumentalisation, lorsqu'elle est possible, peut être compliquée à mettre en place et n'est généralement pas suffisante pour *connaître l'état des ressources* : l'environnement n'a pas d'information sur une ressource n'exécutant pas de tâche et la plate–forme peut être multi–utilisateurs. L'évaluation des capacités de la plate–forme doit donc prendre en compte son évolution au cours du temps. Ceci se traduit par **le déploiement de capteurs** sur les ressources. Leur utilisation est nécessaire pour donner un minimum de réactivité à l'ordonnanceur. Pratiquement, elle doit être étudiée et appliquée soigneusement : il faut déterminer quelles ressources sont à surveiller et quels paramètres s'avèrent intéressants. Les moniteurs engendrent au mieux des communications et donc perturbent le système. Au pire, à cause des différentes mesures calculées, de la technique utilisée ou de leurs fréquences, ils génèrent des rapports dont les valeurs se trouvent biaisées par leur propre calcul. L'intrusion provoquée par l'utilisation des senseurs demande donc une réelle attention sur leur déploiement.

De très nombreux projets existent, chacun pouvant être construit autour d'une architecture propre. Ils peuvent être conçus pour un type de grille ou un environnement de soumission en particulier, ou leurs objectifs se différencient par leur domaine d'applicabilité, leurs spécificités à un type d'application ou au contraire leur souci de généralité.

La frontière est souvent étroite entre les utilitaires permettant de mesurer et de recueillir les informations d'un système et ceux qui constituent fréquemment les bases d'un module de prédiction de performances. Ainsi, NWS [WSH99], que nous décrivons ici, est un projet qui a évolué et qui fournit en plus du système de surveillance, des prédictions sur l'état du système selon plusieurs méthodes de calcul.

Ne pouvant être exhaustifs, nous présentons ici deux projets de surveillance réseau, chacun ayant un domaine d'utilité propre : l'objectif de la suite d'outils GridMon est «d'identifier les pannes et les causes d'inefficacité» dans une grille au niveau réseau alors que NWS est l'une des références dans le domaine des senseurs à but applicatif.

Suite d'outils GridMon

GridMon[49] est une suite d'outils destinée à fournir des informations concernant *l'ensemble des réseaux* mis en jeu dans les grilles e-Science britanniques pour en tirer des performances globales sur les applications. Ainsi, elles pourront être visualisées en temps réel par les utilisateurs ou les équipes de maintenance. Les intergiciels pourront fournir les renseignements adéquats utiles aux clients ou aux applications. De plus, GridMon permet l'étude des comportements qualitatifs des protocoles TCP dans le cadre des réseaux hauts débits (connexions multiples en canaux, tailles des fenêtres TCP).

La suite, conçue plus particulièrement pour les Data Grids, permet de mesurer la **perte de paquets** (physique ou par altération), la **connectivité** (possibilité de se connecter à un site distant) pour identifier les pannes ou en déduire les périodes de congestion du réseau, la **gigue** (variation de la latence, c'est–à–dire délai d'inter–arrivée des paquets) pour les applications multimédia par exemple, le **RTT** et différents débits TCP et UDP (débit réseau et bande passante).

La partie *monitoring* se fait grâce à un ensemble de programmes installés sur les ressources : IperfER, PingER, UDPmon, MiperfER, bbcp/ftp. Comme résumé sur la figure 3.2, les informations sont stockées localement et diffusées aux utilisateurs par l'intermédiaire d'une interface Web ou aux intergiciels de grille grâce à un service de publication (LDAP, grille ou Web). Les mesures sont généralement prises toutes les demi–heures et transmises à l'ensemble des sites.

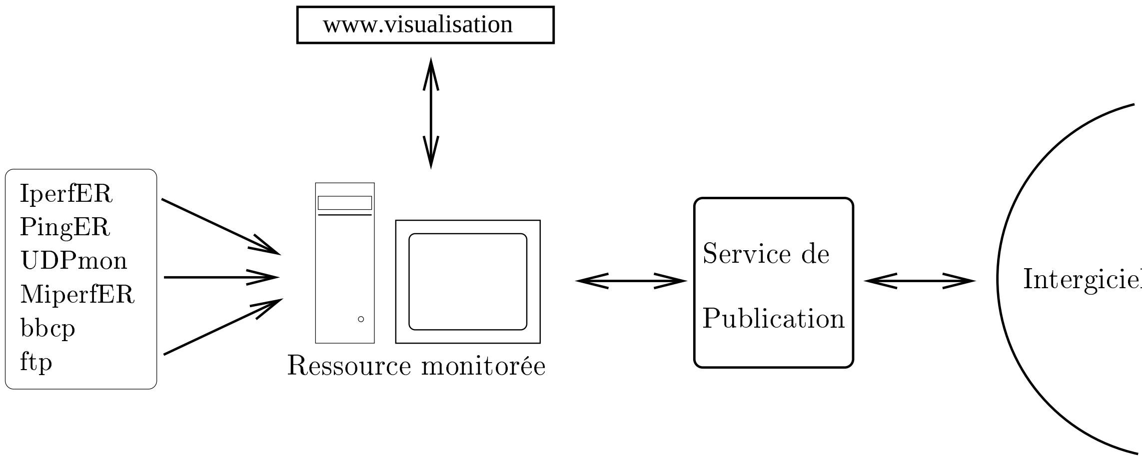


FIG. 3.2 – Architecture de l'environnement de surveillance réseau GridMon

Remarquons que la machine hôte doit être dédiée à la surveillance et faire référence pour l'ensemble des autres machines du site. Ceci sous–entend que les données recueillies auprès de l'hôte sont représentatives : la machine doit donc disposer des mêmes conditions d'accès que les autres, en particulier des mêmes mécanismes de connexion.

[49] `http://gridmon.dl.ac.uk/`

Network Weather Service

NWS [WSH99] (*Network Weather Service*) est un projet développé à l'Université de Californie, Santa–Barbara. C'est un environnement distribué permettant de récupérer des informations pertinentes pour la prédiction de performances, par l'intermédiaire de senseurs déployés sur les ressources participant à la grille, qu'elles soient réseau, de stockage ou de calcul. Ses qualités l'ont fait élire comme remplaçant du projet GloPerf[50] [LWF+99], (*GLObus network PERformance measurement tool*), développé pour et par l'équipe de Globus [Pro], dont le déploiement posait quelques problèmes. Notons que ce projet reposait lui–même sur un projet plus ancien de benchmark réseau : netperf [Com95].

NWS est certainement l'un des systèmes de surveillance les plus populaires. Beaucoup d'environnements de calcul, comme AppLeS [BW97, BWF+96], APST [COBW00], NetSolve [CD96], prévoient des extensions particulières afin d'utiliser ses informations.

NWS fonctionne sur une architecture à deux niveaux, instrumentation et gestion, et quatre composants. Dans la première couche se trouvent les senseurs. Dans la deuxième, trois modules permettent de fédérer les senseurs et leurs données : les serveurs mémoire, les prédicteurs et les serveurs de nom.

Décrivons plus en détail ces composants [Atl01] :

- Les senseurs : ils sont chargés de servir de nœud terminal lors d'une évaluation de la charge du réseau entre deux serveurs et de récupérer les informations sur la ressource hôte : par consultation de pseudo–fichiers (accessibles dans `/proc/` sous UNIX) ou en lançant un processus de prise de mesures (*capacity*) selon leurs capacités (*skills*).
- Les serveurs de nom : ils permettent de garder une trace de l'ensemble des senseurs en fonction de leurs capacités, des activités en cours ainsi que des données déjà disponibles. Ils utilisent un système proche de LDAP qui présente sous forme de fiches les différentes entrées du serveur.
- Les serveurs mémoire : ils servent sur disque à stocker les valeurs obtenues par les senseurs. En mode périodique, chaque senseur leur envoie régulièrement une nouvelle valeur de la mémoire disponible.
- Les prédicteurs : en plus de proposer une solution de surveillance, grâce à ses autres modules et avec suffisamment de prises d'informations, NWS est capable de prévoir leur évolution en les considérant comme une série statistique. Un prédicteur dispose de plusieurs méthodes de prévision statistique qui peuvent être appliquées à des séries différentes en fonction du type de données manipulées. Les différentes méthodes disponibles sont enregistrées dans le serveur de nom. La sélection de la méthode de prévision se fait de façon dynamique en choisissant celle des différentes méthodes proposées minimisant les erreurs calculées jusqu'à présent.

Nous donnons le schéma des échanges expliqués précédemment sur un exemple à la figure 3.3 où nous avons représenté deux hôtes : l'hôte 1 dispose d'un senseur et d'un serveur

[50] `http://www-fp.globus.org/details/gloperf.html`

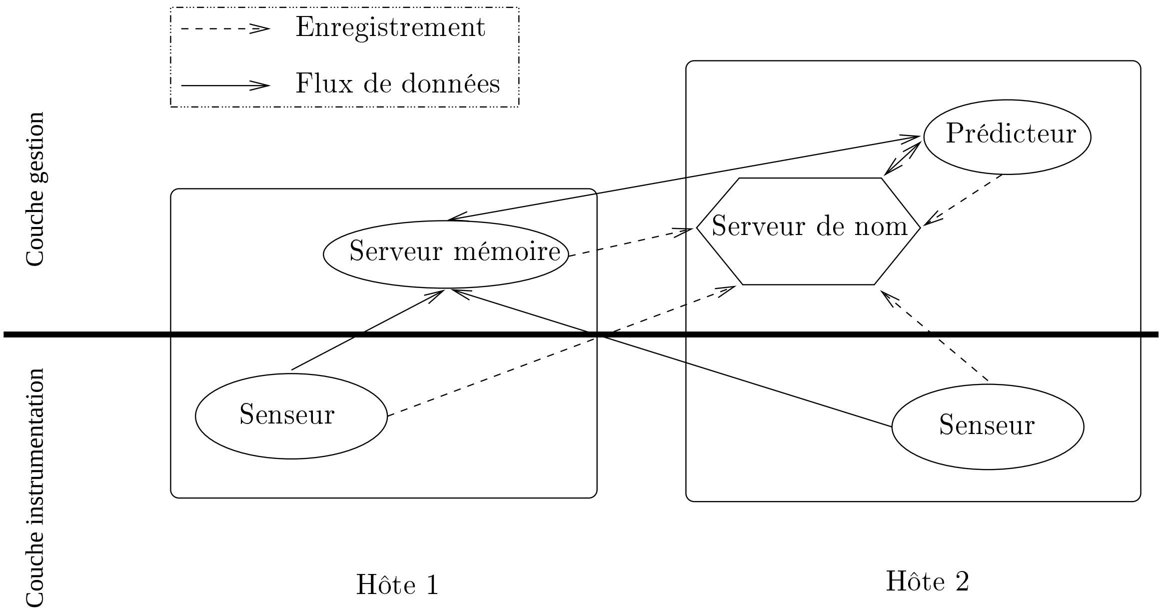


FIG. 3.3 – Architecture de NWS

de mémoire ; l'hôte 2 d'un senseur, d'un prédicteur et d'un serveur de nom. Initialement, chaque composant s'enregistre auprès du serveur de nom. Les senseurs délivrent périodiquement leurs informations au serveur mémoire. Lorsqu'un client souhaite une prédiction sur l'évolution d'une caractéristique de la ressource, il contacte le prédicteur. Ce dernier récupère l'identité du serveur mémoire auprès du serveur de nom. Après chargement des données, il calcule la prédiction demandée.

NWS renseigne sur la **latence**, la **bande passante** telle que perçue par l'application sur l'ouverture d'une communication par socket UNIX : la mesure obtenue est la performance qu'elle peut espérer et non la capacité réelle du réseau (par exemple, on ne peut exploiter la puissance de liens gigabits qu'en l'utilisant en multi–canaux, c'est–à–dire en ouvrant plusieurs canaux de communication simultanément). Des informations sur les ressources de calcul sont aussi disponibles. On trouve la **charge processeur actuelle**, la **quote–part** qu'un nouveau processus pourrait espérer en commençant son exécution, la **mémoire disponible en RAM ou sur disques**.

L'utilisation de NWS fournit des informations intéressantes, tant par ses rapports que par ses prédictions, mais elle est clairement intrusive. Afin de limiter l'utilisation des ressources du système sous surveillance, NWS propose un mode clique qui consiste à regrouper un certain nombre de capteurs entre eux. Chaque senseur envoie à tour de rôle la donnée observée en faisant passer le jeton, essayant ainsi de ne pas perturber la prise de mesures des autres [GWT00]. De plus, il faut régler les fréquences des rapports de sorte qu'ils n'interfèrent pas les uns avec les autres [Qui03] (ainsi, NWS ne donne pas plus d'un rapport toutes les trente secondes sous peine de biaiser ses calculs).

Remarques

On peut citer deux autres projets concurrents très complets : **Ganglia** [MCC04] est un projet open source initialement développé à l'Université de Californie, Berkeley, dans le cadre du projet Millenium. Conçu pour la surveillance d'un cluster, il est devenu un système distribué de surveillance pour des grilles destinées à du calcul haute performance basé sur une architecture hiérarchique ciblant particulièrement les fédérations de clusters. Il est maintenant intégré dans le MDS (*metadirectory service*) de la boîte à outils Globus. Développée pour l'environnement GridLab[51], la suite **Mercury [NGB], Pythia et Delphoi** est implantée selon les recommandations du GMA du Global Grid Forum et est standard à la norme OGSA.

3.5 Méthodes et outils de prédiction

Nous avons rencontré dans ce qui a été présenté certains des moyens mis en œuvre pour recueillir des informations statiques et dynamiques sur les applications/tâches à effectuer sur une plate–forme, comme par exemple l'instrumentalisation des codes source ou l'utilisation de capteurs. Les méthodes que nous décrivons ici utilisent généralement ces informations, qu'elles aient été obtenues sur des applications similaires ou par observation de multiples traces d'exécution.

Nous discernons ici trois catégories de méthodes de prédiction : l'utilisation du benchmarking pour prédire la durée des applications ; les outils statistiques qui fournissent des prévisions découlant des observations faites jusqu'à présent et enfin les outils probabilistes dont les estimations reposent sur des distributions de probabilité en plus des rapports des capteurs.

Toutefois, relevons l'approche récente présentée dans [XDCC04] où les auteurs utilisent l'émulateur MicroGrid (présenté brièvement à la section 3.6.2) pour établir des prédictions de performances sur les applications. Si un émulateur ne peut être utilisé en temps réel pour être intégré dans le cadre GridRPC (le temps d'émulation n'est pas négligeable par rapport aux délais d'évolution de la plate–forme), les bons résultats montrent que la démarche est prometteuse.

3.5.1 Prédiction statique de la durée des applications

Plusieurs méthodes utilisant le benchmarking peuvent être employées pour prédire la durée des tâches et des applications. La méthode retenue dépend de la disponibilité du code source et de son analyse ainsi que de la durée d'accès possible, en mode dédié, à la ressource où sont effectués les tests.

[51]http ://www.gridlab.org/

Nous avons vu à la section 3.3.2 que le benchmarking est un moyen d'exprimer les capacités des serveurs et de pouvoir les classer malgré des architectures potentiellement différentes. Parallèlement, les résultats des benchmarks sur les ressources renseignent sur le temps mis par les machines test pour effectuer certaines opérations bien définies. La *disponibilité d'informations* sur les routines qui seront exécutées sur ces ressources (fondées sur la disposition et l'analyse de leur code source) permet donc d'estimer la durée de l'exécution de la tâche sur le serveur dédié. Si on considère le ratio de la durée par le nombre d'instructions du benchmark, un simple produit du nombre d'instructions de la tâche considérée par ce ratio fournit l'évaluation cherchée. Typiquement, des environnements de calcul GridRPC comme NetSolve, que nous avons présenté à la section 2.5.4, utilisent ce procédé lorsque les serveurs s'enregistrent auprès de l'agent. Ils lui communiquent ce qu'il convient d'appeler la **performance crête de calcul** du serveur. Elle est ensuite utilisée lors des évaluations de performances grâce aux méta–informations de chaque problème que le serveur est capable de résoudre par défaut. Elles contiennent en particulier le polynôme de complexité, qui donne une estimation du nombre d'instruction de la tâche, dans le fichier PDF (*Program Description File*) lors de la création d'un nouveau problème.

Il est aussi possible d'utiliser la tâche que l'on souhaite tester comme benchmark. La même tâche est exécutée plusieurs fois avec des tailles d'entrées différentes sur la même ressource dédiée au test. L'objectif est de pouvoir déduire pour cette ressource une approximation d'une fonction de la durée en fonction de la taille des entrées (c'est–à–dire une approximation du polynôme de complexité). Ceci suppose *a priori* que la durée de la tâche est liée à la taille des données et non à leur contenu.

3.5.2 Outils statistiques

Nous présentons tout d'abord NWS et RPS, deux outils concurrents d'analyse et d'étude statistique des rapports générés par des capteurs sur les ressources supervisées. Tous les deux donnent des prédictions à court terme sur la future variation de charge des ressources. Toutefois, RPS inclue aussi quelques fonctions de prédictions applicatives. Ensuite, nous introduisons FAST, le module de prédiction de performances de DIET (section 2.5.4), construit et déployé au–dessus de NWS. FAST améliore les capacités de NWS et fournit au module d'ordonnancement des prédictions relatives aux durées des tâches.

Network Weather Service : NWS

Parmi les outils à base de senseurs, NWS a été présenté plus tôt comme outil de surveillance des ressources d'un environnement. Il contient aussi des *fonctions de prédiction de la charge des ressources* basées sur les séries statistiques formant quatre familles : la moyenne, la médiane, la moyenne amortie et la dernière valeur. Chaque famille instancie plusieurs méthodes. Pour chaque prédiction, toutes les méthodes sont appelées pour effectuer les calculs statistiques correspondants. La méthode ayant donné la plus faible erreur est retenue et son estimation délivrée. Dans [YFS03, LS03], les auteurs proposent d'autres modèles de calcul qui pourraient lui être ajoutés ; ces modèles partent entre autres de la supposition qu'une plus grande décroissance de la charge devrait être constatée quand la charge est haute. Cependant, ils affirment que l'emploi de modèles statistiques complexes est peu approprié au contexte dynamique. En effet, ils demandent un temps de calcul trop important en vue des marges de qualité de prédiction, par rapport à un modèle plus simple comme la méthode AR (*autoregressive*).

Resource Prediction in Distributed Systems : RPS

Même s'il fournit des capteurs comme NWS, RPS ne se définit pas comme un système de surveillance mais comme une boîte à outils extensible pour créer, déployer et évaluer des systèmes capables de prédire le comportement dynamique des ressources de systèmes distribués. RPS (*Resource Prediction in Distributed Systems*) est un projet conduit à Prescience Lab, au Department of Computer Science, Northwestern University et développé en C++. Il utilise de nombreux *modèles de prédiction* comme NWS, dont la moyenne, la dernière valeur, etc. mais aussi de plus complexes comme les modèles ARIMA (*autoregressive integrated moving average*) ou des modèles d'analyse à base d'ondelettes. Capable de générer des traces pour les études *post–mortem* comme NWS, il dispose aussi d'outils graphiques de contrôle et de bibliothèques de fonctions de *prédiction au niveau applicatif* comme celle de la durée d'une tâche.

Fast Agent's System Timer : FAST

Dans le cadre du GridRPC et plus précisément du projet DIET (section 2.5.4), M. Quinson a développé la bibliothèque FAST [Qui02] (*Fast Agent's System Timer*). Son objectif est de fournir à l'ordonnanceur des *informations au niveau applicatif sur des routines séquentielles*. Pour cela, elle repose sur une architecture fondée sur LDAP et utilise les senseurs NWS. Elle donne une plus grande réactivité dans le cas de «bombardement» de requêtes (arrivée de plusieurs requêtes dans un laps de temps très court), calcule des mesures non disponibles directement comme la latence et la bande passante entre deux points tiers et étalonne les coûts en temps et en espace des routines [Qui03]. En combinant ces informations, elle permet de fournir une prédiction du temps de calcul dans les conditions actuellement connues de la plate–forme.

3.5.3 Travaux probabilistes

On trouve de nombreux travaux considérant l'usage des probabilités pour modéliser les ressources et les capacités qu'elles peuvent offrir ou les besoins des applications. Elles permettent d'une certaine manière de définir les modalités du comportement général, donc de réduire l'ensemble des paramètres inhérents à la grille et aux applications. Nous présentons ici des études sur la prédiction de la disponibilité des ressources et de la durée des applications utilisant des lois de distribution.

File d'attente, modélisations structurelles et distribution

Dans [HBD97], les auteurs estiment la *distribution* de la durée de vie d'un processus. Ils l'utilisent ensuite dans des politiques de migration utiles dans des projets comme Mosix (présenté à la section 2.5.1).

D'autres travaux reposent sur la modélisation des ressources, comme [AMT+98, AMT+00] où on trouve une proposition de modélisation de la grille dans le contexte Global Computing. Les systèmes et le réseau sont modélisés selon des files d'attente [Ken51] mais il reste de nombreux travaux à effectuer pour déterminer les distributions qu'il faut leur appliquer.

J. Schopf propose dans [Sch98, YFS03] de définir pour chaque application sa modélisation structurelle. La structure fonctionnelle d'une application peut être perçue comme le graphe de dépendance (défini à la section 3.3.1). Construite sur la même idée que le *skeletton programming*, la méthode consiste à décomposer la performance en fonction de la structure de l'application dans des modèles de haut niveau et des modèles de composants inter–agissants. Ces modèles sont paramétrés par des valeurs stochastiques, dont certaines sont issues de benchmarks, ou des distributions afin de mieux refléter l'état changeant du système. Le modèle est donc dépendant de l'application *et* de la ressource. Les travaux présentent la validation sur une application dans le contexte maître–esclave dans l'environnement de calcul AppLeS [BW97, BWF+96]. On trouve aussi dans [Fig97] la proposition d'un modèle capable d'évaluer le retard que prend l'exécution d'une application en fonction de sa modélisation structurelle et d'une distribution exprimant la probabilité que plusieurs applications utilisent la même ressource en même temps.

Chaînes de Markov

Certains travaux de prédiction utilisent une modélisation par les chaînes de Markov, comme dans [GSW02] où la grille est modélisée dans son ensemble. Dans [MP02], des machines multi–processeurs, supposées fonctionner selon l'ordonnancement Multi–level Feedback [Tan92] des stations Unix–*like* et assimilées à du round–robin (section 5.2.1), sont considérées avec des charges processeurs stochastiques. La modélisation est utilisée pour prédire le temps de terminaison d'une tâche déterministe.

Les chaînes de Markov peuvent être utilisées en parallèle de réseaux neuronaux et d'informations rapportées par des capteurs distribués sur les ressources directement incorporés dans le code de l'application (c'est–à–dire par intrumentalisation du code que nous avons décrite à la section 3.4.1). C'est ce qui est réalisé dans l'utilitaire Autopilot, présenté dans [VR00]. Reposant sur la caractérisation sous forme de chaînes de Markov des entrées/sorties [SR97] et construit sur les outils Pablo [Ree94] (pour l'instrumentalisation, la surveillance et contrôle adaptatif de l'application) et Globus (section 2.5.2), Autopilot fait le lien entre le comportement des applications et les conditions de la grille. Ceci nécessite un travail d'écriture pour placer les capteurs et les actualisateurs dans le code des applications, ce qui rend cette approche peu propice au GridRPC.

De manière générale, l'étude de ces travaux porte sur des estimations à court terme. Dans [SW03], les auteurs proposent un modèle de performance basé sur des résultats du modèle dérivé de [GSW02], pour des prédictions sur plusieurs heures. Le modèle ne suppose plus une priorité plus forte pour les tâches locales. Il est étendu à une équi–priorité entre les tâches des différents utilisateurs des ressources, même distants. Ils montrent des résultats s'améliorant avec la durée des tâches (en heure) avec, par exemple, un peu plus de 10% d'erreur et une variance de 10 pour une tâche de 4 heures s'exécutant sur un seul processeur.

3.6 Outils d'émulation et de simulation

3.6.1 Introduction et problématiques

La volatilité des ressources et la dynamicité de leurs capacités sont autant de facteurs agissant très fortement sur les performances d'une application. Ces paramètres sont toutefois hors de contrôle et l'algorithme d'ordonnancement recourt à des estimations pour le calcul de ses décisions. Mais évaluer des stratégies de placement différentes dans les mêmes conditions d'expérimentation est particulièrement délicat. En effet, comment reproduire les mêmes variations de charge du réseau, et la même utilisation externe des ressources due aux autres utilisateurs ? De plus, dans le contexte de grille, assurer la faisabilité d'une soumission n'est pas aisé puisque l'application peut nécessiter l'utilisation de plusieurs ressources dont les mécanismes locaux d'ordonnancement n'assurent pas la synchronisation entre les réservations effectuées sur chacune d'elles [WS01, CFK99].

Certaines informations obtenues peuvent être injectées dans des simulateurs ou des émulateurs afin de visualiser l'exécution, de créer, tester et modifier des stratégies de placement sans avoir à réellement réserver et occuper les ressources. Les simulations permettent de comparer les performances de ces stratégies sur des plate–formes qui sont de fait identiques d'une exécution à l'autre et dont les ressources évoluent de la même manière (exception faite de la charge occasionnée par l'affectation des applications soumises et allouées aux ressources par l'environnement simulé) : **la reproductibilité des expériences** est aussi assurée. Comme nous l'avons mentionné dans la section précédente, ces outils peuvent aussi **fournir des prédictions de performances sur les applications** comme dans [XDCC04] où l'émulateur MicroGrid, que nous présentons plus loin, est utilisé.

Pour mieux comprendre la différence entre un émulateur et un simulateur, nous donnons dans la figure 3.4 un graphe inspiré de l'introduction à Grid eXplorer[52]. Il exhibe le réalisme d'une exécution en fonction de l'abstraction de l'outil utilisé pour la produire et des outils sont précisés sur le graphe comme exemple. Ainsi, celui donné pour la réalité est PlanetLab [Plaa]. C'est un environnement à l'échelle de la planète communiquant à travers internet. Les conditions d'expérimentation sont réelles et donc non reproductibles et les applications comme les protocoles sont déployées en vraie grandeur. Ensuite, **les émulateurs** utilisent une virtualisation de la grille : ils l'émulent généralement sur une grappe. Les conditions expérimentales sont reproductibles et recourent le plus souvent à des traces qui ont été préalablement recueillies sur des ressources réelles. Les applications sont les mêmes que celles soumises à un environnement réel mais emploient des bibliothèques modifiées qui interceptent les messages afin d'émuler les communications. **Les simulateurs** utilisent une modélisation des ressources de la grille et des applications. Les expérimentations sont reproductibles mais le réalisme des résultats est à quantifier. Enfin, **les modèles mathématiques** à base d'équations sont les outils les moins réalistes.

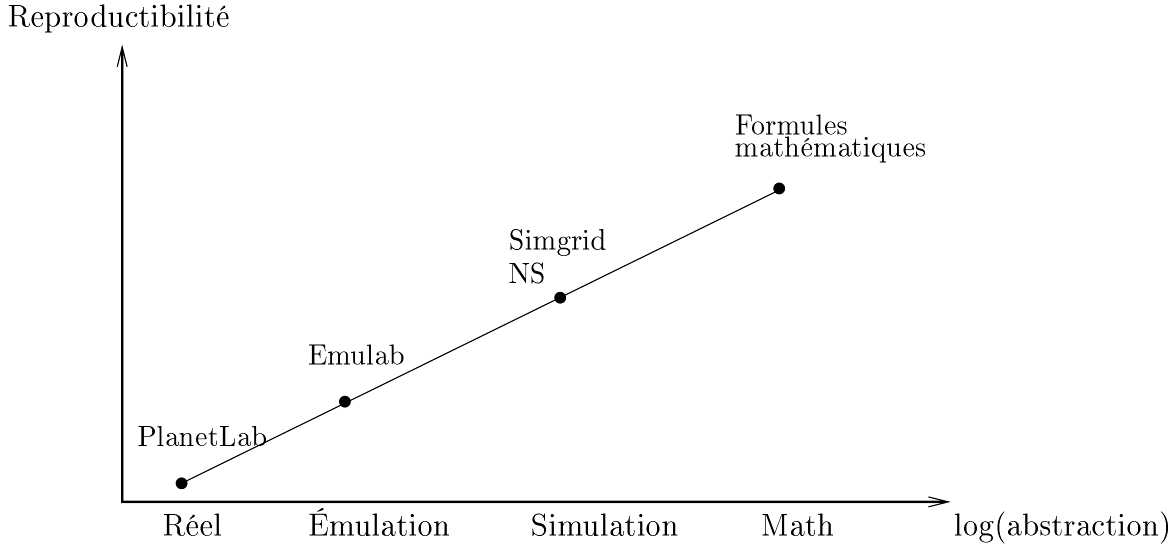


FIG. 3.4 – Reproductibilité des expériences en fonction de l'abstraction des outils

Les émulateurs et les simulateurs permettent la validation et la comparaison d'heuristiques dans de multiples expériences où l'on peut créer des plate–formes sur lesquelles on ne pourrait lancer les applications en réalité, que ce soit à cause du nombre de ressources mises en jeu ou à cause de leurs caractéristiques. Notons que la vitesse d'exécution d'une expérience est croissante avec l'abstraction. Une conséquence directe est l'impossibilité d'utiliser un émulateur dans un module de prédiction de performances dans le cadre d'un ordonnancement dynamique de la plate–forme (section 4.2.3). En revanche, la vitesse de la simulation et la souplesse de la technique permettent de valider des stratégies de placement sur de très nombreuses expériences : les applications, caractérisées par leur graphe de précédence (avec tâches de calcul et communications), peuvent être tirées aléatoirement et les topologies des grilles générées afin d'examiner le rôle de certains paramètres dans les

[52] `http://www.lri.fr/~fci/GdX/`

choix effectués. Nous verrons dans les chapitres 5 et 6 que cette vitesse permet d'utiliser la simulation dans les algorithmes d'ordonnancement. La simulation pour l'ordonnancement permet d'obtenir de très bonnes performances sur de multiples critères d'observation.

Il faut cependant s'assurer de la pertinence de ce qui est généré [FP99, FP01]. Souvent, dans les études actuelles, les travaux d'ordonnancement utilisant des simulateurs supposent des topologies simples, comme des bus, des étoiles, des graphes complets. La dynamicité des ressources n'est généralement pas prise en compte. De nombreuses études [Doa96, FFF99] sont faites pour préciser des critères ou définir des distributions suivant lesquelles la génération de la topologie des grilles peut approcher le plus possible celles observées en réalité (il faut générer les ressources, leur puissance de calcul et de mémoire et gérer l'hétérogénéité relative de la grille). Des travaux assez récents montrent que des cartes topologiques peuvent aussi être utilisées avec succès après avoir été «découvertes» par exemple avec le logiciel [SBW99].

La génération des applications, bien que plus aisée *a priori*, n'est pas facile pour autant. Un constat valable sur des applications générées aléatoirement peut ne plus l'être pour des applications réelles qui disposent d'une structure établie. Cependant, on peut utiliser des graphes simples d'applications connues comme dans [BBR01] dont nous donnons des exemples à la figure 3.1. Il apparaît souvent dans la littérature des études menées sur un ensemble de tâches indépendantes soumises au cours du temps [MAS$^+$99] ou en même temps [CLZB00] (bien que dans ce dernier cas, l'environnement est souvent de type maître–esclave) mais il n'y a pas de tendance définie dans le cadre qui nous occupe.

Dans le contexte GridRPC où l'ordonnancement des tâches se fait à la volée, l'observation des soumissions, encore trop spécifiques, n'est pas assez générale pour obtenir des comportements types et savoir quelles applications sont soumises ainsi que leur fréquence de soumission : les fichiers de logs des serveurs de l'équipe de NetSolve, accessibles lors de la phase de test de l'environnement et à disposition pour effectuer des calculs, ne renseignent pas sur les besoins applicatifs des utilisateurs. Seules des informations sur les tâches y apparaissent et non les relations de précédence entre elles ; souvent, un environnement de soumission de problème déployé par un utilisateur reste confiné à l'exécution de ses applications.

L'avantage des simulations se retrouve encore dans la possibilité de travailler sur les hypothèses que prennent généralement les algorithmes d'ordonnancement, comme une parfaite estimation du temps de calcul. On trouve ainsi dans [CLZB00] une étude du comportement des stratégies de placement en fonction de l'erreur maximale donnée. L'erreur est tirée selon une loi uniforme pour calculer l'estimation du coût de chaque tâche. De la même manière dans [HSvL03], les auteurs considèrent l'impact de l'erreur de l'estimation dans la validation de leurs résultats sur l'amélioration de la qualité de service donnée par leur aménagement de la stratégie de placement min–min (que nous présentons à la section 4.4.2).

3.6.2 Quelques projets

Nous introduisons tout d'abord quelques projets d'émulation et présentons brièvement le projet MicroGrid. Puis, des projets de simulation sont abordés. Nous exposons plus particulièrement Simgrid puisque nous utilisons dans nos travaux des modèles similaires et que nos résultats ont dans un premier temps été validés avec ce simulateur.

Émulateur

Des projets de recherche pour émuler une grille à grande échelle sont en cours d'étude. On peut citer par exemple Emulab [Hom] ou encore Grid–eXplorer [eG] dont l'émulation est à une échelle de 10 à 100 nœuds simulés sur une machine et a pour but des tests sur la sécurité, la performance, etc. L'un des objectifs est de pouvoir fournir des images de grilles réelles se comportant selon des traces qui permettront de reproduire les mêmes conditions expérimentales entre plusieurs tests différents afin de pouvoir les comparer. À une échelle moindre (jusqu'à 100 nœuds par machine), MicroGrid [SLJ+00] permet lui aussi d'émuler des ressources et propose des accès à une virtualisation de la grille.

MicroGrid [SLJ+00]. Faisant partie du Projet GrADS (*Grid Application Development Software*), l'objectif de MicroGrid est de développer et fournir une grille virtuelle sur laquelle des stratégies de gestion de ressources et d'adaptation d'applications peuvent être étudiées.

Les applications, écrites en C, C++ ou Perl, ne doivent pas être portées pour les tester. Il suffit de les compiler (C, C++) ou de les interpréter (Perl) avec une version modifiée des bibliothèques Globus (section 2.5.2) [XDCC04]. MicroGrid émule une grille sur un ensemble de ressources distribuées : la virtualisation des ressources réseau se fait grâce au simulateur de réseau distribué MaSSF construit sur DaSSF [DaS] et les tâches de calcul sont laissées à la gestion d'un contrôleur de charge. Il calcule la charge processeur de chaque hôte virtuel (selon une fenêtre coulissante) et stoppe ses processus si la vitesse réelle de calcul est supérieure au nombre de cycles de l'hôte virtuel (pour empêcher l'exécution virtuelle d'aller plus vite que ce qu'elle devrait).

Simulateur

Il existe plusieurs projets de simulateurs en fonction du domaine regardé : certains sont explicitement orientés réseaux, d'autres versent plutôt dans les études d'ordonnancement au niveau applicatif ou encore dans la compréhension de la grille. La modélisation, les mécanismes de simulation sous–jacents et les moyens mis en œuvre ne sont donc pas les mêmes, leur échelle peut être différente puisque les objectifs peuvent ne rien avoir en commun.

En effet, par exemple dans le premier cas, la simulation peut se situer à bas niveau pour tester l'influence protocolaire des échanges de paquets. Ainsi, NS [Sim] est très utilisé dès qu'il s'agit de tester des mécanismes de routage ou de multicast ou encore de voir l'impact de l'implantation des protocoles, TCP par exemple, sur les performances d'un réseau. De plus, des travaux d'ordonnancement réseau comme l'étude de la redistribution de données peuvent l'utiliser pour leurs expériences [JW04].

Dans notre cadre de travail, où il s'agit d'étudier et comparer des mécanismes d'ordonnancement au niveau applicatif, une échelle macroscopique de la simulation est *a priori* suffisante. Parmi les outils existants, nous pouvons citer les simulateurs Bricks [Tak01], Simgrid [Cas01] et GridSim [MBA01, fRMfPC]. Ce dernier est un projet développé en Java à l'Université de Melbourne en Australie. Son objectif premier était la possibilité de modéliser l'environnement Nimrod [ASGH95] dont les stratégies de placement reposent sur des techniques économiques dans un contexte où les applications sont contraintes par une date buttoir de terminaison, c'est–à–dire une *deadline*.

Simgrid [Cas01]. Simgrid est une API pour la simulation, l'étude et la comparaison d'algorithmes d'ordonnancement, développée en collaboration entre l'École normale supérieure de Lyon et à l'Université de Californie, San–Diego.

Construit autour de modèles simples, SimGrid fournit des appels pour la déclaration des tâches et des ressources. On doit préciser pour ces dernières le mode d'exécution souhaité : le mode **séquentiel** (ce que nous nommons ici espace–partagé ou *space–share*) où l'exécution des tâches allouées est séquentialisée ; le mode **temps–partagé** (ou *time–share*) qui modélise la politique d'ordonnancement round–robin des machines Unix–*like*. L'API donne accès à des fonctions servant à assembler les différentes ressources d'une grille, des fonctions d'allocations des tâches de calcul et de transfert, permettant de pouvoir tester assez rapidement des algorithmes d'ordonnancement applicatif.

Ce projet est en constante évolution et intègre depuis sa version 2.0 des nouveautés (avec la sur-couche Méta–Simgrid) [Leg03] : enrichi avec les travaux d'amélioration de la modélisation macroscopique réseau [CM02], Simgrid présente maintenant l'avantage de proposer l'étude d'ordonnancements distribués particulièrement utile pour modéliser des environnements de type GridRPC distribués comme Diet (présenté à la section 2.5.4).

La simulation d'une charge externe à l'environnement est loin d'être aisée, malgré la présence dans la littérature de travaux tentant de cerner le comportement de la charge : par exemple dans [ML91], forts de l'observation du comportement des machines de leur cluster, M. Mutka et M. Livny concluent que la distribution des périodes des intervalles de temps où les machines ne sont pas disponibles peut être approximée par une loi hyper–exponentielle ; dans [Fei02], les auteurs expliquent la nécessité de modéliser la charge d'une machine, l'avantage de la modélisation sur l'utilisation de traces et comment le faire sans erreur (la modélisation étant dépendante de l'observation statistique de traces, ces dernières doivent être pertinentes et ne pas être biaisées).

Simgrid apporte la possibilité de varier les charges des hôtes simulés, réseau ou serveur, grâce à l'utilisation de traces NWS (présenté à la section 3.4.2). La démarche consiste à distribuer aux tâches simulées de la ressource, à chaque instant, selon son mode et en fonction de la trace utilisée, la puissance totale disponible enregistrée sur la ressource. Il faut donc intégrer la trace pour connaître le temps que prendra la tâche pour finir.

Simuler la charge d'une machine et les rapports de charge de capteurs est difficile. De plus, les valeurs ainsi «rapportées» sont celles de la trace, c'est–à–dire la charge externe à la plate–forme. Elle doit être corrigée et prendre en compte la charge due aux tâches que l'environnement a placé sur les ressources. Donner la part de CPU impartie à une tâche selon une charge externe simulée devient donc problématique si on n'ajoute pas comme contrainte d'allocation qu'une seule tâche est affectée à un serveur. Qu'une telle simulation permette de refléter la réalité n'est pas l'objectif premier. En effet, il s'agit principalement de fournir un terrain réutilisable pour de nombreuses expériences. Pour ces raisons, Simgrid va être réécrit de telle sorte que des applications comme NWS puissent fonctionner directement sur le simulateur. Il permettra aussi d'étudier un programme MPI par exemple sans avoir à le modéliser mais par recompilation avec une bibliothèque fournie. Ses modèles internes seront plus proches de la réalité et l'API plus pratique d'utilisation.

3.7 Conclusion et discussion

Dans ce chapitre, nous avons introduit les caractéristiques des ressources de la plate–forme qu'il convenait de regarder pour connaître ses capacités comme la charge de la machine, la taille mémoire disponible, la bande passante et la latence. Nous avons expliqué l'intérêt des informations statiques et les moyens couramment utilisés pour les obtenir : observation du code ou de l'exécution des applications ou étalonnage des machines grâce à des benchmarks.

La plate–forme étant multi–utilisateurs, nous avons vu que la progression de l'exécution d'une tâche pouvait être suivie en l'instrumentant et que les informations dynamiques des ressources pouvaient être surveillées à l'aide de senseurs.

L'ensemble des informations recueillies permet de calculer des prédictions sur l'état de chaque ressource. Plusieurs méthodes sont utilisées parmi lesquelles nous avons présenté l'utilisation de benchmarks ainsi que des méthodes et outils statistiques et probabilistes.

Les informations obtenues peuvent être injectées dans des émulateurs ou simulateurs. Les simulations, bien qu'elles puissent être utilisées dans un but de prédiction de performances, sont majoritairement exploitées pour tester l'ordonnancement de tâches sur la plate–forme : elles fournissent les mêmes bases de comparaison aux diverses stratégies de placement que l'on souhaite tester.

Cependant, il n'est pas toujours possible de déterminer certaines informations. Ainsi, la durée que prendra une application n'est pas nécessairement fonction de la taille des données. Par exemple, la durée d'exécution des tâches FIIT [Kra99] ou des résolutions de problèmes d'algèbre linéaire denses grâce à des solveurs creux [ADKL01], est directement dépendante du contenu des données sur lesquelles porte le travail à effectuer. Dans ce cas, des solutions consistent à donner une majoration de la durée de l'exécution de la tâche (arbitraire, issue de statistiques ou tirée aléatoirement selon une distribution à déterminer) ou à utiliser des algorithmes de placement n'utilisant pas ce paramètre [PST]. Un problème de performances se pose alors aussi pour les autres tâches du système.

Dans notre travail, une tâche de calcul consiste en un appel GridRPC auprès de l'agent de l'environnement. Nous considérons qu'une tâche est séquentielle : elle ne s'exécute que sur un processeur, au contraire des tâches malléables [Fei99, DEMT04] ou des fonctions de la bibliothèque SCALAPACK[53] qui peuvent s'exécuter sur un nombre de processeurs à définir lors de l'affectation. Nous supposerons de plus à notre disposition la durée de chaque tâche ainsi que la quantité de données à transférer avant et après son exécution (elle peut-être obtenue à l'aide d'une des méthodes citées précédemment). En revanche, le module d'ordonnancement se situant dans l'agent, aucune information n'est disponible sur le graphe de dépendance des applications qui lui sont soumises. Les ressources considérées dans notre travail sont hétérogènes et mono–processeur.

[53]http ://www.netlib.org/scalapack/

Chapitre 4

Ordonnancement d'une plate-forme de calcul distribué

Table des matières

4.1 Introduction

Nous avons évoqué l'ordonnancement et le placement des requêtes à la section 2.4 pour l'envisager un peu plus en détail lors de la présentation des outils et des environnements de soumissions pour la grille à la section 2.5. En particulier, les mode *push* et *pull* utilisés pour résoudre les requêtes sont importants dans sa conception. Généralement, l'objectif d'un ordonnanceur consiste à diminuer les coûts et/ou le temps de production des tâches nécessaires pour accomplir un travail global (qui peut ne contenir qu'une seule tâche). *Ordonnancer des tâches* peut se définir comme *les placer dans l'espace*, c'est–à–dire sélectionner la ou les ressources qui seront utilisées pour les effectuer, et *les placer dans le temps*, donc définir les dates à partir desquelles elles commenceront réellement leur exécution. Pour cela, les différentes informations disponibles sur la plate–forme (capacités actuelles de calcul ou de stockage des ressources, état du réseau) et sur les besoins d'exécution des requêtes permettent tout d'abord de sélectionner des ressources cibles possibles. Ensuite, un algorithme d'ordonnancement est appliqué pour déterminer où les tâches seront allouées et quand elles commenceront.

L'algorithme prend des décisions et fait des choix selon des métriques qu'il a la charge d'optimiser. Il utilise pour cela une fontion de coût qui doit être définie selon les métriques regardées. Il existe de multiples métriques et leur choix dépend du contexte d'utilisation. Par exemple, dans un milieu de production, la date de terminaison du travail global est souvent choisie comme la principale métrique à minimiser alors que les portails web essayent plutôt de minimiser le temps d'attente des requêtes effectuées ou la bande passante utilisée.

Ces observations conduisent *de facto* au constat suivant : à différents objectifs correspondent différentes stratégies. Il est peu probable qu'un algorithme d'ordonnancement soit bon pour tous les problèmes. De plus, l'optimalité souhaitée est particulièrement difficile à obtenir. Ainsi l'ordonnancement d'un ensemble de tâches dont on connaît la durée sur chacune des ressources disponibles et ayant pour objectif la minimisation de la date de terminaison de la tâche finissant le plus tard, est NP-difficile [GJ79, All75]. Cela signifie qu'on peut vérifier la solution avec un algorithme polynomial qui est fonction du nombre d'entrées (ici le nombre de tâches et de serveurs) mais que le calcul de cette solution est particulièrement difficile et long. Les choix s'effectuent donc à l'aide d'algorithmes d'approximation ou d'heuristiques d'ordonnancement qui sont plus faciles à mettre en œuvre. Ceci explique les nombreux efforts déployés ces dernières années pour proposer des algorithmes d'ordonnancement efficaces pour la grille.

Nous commençons ce chapitre par la présentation d'une taxonomie sur l'ordonnancement dans la section 4.2. Nous y définissons les différents types d'ordonnancement. Le contexte du travail y est aussi commenté. Nous décrivons dans la section 4.3 des métriques que l'on peut considérer dans la conception de la fonction de coût des algorithmes d'ordonnancement ou qu'il est possible d'utiliser comme base de comparaison entre deux algorithmes. Nous y introduisons également les notations employées dans la suite du manuscrit. Dans la section 4.4, nous présentons certaines des heuristiques les plus connues dans les environnements de calcul, dont trois en lien direct avec le contexte GridRPC où les tâches sont ordonnancées dès leur soumission. Nous concluons finalement à la section 4.5.

4.2 Taxonomie et terminologie

Un système informatique doit gérer les différentes ressources qui le composent : temps processeur, mémoire, entrées/sorties. Il doit être suffisamment performant pour répondre aux besoins de chaque utilisateur. Ceci vaut pour les caractéristiques matérielles de ses ressources mais aussi pour leur gestion.

Dans le cas des grilles de calcul, les ressources sont distribuées. Dans le cadre GridRPC, c'est l'environnement de soumission de calcul qui assure la gestion de l'ensemble des ressources. Ici, les besoins se mesurent principalement sur le temps de calcul que peuvent offrir les serveurs. Il est important d'ordonnancer au mieux les tâches sur les serveurs pour des raisons de performance : mal gérées, des ressources possédant matériellement les requis nécessaires à l'exécution d'un groupe de tâches peuvent rendre caduc la délocalisation des calculs en terme d'efficacité ou de faisabilité.

Il existe dans la littérature de nombreuses taxonomies. Parmi celles–ci, dans [Tal91], l'auteur propose une classification des algorithmes d'allocation suivant les stratégies utilisées dans le maintien de l'état courant du système et dans le placement des processus dans les systèmes parallèles et distribués. Plus proche du contexte qui nous occupe, nous trouvons dans [BSB+98] une taxonomie définie en trois sections principales selon les modèles utilisés pour les applications et les communications, selon la caractérisation des ressources utilisées et enfin selon les caractéristiques des algorithmes réduits ici à des heuristiques. Celle que nous présentons maintenant complète la classification hiérarchique qui est proposée dans [Jon02].

4.2.1 Ressources homogènes et hétérogènes

L'ordonnancement doit tenir compte des performances des ressources. On discerne les ressources homogènes et les ressources hétérogènes. Les **ressources homogènes** ont des caractéristiques identiques. C'est le cas des machines dans la plupart des clusters. Elles sont dotées du même processeur, de la même quantité de mémoire, des mêmes périphériques et accès au réseau. Aucune distinction n'est donc à faire entre deux hôtes dans le même état même s'il faut considérer la topologie et les performances du réseau les interconnectant. À l'opposé, des **ressources hétérogènes** possèdent leurs propres caractéristiques matérielles. Elles ont donc des capacités de traitement de calcul, de stockage ou de communication différentes.

4.2.2 Ordonnancement local et global

L'ordonnancement global est celui opéré par le système réparti : c'est l'affectation d'une tâche à un ensemble de machines à une date qu'il doit déterminer. Du fait de la répartition géographique des composantes de la plate–forme, les ressources disposent de leurs propres mécanismes de gestion. **L'ordonnancement local** détermine la façon d'exécuter les tâches sur la ressource. Par exemple, il peut être FCFS (*First Come First Served*, les tâches sont exécutées dans l'ordre de leur arrivée), ou round–robin (plusieurs tâches disposent successivement de la puissance de calcul du processeur pendant un petit laps de temps).

4.2.3 Ordonnancement statique et dynamique

Il existe deux familles principales de stratégies d'ordonnancement : statiques et dynamiques [MBS99]. Elles diffèrent principalement sur le moment où sont prises les décisions de placement mais on trouve dans [WM85] des démarcations entre ces deux familles en fonction des multiples niveaux d'utilisation et de disponibilité des informations sur la plate–forme.

Ordonnancement statique

Il repose sur l'ensemble des informations disponibles avant la soumission des tâches, comme par exemple à la compilation de la tâche. On parle alors de mode *Offline* ou déconnecté, puisque les décisions sont propres à une plate–forme et à un ensemble de tâches donnés et n'évoluent plus une fois prises. L'ensemble des décisions constitue ce qu'on appelle *un plan d'ordonnancement*. La prise de décision suppose souvent la disponibilité d'informations sur l'évolution des ressources, comme des modèles stochastiques sur la charge processeur [dSeSG91], ou sur la connaissance totale des caractéristiques des tâches qui seront soumises, comme la date de soumission et les besoins requis. On peut par exemple trouver une étude sur des heuristiques statiques dans [Kwo97] et une synthèse comparant onze différentes techniques de placements statiques connues dans [BSB+99].

Ordonnancement dynamique

Les algorithmes de placement statiques ne tiennent pas compte des évolutions dynamiques des ressources ni de soumissions supplémentaires. On parle donc **d'ordonnancement dynamique** lorsque les décisions sont prises au moment la soumission des requêtes. On parle aussi de mode décisions prennent en compte l'état du système lors de la soumission des requêtes. Ceci permet d'adapter les décisions aux changements de comportement des ressources. Une étude comparative entre des techniques statiques d'une part et dynamiques d'autre part qui sont couramment utilisées est donnée dans [SA99]. Une autre, fondée sur le modèle MQMS (*Multiple Queue Multiple Server*) et validée par des simulations, est proposée dans [BZ92].

On distingue toutefois deux modes d'ordonnancement dynamique : le **mode Batch** (ordonnancement par lot) et le **mode Online** (ordonnancement continu). Dans le premier cas, une partie voire la totalité des requêtes sont soumises avant le début des allocations. On parle généralement d'un *lot* ou *sac de tâches* à ordonnancer [dSCH04]. En conséquence, l'algorithme dispose d'une partie ou de la totalité des informations sur l'ensemble des requêtes qui seront soumises (par exemple des contraintes sur la date de terminaison ou la durée estimée, etc.). Les décisions d'allocation sont prises à des dates données, appelées *dates d'ordonnancement*. Il est ainsi possible d'appliquer un filtre sur l'ensemble des tâches non encore allouées ou de les classer en fonction de leurs besoins (dans des files d'attentes selon l'estimation de leur durée pour le système de soumission Batch *Portable Batch System* PBS [PBS] par exemple). Notons que généralement, quand un travail Batch commence, il ne s'arrête que si survient une erreur (dépassement d'un temps de réservation, panne, suppression) ou lorsque les résultats sont obtenus.

D'autres techniques d'ordonnancement, légèrement différentes du mode Batch, peuvent être employées sur des groupes de tâches comme par exemple la technique de la fenêtre coulissante (*sliding window*) dans [HNK+00]. Les tâches sont des instructions de la MVM (*MATCH Virtual Machine*), une machine virtuelle : afin d'optimiser la compilation d'un code MATLAB [MAT] et de dégager plus de parallélisme, le compilateur regarde une quantité d'instructions égale à la taille de la fenêtre au lieu de simplement les scruter une à une. La dernière tâche entrée n'est donc pas systématiquement ordonnancée.

Au contraire, un algorithme fonctionnant en mode continu ne dispose des informations sur une requête qu'à son arrivée et il l'alloue immédiatement à une ressource. C'est donc typiquement le mode adopté pour le contexte GridRPC propre à notre travail, où les tâches sont traitées à la volée.

4.2.4 Ordonnancement déterministe et probabiliste

Les stratégies d'**ordonnancements déterministes** n'utilisent aucun tirage aléatoire dans leurs décisions et reposent entièrement sur les seules informations qu'elles ont à disposition. Par exemple, dans le cas dynamique, les décisions seront toujours les mêmes dès que l'état du système est identique (mêmes informations sur la nouvelle requête et sur les ressources comme la charge, la mémoire, la bande passante, etc.).

Cependant, des lois de distribution peuvent être employées afin de déterminer la ressource où allouer le travail demandé ; on parle alors d'algorithmes d'**ordonnancements probabilistes**. Dans un cas extrême, chaque décision peut être tirée aléatoirement (à l'aide de l'algorithme *Random*). Il arrive aussi que le module d'ordonnancement ait à choisir entre deux serveurs alors que les calculs permettant de faire le choix retournent des valeurs identiques. Par exemple l'algorithme JSQ (*Join Shortest Queue*) choisit la ressource comportant le moins de travaux soumis. Iinitialement, tous les calculs retournent zéro. Si le choix est laissé à un tirage, l'algorithme est probabiliste alors que si le choix se fait sur le serveur où la tâche termine le plus tôt, l'algorithme est déterministe.

4.2.5 Ordonnancement centralisé et distribué

Le module d'ordonnancement peut être exécuté sur une seule machine, généralement dédiée à cette tâche ; on dit que l'**ordonnancement est centralisé**. Au contraire, s'il est distribué sur plusieurs hôtes éventuellement répartis géographiquement, on parle alors d'**ordonnancement distribué**.

L'inconvénient principal d'un ordonnancement centralisé vient justement de sa caractéristique : la centralisation peut devenir un goulet d'étranglement au niveau de la performance, surtout pour des plate–formes étendues qui nécessitent un nombre de communications conséquent pour tenir les informations à jour. En revanche, une fois les informations assemblées au même point, leur traitement pour décider de l'allocation est aisé.

Il est difficile de tenir des informations cohérentes et à jour dans un système complètement distribué, car cela génère vite un grand nombre de communications qui sont pourtant nécessaires pour informer les autres modules des décisions d'ordonnancement prises. On dispose d'une structure permettant une certaine tolérance aux pannes mais au prix d'un effort pour concevoir un algorithme d'ordonnancement *efficace*.

En référence au chapitre 2, nous avons présenté deux exemples qui rendent compte de ces différences : NetSolve qui est un NES construit sur le modèle Client–Agent–Serveur et où le module d'ordonnancement est intégré dans l'agent ; DIET qui est construit sur une hiérarchie d'agents (les *local agent* et les *master agent*) et où le module d'ordonnancement est réparti sur cette hiérarchie.

4.2.6 A propos de la migration

Nous distinguons également les stratégies d'ordonnancement **non–préemptives** et **préemptives** : dans le premier cas, si une tâche commence son exécution sur un serveur, elle la poursuit jusqu'à ce qu'elle termine (correctement ou à cause d'une erreur ou d'une panne). Elle ne peut être stoppée dans le but de reprendre les calculs ultérieurement. Dans le second cas, une tâche peut être interrompue puis redémarrée éventuellement sur une autre ressource où elle poursuit son exécution : **elle migre**. La migration est employée dans les systèmes où il faut s'assurer de l'équilibrage de charge entre les ressources (par exemple dans openMosix présenté à la section 2.5.1). Naturellement, elle engendre des mécanismes conséquents sur les hôtes et des transferts. Il faut donc prendre en considération ces coûts avant de l'envisager.

4.2.7 Positionnement du travail

L'objectif de ce travail de thèse est d'étudier l'ordonnancement sur une plate–forme de métacomputing, donc dans le contexte GridRPC, en proposant des alternatives pertinentes pour le module d'ordonnancement implanté au sein de l'agent. L'ordonnancement se fait donc à un niveau global et doit gérer des ressources distribuées et hétérogènes. Il traite les tâches au fur–et–à–mesure de leur soumission et elles ne peuvent migrer : l'ordonnancement est dynamique, continu et centralisé. Puisque les tâches commencent à s'exécuter dès que les données ont été transférées au serveur, l'ordonnancement est restreint ici à de l'affectation de tâches sur les ressources (*matching*).

Nous supposerons par la suite que les ordonnanceurs locaux aux ressources fonctionnent en round–robin. De plus, même si certains NES offrent, comme NetSolve depuis sa version 2.0, la possibilité au client d'interrompre définitivement une tâche ayant commencé son exécution, nous n'en tiendrons pas compte directement au sein du module d'ordonnancement en supposant des comportements probabilistes par exemple qui seraient par ailleurs difficilement quantifiables. Les méthodes que nous proposons sont déterministes.

4.3 Métriques d'ordonnancement

4.3.1 Introduction

Un *plan d'ordonnancement* décrit la succession des assignations des tâches aux ressources. Il est créé selon un algorithme déterminé qui tente de répondre aux contraintes éventuellement exprimées en l'optimisant selon des métriques données. L'optimisation se calcule à l'aide d'une fonction de coût établie par rapport aux objectifs à atteindre. Ces objectifs peuvent être de natures différentes. En effet, il y a de nombreux aspects à considérer comme ce que souhaite le client, les attentes d'utilisation de chaque ressource et le maintien des performances de l'intergiciel.

Les ressources de la grille sont virtualisées et de ce fait, les métriques de la grille doivent être considérées différemment des caractéristiques physiques que nous avons présentées à la section 3.2 et donner un plus haut niveau d'abstraction. D'un point de vue performance, des caractérisations des métriques des applications sont possibles. Par exemple, les plus simples critères pour une application parallèle sont le speedup, l'efficacité, la parallélisation et la granularité [Ess04]. Ces valeurs donnent suffisamment d'informations pour décrire une application globalement. Cependant, elles n'ont plus la même qualité dans une grille puisque les ressources sont hétérogènes et peuvent de plus changer entre deux exécutions. Ils nous faut donc considérer d'autres critères.

Dans ce qui suit, nous introduisons d'abord quelques éléments de notation puis nous définissons des termes couramment utilisés. Ensuite nous discutons des métriques qu'il est possible d'observer et d'optimiser, dont celles que nous jugeons pertinentes dans le cadre de notre travail.

4.3.2 Notations

On note a_i l'arrivée de la tâche i. Sa durée sur le serveur de calcul j non–chargé est notée $d_{i,j}$ et sa durée estimée $e_{i,j}$. La date de terminaison est notée $C_{i,j}$ et vaut $a_i + e_{i,j}$. Lorsque le serveur j est implicitement connu, on note $C_i = C_{i,j}$. Enfin, P_i est la ressource de calcul où la tâche i est affectée.

Cette notation est celle utilisée dans de nombreux travaux [MAS+99, MBS99, HSvL03, BSB+99]. Dans le cas *d'ordonnancement par lot*, les travaux qui comparent des heuristiques entre elles supposent généralement que les tâches sont exécutées *séquentiellement* sur les serveurs. La date à laquelle une tâche i commence son exécution sur le serveur de calcul j est donc la date à laquelle le serveur a terminé l'ensemble des tâches qui lui était déjà affecté.

4.3.3 Définitions

Du côté système, **l'utilisation des ressources** est déterminante puisqu'il faut les employer le plus possible pour les rentabiliser. Elle s'apprécie par exemple en mesurant les intervalles de temps où la ressource n'a pas été utilisée (on dit qu'elle est non–chargée, vide, libre ou *idle*) ou en calculant le **sumflow**. Du côté applicatif, on parle du **temps de réponse** ou **flow** d'une tâche pour désigner sa durée une fois celle–ci terminée et du **makespan** pour son temps de terminaison.

Sumflow [Bak74]. C'est la somme des durées des tâches qui ont été exécutées sur le serveur considéré. Pour le serveur j, c'est donc la quantité

$$sumflow = \sum_{i|P_i=j} (C_{i,j} - a_i)$$

Des problèmes de **répartition de charge** peuvent apparaître pour deux raisons. D'une part, un maximum de ressources doit être exploité puisqu'il est de peu d'intérêt d'impliquer dans un environnement de résolution des machines moins performantes si elles ne servent pas. D'autre part, une attention particulière doit être portée sur l'utilisation des serveurs les plus rapides afin qu'elle ne devienne pas excessive au risque de les faire s'écrouler (section 5.5.3).

Temps de réponse. Il permet de mesurer la réactivité du système et sa performance à assurer des délais d'exécution. On le définit pour chaque tâche i par

$$f_i = C_{i,j} - a_i$$

Makespan. Le *makespan d'un ensemble de tâches* S (regroupées ou non dans une application) est le temps de terminaison de la dernière tâche qui finit son exécution, i.e.,

$$makespan_S = \max_i C_i$$

4.3.4 Métriques d'ordonnancement et discussion

D'après D. G. Feitelson dans [FR98], le problème lié à la sélection d'une métrique de performance est que dans un système, plusieurs métriques peuvent être pertinentes pour différentes tâches. Par exemple, le temps de réponse est peut–être la métrique la plus importante pour des tâches interactives, tandis que l'utilisation du système est le plus important sur une machine ordonnancée par un mécanisme de soumission Batch. Pour un système Online, l'utilisation est largement déterminée par les dates d'arrivée et les besoins des tâches, ce qui laisse le temps de réponse comme métrique principale. Cependant, d'autres métriques peuvent également être surveillées.

L'utilisation d'une ressource peut être maximisée, notamment par l'usage du **sumflow** et de métriques à caractère économique calculées à partir du **coût associé à l'utilisation de chaque ressource** comme cela a été entrepris dans [ASGH95].

Slowdown et Stretch [BCM98]. Il semble que minimiser le temps de réponse privilégie les longues tâches, alors que les tâches courtes sont plus courantes. Une solution est de normaliser la valeur en utilisant le slowdown plutôt que la seule valeur du temps de réponse. Le **slowdown**, ou **stretch** est défini par le ratio de la durée de la tâche constatée à l'exécution et de sa durée sur le même serveur non chargé, c'est–à–dire

$$S_i = (C_i - a_i)/d_i$$

Le problème de cette métrique est que des tâches extrêmement courtes et qui sont fortement retardées conduisent à des valeurs importantes du slowdown. La solution pour un algorithme désirant optimiser le slowdown est d'utiliser une valeur seuil comme borne inférieure sur la durée des tâches : les tâches de durée plus courte que cette valeur sont considérées comme ayant cette valeur comme durée lors du calcul du slowdown.

Maxflow et MaxStretch [BCM98]. Le temps de réponse est une métrique observée sur la moyenne des flows des tâches exécutées. Cependant, les pires cas peuvent aussi être considérés. Parmi eux, on peut citer le **maxflow** et le **maxstretch** qui sont respectivement la durée la plus longue d'une tâche et le ralentissement maximum observé dans le système :

$$maxflow = \max_i (C_i - a_i)$$

$$maxstretch = \max_i S_i$$

Qualité de Service. Des travaux récents montrent aussi un intérêt grandissant sur l'utilisation de critères orientés vers la **qualité de service**. On la trouve définie de multiples façons. Dans [GÖD04], les auteurs introduisent les métriques de performance suivantes : la *timeliness* qui regroupe toutes les contraintes temporelles, parmi lesquelles on trouve la durée d'exécution ou **délai**, les contraintes sur **la date de terminaison** ou *deadlines*, la variation de la durée d'une tâche aussi appelée **la gigue** ; la *reliability* qui regroupe les aspects de sûreté, dans le sens où une longue tâche a plus de chance de ne pas terminer à cause d'une défaillance dans le système ; la métrique *versions* qui suppose des versions d'exécutables différentes avec des requis d'exécution différents ; la *priority* dans le but de donner des priorités différentes aux tâches lors de l'ordonnancement afin de faciliter la résolution des plus importantes. Dans [Jon02], l'auteur introduit le concept de *share–scheduling*. L'objectif est d'essayer de donner à un groupe de tâches (par exemple à toutes les tâches d'un client ou d'un département) une quote part définie de la capacité de traitement de l'ensemble du système multi–processeurs ou distribué. Cette quote part est appelée l'équité (*fairness*). On y trouve aussi les métriques d'**équité par groupe** (*group fairness*) et d'**équité par application** (*application fairness*).

Enfin, il est préférable de garantir une certaine qualité du comportement de l'algorithme d'ordonnancement, ce que nous appelons **la robustesse**. Elle peut se mesurer de deux façons : par des performances semblables sur des expériences de même type ou par la sensibilité de son comportement face aux erreurs d'estimation sur les performances de la tâche. Ainsi, on trouve dans [COBW00, HSvL03] une étude sur les performances des heuristiques proposées et testées en fonction de la dégradation de la qualité des estimations.

Pourcentage de tâches terminant plus tôt. Dans notre travail, nous avons principalement évalué la qualité des performances des heuristiques en les comparant par exemple sur le makespan qui est la métrique de référence et sur le sumflow qui permet de chiffrer l'utilisation des ressources. Mais nous avons tenté d'optimiser en même temps la performance que perçoit l'utilisateur de la grille de calcul : son souhait est *a priori* que son application (ou ses applications), qu'elle soit composée de tâche(s) indépendante(s) ou d'un graphe complexe, termine le plus tôt possible. Indépendamment du makespan, le temps moyen d'attente de la terminaison d'une requête devient alors un enjeu important. Nous avons donc défini *un critère de comparaison* entre deux heuristiques H_1 et H_2 sur la soumission de n tâches indépendantes : **le pourcentage de tâche terminant plus tôt**, c'est–à–dire

$$\frac{100 * \mid \{i | {C_i}^{H_1} < {C_i}^{H_2}\} \mid}{n}$$

où pour chaque tâche, les dates de terminaison obtenues par chacune des heuristiques sont comparées.

4.4 Algorithmes d'ordonnancement

4.4.1 Introduction

L'ordonnancement *Offline* optimal du makespan de tâches indépendantes sur des ressources distribuées est connu depuis longtemps pour être un problème NP–difficile [GJ79, All75]. Cela signifie que la complexité est telle qu'on ne sait pas actuellement trouver de solution avec un algorithme polynomial en fontion du nombre d'entrées (ici le nombre de tâches et de serveurs). L'ordonnancement de tâches en mode Batch ou Online est donc au moins aussi difficile puisque l'algorithme d'ordonnancement ne dispose pas de la totalité des informations.

Un algorithme d'approximation propose des garanties théoriques sur l'approximation maximale ou moyenne par rapport à une solution optimale du problème donné. Ainsi, dans [HSW96], les auteurs étudient les bornes sur la somme pondérée des dates de terminaison des tâches (c'est–à–dire $\sum \omega_i C_i$) pour des algorithmes statiques. Ils proposent une méthode d'ordonnancement par intervalle pour l'ordonnancement dynamique en mode Batch (l'ordonnancement considère dans ses calculs les intervalles de terminaison des tâches), afin d'assurer une date de terminaison à un facteur $4 + \epsilon$ de l'optimal.

L'objectif le plus fréquent de nombreux travaux est de minimiser le makespan de l'ensemble des tâches soumises. Des études apparaissent donc pour fournir des algorithmes multi–critères qui visent à proposer des ordonnancements optimisant simultanément plusieurs métriques à la fois. Ainsi dans [DEMT04], les auteurs s'attachent à l'ordonnancement de tâches malléables [Fei99] : le nombre de ressources de calcul à charge d'exécuter une tâche malléable se détermine avant son exécution et ne change pas avant sa terminaison. Ils proposent ainsi un algorithme Batch bi–critères qui permet de réduire efficacement le temps de réponse moyen tout en garantissant un makespan à un ratio de $3 + \epsilon$ de l'optimum.

Il existe d'autres types d'algorithmes qui ne garantissent pas l'optimalité de la solution qu'ils proposent mais donnent de bons résultats en général. Un important travail a été réalisé ces dernières années pour trouver des **heuristiques d'ordonnancement** efficaces pour la grille. La conception d'une heuristique doit prendre en compte la disponibilité de connaissances sur les applications soumises comme le graphe de précédence, la durée des tâches, etc. ainsi que les connaissances sur l'environnement et les modalités de soumission (ordonnancement des tâches en mode Batch, par fenêtre coulissante ou à la volée). S'il existe un grand nombre d'heuristiques, dont nous présentons ci–après un échantillon des plus connues, celles mises en œuvre dans les intergiciels sont beaucoup moins nombreuses et généralement tournées vers l'optimisation du makespan de l'ensemble des tâches soumises. Des études se sont donc orientées vers la possibilité de fournir aux utilisateurs une certaine qualité de service (*QoS, Quality of Service*), comme la minimisation de l'utilisation de la bande passante à partir d'une variante de l'heuristique statique min–min dans [HSvL03].

Avec le contexte GridRPC arrivent de nouveaux défis venant s'ajouter au traditionnel problème de l'ordonnancement : les ressources ne sont pas seulement distribuées et hétérogènes mais aussi partagées entre plusieurs utilisateurs. De plus, l'ordonnancement est continu, les tâches peuvent interférer entre elles, sur les liens réseau mais aussi sur les hôtes puisqu'elles se partagent les ressources de calcul.

Nous avons vu dans la section 4.2.3 qu'il existe deux principaux modes d'ordonnancement dynamique. La suite de la section est donc organisée de la façon suivante : nous présentons dans un premier temps quatre heuristiques Batch parmi les plus connues puis trois *Online*, directement issues du mode statique.

4.4.2 Batch

Les heuristiques Batch que nous présentons ici fonctionnent sur le même schéma illustré dans l'algorithme 1. Les algorithmes de cette section sont détaillés dans l'annexe A. Pour chaque tâche, la date de terminaison est estimée sur chacune des ressources. Ensuite, le résultat d'une fonction de coût f_1 sur une métrique donnée est calculé à partir des dates obtenues et est stocké dans val_i. Lorsque toutes les tâches ont été examinées, c'est–à–dire les $|S|$ valeurs val_i affectées, le choix de la tâche à allouer est effectué à l'aide d'une deuxième fonction de coût f_2 sous–entendue dans le choix de la «meilleure» valeur val_i. La tâche sélectionnée est ensuite allouée à la ressource où elle termine le plus tôt et retirée du sac de tâches restantes. Les prochaines estimations du temps de terminaison sur cette ressource prendront en compte un début d'exécution au moins supérieur à la prédiction de la date de terminaison obtenue à cette itération.

Algorithme 1: Schéma des algorithmes Batch min–min, min–max, Sufferage et XSufferage

Données : L'ensemble S des tâches soumises et non affectées
Résultat : L'ordonnancement des tâches sur les ressources
tant que *il reste une tâche non affectée* $(S \neq \emptyset)$ **faire**
    pour chaque *tâche i à ordonnancer* **faire**
        pour chaque *ressource de calcul j* **faire**
            Calculer $C_{i,j}$, la date de terminaison de la tâche i sur la ressource j;
        Calculer $val_i = f_1(C_{i,1}, C_{i,2}, \ldots)$;
    Choisir i_0 donnant "la meilleure" valeur val_i;
    Affecter la tâche i_0 sur la ressource r où elle termine le plus tôt, c'est–à–dire r telle que $C_{i_0,r} = \min_j C_{i_0,j}$;

min–min

La première fonction f_1 de l'heuristique min–min [BASM01] est l'opérateur min : elle retourne l'estimation de la plus petite date de terminaison de la tâche considérée. La fonction f_2 correspondant à la notion de "meilleure" est elle aussi l'opérateur min. On obtient ainsi la tâche qui termine le plus tôt. L'objectif désiré est de réduire globalement le makespan en commençant par l'exécution des tâches qui terminent le plus tôt.

max–min

Comme pour min–min, la fonction f_1 de max–min retourne le minimum des estimations de la date de terminaison de la tâche regardée. En revanche, la "meilleure" valeur, donnée par f_2, est le maximum des dates de terminaison val_i. On obtient ainsi la tâche qui termine le plus tard. L'objectif est de réduire le makespan en plaçant les tâches de telle sorte que les tâches les plus rapides «cachent» le makespan des plus lentes déjà affectées.

Sufferage

Sufferage est basée sur l'idée qu'une tâche doit être affectée à la ressource où elle «souffrirait» le plus si elle n'y était pas affectée. La valeur de «souffrance» d'une tâche est calculée grâce à f_1 et est définie comme la différence entre ses deux meilleurs temps de terminaison. Ensuite, l'algorithme choisit la tâche qui souffre le plus de ne pas être affectée sur la ressource minimisant son makespan et qui doit être en conséquence affectée en priorité. La fonction f_2 retourne donc le maximum des souffrances val_i. L'objectif est d'ordonnancer les tâches par ordre croissant selon la perte de performance estimée si d'autres affectations étaient effectuées avant celle de la tâche considérée. Ainsi, on espère accroître les performances d'un nombre plus important de tâches.

XSufferage

XSufferage est une extension de Sufferage qui tente de prendre en compte la localité des données et l'effet de la bande passante, généralement plus importante au sein de chaque cluster constituant la grille. Pour cela, le temps de terminaison pour chaque cluster, c'est–à–dire pour les ressources constituant le cluster, est calculé. La valeur de souffrance est ensuite obtenue en évaluant f_1, donnant la différence des deux meilleurs temps de terminaison au niveau du cluster. f_2 retourne la tâche qui souffre le plus au niveau du cluster. Elle est allouée en priorité et placée sur la ressource où elle finit le plus tôt dans le cluster.

4.4.3 Online

Les heuristiques d'ordonnancement présentées ci–après, OLB MET et MCT, font référence dans la littérature. Elles ont initialement été introduites dans le projet Smartnet [FGA+98] et ont été largement étudiées.

OLB : Opportunistic Load Balancing

Lorsqu'elle est ordonnancée par OLB [FGA+98], chaque nouvelle requête est placée sur la ressource qui deviendra non–utilisée la première, quels que soient ses besoins en temps de calcul. Si plusieurs affectations sont possibles, la ressource est choisie arbitrairement : dans l'algorithme 2, en cas d'égalité, le premier traité est choisi.

Algorithme 2: Heuristique d'ordonnancement OLB

! **Données** : une tâche t
Résultat : Le placement de t sur une ressource
début
  pour chaque *ressource de calcul* j **faire**
    Calculer $idle_j$, la date où la ressource sera libre;
  Calculer le premier serveur libre $s|idle_s = \min_j idle_j$;
  Affecter la tâche t au serveur s;
fin

MET : Minimum Execution Time

MET place la tâche nouvellement soumise sur la ressource où elle nécessite le moins de temps de calcul, indépendamment de la charge actuelle de l'hôte. L'objectif est d'allouer chaque tâche à la machine qui lui convient le mieux. Ceci peut conduire à de sérieux problèmes d'équité de charge entre les machines et n'est pas applicable pour des plate–formes hétérogènes où les tâches demandent un temps d'exécution relatif à la vitesse de la machine : il doit y avoir dans la plate–forme des affinités entre des tâches et des types d'architecture (vectorielle par exemple).

Algorithme 3: Heuristique d'ordonnancement MET

! **Données** : une tâche t
Résultat : Le placement de t sur une ressource
début
  pour chaque *ressource de calcul* j **faire**
    Calculer $d_{t,j}$, la durée de la tâche t sur le serveur j dédié;
  Calculer $s = \min_j d_{t,j}$ le serveur où la tâche est la plus courte;
  Affecter la tâche t au serveur s;
fin

MCT : Minimum Completion Time

MCT est une variante de l'algorithme glouton utilisé dans le projet SmartNet [FGA+98] : il correspond à l'algorithme min–min dans le cas Online. Son objectif est de minimiser le makespan global de l'ensemble des soumissions en choisissant la ressource de calcul qui minimise le temps de terminaison de chaque nouvelle requête. Ce temps est évalué sur chaque ressource disponible.

Cette heuristique sert fréquemment de référence pour comparer des heuristiques entre elles, généralement par des simulations où les requêtes soumises sont indépendantes et exécutées séquentiellement sur les serveurs [SA99]. Cette heuristique montre de très bonnes performances pour une grande simplicité d'implantation [MAS+99].

Algorithme 4: Heuristique d'ordonnancement MCT

! **Données** : une tâche t
Résultat : Le placement de t sur une ressource
début
  pour chaque *ressource de calcul j* **faire**
    Calculer $C_{t,j}$, la date de terminaison de la tâche t sur le serveur j dédié;
  Calculer $s = \min_j C_{t,j}$ le serveur où la tâche termine le plus tôt;
  Affecter la tâche t au serveur s;
fin

4.4.4 Ordonnancement pour le GridRPC

Dans les travaux qui ont été présentés ici, les algorithmes supposent toujours que les exécutions des tâches assignées à une ressource sont séquentialisées. Pourtant, dans les implantations des Nes (NetSolve ou Diet), une tâche commence son exécution dès que la totalité des données nécessaires au calcul a fini d'être transférée sur le serveur. L'heuristique employée est souvent une variante de MCT. Elle possède des mécanismes de correction de charge afin de prendre en compte les décisions de placement prises entre deux rapports de charge. Ainsi, deux tâches soumises en même temps ne sont pas nécessairement affectées au même serveur.

Cependant, le choix de MCT est critiquable puisque son orientation makespan la destine normalement à un environnement où les tâches font toutes partie d'une même application, or un environnement est multi–utilisateurs. De plus, minimiser la date de terminaison de chaque tâche n'implique pas de bonnes performances sur le makespan total, bien qu'en mode séquentiel elle donne de bons résultats [MAS+99]. J. Sitdham fait la différence dans [Sti85] entre les stratégies d'optimisation individuelles et sociales : les premières s'intéressent à la terminaison de chaque tâche et la seconde, au temps de réponse moyen. Il a trouvé que les politiques individuelles chargeaient les serveurs les plus rapides, ce qui est facilement observable avec MCT (section 5.5.3).

Les mécanismes de correction de charge sont pertinents dans le sens où ils donnent une courte mémoire dans la prise de décision. Ils ne sont cependant pas suffisants pour assurer la pérennité des décisions prises jusqu'à présent : une tâche peut commencer son exécution alors qu'une autre n'a pas terminé. L'objectif à atteindre ne dépend pas seulement des futures allocations mais également du suivi de ce qui a été fait jusqu'à présent. Grâce à des expériences de simulation, Weissman a étudié dans [Wei96] l'influence du partage de la puissance du processeur sur les retards occasionnés sur les tâches soumises. L'heuristique doit prendre en considération ces éléments dans ses décisions et requiert un module de prédiction de performances capable de lui fournir plus d'informations que la seule date de terminaison de la dernière requête : c'est l'objet du prochain chapitre.

4.5 Conclusion

Après avoir exhibé une classification possible de l'ordonnancement, nous avons positionné notre travail en décrivant les critères propres au GridRPC qui devaient le caractériser : l'ordonnancement se fait au niveau global et doit gérer des ressources hétérogènes et distribuées. Le module d'ordonnancement est implanté dans l'agent de l'environnement. L'algorithme n'est pas probabiliste et est évalué à chaque soumission de tâche, laquelle ne peut migrer. L'ordonnancement est donc centralisé, déterministe, dynamique et continu.

Puis, nous avons présenté des métriques qui pouvaient être prises en compte lors des décisions d'affectation ou observées pour établir des comparaisons sur les performances escomptées entre deux algorithmes et ainsi les départager. Notre travail se veut multi–critères. Nous souhaitons minimiser le makespan des applications de chaque client tout en fournissant certaines qualités de service : le temps de réponse doit être bas et les tâches doivent avoir une plus grande chance de terminer plus tôt en utilisant notre algorithme qu'un autre.

Des algorithmes d'ordonnancement qui font référence ont ensuite été décrits : quatre algorithmes *Batch* et trois algorithmes *Online*. Une variante de MCT (*Minimum Completion Time*) est parmi les plus implantées dans les environnements de soumissions actuels pour sa performance et sa facilité de mise en œuvre. Cependant, elle ne dispose pas de mémoire et ne permet pas d'assurer la pérennité de ses décisions : la pertinence de ses choix précédents peut être remise en cause dès lors que les affectations présentes et futures peuvent interférer avec eux.

Nous visons l'ordonnancement d'une plate–forme de type server Grid reposant sur le GridRPC. Le module d'ordonnancement se situe dans l'agent. Nous n'avons donc pas de connaissance particulière sur des relations de précédence entre les requêtes soumises et l'ordonnancement repose sur une heuristique. Nous supposerons par la suite que les serveurs de calcul utilisent un ordonnancement local round–robin et que chaque requête émise peut être exécutée sur chaque serveur, c'est–à–dire que chaque serveur fournit l'ensemble des services disponibles. Nous souhaitons considérer l'historique des affectations des tâches sur les ressources afin de «donner de la mémoire» à l'heuristique implantée dans l'agent. Ainsi, la notion de qualité de service peut être envisagée puisque les décisions prises jusqu'alors sont considérées dans le calcul des affectations futures.

Chapitre 5

Ordonnancement multi–critères pour le modèle client–agent–serveur

Table des matières

5.1 Motivation

Les ordonnanceurs des environnements de calcul GridRPC font habituellement appel à des modules de prédiction de performances reposant sur les informations délivrées par des senseurs installés sur les différentes ressources de la plate–forme (section 3.4.2). La date de terminaison de la requête a toujours été [MAS+99, COBW00] et reste [DCB02] prédominante lorsqu'il s'agit d'affecter la tâche à une ressource ou à un groupe de ressources [CDS04, EHS+02] quel que soit l'environnement considéré.

Dans [Wei96], Weisman observe *le délai* à l'aide d'expériences de simulation, c'est–à–dire la différence entre la date de terminaison lors de la soumission et la date de terminaison réelle. Il déduit l'importance de la perturbation des tâches les unes sur les autres lors de la prise de décision de l'ordonnanceur. Nous avons observé en utilisant NetSolve sur des ressources distribuées dans le laboratoire, que la charge des serveurs n'est pas seulement modifiée par une source externe à l'environnement : pour des raisons de performance, les serveurs les plus rapides sont aussi les plus démandés. Ainsi, plusieurs tâches peuvent être affectées et exécutées simultanément sur un même serveur. L'exécution des tâches étant concurrente sur les serveurs, on doit donc discerner la charge externe à la plate–forme, due à l'exploitation de certaines ressources par un utilisateur, de la charge interne qui est due aux décisions de placement de l'ordonnanceur. La prédiction de performances donc prendre en compte les informations dont l'environnement a connaissance.

Nous avons tenté avec le gestionnaire de l'historique des tâches, que nous dénommerons par la suite HTM pour *Historical Trace Manager*, d'instaurer une réelle complémentarité entre la prédiction de performances et la prise de décisions. Le HTM est un module de prédiction de la durée des tâches soumises à l'environnement de résolution de problèmes. Il permet de déduire la charge interne de la plate–forme de façon non-intrusive et donne immédiatement des réponses sur son état. Comme l'ordonnanceur, il est directement intégré au fonctionnement de l'agent.

Parmi les objectifs du HTM, nous souhaitons en particulier voir apparaître les points suivants. Dans un environnement temps partagé, la charge d'une machine n'est pas nécessairement due à une charge externe : deux tâches peuvent être exécutées simultanément sur un même serveur. Aussi, le HTM doit enregistrer les informations relatives à chaque requête comme sa durée et l'affectation proposée par le module d'ordonnancement. Il doit permettre de prévoir *a priori* les prochaines variations de la charge CPU d'un serveur occasionnées par l'intergiciel. Le HTM doit rendre compte des interférences que va occasionner l'affectation d'une tâche sur un serveur. Plus l'estimation est précise, plus l'ordonnanceur appliquera sa politique de placement correctement. Grâce aux informations qu'il a à sa disposition, le HTM doit rendre possible une estimation de la date de terminaison *de chaque tâche* dans le système à n'importe quel instant. Il doit donc prendre en compte la perturbation que les tâches ont les unes sur les autres dans un environnement temps–partagé. L'exécution d'une nouvelle tâche peut dramatiquement perturber le système et remettre en cause les précédentes affectations. Le module d'ordonnancement doit avoir accès à ces informations. L'heuristique utilisée peut ainsi optimiser son choix dans l'état actuel du système tout en assurant une certaine pérennité de ses décisions dans un futur proche. Le HTM doit rendre possible un ordonnancement qui n'est plus fondé sur l'hypothèse qu'une seule et unique application est soumise et exécutée sur la plate–forme : il faut pouvoir prendre en compte les besoins de chacune sans que cela ne se traduise par une importante baisse de performance pour les autres. Les heuristiques doivent avoir à disposition différentes informations relatives au système afin de ne pas limiter leurs choix en les optimisant selon un seul critère : les besoins sont tels que des critères de qualité de service ou de consommation des ressources doivent pouvoir être pris en compte. Grâce aux informations données par le client, par exemple la taille des données en entrée, grâce aux caractéristiques du problème demandé (comme une estimation de la durée sur le serveur à vide ou la complexité de l'algorithme ou encore le monôme de plus haut dégré du polynôme de complexité) et par l'observation des temps d'exécution de problèmes analogues, la qualité des estimations doit pouvoir s'affiner avec le temps : le HTM doit «apprendre» son environnement. La plate–forme n'étant pas nécessairement dédiée, il faut que le HTM fournisse continuellement des informations pertinentes dans ce cadre. Le procédé d'évaluation doit être particulièrement rapide afin de garantir une grande réactivité de l'agent : la prise de décision doit être négligeable devant les durées des tâches et devant les délais d'évolution de la plate–forme.

Dans ce qui suit, nous présentons tout d'abord les mécanismes mis en jeu et leurs implantations dans une plate–forme réelle dans la section 5.2. Puis, nous introduisons dans la section 5.3 des heuristiques multi–critères qui reposent sur les informations du HTM. Nous examinons ensuite dans la section 5.4 leurs performances sur des expériences de simulations, puis, dans la section 5.5 celles observées lors de multiples expériences effectuées dans des environnements réels. Pour finir, nous concluons à la section 5.6.

5.2 Gestionnaire de l'historique des tâches : description et algorithme

5.2.1 Observations préliminaires

Dans les systèmes temps–partagé, chaque tâche doit progresser dans son exécution. Ainsi, sur les ressources instanciant une variante du système Unix, chaque processus est exécuté pendant un tronçon de temps très bref donnant l'illusion qu'ils sont exécutés en parallèle, ce que l'on nomme *le multitâches*. Dans [MP02], les auteurs montrent avec des chaînes de Markov qu'en théorie, les algorithmes d'ordonnancement locaux des ressources sous Linux, Solaris ou Irix peuvent se modéliser comme du **round–robin** : chaque tâche de calcul acquiert équitablement la puissance de calcul du serveur et progresse concurremment aux autres, «à la même vitesse». Nos soumissions de tâches de calcul lors d'expériences réelles sur ces mêmes systèmes d'exploitation (sous Linux noyaux 2.4 et 2.6 [BC00, BC02, Aiv02, Mol] et Solaris SunOS 5.7) viennent confirmer ces résultats dans la pratique.

5.2.2 Description

La variation de la charge constatée lors des rapports des senseurs distribués sur les ressources de la plate–forme ne provient pas nécessairement de leur utilisation par des personnes extérieures à l'environnement : à cause de l'hétérogénéité des ressources et pour des raisons de performance, si un serveur conduit à minimiser la somme des durées de transferts et des calculs, il sera choisi plus fréquemment que les autres. Comme les tâches commencent leur exécution dès que les données en entrée sont transférées, plusieurs tâches peuvent se partager la ressource de calcul à un instant donné. Les dates de terminaison des tâches déjà affectées s'en trouvent retardées et les décisions d'ordonnancement précédentes peuvent perdre leur pertinence : le temps de réponse s'accroît, le makespan de l'application risque d'augmenter, etc.

Selon la modélisation d'une application présentée à la section 3.3.1, une tâche ne communique pas en dehors des phases de transmission des données en entrée et en sortie. Dans le contexte GridRPC, les requêtes soumises à l'environnement sont des tâches intensives en calcul. Aussi, nous avons modélisé le partage de chaque ressource pour en déduire des informations pertinentes à donner à l'ordonnanceur. Le gestionnnaire de l'historique des tâches prend une modélisation des ressources de la plate–forme et y simule l'exécution des tâches (communication et calcul) en fonction des informations que les autres modules de l'agent lui communiquent.

Comme dans Simgrid, que nous avons présenté à la section 3.6.2, les ressources sont modélisées en temps–partagé par défaut, afin de simuler l'ordonnancement local similaire à *round-robin* des stations UNIX–*like*. Ainsi, si deux tâches sont soumises et exécutées sur le même serveur alors que la date d'arrivée de la deuxième est inférieure à la date de terminaison de la première, elles partagent en particulier la puissance de calcul de façon équitable. L'avantage de la simulation est qu'elle est totalement non-intrusive et les décisions de l'ordonnanceur effectuées à un instant donné sont directement prises en compte pour les futures estimations.

Grâce à la simulation des requêtes sur la plate-forme modélisée, le HTM est capable de répondre aux diverses demandes d'information du module d'ordonnancement. Notons dès à présent que ses informations permettent de donner une mémoire aux heuristiques d'ordonnancement et de saisir d'autres critères que la date de terminaison de la dernière tâche généralement utilisée dans leurs décisions, par exemple le temps de réponse escompté. Cependant, il faut naturellement s'assurer que la simulation est cohérente avec la réalité et que les estimations ne se dégradent pas avec certains facteurs comme le temps ou les dépendances du graphe des tâches. Ceci fait l'objet du chapitre suivant.

5.2.3 Notations

Nous utiliserons dans tout ce qui suit les notations suivantes. Elles correspondent à ce que l'on peut observer dans la figure 5.1.

a_i est l'arrivée de la tâche i dans le système. T'_i est sa date de terminaison obtenue par simulation dans l'état courant du système et C_i est la date réelle de terminaison, c'est–à-dire celle observée *post-mortem* (après la fin de l'expérience). Le HTM peut simuler l'exécution d'une nouvelle tâche n et donner les nouvelles dates de terminaison T_i de toutes les tâches $i \leq n$ affectées à la ressource. On définit pour tout $k \leq n$, la perturbation $\delta_k = T_k - T'_k$ occasionnée par l'affectation de la nouvelle tâche sur toutes les tâches en train d'être exécutées sur la ressource. Par extension, on dira par la suite que δ_k est la perturbation *engendrée par la tâche n*. De plus, on dira que le temps restant de la tâche k est la valeur $D_k = T_k - a_n$, pour tout $k \leq n$; cette valeur est généralement différente du temps restant de la tâche sur la ressource à vide. Enfin, on notera $P(i)$ la ressource où la tâche i est affectée et d_i sa durée totale sur la ressource à vide.

5.2.4 Modèle

Pour schématiser l'ordonnancement round-robin, on suppose que la ressource peut exécuter simultanément et équitablement l'ensemble des tâches qui lui est assigné. La durée des changements de contexte est donc considérée comme négligeable par rapport aux durées des tâches. Une ressource modélisée par le HTM est donc construite autour du macro modèle suivant : s'il y a n tâches se partageant la ressource à un instant, chacune d'elles profite de $1/n$ de la puissance crête de la ressource. Ainsi, nous avons représenté en haut de la figure 5.1 le diagramme de Gantt illustrant l'état de la ressource au temps de soumission de la nouvelle tâche (c'est–à–dire à $t = a_3$, date d'arrivée de la tâche 3). On voit les conséquences de l'affectation de cette nouvelle tâche sur le diagramme présenté en bas de la figure. Les trois tâches se partagent la ressource uniformément jusqu'à la terminaison de l'une d'elles (ici jusqu'à T_1 à laquelle termine la tâche 1), les autres disposant de 50% de la ressource ensuite jusqu'à la date T_2 après laquelle la tâche 3 dispose de la totalité de la puissance de la machine.

Le HTM permet de simuler une tâche quelle que soit sa date d'arrivée dans le système. On appelera *date courante* la date actuelle, généralement la date de soumission de la dernière requête, qui doit être à ce moment ordonnancée : les tâches et leurs caractéristiques sont enregistrées au fur et à mesure de leur soumission et la date courante prend successivement pour valeur leur date d'arrivée dans le système dès qu'elles sont allouées à un serveur.

Ce modèle simple permet d'effectuer rapidement les multiples calculs. Il est cependant suffisamment précis pour que les informations rendent compte correctement de ce qu'il se passe en réalité sur la plate-forme.

5.2.5 Algorithme et complexité

Une requête se décompose en une tâche de transfert des données en entrée du problème, une tâche de calcul et une de transfert des données en sortie. Pour chaque tâche, on

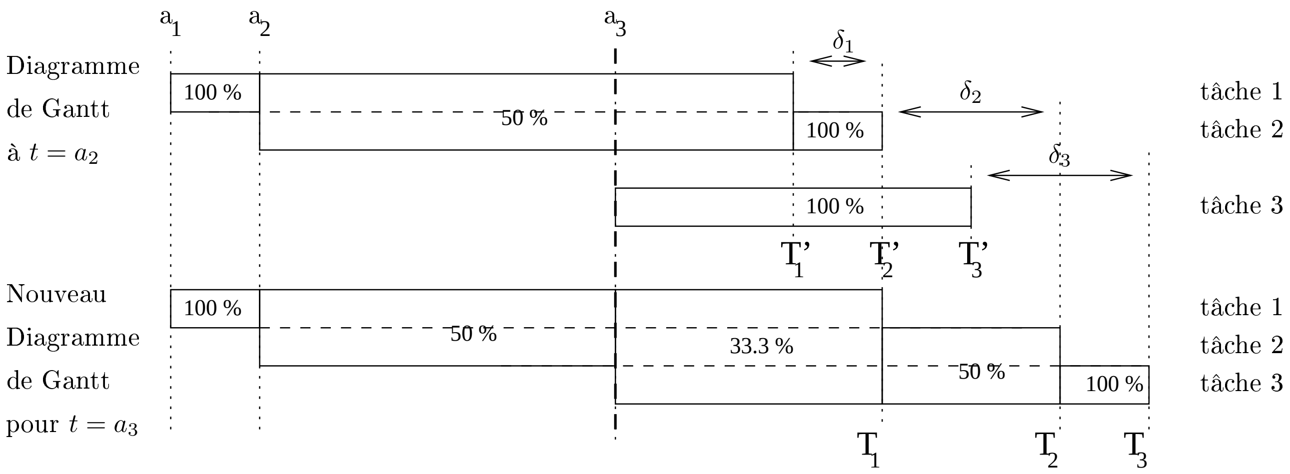


FIG. 5.1 – Notations du gestionnaire de l'historique des tâches, le HTM

distingue en particulier les valeurs : *date d'arrivée*, *date de terminaison*, *durée sur la ressource non–chargée* et *durée restante* (nécessaire à la tâche pour terminer si elle disposait *seule* de la ressource). La date d'arrivée d'une tâche de calcul est la date de terminaison de la tâche de transfert correpondante. De même, la date d'arrivée d'une tâche de transfert des données en sortie est la date de terminaison de la tâche de calcul.

Pour simplifier, l'algorithme 5 que nous donnons explique la simulation opérée sur *un serveur de calcul* comportant N tâches. Notons que la nouvelle requête *n'est pas* nécessairement la dernière à commencer son exécution sur le serveur et que les tâches affectées sur le serveur n'ont pas nécessairement des indices contigus.

Lors de la soumission d'une requête, un algorithme similaire est exécuté : il prend en compte les parcours des tâches de communication jusqu'au serveur cible puis le retour vers le client.

Explications. Passé l'initialisation, la boucle principale simule l'arrivée de chacune des tâches de la ressource que l'on appelle *tâche courante*. Dans ce but, on y trouve deux étapes majeures qui se traduisent chacune par une boucle. la première consiste à mettre à jour les données des tâches dont on a déjà simulé l'arrivée. Cela se traduit par mettre à zéro la quantité *durée restante* des tâches qui s'avèrent être finies avant l'arrivée de la tâche courante (leur date de terminaison est déjà obtenue des itérations précédentes de la boucle principale). Il faut aussi considérer les cas où des tâches peuvent s'exécuter, après que d'autres aient terminé. Pour faciliter la compréhension, nous donnons dans la figure 5.2 un exemple où les tâches sont ordonnées selon σ, la tâche courante est $t_{\sigma^{-1}(5)} = t_7$ et les ensembles sont indiqués. Ainsi, Finies= $\{t_{12}, t_3\}$ et EnCours= $\{t_4, t_1\}$. Dans cet exemple, f= $\sigma^{-1}(1) = 12$. Lors de la deuxième étape, l'algorithme simule l'exécution de l'ensemble des tâches déjà arrivées afin d'estimer leur date de fin.

Algorithme 5: Simulation d'une tâche sur la modélisation de la plate-forme

Données : La date d'arrivée a_R de la nouvelle tâche t_R ;
La durée d_R de la tâche sur le serveur non–chargé ;
Les informations sur les N tâches déjà affectées sur le serveur ;
Le diagramme de Gantt sur le serveur G ;

Résultat : Diagramme de Gantt résultant sur le serveur

début

- **si** $N = 0$ **alors retourner** $T_R \leftarrow a_R + d_R$;
- Soit σ la permutation donnant la suite d'indices des tâches de P(t_R) comprenant t_R, classées par date d'arrivée ;
- **pour chaque** $1 \leq i \leq N+1$ **faire**
  - Initialiser $T_{\sigma(i)}$ à $a_{\sigma(i)} + d_{\sigma(i)}$ et la durée restante à effectuer $d^r_{\sigma(i)}$ à $d_{\sigma(i)}$;
- **pour chaque** $1 \leq i \leq N+1$ **faire**
  - /* met à jour les données sur les tâches par rapport à $a_{\sigma(i)}$ */;
  - Finies = { tâches $t_{\sigma(j)}$ tq $T_{\sigma(j)} < a_{\sigma(i)}$ et $d^r_{\sigma(i)} > 0$} ;
  - EnCours $\leftarrow$ {$t_{\sigma(j)}$ | tel que $\sigma(j) < \sigma(i)$ et $T_{\sigma(j)} > a_{\sigma(i)}$} ;
  - Calculer restante $\leftarrow$ |EnCours| ;
  - **si** |Finies| >0 **alors**
    - Soit f tel que $t_f \in$ Finies et $T_f = \max_k T_{\sigma(k)}$;
    - **pour chaque** $t_j \in$ Finies **faire** $d^r_j \leftarrow 0$;
    - Calculer la durée sans évènement $E = a_{\sigma(i)} - T_f$;
    - **si** restante $\neq 0$ **alors**
      - **pour chaque** $t_j \in$ EnCours **faire** $d^r_j \leftarrow d^r_j - E/$restante;
  - **si** restante $\neq 0$ **alors**
    - Calculer la durée sans évènement $E = a_{\sigma(i)} - a_{\sigma(i)-1}$;
    - **pour** $t_j \in$ EnCours **faire** $d^r_j \leftarrow d^r_j - E/$restante ;
  - /* Calcul du diagramme de Gantt par simulation des exécutions */ ;
  - Classer EnCours selon d^r_j décroissant ;
  - **pour chaque** $t_j \in$ EnCours **faire**
    - copiedurée[$\sigma(j)$] $\leftarrow d^r_{\sigma(j)}$;
  - **pour chaque** $t_j \in$ EnCours **faire**
    - **pour chaque** $t_k \in$ EnCours tel que $k \geq j$ **faire**
      - copiedurée[k] $\leftarrow$ copiedurée[k] $-$ copiedurée[j] ;
      - $T_k \leftarrow T_k+$ copiedurée[j] ;
- **retourner** $T_{\sigma(i)}$ *pour* $1 \leq i \leq N+1$;

fin

À chaque nouvelle tâche, la simulation sur une ressource requiert $O(n^2)$ opérations, n étant le nombre de tâches sur la ressource. Chaque tâche pouvant entraîner la création d'une tâche sur une autre ressource (tâche de calcul après une tâche de communication par exemple), à chaque soumission l'algorithme demande donc $O(rN^2)$ opérations, où r

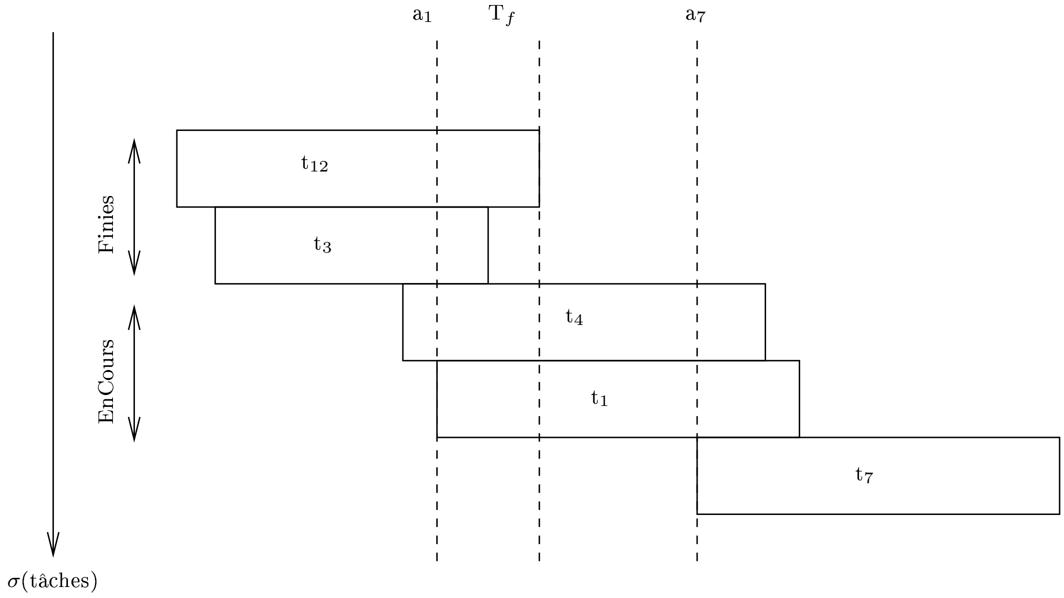


FIG. 5.2 – Exemple d'exécution de l'algorithme du HTM

est le nombre de ressources de la plate-forme et N le nombre de requêtes, pour simuler la requête.

Des mécanismes peuvent être utilisés pour réduire la complexité de l'algorithme, tels une simulation d'une partie de la plate-forme si sa structure s'y prête ou grâce au stockage d'informations intermédiaires permettant de ne pas avoir à tout resimuler.

5.3 Heuristiques

Toutes les heuristiques que nous avons construites et testées reposent sur les informations que leur propose le HTM. Ses estimations sont obtenues par simulation de la nouvelle requête sur la modélisation de la plate–forme. Cette évaluation prend donc en compte les délais occasionnés par l'exécution concurrente des tâches sur les ressources.

Le but de ces heuristiques d'ordonnancement est d'améliorer les performances de celles qui sont habituellement utilisées dans les environnements de résolution de problèmes, notamment MCT, que nous avons décrit dans la section 4.4. Contrairement à [BW97, COBW00, CLZB00, IO98, WZ97] par exemple, les seules informations dont elles disposent sont les informations statiques et dynamiques que nous avons définies à la section 3.1 propres au contexte GridRPC. Aucun renseignement n'est fourni sur les relations de dépendance entre les requêtes soumises ou sur l'arrivée ultérieure d'autres requêtes. Aucune information n'est utilisée sur l'identité de l'utilisateur. En revanche, la notion de deadline pour chaque tâche peut être envisagée.

Contrairement à [MAS+99, BSB+99], nous souhaitons considérer d'autres critères dans l'évaluation du meilleur serveur que le simple makespan, comme ceux que nous avons décrits dans la section 4.3. En effet, les heuristiques mono-critère basées sur le makespan supposent que l'ensemble des tâches forme une unique application dont les performances sont axées sur le point de vue d'un unique utilisateur. Ceci n'est naturellement pas le cas lors de l'utilisation d'un environnement de résolution.

Une heuristique décide du serveur de la plate-forme où affecter chaque nouvelle requête selon une fonction de coût à préciser afin de maximiser plusieurs critères *simultanément*. Parmi nos objectifs, nous souhaitons qu'elle donne une certaine qualité de service aux différents clients sans que ceux-ci soient expressément identifiés ou que l'affiliation d'une tâche dans une application soit donnée, tout en conservant l'optique de minimiser le temps de terminaison de l'ensemble des tâches. Pour cela, les informations du HTM doivent assurer la pérennité des choix précédents : il est en effet possible de considérer la perturbation occasionnée par l'affectation de la nouvelle tâche sur le serveur afin de ne pas trop retarder la terminaison des tâches en cours d'exécution.

5.3.1 Historical MCT : HMCT

Inspirée de la stratégie de placement de MCT, HMCT alloue la requête au serveur sur lequel la simulation indique qu'elle termine le plus tôt (algorithme 6) : on espère ainsi minimiser la date de terminaison de l'application, c'est–à-dire le makespan, en minimisant le temps de terminaison de chaque nouvelle requête qui est potentiellement la dernière requête. Tout comme pour MCT, ceci suppose donc que toute requête est une composante d'une unique application et que c'est la date de terminaison de la dernière requête qui donne le makespan.

Algorithme 6: Algorithme de HMCT

Données : Le HTM dispose du diagramme de Gantt de chaque ressource;
Le HTM dispose de la modélisation de la plate–forme;
Résultat : Le serveur où sera affectée la requête
début
  pour chaque *nouvelle tâche t* **faire**
    pour chaque *serveur j capable de résoudre la tâche* **faire**
      Demander au HTM de calculer T_j la date de terminaison de la tâche t si t est exécutée sur le serveur j;
    Déterminer j_0 qui minimise T_j;
    Prévenir le HTM de l'affectation de la requête sur le serveur j_0;
    retourner *[j_0]*;
fin

Comme nous le verrons dans la section 5.5.3, le comportement notable de cette heuristique est qu'elle tend à surcharger les serveurs les plus rapides au risque de les saturer.

Cela profite donc au fournisseur s'il y en a un, puisque leurs utilisations sont aussi les plus chères. En revanche, cela peut signifier des retards pour les clients.

5.3.2 Advanced Historical MCT : AHMCT

Nous venons de voir que l'usage de HMCT suppose entre autres que la date de terminaison de la dernière soumission fait le makespan. Mais MCT [MAS+99] repose initialement sur un ordonnancement local FCFS (*First Come First Served*). Selon ce modèle d'exécution, soit le makespan devient la date de terminaison de la nouvelle tâche sur l'ensemble des serveurs et ce choix est optimum car il minimise bien le makespan (la tâche terminerait plus tard en étant affectée sur un autre serveur), soit ce n'est pas le cas et le choix est optimum car l'ancienne valeur du makespan, qui était minimum, n'est pas modifiée. Ainsi, affecter la tâche au serveur où elle termine le plus tôt minimise le makespan dans l'état actuel de la plate-forme.

Exemple 1 *On montre sur l'exemple suivant la différence entre les heuristiques HMCT et AHMCT, c'est–à–dire que minimiser la date de terminaison d'une tâche sur un serveur, stratégie de HMCT, est différent de minimiser le makespan local où la tâche est exécutée.*

On considère S_1 et S_2, deux serveurs, dont la puissance de calcul du premier est deux fois moindre que celle du second. Il y a quatre requêtes (les durées sont données pour S_2) : r_1 de durée 30 secondes arrive à $t = 0$; r_2 de durée 70 secondes arrive à $t = 20$; r_3 de durée 20 secondes arrive à $t = 30$ et enfin r_4 de durée 5 qui arrive à $t = 60$ secondes.

Nous donnons sur la figure 5.3 le diagramme de Gantt que l'on peut observer sur chaque serveur. HMCT et AHMCT affectent r_1 et r_2 à S_2, les tâches devant terminer alors respectivement à $t = 40$ et $t = 100$. On voit que les tâches se partagent la ressource entre $t = 20$ et $t = 40$. Notons qu'affecter r_3 sur S_2 conduirait à la faire terminer à $t = 75$ et le makespan local constaté sur S_2 serait 115 secondes donc r_3 est affectée à S_1 où elle prend le double de temps et le makespan local devient $T_{r_3} = 70$ secondes.

La requête r_4 est représentée avec la durée qu'elle aurait si elle était affectée sur les deux serveurs. HMCT affecterait la tâche r_4 à S_2 où elle devrait terminer à $t = 70$ secondes car la simulation de r_4 sur S_1 prévoit sa date de terminaison à $t = 80$ secondes. En revanche, AHMCT affecte r_4 à S_1 où le makespan local devient 80 : en effet, r_4 y termine (simultanément avec la troisième requête) à $t = 80$, contre un makespan égal à 105 sur S_2 sinon.

Mais ce n'est plus vrai dans le modèle temps–partagé comme le montre l'exemple 1. Or, l'usage des informations du HTM permet de calculer le makespan de l'ensemble des tâches actuellement soumises à l'environnement. Ainsi, on peut implanter une heuristique fondée sur la réelle stratégie de MCT : minimiser le temps de terminaison de l'ensemble des tâches vu comme une unique application en minimisant son temps de terminaison global, à chaque soumission, pour les systèmes temps–partagé.

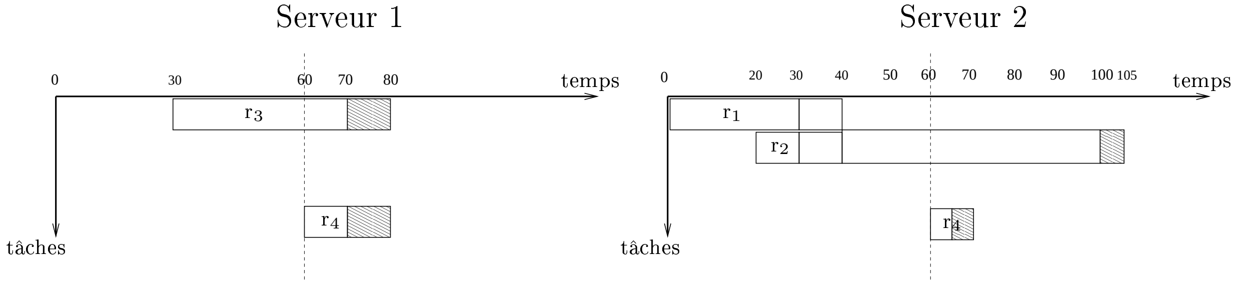


FIG. 5.3 – Exemple d'exécution pour AHMCT

A chaque nouvelle requête, le HTM simule son affectation sur chaque serveur. Pour chaque affectation, AHMCT calcule le makespan obtenu sur l'ensemble des requêtes : l'heuristique calcule pour chaque serveur le temps de terminaison des requêtes qui lui sont affectées (makespan local). AHMCT itére ainsi pour tous les serveurs et choisit finalement celui qui minimise les makespan locaux obtenus précédemment (algorithme 7).

Algorithme 7: Algorithme de AHMCT

Données : Le HTM dispose du diagramme de Gantt de chaque ressource;
Le HTM dispose de la modélisation de la plate–forme;
Résultat : Le serveur où sera affectée la requête
début
  pour chaque *nouvelle tâche t* **faire**
    pour chaque *serveur j capable de résoudre la tâche* **faire**
      Demander au HTM de calculer M_j, le maximum des temps de terminaison des tâches exécutées sur le serveur j si t est affectée au serveur j;
    Choisir le serveur j_0 qui minimise M_j;
    Prévenir le HTM de l'affectation de la requête sur le serveur j_0;
    retourner *[j_0]*;
fin

De même que HMCT, AHMCT a l'inconvénient de surcharger les serveurs les plus rapides, ce qui a deux conséquences : la terminaison des tâches en train d'être exécutées est retardée et les serveurs peuvent être contraints de refuser des tâches devant la trop grande fréquence d'arrivée des requêtes.

5.3.3 Minimum Perturbation : MP

MP choisit le serveur qui minimise la somme des perturbations occasionnées par l'affectation de la nouvelle requête sur ce serveur (algorithme 8). En cas d'égalité, par exemple tout au début de l'expérience lorsqu'aucun serveur n'est encore chargé, le serveur minimisant la date de terminaison de la requête est choisi (on applique donc en cas d'égalité

Algorithme 8: Algorithme de MP

Données : Le HTM dispose du diagramme de Gantt de chaque ressource;
Le HTM dispose de la modélisation de la plate–forme;
Résultat : Le serveur où sera affectée la requête
début
    pour chaque *nouvelle tâche t* **faire**
        pour chaque *serveur j capable de résoudre la tâche* **faire**
            Demander au HTM de calculer $P_j = \sum_i \delta_i$, la somme des perturbations occasionnées après affectation de la tâche t au serveur j;
        si *tous les P_j sont égaux* **alors**
            Choisir le serveur j_0 qui minimise T_j;
        sinon
            Choisir le serveur j_0 qui minimise P_j;
        Prévenir le HTM de l'affectation de la requête sur le serveur j_0;
        retourner $[j_0]$;
fin

la stratégie orientée makespan de HMCT).

L'objectif de MP est de délivrer une meilleure qualité de service à chaque tâche en cherchant à retarder le moins possible les tâches actuellement en cours. L'inconvénient majeur est qu'en conséquence les ressources peuvent être sous–exploitées : en effet, la durée de la nouvelle requête n'étant pas directement prise en compte dans la fonction de coût, une tâche peut être affectée à tort à un serveur lent, par exemple dès qu'il y en a un oisif. En revanche, dès que tous les serveurs sont utilisés, les tâches sont affectées en priorité aux serveurs les plus rapides.

5.3.4 Minimum SumFlow : MSF

MSF utilise les informations du HTM afin de calculer la somme des durées de chaque requête d'un serveur, avant et après simulation de l'affectation de la dernière requête. Ensuite, l'heuristique retourne l'identité du serveur qui minimise l'ensemble des sommes calculées, plus précisément :

$$\min_j \left(\sum_{i=1}^{i=n} (T_{i,k} - a_{i,k}) \right)$$

La figure 5.4 montre un exemple où deux serveurs exécutent chacun deux tâches. Les diagrammes de Gantt sont donnés sur la gauche de la figure. Une requête est soumise et simulée sur les deux serveurs dont on voit le diagramme de Gantt après simulation sur la droite. On s'aperçoit que la différence entre les deux valeurs de sumflow (somme des aires des rectangles blancs et gris représentant les durées des tâches) n'est due qu'aux

Algorithme 9: Algorithme de MSF

Données : Le HTM dispose du diagramme de Gantt de chaque ressource;
Le HTM dispose de la modélisation de la plate–forme;
Résultat : Le serveur où sera affectée la requête
début
  pour chaque *nouvelle tâche t* **faire**
    pour chaque *serveur j capable de résoudre la tâche* **faire**
      Demander au HTM de calculer $P_j = \sum_i \delta_i + T_j - a_t$, la somme des perturbations occasionnées plus la durée de la tâche t sur la plate-forme après affectation de la tâche t au serveur j;
    Choisir le serveur j_0 qui minimise P_j;
    Prévenir le HTM de l'affectation de la requête sur le serveur j_0;
    retourner *[*j_0*]*;
fin

perturbations induites par l'affectation de la dernière requête et à sa durée simulée sur le serveur (en gris). En conséquence, l'heuristique n'a besoin de calculer que la quantité $\sum_{i=1}^{t-1} \delta_i + T_t - a_t$ pour chaque serveur (algorithme 9). On s'aperçoit sous cette forme que MSF est la même heuristique que MSI que Weissmann présente dans [Wei96] sur des expériences de simulation.

Comme les deux calculs le montrent, MSF a pour but de réduire le coût total sur les ressources tout en profitant des avantages de HMCT et de MP (c'est–à–dire concrètement de conserver une stratégie orientée makespan tout en essayant de donner une meilleure qualité de service à chaque tâche).

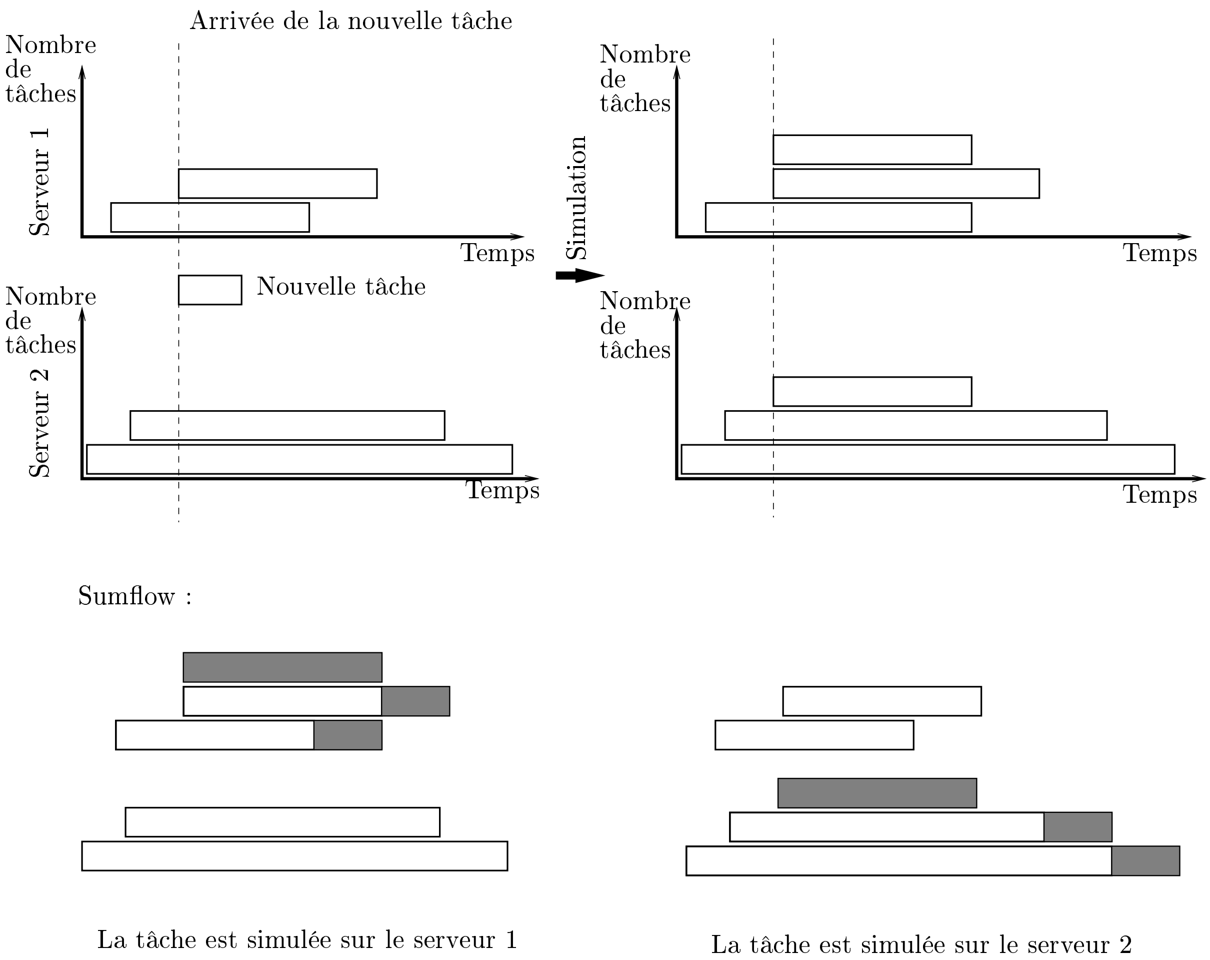


Fig. 5.4 – Simplification des calculs pour MSF

5.3.5 Minium Length : ML

ML a été developpée conjointement à MSF : les objectifs sont les mêmes que ci-avant : ML a pour but de donner un bon makespan global tout en essayant de minimiser le temps de réponse moyen ainsi que le temps de terminaison des requêtes. Toutefois, la stratégie utilisée ici consiste à minimiser la quantité de travail pondérée qu'il reste à produire au serveur avant de devenir oisif. Aussi, après la simulation de la nouvelle requête sur l'ensemble des serveurs, ML choisit le serveur qui minimise la somme des durées simulées restantes des tâches en cours (algorithme 10).

L'objectif de ML est encore de réduire les délais supplémentaires que l'affectation de la nouvelle requête va nécessairement occasionner sur celles présentes sur le serveur, tout en conservant une stratégie orientée makespan. En effet, les délais induisent la quantité restante de travail à produire pour chaque tâche, donc de celle qui donnera le makespan et la durée simulée de la nouvelle requête est prise en compte dans la fonction de coût.

Algorithme 10: Algorithme de ML

Données : Le HTM dispose du diagramme de Gantt de chaque ressource;
Le HTM dispose de la modélisation de la plate–forme;
Résultat : Le serveur où sera affectée la requête
début
  pour chaque *nouvelle tâche t* **faire**
    pour chaque *serveur j capable de résoudre la tâche* **faire**
      Demander au HTM de calculer $P_j = \sum_i (T_i - a_t) = \sum_i D_i$, la somme des durées à effectuer avant terminaison pour chaque tâche i exécutées sur le serveur, après affectation de la tâche t au serveur j;
    Choisir le serveur j_0 qui minimise P_j;
    Prévenir le HTM de l'affectation de la requête sur le serveur j_0;
    retourner *[j_0]*;
fin

5.4 Expériences de simulation

Nous souhaitons étudier ici le comportement et les performances des heuristiques que nous venons de décrire. Dans les expériences de simulation que nous allons présenter, les requêtes sont toutes indépendantes les unes des autres. Elles peuvent être considérées comme les soumissions d'un ou de plusieurs clients, chacun ayant au moins un problème à résoudre.

L'étude préliminaire des heuristiques HMCT, MP et MSF sur des expériences de simulation a été publiée dans les actes de *Renpar*'02. Les résultats suivants ont été en partie publiés dans les actes de l'école *Grid'02* [Can02] et font l'objet d'une publication dans les actes de la conférence *QOS and Dynamic System workshop*, QDS'04 [CJ04a].

5.4.1 Modélisation

La plate–forme considérée comprend un nombre fixé de 25 serveurs. Sa modélisation est très simple : nous supposons qu'il existe un lien direct entre chaque client et chaque serveur. On remarque que le nombre de clients n'est pas important en soi puisque si un serveur reçoit des requêtes de plusieurs clients en même temps, le partage du lien de communication sera géré au niveau du serveur.

Chaque serveur est mono–processeur et dispose d'une puissance de calcul propre. Les caractéristiques du réseau (bande passante, latence) et des serveurs (puissance de calcul en Mflops) sont supposées connues. L'heuristique dispose d'informations sur chaque requête au moment où elle est soumise comme le nombre d'opérations à effectuer pour exécuter complètement la tâche.

Les suppositions de ces paramètres sont communément admises pour l'étude du comportement des heuristiques et sont obtenues par exemple à l'aide de benchmarks présentés dans la section 3.3.

5.4.2 Expériences

Implantation de la modélisation

Chaque processeur dispose d'une capacité de calcul comprise entre 150 et 500 Kflops générée aléatoirement. Les liens et cartes réseaux sont identiques et leur capacité est de 100 Mbits/secondes. Pour décrire les dates d'arrivée des tâches dans l'environnement de résolution de problèmes, nous avons utilisé une loi de Poisson dont le paramètre μ est la moyenne des temps d'inter–arrivée. Les quantités de données à transférer en entrée et en sortie entre un client et un serveur sont générées à l'aide d'une loi Uniforme entre 1 Ko et 300 Mo. Le nombre de calculs est lui aussi généré avec une loi Uniforme, selon les deux règles suivantes :

– La phase de calcul doit durer plus de dix fois la phase de transfert, ce qui justifie de déporter les calculs.
– La phase de calcul ne doit pas être supérieure à 600 secondes sur le plus rapide des 25 serveurs générés.

Implantation logicielle

MCT. Pour estimer la date de terminaison de la requête, MCT multiplie la durée à vide de la tâche sur chaque serveur par leur charge respective. La charge modélisée ici correspond à la définition donnée dans la section 3.2.1 : sa valeur est le nombre de tâches en cours d'exécution sur le serveur où la prédiction a lieu.

Environnement de soumission. Nous avons utilisé l'API Simgrid, brièvement présentée dans la section 3.6.2, pour simuler l'environnement. Le HTM et les heuristiques qui ont été décrits précédemment ont été implantés en C. Le développement pour cette partie a conduit à l'écriture de plus de 5000 lignes de code. L'usage direct de Simgrid pour construire le HTM n'était pas possible : tout d'abord en raison de l'état d'avancement du projet quand les expériences ont été faites et par ailleurs, Simgrid ne dispose pas d'une fonction d'estimation de la durée d'une tâche sur la plate–forme modélisée autre que celle sur la ressource oisive (éventuellement perturbée selon une loi de probabilité). Une fois une tâche simulée, il ne permet pas de retour dans le temps, c'est–à–dire que la fonction temps est monotone croissante. Il n'était pas possible d'instaurer facilement des points de sauvegarde d'état. En revanche, nous avons comparé les résultats des deux simulateurs pour valider le HTM. D'autre part, une surcouche basée sur Simgrid a aussi été developpée pour simuler l'arrivée des requêtes. Enfin, pour toutes les utilisations statistiques et probabilistes, nous avons employé les fonctions de la bibliothèque GSL [GT96].

5.4.3 Résultats

Analyse des données

Nous présentons les résultats des performances de chaque heuristique sur cinq graphes pour cinq métriques observées. Excepté sur le premier, nous donnons le gain obtenu par rapport à l'heuristique de référence MCT. Ainsi, les cinq graphes indiquent le *pourcentage de tâche terminant plus tôt que MCT*, le *gain sur le makespan*, le *gain sur le sumflow*, le *gain sur le maxflow* et le *gain sur le maxstretch*. Les tâches étant toutes indépendantes, on remarque que le gain obtenu sur le sumflow est aussi le gain sur le *temps de réponse moyen*.

Pour chaque graphe, le nombre de tâches varie de 10 à 250. La moyenne de l'inter–arrivée des tâches μ varie de 0 à 70 secondes (par simplicité, nous omettrons l'unité par la suite). Chaque point est la moyenne de la performance observée sur 1000 simulations où tout est regénéré. Cela conduit pour chaque heuristique à 200000 simulations. Remarquons qu'au contraire des autres graphes où un gain est obtenu dès que la valeur est positive, sur le premier graphe la valeur doit en revanche dépasser 50% pour montrer une amélioration : notre heuristique doit placer plus de tâches terminant plus tôt que MCT.

Des courbes ont aussi été calculées à partir d'un nombre moins important de simulations où jusqu'à 970 tâches étaient soumises. Elles montrent le même comportement et ne sont donc pas présentées ici.

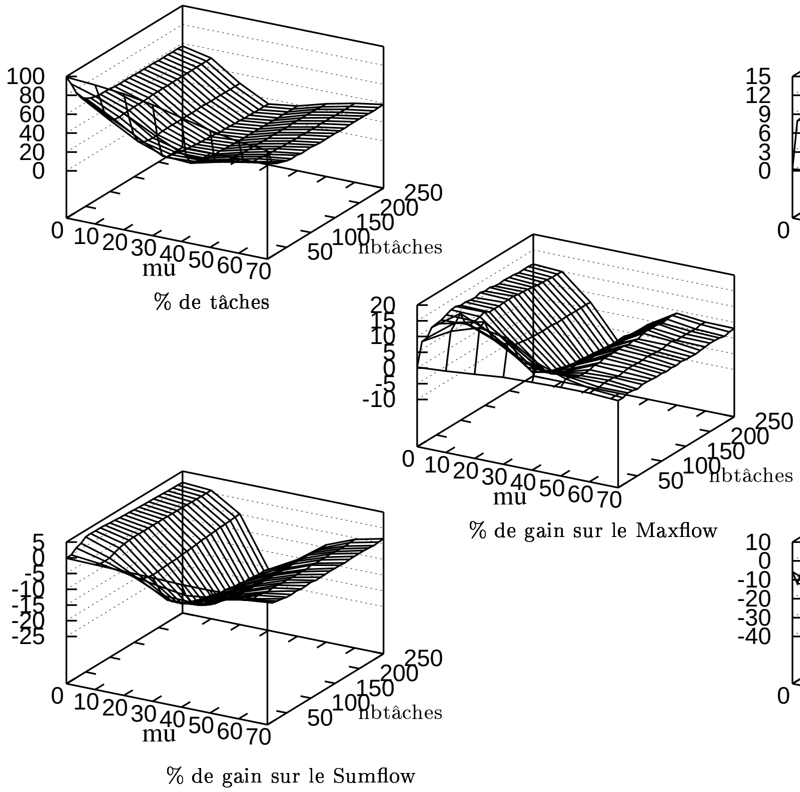


FIG. 5.5 – HMCT

HMCT

On observe sur la figure 5.5 que l'emploi de HMCT amène un gain atteignant 10% sur le makespan pour des valeurs moyennes d'inter–arrivée $\mu < 30$. Lorsque μ est supérieur, les gains sont toujours positifs même si presque nuls. Le sumflow de HMCT est toujours supérieur à celui de MCT pour $\mu > 10$, ce qui conduit à des dégradations de performances (gain négatif). Ceci peut s'expliquer de la façon suivante : MCT suppose dans ces calculs que les tâches en cours d'exécution sur le serveur ne termineront pas pendant l'exécution de la nouvelle requête et maximise en conséquence l'estimation de la durée. En revanche, HMCT dispose d'une estimation plus précise de la durée des tâches et tente de maximiser l'utilisation des serveurs conduisant à une durée d'exécution minimale. Ceci a l'inconvénient d'allonger le flow des tâches présentes sur le serveur et explique aussi les mauvais résultats obtenus sur le pourcentage de tâche terminant plus tôt que pour MCT (moins de 20% pour $\mu = 30$). La courbe sur le maxflow nous indique en revanche que sauf pour $\mu = 30$, la tâche la plus longue est moins longue pour HMCT que MCT. La courbe sur le maxstretch nous renseigne qu'une tâche peut subir un ralentissement allant jusqu'à 30% de plus que MCT pour $\mu \geq 40$ dans le pire des cas. Ceci peut s'expliquer avec le petit exemple suivant : supposons deux serveurs de performances semblables dont l'un exécute une tâche qu'il mettra 2 secondes à terminer. Si une nouvelle tâche est soumise, MCT calculera une estimation presque deux fois supérieure à la réalité en cas d'affectation. Il choisira donc le second serveur. HMCT estime que la tâche termine plus tôt sur le premier serveur et la lui affectera au détriment de la première tâche.

AHMCT

Les résultats des performances obtenues pour AHMCT sont très comparables à ceux de HMCT. Les quelques différences qui apparaissent entre les figures 5.5 et 5.6 montrent que HMCT est meilleure qu'AHMCT. En effet, le pourcentage de tâches terminant plus tôt est toujours inférieur à 50% pour un nombre de tâches supérieur à 20. Les gains sur le makespan sont de l'ordre de 10% pour $\mu \leq 30$ et sont positifs, même si quasiment nuls pour des valeurs supérieures de l'inter–arrivée moyenne des tâches. AHMCT requiert plus de temps de travail sur l'ensemble des ressources et donc affiche des performances négatives en comparaison à MCT sur le sumflow. Les courbes du maxflow de HMCT et AHMCT étant similaires, sauf pour $\mu = 30$, la tâche la plus longue est moins longue pour AHMCT que MCT. En revanche, AHMCT améliore le stretch pour $\mu \leq 20$. Pour les valeurs supérieures, elle rejoint le même comportement que celle de HMCT.

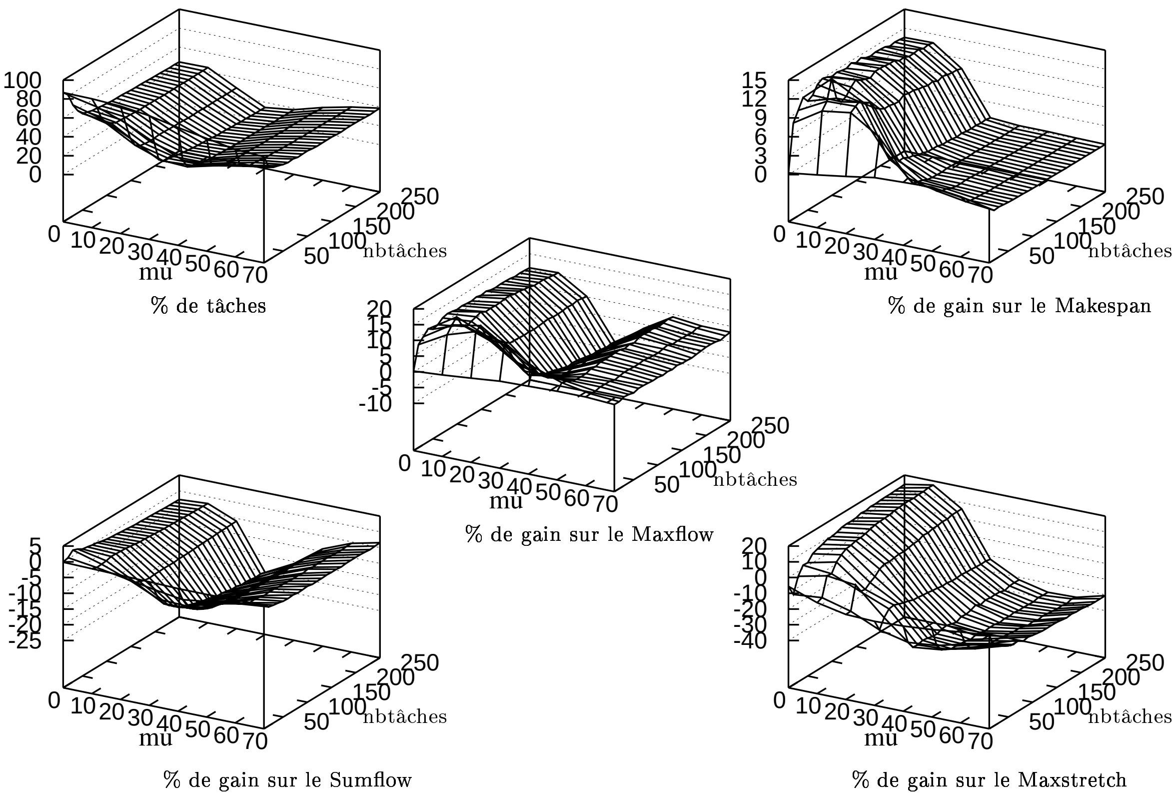


Fig. 5.6 – AHMCT

MP

D'après la figure 5.7, pour un nombre de tâches supérieur à 20, MP donne toujours un meilleur makespan que celui obtenu par MCT. En particulier, pour $\mu < 30$, le gain est de 5%. En parallèle, le nombre de tâches terminant plus tôt est toujours supérieur à 60%, avec un pic à 75% pour $\mu = 30$. Le gain sur le sumflow, donc sur le temps de réponse moyen, atteint son maximum avec une amélioration de 15% pour $\mu = 30$. La tâche la plus longue est en revanche plus longue de 10% pour $\mu = 30$ mais est toujours moins longue sinon. Finalement, MP permet un gain toujours positif sur le masxtretch, avec plus de 90% pour $\mu = 30$. Il est intéressant de constater ici, pour $\mu = 30$, que MP donne à la fois «une tâche la plus longue» plus grande de 10% que celle de MCT, mais qu'une tâche (pas nécessairement la même !) subit un ralentissement maximum qui est inférieur de 90% par rapport au maximum obtenu par MCT. De plus, on observe ici une découpe autour de l'axe $\mu = 30$ en raison de la nature même de MP : une ressource libre est choisie en priorité. Or, à partir de l'inter–arrivée moyenne égale à 30, la ressource libre choisie a de grandes chances d'être performante : elle a terminé les tâches qui lui ont été affectées précédemment .

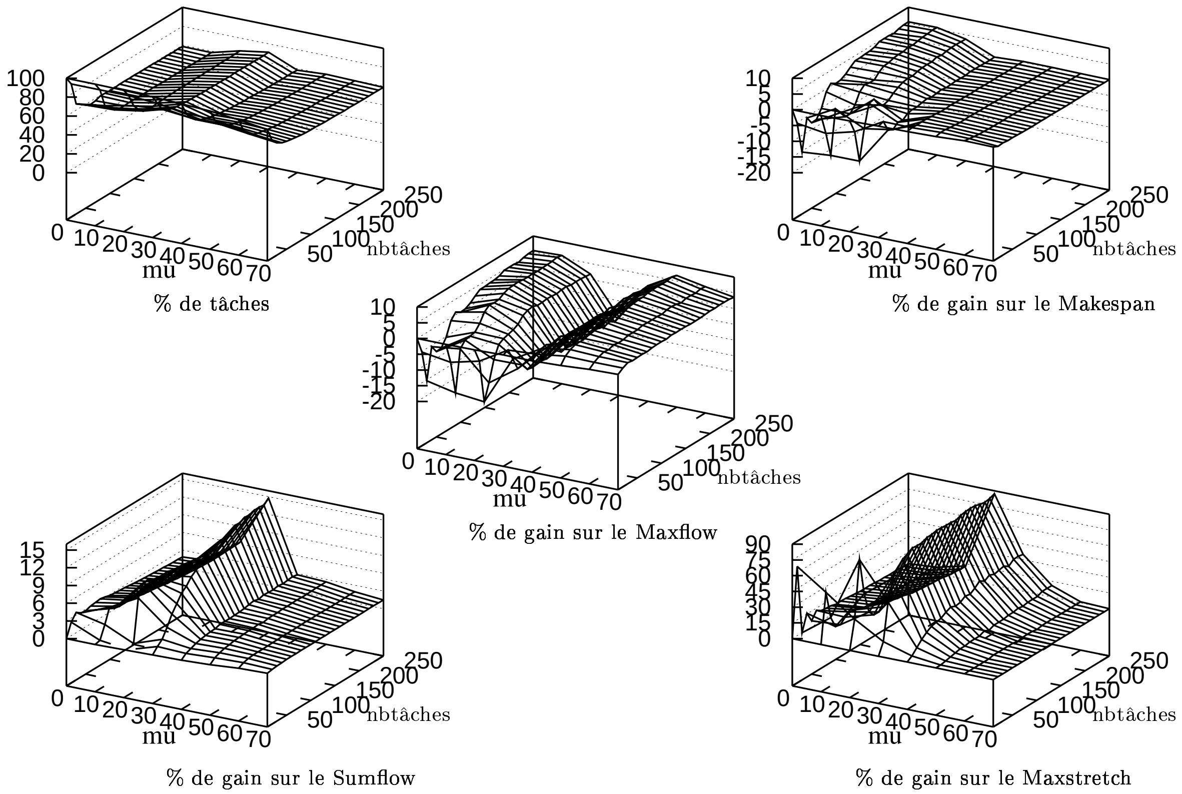


FIG. 5.7 – MP

MSF

Comme le montre la figure 5.8, MSF donne des gains sur le makespan par rapport à MCT qui sont toujours positifs. Ils sont autour de 9% pour $\mu \leq 20$ et décroissent avec μ. MSF et MCT demandent globalement le même temps d'utilisation générale des ressources puisque le gain réalisé sur le sumflow en comparaison avec MCT oscille entre 2% et -3%. De la même manière, le pourcentage de tâches terminant plus tôt oscille autour de 50% : il commence à 60% pour $\mu = 0$ puis décroît jusqu'à 40% pour $\mu = 40$ et augmente ensuite jusqu'à un peu plus de 50% pour $\mu = 70$. En revanche, la courbe sur le maxflow montre que dans le pire des cas de MSF et de MCT, «la tâche la plus longue» de MSF passe moins de temps dans le système que celle de MCT. Elle peut être jusqu'à 25% plus courte, pic atteint pour $\mu = 30$. Il est intéressant de constater que pour tout μ (sauf $\mu = 70$ où le gain est de l'ordre de 7%), le gain est au minimum de 10%. Le gain réalisé sur le maxstretch est constant à 5% pour $\mu \leq 30$ puis se transforme en perte de performances avec un pic à 20% pour $\mu = 60$. Il commence un retour vers 0 ensuite. Notons que le comportement de MSF change de part et d'autre de l'axe $\mu = 30$ où il s'avère que l'affectation des tâches conduit à des durées moyennes un peu plus longues qu'avec MCT. Pourtant, dans l'ensemble, le makespan est positif et le pire cas est meilleur.

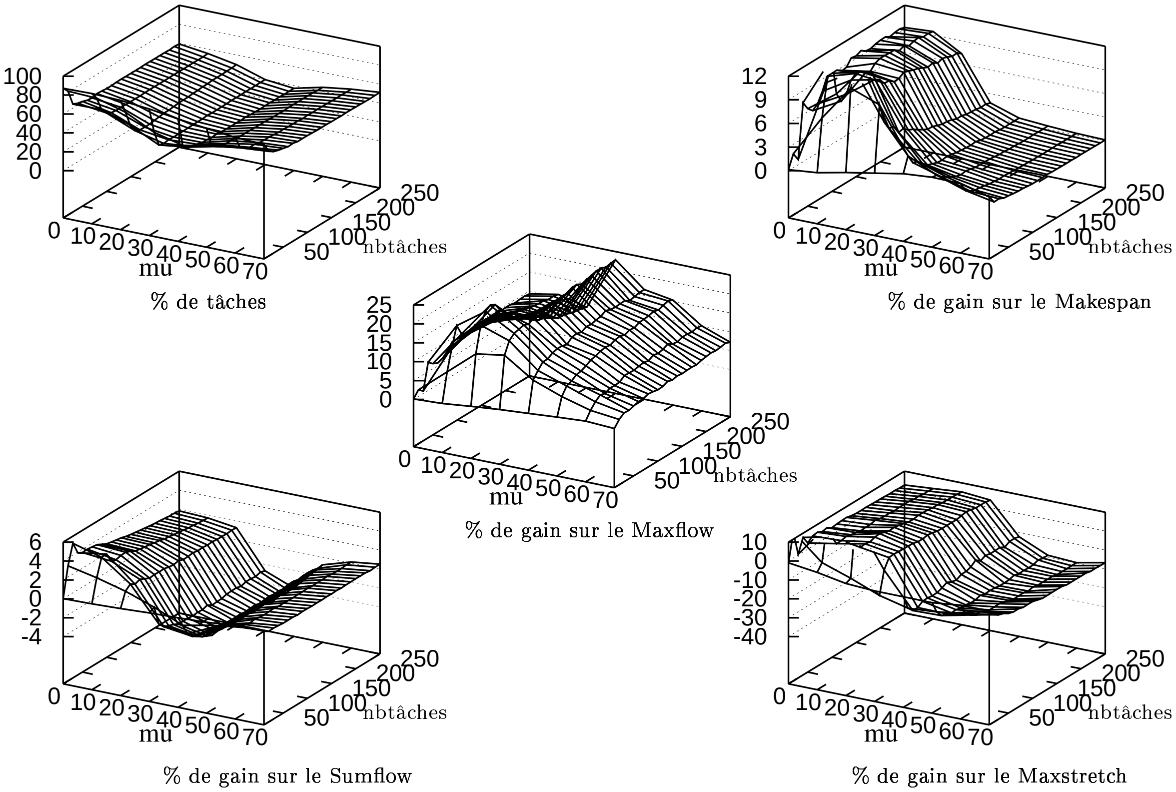


Fig. 5.8 – MSF

ML

On voit sur la figure 5.9 qu'à part pour $\mu = 0$, les gains sur le makespan sont toujours positifs, mais presque nuls dès que $\mu \geq 20$. Notons la perte pour le cas critique $\mu = 0$ avec un nombre de tâches supérieur à 150 de l'ordre de 10%. Sur le sumflow, ML obtient un gain atteignant 15% pour $\mu = 10$ puis des gains faibles pour $\mu \geq 20$. En plus d'améliorer le temps de réponse, il y a plus de tâches terminant plus tôt avec ML qu'avec MCT : au moins 60% des tâches avec un premier pic à 70% pour $\mu = 10$ et un deuxième à 80% pour $\mu = 60$. La courbe sur le maxflow montre que dans le pire cas, une tâche a une durée moindre lorsque l'expérience est ordonnancée par ML que par MCT. Le gain maximum constaté vaut 25% pour $\mu = 10$. Enfin, on peut constater que ML est aussi globalement meilleure que MCT sur le maxstretch en proposant un pic de gain à 35% pour $\mu = 10$ et une légère perte autour de 5% pour $\mu = 20$. On constate qu'à l'instar de MP, le comportement de ML change autour de l'axe $\mu = 20$. Pour des arrivées plus espacées dans le temps, ML se comporte principalement comme MCT tout en gardant la pertinence des informations du HTM : elle utilise davantage les serveurs les plus rapides même s'ils sont employés (la tâche en cours d'exécution pour $\mu = 60$ par exemple est presque terminée) à condition de ne pas trop perturber le système. Ainsi, on observe plus de tâches terminant plus tôt sans observer de gros gains sur le temps de réponse.

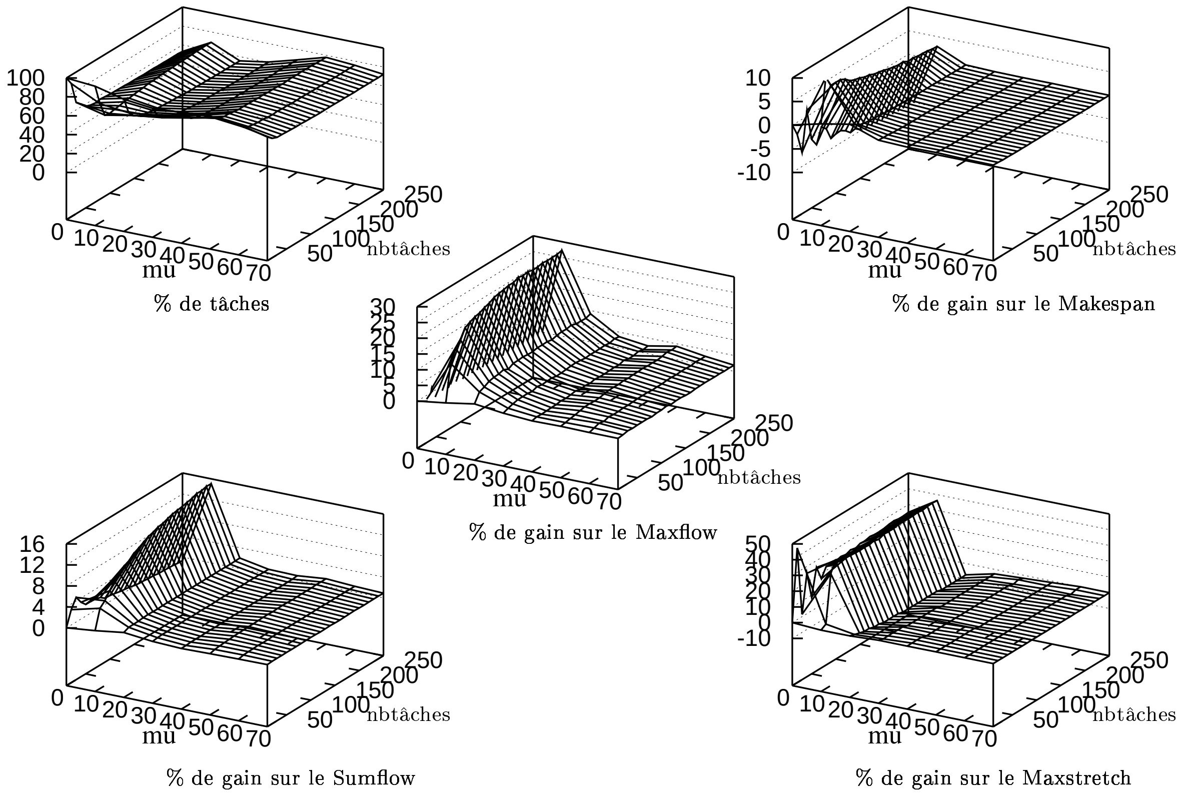


Fig. 5.9 – ML

5.5 Expérimentations en grandeur réelle

Nous souhaitons évaluer nos heuristiques à une échelle réelle et observer leurs performances lors d'expériences comprenant des tâches indépendantes et/ou des applications réelles, ce qui induit des relations de dépendance. Pour cela, nous avons implanté les différentes heuristiques et le HTM décrits précédemment dans l'intergiciel de grille NetSolve. Dans cette section, nous comparons donc les heuristiques HMCT, MP et MSF à MCT, l'heuristique de NetSolve telle qu'elle est implantée dans sa version 1.4.1. Remarquons que, mises bout à bout, la totalité des exécutions des expériences présentées dans cette section requiert plus de cinquante jours complets d'occupation ininterrompue des machines.

Dans ce qui suit, nous décrivons tout d'abord la mise en place des expériences. Puis, nous présentons les ressources composant la plate–forme de test. Nous définissons ensuite les modèles d'expérimentation utilisés. Enfin, nous donnons les résultats que nous avons scindés en deux sous–parties : la première concerne les tâches indépendantes et la seconde les autres expériences.

La section 5.5.3 fait l'objet d'une publication dans les actes de la conférence *Heterogeneous Computing Workshop*, *HCW'*03 [CJ03]. Une partie des résultats de la section 5.5.4 a été publiée dans les actes de la conférence *Euro–par'*04 [CJ04b].

5.5.1 Implantation dans l'intergiciel

NetSolve

NetSolve est l'intergiciel avec lequel nous avons travaillé pour réaliser l'ensemble de nos tests. Nous l'avons brièvement décrit dans la section 2.5.4. Le choix de l'intergiciel prend en compte les facteurs suivants : il se déploie facilement sans nécessiter de droits quelconques sur les serveurs de la plate-forme et est codé en C ; la disponibilité de son code nous a permis de disposer de son architecture construite sur le modèle client-agent-serveur et de ses mécanismes de soumission GridRPC. De plus, la découpe de ses fichiers permet d'implanter relativement intuitivement des modifications dans le module d'ordonnancement. Elle a aussi grandement facilité l'insertion de notre module de prédiction de performances, le HTM.

MCT dans NetSolve

MCT bénéficie de mécanismes de correction de charge : quand la charge d'un serveur varie plus qu'une valeur seuil, le serveur envoie un rapport de charge à l'agent, ceci dans la limite d'un message par minute [ACD02]. De plus, après affectation d'une tâche sur un serveur, un second mécanisme modifie la dernière valeur de charge enregistrée de ce serveur afin de rendre compte d'une prochaine hausse de charge. Ainsi, MCT dispose d'une

Rôle	Machine	Processeur	Vitesse	Mémoire	Swap	Système d'exploitation
server	spinnaker	xeon	2 GHz	1 Go	2 Go	linux-2.4
	artimon	pentium IV	1.7 GHz	512 Mo	1024 Mo	linux-2.4
	pulney	xeon	1.4 GHz	256 Mo	533 Mo	linux-2.4
	cabestan	pentium III	500 MHz	192 Mo	400Mo	linux-2.4
	valette	pentium II	400 MHz	128 Mo	126 Mo	linux-2.4
	chamagne	pentium II	330 MHz	512 Mo	134 Mo	linux-2.4
	soyotte	sparc Ultra-1		64 Mo	188 Mo	SunOS 5.7
	fonck	sparc Ultra-1		64 Mo	188 Mo	SunOS 5.7
agent	xrousse	pentium II bipro	400 MHz	512 Mo	512 Mo	linux-2.4
client	zanzibar	pentium III	550 MHz	256 Mo	500 Mo	linux-2.4

Tab. 5.1 – Ressources des plate-formes d'expérimentation

courte mémoire pour affecter une nouvelle requête soumise avant le rapport de charge du serveur. Enfin, l'agent est informé par un message lorsqu'une tâche termine. Il corrige en conséquence la dernière valeur de la charge enregistrée du serveur afin de rendre compte de l'accroissement de ses capacités de calcul lors des futures décisions en attendant son rapport de charge.

Implantation du HTM et des heuristiques

Le code du HTM implanté dans NetSolve est en grande partie issu des développements faits pour les expériences de simulation. Il a été intégré dans le code de l'agent. Des modifications lui ont aussi été apportées, notamment sur les mécanismes d'enregistrement et de pulsation, afin d'assurer la cohérence des modules et des données entre eux.

La durée des tâches utilisées dans les expériences et présentées plus loin a été mesurée. Nous avons reporté la moyenne des durées observées sur cinq exécutions dans le code de NetSolve. En conséquence, nos heuristiques ont des informations tirées du HTM qui utilise ces valeurs. Dans un souci d'équité, nous avons modifié NetSolve afin que ses prédictions de performances se basent aussi sur ces valeurs donnant ainsi à MCT des informations plus précises.

5.5.2 Modalité des expériences

Ressources et Modélisation de la plate–forme

L'ensemble des expériences que nous présentons ici a été réalisé à l'aide de l'environnement de résolution de problèmes NetSolve sur des plate-formes composées d'un client, d'un agent et de quatre serveurs. Les ressources, dont nous donnons les caractéristiques dans le tableau 5.1, sont distribuées dans le laboratoire du LORIA. Elles sont interconnectées par le réseau du centre de recherche ; aussi, si les serveurs sont dédiés à l'environnement, ce n'est pas le cas du réseau. Notons que les machines composant effectivement les ressources de calcul de la plate–forme sont données dans les sections correspondantes.

La modélisation est similaire à celle utilisée dans les expériences de simulations. La plate–forme considérée comprend un nombre fixé de 4 serveurs. Sa modélisation est très simple : nous supposons qu'il existe un lien direct entre chaque client et chaque serveur. Dans la modélisation, le nombre de clients n'est pas important en soi puisque si un serveur reçoit des requêtes de plusieurs clients en même temps, le partage du lien de communication est géré au niveau du serveur.

Tâches

Elles sont de deux types : dgemm et wastecpu. Un dgemm est une multiplication de matrices de la bibliothèque BLAS. Son exécution requiert plus ou moins de mémoire, en fonction de la taille des entrées générées aléatoirement. À cause de l'hétérogénéité de la plate-forme, nous avons dû faire face à des difficultés en utilisant MCT. En effet, certains serveurs s'écroulaient par manque de mémoire comme nous le verrons par la suite. Pour poursuivre nos comparaisons, nous avons donc créé un problème tenant le processeur en activité et ne nécessitant pas de mémoire, que nous avons appelé *wastecpu*.

Taille de la matrice carrée	Besoins en mémoire (Mo)		Phase	temps en secondes			
	entrée	sortie		pulney	artimon	cabestan	chamagne
1200	21.97	10.98	temps d'envoi des entrées	3	3	4	4
			temps de calcul	14	18	70	149
			temps d'envoi des sorties	1	1	1	1
1500	34.33	17.16	temps d'envoi des entrées	5	5	5	6
			temps de calcul	25	33	136	292
			temps d'envoi des sorties	1	1	2	2
1800	49.43	24.72	temps d'envoi des entrées	7	8	8	8
			temps de calcul	40	53	231	504
			temps d'envoi des sorties	2	2	3	3

TAB. 5.2 – Besoins des problèmes Dgemm

À chaque type de problèmes correspond trois tailles possibles des entrées, ce qui donne trois durées différentes pour un serveur de calcul donné que nous avons répertoriées dans les tableaux 5.2 and 5.3). Les durées des problèmes dgemm que nous utiliserons tout au long de nos expériences sont celles de multiplications de matrices carrées de tailles 1200*1200, 1500*1500 ou 1800*1800 (que nous noterons par abus 1200, 1500 et 1800 respectivement). Les paramètres donnés au problème wastecpu sont 1200, 1500, 1800 dans le but de refléter les durées obtenues avec les dgemm. Dans les sections suivantes, nous donnerons parmi les informations le type de problèmes utilisé et parlerons indifféremment de la tâche la plus courte ou de tâche de taille 1 en référence à la tâche du même type dont l'entrée est 1200. De même, la tâche la plus longue ou de taille 3 fait référence à la tâche dont l'entrée est 1800.

Paramètre	Phase	temps en secondes					
		spinnaker	artimon	cabestan	valette	soyotte	fonck
1200	temps d'envoi des entrées	0.09	0.12	0.1	0.08	0.10	0.16
	temps de calcul	16	17.1	74.86	97.81	128.09	127.56
	temps d'envoi des sorties	0.05	0.03	0.03	0.03	0.06	0.05
1500	temps d'envoi des entrées	0.14	0.13	0.09	0.08	0.18	0.16
	temps de calcul	30.6	33.2	148.48	182.52	255.56	254.09
	temps d'envoi des sorties	0.06	0.03	0.03	0.03	0.06	0.05
1800	temps d'envoi des entrées	0.09	0.14	0.08	0.13	0.14	0.16
	temps de calcul	45.6	49.4	222.26	273.28	382.5	380.66
	temps d'envoi des sorties	0.05	0.03	0.03	0.03	0.06	0.05

TAB. 5.3 – Besoins des problèmes Wastecpu

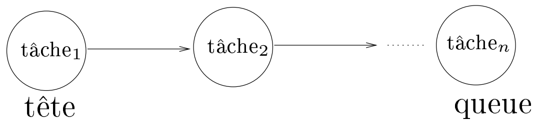


FIG. 5.10 – Une application linéaire 1D-mesh composée de n tâches

Tâches indépendantes

Il existe de nombreux travaux qui étudient les performances d'heuristiques sur cette classe d'application [BSB+99]. Pour certains d'entre eux [CLZB00], les tâches sont toutes connues avant la première soumission alors que pour d'autres [MAS+99], les tâches arrivent dynamiquement dans le système. Dans notre travail, la soumission de tâches indépendantes peut être le fruit d'un client, ou de plusieurs clients soumettant au moins une tâche : nous observons nos heuristiques multi–critères sur le makespan, sur le temps de réponse moyen, etc.

Applications

Les premiers résultats obtenus sur les expériences à base de tâches indépendantes étant plutôt encourageants et pour comparer les performances sur le makespan, nous avons entrepris des expériences contenant une ou des soumissions d'applications structurées sur un graphe de tâches. Cependant, il commence seulement à émerger des benchmarks d'applications type et peu sont aisément implantables : nous avons donc choisi d'observer les performances des heuristiques sur des applications linéaires, ou 1D-mesh (figure 5.10), et des applications stencil (figure 5.11, section 5.5.4). Selon la modélisation présentée à la section 3.3.1, un sommet indique une tâche de calcul qui doit être exécutée sur un serveur et les arcs sont les liens de précédence : ils indiquent un transfert de données qui donne lieu à une tâche de communication sur le réseau. Une tâche de calcul ne peut commencer avant que le serveur où elle est affectée n'ait reçu la totalité des entrées.

Applications linéaires. Par la suite, lorsque nous parlerons des applications 1D-mesh, nous dénommerons la première tâche la *tête* de l'application, et la dernière tâche sera appelée la *queue*. Sa taille est le nombre de tâches la composant. Comme on le voit sur la figure 5.10, chaque tâche a une unique fille sauf la queue. Aussi, dès qu'une tâche est finie (c'est–à–dire quand le client a terminé de recevoir les données du serveur où était exécutée la requête), le client soumet immédiatement la fille jusqu'à la queue de l'application. Ce phénomène est communément appelé *job permanent* puisqu'une tâche de l'application est toujours en cours d'exécution ou de soumission.

Stencil. Les graphes des applications stencil ne sont composés que de tâches de la même taille, comme dans [BBR01]. Il y a deux autres informations majeures pour décrire ce type de graphes : la largeur l et la hauteur h. On voit sur la figure 5.11, où $l = 5$ et $h = 4$, qu'il y a l tâches composant la tête de l'application et l tâches composant la queue de l'application. Excepté pour ces tâches, une tâche a deux ou trois filles. On numérotera (i, j) la tâche de la ligne i et de la colonne j, la première tâche étant $(0, 0)$.

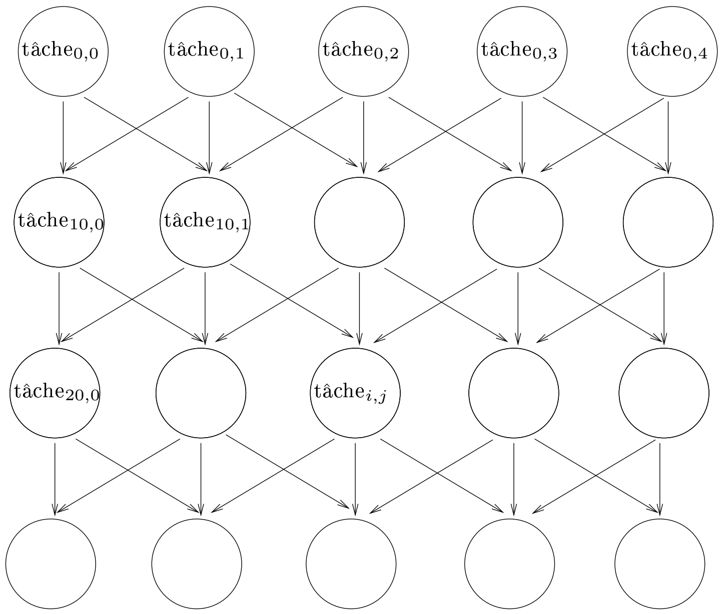


FIG. 5.11 – Graphe stencil 5x4

Scénario

Un scénario regroupe plusieurs expériences ayant les mêmes caractéristiques : les expériences le composant sont générées avec des graines différentes. Un scénario n'implique qu'un seul type de tâches à la fois parmi les problèmes dgemm ou wastecpu. Lorsqu'un scénario est généré, on utilise une loi uniforme pour obtenir la taille de l'entrée de la tâche (parmi 1200, 1500 et 1800).

Scénario	Application(s)		Tâches indépendantes		Expérience	
	nbclients	largeur x hauteur	nbtâches	μ (sec)	nbgraines x nbrun	ntotal
(a) 500 tâches indép. dgemm	-	-	500	20 et 15	4 x 3	500
(b) 500 tâches indép., 2 plate-formes	-	-	500	20, 17 et 15	3 x 3 , 3 x 5	500
(c) 1D-mesh	10	1x50	-	-	4 x 6	500
(d) 1D-mesh	10	1x variable	-	-	4 x 6	500
(e) 1D-mesh + 250 tâches indépendantes	5	1x50	250	20	4 x 6	500
(f) stencil (tâches de taille 1)	1	10x50	-	-	1 x 6	500
(g) stencil (tâches de taille 3)	1	10x50	-	-	1 x 6	500
(h) stencil + 174 tâches indépendantes	1	10x25	174	28	2 x 3	424
(i) stencil + 86 tâches indépendantes	1	10x25	86	40	4 x 6	336
(j) stencil + 86 tâches indépendantes	1	5x25	86	25	4 x 6	211

TAB. 5.4 – Scenarii, modalités et nombre d'expériences

On trouve dans le tableau 5.4 un résumé des différents scénarii entrepris à partir desquels les résultats présentés ici ont été obtenus. Ils sont au nombre de dix, numérotés de (a) à (j). Un scénario comprend la soumission d'une ou de plusieurs applications concurrentes et/ou des tâches indépendantes. Le nombre de clients donne le nombre d'applications soumises, dont on a les caractéristiques dans la colonne «*largeur* × *hauteur*». On trouve ensuite le nombre de tâches indépendantes soumises s'il y en a, ainsi que leur fréquence d'arrivée (qui est injectée dans une loi de Poisson dont on tire les inter-arrivées : on en déduit ensuite les dates d'arrivée). Chaque scénario est composé de *nbgraines* expériences différentes, chacune générée avec une graine différente et qui a été exécutée *nbrun* fois : deux exécutions ne donnant pas nécessairement les mêmes résultats, les expériences ont généralement été reconduites jusqu'à ce qu'il n'y ait plus de variations en moyenne dans les résultats. Le nombre total de tâches soumises lors du scénario est finalement donné dans la colonne *ntotal*. Notons que pour les scénarii (e), (h), (i) et (j), le nombre de tâches indépendantes a été «étalonné» sur l'ensemble des heuristiques afin que la soumission des applications soit toujours concurrente à la soumission de tâches indépendantes.

Contrairement à [MAS+99], la durée d'un type de tâches d'une taille donnée est inverse à la rapidité des serveurs (il n'y a pas de dépendance entre la durée et l'architecture du serveur, elle ne dépend que de sa vitesse). Nous appelons donc *coefficient d'hétérogénéité de calcul de la plate-forme* le maximum sur les serveurs et sur les tâches de la division de la durée maximale par la durée minimale pour le même type de problèmes, c'est–à–dire :

$$\max_{\text{tâche}_i}\left(\frac{\max_{serveur_s} d_{i,s}}{\min_{serveur_s} d_{i,s}}\right)$$

Remarques

En résumé, les modalités suivantes sont valables pour chaque scénario, quelle que soit la plate-forme utilisée :

- La plate-forme est constituée d'une machine client, d'une machine agent et de quatre serveurs, dont les noms sont respectivement donnés dans les sections suivantes.
- Un seul type de problème (dgemm ou wastecpu) est utilisé au cours d'un scénario. Une expérience est soumise *nbruns* fois, chaque exécution étant appelée un *run* et les informations attenantes sont données dans le tableau 5.4. Les résultats présentés plus loin sont issus du calcul de la moyenne effectué sur les runs conduits.
- Pour générer une tâche, trois tailles d'entrée sont possibles. Elle est tirée aléatoirement selon une loi uniforme. On peut par exemple générer 3^n applications 1D-mesh différentes composées de n tâches.
- La prise de décision a un coût négligeable (moins de 0.1 seconde) devant la durée d'une tâche ou des délais d'évolution de la plate-forme, et ce, quelle que soit l'expérience considérée, quelle que soit l'heuristique d'ordonnancement utilisée.
- MCT et nos heuristiques disposent des mêmes connaissances statiques dont l'évaluation de la durée de la tâche sur chacune des ressources non chargées (issue de la moyenne de cinq exécutions effectuées préalablement aux tests et données dans

les tableaux 5.2 and 5.3). Contrairement à MCT qui dispose des senseurs propres à NetSolve, nos heuristiques ne se basent que sur les informations du HTM, qui simule la réalité sans mécanisme de synchronisation.

5.5.3 Tâches indépendantes : scénarii (a) et (b)

Dgemm

Les expériences ont été conduites avec les ressources suivantes : zanzibar (client) ; xrousse (agent) ; pulney, artimon, cabestan et chamagne (serveurs). Les tâches sont des multiplications de matrices carrées de tailles 1200, 1500 ou 1800, générées aléatoirement et tirées selon une loi uniforme (comme il y a 500 tâches soumises par expérience, il y a donc 3^{500} expériences possibles). Sur cette plate-forme, le coefficient d'hétérogénéité est égal à $504/40 = 12.6$. L'inter-arrivée de deux tâches successives est tirée d'une loi de Poisson de paramètre $\mu = 20$ et $\mu = 15$ secondes. Les moyennes des performances sont reportées dans les tableaux 5.5 et 5.6 respectivement.

On trouve dans chaque tableau les performances obtenues par chaque heuristique sur les différentes métriques que nous avons présentées à la section 4.3. Notons toutefois un point particulier qui apparaît dans le tableau 5.6 : pour $\mu = 15$ secondes, MCT et HMCT font s'écrouler un serveur pendant les expériences, vraisemblablement à cause de la demande trop importante en mémoire. Le nombre de tâches ayant été exécutées qui apparaît dans le tableau est donc le *maximum constaté sur l'ensemble des runs effectués.* On constate aussi que MCT fait mieux que HMCT, grâce aux mécanismes de tolérance aux pannes de NetSolve dont ne profite pas HMCT : MCT envoie une liste classée de serveurs, donc après quelques tentatives infructueuses de soumission sur le serveur plébiscité par MCT, le client soumet au deuxième. Les valeurs sont donc marquées pour éviter une mauvaise interprétation des résultats.

Pour $\mu = 20$ secondes, les résultats sont mitigés pour MP, du fait qu'il n'y a pas suffisamment de requêtes pour provoquer beaucoup d'interférences selon sa politique de placement. Le maxstretch est en conséquence plus bas que celui des autres heuristiques. Au contraire des autres, il utilise le serveur chamagne et maximise ainsi le maxflow. Pour HMCT et MSF, il n'y a pas d'amélioration des performances en ce qui concerne le makespan, comme on pouvait s'y attendre : le makespan est ici principalement fondé sur le temps de soumission de la dernière requête. En revanche, on obtient des gains intéressants par rapport à MCT sur toutes les autres métriques : 1 heure 50 minutes de moins sur 7 heures sur la consommation des ressources et un temps de réponse plus rapide, plus de 50% de gains sur le maxflow et maxstretch (dans le pire cas, une tâche dure deux fois plus longtemps si l'agent instancie MCT). Enfin, un client a 66% de chance que sa tâche termine en avance en utilisant HMCT, MP ou MSF.

Pour $\mu = 15$ secondes, seules MP et MSF parviennent au bout des expériences avec toutes les tâches soumises et exécutées. On remarque d'ailleurs des gains importants par

	MCT	HMCT	MP	MSF
nombre de tâches exécutées	500	500	500	500
makespan	9906	9908	10162	9905
sumflow	25922	19934	26383	19702
maxflow	230	103	517	97
maxstretch	12.8	5.8	3.7	5.3
pourcentage de tâches terminant plus tôt que MCT	-	65%	66%	65%

TAB. 5.5 – Scénario (a), $\mu = 20$ sec : résultats en secondes avec des tâches dgemm

	NetSolve's MCT	HMCT	MP	MSF
nombre de tâches exécutées	<495>	<358>	500	500
makespan	<7880>	<5600>	7648	7626
sumflow	<89254>	<25092>	34677	31375
maxflow	<1780>	<500>	720	250
maxstretch	<99>	<27.8>	6.3	11.3
pourcentage de tâches terminant plus tôt que MCT	-	<85%>	84%	87%

TAB. 5.6 – Scénario (a), $\mu = 15$ sec : résultats en secondes avec des tâches dgemm

rapport à MCT où certaines tâches n'ont pas été exécutées et donc n'ont pas interféré avec celles déjà affectées. Parmi eux, on notera particulièrement que 85% des tâches terminent plus tôt en moyenne avec MP ou MSF. MP obtient le maximum pour le maxflow car le serveur chamagne s'est vu affecter deux tâches à un instant donné. En revanche, MSF ne l'a pas utilisé pendant le scénario mais au prix de perturbations supplémentaires sur les serveurs plus rapides conduisant à un maxstretch plus important.

Wastecpu

Les expériences ont été conduites sur deux plate-formes différentes, composées respectivement des serveurs : spinnaker, artimon, cabestan et valette d'où un coefficient d'hétérogénéité de 6 et spinnaker, artimon, soyotte et fonck, d'où un coefficient d'hétérogénéité de 8.9. Les résultats étant comparables, nous présentons ici ceux obtenus sur la deuxième. Remarquons que certaines des tâches ordonnancées par HMCT n'ont pas été exécutées car les ressources de calcul, surchargées, ont refusé le service. Les résultats des tableaux 5.7, 5.8 et 5.9 sont marqués en conséquence.

Si la génération des applications est exactement la même qu'auparavant, notons les différences suivantes dans les scénarii : le type de problèmes utilisé est ici le type wastecpu afin de ne pas être confronté aux problèmes de mémoire vus précédemment ; des expériences supplémentaires ont été conduites avec une fréquence moyenne de soumission des requêtes égale à 17 secondes.

Consommation des ressources. Nous avons reporté dans le tableau 5.8 les pourcentages moyens de tâches (réalisés sur l'ensemble du scénario) affectées par serveurs pour les trois fréquences de soumission. Nous avons précisé entre parenthèses la même information rangée selon la taille de la tâche. Cette information permet de voir si la durée de la tâche donne une priorité par rapport au serveur choisi. On trouve aussi le sumflow moyen des tâches affectées par serveur et le coût total moyen en temps de l'ensemble du scénario.

En parcourant le tableau dans le sens de la fréquence d'arrivée, nous observons quelques changements de comportement : MCT utilise un peu les serveurs lents pour $\mu = 15$ secondes ; HMCT, MP et MSF semblent moins charger spinnaker, au profit d'artimon voire pour MP de fonck et soyotte. En revanche, le pourcentage de tâches selon la taille montre que plus la taille de la requête est grande, moins MSF choisit un serveur lent pour l'exécuter (la durée de la requête entre dans le calcul du choix). En revanche, MP ne prend pas en compte la taille de la tâche et on trouve en conséquence des pourcentages semblables (qu'il faut mettre en relation avec la loi uniforme lors de la génération).

Si on compare les heuristiques pour une fréquence donnée, on s'aperçoit que toutes utilisent prioritairement les serveurs les plus rapides.

- HMCT, qui a une meilleure estimation de la date de terminaison de la requête, utilise plus encore spinnaker que MCT (mais conserve les mêmes rapports d'affectation en fonction de la taille de la tâche). Elle obtient pourtant un sumflow moindre, les affectations sur spinnaker sont donc plus nombreuses mais plus espacées, ce qui réduit les délais.
- MP utilise plus que les autres les serveurs les moins rapides, ce qui lui permet d'obtenir les sumflows les plus bas sur les serveurs les plus rapides. Le coût total en temps de calcul ou sumflow, et la position de MP parmi les autres, dépend de la fréquence. En revanche, on peut constater que MP réduirait le budget imparti pour l'exécution du scénario, quelle que soit la fréquence, si chaque sumflow était pondéré d'un prix proportionnel à la vitesse du serveur.
- MSF semble se comporter comme MCT et HMCT pour des inter–arrivées moyennes de 20 et 17 secondes. Mais lorsque la fréquence est de 15 secondes, il utilise sciemment les serveurs les plus lents pour les tâches les plus rapides.

Performances. Le tableau 5.9 présente finalement, pour chaque fréquence, les résultats obtenus par chacune des heuristiques sur les autres métriques que nous avons introduites à la section 4.3.

- On voit que quelle que soit la fréquence, les quatre heuristiques donnent à peu près la même performance pour le makespan. Cependant, MP est moins performant que les autres car il affecte l'une des dernières tâches sur un serveur lent.
- HMCT et MSF ont les mêmes performances sur le maxflow et le maxtretch. MP est en deçà des performances d'HMCT et MSF sur ces métriques à cause de l'usage des serveurs fonck et soyotte où à un moment de l'expérience, deux tâches sont en cours d'exécution.
- Un client a plus de chance de voir sa requête rapidement complétée si l'ordonnanceur utilise l'une de nos heuristiques, avec jusqu'à 84% de tâches terminant plus tôt avec MP contre 14% pour MCT. L'utilisation des ressources les plus lentes permet de moins retarder celles en cours d'exécution sur les plus rapides.

Remarque. Nous présentons dans le tableau 5.7 le temps de réponse moyen (qui est aussi dans notre contexte la durée moyenne) observé d'une tâche ordonnancée par MCT et les gains relatifs constatés pour chacune de nos heuristiques, pour chaque fréquence. Pour ce scénario, le temps de réponse est le sumflow divisé par le nombre de tâches.

Sans aucune surprise, la durée augmente avec la fréquence d'arrivée (quand μ décroît) car cela génère plus de retards. Pour $\mu = 20$ secondes, HMCT et MSF ont des performances similaires avec un gain de 20% de durée en moins que MCT alors que MP accuse une perte de 30%, à cause de l'usage des serveurs les moins rapides. Pour $\mu = 17$ secondes, les gains sont meilleurs, HMCT et MSF restant concurrents avec 26% et MP donne 10% de réduction de durée. Pour $\mu = 15$ secondes, on voit que des requêtes placées par HMCT sont refusées par les serveurs les plus rapides (toutefois, cela n'a pas d'incidence majeure ici puisqu'aucune tâche ne dépend de l'exécution d'une autre). MP donne de meilleurs résultats que MSF, grâce à une plus grande utilisation des serveurs lents.

	Temps de réponse moyens	gain en **pourcentage**		
	MCT (sec)	HMCT	MP	MSF
$\mu = 20$	44.2	19.7	-28.4	19.9
$\mu = 17$	70.4	25.4	9.9	25.9
$\mu = 15$	135.8	$<$24.3$>$	42.2	35.8

TAB. 5.7 – Scénario (b) : pourcentage de gain sur le temps de réponse moyen pour chaque tâche

Moyenne Scénario (b), $\mu = 20$ secondes								
serveur	MCT		HMCT		MP		MSF	
	% de tâches	sumflow	% de tâches	sumflow	% de tâches	sumflow	% de tâches	sumflow
spinnaker	57.3 (19.4 18.6 19.3)	12428	60.1 (19.8 19.7 20.6)	10149	53.1 (17.7 17.2 18.2)	7986	58.9 (19.8 18.9 20.2)	9859
artimon	42.7 (13.6 14.3 14.7)	9677	39.9 (13.3 13.2 13.4)	7369	36.1 (11.8 12.5 11.9)	6145	41.1 (13.3 14.0 13.8)	7623
soyotte	0.0 (0.0 0.0 0.0)	0	0.0 (0.0 0.0 0.0)	0	4.7 (1.1 1.7 1.9)	6580	0.0 (0.0 0.0 0.0)	0
fonck	0.0 (0.0 0.0 0.0)	0	0.0 (0.0 0.0 0.0)	0	6.1 (2.5 1.6 2.0)	7430	0.0 (0.0 0.0 0.0)	0
total sumflow		22105		17518		28142		17483
Moyenne Scénario (b), $\mu = 17$ secondes								
serveur	MCT		HMCT		MP		MSF	
	% de tâches	sumflow	% de tâches	sumflow	% de tâches	sumflow	% de tâches	sumflow
spinnaker	56.0 (18.9 18.3 18.8)	19370	55.1 (17.1 19.1 18.9)	14040	50.6 (17.4 16.8 16.3)	8477	54.5 (17.0 18.6 18.9)	13308
artimon	44.0 (14.2 14.6 15.2)	16009	44.9 (16.0 13.8 15.1)	11992	37.7 (11.6 12.0 14.1)	7503	45.1 (15.6 14.3 15.1)	12063
soyotte	0.0 (0.0 0.0 0.0)	0	0.0 (0.0 0.0 0.0)	0	6.0 (2.3 2.0 1.7)	7336	0.1 (0.1 0.0 0.0)	43
fonck	0.0 (0.0 0.0 0.0)	0	0.0 (0.0 0.0 0.0)	0	5.8 (1.8 2.0 2.0)	7639	0.4 (0.4 0.0 0.0)	258
total sumflow		35379		26031		30955		25672
Moyenne Scénario (b), $\mu = 15$ secondes								
serveur	MCT		HMCT		MP		MSF	
	% de tâches	sumflow	% de tâches	sumflow	% de tâches	sumflow	% de tâches	sumflow
spinnaker	53.8 (18.2 17.3 18.3)	35302	<52.0 (16.4 17.3 18.3)>	<22497>	48.6 (16.4 16.2 16.0)	11710	51.1 (15.2 17.8 18.1)	18792
artimon	45.1 (14.6 15.2 15.3)	31304	<45.5 (15.4 14.5 15.6)>	<23447>	40.5 (13.7 13.0 13.8)	10906	44.5 (14.4 14.2 15.9)	19007
soyotte	0.5 (0.1 0.2 0.2)	1054	<1.1 (1.1 0.0 0.0)>	<716>	5.6 (1.6 2.2 1.8)	7907	2.0 (1.6 0.4 0.0)	1592
fonck	0.6 (0.2 0.2 0.2)	1213	<1.4 (1.4 0.0 0.0)>	<946>	5.3 (1.4 1.5 2.4)	7917	2.4 (1.9 0.5 0.0)	1968
total sumflow		68873		<47605>		38439		41359

Tab. 5.8 – Scénario (b) : utilisation des serveurs

	$\mu = 20$				$\mu = 17$				$\mu = 15$			
	MCT	HMCT	MP	MSF	MCT	HMCT	MP	MSF	MCT	HMCT	MP	MSF
makespan	**10034**	**10019**	**10281**	**10019**	**8532**	**8512**	**8826**	**8507**	**7650**	<**7598**>	**7885**	**7614**
maxflow	**147.3**	**94.2**	**393.4**	**96.0**	**261.7**	**187.2**	**411.3**	**181.6**	**542.2**	<**300.8**>	**464.4**	**292.2**
maxstretch	**3.8**	**2.7**	**9.2**	**2.7**	**7.0**	**5.2**	**10.4**	**8.8**	**14.2**	<**8.9**>	**13.0**	**10.8**
pourcentage de tâches qui terminent plus tôt qu'avec MCT	–	**55**(**18**)%	**53**(**19**)%	**55**(**17**)%	–	**69**(**19**)%	**72**(**18**)%	**69**(**19**)%	–	<**80**(**17**)%>	**84**(**14**)%	**83**(**15**)%

Tab. 5.9 – Scénario (b) : les résultats sont en secondes

5.5.4 Soumissions mixtes

Tout d'abord, nous commentons la soumission d'une application stencil. En particulier, nous exhibons un comportement de MP expliquant ce que l'on peut attendre de ses performances. Ensuite, nous présentons les gains sur le makespan, sur le temps de réponse et le pourcentage de tâches terminant plus tôt obtenus pour HMCT, MP et MSF en comparaison avec MCT.

Remarques sur la soumission d'une application stencil

Implantation. L'une des premières remarques est d'ordre pratique : on ne peut pas pleinement contrôler l'ordre des requêtes soumises à l'agent. Le programme client, chargé d'appeler les fonctions de NetSolve, doit respecter les contraintes de temps et les relations de précédence de l'application, tout en lançant simultanément les tâches indépendantes selon leur date d'arrivée. Nous l'avons développé en tenant compte que NetSolve n'est pas *thread–safe*, ce qui implique certaines contraintes d'implantation.

Au commencement d'un stencil, il y a $largeur$ processus de lancés «simultanément». Par la suite, il peut y avoir des tâches prêtes à être soumises sur des hauteurs différentes dans le graphe. L'ordre de création des processus pour lancer les fonctions est dépendant de la façon dont est écrit le code du client. Dans notre implantation, les tâches sont soumises selon l'ordre lexicographique sur leurs coordonnées (i,j). Cependant, lorsque les processus contactent l'agent, un second ordre peut apparaître à cause de l'ordonnancement système de la machine client (ici zanzibar) et des communications réseau. Deux runs consécutifs d'une même expérience peuvent donc changer car comme nous le présentons juste après, l'ordre de soumission est important.

Comportement de MP par rapport au stencil. En raison de son graphe de tâches assez complexe, le stencil a, au fur et à mesure de sa soumission, des tâches que l'on peut qualifier de critiques (dans le cas d'une application 1D-mesh, chaque tâche est critique puisqu'un retard pour l'une implique un retard de la terminaison de l'ensemble). On considère à la figure 5.12 un stencil de taille 10×8 tel qu'il serait ordonnancé par MP sur une plate-forme composée de deux serveurs identiques rapides et deux serveurs identiques lents dont le coefficient d'hétérogénéité vaut au moins 13. Les tâches affectées aux serveurs lents sont marquées d'un point noir.

Cet exemple exhibe l'inconvénient majeur de MP : s'il y a des serveurs oisifs, ils sont choisis en priorité. Cependant, même si le serveur lent n'est pas oisif et si le serveur rapide finit l'ensemble des tâches qui lui sont affectées avant le serveur lent, MP peut choisir un serveur lent : il suffit que la somme des perturbations soit trop importante sur les serveurs les plus rapides. On voit ainsi apparaître sur la figure 5.12 des «barrières d'attente» délimitant les moments où les serveurs les plus rapides sont en attente de travail alors que la progression de la soumission du graphe est en attente de la terminaison des tâches

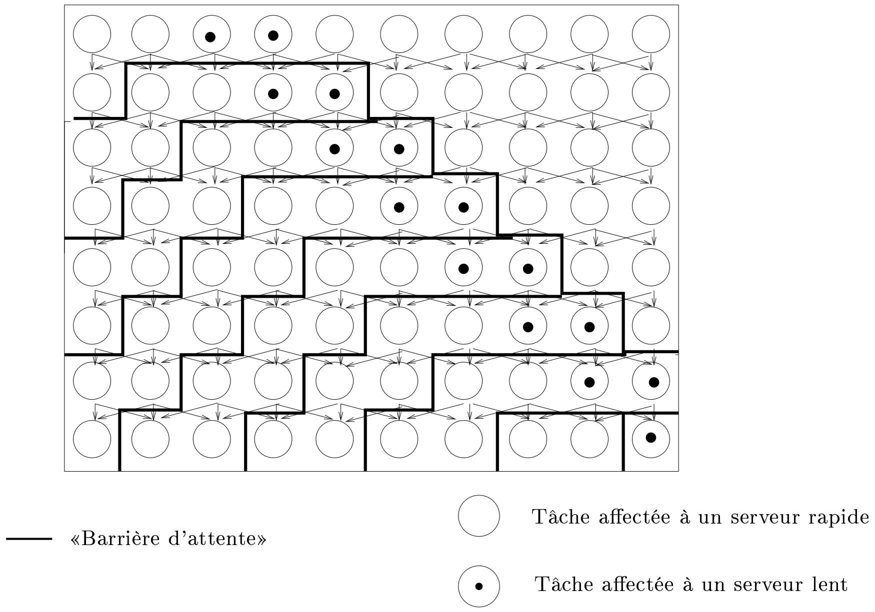


FIG. 5.12 – Exemple d'une exécution d'un stencil de taille 10x8

affectées aux serveurs lents. Dans cet exemple, du fait du coefficient d'hétérogénéité, il est facile de montrer qu'il y a autant de «barrières» que la hauteur du graphe.

Résultats

Nous comparons les performances de nos heuristiques à celles de MCT sur trois métriques. Les résultats sont donc donnés sur trois figures. Le diagramme en barres qu'elles contiennent se lit sensiblement de la même manière : on donne sur l'axe des abscisses le scénario considéré. Il y a pour chaque indice trois rectangles donnant chacun par sa hauteur la performance de l'heuristique considérée par rapport à MCT. La performance observée dépend de la figure et est rappelée sur l'axe des ordonnées. La figure 5.13 indique le gain moyen réalisé *par chaque client* sur le makespan de son application selon le scénario considéré et par heuristique ; les gains moyens réalisés sur le temps de réponse pour chaque tâche indépendante sont donnés par scénario et par heuristique dans la figure 5.14 ; la figure 5.15 donne, pour chaque scénario et chaque heuristique, le pourcentage de tâches indépendantes qui terminent plus tôt pour l'heuristique considérée par rapport à MCT. Ces deux dernières informations permettent de quantifier une certaine qualité de service donnée à chaque tâche. Notons que seuls les scénarii (e), (h), (i) et (j) apparaissent sur ces figures puisqu'elles seules comportent la soumission de tâches indépendantes : nous avons cependant laissé les indices des autres scénarii pour faciliter la lecture entre les graphes.

Makespan. Grâce à la figure 5.13, nous constatons que HMCT et MSF donnent de meilleures performances sur le makespan que MCT quel que soit le scénario, MSF donnant

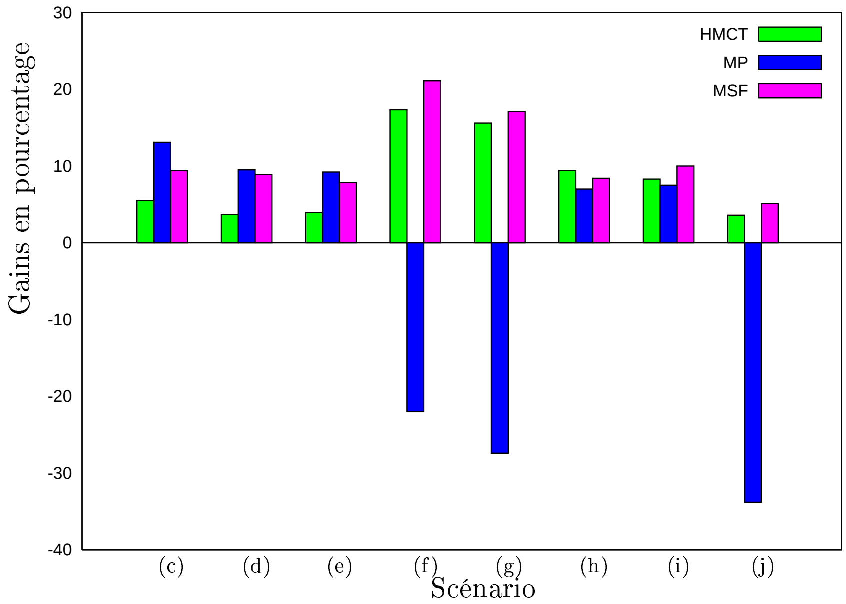


FIG. 5.13 – Gains moyens réalisés *par chaque client* sur le makespan de son application selon l'heuristique et le scénario

une performance crête pour le scénario (f) avec 22% de gain. MP donne de meilleurs résultats pour les scénarii (c), (d) et (e) mais, comme nous l'avons expliqué précédemment, accuse de mauvaises performances pour les scénarii (f), (g) et (j) à cause de l'effet de «barrière d'attente». On voit cependant, avec les scénarii (h) et (i) que la soumission d'autres tâches (ici des tâches indépendantes), peut partiellement régler le problème de la tâche critique. Malgré cela, MP donne de moins bons résultats que HMCT et MSF sur ces deux scénarii.

Temps de réponse moyen. Comme le montre la figure 5.14, toutes nos heuristiques améliorent le temps de réponse de chaque requête indépendante, quel que soit le scénario. Sur les scénarii (h) et (i), MSF donne les meilleures performances avec presque 40% de durée en moins sur chaque requête du scénario (i). C'est aussi pour (h) et (i) que MP se classe le mieux en atteignant plus de 67% de gains pour le scénario (j). Cette performance est toutefois à rapprocher du makespan : MP semble meilleure à ordonnancer des tâches indépendantes que des applications.

Pourcentage de tâches terminant plus tôt. La figure 5.15 est particulière. Tout d'abord, pour une heuristique et un scénario donné, deux informations apparaissent sur les ordonnées : le pourcentage de tâches terminant plus tôt si elles sont ordonnancées par notre heuristique plutôt qu'avec MCT et le rectangle supérieur donne le pourcentage de tâches terminant plus tôt avec MCT qu'avec la nôtre. Le complément à 100% donne évidemment le pourcentage de tâches terminant à la même date. Ensuite, à l'opposé des

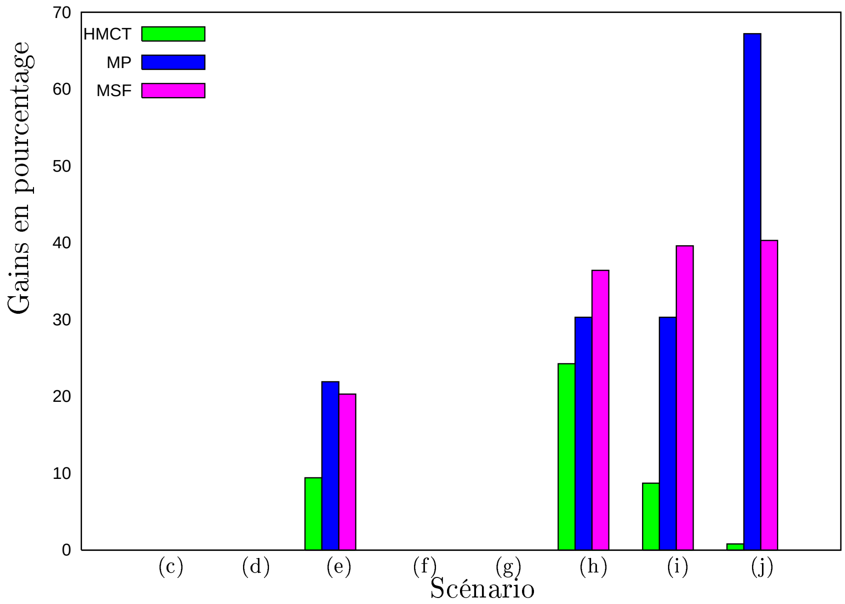


FIG. 5.14 – Gains moyens réalisés *par chaque client* sur le temps de réponse d'une requête selon l'heuristique et le scénario

deux figures précédentes où l'on observait un gain dès que la valeur était positive, on observe ici un gain dès que la surface du rectangle de l'heuristique considérée est plus importante que celle de rectangle supérieur associé.

Nos trois heuristiques donnent ici encore de meilleures performances que MCT quel que soit le scénario considéré, avec toutefois pour le scénario (j) des performances presque identiques entre HMCT et MCT. En moyenne, MP donne les meilleures performances, avec notamment 83% sur le scénario (e) et plus de 90% sur le scénario (j).

Conclusion des expériences HMCT améliore les performances de MCT, quels que soient le scénario et la métrique considérés, grâce aux informations qu'il obtient du HTM. Cependant, les gains réalisés, que l'on peut imputer *a priori* à une meilleure estimation des durées ainsi qu'à la pertinence de la stratégie de MCT (c'est–à–dire minimiser le makespan localement pour tenter de le minimiser à la fin de l'expérience), sont moins importants que ceux d'une stratégie disposant d'une pseudo–mémoire comme MSF, qui prend en compte les délais occasionnés sur les tâches placées précédemment et encore en activité.

Si les bonnes performances de qualité de service de MP sur le scénario (j) sont à nuancer (MP dispose des serveurs les plus rapides pour les tâches indépendantes car l'affectation d'une tâche critique du stencil à un serveur lent bloque la progression de sa soumission), l'heuristique donne de bons résultats dans l'ensemble : seuls des cas particuliers montrent que ses performances peuvent s'écrouler, elle gère mieux que les autres les cas extrêmes où la fréquence des requêtes est élevée et nous verrons dans le chapitre suivant que grâce à sa

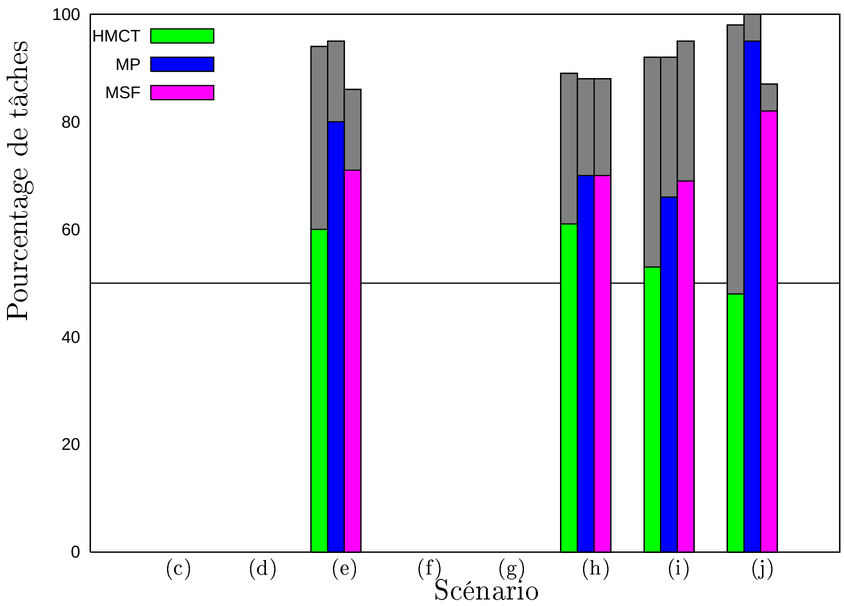


FIG. 5.15 – Pourcentage de tâches terminant plus tôt que MCT selon l'heuristique et le scénario

stratégie de placement, elle permet au HTM en l'état d'assurer les meilleures estimations.

Le HTM confère une certaine mémoire aux heuristiques. MSF conduit ainsi en moyenne aux meilleurs résultats, tous scénarii confondus, ce qui confirme les travaux de Weissmann sur l'importance de la prise en compte des délais dans les politiques de placement de tâches [Wei96].

5.6 Conclusion

Nous avons présenté dans ce chapitre un module de prédiction de performances fondé sur la simulation des requêtes soumises à l'agent d'un environnement de résolution de problème de type GridRPC. Il utilise pour cela une modélisation macroscopique de la plate–forme. Le «gestionnaire de l'historique des tâches», ou HTM (*Historical Trace Manager*), est de fait non–intrusif et donne des estimations instantanément. Sa conception permet de considérer d'autres informations que la date de terminaison de la nouvelle requête dans le choix du serveur. Des heuristiques peuvent ainsi être conçues dans le but d'améliorer les gains escomptés sur le makespan tout en améliorant d'autres critères simultanément.

Ensuite, nous avons décrit de nouvelles heuristiques d'ordonnancement multi–critères capables d'intégrer les informations du HTM afin de parfaire le choix du serveur à chaque nouvelle requête. Grâce à lui, elles disposent indirectement d'un mécanisme de mémoire des décisions prises puisque le nombre d'interférences entre les tâches influe sur leur retard respectif. Parmi l'ensemble des heuristiques regardées, nous avons proposé et étudié cinq heuristiques : Historical MCT, Advanced HMCT, Minimum Perturbation, Minimum SumFlow et Minimum Length.

Le HTM et les cinq heuristiques ont été implantés et les heuristiques ont été comparées à une modélisation de MCT sur des soumissions de tâches indépendantes. Les résultats ont été obtenus à l'aide d'expériences de simulation. Nous avons ainsi pu mettre en évidence certains comportements, valider les performances escomptées et dresser certaines attentes de résultats par rapport à ceux de MCT, en cas de déploiement. En particulier, elles arrivent toutes à améliorer les performances de MCT sur le makespan.

Les modules de prédiction de performances et d'ordonnancement ont été développés et insérés dans l'agent de NetSolve version 1.4.1. Nous avons réalisé une série extensive de tests représentant un total de plus de cinquante jours complets d'occupation ininterrompue des machines. Ils comportent des soumissions de tâches indépendantes, des soumissions simultanées d'applications et les deux en parallèle. Sans pouvoir être exhaustifs, l'objectif est d'avoir une base la plus large possible pour comparer HMCT, MP et MSF à l'implantation de MCT dans NetSolve.

Il est difficile d'établir une comparaison entre les résultats obtenus des expériences de simulation et ceux des expériences réelles, notamment parce que les besoins des tâches, l'hétérogénéité et le nombre des ressources impliquées dans les expériences sont différents. On peut cependant constater une particularité due à la modélisation des expériences de simulation : aucune limite n'est fixée sur la taille mémoire des serveurs, aussi les machines sont supposées accepter toute nouvelle tâche de calcul. En simulation, nos heuristiques surpassent toujours MCT sur le makespan. Cependant, en pratique, sur un nombre limité de ressources dont le degré d'hétérogénéité est plus important, il est difficile d'obtenir un gain sur le makespan pour le même type d'expérience (soumission de tâches indépendantes).

L'étude sur de nombreuses expériences différentes en condition réelle montre la pertinence des heuristiques et donc de l'utilisation du gestionnaire de l'historique des tâches qui fournit les informations conséquentes. Nous obtenons de meilleurs résultats en utilisant simultanément d'autres critères que ceux usuellement regardés dans de tels systèmes et malgré une faiblesse de MP que nous avons expliquée, les heuristiques proposées remplissent les attentes initiales sur les performances : elles surpassent MCT sur plusieurs métriques simultanément.

Chapitre 6

Etude de la précision du HTM et performances des heuristiques

Table des matières

6.1 Introduction

Nous avons vu dans le chapitre précédent que l'utilisation du gestionnaire de l'historique des tâches comme module de prédiction de performances permettait au module d'ordonnancement de saisir des informations nécessaires pour des nouvelles heuristiques multi–critères. Elles engendrent de nettes améliorations de performance par rapport à MCT, quelle que soit la métrique regardée et les applications soumises.

Nous nous intéressons dans ce chapitre à la précision des estimations de la durée des tâches. Nous allons voir quelques cas de soumission où l'on a constaté des prédictions erronées : il arrive qu'à certains moments les informations ainsi calculées par le HTM ne corrèlent plus avec la réalité, voire en divergent. La modélisation peut donc à terme se désynchroniser. Nous avons donc entrepris un travail de synchronisation de la simulation opérée par le HTM avec les exécutions sur la plate–forme afin d'éviter définitivement tout problème de divergence et d'accroître l'acuité déjà constatée. Les heuristiques disposant en conséquence d'estimations de meilleure qualité, nous avons entrepris d'observer leurs «nouvelles» performances.

Le plan de ce chapitre est le suivant : nous étudions dans la section 6.2 la précision des informations délivrées par le HTM au cours du temps. Pour parfaire les informations données à l'ordonnanceur, des améliorations ont été apportées au fonctionnement du HTM. Nous décrivons dans la section 6.3 l'algorithme utilisé ainsi que les mécanismes nécessaires à son emploi. Dans la section 6.4, nous étudions la nouvelle précision des prédictions des durées du HTM et nous montrons les retombées sur le comportement des heuristiques et leurs nouvelles performances. Nous concluons à la section 6.5

6.2 Analyse de la précision du HTM

L'étude réalisée sur la soumission de quelques tâches concurrentes pour une ressource montre que la modélisation utilisée donne effectivement de bons résultats sur la qualité des estimations. Cependant, en toute rigueur, elle ne suffit pas à assurer le maintien de la qualité des estimations par rapport au temps ou à la fréquence d'arrivée des tâches par exemple. Pour évaluer le HTM, la précision de ses estimations a donc été enregistrée pendant chacune des expériences qui ont servi à analyser et comparer les performances des heuristiques dans le chapitre précédent, lorque la plate–forme était composée de spinnaker, artimon, fonck et soyotte : une partie du scénario (b) et les scénarii de (c) à (j) sont donc concernés.

Notons que plusieurs facteurs sont peut-être à considérer : grâce au HTM, nous avons observé que les heuristiques ayant *a priori* une meilleure vision de ce qui se déroule sur la plate-forme, les ressources ont tendance à être plus exploitées (c'est en particulier le cas pour HMCT qui ne décide que par rapport au temps de terminaison de la dernière requête). En conséquence, la modélisation des ressources et notamment celles de calcul qui est basée sur le temps partagé, est sévèrement éprouvée. Dès lors, la précision du HTM est dépendante des décisions des heuristiques et doit aussi être prise en considération dans le choix de l'heuristique utilisée, en plus de ses performances.

Nous avons distingué trois grands types de comportements dans l'évolution des estimations du HTM, à cause principalement de la présence de relations de dépendance entre les soumissions. Aussi, après avoir introduit la génération des résultats contenant les informations à analyser et la méthode pour lire les graphes résultants, nous décrivons d'abord les cas où le HTM donne des prédictions avec une faible erreur, puis ceux où l'on peut observer des dégradations temporaires pour finalement présenter ceux où les estimations peuvent à terme diverger de la réalité. Nous donnons ensuite quelques observations avec leur explication sur le comportement général du HTM.

6.2.1 Présentation des résultats

Lorsqu'on étudie une précision, on considère généralement l'erreur relative entre ce qui est mesuré et ce qui est estimé. A la suite d'une expérience, on peut déduire différentes informations parmi lesquelles l'erreur instantanée, l'erreur maximale, l'erreur moyenne, son écart–type et l'évolution de son comportement. Dans notre cas, l'erreur doit être considérée pour chaque serveur et dépend du nombre de tâches qui lui sont affectées. *L'erreur instantanée* ne nous intéresse que par l'identification des paramètres conduisant son *évolution*. Par exemple, elle peut varier en fonction du temps ou des perturbations que la compétition des tâches pour la ressource induit les unes sur les autres. De même, l'erreur moyenne est intéressante pour savoir avec quelle précision la stratégie de placement de l'heuristique est respectée en moyenne. Toutefois, elle doit s'accompagner de l'écart–type associé. Ces données peuvent influencer le choix de l'heuristique à utiliser dans l'environnement de résolution de problème. Nous donnons quelques résultats sur le pourcentage d'erreur moyenne observée et l'écart–type correspondant lors du scénario (b) pour HMCT, MP et MSF en fonction du serveur considéré dans le tableau 6.1.

Génération des graphes. Pour générer les graphes de cette section, nous avons implanté dans le client la tenue d'un fichier de logs afin de recueillir les informations concernant les tâches soumises et leur durée : en effet, dans la version 1.4.1 de NetSolve, il n'y a aucun mécanisme d'identification possible dans le fichier de logs géré localement par l'agent nous permettant d'établir la correspondance entre une tâche d'une application donnée et la soumission enregistrée par l'agent. La durée réelle est donc observée sur la machine client tandis que les estimations du HTM sont prises sur la machine agent. Ensuite, par comparaison entre ces données, les corrélations sont effectuées. Notons que deux tâches arrivant approximativement à la même date peuvent être «interverties» et la comparaison faussée. Nous utilisons les informations récupérées sur la sortie standard du client pour gérer les cas de «litiges» lorsqu'ils se présentent.

Lecture des graphes. Nous donnons les résultats sur l'acuité du HTM principalement sous forme de graphes : ils présentent sur l'axe des abscisses la date de soumission de chaque tâche, et deux informations apparaissent selon l'axe des ordonnées : le ratio de la dernière estimation de la durée faite par le HTM par la durée réelle constatée à la fin du run (qu'on appelle *durée post-mortem*) apparaît en gris foncé ; le nombre réel constaté de tâches ayant été exécutées concurremment à la tâche soumise à la date donnée est en gris clair. Pour un tel graphe, un aggrandissement est généralement pourvu afin de mieux discerner la précision. En revanche, seuls les graphes des serveurs spinnaker et artimon sont donnés : pour des raisons de performance, ce sont les plus utilisés ; quelle que soit l'heuristique, il y a peu d'interférences entre les tâches qui sont allouées aux serveurs sparc fonck et soyotte (lorsqu'il y en a) et quelle que soit la tâche qui leur est affectée, la précision de l'estimation est très proche de 100%.

Comme il est mentionné avant, le ratio est la dernière estimation faite par le HTM par la division de la durée réelle. L'estimation est donc celle faite lors de la soumission de la tâche et mise à jour en fonction des soumissions ultérieures. Plus le ratio est proche de la valeur 1, plus la précision est bonne. Lorqu'il est supérieur à un, l'estimation de la durée est plus grande que celle observée en réalité. Comme nous le verrons, les conséquences sont immédiates sur l'ordonnancement.

6.2.2 Fiabilité et robustesse des estimations

Les figures 6.1 et 6.2 de cette section sont obtenues de la moyenne des estimations observées lors d'une expérience du scénario (b). Ceci est possible ici car les décisions d'ordonnancement lors de deux runs consécutifs sont ici les mêmes (les dates de soumission étant fixes, un run ne peut différer que dans le cas très particulier où une légère erreur sur la date de soumission induit une différence notable dans les calculs opérés par l'heuristique). Chaque figure est constituée de quatre graphes, deux pour chacun des deux serveurs les plus rapides : spinnaker (graphes du haut) et artimon (graphes du bas). Les graphes tirés de l'observation de l'estimation au cours d'un run du scénario (j) sont donnés pour spinnaker (à droite) et artimon (à gauche) dans la figure 6.3.

La figure 6.1 montre la précision obtenue pour une expérience où les tâches arrivent à une fréquence moyenne de 20 secondes et sont réparties sur les ressources par HMCT. On constate qu'à de rares exceptions, les durées estimées des tâches sont très proches mais restent supérieures aux durées réelles. Là où les ratios augmentent, on observe qu'il y a au moins quatre interférences avec des tâches différentes. Le tableau 6.1 indique que l'erreur moyenne de l'estimation faite par le HTM pour le scénario dont fait partie cette expérience est de 3.1 % pour un écart–type σ de 5.3 sur spinnaker et de 1.75 % avec $\sigma = 2.4$ sur artimon. Fonck et soyotte ne se voient affecter aucune tâche.

La figure 6.2 montre la précision obtenue pour une expérience où les tâches sont indépendantes et arrivent à une fréquence moyenne de 17 secondes. L'heuristique employée est MP. Ici encore, la qualité des estimations est très bonne mais décroît lorsque plus de quatre tâches interfèrent avec l'exécution d'une autre, par exemple à $t = 7200$ secondes sur spinnaker. Comme l'indique le tableau 6.1, le HTM «piloté» par MP donne sur ce scénario une erreur moyenne de 3.6 % avec un écart–type de 3.2.

Nous présentons finalement avec la figure 6.3 les résultats d'un run du scénario (j)

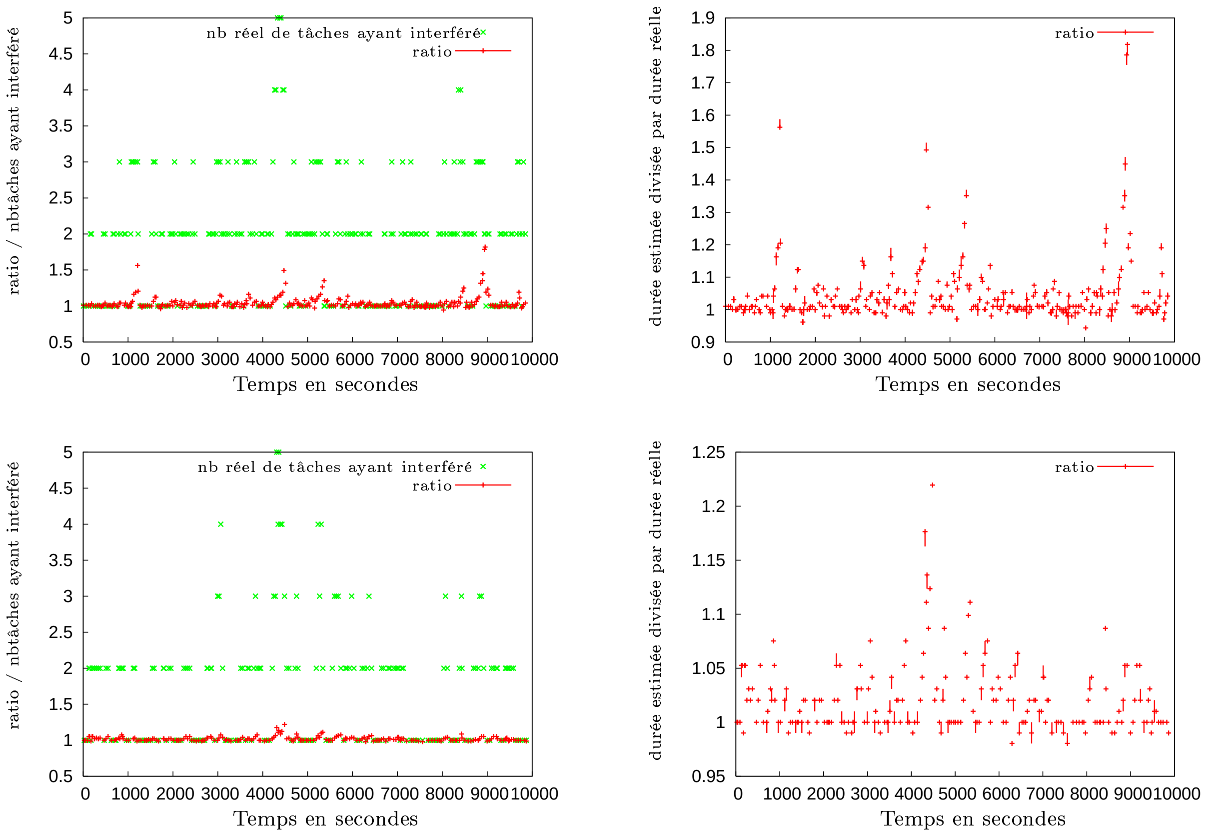


FIG. 6.1 – Scénario (b), μ = 20 sec : précision des estimations de la durée des tâches allouées par HMCT sur spinnaker en haut, sur artimon en bas

ordonnancé par MP. On peut observer la soumission «par paquet de tâches» propre à la soumission du stencil car la fréquence moyenne d'arrivée des tâches indépendantes n'est pas suffisante pour lisser ce phénomène. Les erreurs sont ponctuelles et correspondent encore à plus de 4 tâches ayant interféré avec la tâche observée. Le HTM surestime le plus souvent la durée réelle.

Les cas présentés ici montrent une faible erreur moyenne et un faible écart–type. On peut donc qualifier les estimations de fiables. Il n'y a pas de dégradation des estimations au cours du temps. On peut donc parler de robustesse du HTM sur ces cas de soumissions.

6.2.3 Dégradations passagères des estimations

Nous montrons un exemple intéressant sur le comportement des estimations des durées par le HTM dans la figure 6.4. L'expérience provient du scénario (b) et les tâches indépendantes sont soumises avec une fréquence moyenne de 17 secondes. L'heuristique déployée dans le module d'ordonnancement de l'agent de NetSolve est MSF. Si on remarque que le HTM surestime encore les durées des tâches (les ratios sont supérieurs à 1), on s'aper-

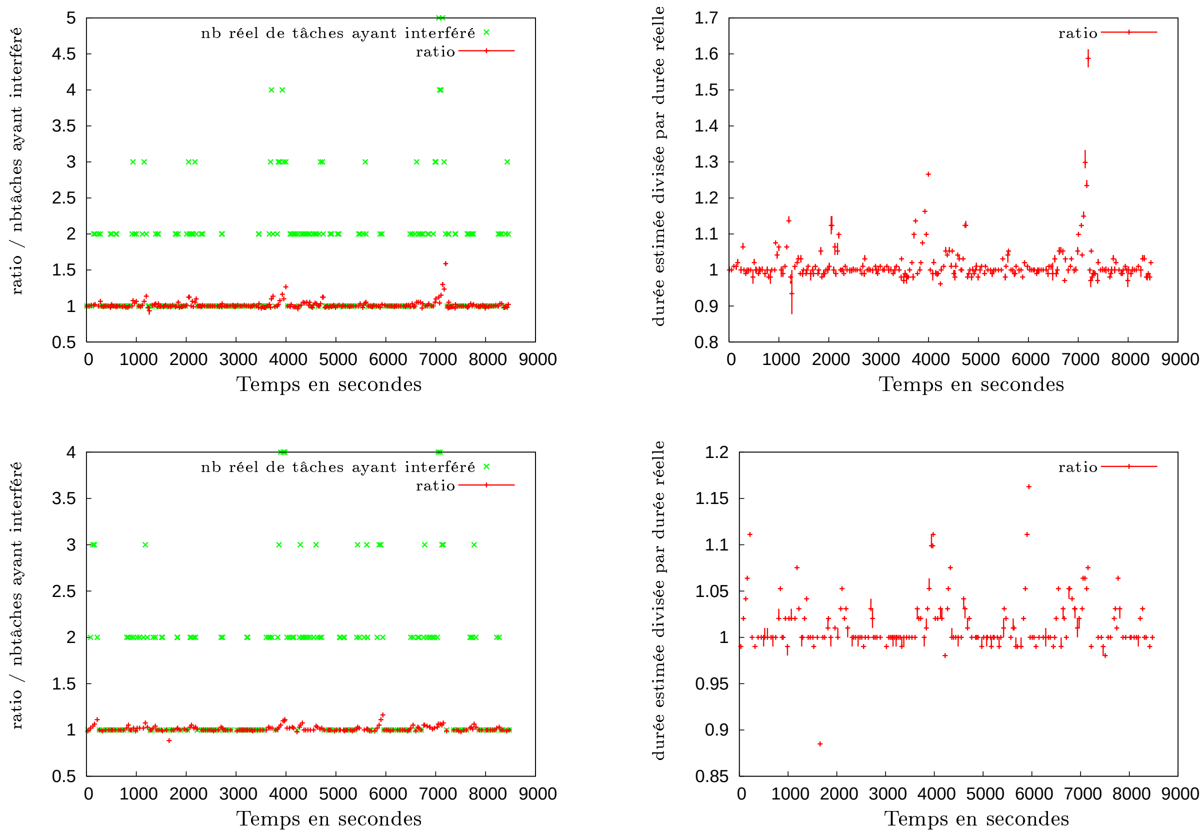


FIG. 6.2 – Scénario (b), $\mu = 17$ sec : précision des estimations de durée des tâches allouées par MP sur spinnaker en haut, sur artimon en bas

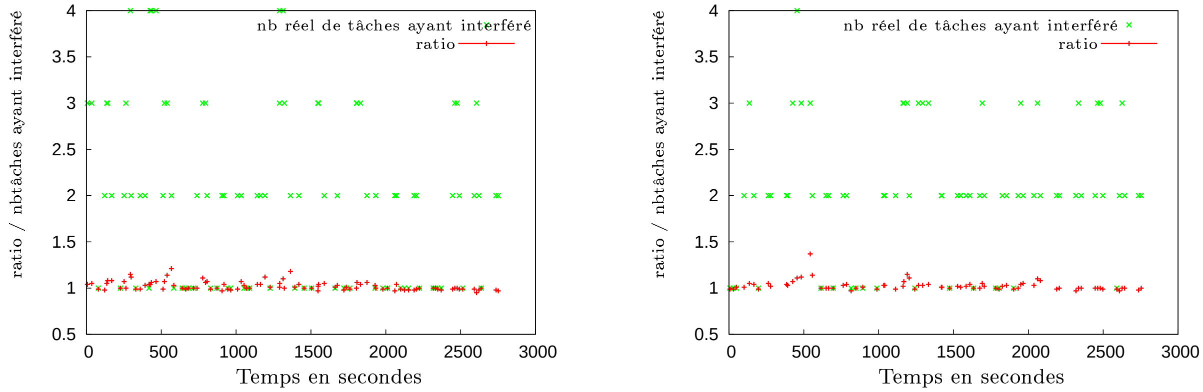


FIG. 6.3 – Scénario (j) : précision des estimations de la durées des tâches lors d'un run ordonnancé par MP observée sur spinnaker (gauche) et artimon (droite)

çoit aussi que l'évolution n'est pas constante au cours du temps. Au contraire de MP qui place des tâches sur les serveurs les moins rapides et malgré la fréquence assez élevée des requêtes, MSF choisit entre spinnaker et artimon. Cela engendre beaucoup d'exécutions concurrentes sur ces serveurs : plusieurs pics à 8 tâches qui interfèrent pendant l'exécution

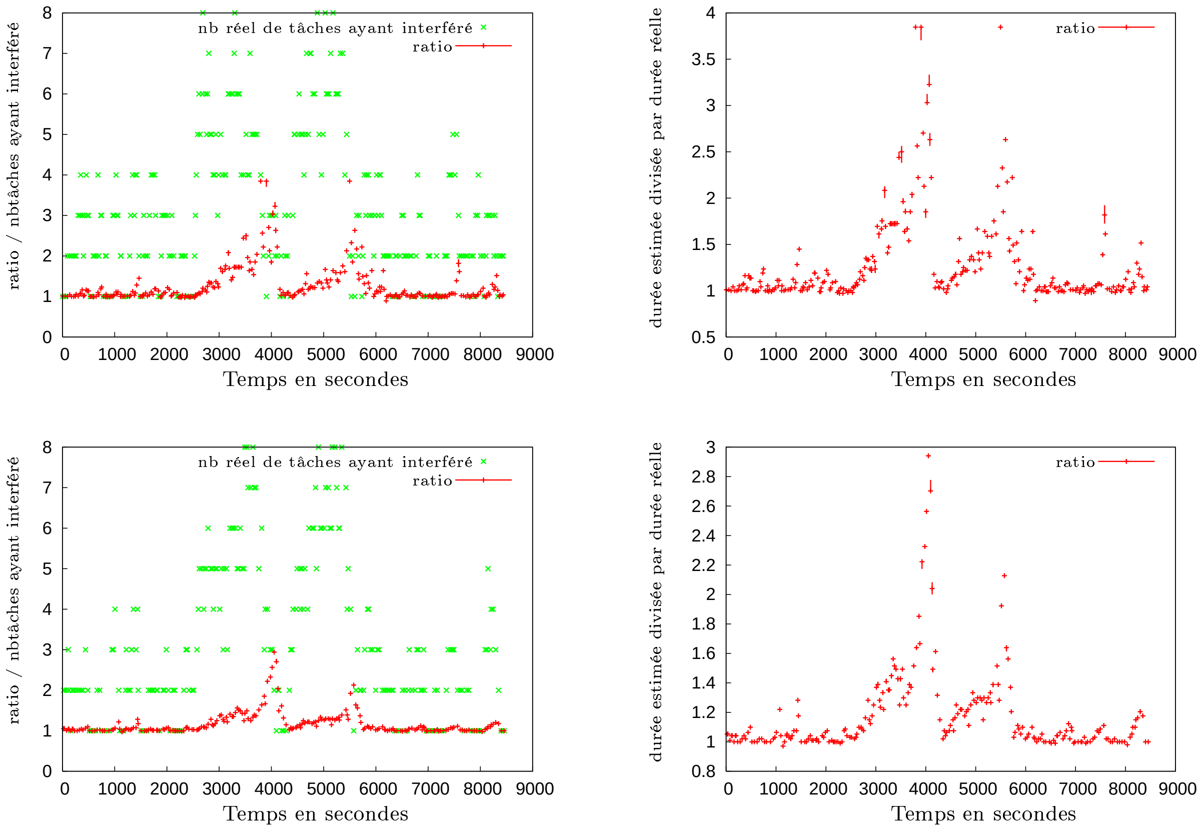


FIG. 6.4 – Scénario (b), $\mu = 17$ sec : précision des estimations de la durée des tâches placées par MSF sur spinnaker en haut, sur artimon en bas

de celle observée. On observe un accroissement des erreurs conséquent sur les estimations des durées des tâches suivantes. On constate ainsi une surestimation égale à 284 % du temps réel pour $t = 3905$ secondes ! L'erreur moyenne constatée pour MSF sur ce scénario est de 11.7 % avec un écart–type $\sigma = 13.4$ comme il est indiqué dans le tableau 6.1. En revanche, le HTM regagne complètement son acuité peu de temps après ce pic : l'heuristique, sous–estimant la puissance de calcul de spinnaker (et de artimon), leur affecte les tâches en conséquence. La complémentarité du HTM et de l'heuristique induit ici un *comportement auto-régulateur* ce qui donne une certaine robustesse au HTM. De plus, on voit nettement ici que le temps n'est pas le facteur influençant le plus l'acuité des informations du HTM à l'instar du nombre d'interférences perçues par certaines tâches à des instants donnés.

La figure 6.5 concerne les affectations sur spinnaker et artimon par MSF d'une application stencil à base de tâches de type wastecpu et de taille 1 du scénario (f). On remarque immédiatement l'aspect «en dents de scie» de la variation des estimations du HTM et du nombre de tâches ayant interféré. On obtient ici un cas plus extrême que le cas précédent. On remarque immédiatement que le nombre de tâches perturbantes est toujours très élevé ; cela est dû en partie à la structure du stencil. De plus, on retrouve un phénomène

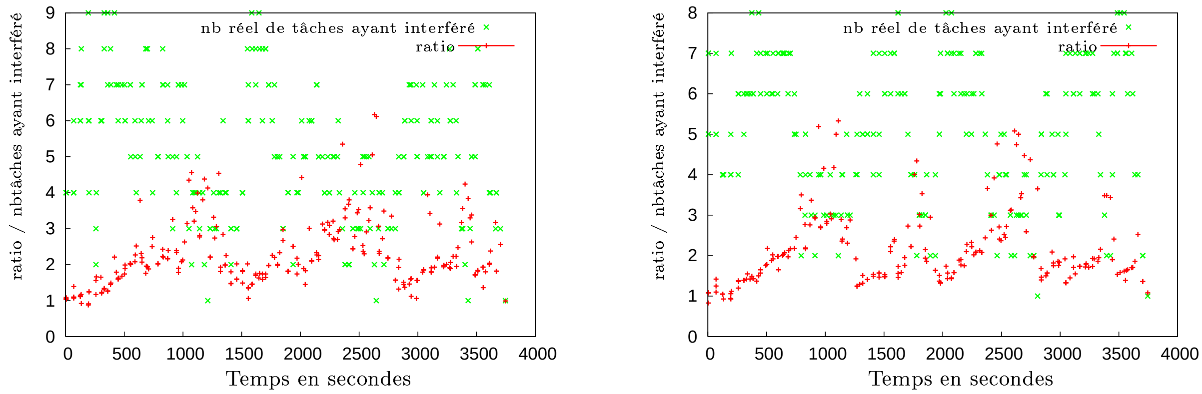


FIG. 6.5 – Scénario (f) : précision des estimations de la durée des tâches lors d'un run ordonnancé par MSF observée sur spinnaker (gauche) et artimon (droite)

d'auto–régulation : la surestimation des durées induit un comportement erroné de MSF qui sous–estime la puissance de calcul des serveurs et qui affecte des tâches sur fonck et soyotte. Les estimations peuvent ainsi retrouver une bonne acuité. MSF affecte alors une majeure partie de l'ensemble des soumissions aux serveurs les plus rapides. Les tâches sont soumises par salves à cause des relations de précédances Ici encore, on voit donc l'importance du nombre de tâches qui interfèrent avec une autre dans l'estimation de sa durée. Par contre, le temps ne semble pas avoir un rôle prépondérant.

6.2.4 Divergence entre les prédictions et la réalité

Nous présentons dans la figure 6.6 les résultats que nous avons observés lors d'un run des scénarii de (e) ordonnancé par HMCT. On constate immédiatement que la qualité des estimations se dégrade très rapidement. Elle décroît toujours avec le nombre de tâches qui interfèrent avec une tâche donnée, mais comme la figure le montre, la dégradation semble plus persistante puisque pour $t > 5000$ secondes, on observe une baisse du nombre d'interférences et la précision des estimations ne revient qu'à la fin du run. Elle semble donc aussi, pour ce scénario, fonction du temps.

6.2.5 Constats et explications

Les graphes 6.1 et 6.2 donnent un exemple d'exécution contribuant aux moyennes données dans le tableau 6.1. On peut y voir les résultats des moyennes des erreurs et les écart–types correspondant des estimations du HTM pour le scénario (b) pour $\mu = 20$ et $\mu = 17$ secondes pour les heuristiques HMCT, MP et MSF. On constate que la stratégie de placement de MP, qui consiste à minimiser les perturbations, permet au HTM de minimiser les erreurs et les écart–types correspondants. On pouvait facilement supposer que l'utilisation de HMCT, qui tente de profiter de la puissance des serveurs au maximum

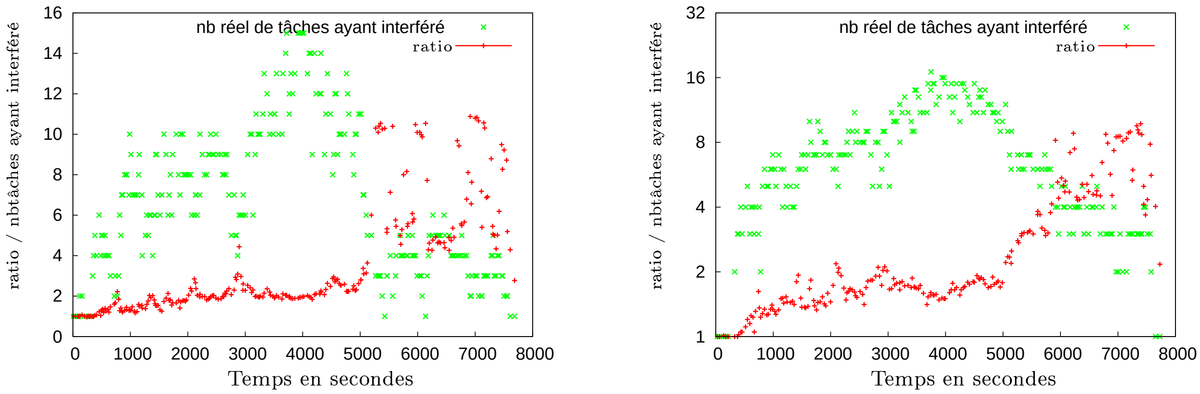


FIG. 6.6 – Scénario (e) : précision des estimations de la durée des tâches lors d'un run ordonnancé par HMCT observée sur spinnaker (gauche) et artimon (droite)

Scénario	Serveur	HMCT % erreur (σ)	MP % erreur (σ)	MSF % erreur (σ)
(b), $\mu = 20$	spinnaker	**3.71** (5.3)	**1.08** (1.6)	**3.02** (4.2)
	artimon	**1.75** (2.4)	**0.74** (1.2)	**1.87** (2.3)
	soyotte	–	**0.18** (0.4)	–
	fonck	–	**0.14** (0.1)	–
(b), $\mu = 17$	spinnaker	**13.9** (14.6)	**3.1** (5.3)	**11.7** (13.4)
	artimon	**9.9** (11)	**3.6** (3.2)	**8.8** (10.3)
	soyotte	–	**0.2** (0.3)	**0.3** (0)
	fonck	–	**0.3** (0.3)	**0.8** (0.3)

TAB. 6.1 – Scénario (b) : pourcentage moyen d'erreur et écart–type selon les serveurs et les heuristiques

pour satisfaire la terminaison de chaque nouvelle requête, ne serait pas profitable au module d'ordonnancement. Le HTM ne se comporte qu'un peu mieux avec MSF.

Le nombre de tâches qui interfèrent est prédominant. Dès qu'il devient supérieur à 4, les estimations commencent à se dégrader. Ceci est dépendant de la fréquence d'arrivée des requêtes observée par l'agent et de la stratégie de l'heuristique utilisée, puisqu'en «allouant/distribuant» les tâches sur les serveurs, elle détermine la fréquence d'arrivée des tâches qu'ils «subissent». De manière générale, les durées sont surestimées, donc les ressources sont potentiellement moins utilisées que ce qui devrait être fait par la stratégie. On peut constater que cela induit un phénomène d'auto-régulation lorsque les soumissions sont des tâches indépendantes ou lorsque les graphes des applications le permettent.

Nous avons aussi constaté que le temps semblait être un paramètre de la dégradation des prédictions du HTM. Ceci traduit *de facto* une désynchronisation temporelle entre le HTM et la réalité. Nous expliquons par un exemple donné dans la figure 6.7 ce qu'il peut se produire lorsqu'un graphe de tâches est soumis. On y observe deux diagrammes de Gantt :

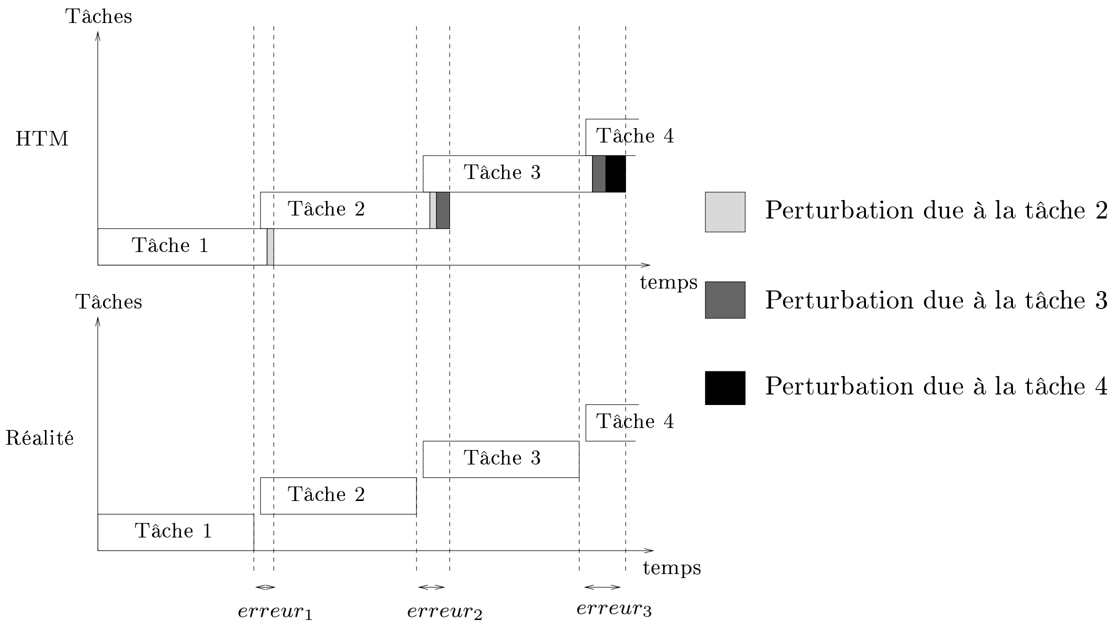


Fig. 6.7 – Possibilité de divergence entre la simulation et la réalité

en haut, celui que le HTM dresse au fur et à mesure des arrivées des requêtes ; en bas, ce qui se déroule en fait dans la réalité. Quatre tâches ont été affectées à un serveur. Chaque tâche dure moins longtemps que ce qu'indique l'exécution du benchmark (dans notre travail, il s'agit de la durée moyenne observée sur cinq exécutions sur le serveur dédié). On s'aperçoit de la croissance d'une erreur au fur et à mesure du temps. Cette erreur est en général petite, et les exécutions sur des tâches indépendantes montrent qu'elles sont négligeables dans beaucoup de cas. Cependant, si les tâches sont issues d'une même application (par exemple une application 1D-mesh), alors on observe une dégradation persistante dans le temps (comme lors de l'expérience représentée dans la figure 6.6). L'erreur grossit d'autant plus qu'elle est pondérée par le nombre de tâches à l'instant considéré. Il nous faut donc un mécanisme de synchronisation du HTM avec la réalité pour s'assurer de la généralité de son utilisation, quel que soit le type des soumissions et quelle que soit l'heuristique utilisée.

6.3 Mécanismes de contrôle pour la synchronisation

6.3.1 Problématique

La prédiction de la durée d'une tâche est dépendante du nombre de tâches interférant pendant son exécution. Des erreurs notables apparaissent dès que leur nombre dépasse 4. De plus, des exemples d'exécution montrent qu'il est possible que la simulation de l'environnement par le HTM se désynchronise de la réalité, au moins temporairement.

L'une des causes supplémentaires s'explique par les relations de précédences qui lient les requêtes et dont l'agent n'a pas connaissance. Or, nous devons nous assurer que le HTM est capable de délivrer des informations précises aux heuristiques, quelles que soient les requêtes et leur fréquence. Le HTM doit donc être synchronisé aux évènements de la plate–forme afin de résoudre ces problèmes. Nous expliquons dans cette section le travail qui a été fait dans ce but.

6.3.2 Message de terminaison

Afin d'assurer une meilleure réactivité, l'agent doit être informé lorsqu'une requête termine sur un serveur. Par le biais d'un *message de terminaison*, le gestionnaire de l'historique des tâches peut rectifier les données qu'il a enregistrées et calculées sur cette requête et prendre immédiatement en compte ces corrections dans l'estimation des durées des autres tâches dès la prochaine soumission.

Le HTM utilise la durée de calcul à vide qu'exige la requête sur le serveur pour pouvoir simuler son exécution (section 5.2). L'information de la terminaison de la tâche n'est donc pas suffisante en soi. Il faut tout d'abord une simulation de la plate–forme pour calculer la durée à vide. La valeur ainsi trouvée sera ensuite injectée dans les simulations qu'occasionne chaque nouvelle soumission.

Pour simplifier l'écriture, l'algorithme 11 ne concerne que la ressource de calcul où vient de se finir la tâche : il retourne le calcul de la durée de calcul sur le serveur. En pratique, nous considérons l'ensemble des ressources.

L'algorithme est composé de deux sections principales insérées dans une boucle qui simule une par une l'arrivée des tâches affectées au même serveur que la tâche dont on a reçu le message de terminaison. Pour chacune d'elles, on utilise sa date d'arrivée et sa durée à vide enregistrée précédemment. Pour la tâche terminée, l'algorithme utilise sa date d'arrivée et sa date de terminaison réelle. Grâce à ces informations, il dresse le diagramme de Gantt tout en sauvegardant la durée que nous souhaitons.

La première section correspond à une mise à jour des quantités «durée restante à effectuer» et «date de terminaison» des tâches dont on a déjà simulé l'arrivée, en fonction de la date d'arrivée de chaque nouvelle tâche simulée t_n. Pour cela, la durée sans évènement est tout d'abord calculée : c'est l'inter–arrivée des tâches t_n et t_{n-1}. La durée qui nous intéresse est alors calculée et renvoyée si la tâche se termine pendant ce laps de temps. Une fois les quantités mises à jour, l'algorithme passe à la simulation de l'exécution des tâches pour obtenir leur date de terminaison : il peut ainsi savoir si des tâches auront fini dans la première partie de la prochaine itération. L'algorithme quitte dans cette deuxième phase si la tâche considérée est la dernière affectée sur le serveur ou si c'était la dernière itération.

On remarquera qu'une fois la fonction de correction appelée, la durée à vide enregistrée de la requête sera corrigée de sorte qu'elle terminera à *la même date* dans la simulation

Algorithme 11: Simulation d'une tâche sur la modélisation de la plate-forme

Données : Le HTM dispose des informations déjà enregistrées et des diagrammes de Gantt qu'il en a tiré sur les ressources ;
La date réelle de terminaison de la requête t_R dont il faut calculer la durée à vide ;

Résultat : Durée totale sur le serveur non chargé que requiert la requête qui vient de finir

début
- A $\leftarrow$ les tâches t_i de P(t_R), classées par ordre d'arrivée ;
- **si** $|A| = 1$ **alors retourner** $C_R - a_R$;
- **pour chaque** $t_i \in$ A **faire**
  - Initialiser les T_i à $a_i + d_i$ et la durée restante à effectuer d_i^r à d_i ;
- **pour chaque** *tâche* t_n *de* A*,* $n{>}1$ **faire**
  - Calculer la durée sans évènement : $E_n = a_n - a_{n-1}$;
  - /* mise–à–jour*/;
  - B = { tâche t_i tq ($t_i \in A$ et terminée pendant E_n) } ;
  - *duree* $\leftarrow$ 0 ;
  - **si** $B \neq \emptyset$ **alors**
    - **si** $t_R \in B$ **alors**
      - Ordonner les tâches de B selon la durée restant à effectuer sur le serveur non chargé ;
      - Simuler les tâches jusqu'à C_R en mettant à jour les durées restantes de chaque tâche, et le temps simulé écoulé dans *duree* ;
      - **retourner** *la durée simulée de* t_R *pour finir à* C_R*, c'est–à–dire duree*
    - Simuler les exécutions des tâches pour obtenir les dates de terminaison;
  - Simuler l'exécution des tâches : mettre à jour la durée restante à effectuer par rapport a_n et le temps simulé écoulé dans *duree* ;
  - /* obtention du diagramme de Gantt provisoire */;
  - **si** $a_R < a_n$ **alors**
    - Simuler l'exécution des tâches jusqu'à C_R et mettre à jour la durée simulée dans *duree* ;
    - **si** R =| A | ou n =| A | **alors retourner** *duree* ;
  - Calculer la date de terminaison des tâches $t_i, i \leq n$;

fin

de la plate-forme du HTM *et* dans la réalité.

6.3.3 Identification d'une tâche

Jusqu'à présent, chaque requête était identifiée pour que le HTM enregistre les informations relatives à la tâche correspondante : date de soumission, quantité de données à transférer avant et après le calcul, etc. L'identification était locale au HTM dans l'agent. Cependant, quelle que soit l'entité qui prévient de la terminaison d'une tâche (client ou serveur), il doit y avoir cohérence entre elle et l'agent : elle doit nécessairement communiquer la même référence. L'identification devient donc globale au système.

L'environnement étant centralisé sur l'agent, il suffit d'implanter un mécanisme simple de distribution d'identifiant unique pour chaque requête. L'incrémentation d'un compteur pour attribuer le numéro d'identification global à chaque requête répond le plus simplement aux besoins de l'environnement.

6.3.4 Implantation dans NetSolve

Algorithme

Pour appliquer l'algorithme, il faut la date d'entrée en phase calcul (qui est la fin de la phase transfert des données en entrée), la date d'entrée en phase communication des données de sortie et la date de terminaison de la tâche.

Les codes des serveurs sont instrumentés : des *timers* totalisent le temps mis lors de l'émission des entrées, du calcul et du transfert des données de sortie. Ces valeurs permettent ainsi d'accomplir l'ensemble des simulations requises dans chaque phase.

Identifiant unique

L'identifiant de la requête est initialisé à -1. Il est généré lors de la soumission à l'agent. Il est ensuite communiqué entre les différentes entités lors des échanges en rapport avec la requête pour finalement prévenir l'agent lors de sa terminaison.

Mécanismes de synchronisation

Nous schématisons dans la figure 6.8 les principaux échanges entre les entités de l'environnement de NetSolve au cours du temps (de haut en bas). Nous avons représenté les appels des fonctions propres au code de NetSolve avec un souci de clarté plus que de correction : à cause des multiples moyens de soumission d'une requête, les processus de communication et imbrications des fonctions sont naturellement plus complexes. Nous bornerons donc les étapes d'une soumission aux informations données dans la figure.

Tout d'abord, on remarque que deux processus tournent sur une machine cliente : le programme contenant l'appel de soumission d'une requête et un démon proxy, qui fait le lien avec les autres entités de la plate–forme. Les requêtes passent par le proxy. Lorsque le client a soumis sa requête avec un appel non bloquant et veut ensuite connaître son état, il demande auprès du proxy qui dispose de l'information. Chaque client possède donc avec son proxy des mécanismes d'identification pour la même requête. De même, le proxy et l'agent dispose d'un identifiant pour une même requête. Cependant, ces mécanismes d'attribution ne sont pas suffisants pour identifier de manière unique la tâche sur *l'ensemble* de la plate–forme. Or, l'identité de la requête doit naturellement être connue par tous et être communiquée lors des échanges. Ainsi, l'application de l'algorithme de correction est cohérente.

Dans la première phase d'une soumission, après l'appel de résolution de NetSolve client, la fonction `CP_sendjobrequest()` contacte le proxy qui se charge d'informer l'agent de la requête. Il lui fournit les informations statiques sur le problème demandé. Ainsi, l'agent peut, lors de l'appel à `assignServer()`, classer les serveurs capables de résoudre le problème donné, en appliquant la stratégie de placement de l'heuristique utilisée (nos heuristiques font un appel au HTM pour répondre). L'agent affecte à ce moment un numéro unique à la requête. Il contacte en retour le proxy pour lui communiquer la liste des serveurs. Le serveur reçoit un message du proxy (`submitProblemToServer`) afin qu'il anticipe le stockage des entrées du problème qui vont lui être soumises, d'où l'appel à `processProblemSolve()` puis celui de `standardservice()` dans un `fork()`. On note que chaque entité a maintenant l'identité globale de la requête traitée, en plus des informations échangées initialement.

Avec l'appel à `CS_sendInput()`, le client conctacte le service `standardservice()` du serveur pour transférer les données nécessaires au calcul du problème demandé. Une fois les données soumises, le serveur exécute la tâche demandée.

Lorsque la tâche est finie, le serveur notifie la terminaison du calcul au proxy en envoyant le message `NS_PROT_JOB_REPORT`. Le client récupère les sorties de sa requête. Une fois le transfert terminé, le serveur prévient l'agent par l'envoi d'un message `NS_PROT_JOB_COMPLETED_FROM_SERVER` alors que le client prévient le proxy qui le signale à son tour à l'agent en lui envoyant le message `NS_PROT_JOB_COMPLETED`. L'identifiant global de la tâche est également communiqué lors de ces échanges.

Dans notre implantation, le HTM corrige ses données sur la réception du message de terminaison en provenance du serveur par un appel à `updateHTM()`.

Remarques.

- L'implantation de l'ensemble des échanges d'informations, d'attribution d'un identifiant unique à chaque requête et de correction des informations, a été effectué dans la version 2.0 de NetSolve. Le HTM et les heuristiques ont donc été «adaptés» pour cette version.
- Pour transmettre l'identité globale, le HTM n'est pas plus intrusif dans NetSolve qu'il ne l'était précédemment car il utilise les périodes de communication où les entités s'échangeaient déjà des informations.
- Nous avons validé l'implantation de l'ensemble des mécanismes sur plusieurs architectures et plate-formes pour les jeux de tests d'installation de NetSolve, en plus de

nos expériences. Le travail est intégré dans les nouvelles versions de NetSolve [54].

- VisPerf [LDR02] est un *module de monitoring graphique* pour environnements de calcul sur la grille. Son utilité est complémentaire à NetSolve : il permet de surveiller en temps réel la charge des différentes ressources de calcul de la plate-forme. Il se sert des mécanismes d'échanges que nous utilisons pour synchroniser le HTM à la réalité dans ce but.
- Le retour de la terminaison d'une tâche a d'autres avantages : il permet d'envisager le renvoi d'une liste classée des serveurs, tout comme le fait MCT. L'identité du serveur ayant exécuté la requête est connue dès lors qu'il contacte l'agent, ce qui permet de re–simuler si besoin est, l'ensemble des soumissions faites depuis l'arrivée de cette requête et de récupérer ainsi une précision convenable ; NetSolve–2.0 permet à l'utilisateur de pouvoir arrêter sa requête en cours. Le message de terminaison de la requête en plus de la prise en compte du message d'interruption peut permettre au HTM de préserver l'acuité des informations dispensées au module d'ordonnancement.

[54] http ://icl.cs.utk.edu/netsolve/software/index.html

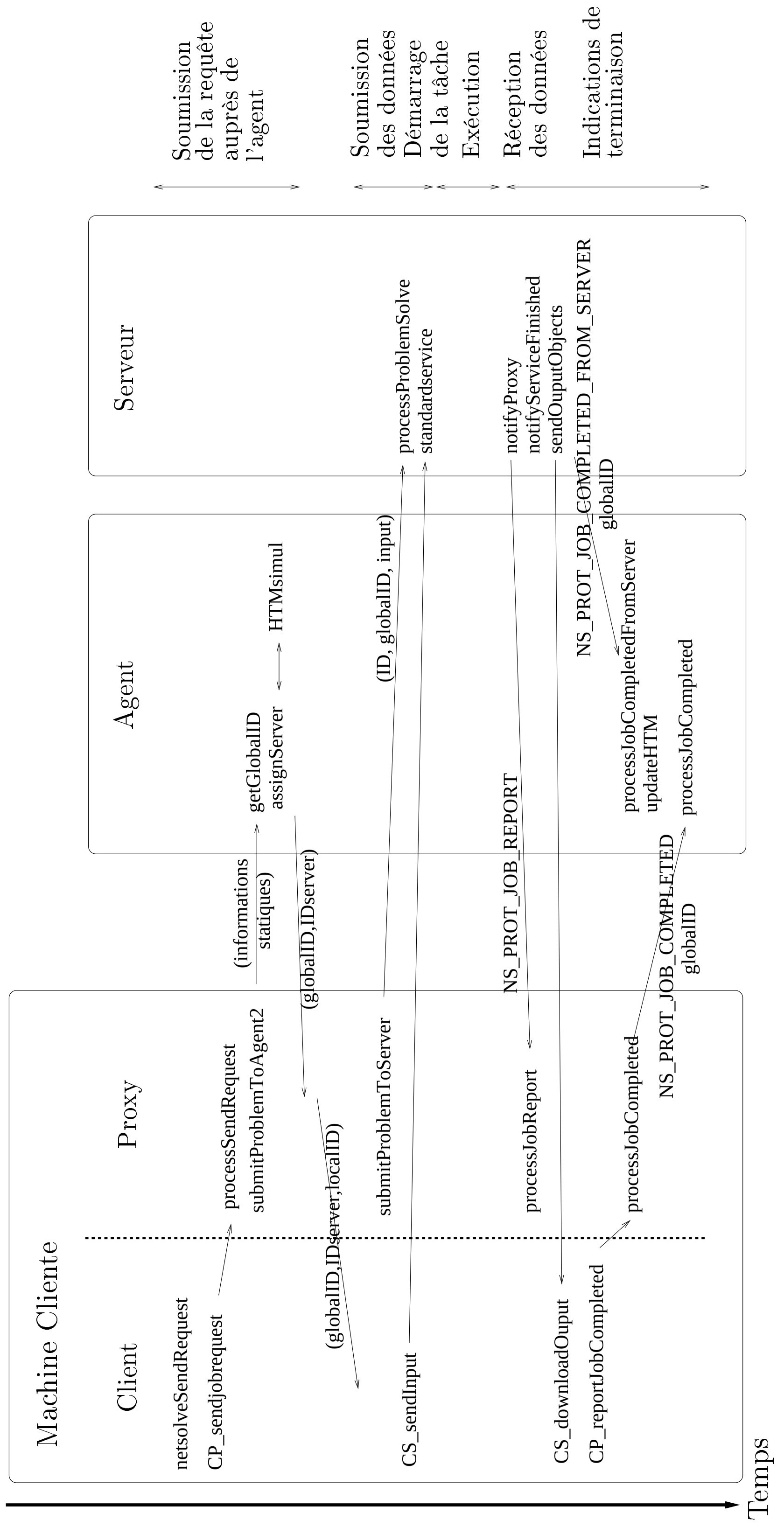


Fig. 6.8 – Schéma des étapes d'une soumission dans NetSolve et synchronisation du HTM

6.4 Expérimentations et résultats

Nous venons de décrire dans la section précédente les mécanismes de synchronisation que nous avons implantés et utilisés dans NetSolve pour affiner les prédictions du HTM. Après avoir donné les modalités des expériences, nous présentons les différents tests qui ont été faits pour valider son acuité dans la section 6.4.2 puis les performances des heuristiques qui bénéficient de meilleures informations dans la section 6.4.3.

6.4.1 Modalités des expériences

MCT dans NetSolve 2.0

En conséquence des résultats du chapitre précédent, l'implantation de MCT dans NetSolve a évolué par des changements dans les mécanismes de correction des charges des serveurs. Son comportement s'apparente maintenant davantage à MP lorsque l'agent attend les nouveaux rapports de charge. En conséquence, lorsqu'il a été besoin de faire de nouveau des expériences ordonnancées par MCT, nous avons utilisé l'environnement dans sa version 1.4.1.

Problèmes, tâches, scénarii et plate–forme

Nous ne considérons ici qu'un seul type de problème : *wastecpu*. La majorité des expériences qui ont été réalisées dans ce chapitre sont fondées sur les mêmes scénarii présentés dans le chapitre précédent. Nous en présentons un résumé dans le tableau 6.2. Les serveurs utilisés pour l'ensemble des expériences suivantes sont spinnaker, artimon, fonck et soyotte. La modélisation de la plate–forme est donc la même que pour les expériences précédentes. Comme le type de problèmes est *wastecpu*, le coefficient d'hétérogénéité est de 8.9.

Les six types de scénarii étudiés ici sont numérotés de (b') à (k') en référence à ceux que nous avons déjà rencontrés dans le chapitre précédent et résumés dans le tableau 5.4. Deux scénarii dotés de la même lettre comportent les mêmes caractéristiques : types d'applications soumises, nombre total de tâches ou encore fréquence de soumission. De plus, ils sont générés en utilisant les mêmes méthodes. En revanche, les nombres d'expériences, graines et runs, diffèrent.

Le scénario (k') est propre à ce chapitre. Il s'articule sur la soumission de deux applications 1D-mesh en parallèle à des tâches indépendantes dont la fréquence moyenne d'inter–arrivée est $\mu = 25$ secondes. Ce scénario arrive en réponse à la question des performances entre les heuristiques sur un environnement où la demande de travail, comportant des relations de précédence, est peu importante mais continue.

Scénario	Application(s)		Tâches indépendantes		Expérience	
	nbclients	largeur x hauteur	nbtâches	μ (sec)	nbgraines x nbrun	n total
(b') 500 tâches indép. wastecpu	-	-	500	20, 17 et 15	3 x 2	500
(e') 1D-mesh + 250 tâches indépendantes	5	1x50	250	20	3 x 2	500
(f') stencil (tâches de taille 1)	1	10x50	-	-	1 x 1	500
(i') stencil + 86 tâches indépendantes	1	10x25	86	40	4 x 2	424
(j') stencil + 86 tâches indépendantes	1	5x25	86	25	3 x 3	211
(k') 1D-mesh	2	1x50	253	25	4 x 3	353

TAB. 6.2 – Scenarii, modalités et nombre d'expériences

Remarque

Deux runs relatifs à la même expérience sont généralement différents, y compris pour les expériences n'impliquant que des tâches indépendantes. Ceci s'explique naturellement par des dates de terminaison des tâches propres à chaque run et prises en compte dans les calculs des heuristiques. Les expériences ont donc été soumises un certain nombre de fois de sorte que la moyenne des résultats observés n'évolue plus.

6.4.2 Acuité du HTM synchronisé avec la réalité

Génération des résultats

Le protocole d'expérimentation diffère de celui utilisé dans le chapitre précédent. Grâce au numéro d'identification global, les données concernant chaque tâche peuvent être enregistrées dans le fichier de logs de l'agent lors des appels aux différentes fonctions. Ainsi, seul ce fichier est parcouru pour extraire les informations utilisées dans les résultats présentés ici.

Présentation des résultats

Nous avons observé la précision relative des estimations du HTM au cours des multiples expériences présentées ci-avant. Les résultats sont exposés sous deux formes : un tableau donnant l'erreur moyenne et la variance pour chaque scénario. Ils permettent de voir avec quelle exactitude la stratégie de placement des heuristiques a été correctement appliquée ; des graphes représentatifs de l'évolution des estimations au cours du temps pour quelques runs, dont la lecture est expliquée à la section 6.2.1.

Etude de la précision du HTM

Les résultats obtenus sur la précision révèlent une nette amélioration des prédictions. La figure 6.9 montre un run du scénario (b') avec $\mu = 17$ secondes, ordonnancé par HMCT sur spinnaker et artimon. Le HTM donnait déjà de bons résultats sur ce type de scénarii. Maintenant, l'erreur est inférieure à 5% (contre 15% pour le même débit de requête auparavant). On observe que les erreurs apparaissent quand plus d'une dizaine de tâches interfèrent avec celle observée, soit deux fois plus qu'avant. On remarque aussi de très bons résultats sur les erreurs moyennes pour $\mu = 15$ secondes pour les heuristiques MP, MSF et ML, qui seules arrivent à placer l'ensemble des tâches, avec par exemple pour ML 3% d'erreur maximum pour un écart–type de 2.

La figure 6.10 présente les prédictions du HTM sur un run du scénario (f') où un stencil est soumis à l'environnement utilisant MSF. Les relations de dépendances n'impliquent

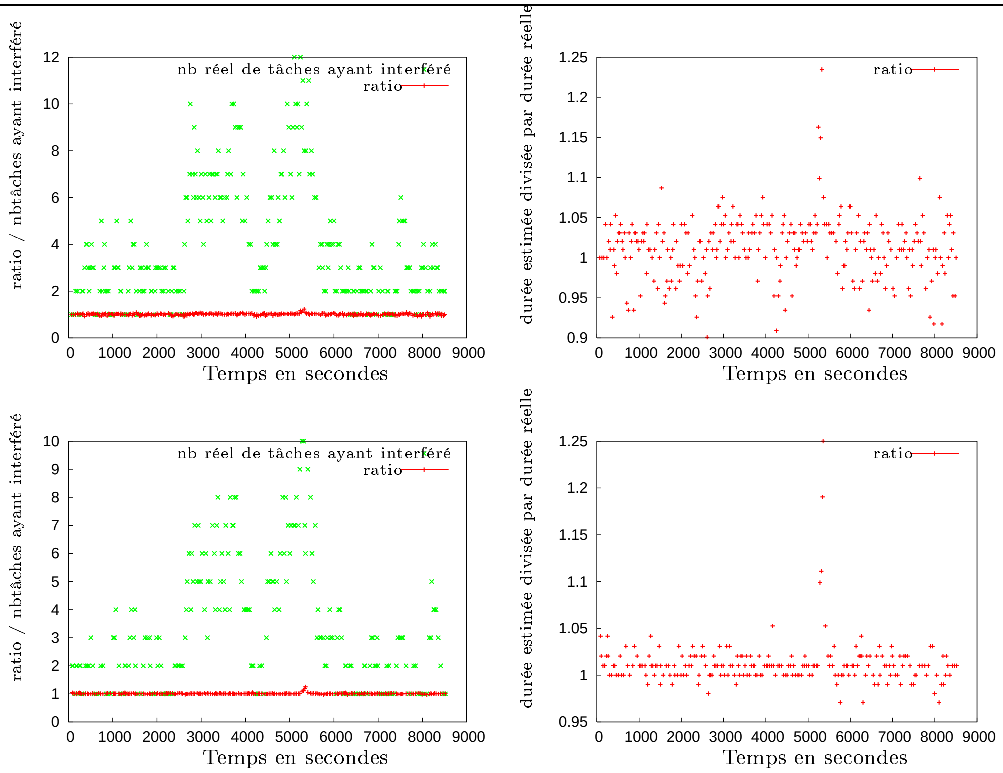


FIG. 6.9 – Scénario (b'), $\mu = 17$ sec : précision des estimations des durées des tâches exécutées par HMCT sur spinnaker en haut, sur artimon en bas

plus de risque de divergence des résultats. On voit très nettement que les tâches sont soumises par paquets d'où la présence de nombreuses interférences. Pourtant, la qualité des estimations est constante et l'erreur est inférieure à 3.4% en moyenne avec un écart–type de 3.4, comme le reporte le tableau 6.3.

Finalement, nous observons sur la figure 6.11 l'influence du nombre de tâches ayant interféré dans l'acuité des prédictions sur un run du scénario (e') ordonnancé par HMCT. Ce run est l'un des seuls à être complet sur toutes les expériences de ce scénario, ce qui explique qu'aucune erreur moyenne n'est donnée pour HMCT dans le tableau 6.3. Nous pouvons remarquer que passé un nombre d'interférences compris entre 10 et 12, les deux courbes suivent la même évolution. Les erreurs commencent alors à devenir importantes mais dès que le nombre d'interférences diminue et redevient inférieur au seuil, l'erreur redescend en dessous de 5%. Sur ce run, l'erreur moyenne relevée est seulement de 5.5% avec un écart–type de 5.

Comme on le constate sur le tableau, l'erreur n'est plus fonction ni de la fréquence d'arrivée, ni de l'heuristique pour les tâches indépendantes. En ce qui concerne les scénarii comportant des soumissions de DAGS, la stratégie de placement de MP et ML est à priori plus équitable pour les serveurs puisqu'elle permet au HTM d'assurer une prédiction des durées dont la précision moyenne est supérieure à 95%.

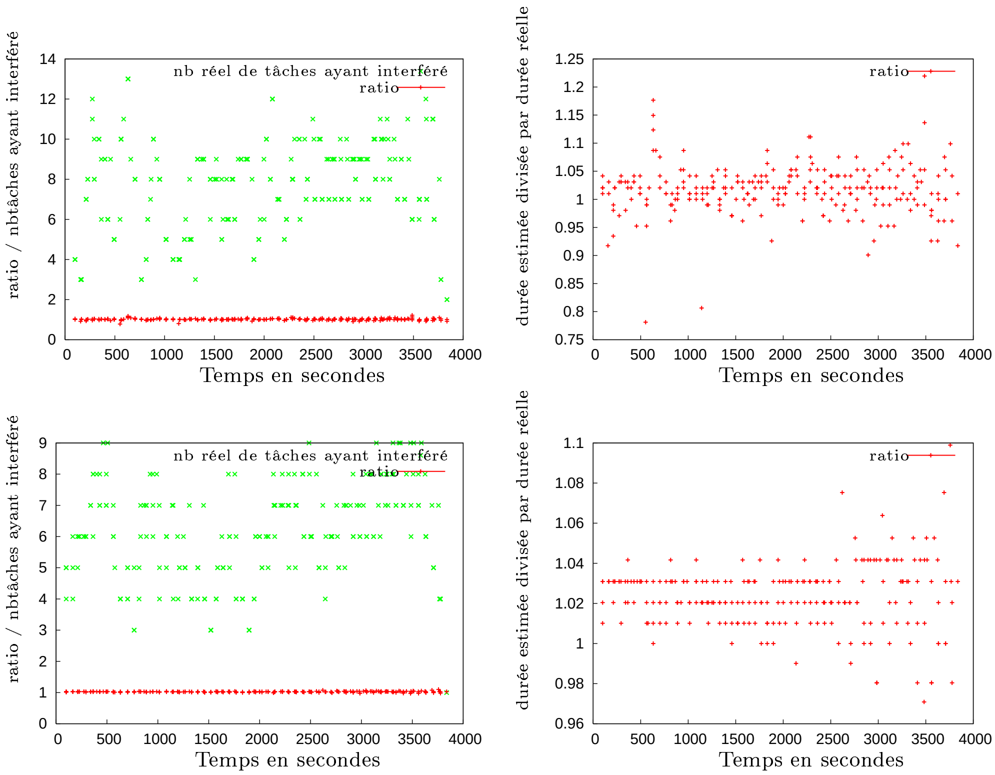


Fig. 6.10 – Scénario (f') : précision des estimations des durées des tâches exécutées par MSF sur spinnaker en haut, sur artimon en bas

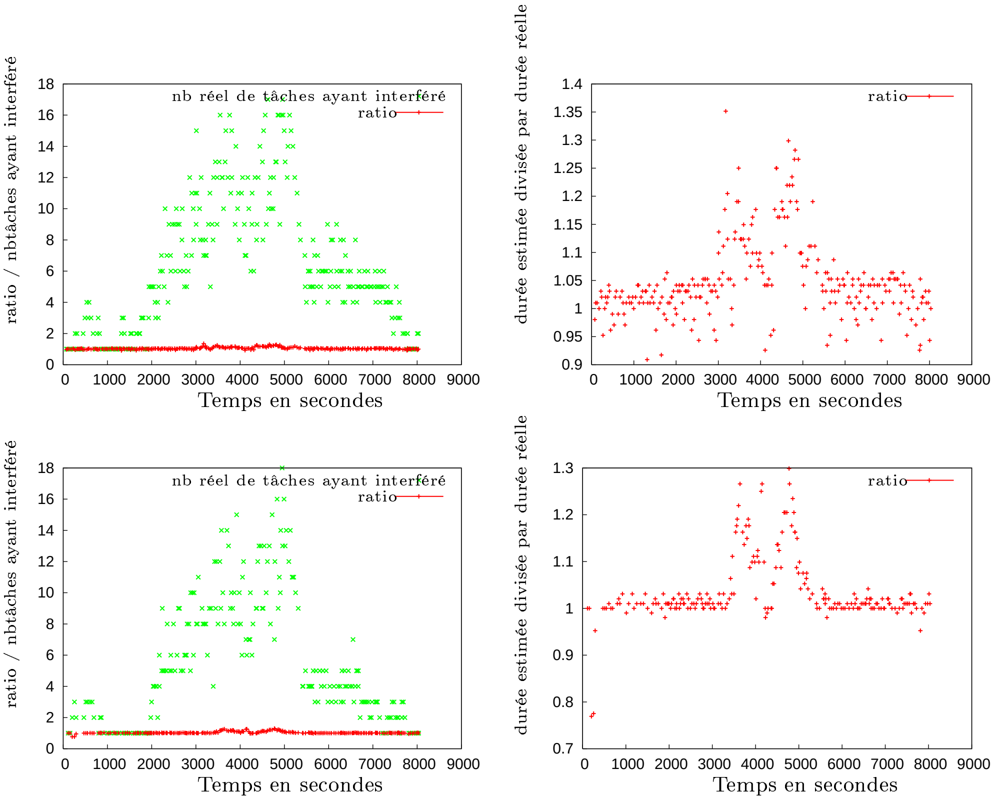


FIG. 6.11 – Scénario (e'), $\mu = 20$ sec : précision des estimations des durées des tâches exécutées par MSF sur spinnaker en haut, sur artimon en bas

	Scénario (b')												
	$\mu = 20$ secondes					$\mu = 17$ secondes					$\mu = 15$ secondes		
	HMCT	MP	MSF	AHMCT	ML	HMCT	MP	MSF	AHMCT	ML	MP	MSF	ML
spinnaker	**3.18** (3.1)	**2.08** (2.5)	**2.96** (3.4)	**2.72** (2.8)	**1.56** (2.3)	**2.83** (2.2)	**2.07** (1.9)	**2.85** (2.2)	**2.77** (2.1)	**2.66** (2.1)	**2.76** (2.18)	**6.50** (6.02)	**3.06** (2.01)
artimon	**1.67** (1.1)	**1.69** (1.4)	**1.66** (1.1)	**1.67** (1.3)	**1.04** (1.06)	**1.26** (1.3)	**1.16** (0.9)	**1.54** (2.2)	**1.36** (1.6)	**1.16** (0.9)	**0.98** (0.80)	**2.68** (4.20)	**1.13** (1.04)
soyotte	-	**0.12** (0.1)	-	-	-	- 0	**0.24** (0.5)	-	-	**0.97** (0)	**0.34** (0.49)	**0.36** (0.46)	**0.42** (0.35)
fonck	-	**0.13** (0.1)	-	-	**0.39** (1.1)	-	**0.19** (0.2)	-	-	**0.61** (0.8)	**0.39** (0.49)	**0.63** (0.78)	**0.36** (0.48)

	Scénario (e')			Scénario (f')				
	MP	MSF	ML	HMCT	MP	MSF	AHMCT	ML
spinnaker	**3.8** (4.5)	**6.82** (6.3)	**4.12** (4.0)	**2.79** (3.2)	**1.38** (1.1)	**3.40** (3.4)	**2.76** (4.1)	**3.17** (3.1)
artimon	**2.12** (3.6)	**3.07** (4.5)	**1.75** (2.6)	**2.66** (2.5)	**2.77** (0.8)	**2.44** (1.3)	**2.28** (2.2)	**2.48** (1.0)
fonck	**1.12** (1.2)	**0.40** (0.5)	**0.50** (0.6)	–	**0.35** (0.5)	**0.15** (0.1)	–	**0.23** (0.2)
soyotte	**0.87** (0.8)	**0.58** (0.9)	**0.32** (0.4)	–	**0.22** (0.2)	**0.35** (0.4)	–	**0.26** (0.4)

	Scénario (i')			Scénario (k')				
	MP	MSF	ML	HMCT	MP	MSF	AHMCT	ML
spinnaker	**5.61** (6.0)	**11.12** (8.3)	**4.23** (3.7)	**2.56** (3.0)	**1.85** (2.0)	**2.38** (2.4)	**2.60** (2.7)	**2.12** (2.1)
artimon	**3.17** (4.4)	**8.12** (8.2)	**2.14** (2.0)	**1.66** (1.2)	**1.61** (1.0)	**1.67** (1.0)	**1.60** (1.0)	**1.63** (1.0)
fonck	**0.5** (0.6)	**0.12** (0.1)	**0.15** (0.1)	–	**0.31** (0.3)	–	–	**0.19** (0.1)
soyotte	**0.57** (0.9)	**0.20** (0.2)	**0.22** (0.2)	–	**0.26** (0.3)	–	–	**0.44** (0.5)

TAB. 6.3 – Pourcentage d'erreur en moyenne selon le scénario et l'heuristique

6.4.3 Etude des performances des heuristiques

Les informations délivrées aux heuristiques par le module de prédiction de performances sont précises à plus de 95% en moyenne. Les résultats que nous obtenons sont donc *a priori* réprésentatifs de la stratégie de placement de chacune des heuristiques.

Présentation des résultats

Nous présentons les performances des heuristiques HMCT, MP, MSF, AHMCT et ML en comparaison aux performances que donne MCT. L'ensemble des résultats est axé sur quatre figures où sont respectivement donnés pour chaque heuristique et selon le scénario :

- les gains moyens sur le makespan *pour chaque application* dans la figure 6.12,
- les gains moyens réalisés sur le sumflow *pour chaque application* dans la figure 6.13,
- les gains moyens réalisés sur le sumflow pour l'ensemble des tâches indépendantes, qui sont par définition les mêmes que ceux observés sur le temps de réponse *de chaque tâche*. Il sont présentés dans la figure 6.14,
- le pourcentage moyen de tâches indépendantes terminant plus tôt lorsque le scénario est ordonnancé par l'une de nos heuristiques par rapport à MCT dans la figure 6.15.

Remarques.

- Pour des raisons de facilité de lecture et de comparaisons inter–figure, chaque scénario apparaît sur chaque figure même lorsqu'il n'est pas concerné par le critère étudié (par exemple le scénario (b') dans la figure des résultats sur le makespan).
- Malgré le nombre d'essais entrepris, la politique de placement de certaines heuristiques, trop axée sur les serveurs les plus rapides, n'a pu donner satisfaction à l'ensemble des requêtes. Ainsi HMCT et AHMCT n'ont pas réussi à ordonnancer les scénarii (b') pour $\mu = 15$ secondes, (e') et (i'). De plus, AHMCT n'a pas pu terminer le scénario (j').

Performances des heuristiques

Makespan. La figure 6.12 montre que HMCT, MSF, AHMCT et ML donnent toujours de meilleurs résultats que MCT sur le makespan, avec une date de terminaison de chaque application jusqu'à 22% plus tôt. En revanche on constate avec les scénarii (f') et (j') que MP peut avoir des difficultés pour ordonnancer une application à cause de l'affectation de tâches critiques sur un serveur lent. On remarque encore que la stratégie de placement de MP, MSF et ML assure la faisabilité de la totalité des expériences et que MSF et ML donnent en moyenne de meilleurs résultats que HMCT et AHMCT.

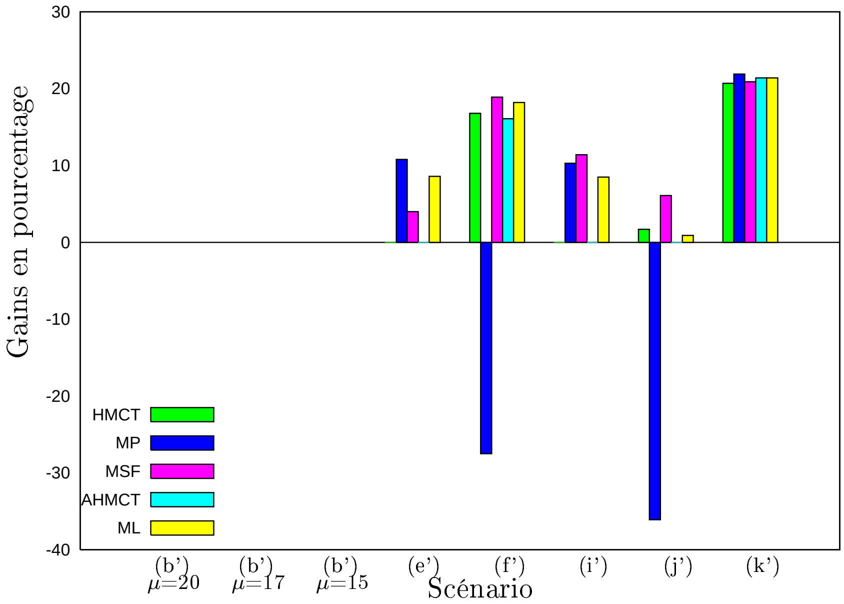


FIG. 6.12 – Gains moyens réalisés *par chaque client* sur le makespan de son application selon l'heuristique et le scénario

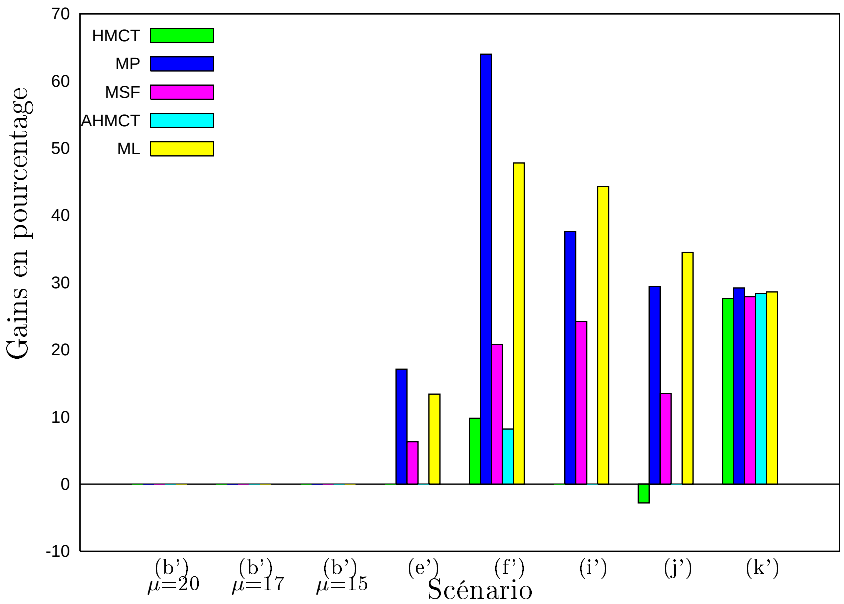


FIG. 6.13 – Gains moyens réalisés sur le sumflow *pour chaque client* selon l'heuristique et le scénario

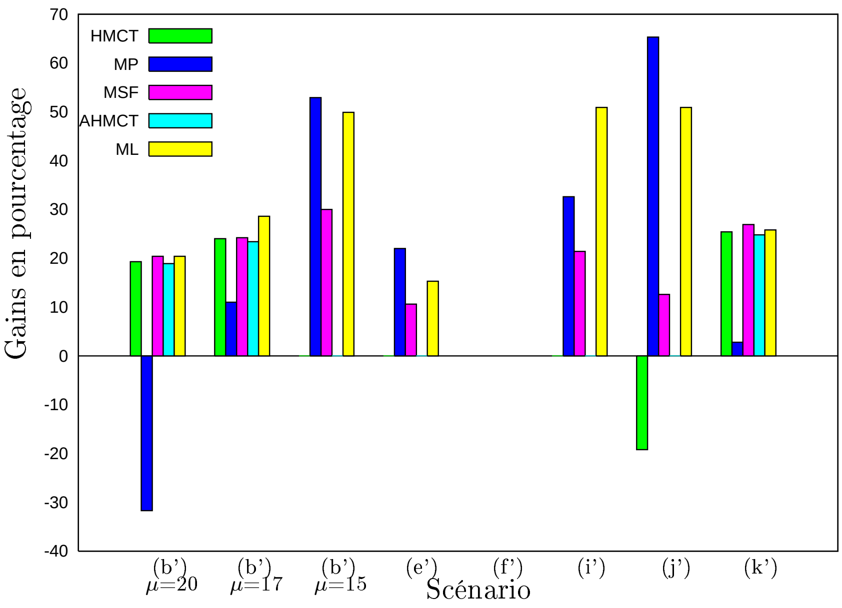


FIG. 6.14 – Gains moyens réalisés sur le temps de réponse *de chaque tâche* selon l'heuristique et le scénario

Sumflow. La figure 6.13 indique les gains moyens obtenus en terme de coût total en temps pour chaque application. On constate qu'en dehors du scénario (j') où HMCT donne des performances négatives, toutes nos heuristiques réduisent le sumflow. MP atteint même une économie de 64% pour le scénario (f') et ML juste derrière avec 45%. On remarque aussi la constance des très bons résultats de ML tout au long des scénarii.

Temps de réponse moyen. De façon générale, nos heuristiques améliorent le temps de réponse moyen (et donc le coût en temps que requiert l'exécution de l'ensemble des tâches indépendantes) que demandent les tâches ordonnancées par MCT. On voit que la stratégie de placement de MP paye dès que la fréquence de soumission est suffisamment élevée pour dégager de nombreuses interférences : le seul scénario où MP obtient de moins bons résultats que MCT est (b') pour $\mu = 20$ secondes. MSF et ML donnent les mêmes performances sur les scénarii (b') avec $\mu = 20$ et $\mu = 17$ secondes et (k'), mais ML permet des gains supérieurs sur les autres avec jusqu'à 4 fois plus d'économie en temps de calcul sur l'environnement pour le scénario (j').

Pourcentage de tâches terminant plus tôt. Le graphe de la figure 6.15 représente pour chacune de nos heuristiques et pour chaque scénario, le pourcentage moyen de tâches qui ont terminé plus tôt que si placées par MCT. Lors du calcul du pourcentage, nous avons considéré que deux tâches terminaient *à la même date* si la différence entre les valeurs étaient inférieure à deux secondes. Nous avons donc aussi représenté le pourcentage de tâches qui terminait plus tôt avec MCT. Un gain apparaît dès que la surface du rectangle

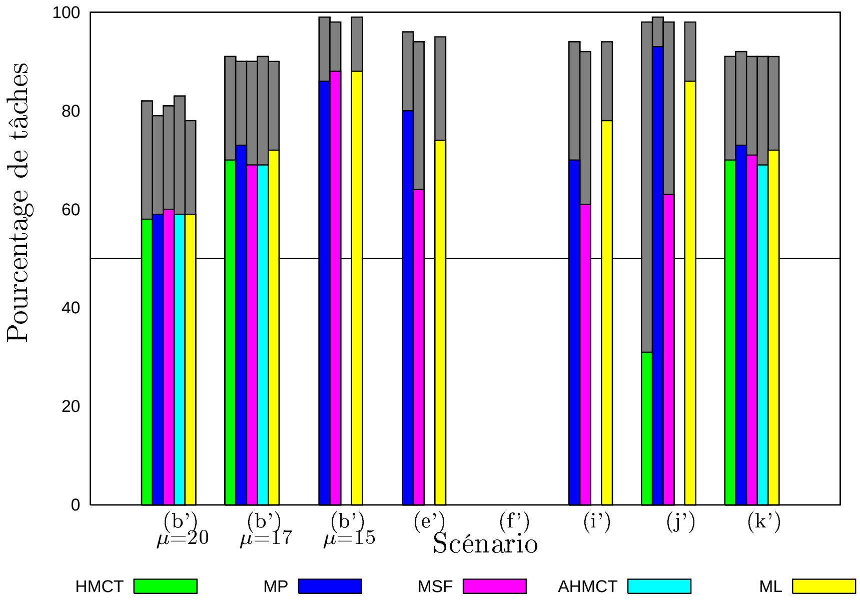


FIG. 6.15 – Pourcentage de tâches terminant plus tôt et aussi tôt qu'avec MCT, selon l'heuristique et le scénario

de l'heuristique considérée est plus importante que celle du rectangle supérieur associé. Hormis pour HMCT au cours du scénario (j'), on s'aperçoit qu'une tâche a plus de chance de finir tôt si l'environnement de calcul utilise l'une de nos heuristiques quel que soit le scénario considéré. MP, MSF et ML établissent les meilleurs résultats, avec par exemple le scénario (j') où MP atteint 93% et ML 86%.

Bilan Nos heuristiques peuvent prendre en compte plusieurs critères afin d'optimiser les métriques observées : elles assurent la pérennité de leurs choix en injectant les choix précédents dans le calcul des choix futurs. Nous avons vu que les politiques de placement visant principalement les serveurs les plus rapides comme HMCT et AHMCT ne sont pas les meilleures en pratique : elles ne sont pas adaptées à des plate–formes hétérogènes car les serveurs les plus lents sont peu voire jamais sollicités, et les plus rapides risquent à terme de refuser du travail. De même, les bonnes performances de MP sont à nuancer : le gain constaté sur le sumflow du stencil des scénarii (f') et (j') est en relation avec la soumission de ses tâches espacée dans le temps (phénomène «tâche critique»), il y a donc moins d'interférences. Ainsi, MP dispose de plus de ressources pour les tâches indépendantes pendant les runs du scénario (j'). Cependant, ses performances sur le reste des scénarii, en particulier le scénario (i') où la soumission est aussi celle d'un stencil, montre qu'elle reste envisageable sur une plate–forme de production. MSF présente de bons résultats quelle que soit la métrique observée et le scénario considéré. ML donne dans l'ensemble les meilleurs résultats des heuristiques que nous avons proposées. De plus, grâce à sa stratégie de placement, le HTM est capable de fournir continuellement des estimations de qualité.

Mécanismes de synchronisation et performance. L'utilisation d'informations plus précises assure des stratégies appliquées plus justement. On ne peut cependant pas affirmer que toutes les heuristiques donnent en conséquence de meilleurs résultats. En effet, de nombreux runs apparaissent où des serveurs refusent des tâches placées par HMCT. Dans l'ensemble, MSF donne de moins bons résultats sur les métriques observées. Par contre, les performances de MP sont sensiblement les mêmes.

Les estimations des durées des tâches faites par le HTM avant l'implantation des mécanismes de synchronisation sont généralement supérieures à celles obtenues après. Avant, les retards étaient plus surévalués et les serveurs les plus rapides moins utilisés que ce qu'ils ne devaient l'être. Ce comportement, qui tend à baisser quantitativement les interférences, semble donner de meilleurs résultats pour les heuristiques individuelles comme HMCT et AHMCT. Notons que la variation des performances de MP ne permet pas de statuer sur l'apport des mécanismes : nous avions vu qu'elle disposait déjà d'informations précises car sa stratégie de placement permet de répartir plus équitablement la charge. En conséquence, l'emploi du modèle temps partagé pour modéliser la plate–forme était relativement correct.

6.5 Conclusion

Ce chapitre traite de l'étude pratique de la faisabilité du déploiement des heuristiques et du HTM qui ont été décrits dans le chapitre précédent. Dans un premier temps, nous avons étudié la précision des estimations calculées par le HTM lors des expériences réalisées précédemment. Nous avons cerné des cas où le HTM était susceptible de se désynchroniser des exécutions réellement effectuées sur la plate–forme.

En conséquence, nous avons proposé des mécanismes de synchronisation que nous avons validés par leur implantation dans l'environnement de résolution de problèmes NetSolve et par de nouvelles expériences. Elles reprennent certains scénarii de la série d'expériences sur plate–forme réelle présentées dans le chapitre précédent.

L'usage du HTM procure des informations pertinentes sur l'état du système et confère la possibilité d'employer des heuristiques capables de chiffrer les retards occasionnés par l'affectation d'une requête sur le système. Nous avons vu que les prédictions des durées sont maintenant établies avec une précision supérieure à 95% avec un faible écart–type et que les heuristiques MP et ML, par leur politique de placement, sont plus à même de maintenir une bonne précision au cours du temps.

HMCT et AHMCT donnent de meilleurs résultats que MCT lorsqu'aucun serveur ne rejette de tâches. MP donne des résultats très semblables à ceux qui ont pus être observés dans le chapitre précédent. Elle souffre de ce que nous avons appelé l'effet «tâche critique» lorsque les soumissions comportent des relations de dépendance. MSF donne de bonnes performances et améliore celles obtenues par MCT quel que soit l'expérience considérée. ML donne les meilleurs résultats tant sur ses performances par rapport à MCT et aux autres heuristiques que sur sa constance à optimiser l'ensemble des critères que nous voulions. Elle dispose de plus d'informations de qualité qu'elle aide à maintenir par sa stratégie de placement qui génère moins d'interférences. Des heuristiques étudiées ici, elle semble donc la mieux adaptée à un environnement de soumission de type GridRPC.

Chapitre 7

Conclusion

7.1 Conclusion

L'approche GridRPC et son modèle de programmation simple est un moyen pratique d'exploitation des ressources hétérogènes et distribuées d'une grille de calcul. Les environnement de calculs que l'on désigne par NES (*Network Enable Server*) ou plus récemment par Grid ASP (*Grid Application Service Provider*, fournisseur de services applicatifs pour la grille) reposent sur l'exécution de tâches à distance et sur l'architecture client–agent–serveur. Les clients n'ont ainsi pas à se soucier de la recherche et de la surveillance des ressources où déporter leurs calculs, de maîtriser des outils parfois complexes de prédiction de performances, ni de réfléchir à comment adapter leurs applications à une grille dont la performance peut subitement changer. Le temps d'implantation s'en trouve aussi considérablement réduit. Un NES permet à ses utilisateurs de profiter du parallélisme de tâches qu'il est possible d'obtenir en décomposant une application selon son graphe structurel. Chaque tâche la composant devient ainsi une requête soumise à l'agent (ou à l'un des agents) de l'environnement. Le problème d'assurer de bonnes performances incombe à l'intergiciel. Il doit utiliser les services qu'il a à sa disposition pour recueillir des informations sur la plate–forme et répartir les tâches de calcul dans le but de donner satisfaction au plus grand nombre de clients.

Dans cette optique, la qualité de l'ordonnancement est cruciale. Pourtant, cet ordonnancement, qui s'effectue au sein de l'agent, n'est pas simple puisque ce dernier ne dispose que de peu d'informations sur la plate–forme. Il reçoit des rapports des senseurs distribués sur les ressources qu'il souhaite surveiller et la date de soumission d'une requête n'est pas connue à l'avance. Sa décision d'affectation doit être immédiate, aussitôt le problème demandé reconnu et les ressources qui proposent sa résolution sélectionnées.

Notre travail prend place dans le mode d'exécution temps–partagé des ressources de calcul. En effet, dans de nombreux NES, dès que la totalité des données nécessaires à un calcul est transmise au serveur de calcul, l'exécution de la tâche de calcul commence. Nous avons proposés dans cette thèse de considérer d'autres critères de sélection que celui du temps de terminaison de la requête qui est l'un des plus employés, notamment avec l'implantation d'une variante de MCT (*Minimum Completion Time* [MAS+99]) dans les NES parmi les plus connus comme NetSolve [CD96], Ninf [NSS99] ou DIET [CDF+02]. Nous avons proposé et décrit les heuristiques d'ordonnancement dynamique HMCT (*Historical MCT*), AHMCT (*Advanced HMCT*), MP (*Minimum Perturbation*), MSF (*Minimum Sumflow*) et ML (*Minimum Length*). Chacune d'elles considère dans ses calculs la perturbation que l'affectation de la nouvelle requête fera subir à celles déjà allouées sur la plate–forme. Ceci permet d'insuffler de la mémoire dans le mécanisme de décision : les décisions précédentes ne sont plus obsolètes après l'affectation de la nouvelle tâche de calcul.

Alors que HMCT et AHMCT poursuivent le même objectif que MCT, les heuristiques MP, MSF et ML ont été conçues dans le but de donner satisfaction au plus grand nombre d'utilisateurs. Nous avons étudié nos heuristiques d'ordonnancement en les comparant à MCT sur de nombreux critères dont le makespan, le sumflow, le temps de réponse moyen et le pourcentage de tâches terminant plus tôt. Ce dernier critère permet de dénombrer la «chance» que possède une tâche de terminer plus tôt si l'environnement de résolution de problèmes utilise l'une de nos heuristiques plutôt que MCT.

Nos heuristiques nécessitent des informations particulières, ce qui nous a amené à développer notre propre module de prédiction de performances, le gestionnaire de l'historique des tâches ou HTM (*Historical Trace Manager*). Il est non–intrusif et est exécuté sur le même hôte que l'agent. Les calculs qu'il doit effectuer nécessitent très peu de temps au regard des évènements et des délais de la plate–forme. Il enregistre les décisions d'ordonnancement au fur et à mesure et est capable d'estimer la date de terminaison de *chacune des tâches* dans le système. Pour cela, il simule l'exécution des tâches sur les ressources en utilisant le modèle temps–partagé. Il délivre ses informations au module d'ordonnancement sous la forme de diagrammes de Gantt.

Le HTM ainsi que toutes les heuristiques ont été implantés dans l'agent de NetSolve. De très nombreuses expériences, correspondant à près de quatre mois de calculs ininterrompus, ont été conduites au cours desquelles la qualité des prédictions du HTM et les performances des heuristiques ont été étudiées.

Tout d'abord, le HTM a montré un taux d'erreur acceptable qui pouvait se dégrader. Nous avons expliqué que la cause principale venait des interférences qu'une tâche peut subir et des relations de précédence qui faussaient la modélisation. Nous avons donc proposé et implanté un mécanisme de synchronisation du HTM aux exécutions de la plate–forme. Il est fondé sur l'envoi d'un message de terminaison lorsque la tâche finit son exécution. Les expériences qui ont ensuite été conduites valident l'intérêt de la démarche. En effet, nous obtenons maintenant une précision supérieure à 95% en moyenne avec un faible écart–type.

Les heuristiques montrent qu'il est possible d'obtenir un gain dès que l'on considère la charge interne de la plate–forme dans les décisions d'ordonnancement. HMCT et AHMCT confirment les résultats de [Sti85] qui veulent que les stratégies d'optimisation individuelle chargent les serveurs les plus rapides. De fait, donner le plus de puissance à chaque nouvelle requête ne semble pas une bonne stratégie si les ressources sont très hétérogènes et pire encore, si elles sont de plus peu nombreuses, le risque est d'avoir alors une incohérence entre ce qui est supposé (l'affectation de la tâche à un serveur donné) et ce qui est fait (la tâche a été affectée à un autre à cause d'un refus dû à une charge trop élevée). En revanche, MP, MSF et ML obtiennent de bonnes performances. L'emploi de MP est toutefois à mettre en relation avec la nature des soumissions, ce qui réduit son intérêt pour un environnement dans le cas général. MSF et ML apportent un gain sur chacune des métriques considérées quelle que soit l'expérience considérée et une tâche a plus de chance de terminer tôt si le NES emploie l'une d'elles.

Il ressort de nos études que l'emploi de l'heuristique ML semble le plus approprié à une plate–forme de métacomputing. Elle correspond à une réponse possible à la problématique de donner une heuristique capable d'optimiser plusieurs critères à la fois afin de satisfaire le plus grand nombre d'utilisateurs de la grille de calcul. De toutes les heuristiques testées, elle donne le plus souvent l'un des meilleurs résultats et ses choix permettent au HTM de maintenir un taux d'erreur particulièrement bas dans ses estimations.

Nos travaux ont donné lieu à des publications. Ainsi, l'étude préliminaire des heuristiques HMCT, MP et MSF sur des expériences de simulation a été publiée dans les actes de *Renpar*'02. Les résultats plus complets ont été en partie publiés dans les actes de l'école *Grid'02* [Can02] et font l'objet d'une publication dans les actes de la conférence *QOS and Dynamic System workshop*, QDS'04 [CJ04a]. La section 5.5.3, présentant la validation des heuristiques lors d'expériences réelles sur la soumission de tâches indépendantes, fait l'objet d'une publication dans les actes de la conférence *Heterogeneous Computing Workshop*'03 [CJ03]. Une partie des résultats des sections 5.5.4 et 6.2 concernant des scénarii d'expériences plus complexes (impliquant aussi des graphes de tâches) a été publiée dans les actes de la conférence *Euro–Par'*04 [CJ04b].

7.2 Perspectives et travaux futurs

De nombreuses poursuites sont possibles à partir de ce travail de thèse, dont on peut concevoir au moins deux directions. En effet, le HTM peut tout d'abord accueillir quelques améliorations. L'ordonnancement de tâches sur un environnement de calcul peut lui aussi être étendu à des tâches parallèles par exemple.

7.2.1 HTM

Certains systèmes d'exploitation sont capables de fournir pendant l'exécution d'une tâche, le temps durant lequel elle a eu accès au processeur depuis qu'elle est lancée. Un mécanisme implanté sur le serveur pourrait ainsi enregistrer la durée demandée par la tâche sur le serveur non chargé et la communiquer au HTM. Ainsi, le HTM disposerait d'une information précise sur ce serveur et qui pourrait être utilisée de nouveau si la résolution d'un problème de même type est de nouveau demandée.

Il est aussi possible d'améliorer l'estimation de la durée d'une nouvelle requête sur un serveur non chargé. Actuellement, certains intergiciels utilisent le monôme de plus haut degré de la complexité. Cependant, si plusieurs tâches d'un même type de problème ont déjà été exécutées, nous disposons alors d'informations précises sur les durées à vide, que nous supposons toujours fonction des tailles du problème demandé. Il est alors possible d'interpoler et d'obtenir une durée *a priori* de plus en plus précise au fur et à mesure des soumissions.

L'une des perspectives à terme est d'amener le HTM à considérer les rapports de charges issus des moniteurs installés sur les ressources. En surveillant la valeur, le HTM pourrait différencier la charge interne de la charge externe de la plate–forme. S'il constate que la charge observée dépasse la charge interne, il pourrait par exemple simuler l'exécution de tâches sur le laps de temps où la valeur a été supérieure. Ainsi, l'ensemble des tâches déjà affectées est retardé et la puissance de la ressource n'est pas surévaluée par rapport à ce qu'elle peut vraiment exécuter. Le mécanisme de correction de charge devrait permettre de stabiliser et de diminuer l'erreur engendrée.

7.2.2 Ordonnancement

L'ordonnancement proposé peut être distribué sur un environnement construit sur une hiérarchie d'agents comme l'est DIET. Un HTM est alors exécuté sur chacune des feuilles de la hiérarchie et ne prend en considération que les serveurs qui y sont enregistrés. Les heuristiques peuvent ensuite être exécutées dans la hiérarchie d'agents.

Grâce à la qualité des ses informations, le HTM permet d'envisager des ordonnancements où chaque requête est soumise avec une *deadline*, c'est–à–dire une date buttoir avant laquelle la tâche doit être terminée. Il faut concevoir les stratégies de placement qui peuvent s'appuyer sur le travail présenté dans [TMCB01]. Notons que les heuristiques proposées ici conviennent à condition de définir un filtre qui permettra d'apprécier le(s) serveur(s) respectant l'ensemble des deadlines des tâches en cours d'exécution. De plus, notons qu'avec le HTM, des réservations sur les ressources sont aussi possibles. L'ordonnancement doit être établi de sorte qu'aucune exécution n'empiète sur la zone ainsi réservée. Inversement, une machine dont l'utilisateur s'enregistre comme étant absent pendant une plage horaire donnée peut être considérée comme dédiée à la plate–forme.

Une autre perspective consiste à étendre nos travaux à des routines parallèles afin d'offrir une approche de parallélisme mixte. La fonction appelée par un client peut effectivement être elle–même une application. L'intergiciel peut sélectionner la méthode qu'il doit utiliser et placer les tâches en conséquence sur les ressources. Par exemple, si l'environnement n'est pas capable de résoudre une multiplication de matrice parallèle par la résolution du problème `pdgemm` (qui par n'est pas installée par exemple), le produit est alors décomposé grâce à la méthode de Strassen comme dans [DS04]. L'ordonnanceur dispose ainsi du graphe de tâches de l'application et le HTM permet d'avoir des prédictions de qualité.

Une extension à cette perspective est de laisser au client la possibilité de donner le DAG et les appels composants son application. Par exemple, nous avons relevé un inconvénient de la stratégie de MP que nous avons appelé «effet de la tâche critique». Si l'agent connaît le graphe de dépendance, il connaît alors les tâches critiques au moment où elles le deviennent. Il peut alors prendre ses décisions en conséquence. On peut de plus espérer un gain notable de performance. Dans l'exemple d'une application stencil, le client fournit les données pour chaque tâche de tête et *a priori*, seuls les résultats des tâches de queue l'intéresse. Les données intermédiaires peuvent donc rester sur les serveurs ou être envoyées directement au serveur devant effectuer les calculs sans transiter par le client. L'ordonnanceur aura alors suffisamment d'informations pour anticiper la sélection des ressources de calcul et celles de stockages des données. Il pourra assurer la pérennité de ses décisions grâce encore au HTM. Remarquons que dans ce cas, les tâches pourront aussi être réordonnancées si nécessaire.

Enfin, le modèle temps–partagé peut être raffiné : tout d'abord, une tâche peut être exécutée sur un serveur avec une priorité différente des autres requêtes. Dans un souci de déploiement, aucun droit particulier ne doit être supposé, en particulier celui d'administrateur système. La priorité ne peut donc que décroître au cours du temps. Ceci induit des modifications dans le HTM et dans les politiques de placement puisque la priorité devient dynamique et doit être déterminée puis communiquée au serveur. Enfin, le prix des serveurs biprocesseurs et quadriprocesseurs a fortement baissé. Il serait intéressant d'étudier comment ces ressources peuvent être modélisées afin que le HTM puisse effectuer ses prédictions.

Annexe A

Algorithmes dynamiques batch de référence

Les algorithmes 12, 13, 14 et 15 sont tous de complexité $O(|S|r)$ où S est le sac de tâche à ordonnancer et r le nombre de serveurs de calcul de la plate–forme, puisqu'il s'agit d'effectuer un calcul par rapport à une métrique sur chaque machine (éventuellement comprise dans un cluster) de la plate–forme.

Algorithme 12: Heuristique d'ordonnancement Min–min

! **Données** : S un sac de tâches
Résultat : Un ordonnancement de S sur les ressources
début
  tant que $S \neq \emptyset$ **faire**
    pour chaque *tâche i de S* **faire**
      pour chaque *ressource de calcul j* **faire**
        Calcule $C_{i,j}$, la date de terminaison de la tâche i sur le serveur j;
      Calcule le serveur j_i où la tâche termine le plus tôt : $j_i = \min_j C_{i,j}$;
    Calcule la tâche i_0 qui termine le plus tôt : $i_0 = \min_i C_{i,j_i}$;
    Affecte la tâche i_0 au serveur j_{i_0};
    S ← S - $\{i_0\}$;
fin

Algorithme 13: Heuristique d'ordonnancement Max–min

! **Données** : S un sac de tâches
Résultat : Un ordonnancement de S sur les ressources
début
  tant que $S \neq \emptyset$ **faire**
    pour chaque *tâche i de S* **faire**
      pour chaque *ressource de calcul j* **faire**
        Calcule $C_{i,j}$, la date de terminaison de la tâche i sur le serveur j;
      Calcule le serveur j_i où la tâche termine le plus tôt : $j_i = \min_j C_{i,j}$;
    Prendre la tâche i_0 qui termine le plus tard : $i_0 = \max_i C_{i,j_i}$;
    Affecte la tâche i_0 au serveur j_{i_0};
    S ← S - $\{i_0\}$;
fin

L'algorithme 14 que nous présentons ici est celui utilisé dans [COBW00]. Il diffère de celui que l'on peut trouver dans [MAS+99], mais est plus simple et donne, d'après les auteurs, les mêmes résultats.

Algorithme 14: Heuristique d'ordonnancement Sufferage

! **Données** : S un sac de tâches
Résultat : Un ordonnancement de S sur les ressources
début
  tant que $S \neq \emptyset$ **faire**
    pour chaque *tâche i de S* **faire**
      pour chaque *ressource de calcul j* **faire**
        Calcule $C_{i,j}$, la date de terminaison de la tâche i sur le serveur j;
      Calcule le serveur $j_{1,i}$ où la tâche termine le plus tôt : $j_{1,i} = \min_j C_{i,j}$;
      Calcule le serveur $j_{2,i}$ où la tâche donne le deuxième meilleur temps de terminaison : $j_{2,i} = \min_{j \neq j_{1,i}}$;
      Calcule la souffrance de la tâche i : $sufferage_i = C_{i,j_{2,i}} - C_{i,j_{1,i}}$;
    Prendre la tâche i_0 qui devrait le plus souffrir : $i_0 = \max_i sufferage_i$;
    Affecte la tâche i_0 au serveur j_{1,i_0};
    S ← S - $\{i_0\}$;
fin

Algorithme 15: Heuristique d'ordonnancement Xsufferage

! **Données** : S un sac de tâches
Résultat : Un ordonnancement de S sur les ressources
début
  tant que $S \neq \emptyset$ **faire**
    pour chaque *tâche i de S* **faire**
      pour chaque *cluster j* **faire**
        Calcule $C_{i,j}$, la date de terminaison minimum de la tâche i sur le cluster j. Soit $k_{i,j}$ le numéro du serveur de ce cluster où elle est obtenue.;
      Calcule le cluster $j_{1,i}$ où la tâche termine le plus tôt : $j_{1,i} = \min_j C_{i,j}$;
      Calcule le cluster $j_{2,i}$ où la tâche donne le deuxième meilleur temps de terminaison : $j_{2,i} = \min_{j \neq j_{1,i}}$;
      Calcule la souffrance de la tâche i : $sufferage_i = C_{i,j_{2,i}} - C_{i,j_{1,i}}$;
    Prendre la tâche i_0 qui devrait le plus souffrir : $i_0 = \max_i sufferage_i$;
    Affecte la tâche i_0 au serveur $k_{i_0,j_{1,i_0}}$;
    S $\leftarrow$ S - $\{i_0\}$;
fin

Bibliographie

[ABG$^+$00] G. Allen, W. Benger, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. The cactus code : A problem solving environment for the grid. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, page p. 253, Pittsburgh, Pennsylvania, August 01 – 04 2000. Available at `http://csdl.computer.org/dl/proceedings/hpdc/2000/0783/00/07830253.pdf`.

[ACD02] D. Arnold, H. Casanova, and J. Dongarra. Innovations of the netsolve grid computing system. In *Concurrency and Computation : Practice and Experience*, volume 14, pages 1457–1479, 2002.

[ADKL01] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. In *SIAM Journal on Matrix Analysis and Applications*, volume 23(1), pages 15–41, 2001.

[Aiv02] T. Aivazian. Linux kernel 2.4 internals, August 2002. Available at `http://www.moses.uklinux.net/patches/lki.html`.

[All75] J. Allman. Np–complete scheduling problems. In *Journal of Computer and System Sciences*, volume 10 :434–439, 1975.

[AMT$^+$98] K. Aida, S. Matsuoka, A. Takefusa, U. Nagashimaand, and H. Nakada. A performance evaluation model for effective job scheduling in global computing systems. In *The Seventh IEEE International Symposium on High Performance Distributed Computing*, page 352, Chicago, Illinois, July 28 – 31 1998. Available at `http://csdl.computer.org/comp/proceedings/hpdc/1998/8579/00/85790352abs%.htm`.

[AMT$^+$00] K. Aida, S. Matsuoka, A. Takefusa, S. Sekiguchi, H. Nakada, and U. Nagashima. A performance evaluation model for scheduling in global computing systems. In *NASA Workshop on Performance-Engineered Information Systems*, 2000. Available at `http://ninf.apgrid.org/papers/nasa98aida/slide.ppt`.

[ASGH95] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod : A tool for performing parametised simulations using distributed workstations. In *4th IEEE Symposium on High Performance Distributed Computing, Virginia*, August 1995.

[ASWB95] C. Anglano, J. Schopf, R. Wolski, and F. Berman. Zoom : A hierarchical representation for heterogeneous applications. Technical Report CS95-451, UCSD CS, 1995.

[Atl01] S. Atlan. Mémoire de dea : Monitoring d'une plate–forme de métacomputing, 2001.

[BAG00] R. Buyya, D. Abramson, and J. Giddy. An economy driven resource management architecture for global computational power grids. *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, june 2000. Available at `http://www.buyya.com/papers/ams2000.pdf`.

[Bak74] K. R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, 1974. ISNB 0-471-04555-1.

[BASM01] T. D. Braun, S. Ali, H. J. Siegel, and A. A. Maciejewski. Using the min-min heuristic to map tasks onto heterogeneous high-performance computing systems. In *Proceedings of the 2nd Annual Los Alamos Computer Science Institute (LACSI)Symposium*, Sante Fe, NM, USA, Oct. 15–18 2001.

[BBR01] O. Beaumont, V. Boudet, and Y. Robert. The iso-level scheduling heuristic for heterogeneous processors. Technical Report RR-2001-22, LIP, ENS Lyon, May 2001.

[BC00] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly, October 2000. ISBN 0-596-00002-2, Available at `http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html`.

[BC02] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, 2nd Edition.* O'Reilly, December 2002. ISBN 0-596-00213-0.

[BCM98] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *SODA : ACM-SIAM Symposium on Discrete Algorithms*, 1998. Available at `http://citeseer.ist.psu.edu/bender98flow.html`.

[BDH+94] J. C. Browne, J. Dongarra, S. I. Hyder, K. Moore, and P. Newton. Experiences with code and hence in visual programming for parallel computing. In *IEEE Parallel and Distributed Technology*, volume 3, No. 1, pages 75–83, Spring 1994.

[BGA00] R. Buyya, J. Giddy, and D. Abramson. An evaluation of economy-based resource trading and scheduling on computational power grids for parameter sweep applications. In Kluwer Academic Press, editor, *Workshop on Active Middleware Services (AMS 2000), (in conjuction with Ninth IEEE International Symposium on High Performance Distributed Computing) in Pittsburgh, USA*, August 2000.

[BSB+98] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao. A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed*

System, pages 330–335, 1998. Available at `http://citeseer.ist.psu.edu/braun98taxonomy.html`.

[BSB+99] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. H., and R. F. Freund. A comparison study of static mapping heuritsics for a class of meta-tasks on heterogeneous computing systems. In *Proceedings of the 8th Heterogeneous Computing Workshop (HCW '99)*, pages 15–29, april 1999. Available at `http://citeseer.ist.psu.edu/braun99comparison.html`.

[BW97] F. Berman and R. Wolski. The apples project : A status report. In *Proceedings of the 8th NEC Research Symposium*, may 1997. Available at `http://apples.uscd.edu`.

[BWF+96] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing'96*, 1996.

[BZ92] S. A. Banawan and N. M. Zeidat. A comparative study of load sharing in heterogeneous multicomputer systems. In *Proceedings of the 25th annual symposium on Simulation*, pages 22–31. IEEE Computer Society Press, 1992. ISBN 0-8186-2765-4.

[Can02] Y. Caniou. Ordonnancement pour le modèle temps partagé. In *Proceedings of WINTER SCHOOL Grid 2002*, Aussois, France, dec 2002.

[Cas01] H. Casanova. Simgrid : A toolkit for the simulation of application scheduling. In *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'01)*. IEEE Computer Society, may 2001. Available at `http://www-cse.ucsd.edu/~casanova/`.

[CD96] H. Casanova and J. Dongarra. Netsolve : A network server for solving computational science problems. In *Proceedings of Super-Computing -Pittsburg*, 1996.

[CD04] E. Caron and F. Desprez. Diet, tour d'horizon. In *GridUSe–2004, Ècole thématique sur la Globalisation des Ressources Informatiques et des Données : Utilisation et Services*, Campus de Metz de Supélec, 21–25 juin 2004.

[CDF+02] A. Caron, F. Desprez, E. Fleury, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. *Calcul réparti à grande échelle*, chapter Une approche hiérarchique des serveurs de calculs. Hermès Science Paris, 2002. ISBN 2-7462-0472-X.

[CDF+04] F. Capello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, and O. Lodygensky. Computing on large scale distributed systems : Xtremweb architecture, programming models, security, tests and convergence with grid. In *Future Generation Computer Systems*, 2004. to appear.

[CDS04] H. Casanova, F. Desprez, and F. Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In *Proceedings of Euro–Par 2004*, Pisa, Italy, August 2004.

[CFK99] K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational grids. In *Proceedings of the Eighth IEEE International Symposium* *on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999. Available at `http://www.globus.org/research/papers/paper3.pdf`.

[CJ03] Y. Caniou and E. Jeannot. New dynamic heuristics in the client-agent-server model. In *IEEE 13th Heteregeneous Computing Workshop - HCW'03*, Nice, France, april 2003.

[CJ04a] Y. Caniou and E. Jeannot. Efficient scheduling heuristics for gridrpc systems. In *IEEE QOS and Dynamic System workshop (QDS) of International Conference on Parallel and Distributed Systems (ICPADS)*, New-Port Beach California, USA, july 2004.

[CJ04b] Y. Caniou and E. Jeannot. Experimental study of multi-criteria scheduling heuristics for gridrpc systems. In *ACM/IFIP/IEEE Euro–Par 2004*, Pisa, Italy, 31 aug–3 sep 2004.

[CLZB00] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings of the 9th Heterogeneous Computing Workshop 2000*, pages 349–363, 2000. ISBN 0-7695-0556-2.

[CM02] H. Casanova and L. Marchal. A network model for simulation of grid application. Technical Report 2002-40, LIP, oct 2002.

[COBW00] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The apples parameter sweep template : User-level middleware for the grid. In *Proceedings of the Super Computing Conference (SC'2000)*, 2000.

[Com95] Information Networks Division Hewlett-Packard Company. Netperf : A network performance benchmark, 1995. Homepage available at `http://www.netperf.org/`.

[DaS] the Darmouth Scalable Simulation Framework DaSSF. Homepage at `http://www.crhc.uiuc.edu/~jasonliu/projects/ssf/intro.html`.

[DCB02] H. Dail, H. Casanova, and F. Berman. A decoupled scheduling approach for the grads program development environment. In *Proceedings of the IEEE/ACM SC2002 Conference*, November 16 - 22 2002. Available at `http://csdl.computer.org/dl/proceedings/sc/2002/1524/00/15240055.pdf`.

[DEMT04] P. F. Dutot, L. Eyraud, G. Mounier, and D. Trystram. Bi-Criteria Algorithm for Scheduling Jobs on Cluster Platforms. In *Proceedings of SPAA 2004*, 2004. to appear.

[DJ01] F. Desprez and E. Jeannot. Adding data persistence and reddistribution to netsolve. Technical Report 2001-39, LIP, Dec 2001.

[DLP03] J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark : Past, present and future. In *Concurrency and Computation : Pratice and Experience*, volume 1(18), 2003.

[Doa96] M. Doar. A better model for generating test networks. In *Proceedings of Globecom'96*, November 1996. Available at `http://citeseer.nj.nec.com/doar96better.html`.

[DS04] F. Desprez and F. Suter. Impact of Mixed-Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms. *Concurrency and Computation :Practice and Experience*, 16(8) :771–797, July 2004.

[dSCH04] F. A. B. da Silva, S. Carvalho, and E. R. Hruschka. A scheduling algorithm for running bag-of-tasks data mining applications on the grid. In *Proceedings of the ACM/IFIP/IEEE Euro–Par 2004*, September 2004. Available at `http://www.ourgrid.org/papers/europar04.pdf`.

[dSeSG91] E. de Souza e Silva and M. Gerla. Queueing network models for load balancing in distributed systems. In *Journal of Parallel and Distributed Computing*, volume 12(1), pages 24–38, 1991.

[eG] Grid eXplorer GdX. `http://www.lri.fr/~fci/GdX/`.

[EHS+02] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Enhanced algorithms for multi–site scheduling. In *Proceedings of the 3rd International Workshop on Grid Computing*, Baltimore, 2002. Springer–Verlag, Lecture Notes in Computer Science, LNCS.

[ELvD+96] D. H. J Epema, Miron Livny, R. van Dantzig, X. Evers, and Jim Pruyne. "a worldwide flock of condors : Load sharing among workstation clusters". In *Journal on Future Generations of Computer Systems*, volume 12, 1996.

[Ess04] M. Essaïdi. *Échange de données pour le parallélisme à gros grain*. PhD thesis, Université Henri Poincaré, Nancy, France, février 2004.

[Fei99] D. G. Feitelson. Scheduling parallel jobs on clusters. In R. Buyya, editor, *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 549–533, 1999.

[Fei02] D. G. Feitelson. Workload modeling for performance evaluation. In M. C. Calzarossa and S. Tucci, editors, *Performance Evaluation of Complex Systems : Techniques and Tools*, pages 114–141. Springer-Verlag, Sep 2002.

[FFF99] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law reltionships of the internet topology. In *SIGCOMM*, pages 251–262, 1999. Available at `http://citeseer.ist.psu.edu/faloutsos99powerlaw.html`.

[FGA+98] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kiddand, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar. 1998. Available at `http://ipdps.eece.unm.edu/1998/hcw/freund.pdf`.

[Fig97] S. Figueira. *Modeling the effects of Contention on Application Performance*. PhD thesis, University of California, San-Diego, 1997.

[FK97] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2) :115–128, 1997.

[FK98] I. Foster and C. Kesselman. The globus project : A progress report. In *Heteregeneous Computing Workshop*, march 1998.

[FK99] I. Foster and C. Kesselman. Morgan Kaufmann Publishers Inc., 1999. ISBN 1-55860-475-8.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid : An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, June 22, 2002. Available at `http://www.globus.org/research/papers/ogsa.pdf`.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. In *International Journal of Supercomputer Applications*, volume 15(3), 2001.

[FP99] S. Floyd and V. Paxson. Why we don't know how to simulate the internet. 1999. Available at `http://citeseer.ist.psu.edu/paxon99why.html`.

[FP01] S. Floyd and V. Paxson. Difficulties in simulating the internet. In *IEEE/ACM Transactions on Networking*, february 2001.

[FR98] D. G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science Vol. 1459*, pages 1–24. Springer-Verlag, 1998.

[fRMfPC] GridSim : A Grid Simulation Toolkit for Resource Modelling, Application Scheduling for Parallel, and Distributed Computing. Homepage at `http://www.gridbus.org/gridsim/`.

[GBG+03] D. Geer, R. Bace, P. Gutmann, P. Mtezger, C. Pfleeger, J. Quaterman, and B. Schneier. Cyberinsecurity : The cost of monopoly, september 2003.

[GG00] M. Good and J.-P. Goux. imw : A web–based problem solving environment for grid computing applications. Technical report, Department of Electrical and Computer Engineering, Northwerstern University, 2000. Available at `http://www-unix.mcs.anl.gov/metaneos/publications/`.

[GJ79] M. R. Garey and D. S. Johnson. Computers and intractability : A guide to the theory of np–completeness, 1979.

[GKM82] S. Graham, P. Kessler, and M. McKusick. gprof : A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, volume 17(6), pages 120–126, June 1982.

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. *A High–Performance, Portable Implementation of the MPI Message Passing Interface Standard*, volume 22(6), pages 789–828. September 1996.

[GMC02] V. Galtier, K. Mills, and Y. Carlinet. Modeling cpu demand in heterogeneous active networks. pages 511–534, San Francisco, CA, May 2002. Available at `http://csdl.computer.org/comp/proceedings/dance/2002/1564/00/15640511ab%s.htm`.

[GRRL04] F. Guirado, A. Ripoll, C. Roig, and E. Luque. Performance prediction using an application-oriented mapping tool. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04)*, page 184, A Coruna, Spain, February 11–13 2004. Available at `http://csdl.computer.org/dl/proceedings/pdp/2004/2083/00/20830184.pdf`.

[GSW02] L. Gong, X. H. Sun, and E. F. Watson. Performance modeling and prediction of non–dedicated network computing. volume 51(9), pages 1041–1055, September 2002.

[GT96] M. Galassi and J. Theiler. The gnu standard library, 1996. Availble at http://www.gnu.org/software/gsl/gsl.html.

[GW97] A. S. Grimshaw and W. A. Wulf. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1) :39–45, 1997.

[GWT00] B. Gaidioz, R. Wolski, and B. Tourancheau. Synchronizing network probes to avoid measurement intrusiveness with the network weather service. In *The 9th IEEE High-performance Distributed Computing Conference*, pages 147–154, August 2000.

[GÖD04] K. S. Golconda, F. Özgüner, and A. Dogan. A comparison of static qos-based scheduling heuristics for a meta-task with multiple qos dimensions in heterogeneous computing. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Heterogeneous Computing Workshop HCW'04*, Santa Fe, New Mexico, USA, April 26 – 30 2004.

[HAF00] T. Haupt, E. Akarsu, and G. Fox. Webflow : a framework for web based metacomputing. In *Future Generation Computer Systems*, volume 16(5), pages 445–451, March 2000.

[HBD97] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15(3) :253–285, 1997. Available at http://doi.acm.org/10.1145/263326.263344.

[HNK+00] M. Haldar, A. Nayak, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary, and P. Banerjee. Match virtual machine : An adaptive runtime system to execute matlab in parallel. In *International Conference on Parallel Processing (ICPP-2000)*, Toronto, CANADA, August 2000. Available at http://csdl.computer.org/dl/proceedings/icpp/2000/0768/00/07680145.pdf.

[Hom] Emulab Network Emulation Testbed Home. Homepage at http://www.emulab.net/index.php3.

[HSvL03] X. He, X. Sun, and G. von Laszewski. Qos guided min-min heuristic for grid task scheduling. In *Journal of Computer Science and Technology, Grid computing*, volume 18(4), pages 442–451, July 2003. Available at www.cs.iit.edu/~scs/psfiles/jcst_XHe-5-28.pdf.

[HSW96] L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time : off-line and on-line algorithms. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 142–151, Atlanta, Georgia, United States, 1996. ISBN 0–89871–366–8.

[Hum03] M. Humphrey. From legion to legion–g to ogsi.net : Object–based computing for grids. In *Next Generation Software Workshop in the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS2003)*, Nice, France, 2003.

[ifVR] MPICH-V : MPI implementation for Volatile Resources. `http://www.lri.fr/~gk/MPICH-V/`.

[IO98] M. Iverson and F. Özgüner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *Proceedings of the 7th Heteregeneous Computing Workshop (HCW98)*, march 1998.

[Jon02] J. De Jongh. *Share Scheduling in Distributed Systems.* PhD thesis, Proefschrift Technische Universiteit Delft - Met lit. opg., Pays–Bas, 2002.

[JW04] E. Jeannot and F. Wagner. Two fast and efficient message scheduling algorithms for data redistribution through a backbone. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, New Mexico, USA, 26–30 April 2004. IEEE Computer Society. ISBN 0-7695-2132-0.

[KBM01] K. Krauter, R. Buyya, and M. Mashewaran. A taxonomy and survey of grid resource management systems for distributed computing. In *Software–Practice and Experience*, volume 0 :1–7. John Wiley & Sons, Ltd., 2001.

[Ken51] D. G. Kendall. Some problems in the theory of queues. In *J. Roy Statist. Soc. Ser. B*, volume 13, pages 151–173, 1951.

[KJF98] N. Kapadia, J.Robertson, and J. Fortes. Interfaces issues in running computer architecture tools via the world wide web. In *Proceedings of Workshop on Computer Architecture Education at ISCA*, Barcelona, Portugal, 1998. Punch available at `http://www.ecn.purdue.edu/labs/punch/`.

[KKM+03] R. Keller, B. Krammer, M. S. Mueller, M. M. Resch, and E. Gabriel. Mpi development tools and applications for the grid. In *Workshop on Grid Applications and Programming Tools, in conjonction with the GGF8 meetings*, Seattle, june 2003.

[Kra99] M. Krajecki. An object oriented environment to manage the parallelism of the fiit applications. In V. Malyshkin, editor, *Parallel Computing Technologies, 5th International Conference, PaCT–99*, volume 1662 of Lecture Notes in Computer Science, pages 229–234, St. Petersbourg, Russia, September 1999. Springer–Verlag.

[KTF03] N. Karonis, B. Toonen, and I. Foster. Mpich–g2 : a grid–enabled implementation of the message passing interface. In *Journal of Parallel and Distributed Computing (JPDC)*, volume 63(5), pages 551–563, may 2003. `http://www.hpclab.niu.edu/mpi/`.

[Kwo97] Y.-K. Kwok. *High-Performance Algorithms for Compile-Time Scheduling of Parallel Processors.* PhD thesis, HKUST, Hong Kong, 1997. Available at `http://citeseer.ist.psu.edu/kwok97highperformance.html`.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. In *Comm. ACM, vol.21,7*, pages 558–565. ACM Press, New York, july 1978. Available at `http://portal.acm.org/citation.cfm?id=359563&coll=portal&dl=ACM`.

[LDR02] D. Lee, J. Dongarra, and R. S. Ramakrishna. visperf : Monitoring tool for grid computing. In *Submitted to the Workshop on Grid Computing, Baltimore, MD*, 2002.

[Leg03] A. Legrand. *Algorithmique parallèle hétérogène et techniques d'ordonnancement : approches statiques et dynamiques.* PhD thesis, École Nationale Supérieure de Lyon, 2003.

[LS02] U. Lang and R. Schreimer. Developing secure distributed systems with corba, 2002.

[LS03] B.-D. Lee and J. M. Schopf. Run–time prediction of parallel applications on shared environments. In *Prceedings of Cluster'03*, 2003.

[LWF+99] C. Lee, R. Wolski, I. Foster, C. Kesselman, and J. Stepanek. A network performance tool for grid computations. In *Supercomputing '99*, 1999. Available at `http://www.globus.org/documentation/incoming/gloperf_sc99.pdf.pdf`.

[MAS+99] M. Maheswaran, S. Ali, H. J. Siegel, D. Hengsen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing system. In *Journal of Parallel and Distributed Computing, Special Issue on Software Support for Distributed Computing*, June 1999.

[MAT] MATLAB. Homepage at `http://www.mathworks.com/`.

[MBA01] M. Murshed, R. Buyya, and D. A. Abramson. Gridsim : A toolkit for the modeling and simulation of global grids. Technical Report 2001/102, 2001.

[MBS99] M. Maheswaran, T. D. Braun, and H. J. Siegel. Heterogeneous distributed computing. *Encyclopedia of Electrical and Electronics Engineering*, 8 :679–690, 1999. Available at `http://meseec.ce.rit.edu/eecc722-fall2002/papers/hc/2/maheswaran99heter%ogeneous.pdf`.

[MCC04] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system : Design, implementation, and experience. 30(7), July 2004. Available at `http://www.cs.berkeley.edu/~massie/papers/science.pdf`.

[ML91] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. In *Proceedings of the 12th IFIP WG 7.3 Symposium on Computer Performance*, volume 12(4, 1991.

[Mol] I. Molnar. Goals, design and implementation of the new ultra-scalable o(1) scheduler. in kernel archive : linux-2.6.*/Documentation/shed-design.txt.

[MP02] T. Monteil and P. Pascal. Influence of deterministic customers in the time sharing scheduler. Technical Report 02509, LAAS, Laboratoire d'Analyse et d'Architecture des Systèmes, November 2002.

[MPI] MPI. Mpi forum webpage at `http://www.mpi-forum.org/`.

[MRW92] A. Malony, D. Reed, and H. Wijshoff. Performance measurement intrusion and perturbation analysis. In *IEEE Transactions on Parallel and Distributed Systems*, volume 3(4), July 1992.

[NBK+00] M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++ : scalability issues in global computing. In *Concurrency : Practice and Experience*, volume 12(8), pages 727–753, 2000.

[NGB] Zs. Németh, G. Gombás, and Z. Balaton. Performance evaluation on grids : Directions, issues, and open problems. In *Proceedings of the Euromicro PDP 2004*, pages 290–297, A Coruna, Spain. IEEE Computer Society Press. Available at http://csdl.computer.org/dl/proceedings/pdp/2004/2083/00/20830290.pdf.

[NMS+03] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications, December 2003. Available at https://forge.gridforum.org/projects/gridrpc-wg/document/GridRPC_EndUse%r_16dec03/en/1.

[NPRC00] M. O. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0 : Java-based parallel computing on the internet. In *Proceedings of Euro–Par 2000*, Munich, GERMANY, August 28 - September 1, 2000. Available at http://javelin.cs.ucsb.edu/docs/europar.pdf.

[NS03] Zs. Németh and V. Sunderam. *Characterizing Grids : Attributes, Definitions, and Formalisms*, volume 1(1), pages 9–23. 2003.

[NSS99] H. Nakada, M. Sato, and S. Sekiguchi. Design and implementations of ninf : towards a global computing infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15 :649–658, 1999.

[Our] OurGrid. Homepage available at http://www.ourgrid.org/.

[PBS] PBS. Portable batch system. Homepage at http://www.openpbs.org/.

[Phi04] L. Philippe. Gridrpc : Normalisation des api d'accès aux applications de grilles, 21–25 juin 2004.

[Plaa] PlanetLab. An open platform for developing, deploying, and accessing planetary–scale services. Homepage at http://www.planet-lab.org/.

[Plab] Personal Power Plant. Homepage at http://www.shudo.net/.

[Pro] The Globus Project. Homepage available at http://www.globus.org.

[Pro99] ProActive. INRIA, 1999. http ://www-sop.inria.fr/oasis/ProActive.

[PST] K. Pruhs, J. Sgall, and E. Torng. Online scheduling.

[PVM] PVM. The parallel virtual machine. Homepage at http://www.csm.ornl.gov/pvm/pvm_home.html.

[Qui02] M. Quinson. Dynamic performance forecasting for network-enabled servers in a metacomputing environment. In *International Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO-PDS'O2)*, april 15-19 2002.

[Qui03] M. Quinson. *Découverte automatique des caractéristiques et capacités d'une plate-forme de calcul distribué*. PhD thesis, Ecole Normale Supérieure de Lyon, 2003.

[Ram96] S. Ramaswany. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, University of Illinois at Urbana–Champaign, 1996.

[Ree94] D. A. Reed. Experimental performance analysis of parallel systems : Techniques and open problems. In *Proceedings of the 7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, pages 25–51, May 1994.

[SA99] H. J. Siegel and S. Ali. Techniques for mapping tasks to machines in heterogeneous computing systems. In *Invited Keynote Paper for the Journal of Systems Architecture, Special Issus on Heterogeneous Distributed and Parallel Architectures : Hardware, Software and Design Tools*, June 1999.

[SBW99] G. Shao, F. Berman, and R. Wolski. Using effective network views to promote distributed application performance. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, june 1999. Available at `http://apples.ucsd.edu/pubs/pdpta99.ps`.

[Sch98] J. Schopf. *Performance Prediction and Scheduling for Parallel Applications on Multi-Users Clusters*. PhD thesis, University of California, San-Diego, 1998.

[Sim] The Network Simulator. NS-2 – Homepage at `http://www.isi.edu/nsnam/ns/`.

[SLJ+00] Song, Liu, Jakobsen, Bhagwan, Zhang, Taura, and Chien. The microgrid : a scientific tool for modeling computational grids. In *Scientific Programming*, volume 8(3), pages 127–141, 2000.

[SR97] E. Smirni and D. A. Reed. Workload characterization of input/output intensive parallel applications. In *Proceedings of the Ninth International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, June 1997.

[SSYS04] T. Sato, J. Slingo, T. Yukutake, and H.D. Simon, editors. The Earth Simulator Center, Japan Agency for Marine-Earth Science and Technology, april 2004. Available at `http://www.es.jamstec.go.jp/esc/images/journal200404/index.html`.

[Sti85] J. R. Stidham. Optimal control of admission to a queueing system. In *IEEE Trans. on Automatic Control*, volume 30(8), pages 705–713, August 1985.

[SW03] X.-H. Sun and M. Wu. Grid harvest service : A system for long-term, application-level task scheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 22-26 april 2003.

[Sys90] MIPS Computer Systems. Umips–v reference manual (pixie and pixstats), 1990.

[Tak01] A. Takefusa. Bricks : A performance evaluation system for scheduling algorithms on the grids. In *JSPS Workshop on Applied Information Technology for Science (JWAITS 2001)*, 2001.

[Tal91] E. G. Talbi. Une taxonomie des algorithmes d'allocation dynamique de processus dans les systèmes parallèles et distribués, 1991. Available at `htt://citeseer.ist.psu.edu/132594.html`.

[Tan92] A. S. Tanenbaum. Modern operating system. In *Prentice–Hall Inc.*, 1992. ISBN 0-13-031358-0.

[TMCB01] A. Takefusa, S. Matsuoka, H. Casanova, and F. Berman. A study of deadline scheduling for client–server systems on the computational grid. In *the 10th IEEE Symposium on High Performance and Distributed Computing, HPDC'01*, San Francisco, California, August 2001.

[TTL03] D. Thain, T. Tannenbaum, and M. Livny. chapter Condor and the Grid. F. Berman and A. Hey and G. Fox, 2003.

[TWS03] V. Taylor, X. Wu, and R. Stevens. Prophesy : an infrastructure for performance analysis and modeling of parallel and grid applications. In *Special section on grid computing*, volume 30(4), pages 13–18. ACM Press New York, NY, USA, 2003. Available at `http://portal.acm.org/citation.cfm?id=773060&dl=ACM&coll=portal`.

[Vam] Vampire. Vampire 2.0, visualization and analysis of mpi programs. Available at `http://www.pallas.com`.

[VAMR01] F. Vraalsen, R. A. Aydt, C. L. Mendes, and D. A. Reed. Performance contracts : Predicting and monitoring grid application behavior. In *2nd International Workshop on Grid Computing*, volume 7(11), pages 154–165, november 2001. Available at `citeseer.ist.psu.edu/vraalsen01performance.html`.

[VR00] J. S. Vetter and D. A. Reed. Real–time performance monitoring, adaptive control and interactives steering of computational grids. In *The International Journal of High Performance Computing Applications*, volume 14(4), pages 257–366, 2000.

[VSF02] S. Vazhkudai, J. M. Schopf, and I. Foster. Predicting the performance of wide area data transfers. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, April 2002. Available at `http://www.globus.org/research/papers/Prediction-Paper-249.pdf`.

[Wei96] J. B. Weissman. The interference paradigm for network job scheduling. In *Proceedings of the 10th International Parallel and Distributed Processing Symposium, HCW*, 1996.

[WM85] Y. T. Wang and R. J. T. Morris. Load sharing in distributed systems. In *IEEE Transactions on Computers*, volume 34(3), pages 204–217, 1985.

[WS01] J. B. Weissman and P. Srinivasan. Ensemble scheduling : Resource co–allocation on the computational grid. In *2nd International Workshop on Grid Computing, Grid'01*, volume 2242. Springer-Verlag, LNCS, 2001.

[WSH99] R. Wolski, N. T. Spring, and J. Hayes. The network service : A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6) :757–768, october 1999.

[WZ97] J. B. Weissman and X. Zhao. Run-time support for scheduling parallel applications in heterogeneous nows. In *Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)*, 1997. ISBN 0-8186-8117-9.

[XDCC04] H. Xia, H. Dail, H. Casanova, and A. Chien. The microgrid : Using emulation to predict application performance in diverse grid network environments. In *Proceedings of the Workshop on Challenges of Large Applications in Distributed Environments (CLADE'04), held in conjunction with the Thirteenth IEEE International Symposium on High-Performance Distributed Computing*, Honolulu, Hawaii, June 2004. Available at `http://www-csag.ucsd.edu/papers/clade04xia.pdf`.

[YFS03] L. Yang, I. Foster, and J. M. Schopf. Homeostatic and tendency-based cpu load predictions. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium, IPDPS'03*, April 2003.