Un outil de prédiction dynamique de performances dans un environnement de metacomputing

Martin Quinson

LIP, UMR CNRS-ÉNS Lyon-INRIA 5668
École Normale Supérieure de Lyon
46, allée d'Italie
69364 Lyon Cedex 07
Martin.Quinson@ens-lyon.fr

Ce travail a été en partie supporté par l'Action de Recherche Coopérative OURAGAN de l'INRIA et du RNTL GASP.

RÉSUMÉ. *Cet article présente un outil de prédiction dynamique de performances dans un environnement de metacomputing de type* Client-Agent-Serveurs. *Dans ce modèle, les différentes routines composant un programme peuvent être délocalisées sur des serveurs distants. Afin d'en faciliter le placement, la bibliothèque* FAST *(*Fast Agent's System Timer*) permet aux applications appelantes d'obtenir des prédictions précises des besoins (en temps d'exécution, en espace mémoire et en quantité de communications) des routines de calcul à ordonnancer dans le système, ainsi que de les comparer aux disponibilités actuelles du système.* FAST *est basé sur des logiciels existants de plus bas niveau pour la surveillance du réseau et des machines. Des développements propres permettent la modélisation des exécutions des routines à placer. L'interface offerte et le fonctionnement du système sont présentés, ainsi que des résultats expérimentaux.*

ABSTRACT. *This paper presents a tool for dynamic forecasting of Network-Enabled Servers performance. In NES-based systems, the computation routines constituting a program may be executed on remote computers. To ease their mapping, the* FAST *library (*Fast Agent's System Timer*) allows client applications to get accurate forecasts of computation routines needs (in terms of completion time, memory space and communication amount), as well as of current system availability.* FAST *relies on existing low level software packages, i.e., network and host monitoring tools, and some of our developments in computation routines modeling. The* FAST *internals and user interface are presented, as well as experimental results.*

MOTS-CLÉS : *Prédiction et modélisation des performances, Surveillance de ressources, Metacomputing.*

KEYWORDS: *Performance forecasting and modeling, Resource monitoring, Metacomputing.*

1. Introduction

L'objectif du metacomputing est de fédérer des ressources informatiques hétérogènes et réparties afin d'en agréger la puissance. L'une des approches les plus classiques consiste à mettre au point un environnement de calculs à la demande construit sur le schéma « client-agent-serveurs » : le client soumet une requête de calcul à un agent, qui la transmet au serveur le plus adapté en termes de disponibilité et de performances escomptées. Ce serveur effectue alors le calcul en exécutant une routine séquentielle ou parallèle issue d'une bibliothèque spécialisée. Le résultat est ensuite renvoyé au client.

L'appréciation de l'adéquation d'un serveur pour un calcul donné est donc l'un des problèmes majeurs à résoudre. De plus, la nature dynamique, partagée et hétérogène de la plate-forme utilisée complique l'obtention de caractéristiques de l'environnement, telles que les besoins des tâches à ordonnancer et les disponibilités du système. De plus, l'environnement étant interactif, il est important de minimiser les temps de réponse.

Cet article présente la bibliothèque FAST (*Fast Agent's System Timer*), dont l'objectif est de fournir à un ordonnanceur les informations nécessaires. FAST est ainsi capable de prédire le temps d'exécution et l'espace mémoire nécessaires aux tâches à ordonnancer dans le système grâce à une modélisation des besoins des routines. Des outils de surveillance adéquats permettent en outre de mesurer les disponibilités dynamiques des différentes ressources. FAST est également capable de mettre ces valeurs en correspondance pour prédire le temps nécessaire à l'exécution d'une tâche donnée sur une machine donnée à un instant donné. L'objectif de FAST n'est cependant pas de réaliser le placement des routines à proprement parler, mais plutôt d'offrir à l'application cliente les connaissances nécessaires à cette tâche, au travers d'un environnement de programmation simple et consistant. De plus, un soin particulier a été apporté afin de réduire les temps de réponse de la bibliothèque pour d'en permettre un usage interactif.

Le problème du placement peut être vu comme une mise en adéquation des besoins des routines et des disponibilités du système. Attachons–nous maintenant à expliciter et définir ces grandeurs.

Les besoins des routines regroupent principalement le temps et l'espace mémoire nécessaires à leur exécution, ainsi que, dans le cas de routines parallèles, le volume de communication généré entre les nœuds. Ces valeurs dépendent naturellement de l'implémentation choisie et des paramètres d'entrée de la routine, mais également de la machine sur laquelle le calcul a lieu. Par exemple, les implémentations des bibliothèques de calcul numérique profitent souvent d'optimisations poussées. De nombreux paramètres matériels tels que la taille des caches influent alors de manière significative sur les performances obtenues pour une même fonction.

Les disponibilités du système concernent tout d'abord le nombre et la puissance des machines. Il faut également savoir si elles sont accessibles, en panne ou réservées au travers d'un système de type batch. Il faut enfin connaître la topologie, les capacités

et les protocoles des réseaux de communication mis en place entre ces machines. Dans ce cadre, la capacité et la disponibilité escomptées des ressources à un instant donné sont plus pertinentes pour un ordonnanceur que les performances en crête ou l'usage qui en est fait.

On peut remarquer que les besoins des routines ne varient pas au cours du temps (ou très rarement, par exemple lors d'une modification matérielle comme l'ajout de mémoire permettant d'accélérer l'exécution), tandis que les disponibilités du système sont extrêmement dynamiques. Différentes techniques doivent donc être mises en place pour collecter ces informations.

FAST a été développé dans le cadre des travaux de l'action de recherche coopérative OURAGAN de l'INRIA, à laquelle participe l'équipe ReMaP du Laboratoire d'Informatique du Parallélisme de l'ENS de Lyon. Cette bibliothèque coopère donc pleinement avec les autres outils développés dans ce cadre, tels que SLiM (Scientific Library Metaserver), qui est chargé de choisir la meilleure fonction disponible pour un problème donné, ou DIET (Distributed Interactive Engineering Toolbox), qui est un environnement de calcul à la demande orienté metacomputing. Mais FAST ne dépend pas de ces outils, comme le prouve son intégration au système NetSolve [CAS 97], une plate-forme de metacomputing développée à l'Université du Tennessee. Nous avons également intégré FAST à Cichlid [1], un outil de visualisation de la charge du système en temps réel développé par le National Laboratory for Applied Network Research (NLANR) à l'Université de Californie, San Diego.

FAST est disponible librement sous licence de type BSD sur notre site web [2]. Il existe également une liste de diffusion [3]. Il fonctionne sur différents systèmes d'exploitation tels que Linux, TRU64 ou Solaris.

La suite de cet article est organisée de la manière suivante. La section 2 nous permettra tout d'abord de donner un rapide état de l'art. Nous exposerons ensuite l'architecture de FAST dans la section 3, avant d'en détailler l'interface à la section 4. La section 5 présentera quant à elle des résultats expérimentaux. Nous conclurons cet article à la section 6 en donnant un aperçu des travaux futurs liés à ce projet.

2. État de l'art

De par ses objectifs, FAST se trouve à l'intersection de différents domaines : la surveillance de systèmes distribués, l'étude des performances de programmes et enfin le cadre plus général du metacomputing.

1. http://moat.nlanr.net/Software/Cichlid/
2. http://www.ens-lyon.fr/~mquinson/fast.html
3. fast-dev@listes.ens-lyon.fr

2.1. *Surveillance de systèmes distribués*

Les outils présentés ici permettent de surveiller les disponibilités d'un système distribué. La plupart du temps, des sondes logicielles sont déployées, et les données collectées sont ensuite centralisées pour pouvoir être traitées.

2.1.1. *NetPerf*

NetPerf [HP 95] est un projet mené par Hewlett-Packard au milieu des années 90. L'architecture suit le modèle client-serveur. Un programme est utilisé pour tester la connectivité entre la machine locale et une machine distante sur laquelle est installée le serveur de NetPerf. Cet environnement est également capable de donner quelques mesures sommaires sur la disponibilité du processeur, mais cette fonctionnalité n'est présente que sur des noyaux HP-UX et nécessite les privilèges de l'administrateur système. Ce projet a surtout été conçu pour pouvoir étalonner les réseaux, afin d'en classer les différents composants matériels en fonction de leurs performances, et non pour connaître leurs disponibilités réelles. De plus, la dernière version publiée date de 1996.

2.1.2. *PingER*

PingER [MAT 00] est une architecture de surveillance réseau déployée sur plus de 600 hôtes dans 72 pays. Comme le nom du projet l'indique, les mesures sont réalisées grâce à l'outil classique d'administration `ping`. De manière périodique, une série de tests est menée entre les paires d'hôtes du système et les résultats sont stockés afin d'être accessibles depuis une page web. Les objectifs du projet sont de faciliter le choix de partenaires pour des calculs intensifs (comme en physique nucléaire), repérer les sites nécessitant une mise à jour, ou permettre de choisir le meilleur fournisseur d'accès web. Bien qu'il soit tentant de permettre à un ordonnanceur d'accéder à la grande quantité de mesures réalisées dans le cadre de ce projet, rien ne semble fait jusqu'à présent pour en permettre un usage interactif.

2.1.3. *Performance Co-Pilot*

Performance Co-Pilot [SGI 01] est un système développé par SGI (Silicon Graphics Inc) pour surveiller les ressources systèmes d'un ensemble de machines distribuées. Ce projet est fondé sur des démons collectant les données sur les différentes machines ainsi que sur un SDK (Software Development toolKit) permettant à un client d'accéder à distance à ces données. Le projet semble séduisant, mais il n'offre que des données de bas niveau telles que les débits d'entrée/sortie, la taille de chaque processus en mémoire, l'occupation du processeur, *etc.* Il est donc plus question de l'usage fait de chaque ressource que des disponibilités pour de nouveaux processus. Ces informations ne sont donc pas directement utilisables par un ordonnanceur pour prédire le temps nécessaire à l'exécution d'une routine et devraient faire l'objet d'un prétraitement non trivial. Par exemple, le mécanisme de priorités utilisé par Unix empêche de déduire simplement la quote-part du processeur dont disposerait un processus à partir

de sa seule charge [WOL 98]. En effet, une machine très chargée par des processus de basse priorité reste relativement disponible pour des processus de priorité plus haute.

2.1.4. *NWS*

NWS (Network Weather Service) [WOL 99] est un projet mené par Wolski à l'Université de Californie San Diego. Il s'agit d'un système distribué de sondes logicielles et de prédicteurs statistiques permettant de centraliser l'état actuel de la plate-forme, ainsi que d'en prédire les évolutions. Il est ainsi possible de surveiller la latence et la bande passante de n'importe quel lien de communication TCP/IP, la charge processeur, la mémoire disponible ou encore l'espace libre sur les disques. En ce qui concerne le processeur, NWS n'est pas seulement capable de décrire la quantité utilisée, mais également la quote-part dont disposerait un nouveau processus arrivant. Cette caractéristique le différencie des projets précédemment cités.

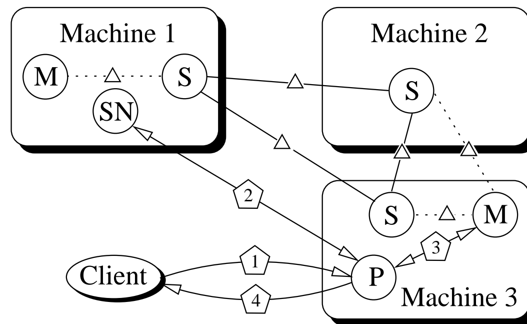


Figure 1. *Fonctionnement de NWS*

La figure 1 représente le fonctionnement de NWS. Le système est composé de quatre types de serveurs :

– Les senseurs (**S**) effectuent les mesures ;

– Les mémoires de stockages (**M**) enregistrent les résultats sur disque pour un usage ultérieur ;

– Les prédicteurs (**P**) effectuent un traitement statistique sur les valeurs passées pour prédire les évolutions du système ;

– Le serveur de nom (**SN**) garde un annuaire du système. Il permet à chaque partie du système de localiser n'importe quel autre serveur.

NWS effectue des mesures à intervalles réguliers, même lorsqu'aucune requête n'est soumise. Les résultats sont ensuite envoyés aux serveurs de stockage. Sur la figure, les communications ayant lieu lors de ce régime permanent sont marquées par un Δ.

Lorsqu'un client soumet une requête au prédicteur (étape 1), ce dernier contacte le serveur de nom pour localiser dans quelle mémoire se trouvent les données correspon-

dantes (étape 2). Dans l'exemple présenté par la figure 1, il s'agit de celle placée sur la machine 3. Le prédicteur contacte ensuite la mémoire pour obtenir l'historique des mesures réalisées (étape 3). Il effectue ensuite un traitement statistique afin de prédire la prochaine valeur de la série, qu'il renvoie au client (étape 4).

Nous avons choisi d'utiliser ce projet pour la surveillance des ressources dans le cadre de FAST, et nous détaillerons dans la partie 3.2 notre usage de NWS, ainsi que les améliorations que nous avons apportées à ce système.

2.2. *Prédiction et analyse de performances*

Cette section présente plusieurs travaux visant à modéliser les performances des programmes pour tenter de prédire leur temps d'exécution.

2.2.1. *Dimensionnement système*

Il existe de nombreux travaux dont l'objectif est d'étudier et de modéliser le matériel afin de pouvoir déterminer l'architecture la plus appropriée pour obtenir de bonnes performances pour un problème donné. Ainsi [JAC 96] donne un modèle analytique de la hiérarchie mémoire et de son impact sur les performances des programmes, tandis que [WEI 00] donne un modèle analytique des systèmes de cache. L'approche analytique semble malheureusement mal adaptée à un outil comme FAST, en raison de la quantité de paramètres à prendre en considération dans le calcul. En effet, il faut tenir compte des performances des différents niveaux de la hiérarchie mémoire, des algorithmes de cache et d'ordonnancement mis en place par le matériel ou le système d'exploitation, *etc.* Cette approche conduit donc à effectuer une simulation complète de l'architecture cible, ce qui semble contradictoire avec l'usage interactif de notre outil.

2.2.2. *Bricks*

Bricks [AID 00] est un outil développé pour simuler une grille de calcul. Il permet ainsi de reproduire les expériences de metacomputing sous les mêmes conditions, ce qui n'est pas possible habituellement en raison du partage des ressources avec d'autres utilisateurs. Cela provoque en effet des perturbations sur l'expérience très difficilement reproductibles. Bricks a été mis au point afin de pouvoir évaluer différentes stratégies d'ordonnancement dans des conditions comparables. Ces travaux se rapprochent malgré tout des nôtres, car simuler une grille de calcul implique d'en avoir un modèle. Bricks décrit le système entier sous forme de files d'attente. Un serveur est par exemple une queue, où les tâches sont stockées lorsqu'elles arrivent, et ensuite débloquées après leur « traitement ». Elles sont modélisées par deux valeurs : la quantité de calcul nécessaire à leur traitement et le volume de communications qu'elles occasionnent.

Cette modélisation présente deux défauts majeurs. Le premier est son manque de précision, car l'efficacité d'une machine (en Flops/s) est supposée constante. Or, cer-

taines opérations d'algèbre linéaire, telle que la multiplication matricielle, sont plus rapides que d'autres sur un ordinateur pipeliné moderne, car elles profitent au mieux des effets de cache. Par ailleurs, cette approche demande de gros calculs pour être menée à bien, ce qui empêche son utilisation dans un cadre interactif.

2.3. *Systèmes de metacomputing*

Cette section présente différents projets existants de metacomputing en mettant l'accent sur les solutions apportées dans les domaines de la surveillance de la plate-forme et de la prédiction des performances.

2.3.1. *RCS*

Remote Computation Service (RCS) [ARB 96] est un système d'invocation de routines à distance (RPC) permettant d'unifier les interfaces de toutes les ressources. Il dispose de son propre système de sondes à la manière de NWS, de sa propre couche de communication et de modélisation du système. Mais RCS n'offre pas la possibilité d'extrapoler les évolutions à venir et les sondes ne permettent pas de surveiller le processeur ou la charge mémoire des machines. Les communications sont basées sur PVM, ce qui gène le déploiement du système dans un contexte de metacomputing.

La modélisation des routines de RCS ne concerne que les temps d'exécution et est basée sur l'analyse manuelle du code source pour déterminer la complexité du calcul. Par exemple, le temps d'exécution de la fonction de factorisation LU de *ScaLAPACK* est estimé par l'expression $\frac{2}{3}n^3 - \frac{1}{2}n^2 + \frac{5}{6}n$, où n est la taille des données à traiter. Mais il ne s'agit là que de la complexité théorique de la fonction, et si cette expression est asymptotiquement juste, elle ne tient pas compte de l'architecture de la machine. De fait, les performances du même algorithme peuvent varier d'un ordre de grandeur, selon qu'il soit implémenté simplement ou qu'il s'agisse d'une version optimisée par le vendeur du matériel.

2.3.2. *Globus*

Globus [FOS 97] est l'un des projets de metacomputing les plus connus et les plus avancés. Tous les éléments nécessaires à la mise en place d'une telle plate-forme y sont représentés. L'outil de mesure de performance associé est le Globus Network Performance Measurement Tool (GloPerf) [FOS 98]. Il s'agit de démons effectuant des mesures sur des liens de communication. Chaque démon maintient une liste de tous les autres démons existants et effectue des tests périodiques.

Il y a au moins quatre désavantages à utiliser ce système. Tout d'abord, cette liste exhaustive des démons est un facteur limitant la capacité de montée en charge de l'ensemble. Ensuite, le système se contente de mesures, et n'offre pas de système de prédiction statistique comme NWS. De plus, GloPerf est trop étroitement lié à Globus et à sa base de données, le MDS. Enfin, GloPerf ne surveille que des ressources réseaux et non la charge processeur ou mémoire.

2.3.3. APPLES

L'objectif du projet APPLES (Application Level Scheduler [BER 97]) est de réaliser les éléments d'un ordonnanceur centré sur l'application. La métrique de qualité de l'ordonnancement n'est donc pas l'usage qui est fait de la machine ou la quantité de tâches terminées par unité de temps, mais le temps de complétion de l'application. Il s'agit donc d'un ordonnanceur suivant l'approche metacomputing. Ce projet a davantage débouché sur une méthodologie, et sur la parallélisation de grands codes existants que sur un environnement de metacomputing complet. En effet, la plupart des composants de l'environnement sont dédiées à l'application qu'ils servent, et doivent être ré-écrits pour chaque nouvelle application.

La surveillance des ressources est assurée par NWS. La modélisation des performances des applications est basée sur un modèle structurel, qui décompose une opération parallèle en opérations séquentielles simples et en communications entre les nœuds. Dans [SCH 97], les auteurs présentent une extension de ce modèle utilisant des valeurs par intervalles. Cette approche est séduisante, mais présente deux problèmes : tout d'abord, l'intervalle de valeurs utilisé représente les moyennes sur les 24 dernières heures, car aucun outil de surveillance réseau à notre connaissance n'offre d'intervalle de confiance sur les valeurs fournies. De plus, l'usage de machines interactives étant très différent le jour et la nuit, ces valeurs ne nous semblent pas très pertinentes. Enfin, il n'existe pas à notre connaissance d'algorithme d'ordonnancement capable de prendre en compte ces informations supplémentaires.

2.3.4. *NetSolve*

NetSolve [CAS 97] est un environnement complet de metacomputing basé sur le schéma *Client-Agent-Serveurs*. Il permet un choix transparent du meilleur serveur pour un problème donné (en terme de temps de calcul prédit), une exécution synchrone ou non, ainsi que des mécanismes de tolérance aux pannes. Il est possible d'appeler le système depuis différents langages comme C, FORTRAN, Java, Mathematica, Matlab ou Scilab.

Cependant, NetSolve souffre de certaines lacunes. Par exemple, les routines à ordonnancer sont modélisées simplement par une fonction de type xt^y, où x et y sont des paramètres déterminés pour ce problème et t est la taille du problème. De plus, NetSolve ne tient pas compte des capacités mémoire des hôtes. La topologie des réseaux est également ignorée : NetSolve ne mesure que les caractéristiques du réseau entre l'agent et les serveurs. Il considère que ces mesures sont valides pour estimer les caractéristiques du réseau entre le client et un serveur, ce qui mène à des estimations erronées lorsque l'agent et le client ne sont pas sur le même hôte.

3. L'architecture de FAST

FAST se découpe en modules séparés pour faire face aux différents problèmes à résoudre. La figure 2, qui présente ces différents modules, se compose de trois parties.

En haut, on trouve la bibliothèque à proprement parler et ses différents composants. Au centre figurent les outils externes utilisés tandis que le bas de la figure représente un outil chargé de modéliser les routines que le système est susceptible d'ordonnancer.

Nous allons maintenant détailler plus précisément le fonctionnement de FAST et les interactions entre ses composants. Après avoir présenté les outils utilisés au paragraphe suivant, nous reviendrons sur le fonctionnement de l'acquisition des disponibilités du système dans la partie 3.2. Ensuite, la section 3.3 nous permettra de détailler comment FAST détermine les besoins des routines. Nous y présenterons le programme chargé de modéliser ces besoins, ainsi que l'usage fait des données résultantes par la bibliothèque lors de l'exécution. Enfin, nous verrons dans la partie 3.4 comment ces valeurs sont agrégées en prédictions directement utilisables par un ordonnanceur.

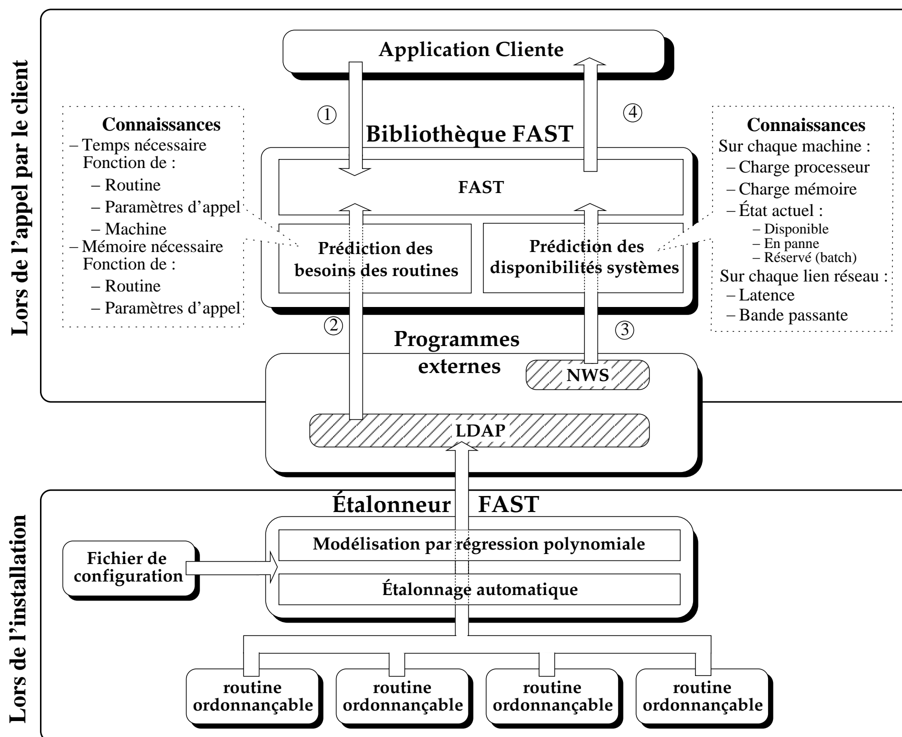


Figure 2. *Architecture de* FAST

3.1. *Les outils utilisés par* FAST

La figure 2 montre que FAST utilise principalement deux outils externes (en grisé). Le premier d'entre eux est NWS (*cf.* section 2.1.4) et le second est LDAP (Lightweight Directory Access Protocol [HOW 99]). Ce dernier est un système distribué et hiérar-

chique de base de données très utilisé, entre autres dans le contexte du metacomputing. Il permet par exemple d'organiser l'arbre des données de façon à suivre le découpage des ressources entre les différents domaines administratifs. Chaque administrateur gère la base de données contenant les informations relatives aux machines qu'il contrôle. Comme de plus, il s'agit d'un protocole ouvert et largement utilisé, sa pérennité est assurée. LDAP est cependant explicitement optimisé pour les opérations de lecture et de recherche, au détriment des opérations d'écriture. Le système n'est donc pas prévu pour stocker des données fortement dynamiques. Il convient en revanche parfaitement pour stocker des données statiques comme la modélisation de chaque routine sur chaque machine.

3.2. *Connaître les disponibilités du système*

Nous allons maintenant étudier la partie de FAST chargée de déterminer les disponibilités du système. L'objectif de ce module est de découvrir la charge induite par l'utilisation des ressources afin de tenir compte du caractère partagé de la plate-forme de metacomputing. Nous avons choisi de baser notre travail sur NWS. Cela nous permet de disposer de fondations solides, largement testées et approuvées. Ce choix n'est nullement limitatif, et il serait parfaitement possible d'utiliser une autre bibliothèque offrant les mêmes informations. Actuellement, FAST est capable de surveiller la charge processeur et la mémoire disponible sur chaque hôte, ainsi que la latence et la bande passante de chaque lien TCP. La surveillance de nouvelles ressources telles que l'espace disque ou les performances de liens non-TCP serait relativement aisée.

L'usage de NWS nous a permis d'identifier certaines de ses faiblesses, et d'apporter des solutions à plusieurs d'entre elles dans FAST. Ainsi, NWS étant composé de quatre sortes de démons répartis sur le réseau liés par des interactions assez complexes, ses temps de réponse ne sont pas négligeables. Comme de plus, NWS réalise ses mesures périodiquement, il est inutile de réinterroger le système avant que la mesure suivante ne soit réalisée. Nous avons donc mis en place un système de cache des réponses permettant de n'interroger NWS que lorsque cela est nécessaire. Les résultats de NWS étant dynamiques, une date limite de validité est associée à chaque valeur stockée dans le cache. Si cette date est passée, cela signifie que NWS a sans doute réalisé une nouvelle mesure, et la valeur dans le cache est alors rafraîchie.

De plus, NWS réalise un traitement statistique des mesures déjà effectuées pour prédire les évolutions futures. Malheureusement, lors de changements radicaux des conditions, comme l'ajout ou la terminaison d'un processus de calcul long sur la machine, l'historique des mesures précédant ce changement faussent la prédiction. Nous avons donc ajouté la possibilité d'indiquer à NWS qu'il doit réinitialiser son historique de mesures, permettant ainsi une collaboration entre un ordonnanceur et NWS à travers FAST.

Enfin, NWS permet à l'administrateur de spécifier les tests réseaux à mener afin de ne faire que les tests nécessaires, et de ne pas saturer le réseau inutilement. Il n'est

malheureusement pas possible de combiner ensuite ces résultats automatiquement. Par exemple, si l'on dispose de trois machines A, B et C, et que l'on a demandé des tests entre A et B d'une part, et B et C d'autre part, NWS est incapable de donner une estimation des capacités de communication entre A et C, car il n'y a pas de test direct entre elles. Dans ce cas, FAST agrège les résultats donnés par NWS pour offrir à l'utilisateur une estimation des paramètres demandés. Il effectue pour cela un parcours de graphe parmi les tests directs pour trouver un chemin entre les machines. La bande passante renvoyée est le minimum de celles rencontrées sur le chemin tandis que la latence est la somme de celles retournées par les tests directs. Les prédictions réalisées de cette manière risquent certes d'être d'une qualité moindre que celles basées sur des tests directs, mais constituent toutefois des informations intéressantes et non négligeables.

3.3. *Connaître les besoins des routines*

Cette partie constitue l'un des apports les plus fondamentaux de FAST. Il s'agit d'être capable de prédire les besoins théoriques en terme de temps de calcul et d'espace mémoire d'une fonction. Ce problème est complexe, et selon le type de routine à évaluer, différentes approches sont possibles.

Certaines routines sont relativement régulières et facilement chronométrables. Il s'agit typiquement de fonctions séquentielles comme la fonction `dgemm()` de la bibliothèque BLAS, qui réalise un produit matriciel dense. L'approche classique est alors de réaliser une modélisation de la routine basée sur une étude manuelle ou automatique du code source. Cette approche n'est pas toujours réalisable car elle nécessite d'avoir accès au code source, ce qui n'est malheureusement pas toujours le cas. Par exemple, la plupart des constructeurs de matériel fournissent une implémentation de la bibliothèque BLAS optimisée pour leur gamme de machines. De plus, même lorsque le code source est disponible, cette approche nous semble trop fastidieuse pour être applicable à notre cas d'étude. En effet, de très nombreux paramètres entrent en jeu lorsqu'il s'agit de prévoir les performances d'un code sur une machine donnée. Il faut bien sûr tenir compte de l'implémentation, mais également des optimisations réalisées par le compilateur, ou même par le processeur, des caractéristiques du système d'exploitation, ou du matériel comme la gestion de la mémoire et des caches. Ce foisonnement de paramètres impose une étude approfondie de chaque couple {code ; matériel}, ce qui représente un travail colossal sur une plate-forme de metacomputing.

Pour pallier ce problème, il est relativement classique d'utiliser un étalonnage générique de la machine pour connaître le nombre d'opérations élémentaires qu'elle est capable de réaliser par unité de temps, puis de dénombrer les opérations nécessaires à chaque routine. Cette approche pèche à nos yeux par sa simplicité car elle ne tient, par exemple, pas compte des effets de cache dont l'influence est pourtant capitale en calcul intensif.

Notre approche consiste à étalonner les performances de la fonction par une série de tests lors de l'installation du système. Ces résultats sont ensuite modélisés par régression polynomiale puis stockés dans une base de données. Cette phase de tests peut être assez coûteuse en temps, mais elle ne doit être faite qu'à l'installation de la bibliothèque. De plus, il est possible de ne tester qu'une seule machine d'un sous-ensemble homogène du parc de machines, et d'utiliser ses valeurs pour toutes les machines identiques.

D'autres routines sont bien plus compliquées à chronométrer, mais plus simples à étudier. C'est par exemple le cas des fonctions régulières parallèles, qui se décomposent naturellement en des appels à une fonction séquentielle sur différents nœuds de calcul, et en communications entre ces machines. Ainsi, le produit matriciel parallèle `pdgemm()` de la bibliothèque ScaLAPACK est basé sur plusieurs invocations concurrentes de la routine séquentielle `dgemm()` sur des sous-matrices. Chronométrer l'exécution de telles fonctions est alors plus difficile [CAR 00] puisqu'il faut tenir compte de nombreux paramètres, comme la distribution des données et de la grille de processeurs utilisée. Il est en outre très difficile de séparer l'usage des ressources par la routine de celui des autres processus.

FAST permet alors d'exprimer la structure de la fonction afin d'en isoler les différentes sous-routines. Les besoins de la fonction sont alors l'agrégation des besoins de chaque élément et des communications entre eux. Bien entendu, il est possible d'obtenir les besoins des sous-routines par des appels (récursifs) à FAST.

Il existe enfin une dernière catégorie de routines, bien plus difficiles à traiter. Il s'agit de celles dont les performances dépendent de caractéristiques des données difficiles à extraire, voire des données elles mêmes, comme c'est le cas en calcul numérique creux ou en traitement d'images. Le traitement de routines de ce type dans FAST reste un problème ouvert, et devra faire l'objet de travaux futurs. L'approche la plus simple serait évidemment d'utiliser un benchmark généraliste pour sélectionner la machine la plus puissante sans chercher à prédire le temps nécessaire.

Dans sa version actuelle, FAST n'est capable de traiter que les routines séquentielles, simplement chronométrables, mais l'intégration du traitement des fonctions parallèles est en cours. Par ailleurs, bien que seuls le temps et l'espace mémoire soient traités pour l'instant, l'intégration de nouvelles ressources, telles que l'espace disque dans les cas *Out-of-Core*, serait relativement aisé.

3.4. *Combiner ces connaissances*

Une fois que les besoins de la routine à ordonnancer et les disponibilités du système sont connus, FAST peut combiner ces résultats en une prédiction du temps d'exécution directement utilisable par un ordonnanceur. Pour cela, il élimine tout d'abord de la liste des machines potentiellement disponibles pour ce calcul celles ne disposant pas de suffisamment de mémoire. Ensuite, le temps d'exécution escompté est calculé de manière classique en divisant le temps théorique par la quote-part dont disposera

le programme. De la même façon, le temps de transfert d'un message est donné par l'addition de la latence constatée avec la division de la taille du message par la bande passante mesurée.

4. L'interface de FAST

Cette section va nous permettre de détailler l'interface offerte par FAST. En réalité, cette interface est double : une API permet à l'application appelante d'obtenir les informations dont elle a besoin tandis qu'un système de configuration par fichiers XML permet d'étendre FAST pour lui permettre de prédire les besoins de nouvelles fonctions. Les deux sections suivantes détaillent chacune de ces interfaces.

4.1. *L'API de* FAST

L'API de FAST comporte plusieurs catégories de fonctions. Certaines permettent d'obtenir le détail des prédictions sur les besoins et les disponibilités du système. D'autres combinent automatiquement ces résultats en valeurs directement utilisables par un ordonnanceur. Toutes les fonctions FAST renvoient un booléen indiquant si l'opération s'est déroulée sans problème. L'API résultante a été conçue de façon à masquer les complexités sous-jacentes à l'application appelante.

Tout d'abord, le système doit être initialisé :

```
fast_init(hôte_LDAP,racine_LDAP, ns_NWS,pred_NWS)
```

Cette fonction indique à FAST comment localiser les informations nécessaires. La base de données LDAP se trouve sur la machine `hôte_LDAP`, à l'emplacement `racine_LDAP`. Le serveur de nom et le prédicteur NWS se trouvent respectivement sur les hôtes `ns_NWS` et `pred_NWS`.

Une fois cette initialisation réalisée, l'ordonnanceur n'a besoin que de deux fonctions FAST pour obtenir les informations doit il a besoin :

```
fast_comm_time(source,dest,taille_msg,valeur)
```

indique le temps nécessaire à l'acheminement de données de taille `taille_msg` de la machine `source` à la machine `dest`, en tenant compte de la charge réseau au moment où l'appel est fait. Le résultat (en secondes) est rangé dans `valeur`.

```
fast_comp_time(hôte,fonction,data_desc,valeur)
```

permet quant à elle d'obtenir le temps de calcul prévu sur la machine `hôte` pour `fonction` selon la valeur des paramètres `data_desc` (dont le format sera détaillé à la fin de cette section) en tenant compte de la charge actuelle de la machine. La fonction renvoie un code d'erreur si la machine choisie est incapable de faire le calcul en question, soit parce qu'elle ne dispose pas d'assez de mémoire, soit parce que la routine correspondante n'est pas installée.

Si le programme appelant désire connaître les besoins des routines et les disponibilités, comme c'est le cas pour un outil de visualisation, ou pour calculer l'agrégation d'une autre manière, une interface de plus bas niveau est également disponible.

`fast_need(ressource,hôte,fonction,data_desc,valeur)`
range dans `valeur` la quantité de `ressource` (*temps* ou *espace mémoire*) nécessaire à l'exécution de `fonction` sur la machine `hôte` pour les paramètres `data_desc`.

`fast_avail(ressource,hôte1,hôte2,valeur)`
permet quant à elle d'obtenir la `valeur` actuellement disponible de `ressource`. Il peut s'agir de la charge CPU, de la quote-part de CPU dont disposerait un nouveau processus [4], de la quantité de mémoire vive disponible, de l'espace disque libre, ou bien de la latence ou de la bande passante du réseau entre deux machines. Le paramètre `hôte2` est destiné aux ressources réseaux, et il est ignoré pour les ressources ne concernant qu'une seule machine.

FAST regroupe la description des arguments passés aux fonctions au sein d'un tableau nommé `data_desc`. Chaque paramètre peut être un scalaire, un vecteur ou une matrice tandis que ses éléments de base sont de type entier, double ou caractère. La structure de données contient des informations sur l'objet qui doivent permettre de prédire le temps de calcul des fonctions. Dans le cas d'un scalaire, on stocke la valeur de l'objet elle même. Dans le cas d'un vecteur ou d'une matrice en revanche, on ne garde que les caractéristiques structurelles telles que ses dimensions et sa forme (*i.e.*, si la matrice est triangulaire inférieure, supérieure, si elle est bande, *etc.*). Nous avons choisi de ne pas stocker la valeur d'une matrice dans cette structure de données car le temps de calcul d'une fonction dépend des informations structurelles et non des valeurs stockées dans la matrice.

Les paramètres décrits par `data_desc` représentent les paramètres logiques du problème et non les paramètres réels de l'appel à la fonction. Ainsi, dans le cas du produit matriciel, la fonction `dgemm` nécessite treize paramètres, qui sont des pointeurs vers les zones mémoires occupées par les matrices à multiplier ainsi que certaines grandeurs comme les dimensions de ces matrices. Étant donné que `dgemm` calcule C dans l'expression $C = \alpha\ op(A)\ op(B) + \beta\ C$ (avec $op(A) = A\ ou\ A^t$), FAST ne considère que les cinq paramètres α, β, A, B et C. Il est capable de faire la conversion entre ces deux visions grâce aux connaissances que lui apportent les fichiers de configurations présentés à la section suivante.

4.2. *Ajouter de nouvelles routines à* FAST

Pour être capable de prédire leurs besoins, FAST doit connaître les routines susceptibles d'être ordonnancées sur le système. En particulier, il faut lui décrire le protocole opératoire de l'étalonnage de la fonction, et les paramètres attendus par la routine. Toutes les informations nécessaires sont regroupées dans un fichier XML, dont la figure 3 donne le squelette.

---

4. Cette grandeur n'est pas triviale à déduire depuis la charge CPU du fait du mécanisme de priorité sous UNIX. Une machine très chargée par des processus de calcul de priorité basse reste assez disponible pour de nouveaux processus de priorité plus haute.

```
<problemSet name="...">
    <problem name="...">
        <description>
            <descInput> ... </descInput>
            <descOutput>... </descOutput>
            <descCode>  ... </descCode>
        </description>

        <implementation>
            ...
        </implementation>

    </problem>
</problemSet>
```

```
<implementation ...>
    <implInput>    ... </implInput>

    <implCode>
        <implDecl> ... </implDecl>
        <benchBody>... </benchBody>
        <implBody> ... </implBody>
    </implCode>

</implementation>
```

Figure 3. *Squelette du fichier XML décrivant un problème à* FAST

Un tel fichier contient un ou plusieurs ensembles de problèmes (noté `<problemSet>`). Chacun d'entre eux contient une partie `<general>`, permettant de donner des informations sur la bibliothèque correspondant à cet ensemble de problèmes, et plusieurs parties `<problem>` pour décrire les différentes fonctions de la bibliothèque. Pour chaque fonction, une partie `<description>` présente une description de haut niveau du problème résolu par la fonction, tandis qu'une partie `<implementation>` donne des informations de plus bas niveau.

Nous allons maintenant étudier l'exemple de la fonction de produit matriciel `dgemm`, de la bibliothèque BLAS. La figure 4 donne la description de haut niveau de cette fonction, tandis que la figure 5 présente les informations relatives à l'implémentation de la fonction.

Comme le montre la figure 4, la partie **<description>** se découpe en plusieurs parties :

– **<descInput>** et **<descOutput>** contiennent respectivement la liste des paramètres et des résultats du problème. Ces paramètres sont identifiés de manière unique par leur nom. Dans le cas de `dgemm`, C étant à la fois un argument en entrée et en sortie, il est présent dans les deux listes.

– **<descCode>** explicite le problème résolu par la fonction sous une forme facilement manipulable par un programme. Dans ce cas, il montre simplement que `dgemm` résout le problème suivant : $C \leftarrow \alpha\ op(A)\ op(B) + \beta C$, avec $op(A) = A$ ou A^t, selon les paramètres d'appel. Notons que la partie `<descCode>` est ignorée par FAST. En effet, elle est plus destinée à un outil capable de choisir parmi les implémentations du même problème décrites dans des fichiers de description séparés. SLiM (Scientific LIbrary Metaserver), actuellement en développement, pourra par exemple sélectionner la fonction `dgemm` pour répondre à une requête de multiplication impliquant deux matrices. Cette partie n'est donc pas indispensable lorsque l'on souhaite utiliser FAST sans SLiM.

La partie **<implementation>** fournit à FAST des informations relatives à l'implémentation de la fonction, comme les arguments de l'appel ou le code source permet-

```
<description>
  <descInput>
    <descMatrix   name="A"       baseType="double"/>
    <descMatrix   name="B"       baseType="double"/>
    <descMatrix   name="C"       baseType="double"/>
    <descScal     name="alpha"   baseType="double"/>
    <descScal     name="beta"    baseType="double"/>
  </descInput>

  <descOutput>
    <descMatrix   name="C"       baseType="double"/>
  </descOutput>

  <descCode>
    <add>
      <mult>
         <arg name="alpha"/>
         <may_transp><arg name="A"/><may_transp/>
         <may_transp><arg name="B"/><may_transp/>
      </mult>
      <mult>
         <arg name="beta"/>
         <arg name="C"/>
      </mult>
    </add>
  </descCode>
</description>
```

Figure 4. *Description de haut niveau de la fonction* `dgemm`

tant d'étalonner la fonction. Actuellement, ces extraits de code doivent être écrits en C, mais l'argument `language` de cette balise permettra à l'avenir de préciser le langage de programmation utilisé. la figure 5 montre comment cette partie s'articule :

– **<implInput>** décrit les différents paramètres de la fonction, leurs liens avec les paramètres de haut niveau décrits par la partie `<description>`, ainsi que le protocole d'étalonnage les concernant. Chacun d'entre eux a un nom unique et est décrit par plusieurs balises :

- **<impl*>** indique le type de l'argument et contient toutes les autres balises à son sujet. Il peut s'agir de `<implChar>`, `<implInt>`, `<implDouble>`, `<implMat>` ou `<implVect>`.

- **<implConvert func=**$\cdots$ **data=**$\cdots$**>** indique comment effectuer la conversion depuis l'argument `data` décrit dans la partie `<description>` correspondant. `func` donne la fonction à utiliser pour convertir l'un en l'autre.

Les fonctions existantes sont `MATcolumns` ou `MATrows` qui donnent respectivement le nombre de colonnes et de lignes d'une matrice. `BLAScolumns` ou `BLASrows` donnent la même information, mais en l'inversant si la matrice est transposée. Ainsi, `BLAScolumns` donne le nombre de lignes si la matrice est transposée, et le nombre de colonnes dans le cas contraire.

`BLAScoef` peut prendre trois valeurs. Si le paramètre de description vaut 0 ou 1, le paramètre d'implémentation prend la même valeur. Dans tous les autres cas, il prend la valeur 2. Cette fonction est destinée aux coefficients multiplicatifs comme α

```
<implementation language="C">
  <implInput>
    <implChar   name="TransA">
      <implConvert func="BLAStrans" data="A"/>
      <benchChar>nt</benchChar>
    </implChar>
    <implChar   name="TransB">...
    <implInt    name="M">
      <implConvert func="BLASlines" data="A"/>
      <benchNum min="128" max="1152" incr="128"/>
    </implInt>
    <implInt    name="N">...
    <implInt    name="K">...
    <implDouble name="alpha">
      <implConvert func="BLAScoef"  data="alpha"/>
      <benchBLAScoef/>
    </implDouble>
    <implMat    name="A">
      <implConvert data="A"/>
    </implMat>
    <implDouble name="beta">...
    <implMat    name="B">...
  </implInput>

  <implCode>
    <implDecl>
      int LDA,LDB;
    </implDecl>
    <benchBody>
      ... A=malloc(...);memset(A,2,... ); ...
    </benchBody>
    <implBody>
      dgemm_(&TransA,&TransB,&M,&N,&K,&alpha,A,&LDA,B,&LDB,&beta,C,&M);
    </implBody>
  </implCode>
</implementation>
```

Figure 5. *Description de l'implémentation de la fonction* `dgemm`

et β pour dgemm. En effet, les implémentations des BLAS ne font pas le produit de matrice AB si α est nul, et ne multiplie pas la matrice résultat par le coefficient β si ce dernier vaut 1. Dans tous les autres cas, les deux opérations doivent être réalisées. De même, si $\beta = 0$, l'opération effectuée est $C = \alpha AB$.

Si la fonction de conversion est omise, comme c'est le cas pour la matrice A dans cet exemple, cela signifie que la valeur du paramètre de description doit être utilisée comme paramètre d'implémentation sans modification.

- **<bench*>** précise le comportement de ce paramètre lors de l'étalonnage de la routine.

`<benchChar>` énumère les valeurs que l'argument doit prendre. Dans ce cas, `TransA` doit valoir « n » dans une première série de test pour indiquer que la matrice A ne doit pas être transposée, puis « t » dans une autre pour refaire les tests en demandant à transposer la matrice A.

`<benchNum min=··· incr=··· max=···>` permet de spécifier qu'un argument doit prendre toutes les valeurs entre `min` et `max`, avec un pas de `incr`.

Enfin, `<benchBLAScoef>` indique que le paramètre doit couvrir toutes les valeurs significatives d'un coefficient multiplicateur dans les BLAS. Comme nous l'avons vu au paragraphe précédent, il s'agit de 0, 1 ou une valeur arbitraire.

– **<implCode>** contient les extraits de code à utiliser pour appeler cette fonction. Cette partie se découpe à son tour en trois sous–parties :

- **<implDecl>** comporte les déclarations de variables à ajouter au code source généré en plus de celles correspondant aux paramètres d'implémentation.

- **<benchBody>** n'est utilisé que lors de la génération du code chargé d'étalonner cette fonction. Cette balise est typiquement chargée d'initialiser les données avec des valeurs arbitraires en évitant les cas particuliers qui fausseraient l'étalonnage. Cette phase n'est pas nécessaire lorsque l'on veut utiliser la fonction en conditions réelles comme sur un serveur de calcul puisque les données sont alors fournies par le client.

- **<implBody>** contient quant à elle l'appel à la fonction proprement dit.

5. Expérimentations

Après avoir étudié le fonctionnement et l'interface de FAST, nous allons maintenant présenter quelques résultats d'expérimentations. La première série, constituant la section 5.1, porte sur les améliorations apportées à NWS. La partie 5.2 présente la qualité des prédictions de FAST. Enfin, la section 5.3 montre les résultats de l'intégration de FAST dans d'autres projets, tels que NetSolve ou un outil de visualisation en temps réel Cichlid.

5.1. *Contributions à NWS*

5.1.1. *Amélioration du temps de réponse*

Comme nous l'avons souligné à la section 3.2, nous avons corrigé dans FAST certaines faiblesses de NWS. Le premier point que nous voulions améliorer était le temps de réponse du système grâce à divers mécanismes de caches de valeurs. Pour prouver l'efficacité de nos changements, nous avons mesuré le temps de réponse moyen de requêtes simples lorsque tous les éléments NWS et le client se trouvent sur la même machine. La machine utilisée est une station de travail dotée d'un Pentium II sous Linux nommée *pixies*. La figure 6 montre les résultats obtenus. La première mesure (6a) concerne une interrogation directe de NWS sans utiliser FAST, tandis que la seconde et la troisième (6b et 6c) portent sur la même interrogation, mais en utilisant cette fois FAST. Pour (6b), la valeur cherchée n'est pas présente dans le cache, et FAST doit donc interroger NWS après avoir constaté ce défaut de cache alors que pour (6c), la valeur cherchée se trouve dans le cache, et FAST évite donc toute interrogation à NWS. Les valeurs données sont des moyennes sur 30 exécutions.

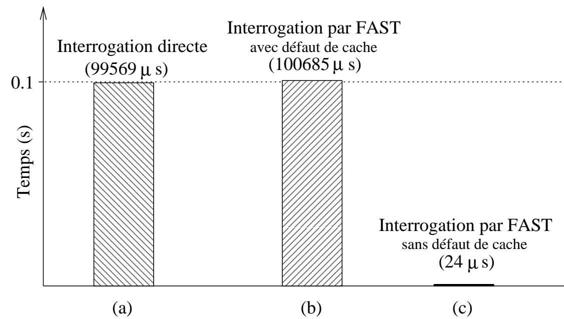


Figure 6. *Temps de réponse NWS et* FAST

On constate que l'usage de FAST présente un surcoût négligeable. En revanche, lorsque la donnée est présente dans le cache, le gain obtenu est considérable. Cela est évidement fort intéressant puisqu'un ordonnanceur demandera ces valeurs pour toutes les machines disponibles chaque fois qu'une requête d'ordonnancement lui sera soumise. En condition réelle, FAST pourra donc être amené à traiter la même requête plusieurs fois par seconde.

5.1.2. *Amélioration de la réactivité*

Le mécanisme utilisé par NWS pour prédire les évolutions nous a semblé manquer de réactivité face aux changements de conditions tels que le début ou la fin d'un processus de calcul. Nous avons donc modifié NWS pour qu'il puisse s'adapter à de tels changements.

Nous présentons maintenant l'expérience que nous avons mise au point pour montrer l'efficacité de ces changements. La plate-forme de test est composée d'une seule machine (*pixies*, décrite plus haut) sur laquelle s'exécutent en parallèle deux systèmes NWS. Le premier est une version non modifiée tandis que le second est capable d'être tenu au courant des changement de conditions. Au début de l'expérience, la machine est laissée suffisamment longtemps sans charge pour que les deux systèmes NWS s'équilibrent. Ensuite, une tâche utilisant l'intégralité du processeur disponible est lancée et dure une minute. Le système NWS modifié est prévenu du lancement et de l'arrêt de cette tâche. Pendant toute l'expérience, les deux systèmes NWS mesurent périodiquement la quote-part de processeur dont disposerait un processus arrivant sur la machine. La valeur théorique est 1 (*i.e.*, 100 %) lorsque la machine est au repos, et 0.5 (*i.e.*, 50 %) lorsque la tâche s'exécute, puisque le second processus devrait partager le processeur avec le premier.

La figure 7 montre les mesures fournies pas les deux systèmes NWS. On constate que le système NWS ne réagit que très lentement au lancement ou à l'arrêt de la tâche, puisque juste avant la fin de la tâche, il indique que 7 % du processeur est utilisé alors que la valeur théorique est 50 %. De la même façon, lorsque la tâche s'interrompt, cette version a besoin d'une minute et demi pour retourner à l'équilibre. En utilisant

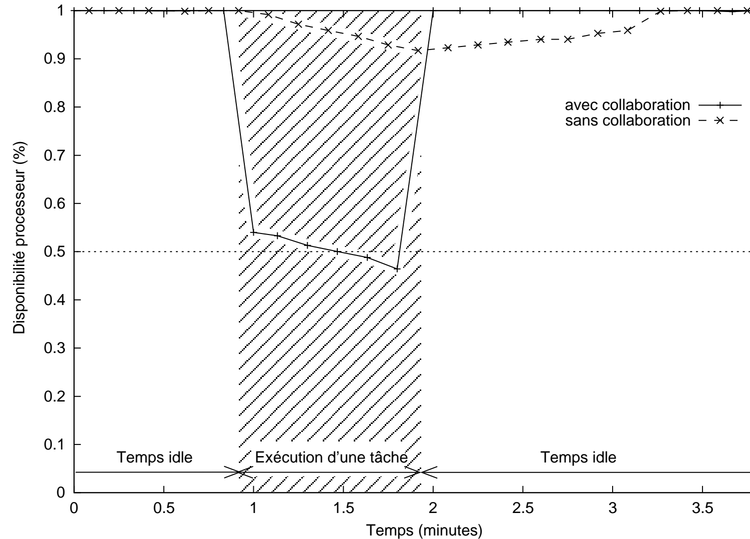


Figure 7. *Amélioration de NWS : collaboration avec la plate-forme*

nos modifications pour informer NWS des changements, on lui permet de réagir dès la mesure suivante (soit en moins de dix secondes), et d'estimer la disponibilité à 5 % près.

5.2. *Qualité des résultats*

Afin de tester la validité de l'approche choisie, nous avons mis au point une série d'expériences pour étudier la qualité des résultats renvoyés par FAST. La première expérience ne porte que sur la modélisation des besoins (en temps et en espace) des routines sans tenir compte de la charge. La seconde traite de la prédiction du temps de calcul d'opérations de base en tenant compte de la charge. Enfin, la dernière expérience porte sur la prédiction du temps de calcul d'une séquence d'opérations sur une plate-forme hétérogène.

5.2.1. *Modélisation des besoins*

Avant de mesurer la qualité des prédictions de FAST, il nous a semblé important de mesurer la qualité de la modélisation utilisée pour les besoins des routines sans tenir compte de la charge. Pour cela, nous avons comparé la valeur modélisée à la valeur mesurée pour le temps et l'espace nécessaires aux trois fonctions BLAS `dgeadd` (addition de matrices), `dgemm` (produit matriciel) et `dtrsm` (résolution triangulaire). Nous avons réalisé cette comparaison pour deux machines différentes : *icluster* et *paraski*. La première une grappe de 225 PC installée à l'IMAG à Grenoble. Il s'agit de Pentiums III disposant de 256 Mo de mémoire vive. *Paraski* est la grappe de l'IRISA

	Opération dgeadd		Opération dgemm		Opération dtrsm	
	icluster	*paraski*	*icluster*	*paraski*	*icluster*	*paraski*
Erreur Maximale	0.02s (6 %)	0.02s (35 %)	0.21s (0.3 %)	5.8s (4 %)	0.13s (10 %)	0.31s (16 %)
Erreur Moyenne	0.006s (4 %)	0.007s (6.5 %)	0.025s (0.1 %)	0.03s (0.1 %)	0.02s (5 %)	0.08s (7 %)

Tableau 1. *Qualité de la modélisation temporelle de dgeadd, dgemm et dtrsm*

située à Rennes. Chacun des 40 nœuds est doté de deux processeurs Pentium II et de 256 Mo de mémoire vive. Toutes ces machines utilisent Linux.

Le tableau 1 présente la qualité de la modélisation temporelle de ces fonctions sur les nœuds de ces grappes pour des tailles de matrices variant entre 128 et 1152. Sur la première ligne figure l'erreur maximale (en valeur absolue) constatée lors de l'expérience. L'erreur relative est indiquée entre parenthèses. La seconde ligne donne la moyenne de l'erreur, à la fois en valeur absolue et en valeur relative. On constate que l'erreur maximale commise est généralement inférieure à 0,2 secondes. L'erreur moyenne est quant à elle de l'ordre du centième de seconde. Cette précision est donc tout à fait acceptable dans ce contexte, puisque nous avons vu lors des expériences menées au paragraphe 5.1 que le temps de réponse de NWS est de l'ordre du dixième de seconde. L'erreur relative est quant à elle d'un peu plus de 5 % en moyenne, et d'un peu plus de 15 % dans le pire des cas.

Dans le cas de l'addition, les erreurs relatives sont plus importantes. Ceci s'explique par le fait que nous avons choisi d'effectuer les mesures dans FAST avec la fonction `rusage`, qui donne le temps système et utilisateur du processus en cours. Cette méthode est insensible à la charge externe, mais sa précision n'est que de 0,01 secondes. L'addition de matrice durant moins d'une demi seconde, l'erreur noté sur ce tableau n'est donc pas seulement l'erreur de modélisation, mais regroupe également des erreurs de mesure.

La modélisation spatiale n'est pas représentée car FAST parvient à modéliser parfaitement (*i.e.*, erreur maximale de 0 octet) l'espace nécessaire à l'exécution des routines en fonction des paramètres. Cette qualité est possible car la taille d'un programme effectuant une opération matricielle correspond à la taille des données, plus une constante pour le code du programme. Le tout est donc facilement exprimable sous forme d'une fonction polynomiale dépendant de la taille des matrices.

5.2.2. *Prédiction du temps de calcul d'une opération*

Le but de cette expérience est de mesurer la qualité des prédictions de FAST en tenant compte de la charge. Cette expérience ne porte que sur le temps d'exécution, car la charge extérieure n'a aucune influence sur la quantité de mémoire nécessaire à l'exécution d'une routine. Nous avons comparé la valeur mesurée à la prédiction

pour le temps d'exécution de la routine `dgemm` par FAST. Pour simuler une charge extérieure au système, nous avons exécuté un programme de calcul lors du déroulement de l'expérience. La plate-forme de test utilisée est la même que pour l'expérience précédente : nous avons utilisé un nœud de *paraski*, et un nœud de l'*icluster*. Les résultats présentés sont des moyennes sur cinq exécutions.

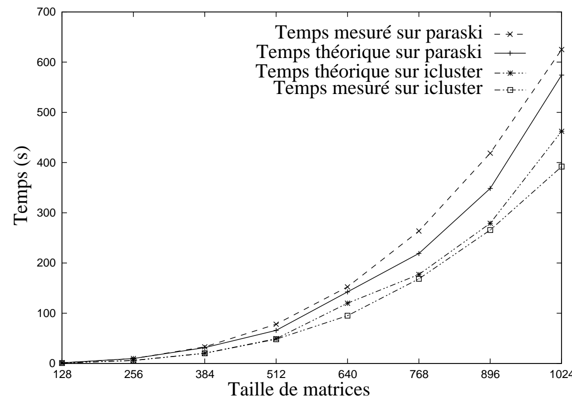


Figure 8. *Comparaison du temps réel et de la prédiction pour une opération (dgemm)*

La figure 8 montre la comparaison les résultats de cette expérience. Malgré la charge externe, FAST parvient à prédire le temps de calcul avec une erreur maximale de 22 %, et une erreur moyenne inférieure à 10 %.

5.2.3. *Prédiction du temps de calcul d'une séquence d'opérations*

L'opération choisie est le produit de matrices complexes. Les données de départ sont deux matrices complexes A et B, séparées en partie réelle et partie imaginaire ($A = (A_r, A_i)$ et $B = (B_r, B_i)$). L'objectif est de calculer la matrice C résultant du produit de A et de B. Le problème à résoudre est donc :

$$C = \begin{cases} C_r = A_r \times B_r - A_i \times B_i \\ C_i = A_r \times B_i + A_i \times B_r \end{cases}$$

Comme l'objectif était de mesurer la précision de la prédiction et non son impact sur les décisions d'ordonnancement, nous avons mis en place une plate-forme d'évaluation hétérogène composée de deux machines. *Pixies* est une station de travail dotée d'un Pentium II et de 128 Mo de mémoire tandis que *Kwad* est un serveur de calcul SMP doté de quatre processeurs Pentium III et de 256 Mo de mémoire. Les deux machines fonctionnent sous Linux. Lors de cette expérience, *Pixies* est dotée d'un processus client et *Kwad* de deux serveurs. Le client interroge FAST pour obtenir une prédiction de temps de calcul de la séquence, puis initie le calcul et mesure le temps réel.

La figure 9 montre la répartition (statique) des sous–calculs entre les machines. La figure 10 montre quant à elle la comparaison entre les temps de calcul prédits et

– Sur le client (*Pixies*) :

- Envoi de A_r, A_i et B_r au serveur 1 ;
- Envoi de A_r, A_i et B_i au serveur 2.

– Sur le serveur 1 (*Kwad*) :

- $C_{r_1} = A_r \times B_r$;
- $C_{i_2} = A_i \times B_r$;
- Envoi de C_{i_2} au serveur 2 ;
- $C_r = C_{r_1} - C_{r_2}$;
- Envoi de C_r au client.

– Sur le serveur 2 (*Kwad*) :

- $C_{r_2} = A_i \times B_i$;
- $C_{i_1} = A_r \times B_i$;
- Envoi de C_{r_2} au serveur 1 ;
- $C_i = C_{i_1} + C_{i_2}$;
- Envoi de C_i au client.

Figure 9. *Répartition des sous–tâches du produit de matrices complexes*

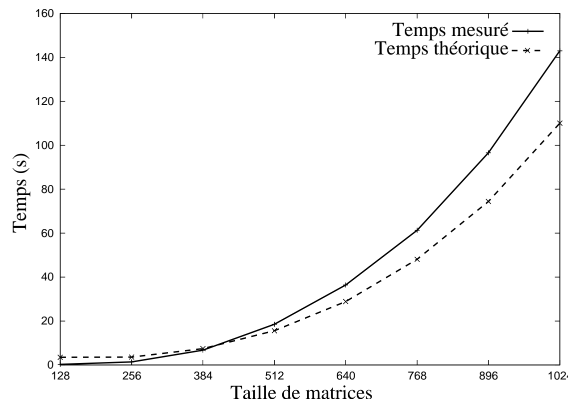


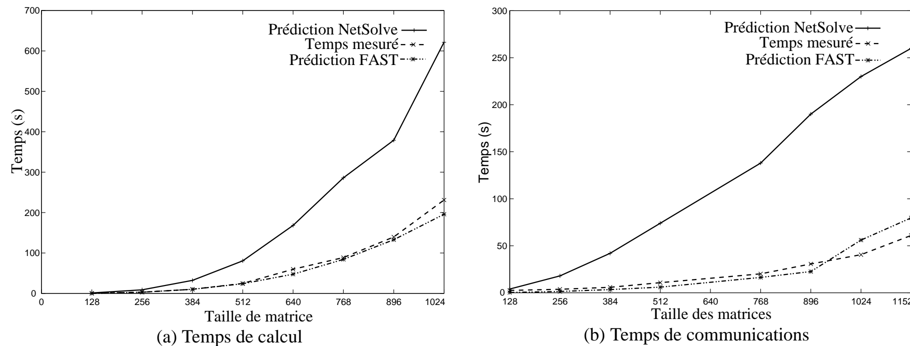
Figure 10. *Comparaison du temps réel et de la prédiction pour une séquence d'opérations*

réels pour l'ensemble de la séquence composant le produit de matrices complexes en fonction de la taille des matrices. On constate que la prédiction et la valeur réelle varient de la même façon. Malgré le fait que la séquence soit composée de six calculs de deux types différents et de six échanges de matrices, FAST parvient à prédire le temps nécessaire avec moins de 25 % d'erreur dans le pire des cas, et 12 % d'erreur en moyenne.

5.3. *Intégration de* FAST *à d'autres projets*

5.3.1. *NetSolve*

Le but de cette expérience est de tester les résultats de FAST en conditions réelles. Pour cela, nous avons modifié l'ordonnanceur de NetSolve (décrit dans la section 2.3.4) pour utiliser FAST. Cela nous a permis de comparer les résultats donnés par notre bibliothèque à ceux de la version 1.3 de NetSolve. Une fois encore, la plate-forme de



(a) Temps de calcul (b) Temps de communications

Figure 11. *Comparaison des prédictions de NetSolve et de* FAST *aux temps mesurés pour* `dgemm`

test utilisée est relativement simple car nous ne voulons comparer que les prédictions, et non leur influence sur la qualité de l'ordonnancement. Nous avons donc utilisé un unique serveur placé sur un des nœuds de *paraski*, tandis que le client était placé sur l'un des nœuds de l'*icluster*.

La figure 11 présente les résultats de cette expérience. La partie (11a) compare les prédictions en termes de temps de calcul de NetSolve et de FAST au temps mesuré, tandis que la partie (11b) compare les temps de communications. Ces résultats sont très encourageants. La partie (11a) montre clairement les avantages à utiliser une estimation des performances ne dépendant pas seulement du problème à traiter, mais aussi de la machine sur laquelle on l'exécute. La partie (11b) démontre quant à elle les bénéfices potentiels d'un outil de surveillance du réseau spécialisé comme NWS.

5.3.2. *Un outil de visualisation*

Afin de pouvoir surveiller simplement le fonctionnement de la plate-forme, nous avons réalisé un outil de visualisation en temps réel. Les valeurs sont bien entendu données par des appels à FAST. L'affichage en réalité virtuelle est géré par la bibliothèque OpenGL. Notre travail est basé sur la bibliothèque d'affichage de graphiques en trois dimensions Cichlid [5], mise au point par le National Laboratory for Applied Network Research (NLANR) à l'Université de Californie, San Diego. Un tel outil permet en outre de naviguer dans la représentation de la plate-forme afin de zoomer sur certaines parties du réseau pour un examen plus approfondi, ou au contraire d'avoir une vue d'ensemble.

5. `http://moat.nlanr.net/Software/Cichlid/`

6. Conclusion et travaux futurs

Dans cet article, nous avons présenté FAST, un outil de modélisation et de prédiction de performances dans un contexte de metacomputing. Après un état de l'art du domaine, nous avons étudié le fonctionnement de cet outil ainsi que l'interface offerte. Nous avons également présenté divers résultats expérimentaux. Les premiers concernent des améliorations apportées aux outils de base tels que NWS. D'autres portent sur la qualité des prédictions pour des opérations simples ou pour des séquences d'opérations.

L'implémentation présentée est pleinement fonctionnelle. Nous l'avons intégrée avec succès à NetSolve, et son intégration à l'environnement de metacomputing DIET est en cours. Nous avons de plus mis au point un outil de visualisation de l'état du système en temps réel.

Mais cette version souffre encore de certaines limitations que nous projetons de corriger dans un avenir proche. Il faudra pour cela permettre à FAST de traiter des routines autres que celles simplement chronométrables. Dans ce cadre, le support des fonctions parallèles régulières est en cours. Une adaptation de FAST aux systèmes d'-exploitations Irix et AIX est également prévue. L'utilisation de plusieurs bibliothèques pour l'obtention des informations sur les disponibilités de la plate-forme simplifierait les pré–requis à l'installation de FAST tout en permettant de comparer ces systèmes entre eux. Nous projetons également d'étendre NWS pour lui permettre de surveiller des réseaux autres que ceux dotés du protocole TCP/IP, par exemple de type Myrinet ou SCI. Enfin, nous avons établi des contacts avec les équipes de NetSolve et NWS pour intégrer nos travaux à la version officielle de ces projets.

Remerciements

Je tiens à remercier le laboratoire Id de l'IMAG et le projet Paris de l'IRISA pour m'avoir ouvert l'accès à leurs ressources de calculs afin de mener les expérimentations présentées dans cet article. Je voudrais aussi remercier les membres des projets NWS et NetSolve pour avoir rendu publics leurs logiciels, qui constituent une base d'expérimentation inespérée, ainsi que pour leur collaboration amicale.

7. Bibliographie

[AID 00] AIDA K., TAKEFUSA A., NAKADA H., MATSUOKA S., SEKIGUCHI S., NAGASHIMA U., « Performance evaluation model for scheduling in a global computing system », *International Journal of High-Performance Computing Applications*, vol. 14, n° 3, 2000, p. 268-279, `http://www.alab.dis.titech.ac.jp/reports/ijhpca00.ps.gz`.

[ARB 96] ARBENZ P., GANDER W., OETTLI M., « The Remote Computation System », LIDDELL H., COLBROOK A., HERTZBERGER B., SLOOT P., Eds., *High Performance Computing and Networking*, vol. 1067 de *Lecture Notes in Computer Science*, p. 820-825, Springer-Verlag, Berlin, 1996.

[BER 97] BERMAN F., WOLSKI R., « The AppLeS Project : A Status Report », *8th NEC Research Symposium, Berlin, Germany*, vol. 16, 1997, `http://www.cs.ucsd.edu/groups/hpcl/apples/pubs/nec97.ps`.

[CAR 00] CARON E., LAZURE D., UTARD G., « Performance Prediction and Analysis of Parallel Out-of-Core Matrix Factorization », *Proceedings of the 7th International Conference on High Performance Computing (HiPC'00)*, Dec 2000.

[CAS 97] CASANOVA H., DONGARRA J., « NetSolve : A Network-Enabled Server for Solving Computational Science Problems », *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, n° 3, 1997, p. 212–223.

[FOS 97] FOSTER I., KESSELMAN C., « Globus : A Metacomputing Infrastructure Toolkit », *International Journal of Supercomputing Applications*, vol. 11, n° 2, 1997, p. 115-128.

[FOS 98] FOSTER I., KESSELMAN C., « The Globus Project : A Status Report », *Proceedings of the Heterogeneous Computing Workshop*, 1998, p. 4–18.

[HOW 99] HOWES I. A., SMITH M. C., GOOD G. S., *Understanding and deploying LDAP directory services*, Macmillian Technical Publishing, 1999, ISBN : 1-57870-070-1.

[HP 95] H-P, « NetPerf : A Network Performance Benchmark », 1995, Hewlett-Packard Company.

[JAC 96] JACOB B. L., CHEN P. M., SILVERMAN S. R., MUDGE T. N., « An Analytical Model for Designing Memory Hierarchies », *IEEE Transactions on Computers*, vol. 45, n° 10, 1996, p. 1180-1194.

[MAT 00] MATTHEWS W., COTTRELL L., « The PingER Project : Active Internet Performance Monitoring for the HENP Community », *IEEE Communications Magazine*, vol. 38, n° 5, 2000, http ://www-iepm.slac.stanford.edu/paperwork/ieee/ieee.ps.gz.

[SCH 97] SCHOPF J., BERMAN F., « Performance prediction using intervals », rapport n° CS97-541, 1997, University of California San Diego, CSE Department.

[SGI 01] SGI, « Performance Co-Pilot : Monitoring and Managing System-Level Performance », `http://www.sgi.com/software/pcp/`, 2001.

[WEI 00] WEIKLE D. A. B., SKADRON K., MCKEE S. A., WULF W. A., « Caches As Filters : A Unifying Model for Memory Hierarchy Analysis », rapport n° CS-2000-16, 1 2000, University of Virginia.

[WOL 98] WOLSKI R., SPRING N., HAYES J., « Predicting the cpu availability of time-shared unix systems », rapport n° CS98-602, 1998, University of California San Diego.

[WOL 99] WOLSKI R., SPRING N. T., HAYES J., « The Network Weather Service : A Distributed Resource Performance Forecasting Service for Metacomputing », *Future Generation Computing Systems, Metacomputing Issue*, vol. 15, n° 5–6, 1999, p. 757–768.

Article reçu le 11 septembre 2001
Version révisée le 3 janvier 2002
Rédacteur responsable : Christophe Cérin

Martin Quinson *est actuellement allocataire de recherche et prépare une thèse au sein de l'équipe ReMaP du Laboratoire de l'Informatique Parallèle (LIP) de l'École Normale Supérieure*

de Lyon sous la direction de Frédéric Desprez. Ses travaux portent sur la prédiction de performances, l'ordonnancement et le déploiement d'une plate-forme de metacomputing.