Découverte automatique des caractéristiques et capacités d'une plate-forme de calcul distribué

Martin Quinson

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

présentée par

Martin Quinson

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique

au titre de l'École doctorale de Mathématiques et Informatique fondamentale

Découverte automatique des caractéristiques et capacités d'une plate-forme de calcul distribué

Date de soutenance : 11 décembre 2003

Composition du Jury :

Directeurs :	Eddy Caron,	ÉNS-Lyon.
	Frédéric Desprez,	ÉNS-Lyon.
Rapporteurs :	Franck Cappello,	Université de Paris-Sud.
	Hervé Guyennet,	Université de Besançon.
Examinateurs :	Jacques Chassin de Kergommeaux,	ENSIMAG, Grenoble.
	Richard Wolski,	Université de Californie de Santa Barbara, USA.

Remerciements

La chose la plus étrange de ce document est sans doute que seul mon nom soit indiqué comme auteur. Comment aurais-je pu en arriver là sans la présence, l'assistance et le soutien dont j'ai bénéficié toutes ces années ? Il est plus que temps de remercier en bonnes et dues formes tous ceux sans qui cette thèse n'aurait pas été ce qu'elle est.

Je voudrais naturellement remercier tout d'abord Armelle, mon grand amour. Je ne serais sans doute arrivé à rien de bon sans notre complicité inouïe ni son soutien sans faille (et plein d'autres choses qu'on ne peut pas décemment dire en public). Son bon sens et sa tête bien sur les épaules sont vraiment précieux dans mes moments de doute. Merci de me laisser rêver en me permettant de réaliser mes rêves. Merci également à mon ptit bonhomme, dont l'affection vaut bien toutes les drogues du monde. Il est bien trop petit pour en être conscient, mais ça n'en est que meilleur...

Je souhaite également remercier tous ceux qui ont su guider mes premiers pas dans le pays joyeux des chercheurs heureux, des reviews gentilles et des bien belles publis. Merci à FredD pour avoir toujours cru en moi, pour avoir composé avec ma tête de lard et pour être parvenu (presque malgré moi) à m'apprendre le travail en équipe (et non l'autonomie ;). Merci également à Eddy pour sa disponibilité, son humour et sa gentillesse tout simplement hallucinants. Merci aussi à Yves pour sa bonne humeur contagieuse, ses réponses à toutes mes questions « hautement scientifiques » et ses grosses blagues recelant si souvent de judicieux conseils. Merci encore à FredS pour m'avoir enseigné la rigueur. Merci aussi à Henri pour NetSolve, qui fut un merveilleux terrain de jeux (tout en laissant la place à des améliorations ;), pour SimGrid qui rend GRAS possible, ainsi que pour nos discussions sur la découverte de la topologie et ses relectures anglophones. Merci à Rich (par delà la barrière des langues) pour sa rigueur théorique alliée à son sens pratique qui font de NWS un bien bel outil avec lequel il a la gentillesse de me laisser jouer en m'accueillant en Californie bientôt.

Je me dois de remercier chaleureusement tous les membres de mon jury. Merci à Franck Cappello et Hervé Guyennet d'avoir accepté d'être mes rapporteurs et de leurs précieux avis sur la version préliminaire de ce manuscrit. Merci aussi à Jacques Chassin de Kergommeaux d'avoir accepté d'être président du jury, et à Rich Wolski d'avoir fait 30 heures d'avion pour assister à ma soutenance bien qu'elle ait eu lieu en français. Merci encore une fois à Fred et Eddy d'avoir encadré ma thèse de la sorte.

Quant à Arnaud, j'aimerais le remercier de son amitié, de sa patience devant mes lacunes mathématiques, de son style de programmation si « particulier » mais toujours efficace, de m'avoir parfois laissé gagner à frozen, et aussi de faire plein de ptits copains à Tristan... Ces quelques années n'étaient (je l'espère) que le commencement d'une longue collaboration.

Il me faut également remercier mes parents et ma famille de m'avoir toujours soutenu, même en DEUG (« Diplôme ? Plome ! »). Je ne peux pas remercier suffisamment tout le monde pour les bonnes raisons ici, mais je me dois d'évoquer François, qui m'a donné le virus de l'informatique (satané TO7 qui se croyait le droit de me dire `Ok`, et qui me traitait de `syntax error` quand je lui disais `Ok` à mon tour) et également le virus de la recherche (je sais, je ne fais que des sciences inhumaines, mais je me soigne). J'ai également une pensée particulière pour Claire et Marie-Laure, ne serait-ce que pour avoir relu ce manuscrit avec attention sans en comprendre un traître mot...

Je serais de plus un monstre d'ingratitude si je ne remerciais pas ici Abdou pour sa gentillesse désormais légendaire, Jean-Yves et Gaël pour m'avoir fait connaître la victoire en me prenant parfois dans leur équipe au baby, Alain de ne jamais s'être moqué de ma NP-naïveté, Oli pour sa patience devant mes lacunes en programmation, pour son aide dans GRAS et pour son amitié. Je lui pardonne même ses « bonnes blagues » de lyonnais... Merci encore à Olivier Beaumont de soutenir les verts à ma place quand je faiblis, à Vince de m'avoir initié aux joies masochistes de la programmation shell, à LN d'avoir joué son rôle de blonde avec tant de constance, à Antoine d'avoir présenté FAST pour moi de l'autre coté de la terre, à Loris de ses tests consciencieux de xbubble, et plus généralement à tous les membres du LIP pour l'ambiance qui règne dans ces lieux.

Merci à Sylvie, Anne-Pascale, Simone, Isabelle et Corinne, dont la compétence, la disponibilité et les sourires sont tout simplement renversants. Elles ont même la délicatesse de laisser croire à d'autres qu'ils ont les rênes de l'étage...

Merci aux laboratoires ID et IMAG de m'avoir prêté leurs gros ordinateurs pour que je fasse joujou. Et puisqu'il faut bien vivre, merci au ministère et à l'INRIA pour les sous-sous. J'ai également une pensée pour Monsieur Gramain, pour ce qu'il a fait pour moi en DEUG, pour Stéphane qui m'a poussé à m'inscrire à l'ENS et pour les membres de l'équipe EURISE qui m'ont donné mes premiers cours d'informatique en tolérant mes errances européennes.

Merci enfin à qui de droit (je suis sûr qu'il se reconnaîtra) et à qui vous savez (même s'il ne se reconnaît jamais).

Table des matières

Introduction générale 1

I Le *Metacomputing* : de la toile à la grille 3

1 Introduction 5
1.1 Présentation . . . 5
1.2 Caractéristiques de la grille . . . 6

2 Quelques environnements pour la grille 9
2.1 Les environnements d'utilisation de type GridRPC . . . 9
2.1.1 Problématiques . . . 10
2.1.2 Études de cas . . . 11
2.2 Infrastructures de fonctionnement . . . 17
2.2.1 Un environnement d'exécution : Globus . . . 18
2.2.2 Un outil de surveillance de la plate-forme : NWS . . . 20

II Prédictions de performance pour la grille 25

3 État de l'art et méthodologie 29
3.1 Acquisition des disponibilités du système . . . 29
3.1.1 Choix des métriques . . . 29
3.1.2 Mesures passives . . . 31
3.1.3 Mesures actives . . . 31
3.2 Caractériser les besoins des routines . . . 32
3.2.1 Décompte des opérations élémentaires . . . 34
3.2.2 Approche probabiliste et chaînes de Markov . . . 37
3.2.3 Le « macro-benchmarking » . . . 37
3.3 Résumé . . . 40

4 Outil de prédiction de performances 41
4.1 Présentation de FAST . . . 41
4.1.1 Surveillance des disponibilités de la plate-forme . . . 42
4.1.2 Étalonnage de routines . . . 45
4.1.3 Combiner ces informations . . . 46
4.2 Interface de programmation . . . 46

4.2.1 Interface de haut niveau . . . 46
4.2.2 Interface pour les réservations virtuelles . . . 47
4.2.3 Interface de bas niveau . . . 48
4.3 Validations expérimentales . . . 48
4.3.1 Amélioration au système de surveillance . . . 48
4.3.2 Qualité des prédictions . . . 53
4.4 Résumé . . . 56

5 Conclusion à propos de FAST 59
5.1 Intégration de FAST à d'autres projets . . . 59
5.1.1 NETSOLVE . . . 59
5.1.2 DIET . . . 60
5.1.3 Grid-TLSE . . . 61
5.1.4 Outil de visualisation . . . 61
5.2 Extension parallèle de FAST . . . 62
5.2.1 Présentation . . . 62
5.2.2 Exemple de modélisation du produit de matrices denses . . . 63
5.2.3 Validation expérimentale . . . 64
5.2.4 Discussion . . . 65
5.3 Résumé et travaux futurs . . . 66

III Découverte de topologie et déploiement automatique 69

6 Objectifs, état de l'art et méthodologie 73
6.1 Objectifs . . . 74
6.1.1 Topologie recherchée . . . 74
6.1.2 Caractérisation des contraintes de déploiement de NWS . . . 74
6.2 Outils et méthodes de découverte de la topologie . . . 76
6.2.1 SNMP et BGP . . . 76
6.2.2 Méthodes tomographiques . . . 77
6.2.3 Mesures passives . . . 79
6.3 Résumé et méthodologie retenue . . . 79

7 Expérience de déploiement de NWS grâce à ENV 81
7.1 Présentation d'ENV . . . 81
7.2 Exemple d'exécution d'ENV . . . 82
7.2.1 Topologie structurelle : cartographie ne dépendant pas du maître . . . 82
7.2.2 Topologie effective : cartographie dépendant du maître . . . 84
7.2.3 Améliorations apportées à ENV . . . 86
7.2.4 Cartographie obtenue . . . 88
7.3 Déploiement de NWS grâce à ENV . . . 88
7.4 Résumé et problèmes ouverts . . . 90

8 Un environnement de développement pour la grille 95
8.1 Présentation de GRAS et de ses objectifs . . . 96
8.1.1 Simulation efficace et transparente . . . 96

8.1.2 Simplicité d'usage . . . 98
8.2 État de l'art . . . 99
8.2.1 Émulation et simulation de grille . . . 99
8.2.2 Bibliothèques de communication . . . 100
8.3 Exemple : un outil de mesure des performances . . . 102
8.3.1 Description des données et des messages . . . 102
8.3.2 Coté client . . . 103
8.3.3 Coté senseur . . . 104
8.3.4 Déploiement de cet outil . . . 105
8.4 État actuel du prototype et travaux futurs . . . 105
8.5 Résumé . . . 108

9 Découverte de la topologie de la grille 109
9.1 Présentation d'ALNeM et de ses objectifs . . . 109
9.1.1 Modélisation . . . 110
9.1.2 Méthodologie de mesure . . . 111
9.1.3 Formalisation du problème . . . 113
9.2 Outils mathématiques . . . 113
9.2.1 Hypothèses . . . 113
9.2.2 Interférence totales et séparateurs . . . 114
9.3 Algorithme de reconstruction . . . 115
9.3.1 Traitement des arbres . . . 115
9.3.2 Traitement des cliques . . . 116
9.3.3 Traitement des cycles . . . 117
9.4 Collecte des informations nécessaires . . . 120
9.4.1 Algorithme intuitif . . . 120
9.4.2 Optimisations . . . 120
9.5 Exemple de fonctionnement . . . 121
9.6 Résumé et travaux futurs . . . 121

10 Conclusion et perspectives 125
10.1 Travaux présentés . . . 125
10.2 Résumé de nos contributions . . . 127
10.3 Perspectives . . . 127

A Démonstrations du chapitre 9 (ALNeM) 129
A.1 Lemme 9.3 (lemme de séparation) . . . 129
A.2 Théorème 9.1 (l'interférence totale est une relation d'équivalence) . . . 130
A.3 Théorème 9.2 (représentativité du séparateur) . . . 133
A.4 Correction de l'algorithme Arbres . . . 133
A.5 Conditions de terminaison de l'algorithme Arbres . . . 136

Bibliographie 137

Liste des publications 147

Introduction générale

> Au commencement, l'Homme dit : « Ordinateurs du monde entier, unissez-vous. » Ainsi firent les ordinateurs. L'Homme fut satisfait du résultat, et l'utilisa pour échanger des courriers électroniques et des fichiers. [...]
>
> La troisième décennie, l'Homme dit : « Ordinateurs du monde entier, constituez une base d'information facile à utiliser. » Ainsi firent les ordinateurs. L'Homme, satisfait, nomma le résultat « the web » et monta quelques *startups* éphémères en bourse. [...]
>
> La cinquième décennie, l'Homme dit : « Ordinateurs du monde entier, constituez un gigantesque méta-ordinateur connectant mon frigo à mon grille-pain. » L'Homme, satisfait de son idée, la nommait déjà « Grid[-pain] ». Mais les ordinateurs n'en firent rien. Visiblement, ils avaient besoin d'un peu d'aide pour cela...
>
> — Première version, refusée.

Afin de répondre aux besoins de puissance de calcul sans cesse croissants, le *metacomputing* est une extension du parallélisme consistant à fédérer des ressources hétérogènes de calcul et de stockage distribuées pour en agréger la puissance. Une machine virtuelle ainsi formée par un large ensemble d'organisations distantes partageant leurs ressources locales est souvent dénommée *grille* (ou *Grid*).

Contrairement aux machines parallèles l'ayant précédée, cette plate-forme présente des caractéristiques intrinsèquement hétérogènes. De plus, les ressources ne sont que rarement réservées à un seul utilisateur, ce qui implique une forte dynamicité des disponibilités.

Pour relever les défis posés par cette plate-forme, une approche classique consiste à utiliser une extension des RPC (Remote Procedure Call – invocations de procédures distantes). Des *clients* soumettent des requêtes de calculs à des *agents* chargés de les ordonnancer interactivement sur des *serveurs* de calculs (utilisant des bibliothèques de calcul parallèles ou séquentielles) en fonction des capacités des serveurs et de leur charge de travail actuelle. L'appréciation de l'adéquation d'un serveur pour un calcul donné est donc l'un des problèmes majeurs à résoudre pour permettre la conception ainsi que la mise en œuvre d'algorithmes et de politiques d'ordonnancement adaptés à la grille.

Cette thèse est une contribution à la résolution des problèmes posés par l'obtention d'informations actuelles et pertinentes à propos de la grille.

Ce document est découpé en trois parties. La première présente les difficultés spécifiques à ce type de plate-forme en se basant sur une sélection de projets d'infrastructures pour la grille et en détaillant les solutions proposées dans ce cadre.

La seconde partie traite de l'obtention efficace d'informations quantitatives sur les capacités de la grille et leur adéquation aux besoins des routines à ordonnancer. Après avoir détaillé les problèmes rencontrés dans ce cadre, nous expliciterons notre approche. Nous présenterons ensuite l'outil FAST, développé dans le cadre de cette thèse, qui met cette méthodologie en œuvre. Nous conclurons cette partie par une étude des applications de FAST.

Dans la troisième et dernière partie de ce manuscrit, nous montrerons comment obtenir une vision plus qualitative des caractéristiques de la grille. Notre objectif est de cartographier automatiquement la topologie d'interconnexion des machines de la grille par une méthode ne nécessitant pas de privilèges d'exécution particuliers sur la plate-forme. Nous présenterons les solutions classiques dans ce domaine, puis nous relaterons une expérience de cartographie du réseau de notre laboratoire avec un outil existant. Les problèmes constatés lors de cette expérience nous ont conduit à développer un nouveau projet de cartographie du réseau (nommé ALNEM), lui même basé sur un environnement de mise au point d'applications pour la grille (nommé GRAS), également développé dans le cadre de cette thèse.

Première partie

Le Metacomputing : de la toile à la grille

Chapitre 1

Introduction

1.1 Présentation

Les besoins de puissance de calcul informatique dans quelque domaine que ce soit (comme le calcul scientifique ou financier, la modélisation, la réalité virtuelle ou la fouille de données) sont toujours croissants, et le parallélisme reste une réponse d'actualité. Pour cela, les super-calculateurs constituent des machines composées de plusieurs centaines (voire milliers) de processeurs connectés par des réseaux rapides et spécialement assemblés par des constructeurs vendant à la fois le matériel et les logiciels permettant d'en tirer les meilleures performances possibles.

Le coût prohibitif des super-calculateurs limite cependant leur usage aux laboratoires et entreprises les mieux dotés. La montée en puissance des stations de travail et de réseaux d'interconnexion peu chers ainsi que l'émergence du système d'exploitation Linux (qui rend ces machines compatibles avec les systèmes UNIX des super-calculateurs) ont permis une autre approche. Elle consiste à connecter des machines à bas prix du commerce par des réseaux classiques pour constituer des grappes de machines (*clusters*). Ces plates-formes offrent des performances parfois comparables aux super-calculateurs pour un prix bien moindre. Cependant, leur usage reste difficile du fait du manque d'intégration entre les différents composants.

Il est à noter qu'à l'heure actuelle, ces différentes solutions cohabitent, et le parc informatique typique d'un département d'entreprise ou d'un laboratoire d'université est souvent très hétérogène. Il regroupe des stations de travail personnelles, des super-calculateurs ainsi que des grappes de calcul.

Parallèlement, les années 90 et l'avènement d'Internet grand public marquent le début d'une augmentation exponentielle du nombre de systèmes informatiques interconnectés et des échanges d'informations par moyens électroniques. L'ensemble des pages, connectées les unes aux autres par des hyperliens, forme ce qui est communément nommé la toile (*the web*). Elle permet la navigation entre des pages en fonction du sujet, et indépendamment de la localisation géographique du serveur hébergeant ces informations.

La tentation est donc grande de rendre l'usage des ressources de calcul et de stockage aussi transparent que l'accès à l'information, rendu possible par la toile. Dans [FE99], Ian Foster et Karl Kesselman font le parallèle entre la simplicité d'accès à la puissance électrique délivrée par les fournisseurs et la simplicité souhaitée d'accès à la puissance de calcul. L'objectif est dès

lors de constituer une infrastructure permettant aux utilisateurs d'accéder simplement aux services fournisseurs de puissance de calcul. Par analogie à l'appellation américaine « Power Grid » désignant l'ensemble des fournisseurs électriques, cette infrastructure est communément appelée *Grid*. L'approche résultante est quant à elle souvent dénommée *Grid computing* ou *metacomputing*.

Dans cette thèse, nous avons préféré l'appellation « metacomputing », rendant à nos yeux plus précisément l'objectif d'agrégation de ressources disparates en vue de constituer une sorte de méta-ordinateur. Nous avons suivi l'usage (en le francisant) en nommant la plate-forme ainsi constituée « grille ». Notons qu'il est de plus d'usage de parler de « la grille » (au singulier) pour décrire toutes les plates-formes construites sur ce modèle. En effet, toutes devraient s'interconnecter à terme pour ne former plus qu'une seule grille tout comme la toile d'informations est unique en regroupant tous les serveurs publiant des informations à destination du public.

1.2 Caractéristiques de la grille

Le *metacomputing* consiste donc à fédérer des ressources hétérogènes de calcul et de stockage distribuées et mises en commun par différentes organisations indépendantes telles que des laboratoires universitaires ou des entreprises afin d'en agréger la puissance. La plate-forme ainsi constituée est alors naturellement une constellation de réseaux locaux (internes à chaque laboratoire) connectés entre eux par un réseau à grande échelle le plus souvent à haut débit.

Les principales différences entre la grille ainsi constituée et les super-calculateurs ou grappes de machines la composant tiennent en son hétérogénéité intrinsèque, et en la dynamicité de ses capacités. En effet, ses ressources ne sont généralement pas réservées à l'usage d'un seul utilisateur, et ce dernier doit donc composer avec d'importantes variations de performances de la plate-forme.

Une autre différence importante est d'ordre administratif. S'il est d'usage que les personnes chargées de la maintenance des grappes et super-calculateurs disposent de droits spécifiques sur les machines pour leur tâche, il est rare que les différentes administrations mettant leurs ressources en commun donnent ces mêmes droits aux personnes chargées d'administrer la grille puisqu'elles ne font pas toujours partie de leur personnel. Aussi anecdotique que cette différence puisse paraître, elle impose de mettre au point des méthodes particulières ne nécessitant aucun privilège sur les machines cibles, ce qui implique de repenser certaines solutions développées pour les super-calculateurs et les grappes.

De plus, un critère important pour juger de l'adéquation d'une solution donnée à la grille est sa capacité d'extensibilité ou de passage à l'échelle (*scalability*). Les super-calculateurs et grappes de calcul regroupent le plus souvent quelques dizaines de nœuds ou quelques milliers pour les plus gros d'entre eux tandis que le nombre de machines sur la grille se compte potentiellement en milliers, voire en centaines de milliers. Cela impose une contrainte supplémentaire importante sur les solutions destinées à être utilisées sur la grille, imposant elle aussi de repenser la plupart de celles développées jusque là.

Notons enfin que de telles plates-formes matérielles existent d'ores et déjà, même si le manque de solutions logicielles adaptées réserve encore leur usage aux spécialistes. Certaines des expériences présentées dans cette thèse ont par exemple été réalisées sur le réseau français

à Vraiment Très Haut Débit (VTHD)[1], qui relie, entre autres, les Unités de Recherche de l'INRIA. Citons aussi les plates-formes d'essai (*testbeds*) NPACI[2] et TeraGrid[3] aux États-Unis, les projets européens E-Grid[4] (dont le réseau VTHD fait partie) et DataGrid[5], ou apGrid[6] en Asie.

[1] URL : http://www.vthd.org/
[2] URL : http://www.npaci.edu/
[3] URL : http://www.teragrid.org/
[4] URL : http://www.egrid.org/
[5] URL : http://eu-datagrid.web.cern.ch/eu-datagrid/
[6] URL : http://www.apgrid.org/

Chapitre 2

Quelques environnements pour la grille

Nous avons vu dans le chapitre précédent que la grille est une plate-forme difficile d'usage en raison de son hétérogénéité et de sa grande dynamicité. Plusieurs modèles de programmation sont possibles, et la plupart d'entre eux étendent des solutions éprouvées pour les adapter aux contraintes spécifiques de la grille. Les solutions logicielles mises au point pour aider à la réalisation d'autres programmes sont souvent appelées *middleware* (parfois traduit par « intergiciel »). Citons par exemple le projet Legion [NNTH+03], offrant un langage objet spécifiquement adapté à la grille, ou les projets MPICH-G [KTF03] et PACX-MPI [KKM+03], constituant des environnements à base de passage de messages pour la grille. Il est également possible d'utiliser sur cette plate-forme des environnements de calcul global comme Nimrod [BAG02] ou XtremWeb [FGNC01], initialement prévus pour des conditions encore plus changeantes que celles de la grille, ou ceux basés sur le Web comme iMW [GG00] ou Punch [KFLR01].

Une autre approche est d'étendre le modèle classique des RPC (*Remote Procedure Call* – invocations de procédures distantes) pour l'adapter à la grille. De nombreux systèmes tels que NetSolve [CD98], NINF [NSS99] ou DIET [CDL+02] sont construits sur ce modèle, en voie de standardisation grâce au GridRPC [MC00a, MNS+00] du *Global Grid Forum*[1].

Ce chapitre présente une sélection de projets visant à simplifier l'usage de la grille. Nous verrons dans la section 2.1 comment l'approche GridRPC tente de répondre aux difficultés des utilisateurs sur la grille. La section 2.2 présentera ensuite quelques infrastructures logicielles destinées à simplifier la réalisation de programmes fonctionnant sur la grille.

2.1 Les environnements d'utilisation de type GridRPC

L'une des approches classiques pour simplifier l'usage de la grille aux utilisateurs est donc d'étendre le modèle d'invocation de routines à distance pour l'adapter aux contraintes de cette nouvelle plate-forme en utilisant les ressources distribuées de façon transparente. Cette approche, nommée GridRPC [MC00a, MNS+00], est en voie de standardisation au sein des groupes de travail du *Global Grid Forum*. Les environnements construits sur ce modèle sont généralement appelés des serveurs de calcul ou NES (*Network Enabled Servers*).

[1] URL : http://www.gridforum.org/

Les applications cibles sont diverses mais elles nécessitent généralement de grandes puissances de calcul et/ou de grosses capacités de stockage. Le grain de calcul peut aller de quelques minutes à plusieurs jours. Le modèle de calcul est habituellement le parallélisme de tâches (utilisant des requêtes asynchrones) mais les serveurs de calcul étant souvent eux-mêmes parallèles, il est possible d'ajouter un parallélisme mixte [Ram96].

Ces environnements sont généralement constitués de cinq composants fondamentaux. Des **clients** constituent l'interface des utilisateurs et leur permettent de soumettre les requêtes de calcul. Les **serveurs** reçoivent les requêtes des clients et exécutent les modules logiciels pour eux. Une **base de données** contient les informations sur les logiciels disponibles, sur les serveurs, et via des **sondes** logicielles (ou *senseurs*), sur leurs performances (temps de communication, performances de calcul, disponibilité mémoire, charge, etc.). La fonction de l'**ordonnanceur** est alors de distribuer les calculs afin d'équilibrer la charge du système.

Plusieurs outils offrant cette fonctionnalité sont déjà disponibles, comme NetSolve [CD98], Ninf [NSS99], NEOS [FMM00], RCS [AGO97] ou DIET [CDL+02], et nous présenterons certains d'entre eux plus en détail dans la suite de cette section. Ils constituent un *middleware* entre les portails applicatifs et les composants pour la grille et peuvent eux-mêmes reposer sur d'autres solutions plus génériques telles que Globus, Legion, XtremWeb ou AppLeS.

Dans la suite de cette section, nous allons examiner les fonctionnalités nécessaires au développement d'environnements de type GridRPC performants et utilisables.

2.1.1 Problématiques

Examinons à présent les divers problèmes qui peuvent se poser lors du développement et du déploiement d'un environnement de type GridRPC.

L'**interface client** doit être la plus simple possible. Il est souhaitable d'offrir une interface depuis différents langages de programmation comme C, Fortran ou Java, ainsi que de permettre de déporter de façon transparente des calculs exprimés par l'utilisateur dans un environnement de résolution de problèmes tel que Mathematica, MatLab ou SciLab. Un groupe de travail du *Global Grid Forum* (GridRPC WG[2]) tente de définir une API standard aux différents environnements de GridRPC. L'un des problèmes de cette standardisation est la description de problèmes et des données associées. Aucun consensus ne s'est dégagé jusqu'à présent, et nous présenterons dans le chapitre 4 (section 4.2.1) une façon de décrire les données dont nous espérons qu'elle sera utilisée comme base de standardisation au sein de ce groupe de travail.

L'**agent** est la partie maîtresse d'un tel environnement. Il est chargé de trouver le serveur le plus adéquat à servir chaque requête des clients. Il doit également équilibrer la charge entre les différents serveurs et donc ordonnancer les requêtes en fonction de leur durée estimée et des coûts de transferts de données. Il doit récupérer les informations disponibles pour la charge des serveurs et les coûts de transfert entre les divers éléments. Pour cela, il interroge la base de données de performances. Sa localisation est importante car l'interrogation d'un agent a un coût de communication qu'il ne faut pas négliger. Tous les environnements de type GridRPC existant utilisent des algorithmes d'**ordonnancement** classiques dans lesquels on

[2]URL : https://forge.gridforum.org/projects/gridrpc-wg

cherche à minimiser le temps d'exécution des requêtes sans tenir compte des dépendances entre les calculs.

En amont des problèmes d'ordonnancement se trouve celui de l'**évaluation des performances**. Elle doit permettre de juger de la pertinence de la déportation d'un calcul sur une machine distante donnée. Il faut donc à la fois évaluer le coût de calcul ainsi que le coût de déplacement des données du client vers le serveur qui va résoudre le problème. La plate-forme cible étant généralement très dynamique, il s'agit de monitorer à intervalles réguliers les divers éléments pour remplir la base de données de performances qui sera ensuite interrogée par l'agent. La difficulté provient du fait qu'il faut être suffisamment précis et à jour pour obtenir une bonne prédiction tout en étant le moins intrusif possible.

L'**ajout de services** dans l'environnement doit pouvoir se faire en cours d'exécution sans devoir recompiler les outils. Les nouveaux services doivent être enregistrés auprès de l'agent qui pourra ainsi répondre aux requêtes des clients. Il faut également pouvoir générer les interfaces clients, si possible sans arrêter l'environnement.

Une fois le serveur identifié, le schéma classique dans ces environnements est le suivant : le client envoie sa requête de calcul avec ses données, le serveur calcule et renvoie le résultat et ceci même si ce résultat est utilisé dans des calculs futurs. Ces aller-retour occasionnent des communications inutiles impliquant une perte de performance. Il est intéressant d'optimiser la **gestion des données** afin d'éviter ces aller-retour lorsque c'est possible. On parle alors de mécanisme de persistance des données.

La plupart de ces environnements utilisent des *sockets* Unix pour leurs communications internes. Cependant, d'autres couches et protocoles de communications offrant une plus grande interopérabilité sont également utilisables comme SOAP ou CORBA.

Certaines applications nécessitent une **sécurité** accrue pour les accès aux serveurs. Les trois aspects principaux sont l'*authentification* des clients auprès des serveurs (savoir qui peut se connecter au serveur), l'*autorisation* (connaître les droits du client identifié) et la *confidentialité* (sécurisation des transferts de données par chiffrement). Les solutions les plus classiques sont Kerberos, utilisant une technologie à clé symétrique, et SSL, utilisant un algorithme à clé publique. On peut également reporter cette partie sur le *middleware* utilisé pour le développement de l'environnement (MicoSec [LS02] dans le cas de CORBA ou GSI [WSF+03] dans celui de Globus que nous présenterons dans la section 2.2.1).

La **tolérance aux pannes** est essentielle si l'on souhaite que l'environnement soit utilisé par le plus grand nombre. Il peut s'agir de pannes de l'environnement lui-même (agents, serveurs, communications, ...) ou des pannes de l'application (routine invoquée sur le serveur). Une solution intéressante consisterait à permettre une reprise sur erreur des serveurs au niveau du *middleware* lui-même. Cela permettrait de relancer les calculs en cas de panne d'un serveur. Il est plus problématique de gérer la tolérance aux pannes au niveau des agents eux-mêmes.

2.1.2 Études de cas

Nous allons maintenant présenter rapidement trois environnements utilisant ce paradigme pour le calcul sur la grille : NETSOLVE, NINF et DIET. Nous tenterons de montrer les solutions mises en place pour répondre aux problématiques introduites dans la section précédente.

2.1.2.1 NetSolve

Présentation NetSolve [CD98] est un environnement à base de serveurs de calcul développé à l'Université du Tennessee, Knoxville. Réalisé dès 1998, cet environnement fut sans doute le premier système à proposer l'approche GridRPC.

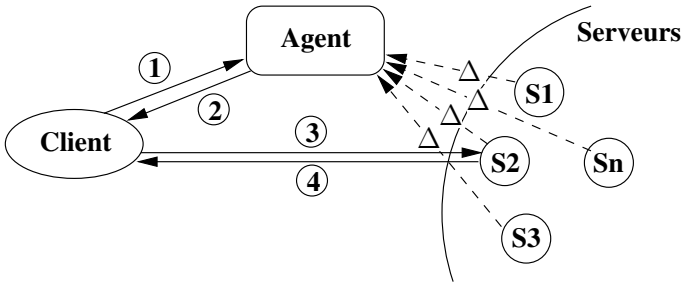


Fig. 2.1 – Architecture et fonctionnement de NetSolve.

Le fonctionnement d'une session NetSolve peut être représenté par la figure 2.1. Tout d'abord, l'agent est démarré. Puis des serveurs s'enregistrent auprès de lui, chacun envoyant une liste des problèmes qu'il est capable de résoudre. Chaque serveur communique ensuite régulièrement à l'agent des informations sur sa charge processeur et sur la latence et la bande passante du réseau entre l'agent et lui même. Les communications réalisées durant ce régime permanent sont en pointillé et marquées d'un Δ sur la figure 2.1.

Une fois l'étape d'initialisation effectuée, un client peut contacter l'agent pour lui soumettre un problème à résoudre (1). L'agent calcule alors le temps nécessaire à la réalisation de cette tâche sur les différents serveurs. Cette estimation est basée sur une formule analytique simple décrivant le comportement asymptotique de la fonction, sur la taille des données impliquées, et sur la charge de la machine et du réseau au moment de l'interrogation. L'ordonnanceur peut alors sélectionner un ensemble de serveurs adaptés au problème soumis en ne conservant que ceux dont les temps de résolutions sont les meilleurs. Cette liste de serveurs est ensuite envoyée au client (2) qui contacte ensuite directement les serveurs (3). Le calcul est effectué par le premier serveur de la liste que le client parvient à contacter, et le résultat est ensuite renvoyé au client dès la fin des calculs (4).

Liaisons avec les problématiques énoncées

Interface client NetSolve suit le standard GridRPC du GGF et propose des interfaces C, Fortran, Java, Mathematica, Scilab.

Agent L'agent maintient une base de données sur l'état des serveurs qui s'y sont enregistrés (performances CPU avec benchmark LINPACK, bande passante et latence réseau, charge, complexités des algorithmes de résolution de problèmes).

L'agent de NetSolve est centralisé, ce qui pose des problèmes de performances. Il est possible de dupliquer l'agent pour obtenir une meilleure extensibilité en lui permettant de ne traiter que les requêtes d'un ensemble de clients. Il ne s'agit cependant pas vraiment d'un ordonnancement distribué puisque chaque agent doit gérer tous les serveurs.

Ordonnancement L'ordonnancement dans NetSolve utilise des algorithmes simples en rendant au client une liste triée de serveurs les plus rapides pour exécuter une application

donnée. Le serveur choisi reçoit une pénalité lors de l'évaluation des requêtes suivantes pour éviter qu'il ne soit sélectionné à nouveau.

Évaluation de performances Les premières versions de NetSolve ne mesuraient les capacités du réseau qu'entre l'agent et chaque serveur et utilisaient les résultats obtenus pour estimer les caractéristiques du réseau entre le client et les serveurs. Si cette approximation est valide quand le client et l'agent sont proches, elle ne semble pas vraiment adaptée à la grille. Pour corriger ceci, les dernières versions utilisent NWS (présenté dans la section 2.2.2) pour estimer les caractéristiques du réseau.

De plus, les besoins des routines sont estimées dans NetSolve par une fonction de la forme xS^y où S représente la taille totale des données passées en paramètre à la routine tandis que x et y sont des constantes dépendantes de la routine et fournies lors de la déclaration du problème par la personne l'ajoutant aux services de NetSolve. Cette étude asymptotique pèche par manque de précision comme nous le verrons expérimentalement dans le chapitre 5 (section 5.1.1).

Ajout de services L'enregistrement de nouveaux services passe tout d'abord par la description de l'interface et des arguments puis par la compilation d'une enveloppe (*wrapper*) spécifique à cette bibliothèque grâce à une description du nouveau service dans un langage de bas niveau.

Gestion de données Les résultats de chaque calcul sont retournés au client dès la fin de celui-ci, ce qui impose des échanges de données coûteux et inutiles lorsqu'il s'agit de résultats intermédiaires dans la série d'opérations que le client souhaite réaliser. Par l'ajout de primitives appropriées, Emmanuel Jeannot a intégré la persistance de données dans NetSolve [DJ01]. Avec ces modifications, il est possible de demander à ne pas récupérer immédiatement les résultats d'une opération à la fin d'un calcul, mais de les laisser sur le serveur. Ensuite, il est possible de déplacer les données d'un serveur à un autre si elles constituent les paramètres d'une opération ordonnancée sur une autre machine.

Ces modifications n'ont malheureusement jamais été intégrées dans NetSolve car les auteurs ont choisi une autre approche consistant à délimiter des parties du programme client dans lesquelles des dépendances peuvent exister [ABD00]. Une analyse statique du code permet alors de générer un graphe d'appel et d'optimiser ainsi la gestion des données sur les serveurs en envoyant un ensemble de requêtes groupées et en ne récupérant que le résultat final. Cette méthode, bien plus simple à utiliser du point de vue de l'utilisateur, n'offre cependant pas la même richesse sémantique que la solution présentée par Emmanuel Jeannot, et son implémentation actuelle impose que tous les calculs d'une même séquence soient effectués sur le même serveur.

Une troisième solution consisterait à utiliser un système de cache de données sur le réseau comme IBP [BBF+02]. Chaque donnée utilisée dans le calcul est inscrite dans le cache lors des premiers calculs et le client peut ensuite récupérer un descripteur (*handle*) sur ces données et les transmettre au serveur choisi pour le calcul suivant. Le problème est alors, comme pour tout cache, de pouvoir parfois supprimer les données inutilisées.

Interface de communications NetSolve utilise une fine bibliothèque au dessus des *sockets* Unix dans laquelle les connexions sont ouvertes en cas de nécessité et sont refermées au plus tôt afin d'en limiter le nombre et garantir une meilleure extensibilité.

Malheureusement, les communications pèchent par leur manque de structure. Toutes les informations sont transmises d'un élément du système à un autre sans notion de mes-

sages clairement délimités. Cela impose que les deux parties communicantes s'entendent sur la signification de chaque octet transmis, et complique grandement l'élaboration et la maintenance du programme.

Par ailleurs, les communications entre le client et les divers composants sont effectués à travers un mandataire (*proxy*). Ceci permet de limiter les modifications à ce mandataire lors des optimisations de l'implémentation du système et de ces protocoles et d'améliorer l'interopérabilité avec d'autres systèmes comme NINF. Ce mandataire a bien sûr un coût puisqu'il relaie toutes les communications entre le client et les autres éléments de l'environnement.

Les commandes sont passées en ASCII.

Sécurité NETSOLVE utilise une simple gestion de listes d'accès avec Kerberos.

Tolérance aux pannes NETSOLVE se contente de supprimer de la base de données les serveurs qui ne répondent plus.

2.1.2.2 Ninf

Présentation NINF [NSS99] est très proche de NETSOLVE au point de vue de ses fonctionnalités et de son architecture. Sa principale différence avec NETSOLVE réside dans le fait que le serveur envoie au client l'interface qui permet de communiquer avec lui avant l'envoi des données et du calcul. Cela permet d'enregistrer dynamiquement des nouveaux services de calcul au cours de l'exécution de NINF. Les bibliothèques intégrées sont les mêmes que NETSOLVE. La figure 2.2 en donne le fonctionnement.

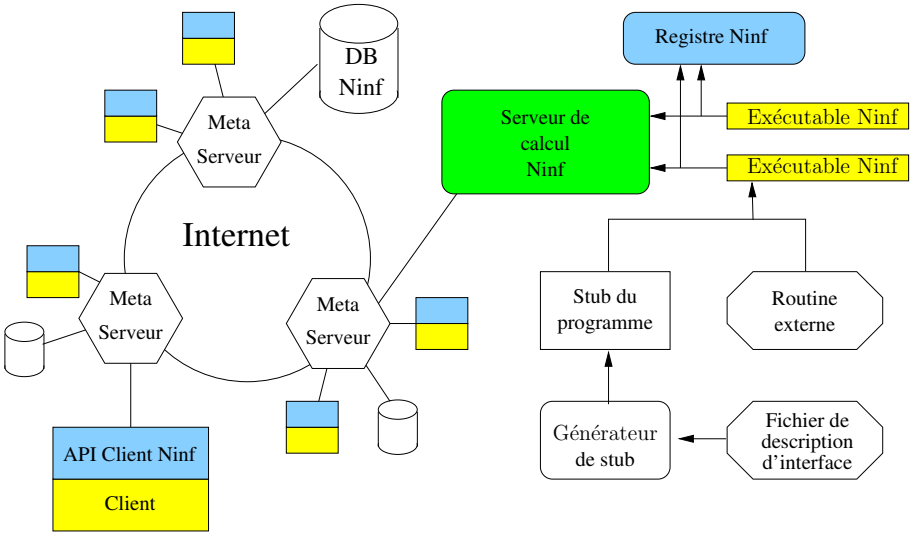


FIG. 2.2 – Architecture de NINF.

Liaisons avec les problématiques énoncées

Interface client NINF suit le standard GridRPC du GGF et propose des interfaces C, C++, Fortran et Java.

Agent L'agent est lui-même découpé en plusieurs composants. L'ordonnanceur est chargé de répartir les requêtes entre les serveurs. Pour cela, il interroge la base de données de performances, qui est elle-même mise à jour par le prédicteur. Ce dernier récupère les informations envoyées par des moniteurs de réseau et des serveurs.

Ordonnancement NINF utilise les mêmes techniques que NETSOLVE dans ce domaine.

Évaluation de performances Dans ce domaine, l'objectif des auteurs de NINF est d'utiliser le simulateur de plate-forme BRICKS développé au sein de leur équipe de recherche. Nous expliquerons plus en détail dans le chapitre 3 (section 3.2.1) les limitations de cette approche.

Ajout de services Une fonctionnalité intéressante de NINF est le transfert des descriptions d'interface (IDL) au client par les serveurs. Cela permet d'alléger les clients. De plus, l'exécution du générateur d'interface NINF sur le serveur génère un programme enveloppant les appels aux routines (*stub*) et un Makefile. La bibliothèque de résolution de problème est ensuite compilée et liée avec les programmes l'enveloppant. Le problème est ensuite enregistré auprès du serveur NINF. Cette technique est donc proche de CORBA.

Gestion de données NINF utilise les mêmes techniques que NETSOLVE dans ce domaine, à ceci près que les commandes sont encodées en XML pour simplifier leur manipulation.

Interface de communications NINF utilise les mêmes techniques que NETSOLVE dans ce domaine.

Sécurité NINF utilise les méthodes de clés publiques de la bibliothèque SSL. De plus, ses auteurs ont développé un module nommé NAA (NES Authentification Authorization) définissant la politique de sécurité entre les clients et les serveurs [MNS+00]. Ce module est proche dans sa conception du module de sécurité de Globus (GSI).

Tolérance aux pannes Tout comme NETSOLVE, NINF ne peut que détecter les serveurs ne répondant plus aux requêtes et les retirer de l'ensemble des serveurs disponibles.

2.1.2.3 DIET : Distributed Interactive Engineering Toolbox

Présentation Un problème majeur de l'architecture de NETSOLVE et NINF est le fait que l'agent constitue un point central unique. Cela forme un goulot d'étranglement lorsque le nombre de clients augmente, et toute erreur au niveau de l'agent est fatale au système entier. De plus, la topologie des réseaux actuels étant fortement hiérarchiques, la localisation de l'ordonnanceur a un impact important sur les performances de l'ensemble de la grille.

C'est pour tenter de corriger ce point que notre équipe a fondé le projet *Distributed Interactive Engineering Toolbox* (DIET) [CDL+02]. Développée dans le cadre des projets GASP du Réseau National des Technologies Logicielles (RNTL)[3] et ASP de l'ACI GRID[4], cette plate-forme de grille a pour objectif de remplacer l'agent unique de NETSOLVE ou NINF par une hiérarchie d'agents afin d'améliorer les performances lorsque le nombre de clients et de requêtes croît. La figure 2.3 présente l'architecture générale de DIET.

De futurs utilisateurs de la grille sont associés au développement de DIET afin de mieux cerner leurs besoins et de les prendre en compte au plus tôt dans l'élaboration de la plate-forme. Ces applications proviennent de domaines d'applications aussi variés que la physique, la géologie, la chimie quantique, la bio-informatique ou encore la micro-électronique.

[3] URL : http://graal.ens-lyon.fr/~gasp/
[4] URL : http://graal.ens-lyon.fr/~gridasp/

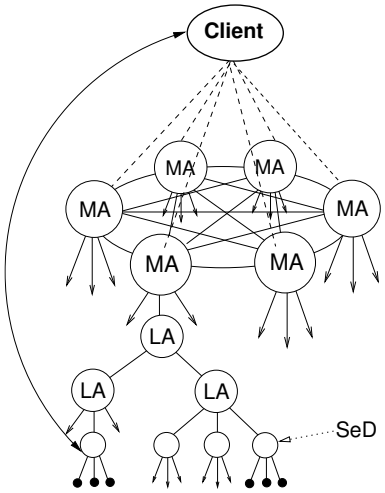


FIG. 2.3 – Architecture hiérarchique de DIET.

Liaisons avec les problématiques énoncées

Interface client DIET suit le standard GridRPC du GGF et propose des interfaces C, C++ et SCILAB.

Agent DIET remplace l'agent centralisé de NETSOLVE par une hiérarchie d'agents afin de supprimer ce point de contention et de panne. Comme le montre la figure 2.3, deux types d'agents sont différenciés : les *Master Agents* (MA) et les *Local Agents* (LA).

Nous avons vu que la grille est le plus souvent formée par la collaboration de différentes institutions, ce qui lui donne alors la forme d'une constellation à grande échelle de réseaux locaux. L'architecture de DIET rend compte de cette particularité, et le réseau local de chaque institution est géré par un *Master Agent* (MA), constituant le point d'entrée pour les clients au système. Actuellement, les MA sont connectés par un graphe complet, mais des approches de connexions plus dynamiques sont à l'étude. Chaque MA forme la racine d'une hiérarchie dont les nœuds sont des *Local Agents* (LA – gérant une sous-partie du réseau local et dont l'objectif est de transmettre les requêtes soumises par l'utilisateur depuis un MA jusqu'aux serveurs afin de se répartir la charge de l'opération d'ordonnancement) et dont les feuilles sont les serveurs de calculs.

Ordonnancement L'utilisation d'une hiérarchie d'agents augmente l'extensibilité mais complique l'ordonnancement. En effet, il faut être capable d'avoir des stratégies d'ordonnancement locales (au sein d'un même domaine) couplées avec des stratégies globales (pour l'environnement en général). Un algorithme de réservation de ressources utilisant cette hiérarchie est présenté plus précisément dans [CDPV03].

Lorsque le client soumet une requête, celle-ci descend dans les branches de l'arbre contenant un serveur à même de la résoudre. Au niveau des feuilles, les SeD évaluent le temps nécessaire à cette résolution, et l'indiquent aux LA les contrôlant. Lors de la remontée, chaque étage de la hiérarchie choisi simplement les serveurs les plus adaptés parmi ceux conseillés par ses fils, et retourne cette liste à son père. L'ordonnancement dans DIET est donc à la fois hiérarchique entre les différents fils d'un même MA et potentiellement parallèle entre les différentes hiérarchies.

Dans la version actuelle, le client ne soumet ses requêtes qu'au MA le plus proche de lui, qui ne la fait passer aux autres MA que si sa propre hiérarchie est incapable de résoudre le problème. Cela permet par exemple d'éviter que la réponse d'un MA distant (et dont l'usage peut être coûteux en termes de temps de communication) ne parvienne au client avant la réponse des MA les plus proches, mais d'autres politiques sont à l'étude pour l'avenir.

Évaluation de performances Comme nous le verrons dans le chapitre 5 (section 5.1.2) les besoins des routines et les disponibilités du système sont fournis à DIET par l'outil de prédiction de performances FAST développé dans le cadre de cette thèse et présenté dans la seconde partie. Cela permet à FAST de profiter d'une base d'expérimentation importante, permettant de mieux comprendre les besoins des ordonnanceurs sur la grille et d'adapter notre outil pour les prendre en compte.

Ajout de services Dans DIET, les serveurs de calcul sont encapsulés dans des *Server Daemon* (SeD) constituant l'interface entre la ressource de calcul et les autres composants de l'infrastructure. Des méthodes comparables à celles de NETSOLVE sont utilisées par ailleurs dans ce domaine.

Gestion de données Les utilisateurs de DIET peuvent spécifier que les résultats d'une opération donnée doivent être laissés sur le serveur après calcul. Ensuite, chaque agent a la connaissance des données distribuées dans sa descendance et tient compte des coups de migration des données lors de l'ordonnancement. Les données ainsi laissées sur les serveurs sont ensuite déplacées automatiquement vers le serveur ayant été sélectionné pour l'opération suivante.

Interface de communications Les communications dans DIET sont basées sur la norme CORBA définie par l'*Object Management Group* (OMG). Ce standard ouvert et reconnu propose une interface de haut niveau pour la construction d'applications distribuées.

Sécurité DIET ne traite pas des problèmes de sécurité explicitement, et repose dans ce domaine sur des implémentations de CORBA tenant compte de cette problématique comme MICOSEC.

Tolérance aux pannes L'architecture de DIET permet à cette infrastructure de déceler les pannes dans les serveurs, mais aussi de résister à une panne partielle de certains de ses composants.

2.1.2.4 Autres environnements

D'autres outils de ce type existent comme RCS [AGO97] qui permet d'utiliser des serveurs ScaLAPACK à distance, NEOS [FMM00] qui permet de tester divers algorithmes d'optimisation combinatoire, NIMROD [BAG02] ou encore la *Virtual Service Grid* basée sur Legion [WL02]. Des systèmes de calcul global comme XtremWeb [FGNC01] peuvent même offrir une interface de programmation de type GridRPC [Dji03].

2.2 Infrastructures de fonctionnement

Après les environnements de hauts niveaux présentés dans la section précédente, nous allons maintenant présenter quelques infrastructures de plus bas niveau visant à simplifier la réalisation d'applications pour la grille au travers de projets phares dans leur catégorie.

2.2.1 Un environnement d'exécution : Globus

Globus [FK97a] constitue sans aucun doute le projet de *metacomputing* le plus avancé à l'heure actuelle et a été fondé par les inventeurs du concept de grille Ian Foster (Argonne National Labs et Université de Chicago) et Carl Kesselman (Université de Californie du Sud). L'objectif de cette boîte-à-outils réalisée par une communauté importante de chercheurs de par le monde est de former un « environnement d'exécution » permettant d'utiliser la grille, tout comme un système d'exploitation permet d'utiliser un ordinateur. Il ne s'agit cependant pas d'un nouveau système d'exploitation, mais plutôt d'une sorte de meta-système d'exploitation, offrant la même interface au programmeur et le même environnement aux applications quel que soit le système local. Il offre pour cela divers services pour la sécurité, la localisation et la réservation de ressource, etc.

2.2.1.1 Architecture

Afin d'atteindre cet objectif, Globus est distribué sous forme d'une boîte à outils composée de nombreux éléments dont la figure 2.4 donne une vue d'ensemble.

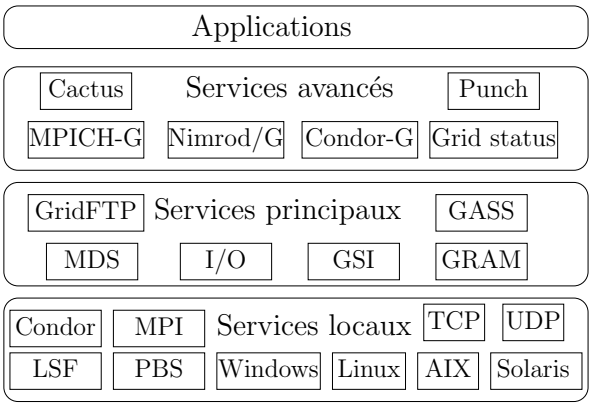


FIG. 2.4 – Architecture générale de Globus.

La boîte inférieure nommée « Services locaux » ne fait pas partie de Globus à proprement parler et regroupe les différents systèmes classiques dont Globus peut tirer parti. D'une certaine manière, Globus vise à constituer une interface unique regroupant toutes les ressources que les programmeurs d'applications distribuées doivent habituellement utiliser. Il s'agit des protocoles réseaux les plus classiques (comme TCP ou UDP), les spécificités des différents systèmes d'exploitation (comme Linux, Solaris ou Windows), mais aussi des systèmes de gestion de queue d'exécution habituellement utilisés sur les super-calculateurs ou grappes de machines comme Condor, LSF ou PBS. Globus peut aussi utiliser les bibliothèques de passage de messages de type MPI pour communiquer.

En haut de la figure se trouvent les applications finales, elles-mêmes basées sur les « Services avancés » de Globus. Il s'agit le plus souvent de bibliothèques pré-existantes modifiées pour utiliser les fonctionnalités de globus adaptées à la grille. MPICH [GLDS96] est par exemple une implémentation de MPI, et MPICH-G est une version de cette bibliothèque

adaptée à la grille grâce à son utilisation de Globus. De même, Cactus [GAL+02] est un environnement de résolution de problèmes (Problem Solving Environnement – PSE) développé initialement pour permettre la résolution des équations différentielles d'Einstein, et généralisé par la suite à d'autres problèmes de physique. Condor [TWML01] est un système d'allocations de ressources (système de *batch*) permettant d'utiliser des machines de bureau pendant leur inactivité et de les libérer automatiquement quand leur utilisateur attitré les utilise. Globus peut utiliser Condor pour gérer les exécutions sur un site local tandis que la version Condor-G utilise Globus pour rendre le même service sur plus d'un site.

Toutes ces bibliothèques utilisent le cœur de Globus (représenté dans la boîte intitulée « Services principaux ») pour offrir leurs services sur la grille. La boîte à outil Globus est composée de différentes briques de base :

MDS : Le *Metacomputing Directory Service* (service d'annuaire pour le *metacomputing*) est un annuaire permettant de stocker et retrouver des données sur la grille. Basé sur LDAP, les principaux atouts du MDS sont un schéma de nommage consistant, et une organisation des différents serveurs LDAP permettant à l'ensemble de passer à l'échelle.

GSI : La *Grid Security Interface* (interface de sécurité pour la grille) est la solution proposée par Globus pour résoudre les problèmes classiques de sécurité pour la grille. Il s'agit de l'*authentification* (s'assurer qu'un utilisateur est bien celui qu'il prétend être), la *délégation* (transmission de la notion de confiance entre entités), l'*intégrité* des messages (s'assurer que les messages ne sont pas modifiés sans le consentement de leur auteur) ainsi que la *confidentialité* des messages (s'assurer que seul leur destinataire peut en prendre connaissance).

Ce sont les problèmes classiques de la sécurité informatique, et GSI utilise et améliore les technologies existantes comme SSL (*Secure Socket Layer*) et les certificats X.509 pour les adapter à la grille. Le problème le plus difficile à résoudre à grande échelle est celui de la délégation. GSI permet de définir des politiques d'acceptation des certificats de façon à ce que les utilisateurs n'aient à s'authentifier qu'une seule fois pour utiliser tous les services de la grille.

GRAM : Le *Grid Resource Access Manager* (gestionnaire d'accès à la grille) constitue un système permettant de lancer des programmes à distance sur la grille avec le confort des systèmes de *batch* classiques.

I/O : Globus offre une interface portable pour utiliser les *sockets* et transmettre des informations en tenant compte des différences d'encodage des données entre les architectures de processeurs.

GridFTP constitue un moyen simple et robuste de déplacer des fichiers sur la grille.

GASS : Le *Globus Access to Secondary Storage* (accès Globus à des stockages secondaires) est un système de gestion des réplicas de fichiers permettant par exemple de centraliser les affichages faits par les programmes sur leur sortie standard (*stdout*) sur la machine de l'utilisateur.

2.2.1.2 Discussion

Leur avance technologique donne aux auteurs de Globus une position de leader dans le domaine du *metacomputing* et dans le forum de standardisation associé, le *Global Grid Forum*[5] (GGF). Pourtant, la solution offerte par Globus pèche encore par certains points.

Par exemple, les protocoles et interfaces utilisés ne sont pas clairement définis et changent de version en version. De plus, les dépendances entre les différentes briques de Globus étant importantes, il est très difficile de n'utiliser qu'une sous-partie de l'ensemble. À cause de ceci, la seule façon d'être compatible avec Globus est d'utiliser Globus. Ceci peut être amené à changer avec la nouvelle version de Globus (GT3) parue en juillet 2003. Implémentant le standard *Open Grid Service Architecture* (Architecture ouverte des services pour la grille) du GGF, elle tend à constituer la jonction entre les recherches sur la grille et celles sur les *web services*. L'Université de Virginie a par exemple débuté une implémentation de OGSA grâce à la plate-forme `.net` de Microsoft. Nous pensons donc que l'avenir apportera plus de souplesse et une meilleure interopérabilité entre les plates-formes de *metacomputing*.

De plus, même si l'objectif est d'offrir une sorte de couche de portabilité aux applications, de nombreux éléments de la boîte à outils ne fonctionnent encore que sous les systèmes d'exploitation Linux et parfois Windows. Les autres systèmes tels que Solaris ou AIX, pourtant encore très utilisés sur les super-calculateurs sont encore mal supportés par Globus.

2.2.2 Un outil de surveillance de la plate-forme : NWS

Le *Network Weather Service* [WSH99] (Service météorologique du réseau – NWS) est un projet mené par le Professeur R. Wolski à l'Université de Californie à Santa-Barbara (UCSB). Il s'agit d'un système distribué basé sur des senseurs logiciels permettant de regrouper des informations sur l'état actuel du réseau et des machines de la plate-forme. Il est ainsi possible d'obtenir la bande passante, la latence et le temps d'ouverture d'une *socket* de chaque lien TCP reliant deux hôtes abritant des senseurs NWS. De même, la charge processeur, les espaces mémoire et disque disponibles ou le nombre de processeurs de chaque hôte est rapporté. À propos du processeur, NWS n'est pas seulement capable de mesurer la charge actuelle, mais aussi d'en déduire la quote-part dont disposerait un nouveau processus démarrant sur cette machine. NWS ne se limite pas aux mesures des disponibilités du système, et permet aussi de prédire les évolutions à court et moyen terme des différentes métriques mesurées grâce à des traitements statistiques.

2.2.2.1 Vue d'ensemble

La figure 2.5 donne une vue d'ensemble de NWS. Le système est composé de quatre composants :

- Les **senseurs** (**S**) réalisent les expérimentations afin de mesurer les disponibilités de la plate-forme ;
- Les **serveurs de mémoire** (**M**) stockent les résultats des mesures sur disques ;
- Les **prédicteurs** (**P**) déduisent les évolutions futures des séries de mesures réalisées dans le passé par l'usage de méthodes statistiques ;

[5] URL : http://www.gridforum.org

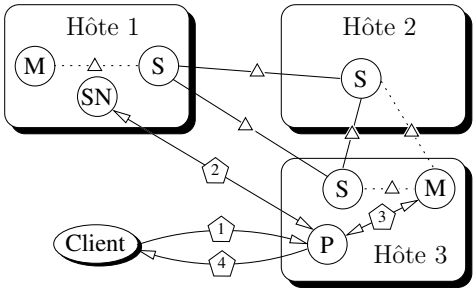


FIG. 2.5 – Architecture de NWS.

– Le **serveur de nom** (**NS**) conserve un annuaire du système à la manière d'une base de donnée LDAP. Cela permet à chaque élément de retrouver comment localiser les autres composants du système. Chaque système NWS ne peut contenir qu'un seul serveur de nom.

En régime permanent, NWS réalise des mesures même si aucun client ne soumet de requête. Les résultats sont stockés dans les serveurs de mémoire pour un éventuel usage ultérieur. Les communications ayant lieu pendant cette étape sont marquées d'un Δ sur la figure 2.5.

Quand un client soumet une requête au prédicteur (étape 1), ce dernier doit tout d'abord obtenir les anciennes mesures correspondantes à cette requête. Pour cela, il interroge le serveur de nom afin d'obtenir le nom du serveur de mémoire stockant ces données (étape 2). Dans l'exemple présenté sur la figure, il s'agit du serveur de mémoire sur l'hôte 3. Le prédicteur le contacte donc pour obtenir les données nécessaires (étape 3). Ensuite, par traitement statistique, le prédicteur estime la prochaine valeur de la série, et la renvoie au client (étape 4).

Dans les deux sections suivantes, nous allons détailler les méthodes utilisées par NWS pour mesurer les performances respectives du réseau et de chaque hôte. Nous présenterons la méthode de prédiction des évolutions futures dans la section 2.2.2.4.

2.2.2.2 Mesures réseaux

NWS peut mesurer les performances en termes de bande passante, latence et durée d'établissement d'une *socket* pour tout lien TCP/IP entre deux de ses senseurs.

La latence est approximée pour le temps d'aller-retour (*Round-Trip Time*) d'un paquet de très petite taille : la machine souhaitant mesurer la latence chronomètre le temps nécessaire pour que la machine cible lui renvoie le paquet de quatre octets qu'elle lui envoie. De même, une machine peut mesurer sur demande le temps nécessaire entre l'ouverture d'une nouvelle *socket* et la fin d'un échange de messages de très petite taille sur cette *socket*. Comme la durée de ce ping-pong est mesurée par ailleurs, cela permet de déduire le temps nécessaire à l'ouverture de la *socket* seule.

Par défaut, la machine source mesure la bande passante en chronométrant le temps nécessaire à l'envoi de 64ko par tranches de 32ko sur une *socket* configurée pour avoir un tampon de 32ko au niveau du système d'exploitation local. Ces valeurs permettent de mesurer un

réseau local classique tout en limitant l'intrusivité des expérimentations, et il est possible de les adapter à d'autres situations comme le cas d'un réseau rapide comme VTHD.

Il convient cependant de rappeler que l'objectif de NWS n'est pas de mesurer des disponibilités absolues de tels réseaux, mais d'estimer les performances que les applications peuvent escompter. En particulier, les mesures réalisées n'utilisent en émission qu'une seule *socket* sur une seule machine alors qu'il est indispensable d'utiliser plusieurs machines pour saturer un réseau offrant un débit aussi important que VTHD (2,5 Gb/s). Même en réglant convenablement les paramètres des tests, NWS détecte une bande passante maximale de l'ordre de 100 Mb/s sur VTHD. Ceci n'est pas une limitation, mais une fonctionnalité puisque cette valeur est une bonne estimation des performances qu'une application utilisant une seule *socket* peut escompter sur ce réseau.

Pour un ensemble de n hôtes, il faut réaliser $n \times (n-1)$ tests car les liens ne peuvent pas être supposés comme symétriques dans le cas général ([Pax97]). De plus, il est important de s'assurer qu'un même lien n'est jamais utilisé par deux tests de bande passante concurrents au même instant, car dans le cas contraire, les flux se partageraient la bande passante disponible sur le lien, ce qui fausserait les résultats (chaque test pourrait donner un résultat égal à la moitié de la valeur réelle). On parle alors de *collision* entre deux mesures du réseau.

Pour résoudre ce problème, NWS introduit la notion de *clique* : tous les hôtes appartenant à une même clique sont réputés partager une ressource réseau, et NWS s'assure que les tests menés entre ces liens n'entrent pas en collision. Un algorithme d'élection de leader par passage de jeton [WGT00] est utilisé pour cela, et seul l'hôte en possession du jeton à un instant donné est autorisé à initier un test réseau. Nous étudierons dans la troisième partie de ce manuscrit comment déterminer automatiquement les cliques nécessaires au bon fonctionnement de ce système.

2.2.2.3 Mesures à propos des hôtes

NWS dispose de nombreuses mesures à propos de chaque hôte parmi lesquelles la charge processeur actuelle, la quantité de mémoire vive libre, l'espace disque disponible sur chaque partition existante, les vitesses de lecture et d'écriture sur le système de fichier, *etc.* En ce qui concerne la charge processeur, NWS peut rapporter à la fois la charge actuelle et la quote-part du temps processeur dont disposerait un nouveau processus en tenant compte des systèmes de priorités (« nice mechanism ») habituels.

NWS effectue ces mesures sans nécessiter de droits privilégiés sur les hôtes afin de simplifier son usage dans un environnement de *metacomputing*. Cela complique l'obtention d'informations précises, et impose souvent l'usage de tests actifs (de façon similaire à l'envoi effectif de données sur le réseau pour en estimer les capacités).

Ainsi, pour mesurer la quote-part du processeur dont disposerait un nouveau processus, les senseurs NWS exécutent régulièrement une tâche cherchant à utiliser intensivement le processeur pendant une seconde. Le ratio cherché est alors le rapport entre le temps durant lequel ce processus test a effectivement occupé le processeur (à la fois le *system time* et le *user time* en termes techniques) et le temps total de l'expérience (une seconde).

De la même façon, l'obtention de l'espace mémoire disponible est rendue difficile par certains mécanismes des systèmes d'exploitation modernes : le noyau utilise couramment une

partie de la mémoire disponible en tampon pour les opérations d'entrée/sortie sur le disque, mais la taille de ce tampon serait réduite si un processus demandait plus d'espace que disponible. De plus, certains programmes actuellement inactifs seraient déplacés sur la mémoire d'échange (*swap*) dans ce cas. La notion de « mémoire disponible » est donc légèrement floue puisque le système s'adapterait en réduisant son occupation mémoire si l'une des applications venait à accroître ses besoins. La valeur renvoyée par le noyau (et rendue accessible par exemple par le programme `free`) correspond ainsi à l'espace mémoire disponible dans les conditions d'utilisation actuelle et non aux disponibilités mémoires maximales que NWS cherche à déterminer.

Une autre méthode serait d'analyser les informations données par le programme `ps` afin de connaître l'espace occupé par chaque processus, mais cela s'avère également impossible. En effet, les bibliothèques dynamiques ne sont chargées qu'une seule fois en mémoire et partagées entre tous les programmes les utilisant. Chaque processus se découpe donc en un segment mémoire partagé et un segment propre, et le manque de détails sur les bibliothèques utilisées par chaque processus complique grandement la détermination fiable de la taille des segments partagés. Enfin, le système rapporte la taille occupée en mémoire par chacun des processus légers (*threads*) d'un programme séparément, alors que dans les faits, ils partagent le même segment de mémoire.

La méthode choisie par NWS est alors simplement de mesurer directement l'espace mémoire accessible au maximum sur la machine en allouant autant de pages mémoire que possible, et mesurer la taille obtenue avant de libérer ces pages.

Même s'il s'agit de la seule façon de mesurer ces grandeurs, ces méthodes perturbent clairement le système respectivement en lançant une tâche inutile ou en forçant le système à utiliser la mémoire d'échange. Pour limiter ces perturbations, NWS alterne ces tests actifs avec des tests passifs consistant simplement à demander une estimation de ces valeurs au système d'exploitation. Le résultat des tests actifs est alors utilisé pour calibrer un facteur correctif pour les tests passifs.

2.2.2.4 Prédiction des évolutions futures

Afin de prédire les évolutions futures des mesures, NWS utilise un ensemble de méthodes statistiques formant quatre grandes familles : la moyenne, la médiane (la valeur de l'élément au milieu de la liste des valeurs une fois triée), la moyenne amortie (ma, la valeur à l'étape n étant donnée par la formule $ma(t, g) = (1 - g) \times ma(t - 1, g) + g \times t$, g étant le gain de la moyenne amortie) et la dernière valeur. Plusieurs variantes existent dans chacune de ces familles comme les moyennes ou médianes sur un nombre variable d'éléments ou les moyennes amorties avec différents gains.

Lorsque le prédicteur NWS doit prédire l'évolution future d'une série numérique formée de n valeurs $D_1, \ldots, D_n$, il applique toutes les méthodes statistiques connues à la série privée de son dernier élément $D_1, \ldots, D_{n-1}$. Chacune des méthodes prédit alors la valeur D_n, qui est connue par ailleurs. Cela permet au système de sélectionner la méthode ayant commis la plus petite erreur à l'étape $n - 1$ pour prédire l'évolution à l'étape n.

2.2.2.5 Discussion

NWS reste le leader incontesté de la surveillance de la grille dans la communauté, et même Globus utilise maintenant NWS à la place de GloPerf, sensé rendre les mêmes services dans les premières versions de Globus.

Cette réussite est due à nos yeux à divers éléments. Tout d'abord, le système est à la fois léger et auto-suffisant, permettant à NWS de ne dépendre d'aucun autre outil, et de simplifier grandement l'installation. Mais avant même ce coté pratique, c'est la philosophie même du projet qui est innovante. Il s'agit à notre connaissance du seul projet utilisable de surveillance de la plate-forme destiné non pas à détecter les problèmes du réseau afin de les corriger, mais résolument orienté vers l'ordonnancement. Cet objectif se reflète dans le choix de métriques de haut niveau (la bande passante escomptée et non le nombre de paquets perdus par segment), et dans la capacité à prédire les évolutions futures. Cette approche se ressent aussi dans les méthodes de mesure utilisées. NWS se démarque des autres projets existants en cherchant à mesurer les phénomènes plus qu'à les comprendre. Il s'agit d'une approche classique en physique, mais bien plus novatrice en informatique. L'environnement informatique est habituellement supposé connu puisque entièrement construit par l'homme. L'extrême diversité de la grille remet cependant en cause cette supposition, comme nous le verrons dans le chapitre 3.

Deuxième partie

Prédictions de performance pour la grille

Introduction

Comme nous l'avons vu au chapitre 1, l'hétérogénéité et la dynamicité de la grille compliquent l'obtention de connaissances précises et actualisées sur cette plate-forme que ce soit à propos des besoins des routines ou des disponibilités du système. Ceci est pourtant indispensable à un usage rationnel de la grille. Cette partie détaille ces problèmes et tente de leur apporter des éléments de réponse.

Nous commencerons dans le chapitre 3 par donner un aperçu des défis à relever dans ce cadre, et des méthodes classiquement utilisées dans la littérature pour y répondre. Nous présenterons ensuite l'approche retenue, nommée *macro-benchmarking*, et dont la caractéristique principale est de mettre l'accent sur les mesures des variations de performances plus que sur leurs causes.

Nous appliquerons ensuite cette approche au chapitre 4 en présentant un outil développé dans le cadre de cette thèse et nommé *Fast Agent's System Timer* (FAST). Son objectif est d'offrir de manière interactive aux ordonnanceurs les informations sur la grille dont ils ont besoin. Nous montrerons les réponses apportées par cet outil aux différents problèmes en détaillant une partie de ses mécanismes internes et de son interface. FAST utilise NWS [WSH99] (présenté dans le chapitre 2, section 2.2.2) pour obtenir des connaissances sur l'état actuel de la grille. Nous présenterons également les modifications que nous avons apportées à cet outil. Nous terminerons ce chapitre par la présentation de certains résultats expérimentaux.

En conclusion de cette partie, nous détaillerons dans le chapitre 5 l'intégration de FAST dans divers projets de *metacomputing* et une extension de cet outil pour les routines parallèles réalisée par Eddy Caron et Frédéric Suter. Nous évoquerons enfin les évolutions futures de cet outil et de l'approche *macro-benchmarking*.

Chapitre 3

État de l'art et méthodologie

Ordonnancer des tâches sur des ressources de calcul revient à trouver la correspondance entre les besoins des routines et les disponibilités du système maximisant une fonction de qualité donnée. L'objectif n'étant pas ici de réaliser cet ordonnancement, nous ne détaillerons donc pas cette partie du problème. Nous allons en revanche nous attacher à énumérer ces différentes métriques, et les problèmes posés par leur obtention.

Les besoins des routines regroupent principalement le temps et l'espace mémoire nécessaires à leur exécution, ainsi que le volume de communication généré dans le cas de routines parallèles. Ces grandeurs dépendent naturellement de l'implémentation choisie et des paramètres d'appel de la routine, mais aussi de la machine sur laquelle la routine est exécutée, comme nous le verrons dans la section 3.2.

Les disponibilités du système regroupent le nombre et la vitesse des machines disponibles, ainsi que leur état (fonctionnelle, en panne, réservée au travers d'un système de *batch*, occupée par d'autres utilisateurs). Il est également nécessaire de connaître la topologie, les capacités et les protocoles du réseau interconnectant ces machines. Dans le contexte de l'ordonnancement, il est souvent plus efficace de mettre l'accent sur les performances escomptées du système plutôt que sur l'usage passé ou sur les performances de crête théoriques possibles.

3.1 Acquisition des disponibilités du système

3.1.1 Choix des métriques

Le but de l'acquisition des disponibilités du système est de permettre la prise en compte de la charge externe sur les ressources dues au partage des ressources, ainsi que celle due aux tâches ordonnancées par le système. Les trois métriques suivantes sont particulièrement importantes pour déterminer ces charges.

Charge processeur Les métriques de qualité d'un algorithme d'ordonnancement sont le plus souvent basées sur le temps nécessaire à l'exécution des tâches soumises. Il est donc crucial que l'ordonnanceur dispose d'informations sur les ressources susceptibles d'être allouées.

Charge mémoire Alors que la charge processeur influe sur le temps d'exécution de la tâche, une charge mémoire trop importante peut rendre l'exécution impossible sur une machine donnée. Cette information doit donc également être fournie à l'ordonnanceur.

Caractéristiques du réseau La grille est intrinsèquement distribuée, et les données d'un client doivent être transférées de leur localisation précédente vers les serveurs de calcul. De plus les performances de routines parallèles peuvent être fortement dépendantes des capacités réseaux. Il est donc important que ces informations soient également collectées par notre outil.

Il existe différentes façons de caractériser les capacités du réseau, dépendant de l'usage de ces informations. Les administrateurs réseau souhaitant s'assurer que leur système fonctionne correctement ont besoin d'informations de bas niveau comme le temps d'aller-retour d'un paquet (*Round-Trip Time* ou RTT) entre les différentes machines, et le taux de perte de paquets sur les différentes routes. Un utilisateur (ou à plus forte raison un ordonnanceur) est généralement plus intéressé par le temps nécessaire pour effectuer un transfert donné. Dans ce contexte, la bande passante et la latence deviennent les métriques les plus utiles. Il est vrai que ces valeurs ne dépendent pas seulement des capacités du réseau, mais également de la charge processeur des différentes machines impliquées dans la communication, mais comme nous l'expliquerons dans la section 3.2, nous préférons utiliser un modèle simple même s'il implique une légère perte de précision plutôt qu'un modèle complet d'usage malaisé. C'est pourquoi notre outil ne collecte que les bandes passantes et les latences entre les différentes machines de la plate-forme.

De manière générale, l'ordonnanceur est moins intéressé par l'usage passé des ressources que par les disponibilités futures, et notre outil doit donc se concentrer sur cet aspect. Les objectifs généraux du système de surveillance de la plate-forme listés ci-après sont parfois contradictoires, et il est donc nécessaire de trouver un compromis entre eux.

Simplicité d'usage L'objectif principal d'un outil comme FAST est de masquer la complexité de la surveillance de la plate-forme aux applications utilisatrices. Pour cela, l'interface de programmation offerte doit être simple et consistante, et les dépendances sur d'autres paquets logiciels doivent être limitées au strict minimum ;

Précision Une piètre précision du système de surveillance de la plate-forme mène à de mauvaises décisions d'ordonnancement. Ceci est d'autant plus problématique qu'il est coûteux (voire impossible) de migrer les tâches vers une meilleure localisation après leur placement initial ;

Interactivité Étant donné que FAST est utilisé par l'ordonnanceur pour prédire les performances de chaque tâche à ordonnancer sur toutes les machines disponibles, il est particulièrement important que notre outil délivre ses prédictions le plus rapidement possible ;

Réactivité Le système doit détecter le plus rapidement possible les changements de condition de la plate-forme ;

Exhaustivité L'ordonnanceur doit obtenir par le biais du système toutes les informations dont il a besoin. Si le système de mesure sous-jacent ne peut fournir de valeurs, notre outil doit pouvoir déduire une estimation cohérente des mesures réalisées. Du point de vue de l'ordonnanceur, il est crucial d'avoir une estimation de toutes les valeurs, même si elle s'avère moins précise qu'une mesure directe ;

Intrusivité limitée Le système de surveillance de la plate-forme doit induire le moins de perturbations possibles sur la plate-forme en limitant ses consommations de ressources.

Il existe deux grandes approches pour obtenir des mesures à propos du réseau. Les mesures passives consistent à ne pas injecter de trafic supplémentaire, et à instrumenter les communications réalisées par les applications afin d'obtenir un historique des capacités passées. Les mesures actives consistent au contraire à injecter à intervalle régulier des communications de mesure.

3.1.2 Mesures passives

La première possibilité pour réaliser des mesures sur le réseau sans injecter de trafic supplémentaire est de placer les instrumentations nécessaires dans les applications, comme cela a été fait pour l'outil de transfert de fichier de la suite Globus dans [VSF02]. Mais cela implique que l'outil de surveillance de la plate-forme ne dispose d'informations que lorsque la connexion est utilisée, ce qui peut s'avérer problématique au démarrage ou après une longue période d'inactivité puisqu'aucune donnée récente n'est disponible.

Une autre façon d'obtenir des mesures passives sur le réseau est de modifier le système d'exploitation afin qu'il collecte des statistiques sur les transferts effectués. Ainsi, le projet **Shared Passive Network Performance Discovery** [SSK97] constitue une pile réseau modifiée offrant des statistiques sur le taux de pertes de paquets et la latence de route à destination des routeurs utilisant cet outil. Cette approche est mal adaptée au *metacomputing*, puisqu'installer une version modifiée du système d'exploitation nécessite des privilèges sur la machine plus élevés que ceux habituellement accordés aux administrateurs de la grille.

Enfin, une troisième solution est de mettre en place une infrastructure destinée à exporter les mesures offertes par le système d'exploitation pour les rendre accessibles à distance. Il s'agit de l'approche choisie par le projet **Performance Co-Pilot** [SGI] de Silicon Graphics Inc (SGI). Des serveurs dédiés rendent disponibles sur le réseau les valeurs accessibles sur le système de fichier `/proc` de chaque machine et offrent une interface de programmation permettant d'accéder à distance à ces données. Malheureusement, ce projet n'offre que des mesures brutes sur l'usage des ressources, comme le nombre d'octets communiqués sur une carte réseau donnée. Ces données ne peuvent être utilisées directement par l'ordonnanceur souhaitant connaître le temps nécessaire à une communication, car déduire les disponibilités futures à partir des usages passés est très compliqué. Ainsi, la bande passante entre deux machines ne peut que difficilement se calculer à partir de la quantité d'octets lus et écrits sur chaque carte réseau. De la même façon, il est très difficile de déduire la quote-part dont disposerait un nouveau processus arrivant sur une machine à partir de la charge actuelle du système en raison des mécanismes de priorité utilisés dans les systèmes d'exploitation ([WSH00]). En effet, une machine chargée par des processus de priorité basse reste potentiellement disponible pour des processus de priorité plus élevée.

3.1.3 Mesures actives

Les mesures actives constituent la façon la plus naturelle d'obtenir des informations périodiques sur le réseau sans nécessiter de privilèges particuliers sur les différentes machines

impliquées. C'est pourquoi la plupart des projets de *metacomputing* utilisent cette approche. Le projet le plus avancé en la matière est sans conteste le **Network Weather Service** (NWS, [WSH99]), présenté dans le chapitre 2, section 2.2.2, et sur lequel nous avons fondé nos travaux. Il existe cependant quelques projets concurrents que nous allons maintenant détailler.

Le plus vieux projet de mesure active du réseau à notre connaissance est **NetPerf** [Hew95], réalisé par Hewlett-Packard dans les années 90. Basé sur un modèle client–serveur, il permet de tester la connexion entre la machine locale et une machine distante sur lequel le serveur NetPerf est installé. Il est également possible d'obtenir certaines informations sur l'utilisation du processeur, mais uniquement sur le système d'exploitation HP/UX et à condition que le serveur dispose de privilèges avancés. Le but principal de ce projet était de mesurer les performances de crête des différents types de matériel réseau, et non de déduire interactivement leurs performances actuelles.

Un autre système précurseur fut l'environnement de résolution de problèmes **Remote Computation Service** (RCS, [AGO96]). Ce système disposait de ses propres senseurs pour mesurer les performances de la plate-forme, même s'ils étaient limités au réseau et ne pouvaient donner d'informations à propos de la charge du processeur et de la mémoire.

Un outil de surveillance de la plate-forme fut également développé dans le cadre de Globus. Nommé **GloPerf**[LWF+98], il était composé de démons mesurant périodiquement les caractéristiques du réseau. Ce système souffrait de plusieurs défauts. Chaque démon maintenait une liste de tous les autres démons existants, ce qui limitait l'extensibilité de l'ensemble. De plus, le système était grandement lié à la suite Globus, et difficile à utiliser dans d'autres contextes. Enfin, les mesures étaient limitées au réseau sans possibilité de mesurer la charge du processeur ou de la mémoire. Ces limitations ont depuis poussé les auteurs de Globus à abandonner GloPerf au bénéfice de NWS.

PingER [MC00b] est une architecture distribuée de surveillance des performances réseaux entre plusieurs centaines d'hôtes répartis sur tous les continents. Des tests périodiques sont menés entre les différentes machines participant à la plate-forme grâce à l'outil `ping`, et les résultats sont rendus disponibles sur une page web. Le but originel du projet est de faciliter la recherche de partenaires dans des projets de calcul intensif (par exemple en physique nucléaire) en permettant de quantifier la bande passante entre les différents laboratoires impliqués, ou d'aider à la planification et à la maintenance des réseaux internationaux. Le principal défaut de ce projet, dans notre contexte, est qu'il n'offre qu'une vue microscopique du réseau (par opposition à la vue macroscopique que nous souhaitons acquérir) par des métriques telles que le RTT ou le taux de perte des paquets.

Le tableau 3.1 résume les caractéristiques des différentes solutions permettant d'obtenir les disponibilités de la plate-forme à un instant donné. Il en ressort clairement que NWS est le projet le plus adapté à nos besoins.

3.2 Caractériser les besoins des routines

Nos travaux sur la caractérisation des besoins des routines constituent l'un des apports les plus fondamentaux de FAST. Il s'agit de prédire les besoins des routines en termes de temps

Projet	GridFTP	SPNPD	PCP	NetPerf	PingER	RCS	GloPerf	NWS
Type de mesures	passive	passive	passive	active	active	active	active	active
Privilèges nécessaires	non	root	oui	pour CPU	non	non	non	non
Mesure réseau	oui	oui	oui	oui	oui	oui	oui	oui
- Perte paquets	non	oui	non	oui	oui	non	non	non
- RTT	non	oui	non	oui	oui	non	non	non
- Latence	oui	oui	non	oui	non	oui	oui	oui
- Bande passante	oui	non	non	oui	non	oui	oui	oui
Mesure CPU	non	non	oui	si HP/UX	non	non	non	oui
- Usage	-	-	oui	oui	-	-	-	oui
- Disponibilité	-	-	oui	non	-	-	-	oui
Mesure mémoire	non	non	oui	non	non	non	non	oui
Prédictions	non	non	non	non	non	non	non	oui
Accès distant	non	non	oui	non	web	oui	oui	oui
- Extensibilité	-	-	relative	-	-	faible	faible	relative

TAB. 3.1 – Résumé des outils d'obtention des capacités de la plate-forme.

de calcul et d'espace mémoire. Nous allons maintenant présenter les méthodes classiquement utilisées pour cela, ainsi que la méthode que nous avons développée.

3.2.1 Décompte des opérations élémentaires

La méthode la plus simple pour caractériser la puissance de la machine consiste à compter le nombre d'opérations élémentaires qu'elle est capable de réaliser par unité de temps. Les processeurs modernes ayant des unités spécialisées différentes pour le traitement des nombres à virgule flottante et pour le traitement des nombres entiers, on différencie habituellement les performances dans chaque catégorie. Les pseudo-unités de mesures utilisées sont respectivement les Mflop/s et les Mip/s.

La mesure de ces grandeurs est effectuée le plus souvent par étalonnage (*benchmarking*) des performances obtenues par des programmes prévus à cet effet tels que LINPACK [DLP03] dans le domaine du calcul matriciel. Les besoins des routines sont ensuite naturellement modélisés par le nombre d'opérations élémentaires nécessaire à leur exécution en fonction des paramètres d'entrée. La plupart du temps, ces besoins sont déterminés par analyse manuelle ou automatique du code source.

Cette approche est utilisée dans différents projets, tels que NetSolve que nous avons présenté dans le chapitre 2, section 2.1.2.1. Nous avons vu que les besoins des routines y sont données en Mflop/s par une expression asymptotique déterminée manuellement. La précision de cette méthode sera quantifiée dans le chapitre 5, section 5.1.1. Par ailleurs, NetSolve ne tient pas compte des besoins en termes d'espace mémoire.

Ninf-G [SNS+02] vise à constituer un portail web pour la grille s'interfaçant automatiquement avec une plate-forme très semblable à NetSolve par ses objectifs et tirant parti de Globus pour les communications et pour l'authentification. Tout comme dans NetSolve, les besoins des routines sont exprimés de manière asymptotique par le fournisseur de service, et les besoins en termes de mémoire ne sont pas pris en compte. Dans de futures versions, il est prévu d'enrichir cette stratégie par l'utilisation du simulateur **Bricks** [ATN+00]. Cet outil a pour principal objectif de comparer des algorithmes d'ordonnancement sur la grille en permettant de reproduire les conditions d'une expérience, ce qui est habituellement impossible en raison du côté intrinsèquement imprédictible des perturbations dues à l'usage des ressources par d'autres utilisateurs.

Comme le simulateur **SimGrid** [CLM03] que nous avons utilisé dans la troisième partie de cette thèse (et présenté plus précisément dans le chapitre 8, section 8.1.1), Bricks compte parmi les simulateurs demandant les calculs les moins intensifs pour leur mise en œuvre, mais leur usage dans un cadre interactif reste discutable à nos yeux.

Le principal défaut du décompte d'opérations élémentaires est que la puissance de la machine est supposée constante. Mais les performances d'une machine donnée peuvent varier grandement d'un programme à un autre, par exemple en raison des effets de cache [CCLU99, WPD01]. De plus, cette méthode ne tient pas compte des besoins en termes d'espace mémoire ni de la charge du processeur due à d'autres programmes.

3.2.1.1 Modèles analytiques complets

Une solution pour résoudre certains problèmes posés par cette approche est d'enrichir le modèle de la machine pour prendre plus d'éléments en compte. De nombreux travaux dans la littérature présentent des modèles analytiques précis pour le système de cache [WSMW00], la hiérarchie mémoire [JCSM96], les performances des entrées/sorties [SHS00] ou même un ordinateur complet [AE98, SS96].

Ces modèles sont complexes en raison de la quantité de paramètres à prendre en compte. Il faut évidemment étudier l'architecture matérielle de la machine et les performances de chacun de ses composants, mais également le système d'exploitation utilisé et ses interactions avec le matériel, qui peuvent varier entre deux versions du même système. L'étude détaillée de tous ces éléments constitue une tâche colossale, et cette approche semble mal adaptée à une plate-forme aussi diversifiée et changeante que la grille. De plus, ces modifications compliquent le modèle, rendant difficile la détermination des besoins des routines en fonction des différents paramètres du modèle.

Enfin, cette approche ne peut prendre en compte certaines optimisations réalisées par les systèmes modernes. Les plus courantes sont celles des compilateurs, déroulant certaines boucles du programme pour mieux les adapter au processeur. Mais les processeurs modernes optimisent également le code qu'ils exécutent. Ils sont par exemple capables de réaliser plusieurs opérations élémentaires en parallèle, voire de réordonner les instructions ou de réaliser des exécutions spéculatives, c'est-à-dire d'exécuter des parties de codes à l'avance même s'il n'est pas certain qu'il s'avère nécessaire de les réaliser.

3.2.1.2 Le « micro-benchmarking »

Grâce au *micro-benchmarking*, il est possible d'automatiser l'élaboration des modèles décrivant les machines. Ainsi, le projet **lmbench** [MS96] vise à étalonner automatiquement les performances de la machine par exemple en termes de débit et latence des accès à la mémoire et au disque. Les performances du système d'exploitation sont aussi mesurées en termes de temps de traitement d'un signal, de création de processus, de communication inter-processus ou de changement de contexte.

Ces outils sont bien adaptés à la recherche en architecture pour déterminer la machine idéale pour résoudre un problème donné ou pour mesurer les efforts d'optimisation dans le système d'exploitation. Mais l'accumulation de paramètres mène à des modèles complexes dont la mise en œuvre requiert des quantités de calcul non négligeables, ce qui les rend peu adaptés à un outil interactif tel que FAST. Par ailleurs, ces solutions compliquent encore l'établissement de la description des applications en ajoutant d'autres paramètres au modèle.

ChronosMix [Bou00] constitue un ensemble complet de prédiction de performances pour les applications distribuées et parallèles. Cet outil peut être utilisé en mode statique ou semi-statique. Dans le premier mode, les performances de la machine sont modélisées en mesurant les temps d'exécution de 177 instructions élémentaires comprenant les opérations classiques sur les différents types de données (double, double long, entier, *etc.*) et les opérations mathématiques trigonométriques. Une analyse statique du code source de la routine est ensuite menée pour dénombrer les opérations nécessaires en fonction des paramètres. En mode semi-statique, le code est instrumenté automatiquement, et exécuté pour chronométrer les performances de

blocs de plus grande taille. Dans les deux cas, l'instanciation du modèle d'application extrait nécessite l'usage d'un simulateur, ce qui semble compromettre son usage dans un cadre interactif.

3.2.1.3 Signatures d'application

Certains travaux dans la littérature étudient les variations des besoins des routines au cours de leur exécution. Ainsi, dans [Lu02, VAMR01], les auteurs extraient ce qu'ils nomment la « signature de l'application ». Il s'agit d'une courbe paramétrée par le temps et parcourant l'espace des besoins de la routine en fonction du temps processeur, de la quantité d'entrées/sorties et du volume de communication. Cette courbe est ensuite projetée dans l'espace des disponibilités actuelles rapportées par NWS. Empruntant au vocabulaire de la biologie, les auteurs rapprochent la signature au « comportement intrinsèque » de l'application et les disponibilités du système aux « stimulis » ressentis par l'application.

Cette approche, bien que très attirante, semble difficile à mettre en œuvre. Le premier problème est l'obtention du comportement intrinsèque de l'application. Les auteurs utilisent un échantillonage de données issues d'une exécution précédente. À nos yeux, il serait nécessaire de placer l'application dans un environnement sans facteur limitant pour obtenir son véritable comportement intrinsèque. Une solution pourrait être d'utiliser des données récoltées lors de l'exécution de l'application sur une plate-forme bien plus puissante que celle ciblée afin de s'assurer que l'application ne rencontrera aucun facteur limitant et délivrera ainsi son vrai comportement intrinsèque. Mais cette méthode semble également difficilement applicable, car il est rare que les utilisateurs disposent d'une plate-forme très puissante pour calibrer leurs outils, et décident ensuite d'exécuter leur application sur une plate-forme bien moins puissante.

À notre avis, une telle signature intrinsèque devrait également prendre en compte bien plus de métriques que celles utilisées par les auteurs. Ainsi, une application suffisamment régulière pour permettre un remplissage continu de plusieurs canaux de traitement des nombres en virgule flottante bénéficierait d'optimisations réalisées par les processeurs modernes. Nous pensons donc que les signatures présentées dans ces travaux sont difficilement portables entre les différentes architectures matérielles.

Enfin, les variations de performances lorsque l'une des ressources vient à manquer restent mal étudiées. L'existence d'une chute des performances lorsque le système manque de mémoire et se voit contraint d'utiliser la mémoire d'échange est par exemple bien connue. Mais tirer le plein parti des signatures d'application demanderait de quantifier cette chute, ce qui semble extrêmement difficile.

3.2.1.4 Étude de traces « post-mortem »

Un défaut commun à la plupart des méthodes présentées jusqu'ici est la nécessité d'analyser le code source de l'application. Cela empêche leur utilisation dans le cas des bibliothèques de calcul optimisées comme celles vendues par les vendeurs de super-calculateurs. Elles sont également difficilement applicables aux bibliothèques les mieux optimisées comme ATLAS [WPD01] ou FFTW [FJ98]. En effet, ces bibliothèques contiennent plusieurs implémentations des routines, et détectent automatiquement quelle version est la plus performante.

Dans le cas d'ATLAS, cette sélection est réalisée lors de la compilation de la bibliothèque (en fonction des caractéristiques du compilateur) tandis qu'elle a lieu à l'exécution dans les cas de FFTW (en fonction des paramètres utilisés).

L'analyse *post-mortem* consistant à étudier les traces produites lors d'exécutions précédentes de l'application permet d'éviter toute analyse du code source et reste applicable à des applications présentant des interactions complexes avec le système.

Par exemple, **Dimemas** [BEG+03] est un outil de prédiction de performances développé à l'Université Polytechnique de Catalogne (UPC) et distribué par PALLAS GmbH. Il utilise pour cela les schémas et traces de communication MPI capturées par le programme Vampir (du même distributeur). D'après les auteurs, cet outil est plus adapté à l'étude des variations de performances en fonction du nombre de processeurs ou de la vitesse du réseau d'interconnexion qu'à la prédiction interactive des performances d'un programme en fonction de la charge courante de la plate-forme.

3.2.2 Approche probabiliste et chaînes de Markov

Les probabilités et statistiques constituent une autre approche pour réduire le nombre de paramètres des modèles d'architecture matérielle. Des travaux ont abouti à la modélisation des entrées/sorties [OR02], de la hiérarchie mémoire [JG99] ou même de la grille dans son ensemble [GSW02] par des chaînes de Markov. Cela permet de s'abstraire des instabilités de la plate-forme en considérant le comportement moyen des applications sur plusieurs centaines d'exécutions sur une longue période de temps.

L'instanciation des différentes variables de ces modèles à un instant donné afin d'en permettre un usage interactif reste toutefois difficile. Ainsi, Gong *et Al.* supposent dans [GSW02] que l'utilisateur d'une station de travail utilise sa machine (interdisant son usage par les outils de grille) selon une loi de type M/G/1 (*i.e.* les chances pour que l'utilisateur commence à utiliser sa machine suivent une loi exponentielle, et l'utilise ensuite pour une durée suivant une loi arbitraire), mais ne fournit aucune solution pour mesurer les paramètres de ce système afin de permettre de prédire le temps d'exécution d'une application donnée à un instant précis.

3.2.3 Le « macro-benchmarking »

Toutes les approches présentées jusqu'ici ont un point commun. Elles tentent toutes de prédire les performances à venir en identifiant les raisons menant à leur variation. Pour cela, elles découpent le système en sous-systèmes et modélisent séparément chaque partie afin d'obtenir une meilleure compréhension de chaque élément. La réponse classique au manque de précision du modèle d'un sous-système est son découpage en sous-parties encore plus petites.

Notre approche, que nous nommons **macro-benchmarking**, consiste au contraire à considérer le couple { programme ; machine } comme un tout et d'en mesurer les performances sans chercher à comprendre quelles parties du système impliquent les variations que nous observons. Pour cela, nous étalonnons les routines potentiellement ordonnancées pour un jeu de paramètres d'entrée représentatif sur chaque machine de la plate-forme (comme nous le détaillerons dans le chapitre 4, section 4.1.2). Les résultats sont ensuite interpolés par régression polynomiale afin de simplifier leur manipulation et leur partage entre les éléments de l'environnement distribué.

Le principal avantage de cette approche est qu'elle ne pose aucune condition sur le système sous-jacent. Notre cadre de travail est applicable à toutes les routines dont les performances varient régulièrement avec les paramètres d'entrée et sont chronométrables. De plus, la simplicité de l'approche nous permet d'ajouter aisément des métriques à notre étude lorsque le besoin s'en fait sentir. Ainsi, notre outil tient compte des besoins en termes d'espace mémoire des routines. Au contraire, ajouter de nouvelles métriques dans les systèmes cherchant à analyser la plate-forme en détail demande de les enrichir considérablement, ce qui les complique encore.

La critique la plus courante de cette approche est qu'il est en théorie possible d'expliquer complètement les systèmes informatiques puisqu'ils sont réalisés par l'homme. Notre approche peut alors être ressentie comme un renoncement par certains scientifiques. Pourtant, l'approche *macro* n'est pas nouvelle, et elle est par exemple couramment utilisée en thermodynamique. La théorie microscopique de la thermodynamique permet de calculer très précisément tous les phénomènes de la physique, mais son formalisme est tellement complexe qu'il est courant d'utiliser la théorie macroscopique introduisant certaines simplifications pour raisons d'efficacité. En ce sens, le *macro-benchmarking* est une méthode simplement applicable à tout type de matériel et à de nombreux logiciels, simplement extensible à l'ajout de nouvelles métriques et menant à des pertes de précision tolérables dans le cadre du *metacomputing* (comme nous le montrerons dans le chapitre 4, section 4.3.2).

Le principal défaut du *macro-benchmarking* est que l'étape d'étalonnage lors de l'installation du logiciel peut s'avérer longue. Mais il est possible de la réaliser en parallèle dans le cas des grappes de machines, le plus souvent homogènes. De plus, cette étape ne doit être réalisée qu'une seule fois car les résultats sont stockés dans une base de données distribuée et réutilisés ensuite sans devoir être recalculés à chaque fois.

Cependant, cette approche est limitée aux cas où il est possible de s'abstraire de la charge extérieure lors de l'étalonnage et où les performances des routines dépendent de caractéristiques simples des paramètres d'entrée et non du contenu même de ces données. Nous allons maintenant étudier ces deux cas, et présenter des stratégies possibles pour chacun d'entre eux.

3.2.3.1 Isoler les besoins des routines de la charge externe

La base du *macro-benchmarking* est d'étalonner précisément les besoins des routines. Lors de cette étape, il est indispensable de pouvoir isoler les effets de la routine des perturbations comme celles que peuvent induire d'autres programmes utilisant les mêmes ressources. Ceci est relativement simple pour des métriques telles que le temps d'exécution ou l'espace mémoire car tous les systèmes d'exploitation permettent d'obtenir la quantité de ces ressources utilisées par un processus donné.

En particulier, les temps système et utilisateur d'un processus représentent le temps d'exécution de la routine passé respectivement en mode noyau et en mode utilisateur. La somme de ces deux temps représente l'usage du processeur par le processus. Cette durée est différente du *wall clock*, qui représente le temps écoulé entre le moment où la routine commence et sa fin. En effet, les temps dévolus à d'autres programmes ou à des opérations d'entrée/sortie ne sont décomptés que dans le *wall clock* tandis que la somme des temps système et utilisateur

constitue une mesure exacte[1]. La précision offerte par ces grandeurs est cependant légèrement moindre que celle de l'horloge « wall clock » (de l'ordre du dixième ou centième de seconde contre la micro-seconde), mais cette perte de précision nous semble acceptable pour le grain relativement gros des opérations communément ordonnancées sur la grille, où la durée de chaque tâche dépasse généralement trente secondes.

Malheureusement, il n'existe aucun moyen de s'isoler de la sorte de la charge extérieure à propos du réseau. Cela rend impossible d'étalonner les routines parallèles de manière reproductible. Mais de telles fonctions sont en général naturellement décomposées en sous-fonctions séquentielles et communications inter-nœud (avec parfois un recouvrement entre ces différentes étapes). Au contraire des routines séquentielles il est alors possible d'analyser précisément ces fonctions pour extraire le schéma de communication et de calcul utilisé. Cela permet d'établir une simple équation exprimant le temps d'exécution en fonction des paramètres de la fonction, des temps d'exécution des parties séquentielles (donnés par FAST) et des capacités actuelles du réseau (que FAST est également capable de donner).

Comme nous le verrons dans le chapitre 5 section 5.2, cette approche a été appliquée avec succès à la bibliothèque ScaLAPACK par analyse manuelle du code source.

3.2.3.2 Routines dont les performances dépendent du contenu des données

L'approche par *macro-benchmarking* suppose que les performances des routines dépendent directement des paramètres d'entrée et de leurs caractéristiques (comme leur taille dans le cas des matrices). Bien que cette supposition soit justifiée pour la plupart des problèmes comme les opérations d'algèbre linéaire dense, ce n'est pas toujours le cas. Par exemple, dans le cas de matrices creuses, le temps d'exécution dépend en général non seulement de la taille des données, mais également du nombre et de la position des éléments non-nuls. L'obtention de ces informations nécessite une phase de calcul préalable d'un coût assez important.

Le traitement de telles fonctions dans FAST reste un problème ouvert, et différentes approches semblent possible. La plus simple est naturellement d'utiliser un étalonnage générique afin de sélectionner la machine la plus puissante pour une tâche sans chercher à prédire son temps d'exécution.

Une méthode plus satisfaisante serait de décomposer les opérations en différentes étapes. Ainsi, les solveurs creux tels que MUMPS [ADKL01] ou PASTIX [HRR02] sont naturellement décomposés en trois phases : l'analyse étudie la matrice et la réordonne afin de limiter le remplissage dû à la factorisation réalisée lors de la seconde phase. La résolution constitue la troisième phase du processus. Les deux dernières étapes sont celles demandant les calculs les plus intensifs. L'avantage d'une telle décomposition est que les performances des deux dernières étapes peuvent être estimées à partir des caractéristiques de la matrice calculées lors de la première étape, et que cette phase d'analyse représente souvent moins de 10% du temps de calcul total.

Une approche serait alors d'exécuter la phase d'analyse sur une machine de test, puis d'utiliser FAST pour prédire les performances des deux dernières étapes pour les différentes machines disponibles. La principale limitation de cette méthode est que l'implémentation

[1] Ceci n'est pas complètement vrai puisque l'exécution d'autres processus peut modifier le remplissage du cache, ce qui peut impliquer une légère baisse de performances. Cette perturbation reste cependant négligeable.

actuelle de MUMPS (et de tous les autres solveurs creux que nous connaissons) ne fournit pas d'interface séparée pour les différentes phases. Cela implique que si l'ordonnanceur décide d'exécuter la résolution sur une machine différente de celle sur laquelle a eu lieu l'analyse, cette phase devrait être refaite sur la machine choisie. Une telle décomposition reste cependant une approche intéressante pour les routines dont les performances dépendent de caractéristiques des données difficiles à extraire.

3.3 Résumé

Dans ce chapitre, nous avons présenté nos objectifs quant à la modélisation des besoins des routines et à l'acquisition des disponibilités de la plate-forme. L'approche choisie, nommée *macro-benchmarking*, cherche à établir des modèles simples et robustes utilisables dans un environnement interactif. Nous avons également étudié certaines limitations de l'approche, en particulier vis-à-vis des besoins de communication des routines parallèles et vis-à-vis des routines dont les performances dépendent de caractéristiques des paramètres difficiles à extraire, ainsi que des pistes pour dépasser ces limites.

Chapitre 4

Outil de prédiction de performance automatique pour la grille

L'objectif de ce chapitre est de présenter plus précisément la bibliothèque FAST (*Fast Agent's System Timer*). Implémentant l'approche présentée au chapitre précédent, son objectif est de fournir à d'autres applications des informations sur les besoins de routines en termes de temps d'exécution et d'espace mémoire ainsi que sur les disponibilités de la plate-forme.

Elle est principalement destinée à être utilisée par un ordonnanceur souhaitant placer une routine séquentielle sur la grille, mais nous verrons dans le chapitre suivant qu'elle a également été utilisée dans d'autres cadres comme la visualisation de l'état courant de la plate-forme ou l'étude de l'adéquation d'un algorithme particulier à un jeu de données particulier, ainsi que pour le traitement de routines parallèles. Le but de FAST n'est cependant pas de réaliser l'ordonnancement à proprement parler.

FAST utilise NWS présenté dans le chapitre 2, section 2.2.2 pour obtenir les prédictions des disponibilités du système, et nous présentons également ici les modifications apportées à cet outil, ainsi que certains résultats expérimentaux.

4.1 Présentation de FAST

La figure 4.1 présente l'architecture générale de FAST, découpée en deux parties principales. En bas de la figure, un programme d'étalonnage permet de mesurer automatiquement les besoins des routines sur chaque machine du système. En haut de la figure, une bibliothèque offre des prédictions interactives aux applications clientes. Cette bibliothèque est elle-même découpée en deux sous-modules. Celui présenté sur la droite de la figure est en charge de retrouver les disponibilités de la plate-forme tandis que celui figurant à gauche utilise les résultats de la phase d'étalonnage préliminaire pour déduire les besoins des routines en fonction des paramètres d'appel.

Au centre de la figure, les deux boîtes grisées présentent les outils externes utilisés par FAST. Il s'agit d'un système de surveillance de la plate-forme et d'une base de données. Ces mécanismes sont modulaires, et il est relativement aisé de greffer un nouveau système équivalent dans FAST.

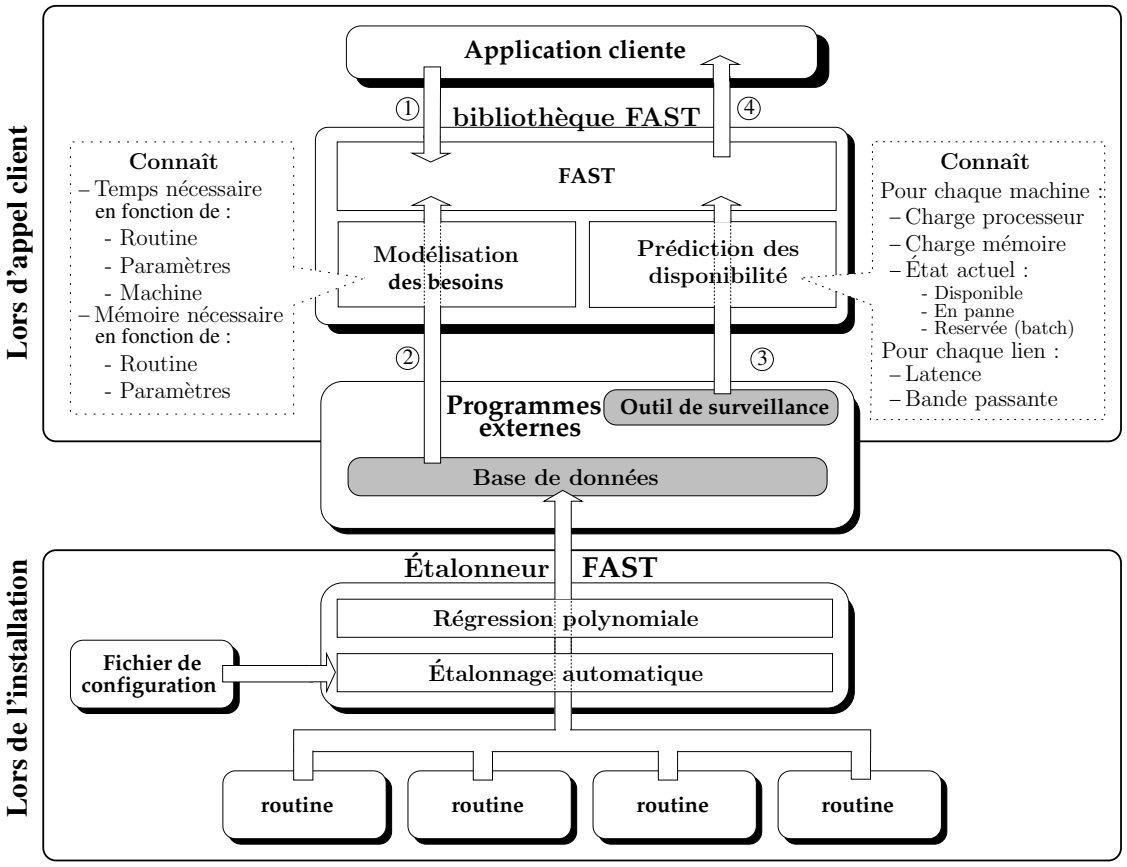


FIG. 4.1 – Architecture générale de FAST.

La base de données est utilisée pour stocker les résultats de la phase d'étalonnage. Le module de base de données le plus utilisé est fondé sur **LDAP** [HSG99], qui est un système permettant d'établir des annuaires distribués. Il est possible de découper la base de données sur plusieurs sites de façon transparente pour l'utilisateur. Cela peut permettre à l'administrateur de chaque site de gérer les données relatives aux machines sous sa responsabilité de manière indépendante. Cette fonctionnalité peut grandement aider à la mise en place de grilles partagées entre plusieurs organisations. Par ailleurs, LDAP constitue un système ouvert et largement utilisé dans le cadre du *metacomputing* et ailleurs, ce qui semble garantir la pérennité de cette technologie.

La section suivante présente le sous-module d'acquisition des disponibilités de la plate-forme et les améliorations apportées à l'outil NWS tandis que la section 4.1.2 détaillera la modélisation des besoins des routines. La section 4.1.3 montrera comment FAST combine ces deux sources d'informations afin de produire les prédictions attendues par l'ordonnanceur.

4.1.1 Surveillance des disponibilités de la plate-forme

Afin de bénéficier de bases solides et largement testées, le principal outil de surveillance de la plate-forme est NWS. Cependant, cette partie de FAST est modulaire, et greffer un autre outil équivalent serait une tâche relativement aisée. Afin de simplifier autant que possible

l'installation de FAST, il est ainsi possible d'obtenir la charge processeur d'un senseur interne limité lorsque NWS n'est pas disponible.

Dans sa version actuelle, FAST est capable de surveiller les charges du processeur et de la mémoire ainsi que la latence et la bande passante de tout lien TCP. En plus de ce qui est disponible dans la version officielle de NWS, FAST peut aussi être utilisé pour obtenir le nombre de processeurs des machines distantes. De plus, l'ajout de nouvelles métriques dans FAST comme l'espace disque disponible ou les capacités de réseau non TCP serait assez aisé.

Notre utilisation de NWS nous a permis de mieux comprendre ses forces et faiblesses. Bien qu'étant assez précis et induisant relativement peu de perturbations sur la plate-forme, ce système souffre d'une latence relativement élevée et d'une réactivité parfois faible. De plus, il est nécessaire de réduire la quantité d'expérimentations réseau directes lorsque la plate-forme grandit, ce qui a une influence néfaste sur l'exhaustivité. Nous allons maintenant détailler ces manques, et étudier les réponses apportées par FAST.

4.1.1.1 Latence

Étant composé de quatre types de démons différents répartis à travers le réseau, NWS souffre parfois de temps de réponses élevés. Comme les mesures sont réalisées périodiquement, FAST enregistre les informations obtenues dans un *cache* local pour éviter de réinterroger le système lorsque la réponse est déjà connue. Les valeurs sont rafraîchies lorsque leur âge dépasse la périodicité des mesures.

Afin de diminuer encore les temps d'interrogation, FAST utilise une fonctionnalité de NWS permettant au processus client de s'enregistrer auprès des serveurs adéquats afin de recevoir les nouvelles mesures dès leur publication. Grâce à ce mécanisme (complètement transparent), les données sont présentes en mémoire lorsque le client (par exemple l'ordonnanceur) en a besoin, et les temps d'interrogation à NWS dûs aux communications inter-processus sont entièrement évités.

Les bénéfices de ces optimisations sont montrés expérimentalement dans la section 4.3.1.1.

4.1.1.2 Réactivité

Depuis nos premières expérimentations sur la réactivité de NWS aux changements de condition sur la plate-forme [Qui02], les choses se sont grandement améliorées. Le senseur à propos de la charge processeur est maintenant capable de détecter les changements sur la plate-forme en moins de 15 secondes. Mais l'usage de FAST en condition réelle grâce à DIET montre que cela peut ne pas être suffisant [vH03]. En effet, les clients de la grille soumettent parfois un grand nombre de requêtes similaires afin de parcourir l'espace des paramètres et trouver la meilleure solution possible. Cette technique est communément nommée *parameter sweeping*, et est par exemple utilisée dans [CLZB00a]. Dans ce cas, il est très probable que deux requêtes consécutives soient traitées par l'ordonnanceur en moins de 15 secondes, ce qui empêche NWS de détecter les effets de la première tâche avant que la seconde ne soit ordonnancée. Il en suit que les deux requêtes sont alors placées sur le même serveur, même si d'autres ressources sont disponibles. Nous nommons ce problème le « bombardement de requêtes ».

Pour résoudre ce problème, FAST doit disposer d'informations sur les décisions d'ordonnancement afin de corriger ses prédictions en conséquence. Pour cela, l'ordonnanceur doit créer une réservation virtuelle au sein de FAST pour chacune des tâches qu'il assigne à des serveurs de calcul. Ces réservations ne sont que virtuelles car les ressources distantes ne sont pas effectivement réservées pour la tâche à venir. En effet, un tel mécanisme nécessiterait un échange de messages avec la machine distante impliquant une latence plus élevée du processus. De plus, il est impossible de réserver du temps processeur sur les machines qui ne sont pas gérées par un système de *batch*. En revanche, cela permet à FAST de tenir compte des tâches à venir sur les différentes ressources et de corriger ses prédictions en diminuant la disponibilité des ressources indiquées. Du point de vue de l'ordonnanceur, tout se passe donc comme si la réservation était réalisée et si NWS actualisait ses mesures instantanément.

Il est à noter que ceci ne constitue qu'une solution partielle au problème puisque les autres ordonnanceurs ne sont pas informés de la réservation. Ainsi, lorsque deux ordonnanceurs séparés placent en même temps des tâches comparables sur le même ensemble de machines, le second ne serait pas informé de la décision du premier. Malgré tout, cela permet d'atteindre une réactivité parfaite sans la moindre latence (puisqu'aucun échange de message n'est nécessaire) dans le cas le plus courant où un ensemble de machines donné est sous la responsabilité d'un seul ordonnanceur.

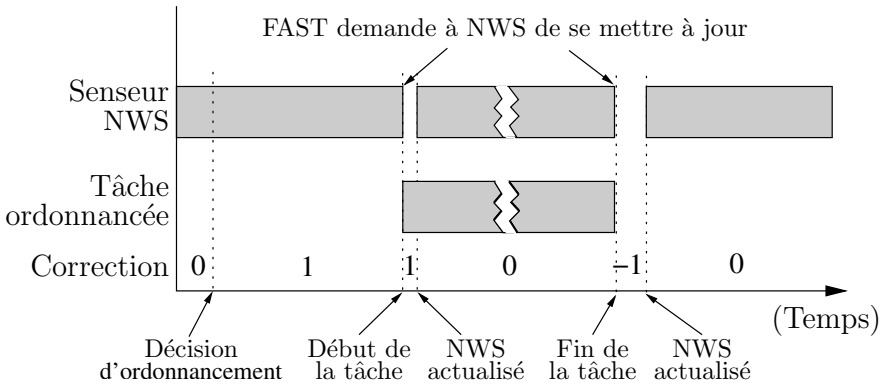


FIG. 4.2 – Diagramme de Gantt d'une réservation virtuelle.

La figure 4.2 présente le diagramme de Gantt d'une réservation virtuelle typique. Dès que la décision d'ordonnancement est prise, la réservation est créée et FAST corrige ses prédictions pour simuler le fait que la ressource correspondante est réservée. Lorsque la tâche débute effectivement, l'ordonnanceur en informe FAST qui demande alors à NWS de redémarrer ses senseurs afin de tenir compte de ce changement. Les prédictions restent corrigées jusqu'à l'actualisation effective de NWS. Lorsque la tâche se termine, l'ordonnanceur doit prévenir FAST qui demande à NWS de redémarrer le senseur une nouvelle fois. Les prédictions sont corrigées dans le sens inverse jusqu'à l'actualisation de NWS afin de simuler le fait que des ressources rapportées comme chargées sont libres en réalité.

L'interface de programmation permettant ces interactions est présentée dans la section 4.2.2 tandis que la section 4.3.1.2 montre expérimentalement ses avantages.

4.1.1.3 Extensibilité et exhaustivité des mesures réseau

À propos des mesures du réseau, il est nécessaire de mener $n \times (n-1)$ séries d'expériences pour obtenir des informations entre tous les couples de machines. Ceci n'est clairement pas extensible lorsque le nombre de machines croît. Cependant, la grille est souvent constituée d'une constellation de réseaux locaux interconnectés par des réseaux nationaux ou internationaux. Une infrastructure hiérarchique semble donc une solution naturelle pour réduire le nombre d'expériences nécessaires. Les connexions inter-sites sont alors mesurées séparément des connexions intra-sites.

Malheureusement, NWS ne peut pas combiner les mesures. Par exemple, étant donné trois machines A, B et C, si les seules connexions mesurées sont (A, B) et (B, C) il est impossible d'obtenir une estimation de la connexion (A, C) de NWS. Au contraire, FAST combinera dans ce cas les résultats existants pour estimer les valeurs manquantes. Cette approche suppose que les tests sont correctement configurés, et que la machine B est par exemple la passerelle entre A et C. La latence de (A, C) peut alors être estimée par la somme des latences mesurées sur (A, B) et (B, C) tandis que la bande passante peut être estimée comme le minimum de celles mesurées sur les différents segments du chemin. Ces estimations peuvent se révéler moins précises que des mesures directes, mais restent pertinentes lorsqu'aucune mesure exacte n'existe.

Nous reviendrons plus en détails sur cette approche, ses avantages ainsi que les difficultés qu'elle pose dans la troisième partie de cette thèse.

4.1.2 Étalonnage de routines

FAST fournit un environnement d'étalonnage générique et portable (représenté en bas de la figure 4.1). Il permet de mesurer et modéliser automatiquement les besoins de toute routine séquentielle en termes de temps d'exécution et d'espace mémoire nécessaires en fonction de ses paramètres d'appel. Pour cela, le fournisseur de service souhaitant permettre à FAST de prédire les performances d'une routine donnée (par exemple pour la rendre accessible sur la plate-forme DIET) doit écrire un programme de test lançant la routine étudiée avec les paramètres d'appel passés sur la ligne de commande du programme. Il doit aussi écrire un petit fichier de configuration décrivant le scénario de test, c'est-à-dire le jeu de paramètres sur lesquels il est nécessaire de tester la routine pour obtenir une vision globale de ses performances sur une machine donnée. Ces deux fichiers sont ensuite passés à un programme en charge de réaliser l'étalonnage en lançant le programme fourni pour chaque cas décrit dans le scénario, et en réalisant les mesures nécessaires.

Mesurer le temps de calcul nécessaire est aisé puisqu'il suffit de demander au système d'exploitation le temps pris par le processus fils en mode noyau (*system time*) et en mode utilisateur (*user time*) lorsque celui-ci se termine. Comme nous l'avons vu dans le chapitre 3, section 3.2.3.1, cela permet de plus de s'isoler de la charge externe et de ne mesurer que la consommation du processus.

En revanche, le système d'exploitation ne permet généralement pas d'obtenir l'occupation mémoire d'un processus après sa terminaison (plus exactement, il rapporte une taille nulle, puisque toutes les ressources du processus sont libérées automatiquement lorsqu'il se termine). De plus, la valeur la plus pertinente ici est l'occupation maximale du processus tout au

long de son exécution. Pour obtenir cette information, FAST utilise les mêmes méthodes que les débogueurs tels que `gdb` et lance les processus de test dans un mode permettant de les interrompre après chaque appel système et de les relancer à la demande. Lors de chaque interruption, FAST mesure l'espace actuellement occupé par le processus et garde le maximum des valeurs rencontrées. Nos expériences montrent que ceci n'influe pas sur le temps total d'exécution du processus de test, et il est donc possible de mesurer les besoins en espace mémoire et en temps de calcul au cours de la même expérience sans perte de précision.

Comme indiqué dans le chapitre 3, section 3.2.3.1, cette approche ne permet pas de mesurer les besoins des routines parallèles, mais nous pensons qu'une analyse structurelle est plus adaptée à de telles routines. Nous étudierons dans le chapitre 5, section 5.2 une extension parallèle de FAST dans le cadre de la bibliothèque parallèle ScaLAPACK.

4.1.3 Combiner ces informations

Une fois que les besoins des routines et les disponibilités de la plate-forme sont connus, FAST peut combiner ces informations et fournir une prédiction du temps de calcul tenant compte de la charge directement utilisable par l'ordonnanceur. Pour cela, on élimine tout d'abord toutes les machines ne pouvant pas effectuer le calcul soit par manque de mémoire disponible soit parce que les bibliothèques nécessaires n'y sont pas installées.

Ensuite, le temps de calcul est égal au temps que l'exécution nécessiterait en l'absence de charge extérieure divisé par le pourcentage disponible du processeur. Le temps de communication pour déplacer les données est classiquement égal à la taille du message divisé par la bande passante actuelle à quoi s'ajoute la latence.

4.2 Interface de programmation

Afin de permettre au lecteur de mieux comprendre les services rendus par FAST, cette section présente une partie de l'interface de programmation offerte. Elle est conçue pour masquer la complexité de l'obtention d'informations sur la grille et rester relativement simple. Il est possible de distinguer différentes parties. L'ordonnanceur peut obtenir des valeurs prêtes à l'emploi d'une interface de haut niveau présentée dans la section 4.2.1. Afin d'améliorer la réactivité de l'ensemble, les décisions d'ordonnancement doivent être communiquées à FAST grâce à l'interface présentée dans la section 4.2.2. Une interface de plus bas niveau offrant tous les détails des besoins des routines et disponibilités du système est également offerte et présentée dans la section 4.2.3

4.2.1 Interface de haut niveau

Après l'initialisation, il suffit de deux fonctions pour fournir à l'ordonnanceur toutes les informations dont il a besoin :

```
fast_comm_time(source, destination, taille)
```

permet de retrouver le temps nécessaire à l'envoi d'un message de la `taille` donnée (en octets) entre la `source` et la `destination` en tenant compte de la charge actuelle du réseau.

```
fast_comp_time(hôte, fonction, desc_données)
```

donne le temps de calcul de cette `fonction` (identifiée par son nom sous forme de chaîne de caractères) sur l'`hôte` (identifié lui aussi par son nom ou son adresse IP sous forme de chaîne de caractères) pour les paramètres décrits par `desc_données`. Une erreur est retournée si cet hôte ne peut réaliser ce calcul soit par manque de mémoire disponible soit parce que cette fonction n'est pas disponible sur cette machine. La charge actuelle du processeur et celle de la mémoire sont prises en compte dans ce calcul.

L'argument `desc_données` regroupe la description de chaque paramètre d'appel de la routine. Étant donné que les besoins des routines dépendent généralement des paramètres, FAST a naturellement besoin de ces informations. Chaque argument est décrit par son type de base (*char*, *int* ou *double*) et son constructeur (`aucun`, `vecteur`, `matrice`, *etc.*) ainsi que par toute autre propriété grâce à un mécanisme générique de dictionnaire associant une valeur à un nom de propriété donné. Il est ainsi possible d'ajouter la taille des données dans le cas des matrices et vecteurs. FAST fournit naturellement les fonctions nécessaires à la manipulation de ces structures de données.

Cette description d'arguments étend la notion d'`ArgStack` présentée dans le standard proposé d'interface pour le GridRPC [NMS+02]. Notre implémentation diffère de ce modèle pour assurer que les méta-données décrivant les données sont fournies avec les données elles-mêmes. FAST utilise principalement ces méta-données et ignore par exemple le contenu des matrices. Nous espérons que de futures versions du standard expliciteront de la sorte les méta-données, permettant ainsi l'utilisation de FAST avec tout *middleware* suivant ces spécifications.

4.2.2 Interface pour les réservations virtuelles

Comme expliqué dans la section 4.1.1.2, l'ordonnanceur doit communiquer ses décisions à FAST pour lui permettre d'améliorer la réactivité des mesures. Pour cela, il crée des réservations virtuelles de ressources pour chacune des tâches qu'il place et les tient à jour pour refléter leurs évolutions.

Tout d'abord, la réservation est créée grâce à la fonction

```
fast_cbook_create(hôte,taille,charge_induite)
```

`hôte` est naturellement le nom de la machine sur laquelle la tâche est ordonnancée. `taille` représente l'occupation mémoire maximale (en octets) prévue lors de l'exécution de l'opération correspondante tandis que `charge_induite` est le pourcentage du processeur qu'il est attendu que la tâche utilise au maximum. Pour une tâche séquentielle sans opération d'entrée/sortie et donc seulement limitée par le processeur, la charge induite est de 1. Pour une tâche également limitée par d'autres facteurs, la charge sera inférieure tandis qu'elle sera plus grande pour des tâches parallèles ou tirant partie des processus légers. Les valeurs `taille` et `charge_induite` doivent toutes les deux être fournies par l'utilisateur de cette fonction.

Lorsque la tâche débute effectivement sur le serveur, l'ordonnanceur doit le signaler à FAST grâce à la fonction

```
fast_cbook_apply(réservation)
```

où `réservation` est un identifiant retourné par `fast_cbook_create`. Il est également possible d'annuler une réservation avant son application effective en utilisant

```
fast_cbook_cancel(réservation)
```

Cela peut s'avérer utile si l'ordonnancement est réalisé par plusieurs processus en parallèle et que la décision d'un ordonnanceur est invalidée par celle d'un autre. Finalement, lorsque la tâche est terminée, l'ordonnanceur le signale à FAST grâce à

```
fast_cbook_end(booking)
```

Les actions réalisées par FAST lorsque ces fonctions sont utilisées sont expliquées dans la section 4.1.1.2.

4.2.3 Interface de bas niveau

Dans certains cas, l'interface de haut niveau et les valeurs prêtes à l'emploi ne sont pas suffisantes. Par exemple, un outil de visualisation comme celui présenté dans le chapitre 5, section 5.1.4 doit pouvoir retrouver le détail des disponibilités de la plate-forme. Au contraire, il peut être utile de retrouver les besoins des routines sans tenir compte de la charge. Cela peut permettre de refaire le calcul d'agrégation différemment, par exemple pour tenir compte de la vitesse des opérations d'entrée/sortie. Le projet de système expert Grid-TLSE (présenté dans le chapitre 5, section 5.1.3) prévoit également d'utiliser ces informations pour choisir le meilleur solveur de système d'équations pour un cas donné.

Cette interface est composée de deux fonctions :

```
fast_need(ressource, hôte, fonction, desc_données)
```

calcule quelle quantité de la `ressource` (temps ou espace mémoire) est nécessaire à la complétion de la `fonction` sur l'`hôte` pour les paramètres décrits par `desc_données`.

```
fast_avail(ressource, hôte1, hôte2)
```

donne la quantité de `ressource` disponible sur `hôte1` (ou entre `hôte1` et `hôte2` dans le cas de ressource réseau). La ressource peut être la charge processeur, la quote-part dont disposerait un nouveau processus arrivant sur cet hôte, la mémoire disponible, ou bien la latence ou la bande passante d'un lien du réseau.

4.3 Validations expérimentales

Après une étude des mécanismes internes de FAST et de son interface de programmation, nous allons maintenant présenter le résultat d'expériences à son sujet. La section 4.3.1 montre les bénéfices des modifications que nous avons apportées à NWS tandis que la section 4.3.2 a pour objectif de montrer la qualité des prédictions faites par FAST.

4.3.1 Amélioration au système de surveillance

4.3.1.1 Latence

Afin de réduire le temps de réponse, FAST conserve dans un cache les résultats des interrogations passées à NWS. La figure 4.3 présente les temps de réponses mesurés. 4.3(a) est le temps mesuré pour une interrogation directe à NWS (lorsque tous les processus du système sont placés sur la même machine hôte), 4.3(b) est le temps nécessaire à FAST pour consulter le cache local et le mettre à jour auprès de NWS lorsque la valeur n'est plus actuelle. 4.3(c)

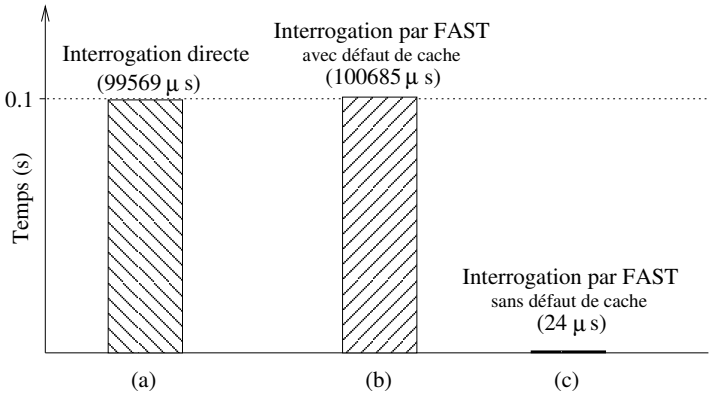


FIG. 4.3 – Temps de réponse.

est le temps nécessaire pour obtenir la valeur du cache lorsque celle ci est encore d'actualité (sans aucune interaction avec NWS).

Cette expérience montre que le surcoût de FAST est assez faible dans le cas d'un défaut de cache (environ 1%) tandis que l'obtention de la valeur depuis le cache est environ 4000 fois plus rapide qu'une interrogation de NWS lorsque tous les processus sont placés sur le même hôte. Ces bénéfices seraient naturellement encore supérieurs dans un environnement distribué du fait de l'augmentation des coûts de communication. De plus, FAST tire profit de la fonctionnalité de NWS nommée *auto-fetching* pour s'assurer que les données sont placées dans le cache de façon transparente (comme expliqué dans la section 4.1.1.1). Les défauts de cache sont donc extrêmement rares en utilisation normale.

4.3.1.2 Réactivité

Comme nous l'avons vu dans la section 4.1.1.2, il est crucial pour l'ordonnanceur que le système de surveillance réagisse le plus rapidement possible aux changements de conditions induits par les tâches qu'il place. Pour cela, il doit naturellement informer FAST de ces décisions d'ordonnancement.

Cette expérience est conçue pour que la charge du processeur varie d'une façon théoriquement connue afin de pouvoir comparer à cette valeur les mesures et les prédictions de NWS (lorsque ce système n'est pas informé des changements) et FAST (en utilisant les fonctionnalités de réservation virtuelle). Le scénario est le suivant : la machine est tout d'abord laissée inoccupée durant plusieurs minutes (la disponibilité est alors de 1, c'est-à-dire 100%). Ensuite une tâche occupant pleinement le processeur est exécutée durant une minute. La disponibilité est alors de 0,5 car un nouveau processus arrivant sur la machine devrait partager le processeur avec cette tâche. Après cette exécution, le système est laissé au repos pendant une minute, offrant une disponibilité de 1.

Par construction, cette expérience place NWS dans une situation difficile puisqu'il reçoit moins d'informations que FAST, et puisque la tâche s'exécute pendant un temps relativement court. Mais notre but ici n'est pas de démontrer le manque de réactivité de NWS, mais plutôt de montrer comment le système de surveillance de la plate-forme peut bénéficier de toute interaction possible avec l'ordonnanceur.

La figure 4.4 présente les résultats de cette expérience. Pour des raisons techniques expliquées plus loin, il s'est avéré impossible d'extraire deux jeux de données de la même exécution. Il n'était pas non plus possible d'utiliser des moyennes sur plusieurs exécutions car cela aurait lissé les résultats, alors que nous souhaitons mettre en valeur des événements discrets. Il en découle que les courbes ne sont pas parfaitement comparables. Par exemple, les prédictions présentées sur la figure 4.4(b) ne sont pas les résultats des mesures présentés sur la figure 4.4(a). Néanmoins, seules les valeurs et dates exactes varient légèrement tandis que la forme générale des résultats reste naturellement comparable.

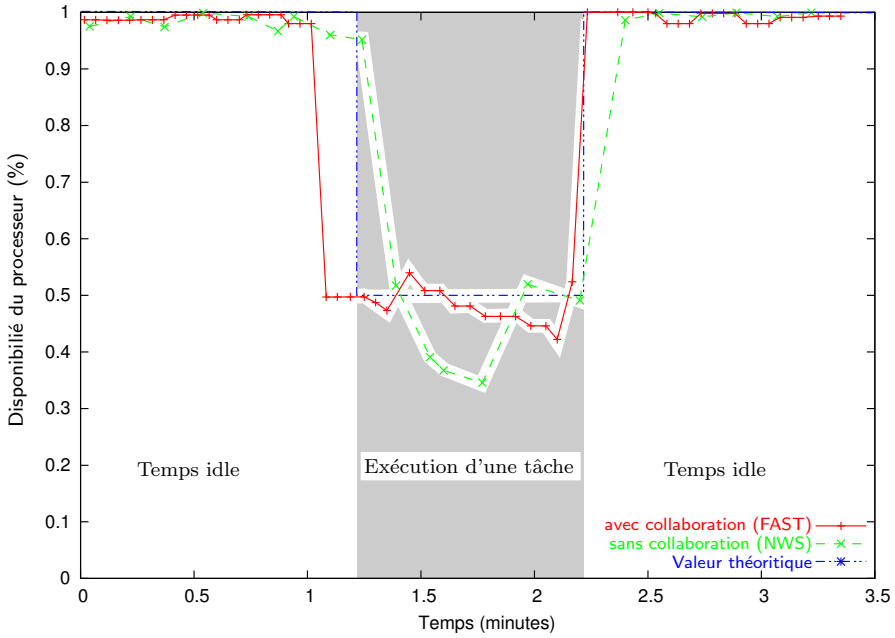
Cette expérience nécessite une analyse attentive, et nous commençons par l'étude des mesures avant de nous pencher sur les prédictions.

Analyse des mesures rapportées. Pour une meilleure lisibilité, les résultats de NWS et FAST de la figure 4.4 sont présentés séparément sur les figures 4.5(a) et 4.5(b), respectivement. La numérotation des points est consistante sur les deux figures, et le point ① de la figure 4.5(a) correspond par exemple approximativement aux mêmes conditions que le point ① de la figure 4.5(b).

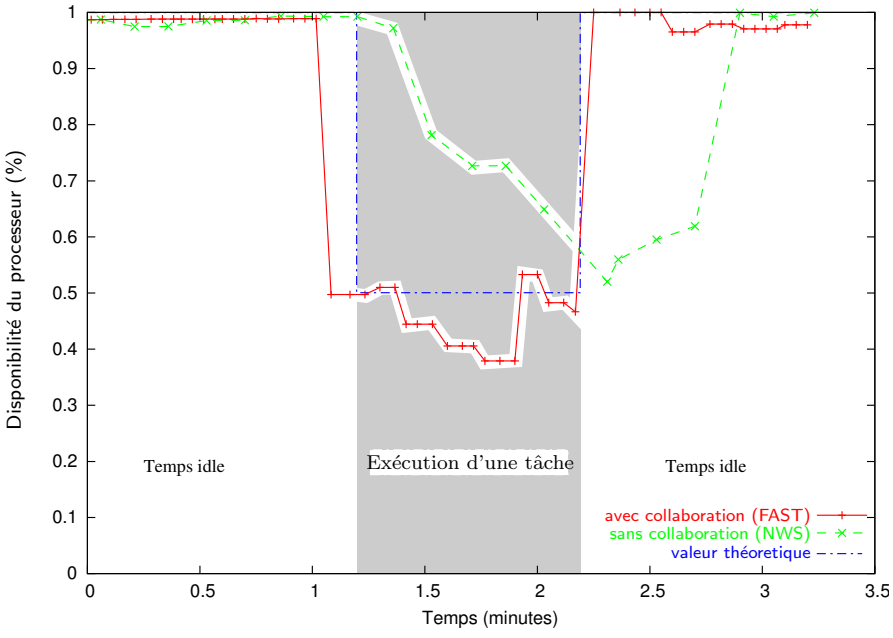
La figure 4.5(a) présente les mesures rapportées par NWS lorsque ce système ne reçoit aucune information sur les changements de condition. Avant d'expliquer ces résultats, il nous faut maintenant étudier plus précisément comment les mesures de NWS sont réalisées. Le système d'exploitation peut fournir une mesure de la charge de la machine de façon simple et sans induire de perturbations, mais cette valeur constitue la moyenne sur la dernière minute et elle est insuffisante pour prendre en compte les mécanismes de priorité entre les processus ([WSH00]). Pour corriger ces problèmes tout en limitant les perturbations dues aux mesures, NWS alterne les mesures actives (en lançant une courte tâche de mesure et en comparant le temps passé sur le processeur à la durée de l'expérience) et les mesures passives (en utilisant la valeur fournie par le système d'exploitation). Ce phénomène est observable pour les points ① et ② sur la figure 4.5(a) : ① est une mesure passive et ne tient pas compte de la charge induite par la tâche récemment lancée tandis que ② détecte ce changement.

Les tests actifs constituent de plus une forme de calibrage, et la différence notée entre la valeur rapportée par le système d'exploitation et celle mesurée est utilisée pour corriger les tests passifs suivants. Ainsi, la différence notée en ② est utilisée pour corriger les points entre ③ et ④. Malheureusement, la qualité de la valeur rapportée par le système d'exploitation va en s'améliorant dans le même temps (puisque la tâche s'exécute depuis plus longtemps). La correction est donc trop importante et induit des résultats inférieurs à la valeur théorique. Ce phénomène dure jusqu'à la mesure active suivante (④), qui permet de trouver une valeur très proche de la valeur théorique. De nouveau, ⑤ est un test passif et l'erreur détectée en ④ est utilisée pour ajuster la valeur. Cette erreur n'est toujours pas égale à zéro puisque la tâche s'exécute depuis moins d'une minute (qui est la durée de la moyenne prise par le système d'exploitation), mais elle est plus petite que celle mesurée en ② car la tâche s'exécute depuis plus longtemps. Cela explique que ⑤ soit plus proche de la valeur théorique que ③.

La tâche se termine entre ⑤ et ⑥. Ce dernier étant un test actif, il détecte la bonne valeur. À partir de ce point, la correction des tests actifs tend à augmenter la valeur des tests passifs (car la moyenne tiendra encore compte de l'exécution de la tâche alors que les mesures directes détecteront qu'elle est terminée). De la même façon que lors du lancement de la tâche, ces corrections seront trop importantes, tendant à indiquer que la disponibilité du processeur

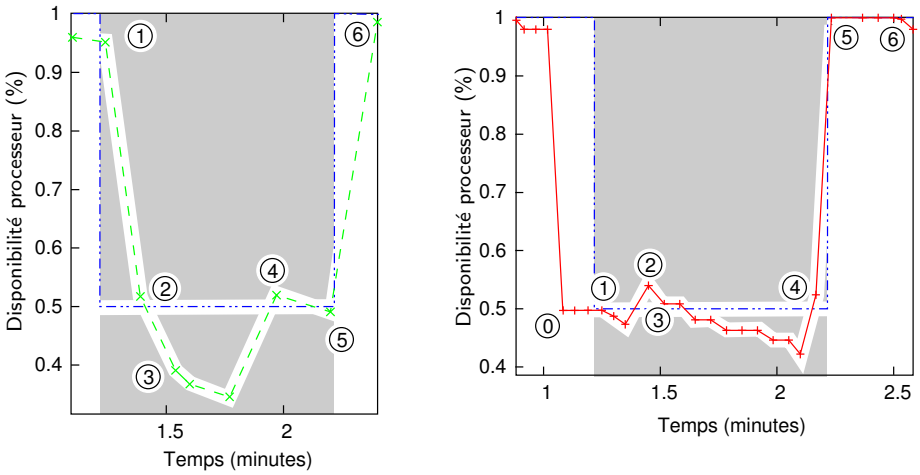


(a) Mesures.



(b) Prédictions.

FIG. 4.4 – Réactivité de NWS et FAST.



(a) Résultats rapportés par NWS.

(b) Résultats rapportés par FAST.

FIG. 4.5 – Détails sur les mesures.

dépasse 100%. Mais NWS détecte que cette machine ne dispose que d'un seul processeur, et corrige donc cette erreur grossière.

Notons également que NWS utilise plus de tests actifs qu'à l'habitude. En effet, la documentation de cet outil indique qu'un test actif est déclenché toutes les minutes. Cette différence est due au fait que nous avons activé une option non documentée permettant d'améliorer la réactivité en forçant NWS à mener un test actif lorsque les variations de la valeur rapportée par le système d'exploitation sortent d'un intervalle donné. Cette expérience montre que cette option permet à NWS de détecter un changement en cinq secondes en moyenne (la période entre les tests étant de dix secondes par défaut) au lieu de trente secondes habituellement (puisque la période entre les tests actifs est de une minute sans cette option). Mais cette option rend impossible la cohabitation de deux familles de senseurs sur la même machine, en raison d'une trop grande réactivité. Chaque senseur détecterait l'exécution de l'autre et rapporterait une disponibilité de moitié par rapport à la valeur réelle. C'est pourquoi nous avons dû nous contenter d'extraire les données de chaque expérience d'une exécution séparée.

La figure 4.5(b) détaille les valeurs rapportées par FAST. Remarquons que les valeurs sont bien plus fréquentes que dans le cas de NWS. Ceci est dû au fait que nous demandons à FAST la valeur présente dans le cache toutes les deux secondes puisque FAST n'effectue pas de mesure par lui même.

La valeur théorique est obtenue dès ⓪, c'est-à-dire quelques secondes avant que la tâche ne commence effectivement, grâce aux informations fournies par l'« ordonnanceur ». L'étude des fichiers de journaux montre que FAST applique cette correction jusqu'au point ②, qui est la première valeur renvoyée par NWS après sa mise à jour. De manière comparable à la figure 4.5(a), ① est donc un test passif de NWS indiquant une disponibilité d'environ 100%. Grâce aux informations de réservation, FAST est cependant capable de corriger cette erreur et de rapporter une valeur très proche de la valeur théorique. Bien que s'agissant d'un test actif, ② indique une valeur supérieure à la valeur théorique (tout comme ④ sur la figure 4.5(a)).

Nous ne savons pas à l'heure actuelle d'où provient cette erreur, et nous l'attribuons aux erreurs de mesure classiques.

Après ②, nous observons la même décroissance des valeurs due à la sur-correction des mesures expliquée plus haut. Le test actif suivant est ④, permettant d'obtenir une mesure très proche de la valeur théorique. Cette fois, le test passif suivant (⑤) a lieu après la fin de la tâche, et sa valeur est corrigée par FAST grâce aux informations fournies par l'ordonnanceur. Cette correction est appliquée jusqu'à ⑥, qui est le test actif suivant la réactualisation de NWS après son second redémarrage.

Un autre phénomène notable est qu'il faut dans ce cas un peu plus de temps avant que NWS ne fasse son premier test actif après le début de la tâche. Il s'agit du temps nécessaire pour effectivement redémarrer les senseurs NWS. Il serait préférable de les remettre à jour plutôt que de les tuer pour les redémarrer afin d'obtenir un premier test actif plus rapide, mais cette fonctionnalité n'est pas encore implémentée dans NWS. De plus, cela n'influe pas sur la qualité des mesures puisque FAST les corrige durant cette période.

En résumé, cette expérience montre que les interactions entre l'ordonnanceur et le système de surveillance de la plate-forme permettent d'améliorer la réactivité des mesures lors du début et de la fin des tâches ordonnancées. En dehors de ces deux événements, les mesures sont relativement comparables.

Analyse des prédictions rapportées. Après cette étude des mesures, nous étudions maintenant les prédictions à court terme comme présentées sur la figure 4.4(b). La réactivité de FAST pour les prédictions est comparable à celle des mesures puisque les mêmes corrections sont appliquées aux deux formes de résultats. En revanche, la réactivité des prédictions de NWS est relativement faible en comparaison des autres jeux de données étudiés ici. Il parvient à « prédire » la charge induite par la tâche près d'une minute après son commencement, c'est-à-dire après que la tâche ne se soit *terminée*. De la même manière, NWS a besoin d'une minute pour actualiser ses prédictions à la fin de la tâche. Ceci est dû à l'utilisation d'un historique relativement long des mesures pour les prédictions, ce qui mène à lisser les variations et à considérer les variations brutales comme des incohérences de mesures à éliminer. Au contraire, FAST se défausse de cet historique lorsque l'ordonnanceur l'informe de changements drastiques des conditions ce qui lui permet de s'adapter au mieux à de telles variations brutales.

4.3.2 Qualité des prédictions

Nous présentons maintenant plusieurs expériences étudiant la qualité de la modélisation des besoins des routines grâce au *macro-benchmarking* comme implémenté dans FAST. La première d'entre elles ne considère que la modélisation tandis que la seconde traite de la capacité de FAST de prédire le temps de complétion d'une opération en tenant compte de la charge. La troisième expérience étudie quant à elle les prédictions de temps de complétion sur une séquence d'opérations distribuées.

La plate-forme de test est composée de deux grappes de machines. *icluster* est la grappe de PC de l'équipe IMAG localisée à Grenoble et composée de bi-processeurs Pentium II dotés de 256 Mo de mémoire vive tandis que *paraski* est la grappe du laboratoire IRISA localisée

à Rennes et composée de Pentium III dotés dotés de 256 Mo de mémoire vive. Toutes ces machines utilisent le système d'exploitation Linux dans sa version 2.2. Les deux grappes sont distantes d'environ 800 kilomètres et sont interconnectées par le réseau VTHD à 2,5 Gb.

4.3.2.1 Modélisation des besoins des routines

Avant d'étudier la capacité de FAST à prendre la charge externe en compte, nous nous sommes attaché à quantifier la précision de la modélisation des besoins des routines en termes de temps et d'espace. Nous comparons ici les modélisations aux mesures pour trois opérations d'algèbre linéaire dense comme implémentées dans la bibliothèques BLAS. Il s'agit de l'addition de matrices `dgeadd`, de la multiplication de matrices `dgemm` et de la résolution triangulaire `dtrsm`.

	`dgeadd`		`dgemm`		`dtrsm`	
	icluster	paraski	icluster	paraski	icluster	paraski
Erreur max.	0.02s 6%	0.02s 35%	0.21s 0.3%	5.8s 4%	0.13s 10%	0.31s 16%
Erreur moy.	0.006s 4%	0.007s 6.5%	0.025s 0.1%	0.03s 0.1%	0.02s 5%	0.08s 7%

TAB. 4.1 – Qualité de la modélisation temporelle de `dgeadd`, `dgemm`, et `dtrsm` sur icluster et paraski.

Le tableau 4.1 résume les résultats de cette expérience concernant le temps d'exécution pour des matrices de taille variant entre 128 et 1152. Les deux premières lignes présentent l'erreur maximale à la fois en valeur absolue et en pourcentage tandis que les deux dernières indiquent l'erreur moyenne observée.

L'erreur maximale observée est généralement inférieure à 0,2 seconde tandis que l'erreur moyenne est d'environ 0,1 seconde. Ces résultats sont très satisfaisants dans notre contexte puisque nous avons vu dans la section 4.3.1 que le temps d'interrogation de NWS est de l'ordre de 0,1 seconde. L'erreur relative est inférieure à 7% et reste inférieure à 15% (sans compter le cas de `dgeadd` sur paraski, discuté plus loin). Dans le cas de la multiplication de matrices sur paraski, nous observons une erreur de 5,8 secondes, mais cela reste acceptable puisque cela ne représente que 4% du temps mesuré.

Dans le cas de l'addition de matrice, les erreurs relatives sont plus importantes. L'explication en est que nous avons préféré utiliser les mesures telles que rapportées par l'appel système `rusage` afin d'isoler les besoins des routines de la charge extérieure (comme expliqué dans le chapitre 3, section 3.2.3.1). Malheureusement, la précision des mesures n'est alors que d'environ 0,02 seconde, et comme l'addition de matrice dure moins d'une seconde, les erreurs de mesures ne sont alors plus négligeables.

FAST modélise parfaitement les besoins de ces routines en termes d'espace mémoire puisque la taille d'un programme exécutant une opération matricielle correspond à la taille du code (qui est constante) et la taille des données (qui est naturellement estimable par un polynôme fonction de la taille des matrices).

En résumé, les erreurs dues à la modélisation restent très inférieures au grain d'opérations habituellement utilisé sur la grille.

4.3.2.2 Prédiction d'une opération

Cette expérience traite des prédictions de FAST en prenant la charge externe en compte. Seuls les temps de complétion sont étudiés bien que le partage de ressources influe également sur la mémoire. Mais nous avons vu à la section 4.1.3 que lorsque la machine cible ne dispose pas d'assez de mémoire pour exécuter l'opération, FAST retourne un code d'erreur afin qu'elle ne soit pas prise en compte par l'ordonnanceur.

Nous comparons ici les prédictions de FAST avec les mesures réalisées après coup pour l'opération `dgemm`. Afin de simuler la charge externe, une tâche de calcul intensif s'exécute en parallèle de ces tests. La plate-forme de test est celle décrite plus haut : deux nœuds de de paraski et icluster (respectivement) sont utilisés.

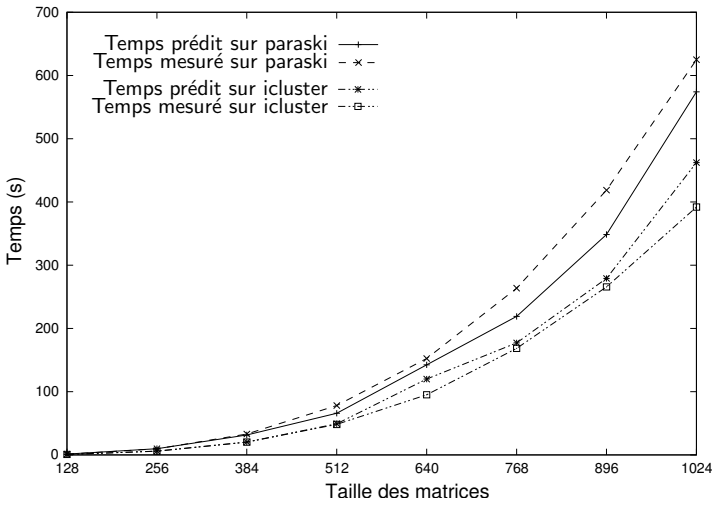


FIG. 4.6 – Comparaison des temps prédits et mesurés pour une opération (dgemm).

La figure 4.6 présente les résultats de cette expérience. Malgré la charge externe, FAST parvient à prédire le temps de complétion avec une erreur maximale de 22% (pour les matrices de petites tailles) et une erreur moyenne inférieure à 10%.

4.3.2.3 Prédiction d'une séquence d'opérations

Cette troisième expérience traite d'une séquence d'opération : la multiplication de matrices complexes. Étant donné deux matrices A et B, décomposées en parties réelles et imaginaires, il faut calculer la matrice C telle que :

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

Puisque l'objectif de l'expérience est d'étudier la précision des prédictions et non leur impact sur la qualité de l'ordonnancement, nous avons utilisé une plate-forme relativement simple mais hétérogène composée de deux machines : *Pixies* est une station de travail Pentium II disposant de 128 Mo de mémoire vive tandis que *Kwad* est un serveur de calcul doté de quatre processeurs Pentium III et 256 Mo de mémoire. Les deux machines utilisent le système

d'exploitation Linux dans sa version 2.2 et sont interconnectées par un réseau local à 100 Mb/s. Pour l'expérience, un processus client est placé sur *Pixies* tandis que deux processus de calcul sont placés sur *Kwad*. Voici le scénario de l'expérience :

- *Pixies* crée les quatre matrices A_i, A_r, B_i *et* B_r ;
- *Pixies* envoie $A_{\{i,r\}}$ et B_r au premier serveur de calcul puis $A_{\{i,r\}}$ et B_i au second ;
- Chaque serveur de calcul exécute ses multiplications de matrices : $C_{r_1} = A_r \times B_r$ et $C_{i_2} = A_i \times B_r$ pour le premier, tandis que le second calcule $C_{r_2} = A_i \times B_i$ et $C_{i_1} = A_r \times B_i$;
- Les serveurs échangent les matrices C_{i_2} et C_{r_2} ;
- Ils terminent le calcul par $C_i = C_{i_1} + C_{i_2}$ et $C_r = C_{r_1} - C_{r_2}$;
- Les résultats sont renvoyés au client sur *Pixies*.

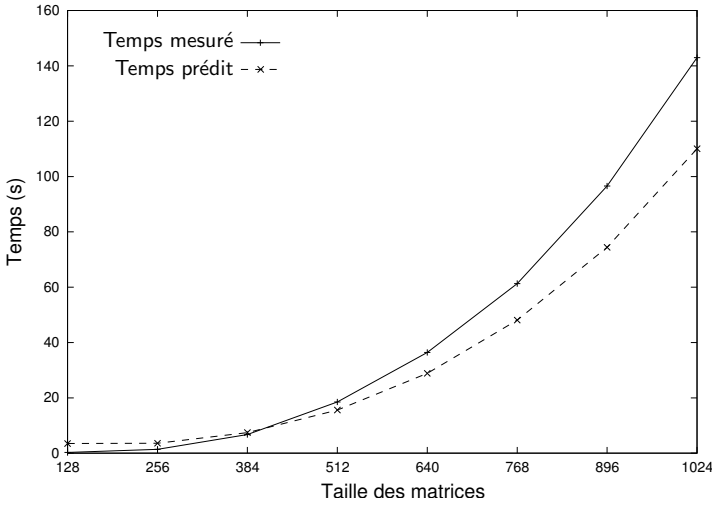


FIG. 4.7 – Comparaison des temps réels et mesurés pour une séquence d'opérations.

La figure 4.7 compare les temps prédits et mesurés pour la séquence d'opération complète. Bien que la séquence soit composée de six opérations matricielles (de deux types différents) et de six échanges de matrices sur le réseau, les deux courbes varient de façon comparable. et FAST parvient à prédire le temps de complétion de la séquence avec une erreur maximale inférieure à 25% et une erreur moyenne de 13%.

4.4 Résumé

Dans ce chapitre, nous avons présenté l'outil FAST (*Fast Agent's System Timer*), qui constitue une implémentation fonctionnelle de l'approche par *macro-benchmarking* présentée dans le chapitre 3. Il permet de modéliser automatiquement les besoins des routines séquentielles à la fois en termes de temps de complétion et d'espace mémoire occupé. Les résultats expérimentaux présentés montrent que la perte de précision due à l'automatisation de l'étalonnage et à la simplicité du modèle reste tolérable dans le cadre du *metacomputing*, où le grain des opérations est relativement gros.

Nous avons également présenté des modifications que nous avons apportées à l'outil de surveillance de la plate-forme NWS afin d'en améliorer la latence (*i.e.* le temps de réponse) et la

réactivité vis-à-vis des changements de conditions dues aux tâches placées par l'ordonnanceur. Des résultats expérimentaux nous ont permis de quantifier les gains obtenus.

Chapitre 5

Conclusion à propos de FAST

Nous étudierons dans ce chapitre les applications présentes et futures de FAST. Nous présenterons dans la section 5.1 comment FAST a été intégré à divers projets de *metacomputing*. Une extension de l'outil (principalement réalisée par Eddy Caron et Frédéric Suter) aux routines parallèles de la bibliothèque ScaLAPACK sera présentée dans la section 5.2. Nous conclurons ce chapitre ainsi que cette partie dans la section 5.3 en évoquant les futures évolutions et applications prévues de cet outil.

5.1 Intégration de FAST à d'autres projets

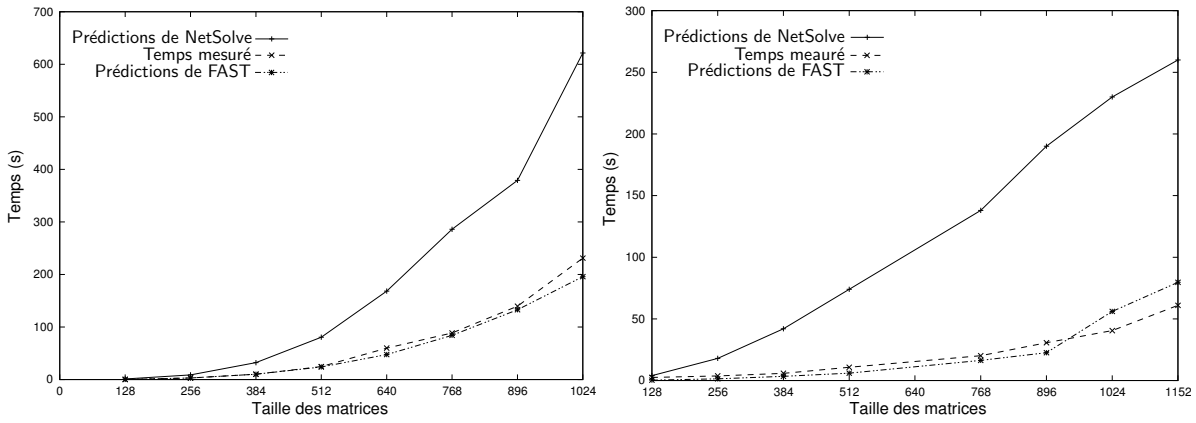
Bien que développé dans le cadre de cette thèse, FAST est un outil relativement mature, et il a fait l'objet d'une intégration dans plusieurs projets, que nous relatons ici.

5.1.1 NetSolve

Nos premières expérimentations sur FAST étaient fondées sur NETSOLVE présenté dans le chapitre 2, section 2.1.2.1. Cela nous a permis de profiter de l'infrastructure de cet outil pour réaliser l'ordonnancement à proprement parler. Nous avons pour cela intégré FAST à NETSOLVE afin de remplacer les prédictions et modèles utilisés par défaut dans cet environnement par ceux de notre outil.

L'expérience suivante compare les prédictions de la version 1.3 de NETSOLVE à celles de FAST. Comme l'objectif ici est d'étudier la précision des prédictions et non leur impact sur la qualité de l'ordonnancement en découlant, la plate-forme utilisée est relativement simple : le seul serveur existant était placé sur l'un des nœuds de la grappe paraski tandis que le client et l'agent était placé sur l'un des nœuds de la grappe icluster.

Les figures 5.1(a) et 5.1(b) présentent les prédictions de NETSOLVE et FAST pour le temps de complétion et le temps de communication (respectivement) pour la multiplication de matrices en utilisant l'implémentation de `dgemm` fournie avec NETSOLVE. Ces résultats sont très encourageants et montrent clairement les bénéfices potentiels d'outils spécifiques pour la modélisation des besoins des routines et la surveillance des disponibilités de la plate-forme.



(a) Temps de calcul.

(b) Temps de communication.

FIG. 5.1 – Prédictions de FAST et NETSOLVE pour `dgemm` comparées aux mesures réelles.

À l'heure actuelle, ces développements ne sont pas encore intégrés à la version officielle de NETSOLVE. Ceci est en partie dû au fait que notre implémentation se fonde sur les travaux d'Emmanuel Jeannot pour la persistance des données (présentées dans le chapitre 2, section 2.1.2.1 et détaillés dans [DJ01]), alors que l'équipe de NETSOLVE a choisi une autre approche pour résoudre ce problème. Cependant, une association NSF-INRIA entre l'*Innovative Computing Laboratory* de l'université du Tennessee développant NETSOLVE et le LIP de l'ÉNS-Lyon en cours devrait simplifier les collaborations entre les deux équipes et permettre une intégration complète et rapide de ces travaux à NETSOLVE.

5.1.2 DIET

FAST est utilisé dans le *middleware* de *metacomputing* DIET présenté dans le chapitre 2, section 2.1.2.3 et développé en partie au laboratoire de l'informatique du parallélisme de l'ÉNS-Lyon dans le cadre du RNTL GASP et de l'ACI GRID GRID-ASP. Cette utilisation de FAST dans un environnement de production est une chance pour l'outil de par la quantité et la qualité des retours sur expérience dont elle nous a permis de profiter.

Grâce à cette intégration, nous avons profité d'une large base de testeurs pour nos développements, nous forçant à adopter une approche plus rigoureuse de la programmation et à nous pencher sur des problèmes comme la documentation, la portabilité du code et la simplicité d'installation. Nous pensons que ce dernier problème reste l'une des barrières principales à la large adoption des méthodologies issues du *metacomputing* : de nombreux projets académiques existants résolvent certains problèmes rencontrés sur la grille, mais ils restent difficiles à mettre en place pour des non–spécialistes, ce qui empêche leur adoption en dehors de la communauté des chercheurs les ayant mis au point.

De plus, nos nombreuses discussions avec les utilisateurs, les fournisseurs de service et les programmeurs de DIET nous ont permis de mieux cerner les problèmes auxquels les utilisateurs de FAST sont confrontés et de tenter d'adapter l'outil pour le rendre plus utile. Ainsi par exemple, la fonctionnalité de réservation virtuelle présentée dans le chapitre 4,

section 4.1.1.2 est le résultat de besoins ressentis lors de l'intégration de l'application de bio-informatique ParBaum par Bert van Heukelom dans DIET (décrite dans [vH03]).

À l'avenir, ces collaborations devraient garantir que l'outil FAST ne restera pas seulement un projet développé dans le cadre de cette thèse, et offrir de nombreux débouchés à nos travaux, ainsi qu'une mine d'inspiration pour les évolutions futures.

5.1.3 Grid-TLSE

Le projet Grid-TLSE [AP03] est un effort conjoint de différents laboratoires et équipes français dans le cadre d'une ACI-GRID afin de mettre au point un système expert capable de choisir en fonction des données la meilleure implémentation parmi les différents outils existants pour résoudre des systèmes d'équations exprimés avec des matrices creuses.

Le projet, lancé en janvier 2003, prévoit d'utiliser les fonctionnalités de FAST afin de prédire les besoins des différentes implémentations existantes pour des problèmes types, et déterminer automatiquement à quelle catégorie de matrice appartiennent les données présentées au système expert. Le *middleware* utilisé pour acheminer et ordonnancer les requêtes d'expertise est DIET.

5.1.4 Outil de visualisation

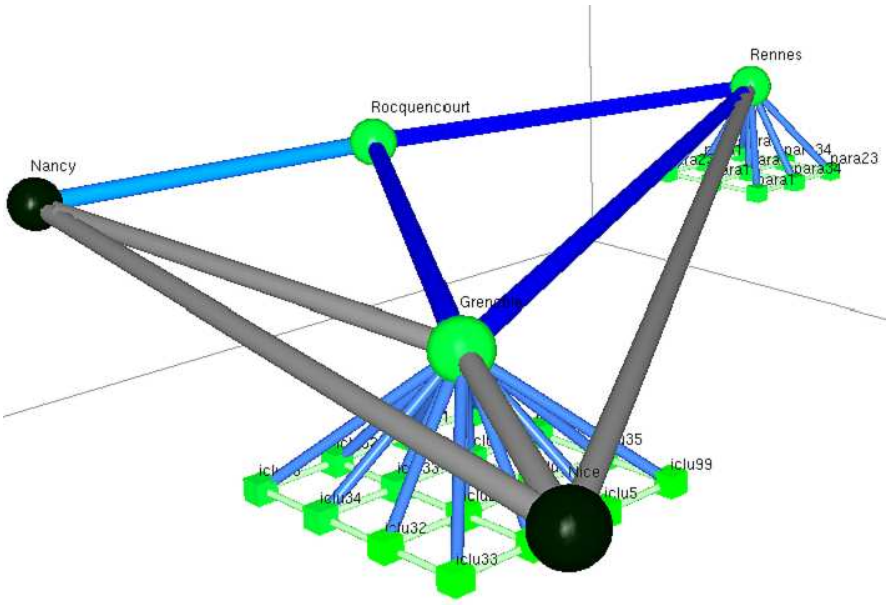


FIG. 5.2 – Capture d'écran de l'outil de visualisation fondé sur FAST.

Nous avons développé un outil de visualisation utilisant les données collectées par FAST, et les présentant en trois dimensions grâce à la bibliothèque OpenGL. Cet outil est fondé sur l'environnement Cichlid [BMB00] développé au National Laboratory for Applied Network Research (NLANR) du Supercomputing Center de San Diego (SCSD) et destiné à la création d'application de visualisation de données par immersion. Grâce à OpenGL, il est possible

de naviguer dans la représentation de la plate-forme, de zoomer sur certaines parties du réseau pour une étude plus approfondie, ou au contraire d'obtenir une vision d'ensemble en visualisant l'intégralité de la plate-forme.

Cet outil a par exemple été utilisé lors des démonstrations de DIET et des outils associés lors de SuperComputing 2002, et de la conférence IPDPS 2003.

5.2 Extension parallèle de FAST

Bien que FAST soit mieux adapté aux routines séquentielles, il a fait l'objet d'une extension parallèle principalement réalisée par Frédéric Suter et Eddy Caron, et que nous allons maintenant brièvement présenter. Pour plus de détails, les lecteurs sont invités à se référer à la thèse de Frédéric Suter [Sut02].

5.2.1 Présentation

Pour obtenir un ordonnancement satisfaisant pour une application parallèle, il est nécessaire de déterminer à priori le temps de calcul de chacune des tâches qui la composent ainsi que les temps de communication induits par le parallélisme. La technique la plus courante pour déterminer ces temps est de décrire l'application et de modéliser la machine parallèle qui l'exécute.

Il existe plusieurs approches classiques pour cela. La première d'entre elles est connue sous le nom de loi d'Amdahl. Malheureusement, les coûts de communication sont alors ignorés, ce qui ne rend pas compte de la réalité dans le cas d'une plate-forme à mémoire distribuée comme une grappe de machines. En effet, les communications représentent une fraction importante du temps d'exécution total d'une application dans ce cas. De plus, les travaux présentés dans [Sut02] montrent qu'une application trop directe de cette loi ne peut permettre de prendre en compte le schéma de communication, impliquant pourtant de grandes variations de performances selon la forme de la grille de processeurs utilisée.

Les modèles *délai* [RS87] et *LogP* [CKP+96] tiennent quant à eux compte des communications. Le premier sous forme d'un délai d constant, alors que le second considère quatre paramètres théoriques : le temps de transmission d'un processeur à un autre (L), le surcoût en calcul d'une communication (o), le débit du réseau (g) et le nombre de processeurs (P). Cependant, il nous a semblé que le modèle délai n'était pas assez précis et le modèle LogP trop compliqué pour modéliser d'une manière simple mais suffisamment réaliste une plate-forme de *metacomputing*.

L'approche choisie ici est d'identifier par analyse du code source les parties séquentielles et les communications impliquées dans l'exécution d'une routine parallèle comme celles offertes par la bibliothèque ScaLAPACK. Étant donné que dans ce cas, les parties séquentielles constituent des appels à la bibliothèque BLAS (que FAST modélise par étalonnage), il est possible de construire une expression mathématique donnant les performances d'une routine parallèle en fonction des caractéristiques des données et des performances actuelles de la plate-forme tout en tenant compte des effets de recouvrement du calcul et des communications. Nous détaillerons dans la section suivante comment cette approche a été appliquée au cas du produit parallèle de matrices denses.

Pour l'estimation des communications point–à–point, nous avons opté pour le modèle classique $\lambda + L\tau$ où λ est la latence du réseau, L la taille du message et τ le temps de transfert par élément, c'est-à-dire l'inverse de la bande passante. L peut être déterminée lors de l'analyse, λ et τ pouvant quant à eux être estimés via des appels à FAST. Dans les opérations de diffusion, λ et τ sont remplacés par des fonctions dépendantes de la forme de la grille de processeurs [CLU00]. Ainsi, sur une grappe de stations de travail connectée par un commutateur, la diffusion pourra être implantée en utilisant un arbre. Dans ce cas, la latence des diffusions sur la ligne des processeurs sera exprimée par $\lceil log_2\, q \rceil \times \lambda$ et tandis que la bande passante sera égale à $(\lceil log_2\, q \rceil\, /p) \times \tau$. Ici, λ doit être interprétée comme la latence d'un nœud et $1/\tau$ comme la bande passante moyenne.

Ce travail peut donc être considéré à la fois comme un client de FAST et comme une extension pour la gestion des routines parallèles. En effet, les estimations de la partie séquentielle de FAST sont injectées dans un modèle obtenu par analyse de code. Une fois ce couplage effectué, le temps d'exécution de la routine parallèle modélisée peut être prédit par FAST et est donc accessible par l'interface utilisateur standard.

5.2.2 Exemple de modélisation du produit de matrices denses

Nous avons choisi de ne présenter ici que la modélisation de cette opération, bien que [Sut02] traite de plusieurs autres opérations, comme la résolution triangulaire à l'aide de la fonction `pdtrsm`.

La routine `pdgemm` de la bibliothèque ScaLAPACK est la version parallèle de la routine `dgemm`. Elle calcule donc également le produit $C = \alpha op(A) \times op(B) + \beta C$ où $op(A)$ (resp. $op(B)$) peut être A ou A^t (resp. B ou B^t). Par souci de simplification, nous ne nous intéresserons ici qu'au cas $C = AB$, les autres cas étant similaires. A est une matrice $M \times K$, B une matrice $K \times N$, et la matrice résultat C est de dimension $M \times N$. Étant donné que ces matrices sont distribuées de manière cyclique par blocs sur une grille $p \times q$ de processeurs, la taille d'un bloc étant R, le temps de calcul s'exprime de la manière suivante :

$$\left\lceil \frac{K}{R} \right\rceil \times \texttt{temps\_dgemm}, \tag{5.1}$$

où `temps_dgemm` est obtenu par l'appel à la fonction `fast_comp_time` de FAST décrite dans le chapitre 4, section 4.2.1. Les matrices passées en paramètres de cet appel sont de tailles $\lceil M/p \rceil \times R$ pour la première opérande et $R \times \lceil N/q \rceil$ pour la seconde.

Pour estimer le temps de communication, il est important de considérer le schéma de communication de la routine `pdgemm`. À chaque étape, les pivots, c'est-à-dire une colonne de blocs et une ligne de blocs, sont diffusés à l'ensemble des processeurs pour permettre l'exécution des multiplications de manière indépendante. Les quantités de données communiquées sont donc $M \times K$ pour la diffusion des lignes et $K \times N$ pour la diffusion des colonnes. Chacune de ces diffusions est effectuée bloc par bloc. On obtient donc :

$$(M \times K)\tau_{ligne} + (K \times N)\tau_{col} + (\lambda_{ligne} + \lambda_{col}) \left\lceil \frac{K}{R} \right\rceil. \tag{5.2}$$

Cela nous donne donc l'estimation suivante pour la routine `pdgemm` :

$$\left\lceil \frac{K}{R} \right\rceil \times \texttt{temps\_dgemm} + (M \times K)\tau_{ligne} + (K \times N)\tau_{col} + (\lambda_{ligne} + \lambda_{col}) \left\lceil \frac{K}{R} \right\rceil . \tag{5.3}$$

Dans le cas d'une diffusion par arbre, on peut remplacer τ_{ligne}, τ_{col}, λ_{ligne} et λ_{col} par leurs valeurs en fonction de τ et λ. Ces deux quantités sont estimées par des appels à la fonction `fast_avail` de FAST. L'équation 5.2 devient alors :

$$\frac{\left(\frac{\lceil log_2\, q\rceil \times M \times K}{p} + \frac{\lceil log_2\, p\rceil \times K \times N}{q}\right)}{\tau} + \left\lceil \frac{K}{R} \right\rceil (\lceil log_2\, q\rceil + \lceil log_2\, p\rceil) \times \lambda. \tag{5.4}$$

5.2.3 Validation expérimentale

Afin de valider notre gestion des routines parallèles dans FAST, nous avons effectué divers tests basés sur le produit de matrices ScaLAPACK. Ces tests ont été exécutés sur l'*icluster*.

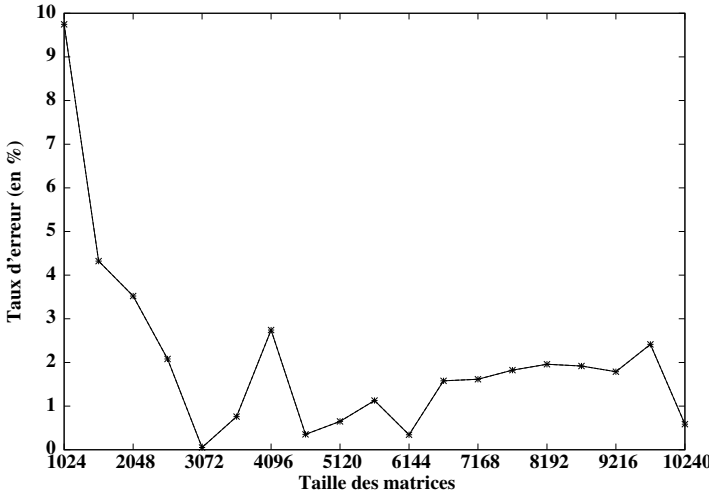


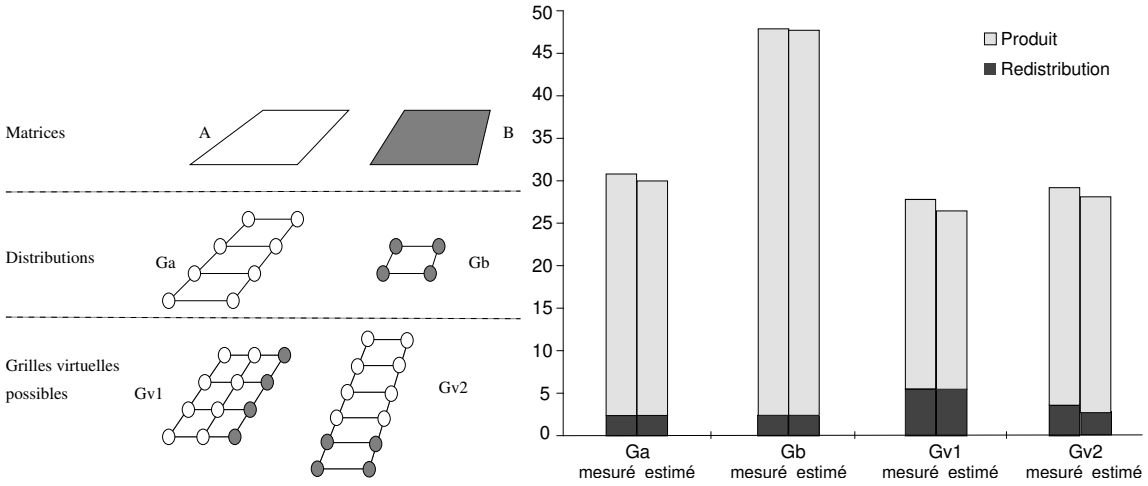
FIG. 5.3 – Taux d'erreur entre prédiction et temps d'exécution sur une grille 8×4 de processeurs pour l'opération `pdgemm`.

Dans cette expérience, nous avons cherché à valider la précision de cette extension pour une grille de processeurs donnée. La figure 5.3 montre le taux d'erreur de la prédiction par rapport au temps d'exécution mesuré pour des produits de matrices effectués sur une grille 8×4 de processeurs. Les tailles de matrices employées vont de 1024 à 10240. Cette extension s'avère très précise puisque nous obtenons un taux d'erreur inférieur à 3% lorsque la taille des matrices est suffisamment grande.

D'autres expériences présentées dans [Sut02] montre que cette extension permet de prédire les performances de `pdgemm` pour toutes les formes de grilles possibles comprenant entre 1 et 32 processeurs pour des matrices de taille 2048×2048 avec une erreur maximale inférieure à 15% et une erreur moyenne inférieure à 4%.

5.2.4 Discussion

L'objectif de FAST est de fournir des informations précises permettant à l'application cliente, typiquement un ordonnanceur, de déterminer quelle est la meilleure solution entre différentes possibilités. La figure 5.4(a) présente une configuration de ce type où la matrice A est distribuée sur la grille G_a et la matrice B sur la grille G_b. Ces deux grilles de processeurs peuvent être agrégées de différentes manières afin de former une grille virtuelle plus puissante. Nous avons retenu deux grilles pour cette expérience : une compacte, G_{v1} ; et une plus allongée, G_{v2}. Ces grilles sont en réalité des ensembles de processeurs de l'*icluster*. Les processeurs sont donc homogènes et les coûts de communication inter- et intra–grilles peuvent être considérés comme similaires.



(a) Formes de grilles virtuelles étudiées.

(b) Temps prédits et mesurés.

FIG. 5.4 – Validation de la gestion des routines parallèles dans le cas d'un alignement de matrices suivi d'un produit.

Malheureusement, la version actuelle de FAST n'est pas capable d'estimer le coût d'une redistribution entre deux ensembles de processeurs. Ce problème est en effet très difficile dans le cas général [DDP+98]. Pour les besoins de cette expérience, nous avons donc déterminé les volumes de données transmis entre chaque couple de processeurs ainsi que le schéma de communication engendré par la routine de redistribution de ScaLAPACK. Nous avons ensuite utilisé FAST pour estimer les coûts des différentes communications point–à–point générées. La figure 5.4(b) compare les temps estimés et mesurés pour chacune des grilles présentées en figure 5.4(a).

Nous pouvons constater que l'utilisation de FAST permet de déterminer avec précision quelle est la solution la plus rapide, à savoir celle utilisant une grille 4×3 de processeurs. Si cette solution est la plus intéressante en ce qui concerne le calcul, elle est en revanche la moins efficace pour ce qui est de la redistribution. L'utilisation de FAST peut donc permettre d'effectuer une présélection en fonction du ratio entre puissance des processeurs et débit réseau. De plus, il est intéressant de constater que si la solution consistant à effectuer le

produit sur G_a est un peu plus coûteuse en temps, elle génère moins de communications et libère 4 processeurs pour d'éventuelles tâches en attente.

5.3 Résumé et travaux futurs

Dans cette partie, nous avons présenté et justifié notre approche pour la collecte efficace des informations sur la grille pertinentes dans le cadre de l'ordonnancement.

La plupart des travaux existant dans la littérature tentent d'isoler les causes des variations de performances en modélisant le plus précisément possible tous les détails de la plate-forme. Les modèles résultants sont complexes et difficiles à mettre en œuvre dans un cadre interactif.

Notre approche, que nous nommons *macro-benchmarking*, met au contraire l'accent sur la mesure de ces variations et l'usage de modèles simples et robustes. Cette approche a été utilisée avec succès dans le projet NWS pour les mesures des disponibilités de la plate-forme et les expériences présentées montrent qu'elle peut être étendue à la modélisation des besoins des routines.

Nous avons également présenté l'outil FAST, qui offre une vue interactive de la grille aux autres applications. Les résultats expérimentaux présentés montre que le *macro-benchmarking* permet de modéliser les besoins des routines avec une précision suffisante au vue du grain d'opérations visé sur la grille. Cet outil est utilisé dans l'environnement de résolution de problèmes DIET, ce qui nous a permis de mieux identifier les besoins dans le cadre de l'ordonnancement sur la grille. Nous avons modifié NWS pour tenter de répondre à ces besoins. Les résultats expérimentaux présentés à ce sujet montre que la latence et la réactivité du système de surveillance de la plate-forme sont améliorées par une plus grande intégration et de meilleures interactions avec les autres outils de la grille.

L'implémentation présentée est d'ores et déjà utilisable : en plus de DIET, elle a été intégrée avec succès dans l'environnement de résolution de problème NETSOLVE, et devrait être intégrée dans le système expert Grid-TLSE. Nous avons également développé un outil de visualisation par immersion en temps réel présenté lors de plusieurs démonstrations scientifiques du produit.

Une extension parallèle de FAST aux routines de ScaLAPACK menée par Frédéric Suter et Eddy Caron a également été présentée. Les résultats expérimentaux montrent que cette extension peut non seulement être utilisée pour prédire les performances de routines parallèles, mais également pour trouver la meilleure forme de grille pour une plate-forme donnée.

Représentant plus de 15000 lignes de code, FAST est librement disponible depuis notre page web[1] sous une licence proche de celle de BSD, et est testé pour les plates-formes Linux, Solaris, Tru64, BSD et MacOS X.

Les évolutions futures de FAST comprennent un portage de l'outil pour d'autres systèmes d'exploitation tels que Irix et AIX. L'intégration des travaux sur l'extension parallèle pour ScaLAPACK dans la version de base de l'outil est également en cours. Grâce à cette expérience, nous espérons pouvoir développer un outil d'aide à l'analyse du code source pour aider les développeurs à réaliser une analyse comme celle de cette extension.

[1] http://graal.ens-lyon.fr/~mquinson/fast.html

Un autre axe d'extension serait d'ajouter d'autres métriques au système, tels que les performances d'entrée/sortie des périphériques de stockage. Cette extension semble toutefois plus compliquée à mener car la mesure par étalonnage des besoins en termes d'entrée/sortie des routines est encore plus difficile à réaliser que pour la mémoire tandis que les disponibilités à ce sujet varient encore plus vite que celles du processeur [DR03], compliquant d'autant leur surveillance.

Nous souhaiterions également étendre les senseurs NWS afin de pouvoir surveiller des liens non-TCP tels que ceux utilisant des technologies Myrinet.

Mais la conclusion la plus intéressante de ces travaux vient à notre avis du chapitre 4, section 4.1.1.3 : la manière la plus naturelle de surveiller une grille de façon extensible nécessite des connaissances précises sur la topologie de la plate-forme. L'obtention de cette connaissance est un problème relativement difficile, auquel nous consacrons la troisième partie de cette thèse.

Troisième partie

Découverte de topologie et déploiement automatique

Introduction

Pour utiliser la grille de calcul, il est d'usage de mettre en place des infrastructures distribuées rendant différents services, telles que celles présentées dans le chapitre 2. Il s'agit en particuler du GIS de Globus ([FK97a] – présenté dans la section 2.2.1) permettant de localiser les différentes ressources ou de NWS ([WSH99] – présenté dans la section 2.2.2), permettant d'obtenir des prédictions sur la disponibilités de ces ressources, ou encore de NetSolve [CD98] ou de DIET [CDL+02] (présentés respectivement dans les sections 2.1.2.1 et 2.1.2.3) permettant de placer des tâches de calcul sur ces ressources. Une caractéristique commune à tous ces systèmes est d'être constitués de processus répartis sur les différentes machines de la plate-forme et collaborant grâce à un protocole applicatif.

La connaissance de la topologie d'interconnexion des différentes machines de la plate-forme est donc essentielle pour le bon fonctionnement de ces composants. En particulier, nous avons vu dans le chapitre 4, section 4.1.1.3 que la mise en place d'une configuration extensible des tests du réseau par NWS nécessite de connaître les influences relatives des différents transferts entre des paires d'hôtes.

La grille de calcul est le plus souvent composée par plusieurs organisations mettant en commun des ressources locales réparties sur plusieurs sites interconnectés entre eux. La configuration manuelle de la plate-forme est donc une tâche difficile et nécessitant de regrouper des connaissances sur la topologie d'interconnexion détenues par les différents administrateurs de chaque organisation.

L'objectif de cette troisième partie est d'automatiser la découverte de la topologie de la grille. Les travaux présentés ici sont le résultat d'une collaboration fructueuse avec Arnaud Legrand, doctorant au LIP.

Nous commencerons par préciser nos objectifs, détailler l'état de l'art du domaine et présenter la méthodologie choisie dans le chapitre 6. Ensuite, nous relaterons dans le chapitre 7 une expérience visant à configurer automatiquement les senseurs de NWS en fonction des informations topologiques collectées par l'outil ENV [SBW99] sur une partie du réseau de notre laboratoire. L'étude des difficultés rencontrées lors de cette expérience nous amènera à la présentation d'un cadre de travail simple et efficace pour la mise au point d'applications destinées à la grille dans le chapitre 8.

Nous présenterons dans le chapitre 9 un outil de découverte de la topologie adapté à la grille fondé sur ce cadre de travail, et offrant une vue plus complète que celle offerte par ENV.

Chapitre 6

Objectifs, état de l'art et méthodologie

La grille est le plus souvent formée de différentes infrastructures rendant un service spécifique. Chacune de ces applications distribuées est constituée de processus répartis sur les différentes machines de la plate-forme. En particulier, nous avons vu dans la seconde partie que la surveillance de la plate-forme nécessite le déploiement sur les différentes machines de sondes logicielles telles que celles du projet NWS (présenté dans le chapitre 2, section 2.2.2).

Dans ce système, les tests visant à mesurer les disponibilités du réseau sont des tests actifs consistant à envoyer un bloc de données sur le réseau et à chronométrer la durée de la communication résultante. Il est donc particulièrement important de synchroniser les tests pour éviter les collisions entre les paquets de test car cela pourrait conduire chaque expérience à ne mesurer qu'une fraction de la bande passante réellement disponible.

Pour ce faire, NWS fournit un mécanisme permettant de s'assurer que les tests menés à l'intérieur d'un groupe de machines donné (nommé *clique*) n'entreront pas en collision grâce à un algorithme de passage de jeton décrit dans [WGT00]. La façon la plus simple pour éviter les collisions est donc de configurer NWS afin que toutes les machines soient placées dans la même clique, et d'assurer ainsi qu'un seul test sera mené à la fois sur toute la plate-forme. Malheureusement, cette solution n'est pas extensible puisqu'elle interdit de mener les tests en parallèle. Son application devient donc problématique lorsque le nombre de machines croît.

Étant donné que les plates-formes de *metacomputing* forment classiquement une constellation de réseaux locaux connectés entre eux par des liens à grande distance, une autre approche est de hiérarchiser les tests, et de mesurer séparément les connexions inter– et intra–site. Il est même possible de pousser cette approche plus loin, et de mettre en place des hiérarchies de senseurs à l'intérieur de chaque site pour suivre le découpage du réseau local puisque ce dernier est très souvent hiérarchique lui aussi.

La configuration hiérarchique des outils de surveillance de la plate-forme nécessite cependant une bonne compréhension des mécanismes internes de ces outils ainsi qu'une bonne connaissance de la topologie d'interconnexion des machines impliquées. Réaliser cette tâche manuellement peut donc s'avérer difficile et constituer une source d'erreurs importante. Il convient d'automatiser cette étape pour simplifier l'utilisation de nos outils et en permettre l'usage en dehors de la communauté des spécialistes de la grille.

6.1 Objectifs

Les travaux présentés dans cette partie sont le fruit d'une collaboration avec Arnaud Legrand. Nos motivations pour la découverte de la topologie de la grille sont donc doubles. En plus de nos objectifs d'automatisation du déploiement des composants de la grille, le simulateur SimGrid [CLM03] (développé en partie dans le cadre de la thèse d'Arnaud) bénéficierait grandement d'un « catalogue » de plates-formes. Il permettrait aux utilisateurs de tester leurs développements sur différentes architectures facilement. Un générateur aléatoire de plates-formes est d'ores et déjà disponible, mais le réalisme d'une plate-forme donnée étant difficile à quantifier ([PF97, CDZ97, FFF99]), un outil permettant de capturer la topologie de plate-forme réelle est le bienvenu.

6.1.1 Topologie recherchée

L'accent est mis ici sur la découverte de la topologie effective du réseau, et de ses effets sur les communications de niveau application et non sur une vue précise et détaillée du réseau tel que le schéma de cablage physique. Notre objectif n'est pas de mettre au point un nouvel outil permettant aux administrateurs du réseau de détecter son éventuel mauvais fonctionnement ou ses pannes. Le but de cet outil est plutôt de permettre aux utilisateurs d'obtenir une estimation des performances qu'ils peuvent escompter du réseau. Les composants de la grille devraient également pouvoir l'utiliser pour s'adapter automatiquement aux changements d'état de la plate-forme.

Il est bien entendu nécessaire pour cela de collecter les performances du réseau de bout en bout (*end-to-end*), c'est-à-dire le débit et la latence (regroupés sous l'appellation *connectivité*) escomptés entre tout couple de machines sur lesquelles est déployée la grille. Mais ces grandeurs ne sont pas suffisantes pour assurer le bon déploiement de NWS ou la prédiction de macro-communications (comme la diffusion par *broadcast*). Il est également indispensable de quantifier les interactions entre les flux de données. Autrement dit, étant donné quatre machines A, B, C et D, nous souhaitons non seulement connaître la bande passante sur (A, B) et (C, D), mais aussi savoir si la saturation du lien (A, B) a une influence sur les performances de (C, D).

Dans la suite de cette thèse, nous désignons par *nœuds* les machines de la plate-forme sur lesquelles les administrateurs de la grille sont autorisés à exécuter des programmes. Cela ne comprend généralement pas les machines du cœur du réseau tels que les routeurs ou les pare-feux dont l'accès est le plus souvent restreint. Notre objectif est donc de découvrir un graphe connectant les différents nœuds de la plate-forme et simulant les interactions entre flux observables dans la réalité. Il existe clairement plusieurs graphes répondant à cette définition, et notre objectif est d'exhiber l'un d'entre eux.

6.1.2 Caractérisation des contraintes de déploiement de NWS

La première contrainte pour le déploiement et la configuration de NWS est due au fait que les expérimentations sur le réseau ne doivent pas entrer en collision. Les tests ne rapportent que la capacité des liens dont leurs paquets ont pu profiter. Donc, si deux paquets de test

transitent par le même lien au même instant, il est probable que chacun ne mesure que la moitié de la capacité réelle du lien[1].

Comme nous l'avons vu précédemment, NWS permet d'éviter ce problème par l'introduction du concept de *cliques* de mesure présenté dans [WGT00]. Elles forment un ensemble de machines sur lequel les tests réseau sont menés en exclusion mutuelle grâce à un algorithme de passage de jeton : seul l'hôte disposant du jeton à un instant donné est autorisé à initier une mesure des performances du réseau. Il est donc garanti que les tests au sein de la clique ne peuvent pas entrer en collision et que seule la charge externe sera mesurée. Des mécanismes pour gérer les pertes du jeton dues par exemple à des problèmes de transmission sont également introduits.

Les algorithmes de passage de jeton sont malheureusement relativement peu extensibles, et la fréquence des mesures pour un couple de machines donné décroît naturellement lorsque le nombre d'hôtes dans la clique augmente. Les cliques doivent donc être découpées en sous–cliques pour assurer une fréquence suffisante des mesures. Cependant, il faut prendre garde à ce que des tests d'une clique donnée ne puissent interférer avec ceux menés au sein d'une autre clique.

Les utilisateurs de NWS peuvent cependant vouloir obtenir les capacités du réseau de bout en bout pour tout couple de machines. Quand aucun test direct ne mesure les capacités du lien entre un couple de machines donné, il est donc nécessaire de pouvoir agréger les mesures réalisées sur les différentes parties du chemin pour estimer les performances de bout en bout.

Par exemple, étant donné trois machines A, B et C, si B est la passerelle connectant A et C, il suffit de mesurer les liens (AB) et (BC). La latence sur (AC) peut alors être estimée en ajoutant les latences mesurées sur (AB) et (BC), tandis que la latence du chemin complet peut être estimée comme étant le minimum des bandes passantes de ses constituants. Ces valeurs peuvent s'avérer moins précises que des tests réels, mais elles restent pertinentes quand aucune mesure directe n'est disponible.

En résumé, le déploiement et la configuration de NWS doivent satisfaire les quatre contraintes suivantes :

Éviter les collisions entre expérimentations. Un lien physique donné ne peut être utilisé que par un seul test à un instant donné. Une façon d'assurer ceci est de s'assurer que tous les hôtes connectés par un même lien physique soient placés dans la même clique.

Extensibilité de l'ensemble. Les cliques doivent être aussi petites que possible de façon à maximiser la fréquence des mesures sur les différents liens afin que le système reste assez réactif aux changements de conditions.

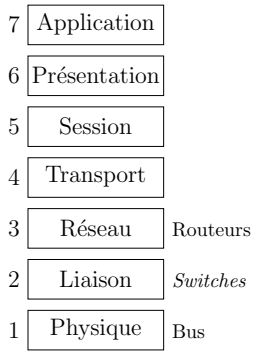
Exhaustivité. Pour tout couple de machines, si aucun test direct n'est réalisé entre elles, le système doit pouvoir estimer leur connectivité par agrégation d'autres tests.

Réduction des perturbations. Afin de réduire les perturbations induites sur la grille, il faut éviter de mener des tests inutiles. Par exemple, la bande passante étant partagée entre tous les hôtes connectés par un bus (*hub*), tous présenteront la même connectivité. Il suffit alors de mesurer la bande passante entre un couple de machines pour déduire celle de tout couple de machines sur ce bus réseau.

[1] Ce n'est cependant pas assuré car il est possible que la capacité du lien partagé dépasse grandement les besoins de l'expérimentation (par exemple limitée par un autre segment du réseau). Dans ce cas, la collision des paquets ne pose pas de problème et peut être tolérée.

La connaissance de la topologie du réseau est clairement fondamentale pour réaliser un bon déploiement de NWS.

6.2 Outils et méthodes de découverte de la topologie



Le modèle de réseau de l'OSI (*Open Systems Interconnection* – interconnexion de systèmes ouverts) présenté sur la figure ci-contre est constitué de sept couches. Chacune offre une vision différente du réseau, et il convient donc de préciser le niveau considéré puisque ce choix influe à la fois sur la topologie, sur la façon de la découvrir et sur ses utilisations possibles. Les topologies de niveaux 2 et 3 sont les plus couramment utilisées.

La couche 2 est nommée « liaison de données », correspondant au protocole *Ethernet*. La couche 3 est nommée « réseau » et correspond aux protocoles tel que le protocole internet (*Internet Protocol* – IP). La couche liaison est plus proche des liens physiques que la couche réseau et peut être utilisée pour obtenir des informations à propos des routeurs non disponibles depuis le niveau 3. En revanche, la couche réseau est plus proche de la vision que les applications ont du réseau.

6.2.1 SNMP et BGP

La couche 2 du modèle OSI est celle où les réseaux locaux (*LAN*) sont définis et configurés. Il est donc possible de demander aux composants du réseau leur configuration telle qu'exprimée par les administrateurs système en utilisant par exemple SNMP (*Simple Network Management Protocol*, protocole simple de gestion du réseau – [BJ00]). Cependant, certains routeurs ne répondent pas à de telles requêtes tandis que d'autres demandent l'usage d'outils propriétaires fournis par leurs constructeurs tel que le protocole de découverte Cisco[2] ou celui de Bay Networks[3].

Pour compléter cette vue par des connaissances sur les réseau longue distance (*WAN*), il est possible d'utiliser BGP (*Border Gateway Protocol*, protocole pour passerelles – [RL95]), utilisées pour échanger des informations sur le routage entre les systèmes autonomes composant Internet.

Le projet **Remos** [DGK+01] utilise cette approche et déduit la topologie du réseau local grâce à SNMP et celle du réseau à grande distance grâce à des étalonnages actifs comparables aux tests de NWS. De plus, ce système est capable de reconstituer les parties du réseau local correspondant à des routeurs ne répondant pas aux requêtes SNMP et obtenir ainsi une topologie complète du réseau [MS00].

Le principal avantage de cette approche est qu'elle permet d'obtenir la configuration du réseau directement de là où elle est exprimée par les administrateurs. Elle est donc relativement rapide et n'induit que très peu de perturbations sur le réseau. Malheureusement, l'usage de ces protocoles est le plus souvent restreint à un petit nombre d'utilisateurs privilégiés. Cette limitation est principalement due à deux éléments. Le premier point est la sécurité, puisqu'il

[2] http://www.cisco.com
[3] http://www.baynetworks.com

est possible de mener des attaques de type « déni de service » sur le réseau par leur biais. En effet, la capacité d'un routeur à traiter les paquets transitant sur le réseau décroit très fortement lorsqu'il est soumis à un flux continu de requêtes SNMP. De plus, les fournisseurs d'accès à Internet sont généralement réticents à permettre l'usage de SNMP sur leur réseau car cela reviendrait à publier les possibles manques de leur infrastructure, ce qui peut nuire à leur image commerciale.

Dans les faits, il est rare d'obtenir le droit d'utiliser SNMP sur les réseaux d'organisations dont on ne fait pas partie. Comme les plates-formes classiques de grilles impliquent le plus souvent plusieurs organisations bien établies telles que des universités, obtenir l'autorisation d'effectuer des opérations non classiques telles que l'accès aux protocoles de la couche liaison peut devenir très coûteux en temps et en efforts en raison de facteurs humains.

En revanche, toute solution fondée sur les couches 3 et supérieures est plus compliquée à mettre en place en raison de mécanismes tels que les VLAN. Cette technologie permet aux administrateurs de présenter une vue logique du réseau différente de la réalité physique en constituant plusieurs réseaux logiques partageant les mêmes liens physiques, ou au contraire de regrouper comme un seul réseau plusieurs liens distincts. Il est donc nécessaire de tenir compte de tels mécanismes lors de l'établissement de méthodes de cartographie automatique du réseau utilisant les couches supérieures du modèle.

6.2.2 Méthodes tomographiques

La tomographie est une méthode utilisée par exemple en imagerie médicale et consistant à reconstruire une vision tri-dimensionnelle d'un objet à partir de plusieurs vues bi-dimensionnelles. De manière comparable, différentes solutions existent pour reconstruire la topologie du réseau à partir d'informations collectées depuis différentes machines. Ces méthodes diffèrent principalement par la façon de collecter les visions locales de chaque machine.

ping Le programme `ping` est classiquement utilisé pour mesurer le temps d'aller retour (*Round Trip Time*, RTT) d'un paquet entre deux hôtes sur le réseau. Des projets tels que IDMaps [FJJ+01] ou Global Network Positioning (positionnement global sur le réseau, [NZ01]) utilisent cet outil pour cartographier le réseau par *clustering* sur cette métrique de distance, c'est-à-dire en regroupant les machines semblant placées à la même distance de certains serveurs donnés.

Cette approche ne donne malheureusement pas assez d'informations dans notre contexte puisque nous avons besoin non seulement de la structure topologique du réseau, mais également de la bande passante escomptée sur chaque lien (que IDMaps peut donner sous certaines conditions), et surtout de la façon dont plusieurs transferts concurrents interféreraient sur ces liens.

traceroute La couche 3 du modèle OSI est celle où est rendue possible l'interconnexion de différents réseaux. Pour éviter les boucles infinies, tous les paquets ont une durée de vie (*Time To Live* – TTL) déterminée à la création. Cette valeur est décrémentée par chaque routeur transmettant le paquet d'un réseau à un autre. Lorsqu'elle devient nulle, le paquet est détruit et un message d'erreur est envoyé à son émetteur.

Cette fonctionnalité est utilisée par l'outil `traceroute` (et donc également par les projets de cartographie automatique l'utilisant tels que TopoMon [dBKB02] ou Lumeta [BCW]). La plupart des routeurs indiquant leur adresse dans le message d'erreur produit quand le TTL devient nul, `traceroute` découvre ainsi tous les sauts nécessaires pour atteindre un hôte distant donné en émettant plusieurs paquets de TTL croissants.

Cette approche présente plusieurs défauts. Tout d'abord, la combinaison des résultats sous forme de graphe peut être difficile car les routeurs utilisent différentes adresses en fonction du réseau depuis lequel le paquet est envoyé. De plus, `traceroute` ne donne aucune information sur la façon dont les transferts concurrents partagent les différents liens rencontrés, ni sur les bandes passantes de ces liens. De fait, `traceroute` ne donne pas les informations dont nous avons besoin : il se concentre sur le chemin suivi par les paquets tandis que nous souhaitons acquérir une vision plus macroscopique indiquant les effets ressentis au niveau applicatif.

pathchar Ce programme est la solution proposée par Jacobson (également auteur de `traceroute`) pour collecter non seulement l'organisation du réseau, mais également la bande passante possible sur chaque segment. Tout comme `traceroute`, il fonctionne par envoi de paquets de différents TTL, tout en modulant de plus la taille des paquets. En analysant le temps nécessaire à la réception du message d'erreur, `pathchar` parvient à déduire la latence et la bande passante de chaque lien composant le chemin réseau étudié, la distribution du temps d'attente sur chaque routeur ainsi que la probabilité de perte de paquets [Dow99].

Le premier problème de `pathchar` est que la validité des traitements statistiques réalisés dépend du fait que ses paquets de test ne rencontrent que des délais négligeables sur les routeurs. Étant donné que la probabilité pour que cette condition soit satisfaite est relativement faible sur un réseau chargé, il est nécessaire d'envoyer de nombreux paquets pour s'assurer que l'un d'entre eux soit traité sans délai par tous les routeurs. Dans l'implémentation la plus courante, plus de 1500 paquets de test sont utilisés pour chaque segment du chemin réseau. Il en découle que la découverte d'un chemin constitué de plusieurs sauts sur un réseau chargé peut durer plusieurs heures. De plus, cet outil ne donne que la bande passante obtenue sur chaque lien, et ne permet pas d'obtenir de détails sur la façon dont ils seraient partagés par plusieurs flux de données concurrents.

De plus, `pathchar` (tout comme `traceroute`) donne une vue relativement microscopique du réseau, correspondant aux chemins empruntés par les différents paquets constituant le flux de données tandis que nous sommes intéressés par une vision plus macroscopique, contenant moins de détails mais plus simple à manipuler pour les utilisateurs et applications clientes.

Enfin, le problème principal de cet outil est que la création de paquets spéciaux tels que ceux utilisés lors des expérimentations nécessite des droits particuliers habituellement réservés aux administrateurs sur les machines où `pathchar` s'exécute. Ceci compromet clairement l'utilisation de cet outil pour la mise en place de l'infrastructure sur la grille.

Autres méthodes tomographiques Une autre limitation importante de l'approche basée sur `traceroute` est qu'elle utilise le protocole ICMP pour effectuer ses tests alors que certains administrateurs l'interdisent sur leurs réseaux pour des questions de sécurité. Les travaux récents de tomographie du réseau se sont donc focalisés sur la mise au point de nouvelles méthodes de mesures basées sur l'envoi de paquets classiques ne nécessitant aucune fonctionnalité particulière du réseau ciblé [Rab03].

Cette caractéristique semble très intéressante dans le contexte de la grille, mais la carte du réseau obtenue par ces méthodes ne correspond pas exactement à la vision que nous souhaitons capturer ici. En effet, l'objectif de ses auteurs est de reconstruire les chemins empruntés par les paquets sur le réseau tandis que nous souhaitons trouver quels flux interfèrent entre eux. Ces deux notions sont très proches, puisque si deux flux ne partagent pas de lien sur le réseau, ils ne sauraient interférer l'un sur l'autre, mais il est possible que deux flux partagent un lien sans pour autant interférer. Cette situation peut survenir si le lien commun est sur-dimensionné et si chaque flux est limité par une autre section de son trajet. Dans ces conditions, le lien partagé est capable de transmettre les deux flux sans que leurs performances ne s'en ressentent. Cette différence d'objectif rend les méthodes classiques de tomographie inapplicables dans notre contexte.

6.2.3 Mesures passives

Comme nous l'avons vu dans le chapitre 3, section 3.1.2, il est possible d'obtenir des informations sur la connectivité entre les machines de manière passive, c'est-à-dire sans injecter de trafic supplémentaire sur le réseau mais en se basant sur les performances obtenues lorsque les applications utilisent le réseau. Nous avons également justifié dans cette section pourquoi nous avons choisi de ne pas utiliser cette approche pour les mesures de bout en bout.

Ces méthodes semblent également mal adaptées à la cartographie automatique du réseau car elles offrent des informations relativement parcellaires sur l'état du réseau, et il serait extrêmement difficile de reconstruire les informations manquantes pour obtenir une vision complète de la topologie.

6.3 Résumé et méthodologie retenue

La plupart des travaux précédents sur la grille mettent l'accent sur les performances de bout en bout du réseau plutôt que sur l'obtention de la topologie d'interconnexion et son impact sur les transferts parallèles [WSH99, Din02]. Les composants de la grille reposent le plus souvent sur une configuration manuelle pour obtenir ces informations, ce qui est une source d'erreur importante.

Afin d'automatiser la cartographie de la plate-forme, il est possible d'obtenir les informations directement de la configuration telle qu'exprimée par les administrateurs grâce aux protocoles SNMP et BGP, mais cela requiert des autorisations spécifiques (en raison de problèmes de sécurité et de confidentialité), ce qui peut s'avérer problématique dans un contexte de *metacomputing*.

Des outils tels que `ping`, `traceroute` ou `pathchar` permettent d'obtenir des éléments de réponse en induisant relativement peu de perturbations sur le réseau, mais ces informations sont trop parcellaires pour l'usage prévu dans le cas des deux premiers outils tandis que le troisième nécessite des privilèges spécifiques sur la machine utilisée pour confectionner les paquets de test, interdisant son usage dans le cadre du *metacomputing*.

Comme dans le cas de l'étude des performances de chaque hôte ou des performances du réseau de bout en bout (étudiées dans la partie précédente), nous avons donc décidé de nous focaliser sur les mesures des performances que les applications peuvent escompter de

la plate-forme plutôt que sur l'établissement d'un modèle complet. Cette approche n'est pas complètement nouvelle dans ce domaine, et elle a déjà été utilisée dans le cadre du projet de cartographie automatique **Effective Network View** (ENV – [SBW99]) fondé par Gary Shao sous la direction de Francine Berman et Richard Wolski à l'Université de Californie de San Diego (UCSD).

Nous reviendrons plus en détail sur ce projet dans le chapitre suivant, qui présente une expérience de déploiement automatisé de NWS à l'aide de ENV sur notre laboratoire.

Chapitre 7

Expérience de déploiement de NWS grâce à ENV

Ce chapitre présente une expérience réalisée en collaboration avec Arnaud Legrand et Julien Lerouge au printemps 2002 et visant à automatiser le déploiement de NWS avec l'outil de cartographie automatique ENV sur le réseau de notre laboratoire. Ce déploiement peut être découpé en trois phases. Avant toute chose, il est nécessaire de cartographier le réseau. Ensuite, il faut planifier la façon dont les processus seront répartis sur le réseau avant d'appliquer effectivement ces décisions dans une troisième étape.

La section 7.1 présente ENV, la section 7.2 détaille le fonctionnement de cet outil grâce à un exemple de mise en œuvre sur le réseau de notre laboratoire. Nous verrons ensuite dans section 7.3 comment les informations fournies par ENV ont été utilisées pour déployer NWS. Enfin, nous conclurons ce chapitre en analysant cette expérience et les principaux problèmes rencontrés.

7.1 Présentation d'ENV

Le projet **Effective Network View** (ENV [SBW99]) a été développé par Gary Shao sous la direction de Francine Berman et Richard Wolski à l'Université de Californie à San Diego (UCSD). Son principe de base est de mesurer directement les interférences entre les flux de données. Pour cela, la bande passante normale entre deux machines données est comparée à celle obtenue lorsqu'un autre lien (entre deux autres machines) est saturé. Une variation indique alors que les deux liens partagent des ressources réseau, ce qui constitue une information capitale pour reconstruire la topologie d'interconnexion des machines.

L'avantage principal d'ENV est qu'il offre la vue du réseau telle qu'elle peut être ressentie par les applications en se fondant uniquement sur des expérimentations de niveau applicatif ne nécessitant donc pas de privilèges ou d'outils spéciaux pour être menées.

En revanche, obtenir une vue complète de la topologie nécessite une quantité rédhibitoire d'expériences. Il faut tester les éventuelles interactions du lien connectant chaque couple de machines avec chaque autre lien, ce qui conduit à un algorithme en $O(n^4)$ pour n machines. De plus, il est nécessaire de laisser le réseau se stabiliser pour chaque expérience, et un

algorithme naïf ne permet de faire que quelques expérience par minutes. Dans ces conditions, la cartographie d'une plate-forme constituée de vingt hôtes nécessite jusqu'à cinquante jours.

Pour résoudre ce problème, ENV ne cherche pas à donner une vue complète du réseau, mais simplement le point de vue d'une machine donnée, ce qui est suffisant dans le paradigme maître/esclaves visé par ENV. Cette simplification (ainsi que d'autres optimisations détaillées dans la section suivante) permet à ENV de mener à bien la cartographie d'une plate-forme d'une vingtaine de machines en quelques minutes. Nous reviendrons dans la conclusion de ce chapitre sur les implications de cette simplification, et nous présenterons dans le chapitre 9 une généralisation de cet algorithme ne souffrant pas des limitations induites.

Implémenté dans le langage de programmation Python, ENV est basé sur les extensions Python pour la grille (*Python Extensions for the Grid* – PEG) qui ont été développées dans le cadre du projet AppLeS [BW97] dont ENV fait partie. PEG constitue un ensemble de modules Python et d'utilitaires destinés à simplifier l'écriture d'applications d'ordonnancement pour la grille. De plus, la cartographie du réseau est rendue disponible dans une forme spécialisée de XML nommée GridML. Il s'agit d'un format flexible décrivant les caractéristiques observées des ressources du réseau constituant la grille. Plus d'informations sur le sujet sont disponibles depuis la page web du projet[1].

7.2 Exemple d'exécution d'ENV

Nous allons maintenant détailler comment ENV collecte les données, en utilisant la cartographie d'une partie du réseau de l'ÉNS-Lyon comme exemple. Ce processus est découpé en deux phases principales où la première collecte les données ne dépendant pas du choix du point de vue dans le réseau tandis que la seconde complète la vision obtenue par des expériences choisies en fonction d'un maître spécifié par l'utilisateur.

7.2.1 Topologie structurelle : cartographie ne dépendant pas du maître

Cette phase se découpe à son tour en trois étapes.

7.2.1.1 Résolution de nom

Pendant cette étape, le cœur de la représentation en GridML est créée en utilisant les noms de machines fournis. Par exemple, la représentation suivante serait créée pour les machines *canaria* et *moby* :

```
1 <?xml version="1.0"?>
2 <GRID>
3     <SITE domain="ens-lyon.fr">
4         <LABEL name="ENS-LYON-FR" />
5         <MACHINE>
6             <LABEL ip="140.77.13.229" name="canaria.ens-lyon.fr">
7                 <ALIAS name="canaria" />
8             </LABEL>
9         </MACHINE>
```

[1] http://apples.ucsd.edu/env/

```
10          <MACHINE>
11              <LABEL ip="140.77.13.82" name="moby.cri2000.ens-lyon.fr">
12                  <ALIAS name="moby" />
13              </LABEL>
14          </MACHINE>
15      </SITE>
16  </GRID>
```

La grille est représentée par un nœud de type `GRID`, divisé en nœuds de type `SITE`, et chacun des hôtes y est représenté par un nœud de type `MACHINE`.

7.2.1.2 Collecte d'informations supplémentaires

Des informations sur les hôtes tels que le système d'exploitation ou le type de processeur sont ensuite collectées. Il est possible d'étendre ENV pour lui permettre d'obtenir toute information pertinente pour l'usage prévu.

```
1  <MACHINE>
2     <LABEL ip="140.77.13.92" name="pikaki.cri2000.ens-lyon.fr">
3         <ALIAS name="pikaki" />
4     </LABEL>
5     <PROPERTY name="CPU_clock" value="198.951" units="MHz" />
6     <PROPERTY name="CPU_model" value="Pentium Pro" />
7     <PROPERTY name="CPU_num" value="1" />
8     <PROPERTY name="Machine_type" value="i686" />
9     <PROPERTY name="OS_version" value="Linux 2.4.19-pre7-act" />
10    <PROPERTY name="kflops" value="17607" />
11 </MACHINE>
```

7.2.1.3 Topologie structurelle

Il s'agit d'une première approximation de la topologie construite à l'aide de `traceroute`, dont le principal objectif est de guider les tests actifs menés dans les phases suivantes. Chaque hôte de la plate-forme indique le chemin réseau entre lui-même et une machine extérieure choisie arbitrairement (dans l'implémentation actuelle, il s'agit d'un serveur de l'UCSD). Ces chemins sont ensuite combinés et les hôtes utilisant le même chemin pour sortir du réseau local sont placés dans la même branche de l'arbre. Cela permet d'obtenir une vision arborescente de la grille comme celle présentée dans la figure 7.1

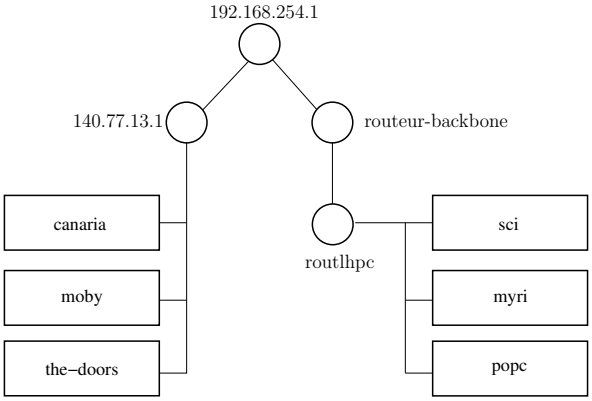


FIG. 7.1 – Topologie structurelle : arborescence initiale dans ENV.

Dans notre exemple, un test impliquant les machines *canaria*, *moby*, *the-doors*, *sci*, *myri* et *popc* produit la représentation GRIDML suivante :

```
<NETWORK type="Structural">
   <LABEL ip="192.168.254.1" name="192.168.254.1" />
   <NETWORK>
       <LABEL ip="140.77.13.1" name="140.77.13.1" />
       <MACHINE name="canaria.ens-lyon.fr" />
       <MACHINE name="moby.cri2000.ens-lyon.fr" />
       <MACHINE name="the-doors.ens-lyon.fr" />
   </NETWORK>
   <NETWORK>
       <LABEL ip="140.77.161.1" name="routeur-backbone" />
       <NETWORK>
           <LABEL ip="140.77.12.1" name="routlhpc" />
           <MACHINE name="sci.ens-lyon.fr" />
           <MACHINE name="myri.ens-lyon.fr" />
           <MACHINE name="popc.ens-lyon.fr" />
       </NETWORK>
   </NETWORK>
</NETWORK>
```

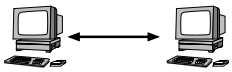
Cette arborescence n'est pas suffisante puisqu'elle ne donne par exemple aucune information sur la façon dont les liens sont partagés. Pour compléter cette vue, il est nécessaire de mener des expériences supplémentaires.

7.2.2 Topologie effective : cartographie dépendant du maître

La seconde phase de la collection des données dépend du choix de la machine maître. Ces expériences peuvent être vues comme des raffinements successifs de la topologie structurelle afin d'obtenir une vision nommée topologie effective et contenant des informations sur les couches inférieures du modèle OSI.

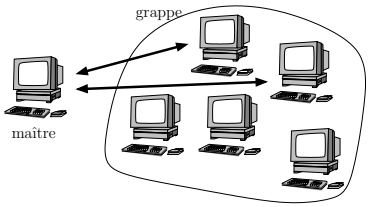
La plupart de ces mesures se fondent sur des seuils expérimentaux arbitraires. Leurs valeurs ont un impact important sur les résultats de la cartographie et ont été déterminés expérimentalement par les auteurs d'ENV.

7.2.2.1 Bande passante de bout en bout



Cette expérience raffine le découpage constitué lors de la première phase de l'algorithme en *clusterisant* les machines présentant une connectivité comparable avec le maître. La bande passante entre le maître et chaque hôte est mesurée séparément. Si le ratio des bandes passantes entre deux hôtes donnés est supérieur à 3, ils sont placés dans deux sous-groupes séparés.

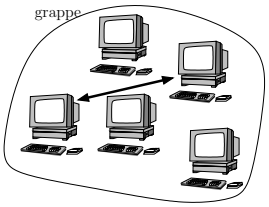
7.2.2.2 Bande passante par paire



Cette expérience raffine encore le découpage en fonction de la façon dont le lien entre les membres du groupe et le maître est partagé. Pour cela, pour chaque paire de machines A et B dans chaque groupe, les bandes passantes de MA et MB sont mesurées en effectuant les transferts de manière concurrente.

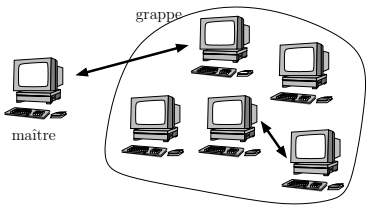
Cette mesure est ensuite comparée à celle obtenue lors de l'étape précédente. Si le ratio $Bande\_Passante(MA)/Bande\_Passante_{//(MB)}(MA)$ est inférieur à 1.25, les deux machines A et B sont déclarées indépendantes et leur groupe est coupé afin de les séparer.

7.2.2.3 Bande passante intra-groupe



Les caractéristiques des communications à l'intérieur de chaque groupe sont obtenues en mesurant la bande passante pour chaque couple de machines du groupe. Cela permet de détecter les groupes ayant une bande passante locale différente de celle obtenue lors de communications avec le maître. Dans notre exemple, la route entre *the-doors* et *popc* passe par un lien à 10 Mb/s tandis que les différentes machines de *popc* sont connectées entre elles par un réseau à 100 Mb/s.

7.2.2.4 Mesure du partage des liens



Pour chaque groupe, la bande passante avec le maître est mesurée lors d'un transfert entre deux autres membres. Cette expérience est répétée cinq fois et la moyenne du ratio $Bande\_passante/Bande\_passante_{\text{partagée}}$ est calculée.

Si cette valeur est inférieure à 0.7, le groupe est considéré comme étant connecté par un lien partagé de type bus, où la bande passante est partagée entre les différents transferts concurrents. Si elle est supérieure à 0.9, ENV considère que le lien interne au groupe est de type *switch*, ce qui signifie que les

transferts concurrents n'ont aucune influence les uns sur les autres. Si le ratio est compris entre 0.7 et 0.9, une erreur est rapportée et l'étude de ce groupe cesse car les mesures ne sont pas suffisamment significatives.

Voici un exemple de description d'un réseau de type *switch* créé lors de cette expérimentation (il s'agit du groupe des machines *sci*).

```
...
<NETWORK type="ENV_Switched">
  <LABEL name="sci0" />
  <PROPERTY name="ENV_base_BW" value="32.65" units="Mbps" />
  <PROPERTY name="ENV_base_local_BW" value="32.29" units="Mbps" />
  <MACHINE name="sci1.popc.private" />
  <MACHINE name="sci2.popc.private" />
  <MACHINE name="sci3.popc.private" />
  <MACHINE name="sci4.popc.private" />
  <MACHINE name="sci5.popc.private" />
  <MACHINE name="sci6.popc.private" />
</NETWORK>
...
```

7.2.3 Améliorations apportées à ENV

Lors de notre expérience, nous avons identifié plusieurs problèmes gênant l'utilisation d'ENV. Cette section présente ceux pour lesquels nous avons réussi à apporter une réponse tandis que les autres sont détaillés dans la conclusion de ce chapitre.

7.2.3.1 Pare-feux

Comme les pare-feux empêchent toute communication, il n'est naturellement pas possible de réaliser tous les tests lorsque certaines machines sont protégées par un pare-feu. Comme le montre la figure 7.2(a), le réseau de l'ÉNS-Lyon est découpé en deux sous-domaines `ens-lyon.fr` et `popc.private`. La plupart des hôtes du second domaine ne peuvent communiquer avec l'extérieur, et elles doivent utiliser les machines *myri0*, *popc0* ou *sci0* comme passerelles.

Nous avons donc lancé ENV sur chaque sous-domaine séparément. Une nouvelle représentation GridML regroupant chaque partie est ensuite générée en utilisant le fichier d'alias présenté ci-après pour fusionner les résultats après coup.

```
popc.ens-lyon.fr popc0.popc.private
myri.ens-lyon.fr myri0.popc.private
sci.ens-lyon.fr sci0.popc.private
```

Cela nous a permis de générer la représentation GridML suivante :

```
<GRID>
    <LABEL name="Grid1" />
```

```
3     <SITE domain="ens-lyon.fr">
4         <LABEL name="ENS-LYON-FR" />
5         <MACHINE>
6             <LABEL ip="140.77.12.52" name="myri.ens-lyon.fr">
7                 <ALIAS name="myri" />
8                 <ALIAS name="myri0.popc.private" />
9             </LABEL>
10            ...
11        </MACHINE>
12        ...
13    </SITE>
14    <SITE domain="popc.private">
15        <LABEL name="POPC-PRIVATE" />
16        <MACHINE>
17            <LABEL ip="192.168.81.50" name="myri0.popc.private">
18                <ALIAS name="myri0" />
19                <ALIAS name="myri.ens-lyon.fr" />
20            </LABEL>
21            ...
22        </MACHINE>
23        ...
24    </SITE>
25 </GRID>
```

7.2.3.2 Machines sans nom d'hôte

Lors de l'utilisation de `traceroute` à la première étape, les hôtes sont supposés faire partie du même domaine (et donc être utilisés pour la représentation structurelle) s'ils partagent le même nom de domaine. Malheureusement, certains d'entres eux ne sont pas configurés pour disposer d'un nom de domaine complètement qualifié et leur adresse IP figure dans les résultats de `traceroute`. Nous avons donc modifié ENV pour qu'il considère les classes d'adresse IP ([KSR90]) lorsque la résolution de nom échoue.

Nous avons également modifié ENV pour lui permettre de traiter correctement les adresses IP non routables. De telles adresses ne peuvent être atteintes que depuis le réseau local et doivent donc également être conservées lors de la première étape. Par exemple, la racine de l'arborescence structurelle représentée par la figure 7.1 est une adresse de cette catégorie. Il est donc clair que supprimer de telles adresses peut avoir un impact important sur la qualité de la cartographie obtenue.

7.2.3.3 traceroute sans réponse

D'après [LOG01], certains routeurs modernes ne répondent pas aux requêtes de `traceroute`. Cependant, ENV parvient à contourner ce problème car les groupes obtenus grâce à `traceroute` sont ensuite raffinés par d'autres tests. Ce manque d'information n'a donc d'influence que sur la représentation structurelle intermédiaire, mais pas sur la représentation effective finale. En revanche, cela rallonge la durée des tests puisque la bande passante entre chaque paire de machines possible dans chaque groupe doit être mesurée séparément. Le nombre d'expériences de la seconde phase augmente donc avec la taille des groupes constitués lors de la première, ce qui implique un temps d'exécution plus important.

7.2.4 Cartographie obtenue

La figure 7.2 présente les résultats de l'exécution d'ENV sur le réseau de l'ÉNS-Lyon. La figure 7.2(a) montre le réseau physique tel que connu par les administrateurs de notre système tandis que la figure 7.2(b) est le résultat de la cartographication automatique grâce à ENV en choisissant *the-doors* comme maître. La topologie physique présentée est simplifiée, et les routes non symétriques ainsi que les réseaux virtuels de type VLAN ne sont pas représentés.

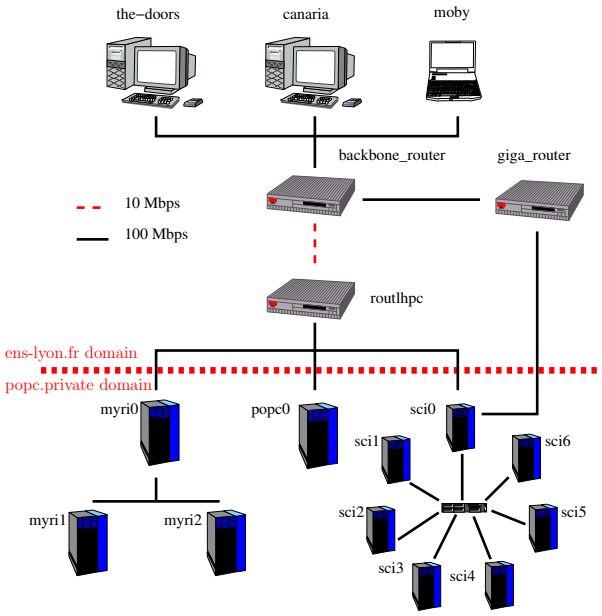
7.3 Déploiement de NWS grâce à ENV

Cette section présente un algorithme simple pour déployer NWS automatiquement à l'aide des informations fournies par ENV. Pour chaque groupe de machines identifié comme constituant un réseau par ENV, notre déploiement contient une clique :

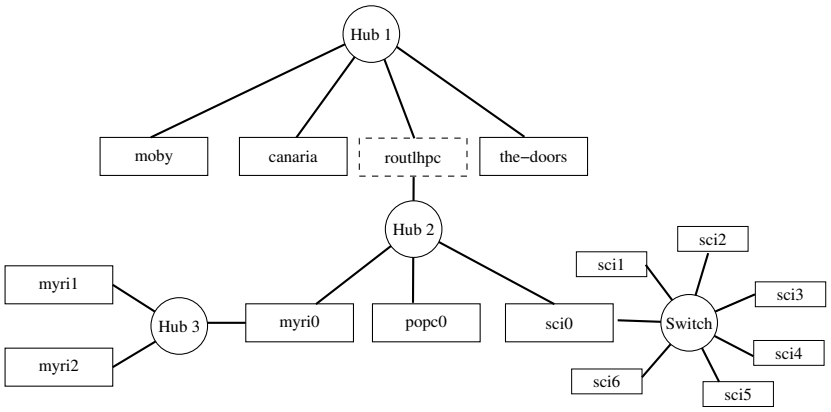
- Si le réseau est partagé (*i.e.*, s'il est formé d'un bus), tous les hôtes partagent la même connectivité. Il suffit donc de mesurer la bande passante et la latence entre un couple de machines donné pour connaître celles entre tout couple de machines. La connectivité intra-groupe est alors mesurée par une clique formée de deux machines choisies arbitrairement.
  Bien que ces mesures suffisent pour mesurer toutes les informations nécessaires, cette méthode révèle un défaut de NWS puisqu'il n'est pas possible à l'heure actuelle d'indiquer au système que la connexion entre deux hôtes (AB) est représentative de celle entre deux autres machines (CD). Il reste de la responsabilité de l'utilisateur de faire cette substitution manuellement. Ce problème devrait être corrigé dans une future version de FAST, en attendant une éventuelle correction dans NWS directement.
- Si le réseau n'est pas partagé (*i.e.*, s'il est formé d'un *switch*), les caractéristiques du réseau pour chaque couple de machines sont indépendantes, et peuvent être mesurées en parallèle. Cependant, il est important qu'un hôte donné ne soit pas impliqué dans plus d'une expérimentation à un instant donné. C'est pourquoi nous déployons alors une clique couvrant toutes les machines du réseau pour nous assurer qu'une expérience au plus sera menée à la fois sur cet ensemble.
  Il s'agit cependant d'une restriction un peu plus forte que nécessaire. En effet, sur un réseau de type *switch*, les expériences (AB) et (CD) n'interféreraient pas entre elles si elles impliquent des machines différentes, *i.e.* si $\{A, B\} \cap \{C, D\} = \emptyset$. L'usage d'une clique implique donc que la fréquence des tests soit légèrement inférieure à ce qu'elle pourrait être si NWS offrait un mécanisme plus adapté pour synchroniser les tests entre eux.

Le plan de déploiement ainsi obtenu pour le réseau de l'ÉNS-Lyon est représenté par la figure 7.3. La grappe *sci* est connectée par un *switch*, et toutes ses machines sont donc regroupées dans une clique. En revanche, la grappe *myri* est connectée par un bus, et seules deux machines sont utilisées pour obtenir les performances du lien interne à la grappe. Les machines *myri0* et *popc0* ont été choisies pour mesurer les performances du *bus2* tandis que *moby* et *canaria* mesurent celle du *bus1*. La connexion entre *canaria* et *popc0* est utilisée pour mesurer le lien entre ces deux bus.

Une fois que le plan de déploiement a été calculé de la sorte, une autre difficulté est de l'appliquer effectivement. Il faut tout d'abord lancer les différents processus de NWS sur



(a) Topologie physique (schéma simplifié).



(b) Topologie effective du point de vue de *the-doors*.

FIG. 7.2 – Topologie physique et effective d'une partie du réseau de l'ÉNS-Lyon .

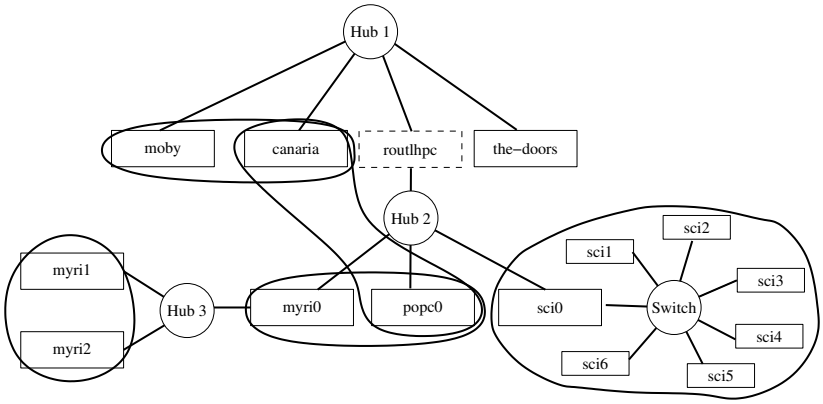


FIG. 7.3 – Déploiement de NWS à l'ÉNS-Lyon.

chaque hôte de la plate-forme puis démarrer les cliques prévues dans le plan de déploiement. Les outils NWS offrent très peu d'aide pour cette tâche, et l'utilisateur doit se connecter manuellement sur chaque machine pour y lancer les programmes en leur passant les bonnes options.

Afin de simplifier cette tâche, nous avons développé un programme de gestion des processus NWS permettant d'appliquer sur chaque machine la configuration spécifiée dans un fichier centralisé. Le déploiement effectif de NWS consiste alors simplement à distribuer le fichier de configuration global sur chaque machine (par exemple en utilisant NFS ou ssh) puis à lancer le gestionnaire sur chacune d'entre elles.

7.4 Résumé et problèmes ouverts

Dans ce chapitre, nous avons relaté une expérience de cartographie automatique d'une partie du réseau de notre laboratoire avec ENV. Nous avons détaillé la méthodologie de cet outil, et nous avons présenté un algorithme permettant le déploiement de NWS en fonction des informations ainsi obtenues. Ce déploiement révèle certaines limitations de l'algorithme évitant que les tests de NWS n'entrent en collision sur le réseau. L'étude de ces problèmes devra faire l'objet de travaux futurs.

ENV semble être l'outil de cartographie automatique existant le plus adapté à nos besoins, puisqu'il ne repose que sur des expériences réalisables sans privilège particulier et est capable de quantifier la façon dont sont partagés les différents liens de la plate-forme entre les flux concurrents. Nous avons de plus étendu ENV pour traiter les problèmes dus aux pare-feux et aux adresses IP impossibles à résoudre en nom de domaine complètement qualifié.

Cependant, cet outil souffre encore de nombreux défauts limitant son utilité dans notre contexte, et le reste de cette partie est consacrée à la recherche d'éléments de réponse à ces problèmes.

Tout d'abord, ENV est implémenté dans le langage de programmation Python, ce qui semble à première vue un bon compromis pour la portabilité sur les différentes plates-formes puisque le même code peut s'exécuter sur chacune d'entres elles sans modification.

Notre expérience pratique est relativement différente, et exécuter ENV sur un ensemble de machines aussi homogène que celui utilisé pour cette expérience (où toutes les machines utilisent Linux sur l'architecture i386) s'est avéré relativement difficile. Nous avons évidemment dû tout d'abord installer l'interpréteur Python sur toutes les machines, car il n'est pas installé sur tous les serveurs de calcul par défaut. Ensuite, ENV nécessite des modules non standards développés dans le cadre du projet AppLeS et dont la compilation demande un soin particulier.

Les plus optimistes penseront sans doute que ce problème est amené à se résoudre de lui-même puisque le langage Python est de plus en plus répandu, mais les pessimistes objecteront que le langage évolue également et que la compatibilité ascendante entre les versions n'est pas assurée. Il est donc possible que ces problèmes empirent avec le temps. De plus, de nombreux langages de script peuvent constituer une couche de portabilité potentielle, mais il reste difficile de déterminer lesquels vont survivre et devenir prédominants dans les années à venir.

Ces constatations nous ont amené à développer un cadre de travail destiné à l'établissement de programmes pour la grille réalisés en C. Nous reviendrons plus en détail sur ce sujet dans le chapitre 8.

De plus, cette expérience nous a également permis de mettre en valeur certains problèmes inhérents à l'algorithme utilisé par NWS pour s'assurer de l'absence de collision entre les tests mesurant les performances du réseau. En effet, il permet de s'assurer qu'un seul couple de machines d'un ensemble donné mèneront des mesures à un instant donné. Cependant, si le réseau est connecté par un bus, il est inutile de mesurer tous les couples possibles, et les tests réalisés entre deux machines de l'ensemble peuvent être utilisés pour tout l'ensemble. De la même manière, si cet ensemble est connecté par un *switch*, il est possible de mesurer plusieurs liens en parallèle à condition qu'un hôte donné soit impliqué dans au plus une expérience à un instant donné.

Le principal avantage de cet algorithme de verrouillage est sa simplicité de mise en œuvre lorsque l'utilisateur connaît mal la topologie du réseau sous-jacent. En revanche, dans le cadre d'un déploiement automatique réalisé par un outil disposant de toutes les informations nécessaires, il serait intéressant de modifier cet algorithme afin de permettre le verrouillage d'un hôte donné, comme proposé par exemple dans [HPBG02].

Enfin, la validité des résultats d'ENV reste à quantifier précisément. Les informations qualitatives et topologiques sur le réseau sont naturellement plus importantes que les estimations quantitatives de bande passante puisque notre objectif est de déployer NWS et que cet outil offre des informations bien meilleures à ce sujet.

Le premier problème est l'évolution possible de la plate-forme et les pannes intermittentes. La cartographie repose sur de nombreux tests dont l'interprétation dépend de leurs résultats respectifs, et une variation avant la fin de l'expérimentation peut mener à des résultats erronés. La solution choisie par ENV est la rapidité. La cartographie de notre plate-forme ne dure que quelques minutes, et nous supposons donc que les conditions ne changent pas de façon drastique sur une durée aussi courte. Il est de plus possible de réaliser plusieurs fois l'expérience pour valider les résultats obtenus. Pour des plates-formes plus grandes, il serait sans doute possible de cartographier les différentes parties séparément et de fusionner les résultats après coup comme nous l'avons fait dans le cas des réseaux séparés par des pare-feux. Une autre solution serait d'intégrer les outils de cartographie de la plate-forme à l'infrastructure de

surveillance de ses disponibilités, afin de leur permettre de détecter les évolutions par des tests réguliers.

L'établissement des seuils peut également s'avérer problématique puisque ceux-ci peuvent dépendre de caractéristiques de la plate-forme comme le type de média utilisé. Il est ainsi possible que les valeurs utilisées ici et déterminées par les auteurs d'ENV soient adaptées aux réseaux locaux, mais mènent à de mauvais résultats sur les réseaux à très haut débit. Quantifier l'adéquation des seuils à une plate-forme reste cependant un problème entier à ce jour.

Un problème plus profond d'ENV est qu'il suppose que les routes du réseau sont symétriques et ne différencie donc pas les routes (AB) et (BA). D'après [PF97], cette supposition ne tient malheureusement pas sur Internet où il est relativement commun de trouver des routes asymétriques du fait de problèmes de configuration ou autre. Ainsi, le réseau physique de l'ÉNS-Lyon est plus complexe que la représentation qu'en donne la figure 7.2(a). Par exemple, la route allant de *the-doors* à *popc* passe par un lien à 10 Mb/s tandis que la route dans l'autre direction n'emprunte que des liens à 100 Mb/s.

Comme ENV ne mène de tests de bande passante que dans une seule direction, il ne peut détecter de telles configurations. Résoudre ce problème demanderait de ré–écrire une grande partie des tests réalisés.

De la même manière, rien ne garantit que la route entre deux machines est unique et que tous les paquets d'un flux emprunteront le même chemin sur Internet. Comme ENV est destiné à cartographier des réseaux locaux, nous supposons toutefois que la route entre deux machines ne changera pas pendant la durée de l'expérimentation, ou, au moins que les mesures ne seront que faiblement affectées par ce problème.

De plus, la consommation de bande passante d'ENV reste importante et peut constituer une gène pour les utilisateurs du réseau. Cependant, elle semble constituer la seule solution possible pour obtenir des informations sur le réseau relevant de la couche 2 du modèle OSI sans nécessiter d'autorisations spéciales. Heureusement, cette étape ne doit être réalisée qu'une fois pour un réseau donné et les résultats peuvent être partagés entre différents utilisateurs. De plus, une meilleure intégration des outils de cartographie automatiques à NWS devrait permettre de réaliser ces expériences de nuit, lorsque le réseau est peu utilisé.

Enfin, le problème principal d'ENV à nos yeux reste son orientation maître/esclaves afin de réduire la complexité algorithmique du programme. Malheureusement, cette simplification implique évidemment que seule une vision partielle de la plate-forme peut être obtenue. Ainsi, la figure 7.4 présente un exemple de plate-forme où cette approche ne permet pas d'obtenir une vision complète de la topologie. Elle est constituée d'un maître et de deux grappes de machines. Les liens A et B placés entre le maître et les grappes peuvent être découverts par ENV, mais le lien C interconnectant les deux grappes ne sera pas traité par ENV puisque ce dernier ne mène aucune expérience entre les différents groupes de machines.

Cette perte d'informations semble être le prix à payer pour la rapidité d'exécution de l'algorithme puisque nous avons vu que l'algorithme intuitif mesurant tous les liens possibles de la plate-forme ne serait pas applicable en pratique. Cette situation reste peu satisfaisante,

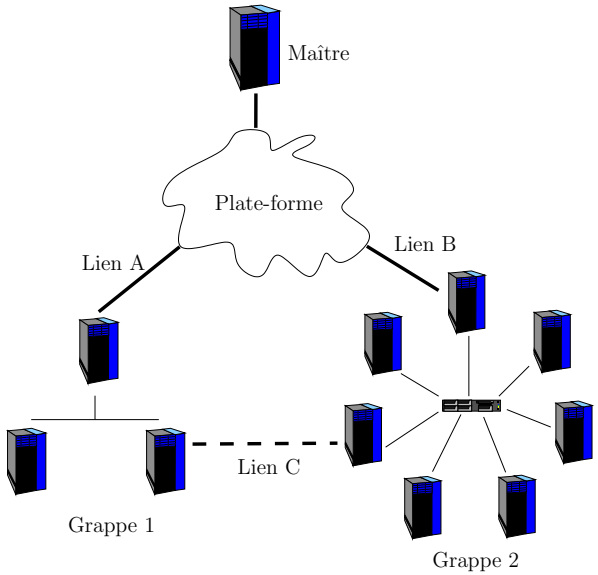


FIG. 7.4 – Exemple de plate-forme ilustrant les limitations de l'approche maître/esclaves.

et nous présenterons dans le chapitre 9 une approche débouchant sur une vision complète de la topologie.

Chapitre 8

Un environnement de développement pour la grille

La nature complexe et changeante de la grille impose aux utilisateurs l'usage d'infrastructures logicielles spécifiques comme celles présentées dans le chapitre 2. Il s'agit par exemple des GIS et MDS de Globus pour la découverte des ressources et données existantes, NWS pour la surveillance des disponibilités de ces ressources, les environnements NetSolve et DIET pour l'ordonnancement et l'exécution à distance de tâches, ou encore ENV (présenté dans le chapitre précédent) pour découvrir la topologie d'interconnexion des machines. Ces infrastructures, constituées d'applications distribuées à grande échelle faiblement couplées, sont difficiles à développer et mettre au point.

Certaines infrastructures existantes telles que NWS utilisent une couche de communication et un protocole applicatif spécifiquement développés pour leurs besoins. De manière comparable, NetSolve repose sur l'usage des *sockets* Unix classiques et de la bibliothèque de conversion des données XDR, tandis que la plupart des projets du groupe AppLeS (tel que l'environnement de déploiement d'applications massivement parallèles *AppLeS Master/Worker Application Template* [CLZB00a]) utilisent la bibliothèque AMPIC (*AppLeS Multi-Protocol Interprocess Communication*, communication inter–processus et multi–protocoles d'AppLeS). Cependant, de telles bibliothèques étant développées pour les besoins spécifiques d'applications données, il peut être difficile de les réutiliser dans d'autres contextes. Le problème est que les standards établis, tels que MPI pour le calcul distribué haute performance ou Corba pour l'invocation de méthodes à distance, n'ont pas été conçus pour prendre en compte les spécificités de la grille telle que la dynamicité et sont donc mal adaptés à ce contexte.

Par ailleurs, l'étude de ces infrastructures est encore compliquée par la complexité des algorithmes sous-jacents ainsi que par l'instabilité de la grille, empêchant de reproduire une expérience dans des conditions comparables. L'établissement d'une plate-forme d'expérimentation demande souvent des efforts très importants de la part des développeurs. Une solution pour s'abstraire de ces difficultés techniques est d'utiliser un simulateur, mais les travaux d'implémentation résultants restent le plus souvent des prototypes nécessitant une réécriture quasi complète du programme pour le rendre utilisable sur la plate-forme réelle.

Ce chapitre présente le cadre de travail GRAS (*Grid Reality And Simulation* – réalité et simulation de la grille) dont l'objectif est de simplifier le développement de certaines appli-

cations distribuées. Sa principale caractéristique est de permettre l'exécution du même code à la fois dans le simulateur SimGrid [CLM03] et sur les plates-formes réelles, grâce à deux implémentations spécifiques de la même interface. Cette solution combine le meilleur des deux mondes : les développeurs bénéficient de la simplicité d'utilisation et de contrôle du simulateur pour réaliser des infrastructures réellement utilisables. Bien que GRAS ne prétende pas résoudre tous les problèmes posés par la grille, nous pensons que son approche combinant la simulation et la réalité est la clé pour le développement rapide d'infrastructures utiles pour la grille.

La section 8.1 détaille les objectifs et les approches de ce projet tandis que la section 8.2 compare ces travaux à l'état de l'art du domaine. La section 8.3 illustre l'utilisation de GRAS sur un exemple d'application. La section 8.4 détaille quant à elle l'état actuel du prototype réalisé et ses futures évolutions.

8.1 Présentation de GRAS et de ses objectifs

Comme noté dans l'introduction, GRAS est conçu pour permettre le développement d'infrastructures robustes offrant un service spécifique sur la grille. Ces application distribuées sont constituées de différentes entités placées sur chaque machine de la plate-forme et collaborant au moyen d'un protocole applicatif.

Le modèle SPMD (*Single Program, Multiple Data* – un seul programme, plusieurs jeux de données) classique en calcul à haute performance sur des grappes de machines n'est pas le plus adapté à de telles applications, plus simplement décrites de manière *événementielle* (en associant des réponses spécifiques du programme aux différents événements pouvant survenir) ou de façon guidée par le flot de contrôle [Tel00]. Ainsi, les applications utilisant l'environnement GRAS sont constituées de fonctions spécifiques (souvent nommées *callback*) définissant les actions à réaliser lorsqu'un événement donné (comme la réception d'un message ou un changement d'état interne) se produit. Cette approche, bien adaptée à la réalisation d'infrastructure distribuée, implique que l'exécution du programme n'est pas forcément linéaire et peut même s'avérer non déterministe puisque l'ordre de messages de provenance différente peut varier en fonction de la charge externe. Cela peut compliquer l'adaptation de codes existants fonctionnant par passage de messages à l'aide des bibliothèques classiques dans ce domaine telles que PVM ou MPI, mais l'objectif de GRAS n'est pas de permettre l'adaptation de code fortement couplé à la grille, mais plutôt de constituer une solution simple pour une classe d'applications spécifique.

La suite de cette section revient plus précisément sur les objectifs de GRAS et leurs implications au niveau de l'implémentation.

8.1.1 Simulation efficace et transparente

Étant attendu que l'utilisateur exécute son application un grand nombre de fois dans le simulateur pendant la phase de mise au point, il est nécessaire que la simulation soit la plus rapide possible.

L'efficacité d'un simulateur est quantifiée par ce qu'il est d'usage de nommer le *facteur d'accélération*, mesuré comme le ratio entre le temps nécessaire à la simulation et le temps

simulé, c'est-à-dire la durée de l'exécution dans le monde simulé. Naturellement, plus la simulation est précise et moins ce ratio est important, impliquant qu'une simulation donnée demandera plus de temps pour son exécution. Ainsi, les simulateurs complets de l'architecture jusqu'au niveau de la micro-instruction (tel que SimOS [Her98]) ou du paquet réseau (tels que NS [BBE+99] ou DaSSF [LN01]) sont ceux présentant les ratios les plus bas, et sont donc mal adaptés à nos besoins. Au contraire, SimGrid [CLM03] utilise des modèles relativement simples, ce qui lui permet d'offrir un ratio d'accélération de l'ordre de 500 pour des simulations simples impliquant quelques dizaines d'hôtes et s'exécutant sur un hôte relativement rapide.

SimGrid constitue une boîte à outils offrant toutes les fonctionnalités nécessaires à la simulation d'applications distribuées dans un environnement hétérogène. Elle est formée d'un simulateur efficace d'événements discrets et d'une interface de plus haut niveau construite sur ces fondations mais plus orientée vers la simulation d'applications distribuées. SimGrid simule la charge externe de chaque ressource par fonction donnant sa puissance au cours le temps. L'instant auquel une tâche donnée se termine est ensuite calculé par intégration de la fonction de disponibilité sur le temps. Comme la fonction donnant la disponibilité représente les traces capturées par des outils tels que NWS, cette approche est dite **guidée par la trace**.

Une simulation typique construite grâce à SimGrid est formée de différents processus simulés s'exécutant sur différentes machines et inter-agissant par échange de messages. Chaque machine est décrite par sa puissance de calcul, quantifiée en Mflop/s[1]. Les machines sont interconnectées par un réseau défini par ses performances en termes de bande passante (en Mo/s) et de latence (en secondes). Toutes ces grandeurs peuvent varier au cours du temps afin de modéliser la charge externe à l'application. Les tâches de calcul sont ensuite simplement modélisées par la quantité d'opérations élémentaires à leur complétion tandis que les messages sont modélisés par leur taille.

Lors du déroulement de la simulation, les fonctions attachées par l'utilisateur à chaque processus simulé sont exécutées les unes après les autres. Elles sont bloquées lorsqu'elles demandent à recevoir ou à émettre un message ou encore à simuler l'exécution d'une tâche. À ce moment, le processus principal de SimGrid est réactivé afin de calculer l'avancement du monde simulé. Lorsque l'un des processus simulé est débloqué par la fin de la communication ou exécution qu'il réalisait dans le monde simulé, le processus principal l'active à son tour.

La durée d'exécution des fonctions de l'utilisateur doit donc être reportée dans le monde simulé. Ainsi, lorsque le code utilisateur effectue dans la réalité un calcul représentant W Mflop, le processus simulé doit être bloqué pendant W/ρ secondes virtuelles si la machine puissance de la machine exécutant la simulation est de ρ Mflop/s. Pour cela, un programme GRAS est annoté au moyen de macros C indiquant que la durée d'une séquence d'instructions doit être reportée dans le monde simulé. La durée dans la réalité peut être chronométrée automatiquement ou fixée par l'utilisateur.

Il est également possible d'indiquer de cette façon qu'une séquence d'instructions ne doit être réalisée que lorsque le code s'exécute sur une plate-forme réelle et omise dans la simulation, ou inversement. Cette fonctionnalité est particulièrement intéressante pour accélérer

[1] Nous avons vu dans le chapitre 3, section 3.2.1 les limites de la précision de cette approche, mais l'efficacité du simulateur est ici privilégiée par rapport à sa précision.

la simulation. En particulier dans le cas d'une application réalisant un produit de matrice distribué, le simulateur n'est utilisé que pour valider la coordination des différentes tâches et le résultat final importe peu. Dans ce cas, certaines parties du code annotées par l'utilisateur ne seront pas exécutées dans le simulateur. Seuls les processus simulés seront bloqués pour la durée correspondante à leur exécution.

Une autre implication du modèle choisi pour les simulations est que le système d'exploitation doit naturellement être virtualisé afin que les appels système interagissent avec le monde simulé et non avec le véritable système d'exploitation exécutant la simulation. Par exemple, l'appel système `time` utilisé pour obtenir l'heure dans les programmes doit utiliser l'horloge virtuelle de la simulation et non l'horloge de la machine servant d'hôte à la simulation, dépourvue de signification au sein de la simulation.

Même s'il est possible de surcharger automatiquement et de façon transparente les appels système réalisés par le code utilisateur, GRAS impose l'usage de fonctions spécifiques enveloppant les différents appels système. Cette approche est plus simple à réaliser, et ces enveloppes forment de plus une couche d'abstraction garantissant une meilleure portabilité du code. En effet, le prototype exact et la sémantique de certains appels système varient entre les systèmes d'exploitation forçant les programmes à tenir compte de ces différences pour être portables. Les enveloppes GRAS autour des appels système constituent donc une interface consistante et portable masquant certaines difficultés pratiques au programmeur.

Les appels système concernés sont entres autres les fonctions `time`, `sleep` ainsi que les fonctions de gestion des processus `fork`, `join` et `wait`. Le système de fichiers doit également être virtualisé pour éviter les conflits lorsque deux processus simulés accèdent à des fichiers de même nom placés sur des machines simulées différentes. Les fonctions ayant trait au réseau pourraient également être virtualisées, mais puisque GRAS offre des fonctionnalités de haut niveau pour l'échange de messages, le code utilisateur ne devrait pas utiliser directement le réseau au travers de l'interface du système d'exploitation, qui n'est donc pas virtualisée.

Enfin, les différents processus simulés sont placés par SimGrid dans des processus légers (*threads*) différents. Étant donné que le simulateur les exécute de manière séquentielle, il n'est pas nécessaire d'utiliser de *mutexes* pour protéger les sections critiques, mais le partage du même espace de *nommage* demande des précautions spéciales pour la gestion des variables globales afin d'éviter les collisions de *nommage* entre les différents processus. Pour cela, les variables globales de chaque processus doivent être placées dans une structure spécifique. Des fonctions spécifiques de GRAS doivent être utilisées pour son allocation et pour y accéder.

8.1.2 Simplicité d'usage

L'un des objectifs principaux de GRAS est d'être simple à utiliser. Nous souhaitons offrir une interface programmation de haut niveau masquant autant que possible les détails d'implémentation aux utilisateurs afin de leur permettre de se concentrer sur les difficultés algorithmiques de leurs applications.

Pour cela, GRAS est fondé sur une interface de passage de messages de haut niveau. Une application typique est alors composée d'une phase d'initialisation où tous les messages sont déclarés auprès de GRAS, et des fonctions sont attachées aux événements que constituent

la réception des différents types de messages. Ensuite, la boucle principale de GRAS est appelée afin qu'elle écoute les messages en provenance du réseau et exécute les fonctions correspondantes. Il est également possible d'attendre explicitement un message d'un type donné sans passer par ce mécanisme afin de simplifier l'écriture de fonctions nécessitant des accusés de réception de la part d'entités distantes. Les messages reçus pendant ces attentes explicites sont stockés automatiquement pour être traités plus tard.

De plus, le contenu des messages GRAS est fortement typé, et sa structure doit être communiquée lors de l'enregistrement d'un message auprès du système. Nous reviendrons dans la section 8.3 sur la façon de déclarer cette structure.

Puisque GRAS ne vise qu'à simplifier le développement d'applications distribuées, aucun mécanisme de découverte dynamique des services n'est offert, et les types de messages ainsi que leur sémantique doivent être connus de tous les participants avant le lancement de l'application. Par exemple, le processus émetteur et le récepteur peuvent inclure tous deux un fichier d'en-tête décrivant les messages échangés. Les problèmes d'interopérabilités induits seront discutés dans la section 8.4.

Afin de simplifier l'usage de cet environnement, GRAS offre aussi les services de base nécessaires à de nombreuses applications distribuées, tels que la gestion des journaux (*logs*, dans l'esprit du projet log4j [Gro01]), des primitives de gestion des erreurs, des types de données avancés (tels que des tableaux dynamiques et des tables de hachage) ou la gestion de la configuration des différents modules.

Enfin, l'une des difficultés principales de la grille vient de sa nature hétérogène, et les outils utilisés doivent donc être portables sur un grand nombre de systèmes d'exploitation. En ce qui concerne GRAS, ce problème est double : tout d'abord, l'environnement de travail et d'exécution lui-même doit être portable et les programmes développés dans ce cadre doivent eux aussi être portables. Le premier point est résolu par l'usage des outils modernes de configuration automatique tels que *autoconf* et *automake*, une implémentation en C standard et l'absence de toute dépendance sur des bibliothèques n'étant pas installées par défaut sur les systèmes modernes. Comme nous l'avons vu dans la section précédente, la simulation impose de virtualiser les appels système, et les enveloppes proposées à cet effet par GRAS constituent une couche de portabilité permettant aux programmes développés dans ce cadre d'être portables sur les différents systèmes d'exploitation.

8.2 État de l'art

De par ses objectifs, GRAS est apparenté à différents domaines tels que les simulateurs de plates-formes et les outils d'aide à la mise au point de programmes distribués. Cette section présente certaines solutions existantes et les compare à notre outil.

8.2.1 Émulation et simulation de grille

Les projets les plus proches de GRAS sont sans doute EmuLab [WLS+02] ou MicroGrid [SLJ+00], dont la fonction principale est de permettre aux utilisateurs de la grille (c'est-à-dire les programmeurs d'applications de calcul sur la grille et non les programmeurs des

différentes infrastructures constituant la grille) d'exécuter leur application sur une grille virtuelle dont le comportement peut être contrôlé plus simplement que la grille réelle.

Ces logiciels permettent de virtualiser toutes les ressources de la grille en termes de mémoire, calcul ou communication. Ceci est rendu possible par l'interception logicielle de tous les appels système liés à l'utilisation des *sockets* ou à la résolution de nom (`gethostbyname`). L'approche retenue s'apparente plus à de l'émulation qu'à de la simulation puisque tous les calculs et les communications sont effectivement réalisés. L'ordonnanceur de chaque système contrôle la vitesse à laquelle les calculs doivent être effectués et utilise le mécanisme des priorités UNIX pour tenir compte des vitesses relatives des différentes machines simulées tandis qu'un simulateur de réseau (comme DaSSF [LN01] dans le cas de MicroGrid) est utilisé pour intercepter les communications. Elles sont alors ralenties artificiellement afin de simuler la distance sur le réseau entre les processus simulés.

Étant donné que la plate-forme est entièrement simulée (en la « repliant » sur une grappe de machines de taille inférieure) et que l'application est effectivement exécutée, le facteur d'accélération de MicroGrid est toujours inférieur à 1. Le temps nécessaire au test d'une application sur une grande variété d'environnements peut donc devenir prohibitif.

Afin de permettre aux utilisateurs d'exécuter leurs applications aussi souvent que nécessaire pendant la phase de mise au point, nous avons décidé de fonder nos efforts sur le simulateur SimGrid. Comme nous l'avons vu dans la section 8.1.1, cette solution offre un facteur d'accélération de plusieurs ordres de magnitude supérieur à celui de MicroGrid en utilisant des modèles relativement simples. De plus, GRAS permet d'annoter le code des applications de façon à omettre certaines sections inutiles à l'avancée de la simulation telles que le calcul effectif de la tâche ordonnancée lors de la simulation d'un environnement d'ordonnancement distribué pour la grille. L'omission de telles sections n'est pas possible dans MicroGrid. De plus, l'approche guidée par la trace pour simuler la charge externe est bien plus souple et efficace que les manipulations sur la priorité des processus réalisées par MicroGrid pour les calculs ou la simulation complète des paquets sur le réseau implémentée par DaSSF.

Dimemas [BEG+03] est l'outil de prédiction des performances associé à l'environnement de mise au point d'application **Vampir** [KOKM02] de Pallas GmbH. Il vise à permettre l'analyse du comportement de l'application sur une autre plate-forme que celle sur laquelle elle a été testée en interpolant la trace du comportement de l'application capturée par Vampir aux caractéristiques de la nouvelle plate-forme. Cela peut donner de bonnes indications sur l'extensibilité de l'application, mais ne constitue pas un simulateur ciblé pour la mise au point d'applications.

8.2.2 Bibliothèques de communication

GRAS constituant entre autres une solution de communication, nous allons maintenant le comparer aux bibliothèques classiquement utilisées à cette fin, telles que **PVM** [SDGM94], **MPI** [Mes93] ou encore **Madeleine** [Aum02]. Ces outils ont été conçus pour les grappes de machines. Elles sont donc particulièrement adaptées aux applications présentant des schémas de communication et d'exécution réguliers et potentiellement fortement couplés destinés à fonctionner sur des plates-formes relativement homogènes. Au contraire, GRAS cible plus particulièrement les applications faiblement couplées utilisant des schémas de communication et d'exécution potentiellement très irréguliers. De plus, l'encodage des données tient une place

prépondérante dans GRAS puisque tous les messages échangés sont structurés. Pour gérer les communications entre des architectures matérielles différentes, PVM et MPI utilisent le format XDR (*eXternal Data Representation*, représentation externe des données – [Mic87]) pour convertir les données du format local de l'émetteur en une représentation standardisée, elle-même reconvertie dans le format local du récepteur ensuite. Ces conversions multiples ont un impact négatif sur les performances des communications que nous souhaiterions éviter comme nous le verrons dans la section 8.4. La bibliothèque Madeleine ne traite quant à elle pas des problèmes de conversion de données entre les architectures.

La bibliothèque **PBIO** (*Portable Binary Input/Output*, entrée/sortie binaires portable – [EBS02]) constitue une solution très efficace pour l'encodage et les conversions de données entre les architectures. Elle permet d'envoyer des structures C traditionnelles dans la représentation native de l'émetteur auxquelles sont adjointes des méta-données permettant au récepteur de les convertir dans sa propre représentation si elle diffère de celle de l'émetteur. De plus, PBIO optimise la conversion en générant les routines de conversion en assembleur lors de l'exécution. Ces performances ne sont cependant pas obtenues au détriment de la flexibilité : lorsque la structure reçue contient plus de champs que celle attendue, la bibliothèque ignore les champs supplémentaires afin que la communication se produise malgré tout. Pour cela, les méta-données contiennent le nom des structures émises.

Cette solution pour la flexibilité ne nous semble cependant pas satisfaisante car si des champs sont ajoutés à la structure, il est probable que la sémantique des autres champs soit également modifiée. Masquer cette disparité à l'application nous semble donc une source de problèmes importante. Par ailleurs, PBIO impose certaines restrictions sur les types de données qu'il est possible d'envoyer, excluant par exemple les graphes contenant des cycles. Nous verrons dans la section 8.4 comment nous comptons résoudre ces problèmes dans GRAS. Enfin, PBIO ne permet que d'échanger des données, sans leur attacher de sémantique tandis que GRAS permet d'envoyer des messages structurés, et d'attacher des fonctions particulières à leur réception. En ce sens, PBIO partage les mêmes objectifs que MPI et semble plus adapté aux applications présentant des schémas de communication réguliers qu'à celles faiblement couplées.

La bibliothèque **AMPiC** (*AppLeS Multi-Protocol Interprocess Communication*, communication entre les processus utilisant plusieurs protocoles par AppLeS – [Hay]) constitue une solution simple pour échanger des messages entre des applications faiblement couplées et pour attacher des fonctions à leur réception. Elle permet d'échanger des structures de données fixes (*i.e.,* sans tableau dynamique ni autres formes de pointeur) à travers les *sockets* directement, en utilisant les bibliothèques MPI ou Globus, ou encore grâce à des connexions SSH.

La principale différence entre GRAS et ces différents outils tient dans l'usage d'un simulateur simplifiant la mise au point des applications. De plus, une solution permettant l'usage d'un simulateur tout en conservant la sémantique de l'une des solutions présentées ici mènerait à une dégradation des performances de simulation. Nous avons en effet vu dans la section 8.1.1 que le développeur doit par exemple ajouter des informations pour permettre de virtualiser les parties optionnelles de son application.

8.3 Exemple : un outil de mesure des performances

Cette section montre la simplicité d'usage de GRAS au travers d'un exemple. Nous présentons un outil volontairement simpliste de mesure des performances de la plate-forme composé de senseurs répartis sur les différents hôtes et d'un client. Les utilisateurs peuvent afficher grâce à ce dernier la charge processeur de toute machine abritant un senseur, ou encore tester la bande passante entre toute paire d'hôtes.

Ce programme utilise un module interne à GRAS pour les tests de bande passante, ce qui lui permet de ne nécessiter que 100 lignes de code pour sa réalisation. Le module de test de bande passante lui-même constitue 200 lignes de code.

Bien que perfectible, le code présenté ici est d'ores et déjà utilisable avec l'implémentation actuelle de GRAS. Nous allons maintenant commenter brièvement le code de ce programme.

8.3.1 Description des données et des messages

Le seul nouveau type de données échangé sur le réseau par cet exemple est le résultat d'une expérimentation concernant la charge processeur d'une machine. Pour indiquer sa structure à GRAS, il suffit de placer sa définition habituelle en C dans un appel de macro `GRAS_TYPE_DEFINE`.

```
1 GRAS_TYPE_DEFINE(s_result,
2  struct s_result {
3    double timestamp;
4    double load;
5  }
6 );
```

La définition de la structure est alors sauvegardée dans une variable de type chaîne pour une utilisation ultérieure. Elle est également reproduite sans modification pour le compilateur afin d'éviter que l'utilisateur n'ait à la dupliquer manuellement.

Seuls deux nouveaux messages sont utilisés par cet exemple : `LOAD_REQUEST` et `LOAD_RESULT`, utilisés respectivement pour s'enquérir de la charge processeur d'une machine distante et pour obtenir les résultats correspondants. Le premier message ne convoie pas de données tandis que le second utilise celles définies à la section précédente.

```
1 #define LOAD_REQUEST 100
2 #define LOAD_RESULT  101
3
4 void my_msg_register() {
5   grasbw_register_messages();
6   gras_msgtype_register(LOAD_REQUEST, NULL);
7   gras_msgtype_register(LOAD_RESULT,  gras_type_get_by_symbol(s_result));
8 }
```

L'appel de fonction à la ligne 5 initialise le module de tests de bande passante en enregistrant ses messages auprès de GRAS. À la ligne 7, la macro `gras_type_get_by_symbol` retrouve la définition de structure sauvée par `GRAS_TYPE_DEFINE`, l'analyse lexicalement et sauve la représentation GRAS résultante.

8.3.2 Coté client

Le client de cet exemple prend ses arguments de la ligne de commande. Le premier argument doit être l'un des mots-clés `load` ou `bandwidth`. La première commande est utilisée pour s'enquérir de la charge processeur d'un hôte distant, qu'il convient de préciser en second argument. L'autre mot-clé est utilisé pour obtenir la bande passante entre deux machines distantes, précisées en second et troisième argument.

```
1  int client (int argc,char *argv[]) {
2    my_msg_register();
3
4    if (!strcmp(argv[1],"bandwidth")) {
5      double duree,bp;
6
7      grasbw_request(argv[2],4000,argv[3],4000,  32000,64000,32000,
8                     &duree,&bp);
9      printf("La bande passante entre %s et %s is %fb/s (le test a duré %f s)\n",
10            argv[2], argv[3], bp, duree);
11
12   } else if (!strcmp(argv[1],"load")) {
13     gras_sock_t     *sock;
14     struct s_result *msg;
15
16     gras_sock_client_open(argv[2],4000,&sock);
17     gras_msg_send_new(sock,LOAD_REQUEST,NULL);
18     gras_msg_wait(3600,LOAD_RESULT,&msg);
19     printf("La charge du serveur %s était %f à l'instant %f\n",
20             argv[2], msg->value, msg->timestamp);
21   } else {
22     printf("Usage : %s [load serveur] | [bandwidth serveur1 serveur2]\n",
23            argv[0]);
24     return 1;
25   }
26   return 0;
27 }
```

Les lignes 5 à 10 utilisent la fonction `grasbw_request()` du module de bande passante pour implémenter la commande `bandwidth` de l'outil construit dans cet exemple et afficher le résultat à l'écran. Les quatre premiers arguments de cette fonction décrivent le nom de machine et le numéro de port de chaque hôte impliqué dans la mesure. Les trois arguments suivants décrivent la quantité de données à échanger lors de l'expérience ainsi que la façon de le faire. En effet, afin de tenir compte des tampons existant au niveau de TCP, les données sont écrites en plusieurs étapes sur la *socket*. Le premier nombre (ici 32000) est le nombre d'octets à écrire sur la *socket* à chaque étape, le second (ici 64000) est le nombre total d'octets à écrire au cours de l'expérience tandis que le troisième (ici 32000) est la taille du tampon de la *socket*. Les deux derniers arguments de la fonction `grasbw_request()` sont utilisés pour obtenir respectivement la durée de l'expérimentation et la bande passante. L'implémentation suit un modèle sur trois niveaux : un message est envoyé au premier hôte pour lui demander la bande passante entre lui-même et le second. Il réalise alors l'expérience et renvoie les valeurs mesurées.

Les lignes 13 à 20 implémentent la commande `load` de notre outil. Après l'ouverture d'une *socket* pointant sur le senseur souhaité à la ligne 16, un message `LOAD_REQUEST` est construit et envoyé à la ligne 17. La ligne 18 attend une réponse de type `LOAD_RESULT` pendant au plus 3600 secondes. Le résultat est ensuite affiché à l'écran.

8.3.3 Coté senseur

Nous présentons maintenant le senseur de cet exemple, qui enregistre une fonction nommée `load_callback()` pour répondre aux messages de type `LOAD_REQUEST`. Cette fonction est implémentée différemment dans la réalité et dans le simulateur grâce aux notions d'exécution conditionnelles présentées dans la section 8.1.1.

```
1  int load_callback(gras_msg_t *msg) {
2    struct s_result *res=malloc(sizeof(struct s_result));
3    res->timestamp = gras_time();
4
5    if (gras_if_sg("cpu load test",0.1)) {
6      res->load = gras_sg_cpuload();
7    } else {
8      /* exécuter le programme uptime et analyser ce qu'il affiche */
9    }
10
11   gras_msg_send_new(msg->sender,LOAD_RESULT,res);
12   return 1;
13 }
14
15 int sensor (int argc,char *argv[]) {
16   gras_sock_t *sock;
17
18   gras_sock_server_open(4000,&(sock));
19   my_msg_register();
20   gras_cb_register(GRASMSG_BW_REQUEST,-1,&load_callback);
21   gras_daemonize(); /* waits for messages for ever */
22 }
```

Au sein du simulateur, la charge courante du serveur est très simplement accessible depuis le simulateur lui-même (comme indiqué à la ligne 6). En revanche, il serait nécessaire de faire appel au programme `uptime` à la ligne 8 pour obtenir cette valeur dans la réalité. Pour choisir entre ces deux implémentations, la fonction `gras_if_sg()` est utilisée à la ligne 5. Lors d'une exécution au sein du simulateur, cette fonction retourne toujours la valeur *vrai* et ordonnance une tâche de la durée fournie sur le processus courant. Sur une vraie plate-forme, cette fonction retourne toujours la valeur *faux*.

La partie principale du senseur formée des lignes 16 à 21 consiste à ouvrir une *socket* en mode serveur, enregistrer les messages et fonctions associés à la réception de chacun d'entre eux. Après cette étape d'initialisation, le senseur passe en mode démon, c'est-à-dire qu'il attend et traite les requêtes reçues indéfiniment.

8.3.4 Déploiement de cet outil

Une fois que tous les composants du système sont écrits, il est nécessaire d'indiquer comment les lancer sur les différentes machines de la plate-forme grâce à un fichier de déploiement, de la forme suivante :

```
1 host1 sensor
2 ...
3 host<N> sensor
4 host1 client bandwidth host1 host2
```

Ce fichier spécifie que le programme nommé `sensor` doit être lancé sans argument sur les machines `host1` à `host<N>` et que le programme `client` doit être lancé avec les arguments fournis sur la machine `host1`.

Il est utilisé par SimGrid pour déployer les programmes sur les processus simulés automatiquement. En revanche, l'implémentation de GRAS pour la vie réelle ne peut pas encore tirer profit de ce fichier pour déployer automatiquement les programmes sur les machines distantes. Cependant, elle l'utilise d'ores et déjà pour obtenir le nom des programmes à construire et écrire automatiquement le `main()` correspondant, appelant les fonctions fournies par l'utilisateur après avoir initialisé la bibliothèque.

8.4 État actuel du prototype et travaux futurs

La figure 8.1 donne une vue d'ensemble de l'environnement GRAS et de l'état actuel de l'implémentation. Le premier objectif est atteint et les fonctionnalités nécessaires à l'exécution du code sans modification au sein du simulateur ou dans la réalité sont présentes.

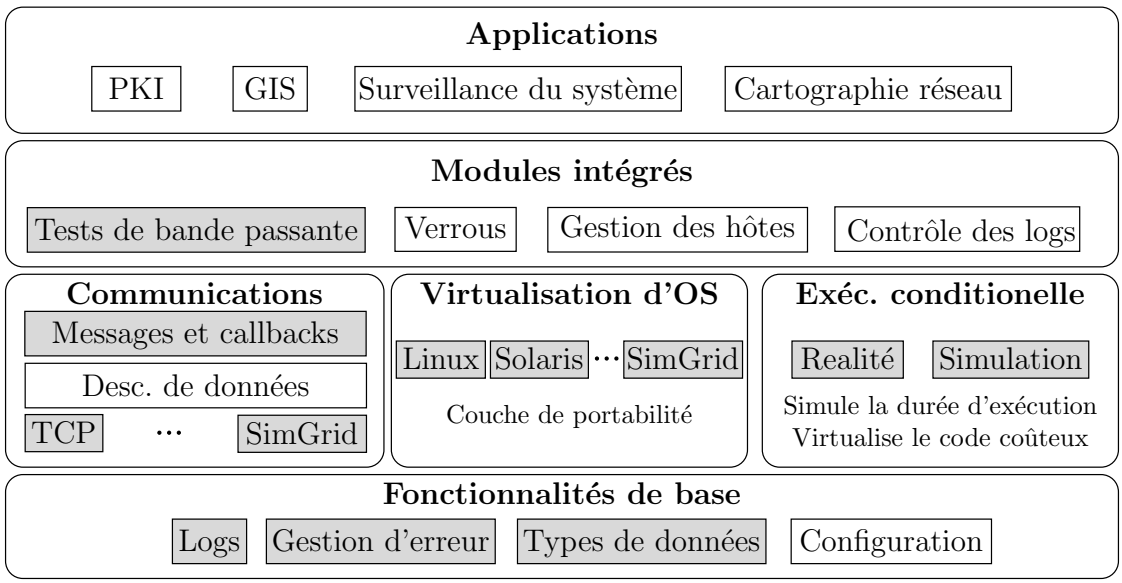


Fig. 8.1 – Vue d'ensemble de GRAS et du prototype actuel.

Cependant, les boîtes non grisées de la figure 8.1 sont les modules nécessitant d'être améliorés ou implémentés. Ainsi, la plupart des modules intégrés tels que les verrous distribués ou la gestion des hôtes (permettant par exemple de relancer des processus à distance) ne sont à l'heure actuelle que planifiés. Les modules les plus avancés (et, par certains cotés, les plus intéressants) tels que le service d'information de la grille (GIS) permettant de rechercher les ressources, ou l'infrastructure d'authentification par clés publiques (PKI) pourraient constituer dans un premier temps des applications de GRAS à part entière avant d'être inclus dans GRAS en tant que modules. De plus la couche de portabilité vis-à-vis des systèmes d'exploitation se limite actuellement à Linux et Solaris, et devrait être adaptée aux systèmes tels que AIX, Irix ou MacOS X.

La bibliothèque de communication de GRAS est sans doute le module laissant le plus de place pour les améliorations futures. Afin de simplifier sa portabilité, elle est découpée en deux niveaux distincts traitant respectivement de la représentation des données et de l'échange effectif de données sur le réseau. Chaque niveau souffre malheureusement de défauts que nous souhaiterions résoudre dans de futures versions.

Expressivité des messages Tout d'abord, la version courante de GRAS ne permet de décrire (et donc placer dans les messages) que des structures statiques. Nous souhaitons bien évidemment lever cette limitation et permettre la description et l'envoi de n'importe quel type de données. En utilisant des techniques classiquement utilisées dans les ramasse-miettes, nous travaillons actuellement à intégrer un module de description de données plus avancé permettant de manipuler toute structure C, y compris celles utilisant des pointeurs pour former des listes chaînées ou même des graphes contenant des cycles.

Performances des communications En plus de ce problème d'expressivité, la couche de description de données souffre également de problèmes de performances. Afin de simplifier les communications impliquant différentes architectures matérielles (et donc des encodages différents pour les données), la version actuelle utilise un format pivot. L'émetteur convertit les données de sa représentation native dans cette représentation, puis le récepteur convertit à son tour la représentation pivot vers son format natif. Cela simplifie l'implémentation puisque chaque architecture ne doit gérer que deux types de conversions : depuis sa représentation native vers le format pivot, et réciproquement. Cependant, les travaux sur PBIO et présentés dans la section 8.2.2 montrent que cette approche implique des pertes de performances relativement importante. Envoyer les données dans le format natif de l'émetteur et ne convertir les données du côté du récepteur qu'en cas de besoin peut ainsi permettre une amélioration des performances de 10 à 50% sur des réseaux locaux. Nous travaillons donc actuellement à l'application de cette méthodologie dans les communications de GRAS.

Enfin, même si cette bibliothèque est conçue pour pouvoir tirer parti de divers pilotes réseaux, seul celui pour TCP a été implémenté jusqu'à présent. Une version complète devrait fournir des pilotes pour les réseaux à haute performance couramment utilisés sur les grappes de machines tels que Myrinet.

Interopérabilité L'interopérabilité constitue également un défi important que doivent relever les infrastructures logicielles modernes telles que celles que nous souhaitons mettre en place

pour la grille. Ce problème est double, et l'aspect le plus souvent étudié est de permettre aux applications de collaborer entre elles, et ce, quelles que soient les bibliothèques de communication utilisées. Mais il est également primordial que plusieurs versions du même programme puisse cohabiter sur le réseau sans que cela nuise à la stabilité ou à l'utilisabilité de l'ensemble.

Concernant l'interopérabilité entre les versions d'un même programme, notre objectif n'est cependant pas de permettre la collaboration entre elles, mais plutôt la cohabitation. Nous souhaitons ainsi que GRAS détecte automatiquement les messages émis par des versions différentes du programme. Notre approche est très proche de celle utilisée par l'éditeur de lien au sein des systèmes d'exploitation modernes utilisant des bibliothèques dynamiques. Il est possible de *versionner* les symboles exportés par une bibliothèque afin de vérifier au lancement des applications si ces symboles sont bien ceux attendus. Dans le cas contraire, le lancement de l'application échoue. De la même manière, les messages au sein de GRAS devraient être *versionnés* pour permettre de vérifier que le message reçu utilise bien la syntaxe et sémantique attendues par la fonction attachée à sa réception. De plus, le tout premier octet de chaque message dénote la version de GRAS utilisée afin de pouvoir détecter (et ignorer) les messages émis par des versions incompatibles de la bibliothèque GRAS elle-même.

À l'heure actuelle, XML constitue le standard de fait pour les questions d'interopérabilité entre les applications tandis que le protocole HTTP (*HyperText Transport Protocol*, protocole de transport d'hypertexte) est largement utilisé pour convoyer différentes sortes de données (par exemple, le protocole pour imprimantes IPP l'utilise pour transporter des fichiers binaires entre les clients, les serveurs et les imprimantes). Différentes solutions de communications entre les applications sont construites au dessus de ces technologies, telles que **XML-RPC**[All03] et **SOAP**[BEK+00].

Cependant, ces solutions sont réputées pour n'offrir que de piètres performances [EBS02]. Pour résoudre ce dilemme, GRAS devrait utiliser un format binaire par défaut pour des raisons de performance, tout en restant capable de recevoir et d'émettre des messages encodés grâce à XML pour des raisons d'interopérabilité. Le tout premier octet de chaque message reçu (dénotant la version de GRAS) peut également être utilisé pour cela. D'après la **RFC2068** [FIG+97] définissant le protocole HTTP, la toute première ligne de chaque envoi doit commencer par les caractères « `HTTP` ». GRAS pourrait donc différencier les messages encodés en XML automatiquement en utilisant le premier octet lu. S'il s'agit de `0x72` (correspondant à la lettre « H »), il s'agit d'un message en XML. Sinon, c'est un message utilisant le format natif de GRAS.

Afin d'offrir la plus grande flexibilité possible sur la syntaxe XML utilisée, GRAS ne devrait cependant pas analyser lui-même de tels messages, mais laisser cette responsabilité aux fonctions enregistrées pour les traiter. Afin de différencier entre les fonctions enregistrées, le contenu du message devrait être enveloppé dans un tag « <gras-message> » dont l'argument dénoterait la fonction visée. Pour les communications sortantes, GRAS devrait offrir une fonction simple pour émettre une chaîne de caractères donnés à un destinataire donné.

La figure 8.2 présente un exemple de communication en XML utilisant GRAS. La figure 8.2(a) donne la chaîne de caractères qu'une entité extérieure devrait écrire sur la *socket* gérée par GRAS pour délivrer la chaîne présentée dans la figure 8.2(b) à la fonction enregistrée sous le nom « `greeting` ».

```
1 HTTP/1.1 200
2 Content-length: 67
3 Content-Type: text/xml
4
5 <?xml version="1.0"?>
6 <gras-message name="greeting">
7 <hello>
8 </gras-message>
```

(a) Chaîne reçue par GRAS.

```
1 <?xml version="1.0"?>
2 <hello>
3
```

(b) Chaîne passée à la fonction « `greeting` ».

FIG. 8.2 – Exemple de communication XML dans GRAS.

8.5 Résumé

Dans ce chapitre, nous avons présenté l'environnement GRAS (*Grid Reality And Simulation*) permettant le développement rapide et confortable d'applications événementielles grâce à l'usage d'un simulateur pour la phase de mise au point. Les applications ainsi développées sont de plus directement utilisables sans modification sur la plate-forme réelle grâce à deux implémentations spécifiques de la même interface.

Ce prototype est d'ores et déjà pleinement utilisable. Nous avons présenté dans la section 8.3 une application très simple tirant parti de cet environnement. Cet exemple permet aux utilisateurs d'obtenir la charge du processeur d'hôtes distants ainsi que la bande passante entre tout couple de machines sur lesquels les serveurs spécifiques sont déployés. Grâce à GRAS, le code de cet exemple représente moins de 100 lignes au total.

Représentant lui-même plus de 10000 lignes de code, GRAS est disponible[2] sous une licence libre, et a été testé sur les systèmes d'exploitation Linux et Solaris.

Nous travaillons actuellement à corriger les limitations de GRAS présentées dans la section 8.4. Nous souhaitons ainsi améliorer les performances, la portabilité et l'expressivité de la bibliothèque de communication utilisée sur la plate-forme réelle. De plus, de futures versions de cet environnement devront permettre aux applications développées en son sein d'interopérer avec des programmes externes en utilisant l'encodage XML lorsque cela s'avère nécessaire.

En parallèle de ces développements, GRAS est d'ores et déjà utilisé comme base de projets de recherche, tels que l'outil de découverte de topologie que nous allons présenter dans le chapitre suivant.

[2] http://graal.ens-lyon.fr/~mquinson/gras.html

Chapitre 9

Découverte de la topologie de la grille

Nous avons présenté dans le chapitre 7 un algorithme permettant d'automatiser le déploiement de NWS sur la grille à partir de la cartographie de la plate-forme. Mais, nous avons également constaté dans ce chapitre qu'ENV ne permet pas d'obtenir la topologie complète de la plate-forme, mais seulement une vue arborescente enracinée en un point arbitraire du réseau. Il est donc impossible d'obtenir par ce biais des informations sur les liens connectant des machines (autres que le maître choisi) entre elles.

Malheureusement, les autres solutions de cartographie du réseau présentées dans le chapitre 6 ne sont pas adaptées à un contexte de *metacomputing*. Ces constatations nous ont poussé à constituer un nouveau projet en collaboration avec Arnaud Legrand (doctorant au LIP) pour satisfaire nos besoins. Ce chapitre présente donc le projet ALNeM (*Application-Level Network Mapper* – outil de découverte de la topologie de niveau applicatif) visant à permettre le déploiement de NWS et plus généralement des constituants de la grille. Bien entendu, nous utiliserons pour cela l'environnement de développement GRAS, présenté dans le chapitre 8 et développé en partie dans ce but.

Ce chapitre suit l'organisation suivante. Nous préciserons dans la section 9.1 nos objectifs pour ALNeM et introduirons certaines notations préliminaires. Grâce aux outils mathématiques présentés dans la section 9.2, la section 9.3 présentera (et démontrera partiellement) un algorithme permettant de reconstruire la vision souhaitée de la topologie sous certaines conditions. La section 9.4 évoquera comment les informations nécessaires à cette reconstruction peuvent être collectées. Nous présenterons les résultats obtenus grâce ALNeM lors d'une simulation dans la section 9.5 avant de conclure ce chapitre. Les preuves des lemmes et théorèmes utilisés dans ce chapitres sont placées dans l'annexe A.

9.1 Présentation d'ALNeM et de ses objectifs

Avant de présenter le fonctionnement d'ALNeM, il est important de détailler ses objectifs. Tout d'abord, les résultats doivent être fournis sous la forme d'un graphe capturant toutes les informations pertinentes puisque la plupart des algorithmes distribués représentent le réseau de cette manière ([Tel00]).

Les informations que nous souhaitons capturer sont plus d'ordre qualitatif que quantitatif. Ainsi, le plus important n'est pas que notre outil soit capable de déterminer avec précision la

bande passante de bout en bout entre les machines, puisque NWS offre déjà ce service. Par ailleurs, nous ne cherchons pas à reconstruire le schéma d'interconnexion physique du réseau, mais plutôt à extraire une vision pratique pour les applications.

En particulier, les machines les plus importantes dans notre contexte sont celles sur lesquelles des constituants de la grille peuvent être exécutés sans privilège particulier. Cela exclut les machines du cœur du réseau telles que les routeurs et pare-feux, qui peuvent être omis dans notre contexte.

Définition 9.1. *L'ensemble des* ***nœuds*** *de la plate-forme (c'est-à-dire des hôtes sur lesquels des processus utilisateurs peuvent être exécutés) est noté* $\boldsymbol{\mathcal{H}}$.

De fait, l'information principale qu'ALNeM doit capturer est l'éventuelle interférence entre les flux de données ayant lieu en même temps. C'est-à-dire que nous souhaitons être capable de prédire à priori si la bande passante entre un couple de machines est affectée par un transfert entre deux autres machines ou si les deux flux peuvent avoir lieu en concurrence sans interférence mutuelle. Cette information serait en effet suffisante pour déployer NWS en choisissant quelles machines doivent être placées dans une clique commune.

9.1.1 Modélisation

La façon la plus naturelle de représenter cette notion d'interférence entre les flux (AB) et (CD) dans un graphe est de faire en sorte que le plus court chemin (en nombre de sauts) entre A et B ait une intersection non nulle avec celui entre C et D. Il est naturellement nécessaire pour ce faire d'ajouter des sommets séparateurs en plus de ceux représentant les machines de la plate-forme.

Nous cherchons donc un graphe $G = (V, E)$ non-orienté composé de deux types de sommets : les *nœuds*, représentant les machines constituant la grille et des *séparateurs* représentant les points de contention du réseau. Cette modélisation permet de ne se concentrer que sur les sommets dans l'algorithme de création de l'objet (ainsi que ceux qui l'utiliseront ensuite). Étant donné que les séparateurs représentent à la fois les délais dus au réseau et ceux dus aux routeurs, toutes les arêtes ont la même signification. Le routage utilisé sur ce graphe est celui du plus court chemin.

Définition 9.2. *Soit* $u, v \in V$ *: la notation* $\left(u \underset{G}{\longrightarrow} v\right)$ *représente l'ensemble ordonné des sommets appartenant à la route dans* G *allant de* u *à* v *(*u *et* v *sont respectivement le premier et le dernier élément de cet ensemble). Cet ensemble dépend du routage utilisé dans le graphe choisi.*

Définition 9.3. *Soit* $a, b, c, d \in \mathcal{H}$ *: Le fait que le chemin* (ab) *interfère avec* (cd) *dans le graphe* G *est noté* $(ab) \between_{\boldsymbol{G}} (cd)$*. Ceci est défini par l'équivalence suivante :*

$$(ab) \between_G (cd) \iff \left(a \underset{G}{\longrightarrow} b\right) \cap \left(c \underset{G}{\longrightarrow} d\right) \neq \emptyset$$

La non-interférence dans le graphe G *entre* (ab) *et* (cd) *est notée* $(ab) /\!\!/_{\boldsymbol{G}} (cd)$*, définie par :*

$$(ab) \between_G (cd) \iff \neg \Big((ab) /\!\!/_G (cd)\Big)$$

Cela définit ce que nous nommons la notion d'interférence théorique (par opposition à l'interférence mesurée que nous introduirons dans la section suivante). Elle est également notée $\curlyvee_{th}$ lorsque le graphe dont il est question est donné par le contexte. Ces définitions nous donnent trivialement le lemme suivant :

Lemme 9.1 ($\curlyvee_{th}$ est une relation symétrique). *C'est-à-dire :*

$$\forall a, b, c, d \in \mathcal{H}, \quad (ab) \curlyvee_{th} (cd) \Leftrightarrow (cd) \curlyvee_{th} (ab).$$

Démonstration. L'intersection d'ensemble est également une relation symétrique. ■

Remarque 9.1. *En revanche, l'équivalence $(ab) \curlyvee_{th} (cd) \Leftrightarrow (ba) \curlyvee_{th} (cd)$ n'est pas toujours vraie puisqu'elle supposerait que le routage est symétrique, ce qui n'est pas garanti dans un graphe orienté tel que G.*

9.1.2 Méthodologie de mesure

Après avoir spécifié l'objet que nous souhaitons construire, nous allons maintenant définir plus précisément les informations disponibles, et comment les obtenir.

Définition 9.4. *Soit $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ le graphe représentant la topologie réelle, avec $\widetilde{V}$ l'ensemble de toutes les machines existantes (y compris celles du cœur du réseau telles que les routeurs) et $\widetilde{E}$ les liens existant entre eux.*

Le routage utilisé dans $\widetilde{G}$ dépend de la configuration de chaque élément de $\widetilde{V}$ et n'est donc défini que localement. Même si le graphe $\widetilde{G}$ reflète une réalité physique, il est rare qu'il soit connu dans les faits. Ainsi, lors de l'expérience de cartographie automatique d'une partie du réseau de notre laboratoire que nous avons relatée dans le chapitre 7, nous avons découvert une inconsistance de configuration entraînant une asymétrie du routage entre les machines *the-doors* et *popc*. En effet, la bande passante maximale dans un sens était limitée à 10 Mb/s tandis que celle dans l'autre sens pouvait monter à 100 Mb/s. Ceci était dû à une erreur de configuration de l'un des routeurs, que même les administrateurs du système ignoraient. D'après [Pax97], cette situation est assez courante sur Internet, et même relativement probable dans le cas des réseaux à grande distance.

De plus, notre objectif de *metacomputing* nous interdit toute mesure nécessitant des privilèges particuliers sur le réseau ou sur les hôtes. Dans ces conditions, le seul moyen de savoir si un lien influe sur un autre semble être de réaliser une mesure directe en comparant la bande passante obtenue habituellement sur un lien à celle obtenue lorsqu'un autre lien est saturé.

Définition 9.5. *Étant donné quatre nœuds a, b, c et d, la bande passante entre a et b en l'absence de trafic entre c et d est notée $\boldsymbol{bw(ab)}$. La bande passante entre a et b lorsque le lien entre c et d est saturé est quant à elle notée $\boldsymbol{bw_{/\!/cd}(ab)}$.*

Cela nous permet de définir une notion d'interférence mesurée en comparant ces deux valeurs. Si leur ratio est égal à 0.5, cela signifie que les deux flux partagent équitablement une ressource réseau. S'il est égal à 1, (cd) n'a aucune influence sur (ab). Pour tenir compte des erreurs de mesures et des perturbations dues à la charge externe, nous sommes amené à utiliser des seuils.

Définition 9.6. *Nous considérons que* (cd) *influe sur* (ab) *si et seulement si*

$$\frac{bw_{/\!/\,cd}(ab)}{bw(ab)} < 0.7$$

Dans ce cas, nous utilisons la notation $(ab) \curlywedge_{mes} (cd)$ *(la notation mes signifiant mesuré).*

Définition 9.7. *Si ce ratio est supérieur à 0.9, nous considérons que les transferts n'interfèrent pas et utilisons la notation* $(ab) /\!/_{mes} (cd)$.

Un ratio entre 0.7 et 0.9 dénote d'une erreur de mesure, nécessitant de réaliser l'expérience une nouvelle fois. La valeur de ces différents seuils est celle déterminée expérimentalement par les auteurs du projet ENV (présenté dans le chapitre 7).

Lemme 9.2 ($\curlywedge_{mes}$ n'est pas une relation symétrique). *C'est-à-dire :*

$$\exists a, b, c, d \in \mathcal{H} \ / \ \Big((ab) \curlywedge_{mes} (cd)\Big) \wedge \neg\Big((cd) \curlywedge_{mes} (ab)\Big)$$

Démonstration. La figure 9.1 donne un contre-exemple à la symétrie de la relation $\curlywedge_{mes}$.

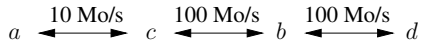


FIG. 9.1 – Contre-exemple à la symétrie de la relation $\curlywedge_{mes}$.

Dans ce cas, il est probable que les mesures donnent à la fois $\frac{bw_{/\!/\,cd}(ab)}{bw(ab)} < 0.7$ et $\frac{bw_{/\!/\,ab}(cd)}{bw(cd)} > 0.9$ car le lien (ab) est limité à 10 Mb/s sur le segment (ac), ce qui réduit suffisamment son influence sur les transferts ayant lieu sur (cd) (qui approchent les 100 Mb/s) pour le rendre indétectable. ■

On peut remarquer que dans ce contre-exemple, les chemins (ab) et (cd) partagent bien un lien de $\widetilde{G}$. De plus, l'influence de (ab) sur (cd) existe, même si elle est trop faible pour être mesurée. Cela nous mène à symétriser la notion d'interférence dans la réalité pour simplifier les algorithmes et démonstrations ainsi que pour nous rapprocher de la notion de $\curlywedge_{th}$.

Définition 9.8. *Nous considérons que* (ab) *et* (cd) *interfèrent dans la réalité si et seulement si* (ab) *influe sur* (cd) *ou si* (cd) *influe sur* (ab). *On note alors* $(ab) \curlywedge_{rl} (cd)$ *(*rl *signifiant réel). On a :*

$$(ab) \curlywedge_{rl} (cd) \iff \begin{cases} (ab) \curlywedge_{mes} (cd) \\ (cd) \curlywedge_{mes} (ab) \end{cases} (ou) \iff \begin{cases} \frac{bw_{/\!/\,cd}(ab)}{bw(ab)} < 0.7 \\ \frac{bw_{/\!/\,ab}(cd)}{bw(ab)} < 0.7 \end{cases} (ou)$$

Dans le cas contraire, on considère que les deux chemins n'interfèrent pas l'un sur l'autre, et on note $(ab) /\!/_{rl} (cd)$.

Par construction, la relation $\curlywedge_{rl}$ est symétrique, et $(ab) \curlywedge_{rl} (cd) \iff \neg\, (ab) /\!/_{rl} (cd)$. Il est possible de stocker la valeur de cette relation pour les différents nœuds dans une matrice à quatre dimensions définie comme suit :

Définition 9.9. *Soit $I(\mathcal{H}, \curlywedge_{rl})$ la matrice d'interférence entre tous les éléments de l'ensemble $\mathcal{H}$ telle qu'induite par la relation $\curlywedge_{rl}$:*

$$I(\mathcal{H}, \curlywedge_{rl})(a,b,c,d) = \begin{cases} 1 & si\ (ab) \curlywedge_{rl} (cd) \\ 0 & sinon \end{cases}$$

9.1.3 Formalisation du problème

Avec ces notations, il nous est possible de définir plus formellement le problème auquel ALNeM tente d'apporter des éléments de réponse :

Définition 9.10. InterferenceGraph *: Étant donné $\mathcal{H}$ et $I(\mathcal{H}, \curlywedge_{\widetilde{G}})$, trouver un graphe $G = (V, E)$ vérifiant :*

$$\begin{cases} \mathcal{H} \subset V\ ; \\ I(\mathcal{H}, \curlywedge_{\widetilde{G}}) = I(\mathcal{H}, \curlywedge_{G})\ ; \\ |V|\ \textit{est minimal.} \end{cases}$$

Puisqu'il est impossible de retrouver le graphe réel $\widetilde{G}$ directement dans un contexte de *metacomputing*, l'idée est donc de chercher un graphe théorique G mimant les effets de $\widetilde{G}$ vis-à-vis des interférences entre les flux de façon à permettre aux utilisateurs d'utiliser G pour obtenir des informations sur $\widetilde{G}$.

Par ailleurs, même s'il était possible d'obtenir $\widetilde{G}$, cet objet serait difficile à manipuler ne serait-ce que de par sa taille. Pour en simplifier l'usage, nous cherchons donc de plus à ce que $|V|$ soit petit dans G. Mais contrairement à ce qui est exprimé dans la formalisation de InterferenceGraph, nous ne chercherons pas un G *optimal* (*i.e.,* minimisant $|V|$), mais plutôt un G *pratique* à l'usage.

Le problème de décision associé à InterferenceGraph est clairement dans NP, mais nous ignorons encore à l'heure actuelle s'il est NP-Complet.

9.2 Outils mathématiques

Cette section présente des outils mathématiques permettant d'étudier théoriquement le problème InterferenceGraph et que nous utiliserons dans la section suivante pour mettre au point des algorithmes résolvant ce problème dans différents cas.

9.2.1 Hypothèses

Nous supposons dans la suite que le graphe $\widetilde{G}$ respecte les hypothèses suivantes :

Hypothèse 1 (consistance du routage). $\forall (a,b,c) \in \widetilde{V}$:

$$c \in \left(a \underset{\widetilde{G}}{\longrightarrow} b\right) \Longrightarrow \left(a \underset{\widetilde{G}}{\longrightarrow} b\right) \cap \left(a \underset{\widetilde{G}}{\longrightarrow} c\right) = \left(a \underset{\widetilde{G}}{\longrightarrow} c\right)$$

Cette hypothèse spécifie que le routage utilisé dans $\widetilde{G}$ ne présente pas d'inconsistance étrange. Si un sommet c se trouve sur le chemin reliant a à b, les paquets transitant de a à b empruntent le même trajet sur le début de leur parcours que ceux transitant de a à c. Le contraire signifierait par exemple que la machine a utilise une machine ρ comme passerelle, et que ρ route les paquets pour b par c tandis que la machine a peut contacter c directement sans passer par ρ.

Hypothèse 2 (symétrie du routage). $\forall (a,b) \in \widetilde{V} : \left(a \underset{\widetilde{G}}{\longrightarrow} b\right) = \left(b \underset{\widetilde{G}}{\longrightarrow} a\right)$

Cette hypothèse spécifie que le routage est symétrique.

Nous savons que ces hypothèses ne sont pas toujours vérifiées sur Internet puisque toutes les inconsistances de routage imaginables existent dans la réalité ([Pax97]). Cependant, même si la version actuelle de notre algorithme ne peut traiter les cas où ces hypothèses ne sont pas respectées, elle permet de les détecter. Nous espérons que cela nous permettra d'offrir un traitement spécial de ces cas dans de futures versions.

9.2.2 Interférence totales et séparateurs

Par souci de clarté, l'ensemble $\left(v \underset{\widetilde{G}}{\longrightarrow} w\right)$ est noté ici $(v \rightarrow w)$.

Définition 9.11. *Deux nœuds a et b sont dits en **interférence totale** si et seulement si tout flux sortant de a interfère avec tout flux sortant de b. On note alors $\boldsymbol{a \perp b}$.*

Plus formellement, nous avons :

$$a \perp b \Longleftrightarrow \forall (u,v) \in \mathcal{H},\ (au) \curlywedge_{rl} (bv)$$

Autrement dit, a et b sont en interférence totale si et seulement s'ils interfèrent avec tous les autres nœuds existants.

Lemme 9.3 (Séparateur).

$$\forall a,b \in \mathcal{H},\ a \perp b \Longleftrightarrow \exists \rho \in \widetilde{V} \ \Big/\ \forall z \in \mathcal{H} : \rho \in (a \rightarrow z) \cap (b \rightarrow z)\,.$$

C'est-à-dire que le fait que a et b soient en interférence totale est équivalent à l'existence d'un sommet ρ commun à tous les chemins reliant a aux nœuds extérieurs et ceux sortant de b. La preuve de ce lemme, détaillée en annexes dans la section A.1, utilise l'hypothèse 1 (consistance du routage), mais n'utilise cependant pas l'hypothèse 2 (routage symétrique).

Définition 9.12. *Pour a et $b \in \mathcal{H}$, un sommet ρ tel que celui donné dans le lemme de séparation (9.3) est dit **séparateur** de a et b.*

Théorème 9.1. *L'interférence totale ($\perp$) est une relation d'équivalence. De plus, pour toute classe d'équivalence de $\perp$, il existe un séparateur commun à tous les couples de la classe.*

La preuve de ce théorème, détaillée en annexes dans la section A.2, utilise le lemme 9.3 (séparation) et l'hypothèse 2 (symétrie du routage).

Théorème 9.2 (Représentativité). *Soit $\mathcal{C}$ une classe d'équivalence pour $\perp$ et ρ un séparateur de ses éléments.*

$$\forall a \in \mathcal{C},\ \forall b, u, v \in \mathcal{H},\ \ (a,u) \curlywedge_{rl} (b,v) \Leftrightarrow (\rho,u) \curlywedge_{rl} (b,v)$$

C'est-à-dire que le séparateur d'une classe d'équivalence a les mêmes interférences avec les autres nœuds que chaque élément de la classe. Autrement dit, le séparateur constitue un représentant valide pour tous les éléments de la classe d'équivalence vis-à-vis des interactions avec les autres nœuds.

La preuve du théorème 9.2 (présentée en annexes dans la section A.3) ne repose pas explicitement sur l'hypothèse 2 (symétrie du routage). Elle repose en revanche sur l'existence d'un séparateur commun à tous les couples d'une classe d'équivalence de $\perp$ donnée. Ceci nous est donné par le théorème 9.1, lui-même démontré grâce à l'hypothèse 2, mais il est peut-être possible d'obtenir ce fait d'une autre manière n'imposant pas la symétrie du routage.

L'existence d'un séparateur commun représentatif à chaque classe d'équivalence de $\perp$ est le point de départ de l'algorithme présenté dans la section suivante, qui consiste à rechercher chaque classe d'équivalence selon $\perp$, puis à remplacer tous ses membres par un représentant unique : le séparateur de la classe.

9.3 Algorithme de reconstruction

Nous allons maintenant élaborer un algorithme construisant un graphe G induisant la même matrice d'interférence I que celle de $\widetilde{G}$. Pour cela, nous allons procéder en plusieures étapes. Nous donnerons tout d'abord une version dont nous démontrerons qu'elle trouve une solution s'il est possible de construire G sous forme d'un arbre. En étudiant les cas où cet algorithme échoue, nous l'étendrons par la suite pour traiter certains cas nécessitant l'introduction de cycles dans le graphe.

9.3.1 Traitement des arbres

Le principe général est de construire le graphe en commençant par les extrémités, de remplacer les sous-arbres par leur séparateur dans la matrice d'interférence, et d'itérer le processus.

ARBRES$(\mathcal{H}, I(\mathcal{H}, \curlywedge_{rl}))$

1. *Initialisation*
   $i \leftarrow 0$; $\mathcal{C}_i \leftarrow \mathcal{H}$; $E_i \leftarrow \emptyset$; $V_i \leftarrow \emptyset$

2. *Recherche des classes d'équivalences et élection du séparateur*
   Soit $h_1, \ldots, h_p$ les classes d'équivalence de $\perp$ sur $\mathcal{C}_i$. Elles peuvent être aisément calculées par un algorithme glouton à partir de $I(\mathcal{C}_i, \curlywedge_{rl})$ en cherchant les ensembles maximaux pour l'inclusion de nœuds en interférence totale.
   Créer un séparateur l_i pour chaque classe d'équivalence h_i.
   $\mathcal{C}_{i+1} \leftarrow \{l_1, \ldots, l_p\}$
3. *Mise à jour du graphe*
   $V_{i+1} \leftarrow V_i$; $E_{i+1} \leftarrow E_i$
   Pour chaque $h_j \in \mathcal{C}_i$, pour chaque $v \in h_j$, $\begin{cases} E_{i+1} \leftarrow E_{i+1} \cup \{(v, l_j)\} \\ V_{i+1} \leftarrow V_{i+1} \cup \{v\} \end{cases}$
4. *Mise à jour de la matrice d'interférence*
   Soit $l_\alpha, l_\beta, l_\gamma, l_\delta \in \mathcal{C}_{i+1}$ représentant respectivement $h_\alpha, h_\beta, h_\gamma, h_\delta$.
   Soit $m_\alpha, m_\beta, m_\gamma, m_\delta \in \mathcal{C}_i$ tels que $m_\alpha \in h_\alpha$, $m_\beta \in h_\beta$, $m_\gamma \in h_\gamma$ et $m_\delta \in h_\delta$.
   $I(\mathcal{C}_{i+1}, \curlywedge)\big(l_\alpha, l_\beta, l_\gamma, l_\delta\big) = I\left(\mathcal{C}_i, \curlywedge\right)\left(m_\alpha, m_\beta, m_\gamma, m_\delta\right)$
5. Répéter les étapes 2 à 4 jusqu'à ce que $\mathcal{C}_i = \mathcal{C}_{i+1}$.

Théorème 9.3 (correction de l'algorithme Arbres). *Lorsque l'algorithme termine avec un seul leader (*i.e. $\exists n \,/\, \left|\mathcal{C}_n\right| = 1$*), le routage par plus court chemin sur le graphe G résultant satisfait $I(\mathcal{H}, \curlywedge_G) = I(\mathcal{H}, \curlywedge_{rl})$.*

Théorème 9.4 (condition de terminaison de l'algorithme Arbres). *Pour une instance de* InterferenceGraph *donnée, s'il est possible de construire une solution G étant un arbre, alors* Arbres *termine avec un seul leader.*

Ces deux théorèmes sont démontrés en annexes respectivement dans les sections A.4 et A.5 en utilisant le théorème 9.2 (représentativité du séparateur) et les deux hypothèses (routage consistant et symétrique).

Remarque 9.2. *Lorsque l'algorithme* Arbres *permet d'exhiber une solution, alors celle-ci est optimale.*

En effet, le théorème 9.2 (représentativité du séparateur) permet d'utiliser l'un des éléments de la classe d'équivalence en séparateur. De cette façon, aucun nouveau point n'est introduit, et l'ensemble des points de G étant réduit à $\mathcal{H}$, G est clairement optimal.

9.3.2 Traitement des cliques

Lorsqu'à une étape i de l'algorithme Arbres, il n'existe plus aucune interférence entre les éléments de C_n, il est naturellement impossible de trouver deux machines données en interférence totale avec les autres. Si cette condition survient, l'algorithme ne parvient donc pas à reconstituer un graphe connexe convenable.

La solution intuitive dans ce cas, consistant à connecter tous les représentants de chaque composante connexe sous forme d'une clique complète, permet alors de trouver une solution convenable. Ceci est exprimé plus formellement par le théorème 9.5 suivant.

Théorème 9.5. *(Validité du traitement des cliques) S'il existe une étape i de* Arbres *telle que $\forall a_i, u_i, b_i, v_i \in \mathcal{C}_i, (a_i, u_i) /\!\!/ (b_i, v_i)$, alors le graphe G connectant deux à deux les éléments de $\mathcal{C}_i$ satisfait $I(\mathcal{H}, \curlywedge_G) = I(\mathcal{H}, \curlywedge_{rl})$ lorsque le routage par plus court chemin est utilisé.*

Démonstration. Cette démonstration est triviale puisque nous savons que la relation $I(a,u,b,v) = 1 \Leftrightarrow \left(a \underset{G}{\longrightarrow} u\right) \cap \left(b \underset{G}{\longrightarrow} v\right) \neq \emptyset$ est respectée dans chaque partie connexe grâce au théorème 9.3 (correction de l'algorithme Arbres). Puisque nous savons que ces composantes connexes n'interfèrent pas, les connecter en clique complète nous permet de nous assurer que le chemin de l'une à l'autre ne passera pas par une troisième. ■

Avec cette extension, il devient possible de traiter les constellations d'arbres avec notre algorithme, c'est-à-dire les graphes formés de plusieurs arbres distincts et dont les racines sont interconnectées par une clique complète.

Remarque 9.3. *Les solutions exhibées par ce traitement des cliques, lorsqu'elles existent, sont également optimales.*

En effet, ce traitement ne demande pas d'introduire de nouveaux points dans G, qui reste donc clairement optimal.

9.3.3 Traitement des cycles

Par application du théorème 9.4 (conditions de terminaison de l'algorithme), si l'algorithme ne termine pas avec un graphe connexe, c'est qu'il n'existe pas d'arbre permettant de reproduire les interférences dénotées par la matrice I. En particulier, il est nécessaire d'introduire un cycle dans le graphe G.

Nous allons maintenant étendre l'algorithme pour lui permettre de prendre en compte les matrices d'interférences produites par certains graphes $\widetilde{G}$ contenant des cycles. Cette étude préliminaire associée aux plus faibles propriétés des graphes contenant des cycles par rapport aux arbres nous ont conduit à choisir pour l'instant une approche plus pragmatique.

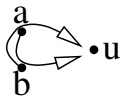
Nous supposons que l'algorithme Arbres a été utilisé pour placer dans la même composante connexe les éléments en interférence totale, mais que la connectivité du graphe reconstruit n'est que partielle. Nous nous trouvons donc à une étape i de l'algorithme, et il n'existe pas de $l_\alpha, l_\beta \in C_i$ tels que $l_\alpha \perp l_\beta$.

Le principe général de notre algorithme traitant ces cas est de trouver deux points proches entre eux sur le cycle, de couper le cycle entre ces deux points afin que l'algorithme Arbres puisse continuer, puis de réintroduire le cycle entre ces deux points après coup.

Cette approche induit deux difficultés différentes. Il s'agit tout d'abord de détecter deux points proches entre eux sur la base des informations contenues dans la matrice d'interférence. Le second problème est de savoir comment réintroduire le cycle dans le graphe construit après coup.

Dans notre algorithme, le cycle sera cassé entre les deux points ayant le plus d'interférences dans la matrice, c'est-à-dire entre les deux points a et b maximisant la taille de l'ensemble $\{u, v : au \curlywedge bv\}$. Il est possible de couper l'ensemble des points en trois sous-ensembles I_1, I_2 et I_3 définis comme suit :

$$\begin{cases} I_1 = \{u \in \mathcal{C}_i : a \in (b \rightarrow u) \text{ et } b \notin (a \rightarrow u)\} \\ I_2 = \{u \in \mathcal{C}_i : a \notin (b \rightarrow u) \text{ et } b \in (a \rightarrow u)\} \\ I_3 = \{u \in \mathcal{C}_i : a \notin (b \rightarrow u) \text{ et } b \notin (a \rightarrow u)\} \\ I_4 = \{u \in \mathcal{C}_i : a \in (b \rightarrow u) \text{ et } b \in (a \rightarrow u)\} \end{cases}$$

L'hypothèse 1 implique trivialement que $I_4 = \{a, b\}$ car s'il existait un élément u différent de a et b dans I_4, nous aurions la situation absurde suivante : 

Lemme 9.4. *(proximité de a et b) Si a et b est le couple ayant le plus de d'interférence dans la matrice $I(C_i, \curlywedge)$, alors aucun graphe plaçant un élément de C_i entre a et b ne peut satisfaire les contraintes imposées par la matrice d'interférence.*

Démonstration. L'existence d'un tel point δ pour tout u, v, impliquerait que $(au \curlywedge bv) \iff ((\delta u \curlywedge bv)$ ou $(au \curlywedge \delta v))$ puisque le chemin sortant de δ passe forcément par b (ou par a). Donc les couples $(a\delta)$ et $(b\delta)$ auraient au moins autant d'interférences que (ab).

Dans le même temps, nous aurions $a\delta \mathbin{/\!/} bu$ pour tout u dans I_3, et $au \mathbin{/\!/} b\delta$ pour tout u dans I_1. Il en découlerait que le couple (ab) aurait moins d'interférences que l'un des couples $(a\delta)$ ou $(b\delta)$ dans le graphe, ce qui est absurde. ■

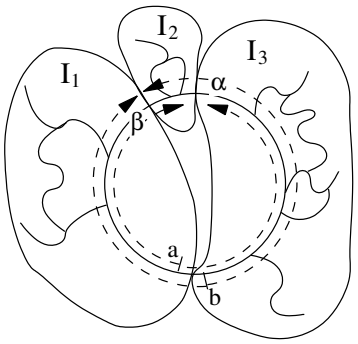


FIG. 9.2 – Découpage de l'ensemble des points pour traiter des cycles.

Nous allons donc couper le cycle entre les points a et b ainsi décrits. En utilisant l'hypothèse 1 (consistance de routage), il est possible de représenter graphiquement le découpage de C_i en I_1, I_2 et I_3 par la figure 9.2. Nous introduisons le point α séparant les ensembles $\{u \in \mathsf{C}_i : b \in (a \rightarrow u)\}$ et $\{u \in \mathsf{C}_i : b \notin (a \rightarrow u)\}$. De manière similaire, nous construisons le point β par rapport à b.

Il nous faut maintenant déterminer les trois ensembles I_1, I_2 et I_3 par analyse de la tranche concernant a et b dans la matrice d'interférence. Nous allons pour cela chercher un réordonnancement de ses éléments plaçant sur des colonnes adjacentes les éléments u tels que $(au \curlywedge bv)$ pour tout v et sur des lignes adjacentes les éléments v tels que $(au \curlywedge bv)$ pour tout u. La figure 9.3 représente graphiquement la forme souhaitée. Il suffit pour cela de trier topologiquement les composantes fortement connexes du graphe dont cette matrice est la matrice d'adjacence. S'il est possible de trouver un tel réordonnancement des éléments,

nous obtenons ainsi la partition des éléments de C_i en I_1, I_2 et I_3 à partir de la matrice d'interférence.

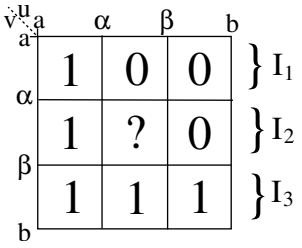


FIG. 9.3 – Matrice d'interférence sur la tranche (a, b) après réordonnancement.

Le dernier problème à résoudre est la connexion des graphes obtenus par la récursion de l'algorithme sur chacun de ces ensembles. Puisque nous avons cassé le cycle en deux points entre a et b d'une part et entre α et β d'autre part. il nous faut maintenant reconnecter ces deux points. Pour le premier, le lemme 9.4 implique qu'il suffit de connecter a et b ensemble. Pour reconnecter le cycle entre α et β, il nous faut tout d'abord trouver quels éléments de C_i étaient les plus proches du point de coupure pour pouvoir les reconnecter dans le graphe final.

Nous savons que la première étape de la récursion de l'algorithme sur l'ensemble I_1 trouvera des éléments en interférence totale puisque nous avons cassé le cycle en deux points, créant ainsi des filaments proches de a et α. Puisque l'algorithme ARBRES ne pouvait plus grouper d'éléments en interférence totale à l'étape précédente, ces filaments sont de plus les deux seuls possibles. Il est également possible de différencier simplement ces deux composantes connexes puisque l'une d'entre elles contient le point a.

Par symétrie sur I_3 et b, nous déduisons donc que la seconde jonction doit se faire entre l'un des points de la composante connexe crée lors de la première itération d'ARBRES sur I_1 ne contenant pas a d'une part, et l'un des points de celle créée dans les mêmes conditions sur I_3 et ne contenant pas b d'autre part. Le choix entre les différents candidats de I_1 et ceux de I_3 se fait en sélectionnant les deux points S_α et S_β de chaque ensemble ayant le plus d'interférences avec les autres éléments de C_i (en utilisant l'intuition que deux points en forte interférence doivent être proches dans le graphe).

Après avoir ainsi identifié les points S_α et S_β à la frontière des ensembles où la jonction doit se faire, il est possible de réaliser cette fusion en exécutant l'algorithme récursivement sur l'ensemble de nœuds $I_2 \cup \{S_\alpha, S_\beta\}$.

Nous obtenons donc ainsi un algorithme permettant de traiter certains cycles. Il ne s'agit cependant pas d'une généralisation complète de l'algorithme, car il est impossible de démontrer que le graphe G ainsi construit respecte parfaitement toutes les interférences de I. Ceci est en partie dû à la méthode de détection des cycles, basée sur des informations locales aux deux points sélectionnés sans tenir compte d'informations présentes dans d'autres parties de la matrice d'interférences et possiblement contradictoires.

9.4 Collecte des informations nécessaires

Nous allons maintenant aborder la façon dont ALNeM collecte les informations d'interférence sur la plate-forme. Comme nous l'avons vu dans le chapitre 7, il s'agit d'une tâche techniquement difficile et potentiellement très coûteuse en temps qu'il convient d'optimiser.

9.4.1 Algorithme intuitif

La façon la plus simple de collecter ces informations se déroule en $|\mathcal{H}|^4$ étapes, chacune consistant pour un quadruplet $(a, b, c, d) \in \mathcal{H}^4$ en :

1. Mesurer la bande passante de (ab) (notée $bw(ab)$) ;
2. Mesurer la bande passante de (ab) lorsque le lien (cd) est saturé (notée $bw_{/\!/cd}(ab)$) ;
3. Calculer le ratio.

Les étapes (1) et (2) doivent être suffisamment longues pour permettre au réseau de se stabiliser. Nous devons attendre que l'expérience de saturation précédente se termine réellement pour éviter toute perturbation tandis qu'il est nécessaire d'attendre que le lien (cd) soit saturé avant l'étape (2). Ces délais interdisent de tester raisonnablement plus de deux quadruplets par minute sur un réseau potentiellement à grande distance. L'exécution de cet algorithme nécessite donc environ $\frac{|\mathcal{H}|^4}{2}$ minutes, ce qui représente environ 50 jours quand $|\mathcal{H}| = 20$. Le manque d'extensibilité condamne cette solution.

9.4.2 Optimisations

Afin d'accélérer le processus, ALNeM mène les expériences en parallèle en tirant parti de l'existence de liens indépendants. Puisqu'ils n'interfèrent pas les uns sur les autres, il est possible de saturer plusieurs de ces liens en parallèle. De cette façon, lorsque nous testons un nouveau lien, il n'est pas testé vis-à-vis d'un seul autre lien, mais vis-à-vis de tous ceux actuellement saturés.

Si la saturation d'un nouveau lien induit une baisse de performance mesurable sur l'un des liens déjà saturés, nous notons cette interférence dans la matrice, mettons fin à la dernière saturation initiée et continuons pour le lien suivant. Si cette nouvelle saturation n'induit aucune variation de bande passante sur les liens déjà saturés, nous continuons par l'ajout d'une saturation supplémentaire.

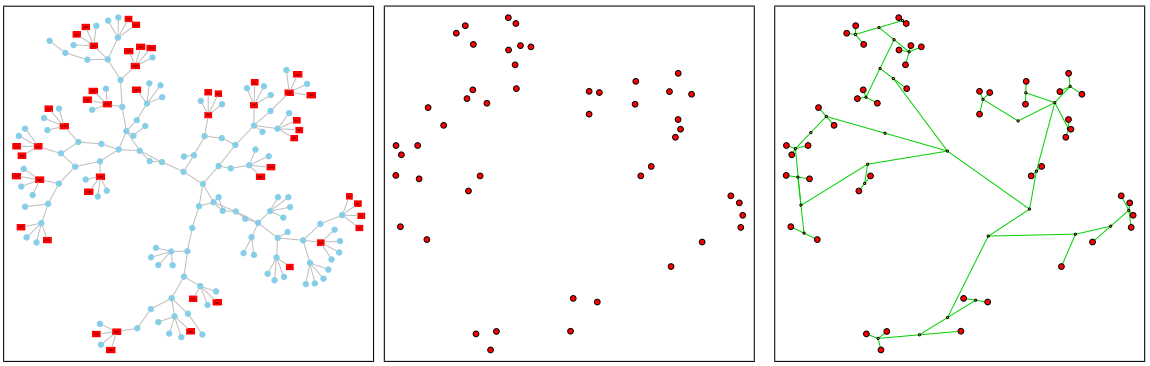
Plus nous parvenons à lancer de saturations en parallèle et plus le temps total de la collecte d'information diminue. Afin de prédire les liens probablement indépendants et guider notre algorithme, une étape préliminaire reconstruit une première approximation de la topologie grâce à `traceroute`. Le graphe résultant peut contenir des erreurs car l'outil utilisé pour le constituer ne capture pas toutes les interférences possibles (du fait par exemple des VLAN, comme nous l'avons vu dans le chapitre 7). Ainsi, il est probable qu'il indique parfois comme indépendants des liens étant dans les faits interférents. Ces informations constituent cependant une bonne approximation permettant de guider relativement efficacement notre algorithme de collecte des informations.

L'accélération offerte par cette optimisation dépend naturellement des caractéristiques du graphe $\widetilde{G}$. Le pire cas est celui où tous les liens interfèrent avec les autres, interdisant tout parallélisme dans les mesures. Le meilleur cas est l'absence totale d'interférence, ce qui permet de tester toutes les combinaisons en parallèle. Cela permet collecter toutes les informations nécessaires en $2|\mathcal{H}|^2$ étapes. Étant donné que la plate-forme cible est typiquement constituée d'une constellation de réseaux locaux formant eux-mêmes le plus souvent des arbres, nous pensons que le gain potentiel de ces optimisations est important.

Par ailleurs, cet algorithme nécessitant une horloge centralisée, les mesures sont synchronisées entre elles par un nœud particulier que nous appelons *maestro*. Le choix de ce nœud ainsi que sa position n'a cependant aucune influence sur les résultats des mesures.

9.5 Exemple de fonctionnement

Cette section introduit un prototype d'ALNeM développé au sein de l'environnement GRAS présenté au chapitre 8. Les résultats présentés ont été obtenus lors d'une exécution dans le simulateur sur une topologie générée par Tiers [Doa96]. La figure 9.4 présente à la fois le graphe utilisé par SimGrid comme plate-forme « réelle » (9.4(a)), les données fournies à ALNeM (9.4(b)) et le graphe G tel que reconstruit par ALNeM (9.4(c)). La figure 9.5 présente les différentes étapes de l'exécution d'ALNeM dans ce cas.



(a) Graphe d'origine ($\widetilde{G}$). (b) Données (plus $I(V, \curlywedge_{rl})$). (c) Résultat d'ALNeM (G).

Fig. 9.4 – Exemple de reconstruction de graphe.

9.6 Résumé et travaux futurs

Dans ce chapitre, nous avons présenté un outil de cartographie du réseau de niveau applicatif nommé ALNeM (Application-Level Network Mapper). Tout comme ENV que nous avons présenté au chapitre 7, cet outil vise à capturer une représentation macroscopique et qualitative du réseau.

Ainsi, nous ne cherchons pas à découvrir le schéma de cablage physique du réseau et les différents chemins empruntés par les paquets (ce qui pourrait être utile à un administrateur

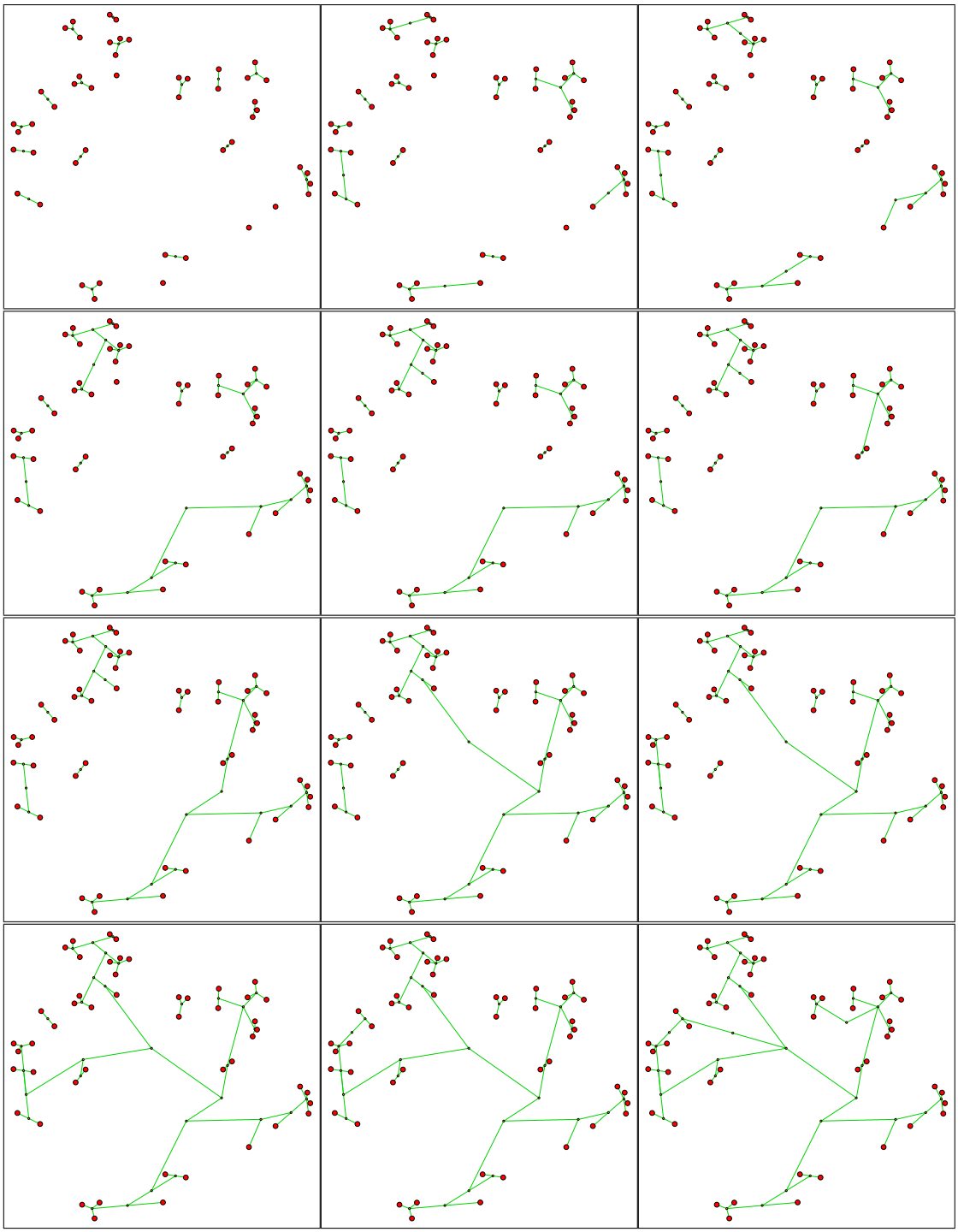


FIG. 9.5 – Étapes successives de la reconstruction de l'exemple.

voulant ausculter son réseau, mais présente peu d'intérêt dans notre contexte), ni même à déterminer avec précision les performances du réseau de bout en bout entre les machines, puisque NWS offre déjà ce service.

L'information qu'ALNeM capture est l'éventuelle interférence (en termes de performances) entre deux flux (ab) et (cd) ayant lieu en même temps, ou au contraire s'ils peuvent s'effectuer en parallèle sans influer l'un sur l'autre. Pour simplifier son usage, cette information est rendue sous forme d'un graphe utilisant le routage en plus court chemin. ALNeM indique que deux flux interfèrent si et seulement si les chemins correspondant dans le graphe produit ont une intersection non nulle.

La reconstruction de cette représentation est un problème difficile, que nous nommons InterferenceGraph. Nous pensons qu'il est NP–complet dans le cas général, même si nous n'avons pas encore réussi à le montrer. Nous avons proposé un algorithme pour résoudre ce problème. Sous certaines conditions, nous avons démontré que cet algorithme trouve une solution optimale s'il existe une solution formant une constellation d'arbres (c'est-à-dire plusieurs arbres dont les nœuds et feuilles sont disjoints et dont les racines sont connectées par une clique complète). Cette démonstration s'appuie sur deux hypothèses sur le graphe réel ayant produit les interférences de l'instance du problème étudiée. La première interdit certaines inconsistances de routage tandis que la seconde demande à ce que le routage soit symétrique.

Nous avons également proposé une extension de cet algorithme pour certaines instances du problème qui imposent d'introduire des cycles dans la solution. L'étude précise du champ d'application de cet algorithme étendu et de ses limites devra cependant faire l'objet de travaux ultérieurs.

Les informations capturées par ALNeM ayant une forme relativement proche de celles offertes par ENV, l'algorithme présenté dans le chapitre 7, section 7.3 reste applicable. Pour tout ensemble de nœuds regroupés par ALNeM sous forme de clique, de sous-arbre ou de cycle, une clique NWS est utilisée pour mesurer la connectivité interne au groupe entre ses membres, et un leader est élu pour mesurer la connectivité avec l'ensemble englobant celui-ci. S'il s'agit d'un sous-arbre (*i.e.*, si ce sous-réseau est connecté à l'extérieur par un bus partagé), nous pouvons encore appliquer l'optimisation consistant à ne mesurer la bande passante qu'entre deux machines représentatives de chaque couple possible. S'il s'agit d'une clique complète ou d'un cycle, nous configurons NWS pour que tous les éléments du sous-réseau fassent partie de la même clique.

Nous avons également présenté comment les informations nécessaires à cette reconstruction peuvent être mesurées sur la plate-forme. Cette étape reste actuellement problématique de par le temps nécessaire à ces mesures. Nous avons donné des pistes pour l'optimiser. La version actuelle semble cependant encore trop limitée pour permettre son usage sur une plate-forme réelle hors du simulateur, malgré son exécution potentiellement parallèle.

Nous avons réalisé une implémentation préliminaire d'ALNeM dans le cadre de GRAS (présenté au chapitre 8) afin de mener facilement des expérimentations sur ALNeM au sein du simulateur tout en conservant la possibilité de déployer cet outil dans la réalité par la suite. Les résultats expérimentaux sur simulateur présentés sont encourageants, mais des tests d'ALNeM sur une plate-forme réelle restent à réaliser.

De plus, nous souhaiterions améliorer la détection et le traitement des cas où les hypothèses 1 et 2 ne sont pas respectées, ainsi que des erreurs de mesures faites lors de la collecte des informations.

Ensuite, nous souhaiterions enrichir notre modélisation pour pouvoir prédire à partir de quels débits ou quel nombre de flux les interférences vont se révéler. Nous avons ici supposé implicitement que deux flux concurrents sont suffisants pour détecter les conflits de ressource potentiels. Ce n'est pourtant pas le cas et certaines ressources ne deviennent limitantes (créant ainsi l'interférence) que lorsqu'elles sont partagées par trois flux ou plus. Nous pourrions prendre ce fait en compte en parlant de N-interférences lorsque N flux sont nécessaires pour révéler le conflit de ressources ou bien en attachant un seuil à chaque séparateur du réseau représentant le débit seuil à atteindre pour que les ressources qu'il représente deviennent limitantes.

Une autre généralisation possible serait de prendre en compte les mécanismes de qualité de service (QoS) faisant que les ressources du réseau peuvent allouer des priorités différentes aux flux concurrents.

Par ailleurs, l'usage des résultats de cet outil devrait également faire l'objet de travaux plus approfondis par exemple basés sur des évaluations supplémentaires au sein du simulateur et sur des plates-formes réelles. L'adéquation de l'algorithme de déploiement de NWS à partir des informations fournies par ALNeM pour remplir les critères de qualité énoncés dans le chapitre 6, section 6.1.2 reste ainsi à étudier.

De même, nous souhaiterions étudier comment appliquer les résultats fournis par ALNeM à d'autres champs que le déploiement automatique des composants de la grille. Les pistes les plus prometteuses semblent être la simulation de plate-forme ou la prédiction du temps nécessaire à des communications de groupe.

Enfin, l'étape des mesures préalables nécessite une étude approfondie pour permettre l'usage de cet outil dans des conditions réelles. Une première piste serait d'augmenter encore son degré de parallélisme par une analyse plus poussée des informations rapidement accessibles grâce à des outils tels que `traceroute` ou `ping`.

Pour limiter le nombre d'hôtes dans chaque expérience de cartographie (et ainsi réduire le temps de mesure), nous prévoyons également d'étudier la fusion de cartographies existantes, au besoin en réalisant des tests complémentaires. De même, une version incrémentale des algorithmes mis en œuvre dans ALNeM permettrait d'éviter de recalculer l'intégralité de la plate-forme lorsque l'un des nœuds de la grille apparaît ou disparaît.

De manière plus générale, nous pensons que l'étape des mesures peut être grandement optimisée si elle interagit plus finement avec le processus de reconstruction, et ordonne les tests en fonction de la représentation actuelle de la plate-forme.

Le point d'orgue de cette approche serait d'intégrer ALNeM à une infrastructure du réseau telle que NWS. Cela permettrait de réaliser la plupart des tests perturbants pour le réseau lorsque les utilisateurs réguliers n'en ont pas besoin. Cela assurerait également que les changements structurels du réseau seraient détectés automatiquement sans nécessiter de re-cartographier le réseau dans son intégralité tout en permettant d'effectuer régulièrement les mesures nécessaires afin de s'assurer de la validité de leurs résultats. NWS pourrait naturellement bénéficier de ces informations pour se configurer automatiquement à la topologie ainsi découverte.

Chapitre 10

Conclusion et perspectives

10.1 Travaux présentés

Les travaux présentés dans cette thèse ont porté sur les difficultés soulevées par l'introduction de l'hétérogénéité et de la dynamicité des plates-formes modernes dans le cadre du calcul distribué et parallèle.

Comme nous l'avons vu dans la première partie introductive de ce manuscrit, le *metacomputing* vise à constituer une plate-forme communément nommée *grille* et résultant de l'agrégation de ressources distribuées. L'utilisation de cette machine virtuelle implique le déploiement d'une infrastructure logicielle spécifique. L'une des approches les plus classiques pour cela (nommée GridRPC) consiste à étendre le modèle des RPC (*Remote Procedure Call* – invocation de procédures à distance) afin de l'adapter à la grille.

En amont de toute solution d'ordonnancement sur la grille se pose cependant le problème de l'obtention d'informations pertinentes et actualisées sur les capacités et caractéristiques de la plate-forme. Par ailleurs, ce manque d'informations est également problématique pour le déploiement et la configuration de la grille elle-même. Il est indispensable pour cela de connaître la topologie d'interconnexion des machines disponibles afin d'adapter au mieux ces logiciels et leur permettre de passer à l'échelle lorsque la taille de la plate-forme augmente.

Nous nous sommes attachés dans cette thèse à apporter des éléments de réponse pour l'obtention de ces informations. Nous avons présenté dans la seconde partie des solutions permettant de fournir efficacement des informations d'ordre quantitatif sur les capacités de la plate-forme tandis que la troisième partie a été l'occasion pour nous d'aborder la façon dont des informations d'ordre plus qualitatif sur ses caractéristiques peuvent être obtenues.

Nous avons détaillé dans le chapitre 3 notre méthodologie pour l'obtention d'informations quantitatives sur la grille, nommée *macro-benchmarking*. Cette approche pragmatique se caractérise par la mesure directe des variations de performances constatées et par l'usage de modèles simples et robustes destinés à simplifier l'usage de ces résultats et non à expliquer ces variations.

Nous avons ensuite présenté dans le chapitre 4 une bibliothèque nommée Fast implémentant cette méthodologie afin d'informer d'autres applications des capacités actuelles de la

plate-forme. Des informations ainsi rendues disponibles concernent à la fois les *disponibilités* de la plate-forme ainsi que les *besoins* de routines séquentielles. FAST utilise l'infrastructure de surveillance de la plate-forme NWS (présenté dans la section 2.2.2) pour obtenir les informations sur les disponibilités de la plate-forme. Nos travaux nous ont permis d'étendre ce système pour en améliorer la réactivité aux changements induits par les décisions d'ordonnancement de l'infrastructure elle-même, ainsi que pour en réduire le temps de réponse et en permettre un usage intensif dans un système interactif. Les besoins des routines en termes de temps de calcul et d'espace mémoire sont mesurés directement par étalonnage des applications au moment de l'installation. Cette technique s'apparente à celles utilisées lors de l'optimisation de bibliothèques à hautes performances et permet de tenir compte de façon transparente de l'adéquation entre l'application, le système d'exploitation et l'architecture matérielle.

Nous avons conclu cette seconde partie en montrant dans le chapitre 5 comment cette bibliothèque est utilisée par divers projets de *metacomputing*. Les applications principales sont l'environnement GridRPC DIET (développé en partie au sein de notre équipe), un outil interactif de visualisation des disponibilités de la plate-forme ou encore le projet GridTLSE, qui vise à développer un système expert permettant de sélectionner l'implémentation de services d'algèbre linéaire creuse la plus adaptée à un jeu de données. Nous avons également présenté une extension de FAST pour les routines parallèles développée par Eddy Caron et Frédéric Suter.

En introduction de la troisième partie, nous avons détaillé notre méthodologie quant à l'obtention d'informations qualitatives sur la grille. Notre objectif principal ici est d'obtenir la topologie d'interconnexion des machines afin de pouvoir déterminer les interférences mutuelles entre différents flux de données concurrents en termes de performances. Il ne s'agit cependant pas de développer un nouvel outil destiné aux administrateurs souhaitant ausculter leur réseau, mais bien de déterminer les performances que les applications peuvent escompter sur ce réseau.

Notre motivation principale pour cela est l'automatisation du déploiement et de la configuration de NWS. En effet, il est indispensable que les expérimentations sur le réseau n'entrent pas en collision car les tests ne rapportent que la capacité dont leurs paquets ont pu profiter. Ainsi, si deux paquets de test transitent par le même lien au même instant, il est probable que chacun ne mesure que la moitié de la capacité réelle du lien. Pour résoudre ce problème, NWS introduit le concept de *cliques* de machines et assure par un algorithme de passage de jetons qu'un seul test sera mené au sein de ce groupe de machines. L'identification des groupes de machines interconnectées par un lien commun et devant donc être placées dans la même clique reste cependant une tâche difficile dont l'automatisation nécessite des informations sur la topologie du réseau.

Après avoir introduit dans le chapitre 6 les différentes solutions de cartographie du réseau existantes, nous avons présenté dans le chapitre 7 une expérience de déploiement automatique de NWS sur une partie du réseau de notre laboratoire grâce à ENV (développé à l'Université de Californie de San Diego). Malheureusement, cet outil ne permet que de reconstruire une vision arborescente de la topologie du réseau n'indiquant donc pas tous les liens existants. Si cette approche est suffisante dans le paradigme maître/esclaves pour lequel ENV a été développé, cette limitation rend malheureusement ce projet inadapté à nos travaux.

Par ailleurs, la non reproductibilité des expériences sur la grille (due à sa dynamicité et au partage des ressources avec d'autres utilisateurs) s'avère très problématique pour la mise

au point d'applications destinées à cette plate-forme. De plus, l'établissement d'une plate-forme d'expérimentation stable demande souvent des efforts très importants de la part des développeurs. Ces constatations nous ont poussé à mettre en place une solution pour la mise au point confortable d'infrastructures destinées à la grille nommée GRAS et que nous avons présenté dans le chapitre 8. Les applications développées dans ce cadre peuvent être testées sur un simulateur lors de leur mise au point avant d'être déployées sans modification sur la plate-forme réelle. Cette approche par simulateur est bien plus efficace qu'une émulation complète de la plate-forme, mais impose naturellement aux applications l'usage de l'interface spécifique de cet outil. Il n'est donc pas possible d'étudier de cette manière une application existante.

Forts de cette solution technique, nous avons présenté dans le chapitre 9 une formalisation mathématique du problème de la cartographie automatique du réseau basée uniquement sur des mesures actives réalisables sans privilèges particuliers sur les machines ou réseaux de la plate-forme. Nous avons proposé une modélisation du réseau dans laquelle l'interférence entre deux flux concurrents est simplement exprimée par le fait que l'intersection des plus courts chemins est non vide. Les outils mathématiques résultants nous ont permis de proposer un algorithme permettant de reconstruire un tel graphe pour certaines instances du problème. Nous avons démontré sous certaines conditions que cet algorithme permet d'obtenir une solution optimale lorsqu'il existe un graphe formant une constellation d'arbres (un ensemble d'arbres dont les racines sont connectées par une clique complète) et vérifiant les contraintes exprimées par le résultat des mesures. Nous avons par la suite étendu cet algorithme pour lui permettre de traiter certains cas où il est indispensable d'introduire des cycles dans le graphe reconstruit pour satisfaire les contraintes imposées. Nous avons également implémenté un prototype nommé ALNeM de cet algorithme grâce à l'environnement GRAS et présenté des résultats expérimentaux préliminaires obtenus au sein du simulateur SimGrid.

10.2 Résumé de nos contributions

- Formalisation de la méthodologie, dessin de l'architecture et réalisation de la bibliothèque de prédiction de performances Fast totalisant plus de 15 000 lignes de code source et utilisé par plusieurs projets externes (chapitres 4 et 5) ;
- modifications de NWS pour en améliorer la réactivité (chapitre 4) ;
- dessin de l'architecture et la réalisation de l'environnement GRAS pour la mise au point d'applications, totalisant plus de 10 000 lignes de code source (chapitre 8) ;
- en collaboration avec Arnaud Legrand (doctorant au LIP), démonstration sous certaines conditions d'un algorithme de découverte de topologie du réseau adapté à la grille et implémentation d'un prototype totalisant plus de 2 000 lignes de code source (chapitre 9) ;
- diffusion des résultats dans plusieurs revues, conférences et colloques et à l'occasion de divers séminaires (page 147).

10.3 Perspectives

Les résultats de ces travaux ouvrent de nombreuses perspectives, et nous allons maintenant présenter quelques unes d'entre elles.

Dans un avenir proche, l'un des objectifs principaux est la suppression des limitations actuelles de l'environnement de mise au point GRAS. Ces problèmes concernent principalement l'expressivité des messages, les performances des applications développées dans ce cadre ainsi que l'interopérabilité avec des infrastructures basées sur d'autres solutions de communication. Des pistes possibles pour cela sont données dans la section 8.4. À plus long terme, nous souhaiterions également intégrer plus de fonctionnalités utiles aux infrastructures distribuées sous la forme de modules de localisation des données, de sécurité ou de surveillance des disponibilités de la plate-forme.

Au niveau de la prédiction des capacités de la grille, notre objectif principal est de confronter notre bibliothèque à un usage dans un environnement de production. Les orientations actuelles comprennent la mise en place de l'architecture afin d'en permettre l'usage à des non-spécialistes, ainsi que la tolérance aux pannes en cas de problèmes au niveau des différentes parties de l'infrastructure déployée. L'intégration au sein de DIET et la base d'expérimentation résultante en est une illustration concluante.

Au niveau de la découverte automatique de la topologie, les perspectives sont également nombreuses. D'un point de vue théorique, nous souhaiterions étudier les limites du champ d'application de l'algorithme proposé afin de le généraliser si cela s'avère possible ou bien montrer la NP-complétude du problème de décision associé dans le cas contraire. D'un point de vue plus pratique, la phase de mesures préliminaires à la reconstruction du graphe nécessite d'être encore optimisée afin d'en améliorer l'extensibilité.

Bien que notre motivation principale pour la découverte de la plate-forme ait été l'automatisation du déploiement de NWS, il est clair que ces résultats peuvent être appliqués dans d'autres cadres, comme la simulation réaliste de la plate-forme ou les prédictions des performances de communication globales. Nous souhaiterions ainsi utiliser ces résultats au sein de l'extension parallèle de notre outil de prédiction des performances afin d'en simplifier l'usage par des non-spécialistes. Par ailleurs, nous souhaiterions également quantifier plus précisément la qualité du déploiement de NWS ainsi obtenu, et dans quelles mesures les contraintes à ce sujet exprimées dans la section 6.1.2 sont satisfaites.

Un autre axe de recherche particulièrement prometteur est l'intégration des algorithmes utilisés dans ALNeM au sein de l'infrastructure déployée par NWS. De cette façon, les tests d'ALNeM pourraient avoir lieu lorsque le réseau est inutilisé par ailleurs, ce qui limiterait les perturbations induites tout en améliorant la qualité des mesures. Cela assurerait de plus la configuration automatique de NWS en lui même et offrirait cette extension naturelle aux informations déjà fournies aux applications clientes. Cette intégration constitue l'objet du stage de post-doctorat à l'Université de Californie de Santa Barbara que j'aurai l'honneur de réaliser l'an prochain.

Annexe A

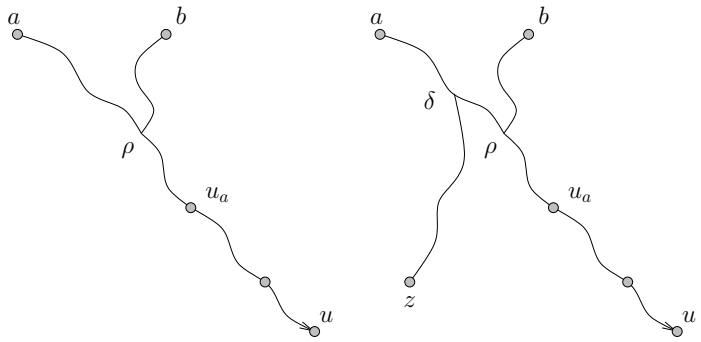
Démonstrations du chapitre 9 (ALNeM)

A.1 Lemme 9.3 (lemme de séparation)

Lemme 9.3 : $\forall a, b \in \mathcal{H},\ a \perp b \iff \exists \rho \in \widetilde{V} \ \Big/\ \forall z \in \mathcal{H} : \rho \in (a \rightarrow z) \cap (b \rightarrow z)$.

Preuve du sens direct Supposons que $a \perp b$. Soit $u \in \mathcal{H} \backslash \{b, a\}$ et u_a le premier nœud distinct de a apparaissant dans $(a \rightarrow u) \cap \mathcal{H}$. La construction de ces points est représentée par la figure A.1(a).

Soit ρ le premier sommet de $(a \rightarrow u_a) \cap (b \rightarrow u_a)$. Notons que $\rho \in \widetilde{V}$.



(a) Construction de u_a et ρ. (b) Une situation absurde.

FIG. A.1 – Esquisse de la preuve du lemme 9.3.

1. Montrons tout d'abord que u_a est le premier nœud distinct de a et b apparaissant dans $(b \rightarrow u_a) \cap \mathcal{H}$.

   Démonstration. Grâce à l'hypothèse 1 (consistance du routage), nous avons les relations suivantes :

$$\begin{cases} (a \rightarrow u_a) = (a \rightarrow \rho) \cup (\rho \rightarrow u_a) \\ (b \rightarrow u_a) = (b \rightarrow \rho) \cup (\rho \rightarrow u_a) \end{cases}$$

– Si $\rho = a$ alors il n'existe pas de $c \in \mathcal{H}\backslash\{b, a\}$ tel que $c \in (b \rightarrow a)$ (dans le cas contraire, nous aurions $bc \not\!/\!/_{rl} au_a$, ce qui est en contradiction avec l'hypothèse $a \perp b$). Ainsi, u_a est le premier nœud distinct de a apparaissant dans $(b \rightarrow u_a) \cap \mathcal{H}$.
– Si $\rho \neq a$, alors $(\rho \rightarrow u_a) \cap \mathcal{H} = \{u_a\}$ de par la définition de u_a. Il ne peut donc pas exister de $c \in \mathcal{H}\backslash\{b\}$ tel que $c \in (b \rightarrow \rho)$. En effet, nous aurions dans le cas contraire $b \rightsquigarrow c \rightsquigarrow \rho \rightsquigarrow u_a$ et $a \rightsquigarrow \rho \rightsquigarrow u_a$. Comme la définition de ρ nous donne $(b \rightarrow \rho) \cap (a \rightarrow u_a) = \{\rho\}$, nous aurions $bc \not\!/\!/_{rl} au_a$, ce qui est en contradiction avec l'hypothèse. u_a est donc le premier nœud distinct de a et b apparaissant dans $(b \rightarrow u_a) \cap \mathcal{H}$. ■

2. Montrons maintenant que $\forall z,\ \rho \in (a \rightarrow z)$.

   Démonstration (par l'absurde). Supposons qu'il existe un z tel que $\rho \notin (a \rightarrow z)$. Par l'hypothèse 1 (consistance du routage), nous aurions la relation suivante (illustrée par la figure A.1(b)) :
   $$(a \rightarrow z) \cap (\rho \rightarrow u_a) = \emptyset$$
   Soit δ le premier sommet apparaissant dans $(a \rightarrow z) \cap (a \rightarrow u_a)$. Par application de l'hypothèse 1 (consistance du routage), nous savons que $\delta \in (a \rightarrow \rho)$ et $(\delta \rightarrow z) \cap (\rho \rightarrow u_a) = \emptyset$.
   ρ étant le premier élément de $(a \rightarrow z) \cap (b \rightarrow z)$ par définition, nous savons que $(a \rightarrow \rho) \cap (b \rightarrow \rho) = \{\rho\}$. Puisque $\delta \in (a \rightarrow \rho)$, nous avons donc $\delta \notin (b \rightarrow \rho)$.
   En utilisant une fois de plus l'hypothèse 1 (consistance du routage), notre relation devient $(a \rightarrow z) \cap (b \rightarrow u_a) = \emptyset$. Il s'agit de la définition de $(ab) \not\!/\!/_{rl} (bu_a)$, ce qui est en contradiction avec l'hypothèse $a \perp b$. ■

3. Par symétrie, nous avons $\rho \in (b \rightarrow z)$.

Nous obtenons donc ainsi le résultat souhaité :

$$\forall z \in \mathcal{H} : \begin{cases} \rho \in (a \rightarrow z) \\ \rho \in (b \rightarrow z) \end{cases}$$

Preuve du sens contraire Cette implication est triviale. Soit $\rho \in \widetilde{V}$ tel que :

$$\forall z \in \mathcal{H} : \begin{cases} \rho \in (a \rightarrow z) \\ \rho \in (b \rightarrow z) \end{cases}$$

Il en découle que $\forall u, v \in \mathcal{H},\ \rho \in (a \rightarrow u) \cap (b \rightarrow v)$, et donc que $(au) \curlywedge (bv)$.

Donc $a \perp b$.

A.2 Théorème 9.1 (l'interférence totale est une relation d'équivalence)

Théorème 9.1. *L'interférence totale est une relation d'équivalence et pour chaque classe d'équivalence, il existe un séparateur commun à tous les couples de la classe.*

Pour montrer que la relation d'interférence totale est une relation d'équivalence, nous devons bien évidemment montrer pour cela qu'il s'agit d'une relation réflexive (ce que nous ferons avec le lemme A.1), symétrique (ce que le lemme A.2 montrera) et transitive (lemme A.3). Ce dernier lemme démontrera également l'existence d'un séparateur commun à toute classe d'équivalence.

Lemme A.1 ($\perp$ est une relation réflexive). *C'est-à-dire* $\forall a \in \mathcal{H}, a \perp a$.

Démonstration. Pour tout $a, u, v \in \mathcal{H}$, $(a \rightarrow u) \cap (a \rightarrow v) = \{a\} \neq \emptyset$. Donc $(au) \between (av)$, et donc $a \perp a$. ■

Lemme A.2 ($\perp$ est une relation symétrique). *C'est-à-dire* $\forall a, b \in \mathcal{H}, a \perp b \Leftrightarrow b \perp a$.

Démonstration. Ceci se déduit trivialement du fait $\between$ est symétrique par construction. ■

Lemme A.3 ($\perp$ est une relation transitive). *Étant donné trois nœuds a, b, $c \in \mathcal{H}$, si $\big(a \perp b\big) \wedge \big(b \perp c\big)$ alors ces trois nœuds ont un séparateur commun et donc $a \perp c$.*

Démonstration. Soit $a, b, c \in \mathcal{H}$ tels que $a \perp b$ et $b \perp c$.

Soit $u \in \mathcal{H}$. Soit u_a le premier nœud distinct de a apparaissant dans $(a \rightarrow u) \cap \mathcal{H}$. Soit ρ le premier nœud apparaissant $(a \rightarrow u_a) \cap (b \rightarrow u_a)$ et σ le premier nœud apparaissant dans $(b \rightarrow u_a) \cap (c \rightarrow u_a)$.
ρ est donc un séparateur de a et b tandis que σ est un séparateur de b et c (ceci, ainsi que l'existence de ρ et σ, est obtenu par application du lemme 9.3). Ces définitions impliquent également que $\rho \in (b \rightarrow c)$ et $\sigma \in (b \rightarrow c)$.

1. Montrons que $\rho = \sigma$ par l'absurde.

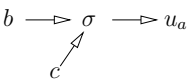
   Démonstration. Supposons pour cela que $\rho \neq \sigma$. Deux cas sont possibles : $(b \rightarrow \rho \rightarrow \sigma \rightarrow c)$ ou $(b \rightarrow \sigma \rightarrow \rho \rightarrow c)$.

   (a) Supposons que $(b \rightarrow \rho \rightarrow \sigma \rightarrow c)$ (Relation notée Ⓐ).

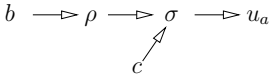
      i. Montrons que $(a \rightarrow \rho) \cap (c \rightarrow \sigma) = \emptyset$ est impossible dans ce cas puisque cela impliquerait $(ab) \mathbin{/\!\!/} (cu_a)$.

      Démonstration.

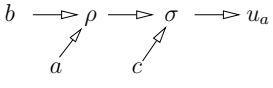
      La définition de σ nous donne :

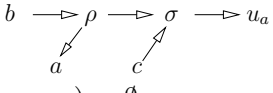
      Ce qui avec Ⓐ devient :

      Ce qui avec la définition de σ devient :

      Par l'hypothèse 2 (de symétrie), il en découle :

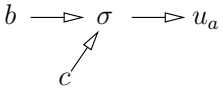
      Puisque $\rho \neq \sigma$, il en découle que $(b \rightarrow \rho \rightarrow a) \cap (c \rightarrow \sigma \rightarrow u_a) = \emptyset$.

Une réécriture de chaque opérande de $\cap$ donne : $(b \to a) \cap (c \to u_a) = \emptyset$.
Donc, $(ab) /\!\!/ (cu_a)$, ce qui est absurde puisque $b \perp c$. ■

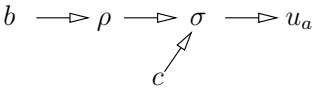
ii. Montrons que $(a \to \rho) \cap (c \to \sigma) \neq \emptyset$ est également impossible dans ce cas.

Démonstration. Soit δ le premier élément de $(a \to \rho) \cap (c \to \sigma)$.

La définition de δ et Ⓐ donnent :



De par la définition de ρ, il en découle :



Donc ρ est placé avant σ sur $(b \to u_a) \cap (c \to u_a)$, ce que est absurde si $\rho \neq \sigma$ car σ est le premier élément de cet ensemble par définition. ■

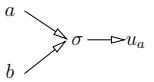
(i) et (ii) étant les deux seuls cas possibles, la relation Ⓐ est donc absurde.

(b) Par symétrie, $(b \to \sigma \to \rho \to c)$ est également impossible.

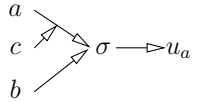
Les deux cas étant impossible, la proposition $\rho \neq \sigma$ est absurde, et donc $\rho = \sigma$. ■

2. σ est donc à la fois le premier élément de $(a \to u_a) \cap (b \to u_a)$ et de $(b \to u_a) \cap (c \to u_a)$.

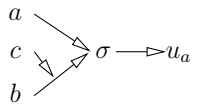
Puisque σ est le premier élément de $(a \to u_a) \cap (b \to u_a)$, on a :



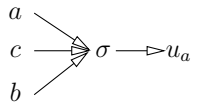
Pour l'ajout de c sur cette figure, seuls les trois cas suivants respectent le fait que σ est le premier élément de $(b \to u_a) \cap (c \to u_a)$:

– 

Cette situation est impossible car elle impliquerait avec les deux hypothèses (routage consistant et symétrique) que $(a \to c) \cap (b \to u_a) = \emptyset$ et donc que $(ac) /\!\!/ (bu_a)$, ce qui est absurde puisque $a \perp b$.

– 

Par symétrie, cette situation impliquerait $(bc) /\!\!/ (au_a)$.

– 

Cette situation est donc la seule possible, et il en découle que σ est le premier séparateur de a et c.

Il en découle que σ est un séparateur commun à a, b et c.
En particulier, c'est un séparateur de a et c, et donc $a \perp c$.

■

Remarque A.1. *Le lemme A.3 (transitivité de la relation $\perp$) n'est vrai que si l'hypothèse 2 (de symétrie du routage) est vérifiée. La figure A.2 montre une situation ne vérifiant pas cette hypothèse et où $a \perp b$ et $b \perp c$, mais $a \not\perp c$ (car $(ad) /\!\!/ (cb)$).*

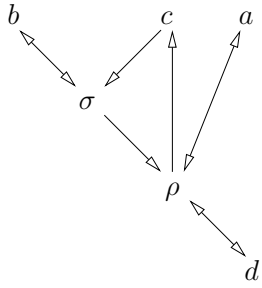


FIG. A.2 – Contre-exemple au lemme A.3 (transitivité de la relation $\perp$) lorsque l'hypothèse 2 (routage symétrique) n'est pas vérifiée.

A.3 Théorème 9.2 (représentativité du séparateur)

Théorème 9.2. *Soit $\mathcal{C}$ une classe d'équivalence pour $\perp$ et ρ un séparateur de ses éléments.*

$$\forall a \in \mathcal{C},\ \forall b, u, v \in \mathcal{H},\ \ (a,u) \curlywedge (b,v) \Leftrightarrow (\rho,u) \curlywedge (b,v)$$

Démonstration. Soit ρ un séparateur commun aux nœuds de $\mathcal{C}$ (dont l'existence est donnée par le théorème 9.1). Soit $a \in \mathcal{C}$ et $b, u, v \in \mathcal{H}$.

Sens direct (par l'absurde) : Supposons que $(a,u) \curlywedge (b,v)$ et $(\rho,u) \not\curlywedge (b,v)$.
Nous avons donc $(a \rightarrow u) \cap (b \rightarrow v) \neq \emptyset$ et $(\rho \rightarrow u) \cap (b \rightarrow v) = \emptyset$.
Avec l'hypothèse 1 (consistance de routage), il en découle que $(a \rightarrow \rho) \cap (b \rightarrow v) \neq \emptyset$.
Soit θ le premier nœud de $((a \rightarrow \rho) \setminus \{\rho\}) \cap (b \rightarrow v)$.
Le chemin de a à v est donc $(a \rightarrow \theta \rightarrow v)$ et ρ en fait partie, ce qui est absurde.

Nous venons de démontrer que : $\big((a,u) \curlywedge (b,v)\big) \vee \big((\rho,u) \not\curlywedge (b,v)\big)$.
C'est-à-dire $\big((a,u) \curlywedge (b,v)\big) \Rightarrow \big((\rho,u) \curlywedge (b,v)\big)$.

Sens contraire : Supposons que $(a,u) \not\curlywedge (b,v)$, c'est-à-dire $(a \rightarrow u) \cap (b \rightarrow v) = \emptyset$.
ρ étant un séparateur de a, il est sur tous les chemins de a vers d'autres nœuds. En particulier, $(a \rightarrow u) = (a \rightarrow \rho \rightarrow u)$.
Un réécriture triviale de la relation précédente donne donc $(a \rightarrow \rho \rightarrow u) \cap (b \rightarrow v) = \emptyset$.
D'où, $(\rho \rightarrow u) \cap (b \rightarrow v) = \emptyset$, c'est-à-dire $(\rho,u) \not\curlywedge (b,v)$.

Nous venons de démontrer que : $\big((a,u) \not\curlywedge (b,v)\big) \Rightarrow \big((\rho,u) \not\curlywedge (b,v)\big)$.
C'est-à-dire $\big((\rho,u) \curlywedge (b,v)\big) \Rightarrow \big((a,u) \curlywedge (b,v)\big)$.

Nous avons donc effectivement démontré : $(a,u) \curlywedge (b,v) \Leftrightarrow (\rho,u) \curlywedge (b,v)$. ■

A.4 Correction de l'algorithme Arbres

Démonstration (par induction). Nous voulons démontrer la relation suivante (notée Ⓐ dans cette preuve) : $I(a,u,b,v) = 1 \Leftrightarrow \left(a \underset{G}{\longrightarrow} u\right) \cap \left(b \underset{G}{\longrightarrow} v\right) \neq \emptyset$.

On note $\{a_i\}$ la suite des leaders des composantes connexes contenant a à l'étape i.

Hypothèse d'induction (HdI) : À l'étape i, chaque quadruplet de nœuds $a.u, b, v$ peut être dans l'un des trois cas suivants :

- complètement déconnectés (*i.e.,* , il n'existe aucun chemin reliant deux de ces points).
- complètement connectés, et Ⓐ est vérifiée.
- partiellement connectés, et les liens existants ne contredisent pas la relation Ⓐ.

Initialisation : étant donné qu'aucun point n'est connecté dans le graphe à l'étape 0, l'hypothèse d'induction est trivialement vraie dans ce cas.

Induction : Supposons HdI vraie pour toutes les étapes j telles que $0 \leq j < i$, montrons qu'elle est alors vraie à l'étape i.

Bien que ne posant pas de difficulté particulière, cette démonstration est relativement complexe. Il est nécessaire d'étudier treize cas, selon que a, u, b et v sont placés dans les mêmes composantes connexes à la fin de l'étape i ou non. La figure A.3 présente ces différents cas, qu'il est possible de ranger en quatre catégories.

1. Le cas ⑬ est trivial, car aucun des point n'est encore connecté.
2. Les cas ⑥, ⑦ et ⑧ sont les symétriques respectifs des cas ③, ④ et ⑤ car $(a, u) \between (b, v) \Leftrightarrow (b, v) \between (a, u)$ par définition de $\between$.
3. Pour les cas ①, ③ et ④, il faut distinguer différents sous-cas dont la plupart impliquent l'existence d'un cas déjà étudié aux étapes précédentes (ce qui, par application de l'HdI, démontre la correction de Ⓐ). Étudions par exemple le cas ③. Selon l'arité de a_i, trois cas sont alors possibles.
   (a) Si a_i est d'arité 1, alors a_{i-1} est également dans la situation ③. Ayant été traité à une étape précédente, nous savons alors par application de (HdI) que Ⓐ est vérifiée.
   (b) Si a_i est d'arité 2, alors seulement deux des nœuds a, u et b étaient connectés à l'étape $i-1$. Les différents cas possibles se ramènent alors clairement à d'autres cas étudiés : $(au)(b)(v)$ est la situation ⑤, $(ab)(u)(v)$ est la situation ⑨ et $(ub)(a)(v)$ est la situation ⑫. Ces différentes situations, ayant eu lieu à des étapes précédentes, Ⓐ est respectée.
   (c) Si a_i est d'arité 3, alors les trois points étaient placés dans des sous-arbres différents à l'étape $i-1$ et donc $a_i \in (a \to u) \cap (b \to v)$. Par ailleurs, ces trois points ne peuvent être regroupés de la sorte au sein d'un même sous-arbre par Arbres, que si $a_{i-1} \perp u_{i-1} \perp b_{i-1}$. En particulier, $(a_{i-1}, u_{i-1}) \between (b_{i-1}, v_{i-1})$. Par application du théorème 9.2 (représentativité du séparateur), il en découle que $(a, u) \between (b, v)$.
   Nous avons bien démontré que $(a \to u) \cap (b \to v) \neq \emptyset$ seulement si $(a, u) \between (b, v)$ dans ce cas, c'est-à-dire que la condition Ⓐ est respectée dans ce cas.

   La condition Ⓐ est donc vérifiée dans tous les sous-cas de ③.
4. Les arguments dans tous les autres cas sont très semblables au sous-cas (3c). Dans cas ② et ⑤, on montre à la fois que $(a \to u) \cap (b \to v) = \emptyset$ et que $(a, u) /\!\!/ (b, v)$ tandis que dans les cas ⑨, ⑩, ⑪ et ⑫, on a à la fois $(a \to u) \cap (b \to v) \neq \emptyset$ et $(a, u) \between (b, v)$.

Donc, dans tous les cas possible à l'étape i, la relation Ⓐ est respectée si elle l'était aux étapes précédentes.

Par induction, le théorème de correction de l'algorithme Arbres est donc démontrée. ■

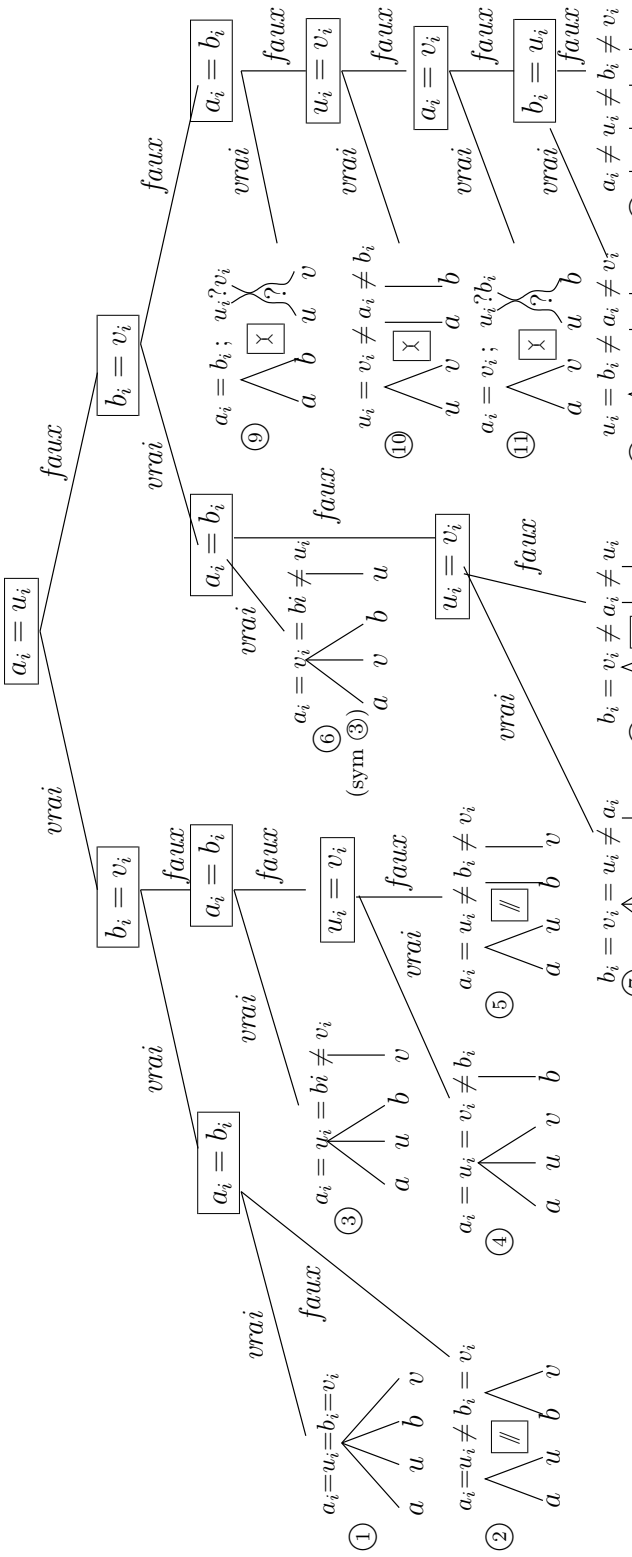


FIG. A.3 – Les différents cas de l'induction montrant la correction de ARBRES.

A.5 Conditions de terminaison de l'algorithme Arbres

Théorème 9.4. *Pour une instance de* InterferenceGraph *donnée, s'il est possible de construire une solution G étant un arbre, alors* Arbres *termine avec un seul leader.*

Cette démonstration est triviale : S'il est possible de construire une solution sous la forme d'un arbre, chaque sous-arbre de cette solution est en interférence totale avec les nœuds externes. Donc, tant que tous les nœuds ne sont pas regroupés au sein de la même composante connexe, c'est qu'il reste des sous-arbre n'ayant pas encore été traités par l'algorithme, qui est donc assuré de pouvoir regrouper des sous-arbres à chaque étape jusqu'à obtenir un graphe connexe.

Bibliographie

[ABD00] D. Arnold, D. Bachmann, and J. Dongarra. Request Sequencing: Optimizing Communication for the Grid. In A. Bode, T. Ludwig, W. Karl, and R. Wismuller, editors, *Euro-Par 2000 Parallel Processing, 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science*, pages 1213–1222, Munich Germany, August 2000. Springer Verlag.

[ADKL01] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[AE98] B. Armstrong and R. Eigenmann. Performance Forecasting: Towards a Methodology for Characterizing Large Computational Applications. In *Proceedings of the International Conference on Parallel Processing*, pages 518–526, August 1998.

[AGO96] P. Arbenz, W. Gander, and M. Oettli. The Remote Computation System. In H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *High Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 820–825. Springer-Verlag, Berlin, 1996.

[AGO97] P. Arbenz, W. Gander, and M. Oettli. The Remote Computational System. *Parallel Computing*, 23(10):1421–1428, 1997.

[All03] M. Allman. An Evaluation of XML-RPC. *ACM Performance Evaluation Review*, 30(4), March 2003.

[AP03] P. Amestoy and M. Pantel. Grid-TLSE: A Web Expertise Site for Sparse Linear Algebra. http://www.enseeiht.fr/lima/tlse/grid_tlse.pdf, June 10-13 2003. St Girons (France).

[ATN+00] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima. Performance Evaluation Model for Scheduling in a Global Computing System. *International Journal of High-Performance Computing Applications*, 14(3):268–279, 2000.

[Aum02] O. Aumage. Heterogeneous Multi-Cluster Networking with the Madeleine III Communication Library. In *Proc. 16th Intl. Parallel and Distributed Processing Symposium, 11th Heterogeneous Computing Workshop (HCW 2002)*, Fort Lauderdale, April 2002. Held in conjunction with IPDPS 2002.

[BAG02] R. Buyya, D. Abramson, and J. Giddy. A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker. *Future Generation Computer Systems, Elsevier Science*, 18(8):1061–1074, October 2002.

[BBE+99] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. Improving Simulation for Network Research. Technical Report 99-702, University of Southern California, 1999. Available at http://citeseer.nj.nec.com/bajaj99improving.html.

[BBF+02] A. Bassi, M. Beck, G. Fagg, T. Moore, J. Plank M. Swany, and R. Wolski. The Internet BackPlane Protocol: A Study in Resource Sharing. In *Proceedings of the second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2002)*, Berlin, Germany, May 2002. http://loci.cs.utk.edu/ibp/files/pdf/StudyResourceSharing.pdf.

[BCW] H. Burch, B. Cheswick, and A. Wool. Internet Mapping Project. http://www.lumeta.com/mapping.html.

[BEG+03] R. Badia, F. Escale, E. Gabriel, J. Gimenez, R. Keller, J. Labarta, and S. Müller. Performance Prediction in a Grid Environment. In *Proceedings of the 1st European Across Grids Conference*, Santiago de Compostela, 2003.

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielson, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. Technical report, World Wide Web Consortium, May 2000.

[BJ00] A. Bierman and K. Jones. Physical Topology MIB. RFC2922, september 2000.

[BMB00] J.A. Brown, A.J MacGregor, and H.W. Braun. Network Performance Visualization: Insight Through Animation. In *Passive and Active Measurement Workshop*, pages 33–41, Hamilton, New Zealand, April 2000.

[Bou00] J. Bourgeois. *Prédiction de performances statique et semi-statique dans les systèmes répartis hétérogènes*. PhD thesis, Université de Franche-Comté, Janvier 2000.

[BW97] F. Berman and R. Wolski. The AppLeS Project: A Status Report. In *Proceedings of the 8th NEC Research Symposium*, volume 16, Berlin, Germany, May 1997.

[CCLU99] E. Caron, O. Cozette, D. Lazure, and G. Utard. Virtual Memory Management in Data Parallel Applications. In *Proc. of HPCN'99 (High Performance Computing and Networking)*, Lecture Notes in Computer Science. Springer, April 1999.

[CD98] H. Casanova and J. Dongarra. Using Agent-Based Software for Scientific Computing in the Netsolve System. *Parallel Computing*, 24:1777–1790, 1998.

[CDL+02] E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[CDPV03] E. Caron, F. Desprez, F. Petit, and V. Villain. A Hierarchical Resource Reservation Algorithm for Network Enabled Servers. In *IPDPS'03. The 17th International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[CDZ97] K.L. Calvert, M.B. Doar, and E.W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997. Available at http://citeseer.nj.nec.com/calvert97modeling.html.

[CKP+96] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Communications of the ACM*, 39(11):78–95, November 1996.

[CLM03] H. Casanova, A. Legand, and L. Marchal. Scheduling Distributed Applications: the SimGrid Simulation Framework. In *Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, may 2003.

[CLU00] E. Caron, D. Lazure, and G. Utard. Performance Prediction and Analysis of Parallel Out-of-Core Matrix Factorization. In *Proceedings of the 7th International Conference on High Performance Computing (HiPC'00)*, Dec 2000.

[CLZB00a] 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 (HCW'2000)*, pages 349–363, 2000.

[dBKB02] M. den Burger, T. Kielmann, and H.E. Bal. TOPOMON: A Monitoring Tool for Grid Network Topology. In *International Conference on Computational Science (ICCS 2002)*, volume 2330 of *LNCS*, pages 558–567, Amsterdam, April 21-24 2002.

[DDP+98] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. Scheduling Block-Cyclic Array Redistribution. In E.H. D'Hollander, G.R. Joubert, F.J. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions*, pages 227–234. North Holland, 1998.

[DGK+01] P. Dinda, T. Gross, R. Karrer, B Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The Architecture of the Remos System. In *10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, August 2001.

[Din02] P. Dinda. Online Prediction of the Running Time of Tasks. *Cluster Computing*, 5(3), 2002.

[DJ01] F. Desprez and E. Jeannot. Adding Data Persistence and Redistribution to NetSolve. Technical Report RR2001-39, LIP, December 2001.

[Dji03] S. Djilali. P2P-RPC: Programming Scientific Applications on Peer-to-Peer Systems with Remote Procedure Call. In *GP2PC2003 (Global and Peer-to-Peer Computing on Large Scale Distributed Systems) colocated with IEEE/ACM CCGRID2003*, Tokyo Japan, May 2003.

[DLP03] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present, and Future. *Concurrency and Computation: Practice and Experience*, 1(18), 2003.

[Doa96] M.B. Doar. A Better Model for Generating Test Networks. In *Globecom '96*, Nov 1996. Available at http://citeseer.nj.nec.com/doar96better.html.

[Dow99] A.B. Downey. Using Pathchar to Estimate Internet Link Characteristics. In *Measurement and Modeling of Computer Systems*, pages 222–223, 1999. Available at http://citeseer.nj.nec.com/downey99using.html.

[DR03] L. DongWoo and R.S. Ramakrishna. Disk I/O Performance Forecast Using Basic Prediction Techniques for Grid Computing. In Victor Malyshkin, editor, *7th International Conference, PaCT03*, volume 2763 of *LNCS*, pages 259–269. Springer, Sept 2003.

[EBS02] G. Eisenhauer, F.E. Bustamante, and K. Schwan. Native Data Representation: An Efficient Wire Format for High-Performance Distributed Computing. *IEEE transactions on parallel and distributed systems*, 13(12):1234–1246, december 2002.

[FE99] I. Foster and C. Kesselman (Eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[FFF99] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-law Relationships of the Internet Topology. In *SIGCOMM*, pages 251–262, 1999. Available at http://citeseer.nj.nec.com/faloutsos99powerlaw.html.

[FGNC01] G. Fedak, C. Germain, V. Néri, and F. Cappello. XtremWeb: A Generic Global Computing System. In *CCGRID2001, workshop on Global Computing*, 2001.

[FIG+97] R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Standard RFC2068, Internet Society, January 1997.

[FJ98] M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[FJJ+01] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM Transactions on Networking*, oct 2001. Available at http://citeseer.nj.nec.com/francis01idmaps.html.

[FK97a] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[FMM00] M. Ferris, M. Mesnier, and J. Moré. NEOS and Condor: Solving Optimization Problems Over the Internet. *ACM Transactions on Mathematical Software*, 26(1):1–18, March 2000.

[GAL+02] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and Applications. In *5th International Conference on High Performance Computing for Computational Science – VECPAR'2002*, Lecture Notes in Computer Science, pages 197–227, Porto, Portugal, June 26-28 2002. Springer.

[GG00] M. Good and J. Goux. iMW : A Web-Based Problem Solving Environment for Grid Computing Applications. Technical report, Department of Electrical and Computer Engineering, Northwestern University, 2000.

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

[Gro01] Apache Group. Log4j project. http://jakarta.apache.org/log4j, 2001.

[GSW02] L. Gong, X.-H. Sun, and E.F. Watson. Performance Modeling and Prediction. *IEEE Transactions on Computers*, 51(9):1041–1055, September 2002.

[Hay] J. Hayes. AppLeS Multi-Protocol Interprocess Communication. http://grail.sdsc.edu/projects/ampic/.

[Her98] S.A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1998.

[Hew95] Hewlett-Packard Company. *NetPerf: A Network Performance Benchmark*, 1995.

[HPBG02] R. Harakaly, P. Primet, F. Bonnassieux, and B. Gaidioz. Probes Coordination Protocol for Network Performance Measurement in Grid Computing Environment. *Journal of Parallel and Distributed Computing Practices, special issue on Internet-based Computing*, 2002.

[HRR02] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.

[HSG99] I. A. Howes, M. C. Smith, and G. S. Good. *Understanding and Deploying LDAP Directory Services*. Macmillian Technical Publishing, 1999. ISBN: 1-57870-070-1.

[JCSM96] B.L. Jacob, P.M. Chen, S.R. Silverman, and T.N. Mudge. An Analytical Model for Designing Memory Hierarchies. *IEEE Transactions on Computers*, 45(10):1180–1194, 1996.

[JG99] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[KFLR01] N.H. Kapadia, J.A.B. Fortes, M.S. Lundstrom, and D. Royo. PUNCH: A Computing Portal for the Virtual University. *International Journal of Engineering Education (IJEE). In special issue on Virtual Universities and Engineering Education*, 17(2):207–219, April 2001.

[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, held in conjunction with the GGF8 meetings*, Seattle, WA, USA, June 2003.

[KOKM02] S.W. Kim, P. Ohly, R.H. Kuhn, and D. Mokhov. A Performance Tool for Distributed Virtual Shared-Memory Systems. In Acta Press, editor, *4th IASTED Int. Conf. Parallel and Distributed Computing and Systems*, pages 755–760, Calgary, Canada, 2002.

[KSR90] S. Kirkpatrick, M.K. Stahl, and M. Recker. RFC 1166: Internet Numbers, July 1990. Obsoletes RFC1117, RFC1062, RFC1020 Status: INFORMATIONAL. Available at ftp://ftp.math.utah.edu/pub/rfc/rfc1166.txt.

[KTF03] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[LN01] J. Liu and J.M. Nicol. *DaSSF 3.1 User's Manual*, April 2001. Available at http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/papers/dassf-manual-3.1.ps.

[LOG01] B.B. Lowekamp, D.R. O'Hallaron, and T.R. Gross. Topology Discovery for Large Ethernet Networks. In ACM Press, editor, *ACM SIGCOMM 2001*, pages 237–248, San Diego, California, 2001. Available at http://citeseer.nj.nec.com/500129.html.

[LS02] U. Lang and R. Schreiner. *Developing Secure Distributed Systems with CORBA*. Artech House, 2002.

[Lu02] C. Lu. *Application Signatures for Scientific Codes*. PhD thesis, Univ. of Illinois at Urbana-Champaign, 2002.

[LWF+98] C.A. Lee, R. Wolski, I. Foster, C. Kesselman, and J. Stepanek. A Network Performance Tool for Grid Environments. In *Proceedings of 7th IEEE International Symposium on High Performance Distributed Computing*, pages 260–267, 1998.

[MC00a] S. Matsuoka and H. Casanova. Network-Enabled Server Systems and the Computational Grid. http://www.eece.unm.edu/~dbader/grid/WhitePapers/GF4-WG3-NES-whitepaper-draft-000705.pdf, July 2000. Grid Forum, Advanced Programming Models Working Group whitepaper (draft).

[MC00b] W. Matthews and L. Cottrell. The PingER Project: Active Internet Performance Monitoring for the HENP Community. *IEEE Communications Magazine*, 38(5), 2000.

[Mes93] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[Mic87] Sun Microsystems. XDR: External Data Representation Standard. Technical Report RFC1014, Internet Society, June 1987.

[MNS+00] S. Matsuoka, H. Nakada, M. Sato, , and S. Sekiguchi. Design Issues of Network Enabled Server Systems for the Grid. http://www.eece.unm.edu/~dbader/grid/WhitePapers/satoshi.pdf, 2000. Grid Forum, Advanced Programming Models Working Group whitepaper.

[MS96] L.W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[MS00] N. Miller and P. Steenkiste. Collecting Network Status Information for Network-Aware Applications. In *INFOCOM'00*, pages 641–650, 2000.

[NMS+02] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. GridRPC: A Remote Procedure Call API for Grid Computing. http://www.eece.unm.edu/~apm/docs/APMGridRPC0702.pdf, July 2002. Grid Forum, Advanced Programming Models Working Group whitepaper.

[NNTH+03] A. Natrajan, A. Nguyen-Tuong, M.A. Humphrey, M. Herrick, B.P. Clarke, and A.S. Grimshaw. The Legion Grid Portal. *Concurrency and Computation: Practice and Experience. Special Issue: Grid Computing Environments*, 14(13-15):1365–1394, Jan 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(5–6):649–658, Oct. 1999.

[NZ01] E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches, 2001. Available at citeseer.nj.nec.com/article/ng01predicting.html.

[OR02] J. Oly and D. Reed. Markov Model Prediction of I/O Request for Scientific Application. In *Proc. of the 2002 International Conference on Supercomputing*, Jun 2002.

[Pax97] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, 1997.

[PF97] V. Paxon and S. Floyd. Why We Don't Know How to Simulate the Internet. In *Proceedings of the Winder Communication Conference*, December 1997. Available at http://citeseer.nj.nec.com/article/floyd99why.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'02)*, April 15-19 2002.

[Rab03] M. Rabbat. Multiple source network tomography. Master's thesis, Rice University, May 2003.

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

[RL95] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC1771, March 1995.

[RS87] V. Rayward-Smith. UET Scheduling with Unit Interprocessor Communication Delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[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.

[SDGM94] V. Sunderam, J. Dongarra, A. Geist, and R. Manchek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20(4):531–547, april 1994.

[SGI] SGI. Performance Co-Pilot: Monitoring and Managing System-Level Performance. http://www.sgi.com/software/pcp/.

[SHS00] E.A.M. Shriver, B. Hillyer, and A. Silberschatz. Performance Analysis of Storage Systems. In *Performance Evaluation*, pages 33–50, 2000.

[SLJ+00] H.J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A.A. Chien. The MicroGrid: a Scientific Tool for Modeling Computational Grids. In *Supercomputing*, 2000. Available at http://www.sc2000.org/techpapr/papers/pap.pap286.pdf.

[SNS+02] T. Suzumura, H. Nakada, M. Saito, S. Matsuoka, Y. Tanaka, and S. Sekiguchi. The Ninf Portal: An Automatic Generation Tool for Grid Portals. In *Proceeding of Java Grande 2002*, pages 1–7, November 2002.

[SS96] R.H. Saavedra and A.J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, 1996.

[SSK97] S. Seshan, M. Stemm, and R.H. Katz. SPAND: Shared Passive Network Performance Discovery. In *USENIX Symposium on Internet Technologies and Systems*, 1997. Available at http://citeseer.nj.nec.com/seshan97spand.html.

[Sut02] F. Suter. *Parallélisme mixte et prédiction de performances sur réseaux hétérogènes de machines parallèles*. PhD thesis, École Normale Supérieure de Lyon, November 2002.

[Tel00] G. Tel. *Introduction to Distributed Algorithms*. Cambridge, 2 edition, 2000.

[TWML01] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – A Distributed Job Scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[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, number 11, pages 154–165, November 2001.

[vH03] B. van Heukelom. Development of Strategies for the Integration of Parallel Applications into the DIET Environment by Example of a Program for Genome Sequence Analysis. Master's thesis, Technische Universität München, Fakultät für Informatik, July 2003.

[VSF02] S. Vazhkudai, J. Schopf, and I. Foster. Predicting the Performance of Wide Area Data Transfers. In *Proceedings of the 16th Int'l Parallel and Distributed Processing Symposium (IPDPS 2002)*, April 2002.

[WGT00] R. Wolski, B. Gaidioz, and B. Tourancheau. Synchronizing Network Probes to avoid Measurement Intrusiveness with the Network Weather Service. In *9th IEEE High-performance Distributed Computing Conference*, pages 147–154, August 2000.

[WL02] J. B. Weissman and B.-D. Lee. The Virtual Service Grid: An Architecture for Delivering High-End Network Services. *Concurrency and Computation: Practice and Experience*, 14(4):287–319, 2002.

[WLS+02] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[WPD01] R.C. Whaley, A. Petitet, and J.J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

[WSF+03] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In IEEE Press, editor, *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.

[WSH99] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems, Metacomputing Issue*, 15(5–6):757–768, Oct. 1999.

[WSH00] R. Wolski, N. T. Spring, and J. Hayes. Predicting the CPU Availability of Time-Shared Unix Systems on the Computational Grid. *Cluster Computing*, 3(4):293–301, 2000. Available at http://www.cs.ucsd.edu/users/rich/papers/nws-cpu.ps.gz.

[WSMW00] D.A.B. Weikle, K. Skadron, S.A. McKee, and Wm.A. Wulf. Caches As Filters: A Unifying Model for Memory Hierarchy Analysis. Technical Report CS-2000-16, University of Virginia, 1 2000.

Liste des publications

Chapitres de livres

[A] Eddy Caron, Frédéric Desprez, Eric Fleury, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. *Calcul réparti à grande échelle*, chapter Une approche hiérarchique des serveurs de calculs. Hermès Science Paris, 2002. ISBN 2-7462-0472-X.

Revues internationales avec comité de lecture

[B] Eddy Caron, Serge Chaumette, Sylvain Contassot-Vivier, Frédéric Desprez, Eric Fleury, Claude Gomez, Maurice Goursat, Emmanuel Jeannot, Dominique Lazure, Frédéric Lombard, Jean-Marc Nicod, Laurent Philippe, Martin Quinson, Pierre Ramet, Jean Roman, Franck Rubi, Serge Steer, Frédéric Suter, and Gil Utard. Scilab to Scilab//, the OURAGAN Project. *Parallel Computing*, 11(27):1497–1519, Oct 2001.

[C] Eddy Caron, Frédéric Desprez, Martin Quinson, and Frédéric Suter. Performance Evaluation of Linear Algebra Routines for Network Enabled Servers. *Parallel Computing, special issue on Cluters and Computational Grids for scientific computing (CCGSC'02)*, 2003.

Revues nationales avec comité de lecture

[D] Martin Quinson. Un outil de prédiction dynamique de performances dans un environnement de metacomputing. *Technique et Science Informatique*, 21(5):685–710, 2002. Numéro spécial RenPar'13.

Conférences internationales avec comité de lecture

[E] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, Aug 2002. Springer-Verlag.

[F] Philippe Combes, Frédéric Lombard, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In *Proceedings of the 7th Asian Computing Science Conference*, number 2550 in Lecture Notes in Computer Science, pages 110–124. Springer-Verlag, Jan 2002.

[G] Frédéric Desprez, Martin Quinson, and Frédéric Suter. Dynamic Performance Forecasting for Network Enabled Servers in an heterogeneous Environment. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, volume 3, pages 1421–1427. CSREA Press, June 25-28 2001.

[H] Martin 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'02)*, April 15-19 2002.

Conférences nationales avec comité de lecture

[I] Frédéric Lombard, Martin Quinson, and Frédéric Suter. Une approche extensible des serveurs de calcul. In *Treizièmes Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, pages 79–84, Paris, La Villette, April 24-27 2001.

[J] Martin Quinson. Un outil de modélisation de performances dans un environnement de metacomputing. In *13ième Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, pages 85–90, Paris, La Villette, April 24-27 2001.

Découverte automatique des caractéristiques et capacités d'une plate-forme de calcul distribué

Ce mémoire traite de l'obtention d'informations pertinentes, récentes et précises sur l'état courant des plates-formes de calcul modernes. Souvent dénommés *grilles*, ces environnements se différencient des machines parallèles les ayant précédés par leur nature intrinsèquement hétérogène et fortement dynamique.

Ce document est découpé en trois parties. La première présente les difficultés spécifiques à la grille en se basant sur une sélection de projets d'infrastructures pour la grille et en détaillant les solutions proposées dans ce cadre.

La seconde partie montre comment obtenir efficacement des informations quantitatives sur les capacités de la grille et leur adéquation aux besoins des routines à ordonnancer. Après avoir détaillé les problèmes rencontrés dans ce cadre, nous explicitons notre approche, nommée *macro-benchmarking*. Nous présentons ensuite l'outil FAST, développé dans le cadre de cette thèse et mettant cette méthodologie en œuvre. Nous étudions également comment cet outil est utilisé dans différents projets.

La troisième partie traite de l'obtention d'une vision plus qualitative des caractéristiques de la grille, telle que la topologie d'interconnexion des machines la constituant. Après une étude des solutions classiques du domaine, nous présentons ALNEM, notre solution de cartographie automatique ne nécessitant pas de privilège d'exécution particulier. Cet outil est basé sur l'environnement GRAS, développé dans le cadre de ces travaux pour la mise au point des constituants de la grille.

Mots clés : Prédiction de performances, cartographie automatique, metacomputing, grille de calcul, développement d'applications distribuées.

Automatic discovery of the characteristics and capacities of a distributed computational platform

This thesis is devoted to the monitoring of modern computational platforms in order to obtain relevant, up to date and accurate information about them. Often called *Grids*, those environments differ from the preceding parallel machines by their intrinsic heterogeneity and high dynamicity.

This document is organized in three parts. The first one presents the specific difficulties introduced by this platform, highlighting them in a selection of grid infrastructure projects and detailing the existing solutions.

The second part shows how to get efficiently quantitative informations about the grid capacities and their suitability to the needs of the routines to schedule. After a discussion of the problems encountered, we detail our approach which we call *macro-benchmarking*. We then present FAST, a tool implementing this methodology. We eventually detail how FAST is used in several other projects.

The third part introduces how to get a more qualitative view of the grid characteristics such as the topology of the network interconnecting the hosts. After a study of the existing solutions in this domain, we present ALNEM our solution to automatically map the network without relying on specific execution privileges on the platform. This tool is based on GRAS, our framework for the development of grid infrastructure.

Keywords: Performance forecasting, topology mapping, metacomputing, grid computing, distributed application development.