Exécutions de programmes parallèles à passage de messages sur grille de calcul

Stéphane Genaud

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Exécutions de programmes parallèles à passage de messages sur grille de calcul

Habilitation à Diriger des Recherches

présentée et soutenue publiquement le *8 décembre 2009*

pour l'obtention du

Habilitation à Diriger des Recherches
de
l'Université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Stéphane Genaud

Composition du jury

Président :	Claude Godart, Professeur à l'Université Henri Poincaré.
Rapporteurs :	Christophe Cérin, Professeur à l'Université de Paris Nord. Frédéric Desprez, Directeur de Recherche à l'INRIA. Thierry Priol, Directeur de Recherche à l'INRIA.
Examinateurs :	Pascal Bouvry, Professeur à l'Université du Luxembourg. Jens Gustedt, Directeur de Recherche à l'INRIA.

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

Mis en page avec la classe thloria.

Remerciements

Quand on arrive à cette page de remerciements, on a généralement fait le plus dur : on vient de finaliser le manuscrit sous la pression constante liée au respect des échéances[1]. Il faut rendre sa copie à l'heure, il faut attendre l'aval des pré-rapporteurs, puis celui de l'école doctorale, celui du conseil scientifique, s'inscrire comme étudiant, et enfin, le plus difficile, trouver une date de soutenance, c'est-à-dire arriver à faire déplacer des gens aux compétences très pointues, au même endroit et à la même heure alors que leurs agendas respectifs sont remplis d'obligations en tout genre. Je remercie donc les membres de ce jury de m'avoir accordé une priorité sans doute assez haute dans l'algorithme d'ordonnancement de leurs tâches. J'en suis très honoré et je les en remercie.

Il faut cependant voir le bon côté des choses. Sans *deadline*, le chercheur ne finirait sans doute rien. On n'a jamais fini de ne pas tout comprendre, alors il faut toujours plus de temps. Cette *deadline* passée, je peux tourner une page et prendre quelques minutes pour regarder dans le rétroviseur. Cette introspection m'amène immanquablement aux personnes que j'ai rencontrées. Peut être encore plus dans ce métier que dans d'autres, en raison de la liberté des choix qui nous est laissée, les gens que nous croisons influencent largement notre parcours.

Je pense en premier lieu à Guy-René, sans qui je n'aurais jamais eu l'idée de faire ce métier. À Besançon, où j'ai commencé ma thèse, je pense à mes congénères avec qui j'ai partagé la bonne humeur du *blockhaus*, David, Christophe, Eric, Laurent, François, Pascal. Vint ensuite la constitution de l'équipe ICPS, qui a pu être perçue à ses débuts comme un groupuscule parallélisant isolé à Illkirch, mais qui s'est révélée être une famille accueillante et solide. Je pense à Phil. Prof. Gégé[2], dont la mauvaise foi atteint son paroxysme quand on parle de l'Alsace[3], à Vincent pour son amour du barbecue et des polyèdres, à 10^6 *Volts* Catherine et son indéfectible empathie, à Benoît le new-yorkais d'Herrlisheim. À Eric aussi, dont la vulnérabilité face aux contraintes humaines n'a d'égale que son invulnérabilité aux échecs, à Philippe, co-bureau d'un temps, dont la quête spirituelle a évolué en quelques années de la théorie des catégories au zen. Aux premières heures de TAG, il y eut Romaric, éternel sceptique, qui vint amener sa sensibilité, puis Benjamin l'esthète, et Arnaud qui se repose d'une dure journée de travail en programmant un module noyau Linux/MIPS pour gérer le port parallèle d'une SGI O2 bonne pour la casse. Il y eut Frédéric, de classe mondiale et pourfendeur de virgule mal placée, avec qui nous avons commencé à travailler ensemble le jour de son départ pour Lyon. Il y eut aussi Guillaume, dont le rythme de migration[4] est bien connu des receveurs autoroutiers. Je pense à Alain qui nous a rejoint pour nous faire profiter de sa culture en informatique et en proverbes orientaux[5] et à Julien, le petit dernier, toujours partant pour un projet[6]. Il y a eu tous les doctorants, dont bien sûr Choopan[7], Olivier, et Jean-Christophe, que je remercie de m'avoir aidé à traquer (pour

[1] dans notre métier, on dit plus souvent *deadlines* : ça sonne mieux, ça semble plus léthal. Et c'est un mot qu'on emploie très souvent ...

[2] tel qu'il m'apparaît sur ICQ.

[3] Les polynômes d'Ehrhart **et** le foie gras ont été inventés en Alsace, et non dans le Périgord.

[4] il suffit d'observer la camionnette pleine de meubles qui suit les axes Bordeaux-Strasbourg ou Strasbourg-Cadarache, apparaissant à intervalles réguliers de deux ans.

[5] je médite toujours sur le proverbe cambodgien *« Le boeuf est lent mais la terre est patiente ... »*.

[6] il va nous coûter cher celui-là ...

[7] dont le papa qui me sera éternellement reconnaissant de m'être occupé de son fils, ne rate pas une occasion de m'offrir des cravates. Choopan, pourrais tu lui dire que je veux bien un autre motif que le

son bien) Choopan afin de l'amener de force au bureau certains matins. En pensant à Jean-Christophe, je dois bien reconnaître que je ne l'ai jamais vu que d'humeur enjouée et optimiste durant toute sa thèse. Avouez qu'il y a là quelque chose de surnaturel, et donc d'inquiétant ... Je pense évidemment aussi amicalement à Marc avec qui j'ai partagé de nombreuses chasses aux bugs et quelques séances (en dilettante) de kung-fu[8].

Dans le laboratoire, je pense aussi à Pierre ("– tu veux un café ?") et à Alexandre les darwinistes, à Jean-Jacques et Stéphane, les fidèles de RGE, avec qui j'ai sillonné les routes du Réseau (autoroutier) Grand Est. J'ai aussi une pensée particulière pour Sébastien, avec qui j'ai passé plus de temps à discuter dans les trains ou en voiture que dans un bureau.

A Nancy, il y eut Jens, que j'ai l'impression de toujours pouvoir trouver à son bureau[9], disponible pour un conseil, Martin, qui m'a accordé tant de *time-slices* de dix minutes qu'en deux ans on a bien dû se parler une journée en temps cumulé (la version non-électronique de la conversation instantanée), Sylvain, sans qui ce document n'aurait pas atteint le bon bureau au service de la recherche et des études doctorales, FredS les bons plans, Louis-Claude (un autre sceptique), Cristian et Pierre-Nicolas. J'ai une pensée amicale particulière pour Emmanuel, qui m'a souvent hébergé, et qui a pu partir l'âme en paix à Bordeaux le jour où il m'a battu aux échecs. Je n'oublie pas Stéphane à qui je ne refuse jamais une longue conversation téléphonique à 23h le dimanche soir, et Virginie, copine de galère Grid'5000.

Le travail présenté dans ce manuscrit n'aurait pas pu avoir lieu de la même manière sans l'effort collectif autour de la plate-forme Grid'5000. Tous les contributeurs à cet effort ont permis de faire émerger une communauté de chercheurs à l'intérieur de laquelle les échanges sont nombreux et fructueux. Je pense aussi à tous ces collègues qu'on a plaisir à croiser au gré de nos déplacements, au hasard d'une attente en salle d'embarquement ou à un *social event*, avec qui ont peut reprendre une conversation interrompue quelques mois plus tôt sur un autre continent. Bien que je sois sûr d'en oublier, je pense entres autres à Camille, Christian, Eddy, Denis, Derrick, Franck, Gabriel, Jaspal, Jean-Marc, Nouredine, Olivier, Pascale, Reiner, Ronan, et tant d'autres[10]. On peut avoir des discussions passionnantes partout, y compris dans les locaux de l'avenue de la forêt noire, n'est-ce pas Brigitte et Monique ?

Enfin, je dédie ce manuscrit à mon frère, à mes parents, et à Sophie, Clara et Léna. C'est bien le moins que je puisse faire après plusieurs années où mon entourage à forcément pensé[11], à un moment ou à un autre *que je passais trop de temps sur mon ordinateur*. Je sais qu'ils disent ça pour mon bien[12]. J'espère qu'ils trouveront avec ce document une explication, ou tout du moins une justification, à tout ce temps que je passe à faire des choses qui n'ont pas l'air d'être du vrai travail, mais qui en revanche, ont vraiment l'air de me passionner ...

---

petit éléphant thaïlandais ?

[8] il n'y a aucun rapport entre les deux, les techniques de kung-fu n'étant d'aucune utilité contre les bugs informatiques. En revanche, j'étais content d'avoir à côté de moi, Marc, grand connaisseur des arts martiaux, sur la ligne bleue du métro de Los Angeles.

[9] même s'il a récemment engagé sur TipIt.com une compétition avec Ingrid sur le nombre de miles parcourus.

[10] *...je n'aurais jamais dû me lancer dans cette liste ...*

[11] ou me l'ont-ils dit ? Je ne me souviens plus très bien, mais à y réfléchir, oui peut être ...

[12] Cette notion est toutefois difficile à définir.

à Sophie, Clara, Léna.

Table des matières

Chapitre 1

Introduction

Contexte Ce document présente les activités de recherche que j'ai menées entre 2002 et 2009. Ces activités se sont déroulées au Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection (LSIIT) à Strasbourg, auquel j'étais rattaché au titre de mon emploi de maître de conférences, et au centre de recherche INRIA Nancy - Grand Est, où j'ai passé deux ans en détachement avec l'équipe AlGorille (sept 2007 – août 2009).

Ce travail est thématiquement homogène. Par le biais d'un projet baptisé *Transformations et Adaptions pour la Grille*, accepté dans le cadre de l'Action Concertée Incitative (ACI) GRID[1], j'ai initié en 2002 un élargissement des thématiques de recherches de mon équipe (ICPS), spécialisée dans le parallélisme, au domaine des grilles. Ce thème est resté un axe de recherche de l'équipe après la fin du projet. La proximité thématique avec l'équipe-projet INRIA m'a permis, lors de mon détachement, à la fois de continuer mon travail sur l'intergiciel P2P-MPI en y impliquant l'équipe AlGorille, et de démarrer des travaux nouveaux (optimisation d'opérations de communications collectives, projet d'ANR autour de SimGrid).

Contenu Mes recherches se sont concentrées sur les programmes parallèles à passage de messages. Ce modèle de programmation, qui se matérialise la plupart du temps par des programmes MPI en C ou Fortran, est celui qui est typiquement employé pour des applications de calcul intensif, sur des machines parallèles ou des clusters. Le coût de possession élevé (climatisation, maintenance) de ce type d'équipement fait qu'encore aujourd'hui, la justification la plus fréquente à l'acquisition d'un cluster est l'exploitation d'applications en mode production.

Dans le même temps, les progrès énormes des réseaux et l'augmentation continue de la puissance des processeurs peuvent laisser penser que des puissances de calcul fantastiques peuvent également être dégagées en agrégeant les capacités de très nombreuses machines. Cette notion assez large de mutualisation et d'accès à des ressources de calcul distantes est celle de *grille de calcul* que j'ai employée dans le titre. Elle peut varier selon les contextes, mais on peut se réfèrer celle donnée dans l'ouvrage fondateur de I. Foster et C. Kesselmann, *The Grid : Blueprint for a New Computing Infrastructure*. On y trouve cette définition : "*A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.*" Aujourd'hui, plusieurs infrastructures satisfont cette définition. Les grilles institutionnelles comme TeraGrid[2], ou la grille européenne pour les sciences EGEE[3] sont maintenant pérennes, et l'ordre de grandeur du nombre de CPUs accessibles est de l'ordre de 100 000 pour EGEE. Le projet BOINC qui

[1] http ://www-sop.inria.fr/aci/grid/public/acigrid.htm
[2] http ://www.teragrid.org/
[3] EGEE (Enable Grid for E-SciencE), http ://www.eu-egee.org/

permet de fédérer des ordinateurs sur la base du volontariat, avec de l'ordre de 1 000 000 utilisateurs enregistrés peut également être considéré comme une infrastructure stable.

Cependant, si ces grilles sont aujourd'hui bien utilisées en mode production, le type des programmes parallèles déployés n'est pas quelconque : il s'agit quasi exclusivement de tâches indépendantes, en général coordonnées par une tâche maître. Dans des cas plus complexes, ce sont des workflows qui sont traités (tâches dépendantes des sorties d'autres tâches). Ce type restrictif de parallélisme est beaucoup plus aisé à mettre en œuvre dans des environnements hétérogènes à large échelle dont les ressources de calcul sont volatiles, car il existe des solutions simples pour remédier aux contraintes classiques de ces environnements. La tolérance aux pannes peut prendre la forme d'une sauvegarde des résultats intermédiaires et d'un redémarrage du calcul non terminé sur une nouvelle ressource, il y a peu de problème de connectivité réseau car les ressources de calcul n'ont souvent qu'à communiquer avec le maître (et non entre elles), et l'équilibrage de charge (nécessaire en hétérogène) peut être géré par le maître qui module la charge de travail affectée aux ressources en fonction de leurs comportements.

Pour caractériser la situation actuelle, on peut dire que dans la très grande majorité des cas, les modèles de programmation employés sont directement liés à l'architecture choisie pour l'exécution : un modèle à passage de messages sur les clusters, et des formes de client-serveur sur les grilles. Cependant, certains problèmes ne peuvent se paralléliser de manière efficace qu'à l'aide d'un modèle de programmation plus général que le client-serveur. Le travail présenté dans ce manuscrit a pour objectif d'étudier ces situations intermédiaires, dans lesquelles on exécute un programme parallèle à passage de messages sur une grille de calcul. Les utilisateurs potentiels de ce type de calcul ne sont pas ceux du calcul intensif, qui nous le répétons, achètent le matériel spécifique qui permet d'obtenir les meilleures performances, ni ceux qui ont des problèmes trivialement parallèles, mais ceux qui cherchent une solution peu coûteuse économiquement grâce à la mutualisation de moyens de calculs, bien que le programme parallèle à exploiter ne soit pas trivial.

Plan Le document est organisé de la manière suivante. La première partie présente des travaux liés à une application pré-existante de géophysique pour la tomographie sismique, développée en langage C et Fortran. La problématique générale dans cette partie est l'étude du portage et des performances qu'on peut attendre d'une telle application sur une grille. Le premier chapitre décrit dans ses grandes lignes la conception et la réalisation de cette application menées en collaboration avec des géophysiciens. Ce travail, commencé deux ans avant ma réorientation thématique sur les grilles, a pu être intégré dans les problématiques liées aux grilles, en tant qu'application exemple. En menant conjointement ces travaux, nous avons en permanence vérifié que les choix de parallélisation étaient compatibles (autant que possible) aussi bien avec des clusters homogènes qu'avec des environnements d'exécution hétérogènes et à forte latence. Le chapitre rapporte les différentes expérimentations menées avec des machines réparties à l'échelle nationale en utilisant essentiellement Globus et MPICH-G2. Le deuxième chapitre illustre l'étude théorique qu'on peut faire de l'équilibrage de charge, directement appliqué au code de géophysique présenté au chapitre précédent. La deuxième partie du document présente le travail fait sur l'intergiciel P2P-MPI. Le premier chapitre de cette partie, de loin le plus important, présente l'ensemble de l'environnement, qui nous a permis de proposer et tester différentes idées, qui mises ensemble, améliorent la prise en charge de l'exécution de programmes parallèles à passage de message sur des grilles. Parmi celles ci, la notion de découverte dynamique des ressources de calcul disponibles à chaque exécution, la tolérance aux pannes par réplication des calculs, la détection des pannes par gossiping. Les deux derniers chapitres, plus courts, présentent chacun une application, l'une de *machine learning*, l'autre de *clustering*, que nous avons parallélisé à l'aide de P2P-MPI en collaboration avec des collègues des domaines respectifs. Ces deux exemples viennent illustrer le type d'utilisation qui peut être fait de

P2P-MPI. Enfin, une synthèse des résultats est présentée, suivie des perspectives de recherches dans le domaine.

Première partie

Application de géophysique

Chapitre 2

Application de géophysique : la tomographie sismique

2.1 Contexte

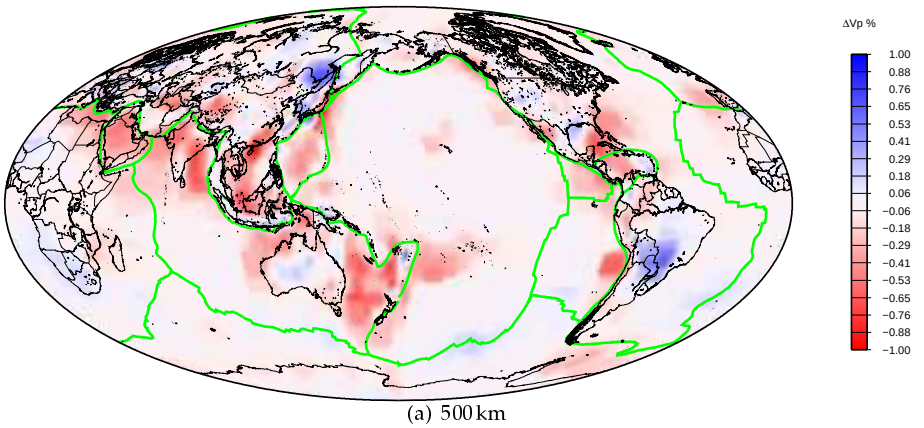
Ce chapitre développe un exemple complet d'application dans le domaine de la géophysique. L'application est une suite logicielle développée dans le cadre de la thèse de Marc Grunberg [8], entre 2001 et 2006, pour calculer des tomographies globales de la Terre. L'intention était de développer des codes performants (notamment en les parallélisant), portables (de façon à être capable d'utiliser des systèmes d'exploitation hétérogènes), et extensibles (garder une accélération des temps d'exécution jusqu'à des centaines de processeurs). L'objectif final, atteint en fin de projet, était de réaliser une tomographie sismique utilisant l'ensemble des données enregistrées par les réseaux de surveillance sismique internationaux depuis 1965, date de leur mise en place. Des tomographies utilisant un jeu de données important (sans toutefois utiliser l'ensemble des données disponibles) ont été réalisées précédemment : van der Hilst et al [16] et Sambridge et Faletic [14] sont deux études importantes ayant utilisé 550 000 rais dans des maillages à base de bloc réguliers pour la première, et de tétraèdres irréguliers pour la deuxième (maillage Delaunay-Voronoi). En ne conservant que les types d'ondes utilisés dans ces études, nous avons été capables de traiter rapidement plus de deux fois plus de rais (1 171 197 rais) et ainsi comparer les résultats produits par nos programmes à ces études de référence.

Nous avons également décidé de publier les programmes écrits sous une licence GPL. Cette démarche était très peu courante à l'époque dans le domaine (seul le logiciel TauP tooklkit [2] était aussi en GPL). La non-divulgation des sources empêche pourtant la vérification des codes utilisés, et oblige à ré-écrire des programmes existants par ailleurs. Au final, pour construire la suite logicielle présentée en figure 2.2, nous n'avons ré-utilisé que le code donnant le modèle de vitesse initial, et le solveur du système d'équations linéaires par moindres carrés [12]. Le but de ce travail était donc aussi d'offrir des codes vérifiables et ré-utilisables.

Le travail a été mené en collaboration avec l'Institut de Physique du Globe de Strasbourg, la thèse étant co-encadrée par Michel Granet, physicien, et Catherine Mongenet, professeur en informatique.

2.2 Introduction

En enregistrant à la surface de la Terre les mouvements du sol provoqués par les séismes, il est possible de recueillir des informations sur la structure interne de notre planète. L'enjeu essentiel de



(a) 500 km

FIG. 2.1 – Résultats tomographiques obtenus dans [8] à 500 km de profondeur.

la tomographie sismique consiste, à partir de ces enregistrements, à modéliser de façon réaliste les caractéristiques physiques des hétérogénéités de l'intérieur de la Terre.

L'énergie libérée par un séisme se propage dans toutes les directions sous forme d'ondes sismiques. Leur vitesse de propagation et leur amplitude sont modifiées par les structures géologiques qu'elles traversent. Les stations sismologiques enregistrent les signaux correspondant à ces ondes, appelés sismogrammes.

Les paramètres qui permettent de déduire les structures géologiques sont les vitesses de propagation des ondes sismiques. Tant qu'il n'y a pas de changement physico-chimique dans le milieu traversé par ces ondes, leur vitesse de propagation reste constante. C'est par une analyse des contrastes de vitesse que l'on arrive à déduire les différentes structures géologiques de la Terre, comme par exemple les limites des plaques tectoniques ou les points chauds dans le manteau.

La figure 2.1 illustre le résultat final d'une tomographie, que l'on peut visualiser sous la forme d'une carte des variations de vitesse par rapport à un modèle de vitesse initial. La couleur bleue (respectivement rouge) représente les zones où les ondes sismiques se propagent plus rapidement (respectivement plus lentement) que dans le modèle de vitesse initial.

Il existe principalement deux familles de méthodes de tomographie. La première famille dite à *temps de trajet* (travel-time tomography) utilise les temps de trajet des différentes ondes de volume, obtenus à partir des sismogrammes enregistrés. Elle tente de trouver un modèle de Terre dans lequel, en simulant la propagation de ces fronts d'onde par des *rais sismiques*[1], on aurait les mêmes temps de trajet que ceux observés. La deuxième famille (waveform tomography) consiste à trouver un modèle de Terre dans lequel un sismogramme synthétique serait semblable au véritable sismogramme enregistré. Dans cette dernière méthode le nombre d'inconnues est beaucoup plus important que pour la méthode à temps de trajet, et les équations reliant les inconnues du modèle aux données conduisent à la construction d'un système non linéaire. C'est pourquoi la tomographie à temps de trajet est la plus couramment utilisée. Dans ce travail nous utilisons également une méthode de cette famille intitulée *ray-shooting* ou *tracé de rais*.

Ce chapitre présente les principales étapes de la tomographie, telle que nous la proposons. Chaque étape est implémentée par un code différent. Dans ce travail, nous avons étudié l'oppor-

[1]Intuitivement le rai sismique correspond au trajet emprunté par l'onde, de la source du séisme au capteur sismologique.

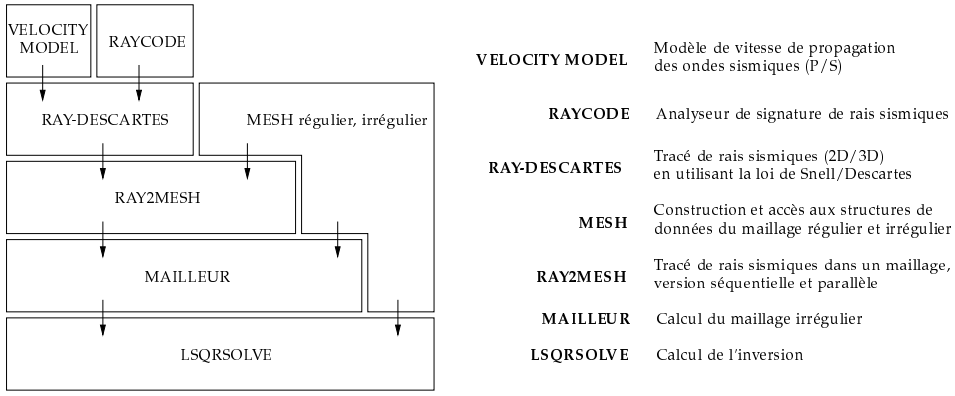


FIG. 2.2 – Les différents packages qui composent la suite logicielle. Chaque package se présente sous la forme d'une librairie, et peut dépendre d'un ou plusieurs autres packages.

tunité de paralléliser chacun des codes afin d'en diminuer le temps de traitement ou de pouvoir traiter des quantités d'information plus importantes. Les principales étapes sont :

Le tracé de rais
Le tracé des rais se fait d'abord dans un maillage dont les cellules ont des dimensions constantes en latitude et longitude pour une couche géologique donnée. La qualité de la solution tomographique dépend principalement de la géométrie des rais sismiques. On dit qu'une cellule est bien illuminée si elle est caractérisée par un bon échantillonnage 3D des rais en azimut et en incidence. Compte tenu d'une distribution non uniforme des séismes et des stations sismologiques, certaines régions à l'intérieur de la Terre sont richement *illuminées*, d'autres très pauvrement. Il en résulte, de ce fait, un biais du modèle tomographique.

Le raffinement du maillage
Pour améliorer le modèle, la deuxième étape de l'application de tomographie consiste à construire un maillage adaptatif de la Terre, à partir du maillage initial régulier enrichi des informations du tracé de rai. L'objectif est qu'une région n'ayant recueilli que peu de rais soit modélisée par de grandes cellules alors qu'inversement une région richement illuminée sera décrite par de nombreuses petites cellules. Cette étape a été parallélisée mais montre une extensibilité limitée.

Le problème inverse
Enfin, la troisième étape dite du problème inverse, consiste à modifier la vitesse de propagation associée à chaque cellule du maillage irrégulier, afin que pour chaque rai, le temps de propagation calculé soit le plus proche possible du temps de propagation observé. Il faut pour cela résoudre un système d'équations linéaires dont les inconnues sont les vitesses et les données sont les longueurs des segments de rais dans les cellules. Nous avons ré-utilisé et étendu ici un code séquentiel existant implémentant une méthode des moindres carrés.

Nous donnerons plus de détails sur la première étape, le tracé de rais, car c'est l'étape la plus coûteuse en temps de calcul, mais celle dont la parallélisation est la plus extensible en raison de la nature du problème. Ainsi, nous avons pu exécuter le code correspondant sur de nombreuses architectures parallèles, comprenant entre autres des grilles à l'échelle nationale.

2.3 ray2mesh

ray2mesh est le nom du code qui calcule le tracé de rai dans un maillage.

2.3.1 Le tracé de rai sismique dans un maillage

Les données fondamentales pour calculer une tomographie sont les sismogrammes collectés, lors de séismes, par différentes stations sismologiques couvrant la surface de la planète. Ces sismogrammes sont analysés pour déterminer les temps d'arrivée des différentes ondes sismiques, ainsi que pour calculer la localisation du foyer du séisme. Des organisations internationales, comme l'ISC (*International Seismic Center*) conservent l'ensemble de ces informations dans des bases de données qui peuvent contenir plusieurs millions de ces temps d'arrivée.

Une onde sismique est modélisée par un ensemble de rais représentant la propagation de son front d'onde du foyer du séisme vers les stations sismologiques. Le trajet d'un rai est constamment perpendiculaire au front d'onde et obéit aux lois de la réflexion/réfraction lorsqu'il atteint une interface géologique. Les changements qui se produisent lors de la propagation du rai constituent la *signature* du rai, codée sous forme de symboles qui permettent de reconstituer son histoire. Par exemple, un rai ayant une signature *PcS* signifie que l'onde de compression se propage jusqu'à l'interface manteau-noyau où elle est convertie en onde de cisaillement et est réfléchie vers la surface. Les sismogrammes, enregistrés par les stations sismologiques lors d'un tremblement de terre, sont analysés pour détecter les différents temps d'arrivée des ondes sismiques et déterminer leur signature.

Le calcul du trajet d'un rai nécessite la connaissance des coordonnées du foyer du séisme et de la station sismologique sur laquelle il a été détecté, de sa signature, ainsi que d'un modèle de vitesse. L'algorithme du tracé de rai repose sur la loi de Descartes en géométrie sphérique, qui décrit les variations de vitesses sismiques. Pour tracer le rai sismique, nous utilisons un modèle de vitesse couramment utilisé en géophysique, le modèle *ak135*, qui ne dépend que de la profondeur. Le tracé de rai est tout d'abord calculé en 2D dans un plan, puis replacé en 3D. La construction du rai en 2D est un processus itératif qui consiste à propager le rai pas à pas, en procédant soit par segments linéaires élémentaires, soit par décalages angulaires élémentaires, en fonction de l'angle d'incidence du rai. Typiquement un rai sismique est souvent discrétisé par plusieurs centaines, voire plusieurs milliers de points en fonction de sa longueur.

Le maillage initial d'une partie ou de la totalité de la Terre est obtenu en décomposant une sphère en un certain nombre de couches concentriques, de la surface jusqu'au centre. Chaque couche est ensuite décomposée, à partir du centre de la Terre, en secteurs angulaires à la fois en latitude et en longitude. Les volumes élémentaires ainsi créés peuvent être approximés par des hexaèdres et constituent les cellules du maillage. La figure 2.3 est une coupe de maillage illustrant le tracé des rais avec notamment un changement de direction à une des interfaces.

L'objectif final étant de construire un maillage adaptatif, le programme de tracé de rais calcule non seulement le trajet du rai, mais aussi certaines informations relatives à la distribution des rais dans les cellules du maillage initial (l'*illumination*).

2.3.2 Parallélisation de ray2mesh

La parallélisation repose sur la décomposition de l'ensemble des N rais à tracer en paquets de rais de taille égale, chaque paquet étant calculé en parallèle. Deux versions du programme parallèle sont disponibles. Pour des machines parallèles homogènes, il est convenable d'utiliser la première version, qui affecte un paquet unique, de taille N/p à chacun des p processus. Dans un cas plus général où les processeurs et/ou le réseau sont hétérogènes, une deuxième version de type *maître/esclave* est conseillée. Celle ci découpe en paquets plus petits (par défaut $N/10p$), de façon à ce que les processus plus rapides puissent traiter plus de paquets que les plus lents. Le maître distribue les paquets aux esclaves, qui à réception, calculent le tracé des rais qui leur ont été affectés. Le tracé est calculé par chaque esclave dans sa propre copie locale du maillage. Dès qu'un esclave a terminé son paquet, il demande un nouveau paquet au maître. Une fois tous les paquets distribués et calcu-

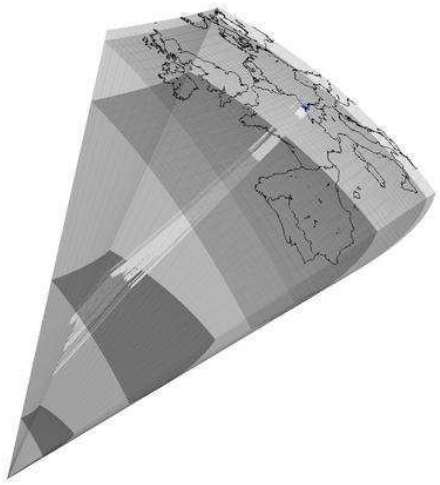


FIG. 2.3 – Tracé de rais arrivant sur l'Alsace. Le tracé est opéré dans un maillage

lés, toutes les instances des maillages locaux doivent être fusionnées pour donner un maillage final dont les cellules portent l'information extraites de tous les rais les ayant traversées.

La fusion est réalisée de la façon suivante. Tous les esclaves divisent de la même manière leurs maillages en p parties (un sous-ensemble des cellules), et chaque esclave devient responsable d'une partie distincte. L'esclave responsable d'une partie attend des autres esclaves qu'ils lui envoient les informations enregistrées dans la partie correspondante de leur propre maillage. A réception des informations, il les fusionne avec ses propres informations. Pour chaque partie dont il n'est pas responsable, l'esclave envoie les informations portées par les cellules de cette partie au processus responsable. Cette fusion implique donc une communication de type *all-to-all*. Dans les expériences présentées par la suite, le volume total de données échangés lors de cette communication est de l'ordre de la dizaine de gigaoctets.

2.3.3 Exploitation de ray2mesh

Équipement parallèle Après la conception et la réalisation, une étape complémentaire consiste à évaluer dans quelles conditions un code comme ray2mesh peut être exploité. Nous avons évalué les premières versions sur un jeu de données composé de tous les séismes enregistrés pour l'année 1999 (342056 rais tracés). Les équipements de calcul parallèle testés à l'époque (2002) étaient des SGI Origin[2] et un cluster dédié au laboratoire[3]. Les exécutions gardaient une efficacité intéressante jusqu'à 64 processeurs, le jeu de données semblant trop petit pour garder une efficacité raisonnable au delà. Ces deux équipements de calcul permettaient de tracer ce jeu de données dans un temps satisfaisant pour l'utilisateur (environ 10 minutes avec 12 processeurs).

Grille de calcul 2002-2003 Parallèlement, nous avions construit une grille de calcul expérimentale dans le cadre du projet d'ACI TAG. Elle consistait en une installation basée sur Globus [5] avec MPICH-G2 [4] comme bibliothèque de communication. Les équipements de calcul impliqués étaient très divers (PC, Sun 450 et 6800, SGI Origin 2000 et 3800) distribuées sur plusieurs sites (Stras-

[2] 32 et 768 processeurs Mips R14K, respectivement dans nos locaux et au centre de calcul CINES
[3] 6 noeuds bi-Pentium Xeon 1.7GHz, 1Go RAM / carte réseaux GE

bourg, Clermont-Ferrand, Montpellier) et reliés par le réseau Renater. La figure 2.4 montre les types d'interconnexions réseau longue distance disponibles à l'époque.

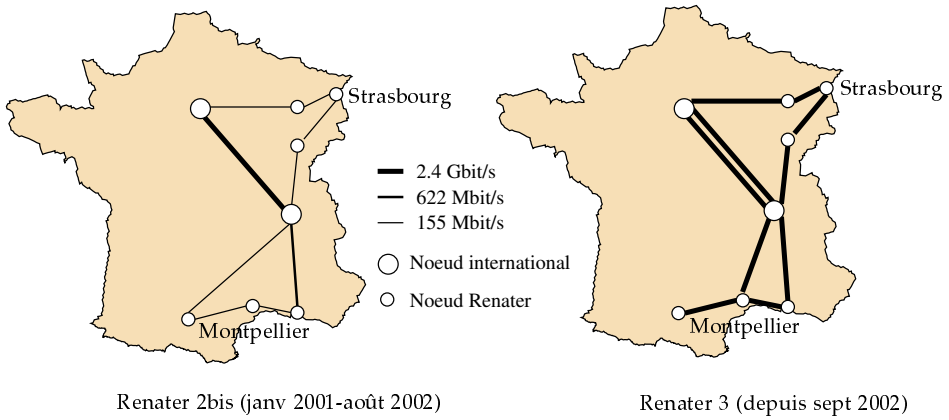


FIG. 2.4 – Partie de Renater empruntée et évolution des débits entre Renater 2bis et Renater 3

Il est légitime pour l'utilisateur de savoir si une telle infrastructure est utilisable pour son application. Deux critères sont primordiaux pour en juger :

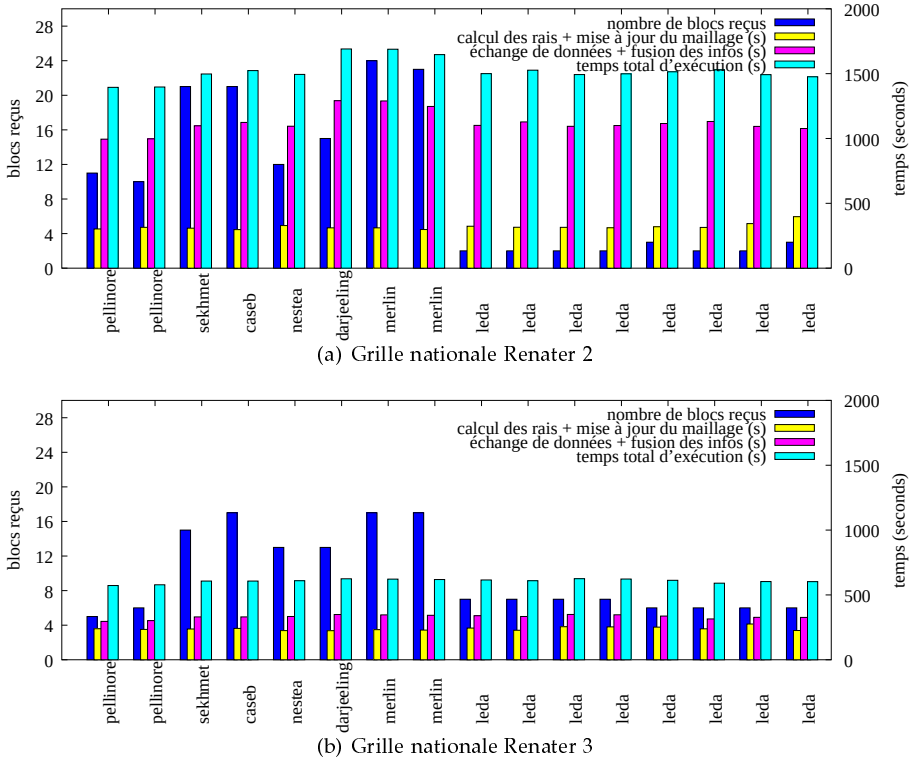
- la performance, qui peut être mesurée par le temps d'exécution du code, ou le temps de restitution (la durée entre la soumission du job et l'obtention des résultats), ou encore la taille du problème qu'on peut traiter,
- et la facilité d'utilisation. Même si elle difficilement quantifiable, le temps de préparation préalable à l'exécution en est un bon indicateur.

Nous nous sommes concentrés sur le seul critère de la performance mesurée par le temps d'exécution, car il est directement dépendant de la conception du programme vis-à-vis du support d'exécution sous-jacent. L'amélioration des autres critères nécessite des efforts dans d'autres domaines, en particulier celui des intergiciels de grille.

Des expériences ont été menées en août 2002, puis en mai 2003 avec le même jeu de données. Entre temps, le réseau Renater a évolué comme le montre la figure 2.4. Nous avons appris de ces expériences que le schéma maître/esclave donne de bons résultats à partir d'un certain seuil de performance du réseau, qui a été atteint dans notre contexte avec Renater 3 en 2003. Auparavant, il n'était pas envisageable d'exploiter des ordinateurs distants pour ce type de code. Ce phénomène est illustré par la figure 2.5, qui montre une exécution avec la première moitié des processeurs localisés sur le même site que le maître (Strasbourg), et l'autre moitié (leda) pris sur une machine parallèle SGI à Montpellier. Les processeurs distants sont tous identiques. A titre de comparaison, l'exécution utilisant des processeurs de la machine parallèle SGI dure 600s.

Dans le graphique du bas, utilisant Renater 3, on constate que les temps d'exécution sont courts, comparables à ceux relevés en n'utilisant que des processeurs de la machine parallèle SGI. Ceci nous amène à considérer l'équilibrage de charge satisfaisant (barres bleu foncé). En revanche, dans le graphique du haut, les processeurs distants sont clairement sous-chargés : ils reçoivent deux à trois fois moins de blocs à traiter, et au final les temps d'exécution sont trois fois supérieurs. Les demandes de travail ainsi que les blocs à traiter sont des petits messages, dépendant essentiellement de la latence, qui ne parviennent pas assez rapidement à destination, provoquant un déséquilibrage de charge. Il est aussi flagrant que le débit est insuffisant si on observe la fusion des données, avec

sa communication all-to-all : elle représente 75% du temps total de l'application (contre 50% avec Renater 3). Des tests de compression des données avant envoi avec la *zlib* n'ont rien donné.



(a) Grille nationale Renater 2

(b) Grille nationale Renater 3

FIG. 2.5 – Performances sur grille avec Renater 2 puis Renater 3

Grille de calcul 2006 La réalisation de la plate-forme expérimentale Grid'5000 [1] nous a permis trois ans plus tard de ré-évaluer la faisabilité du tracé de rai sur une architecture distribuée à large échelle géopgraphique. Les réseaux ont beaucoup évolué par rapport à l'expérience précédente : Grid'5000 utilise Renater 5, dont la plupart des liens entre sites ont une bande passante de 10 Gbps, et la latence a diminué d'un ordre de grandeur (environ 10ms pour 1000kms). Du point de vue géophysique, nous avons acquis entre temps le jeu données utilisé par Engdhal et al dans [3][4]. Ce jeu de données contient environ 100 000 séismes enregistrés par plus de 7 000 stations, et totalise 11,7 millions de phases sismiques et de temps de trajet.

Nous avons mené des expérimentations utilisant de 32 à 458 processeurs, en impliquant de un à cinq sites. Nous avons choisi sur Grid'5000, des processeurs aussi semblables que possible[5] pour minimiser les effets de l'hétérogénité des CPUs et mettre en avant l'impact du réseau sur l'exécution.

La table 2.1 résume ces résultats expérimentaux. Les première et deuxième colonnes donnent le nombre de processeurs et de sites impliqués, tandisque la troisième indique la répartition des processeurs sur les sites. Les trois colonnes suivantes sont les temps (s) moyens et l'écart-type passés

[4]La sismicité mondiale entre 1964 et 1995, enregistrée par l'*International Seismological Centre* et le *US Geological Survey's National Earthquake Information Center.*, corrigée par les auteurs (re-identification des phases sismiques, relocalisation des hypocentres,..).

[5]Les mêmes processeurs, seules leurs fréquences varient : Nancy, Rennes, Nice 2.0 GHz, Toulouse 2.2 GHz, et Orsay 2.4 GHz

par chaque processeur, dans la phase de tracé de rai, la phase de fusion et le temps total. L'avant dernière colonne donne le nombre moyen de rais calculés par processus et l'écart-type. La dernière colonne indique quel volume de données a été échangé entre les processus au total dans la phase de fusion (communication all-to-all).

sites	procs	site(procs)	calcul rai	all-to-all	total	nb rais	en transit	
1	32	nice*(32)	2651.62/86.79	344.85/61.94	3164.19	35351.60/1265.52	7.29 GB	1
	62	nancy*(62)	1316.91/44.94	93.13/11.92	1496.71	17965.60/992.46	9.02 GB	2
	62	nice*(62)	1349.09/45.48	98.78/12.65	1536.36	17965.60/1099.80	8.88 GB	3
	138	nice*(138)	647.12/21.72	37.39/3.32	729.54	8629.14/547.12	15.47 GB	4
2	64	nancy*(62) toulouse(2)	1271.91/43.17	90.89/11.45	1445.46	17395.30/972.66	9.26 GB	5
	128	nancy*(62) toulouse(66)	610.91/20.40	34.68/3.08	688.28	8629.14/573.01	15.29 GB	6
3	128	rennes(42) nancy*(44) toulouse(42)	620.27/21.21	33.56/2.98	699.57	8629.14/648.83	15.47 GB	7
	192	rennes(64) nancy*(64) toulouse(64)	412.16/13.70	30.84/2.23	474.65	5737.70/410.90	16.77 GB	8
5	458	rennes(152) nancy*(32) orsay(184) nice(58) toulouse(32)	177.07/5.69	31.43/1.47	227.53	2398.03/221.91	20.82 GB	9

TAB. 2.1 – Exécution du tracé de rais dans différentes configurations multi-sites. Le symbole * indique l'emplacement du jeu de données.

Par ailleurs, cette campagne d'expérience révèle plusieurs éléments remarquables :
- (lignes 2 ou 3 par rapport à ligne 5) : par rapport à l'exécution sur un seul site avec 62 processus, ajouter un processeur sur un site distant n'est pas handicapant.
- (ligne 7 par rapport à ligne 8) : le temps d'exécution pour 128 processus est quasiment identique, que les processeurs soient distribués sur deux ou trois sites (2% de différence).
- (ligne 8 par rapport à ligne 9) : l'exécution répartie sur trois sites, accélère d'un facteur 1,47 alors que le nombre de processus a augmenté dans une proportion de 1,5 entre 128 et 192 processus.

La figure 2.6 synthétise les accélérations obtenues, par rapport au temps de 32 processus sur un site (l'exécution séquentielle étant impossible avec les ordinateurs à notre disposition). Ces accélérations sont quasi-linéaires. Nous savons qu'une condition nécessaire à de bonnes performances est un bon équilibrage de charge, qui semble atteint ici étant donné l'écart type relativement faible à la moyenne du nombre de rais tracés.

Contrairement à nos attentes, les communications longue distance n'amènent pas un effondrement des performances, y compris quand le nombre de processus est important et plusieurs sites sont impliqués. Pourtant, plusieurs facteurs faisaient penser que l'extensibilité maximale serait vite atteinte. Le nombre de messages dans la communication all-to-all augmente de manière quadratique avec le nombre de processus, mais la quantité de données utile totale calculée reste constante (car elle correspond au même ensemble de rais tracés). En revanche, chaque cellule de maillage contient une entête constante et comme plus de cellules doivent être transmises lorsqu'il y a plus de processus, le volume de données en transit croît, comme on peut le voir dans la table 2.1. En plus du volume de données croissant, les messages de plus en plus petits font que la partie latence des communications devient proportionnellement plus importante.

On constate que les temps de communications du all-to-all décroissent linéairement jusqu'à 128 processeurs, quelque soit le nombre de sites (1, 2 ou 3). À partir de 128 processeurs, le temps de communication s'arrête presque de décroître pour atteindre un plancher d'environ 30 secondes. Alors que nous pensions que cette valeur serait un point d'inflexion, et que le temps de communication augmenterait à partir de ce point en raison de la multiplication des messages et du volume de données échangé croissant, ce temps reste constant jusqu'à 458 processeurs et 5 sites. Ceci reste bien sûr le facteur limitant à l'extensibilité de l'application, mais nous avons été surpris de ne pas la rencontrer avec un tel nombre de processeurs. L'une des raisons importantes à ces performances tient à la

techonologie des réseaux longue distance utilisés dans l'expérience. Des travaux ultérieurs en 2008, dans lequels j'ai essayé d'optimiser l'opération collective all-to-all de MPI entre plusieurs clusters distants, m'ont convaincu que la méthode la plus efficace consistait à envoyer tous les messages le plus tôt possible, quitte à provoquer des congestions. De nombreux autres algorithmes proposés visant à ordonnancer les messages pour éviter la congestion se sont avérés moins efficaces. Or, dans notre tracé de rais, nous avons justement implémenté cette phase d'échange par l'approche naïve consistant à faire envoyer par tous les processus, tous les messages directement aux autres.

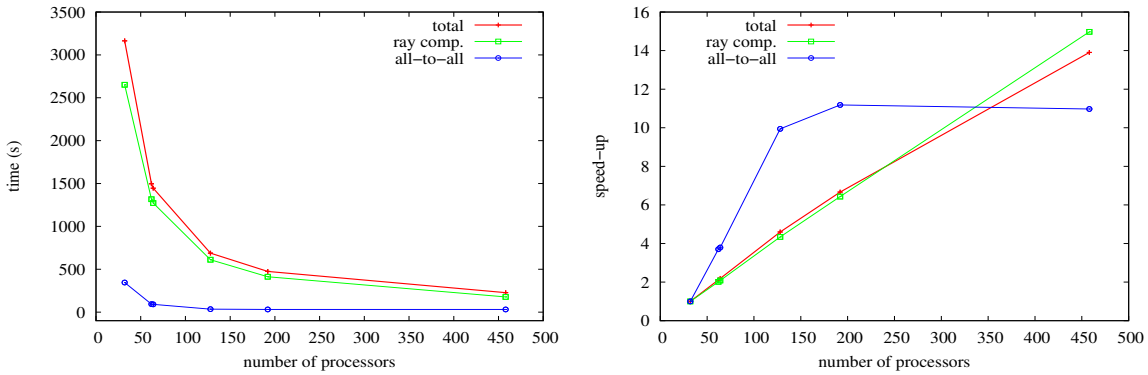


FIG. 2.6 – Temps d'exécutions et accélérations pour différentes configurations.

2.4 Maillage adaptatif

Une fois tous les rais tracés, chaque cellule du maillage a été traversée (ou pas) par un nombre variable de rais. Plus les rais sont nombreux et variés dans leurs angles d'incidences et leurs directions et couvrent bien le volume défini par la cellule, plus l'information apportée est riche. Les géophysiciens parlent de l'*illumination* de la cellule pour qualifier la richesse de cette information. Cependant, de par la répartition géographique très hétérogène des capteurs sismiques (peu dans les océans, 12% seulement des capteurs dans l'hémisphère sud), mais aussi de la concentration des séismes dans certaines zones, de nombreuses cellules recueillent peu ou pas d'information.

Pour améliorer la résolution tomographique, il est intéressant de n'inclure dans le système que des cellules ayant une bonne illumination. Nous avons donc proposé une méthode permettant de construire un maillage adaptatif, à partir du maillage initial et des données issues du tracé de rai. Il faut noter que, bien que le maillage initial soit 3D, il est composé de couches dont l'épaisseur est une cellule. La construction du maillage irrégulier se déroule donc couche par couche, les cellules étant dans un plan. Le principe consiste a agréger des cellules voisines qui n'ont pas atteint un seuil requis d'information si la cellule résultant de la fusion est mieux illuminée que les cellules initiales prises individuellement.

Bien sûr, plusieurs agrégations différentes sont souvent possibles. Pour déterminer quelle est la plus pertinente, nous définissons une fonction de distance à une illumination optimale visée, que nous appelons le *score* (une valeur réelle). Le score permet de définir une relation d'ordre entre les cellules agrégées. Nous limitons les agrégations possibles à des carrés ou rectangles. Il en résulte un maillage irrégulier comportant moins de cellules que dans le maillage initial, mais mieux illuminées. L'avantage est double : d'une part, il diminue de manière significative le nombre d'inconnues du système d'équations linéaires à résoudre dans l'inversion tomographique (chaque cellule représen-

tant une vitesse inconnue), et d'autre part, le système obtenu est plus fortement contraint, donnant un meilleur conditionnement et une meilleure stabilité des solutions obtenues [13].

2.4.1 Méthode de construction

Nous avons proposé dans [10] un algorithme glouton pour construire un tel maillage irrégulier. Il faut remarquer que la solution proposée doit tenir compte de contraintes liées à la géophysique. Ainsi, le maillage irrégulier final ne comporte aucun recouvrement des cellules mais ne doit pas nécessairement former un pavage : notre méthode laisse des zones sans rais non couvertes par des cellules, ce qui permet de ne pas générer des inconnues qui n'ont pas lieu d'être dans le système d'inversion tomographique à résoudre. Par ailleurs, une cellule agrégée ne peut dépasser une certaine taille, ni avoir une forme trop allongée, afin de préserver la qualité de la solution tomographique [15].

L'algorithme de construction du maillage procède en deux phases. La première phase est une phase de calcul où toutes les configurations possibles sont évaluées, c'est-à-dire qu'on forme pour chaque cellule initiale $C_{x,y}$, l'ensemble $\mathcal{M}_{x,y}$ des cellules agrégées telles que $C_{x,y}$ forme le coin supérieur gauche de $\mathcal{M}_{x,y}$ et qui respectent les contraintes imposées citées précédemment (rectangles de taille maximale, facteur de forme). La deuxième phase permet de sélectionner parmi ces cellules agrégées candidates, lequelles feront partie du maillage irrégulier final.

Dans la première phase, les ensembles de cellules agrégées candidates sont ordonnés en une liste triée, selon la relation d'ordre $\mathcal{M}_{x,y} \succ \mathcal{M}_{x',y'} \Leftrightarrow max(score(\mathcal{M}_{x,y})) > max(score(\mathcal{M}_{x',y'}))$. La phase de sélection qui suit commence par sélectionner de manière définitive comme première cellule du maillage irrégulier, la cellule agrégée ayant le meilleur score. Par définition, c'est celle qui a le meilleur score dans l'ensemble en tête de liste. Ce choix fait, l'algorithme parcourt la liste, et élimine les cellules candidates qui ont une intersection avec la cellule choisie (car il ne faut pas de recouvrement). Ensuite, on itère avec l'ensemble suivant dans la liste ordonnée. L'algorithme s'arrête quand tous les ensembles ont été examinés ou qu'ils ne contiennent plus de cellules agrégées candidates.

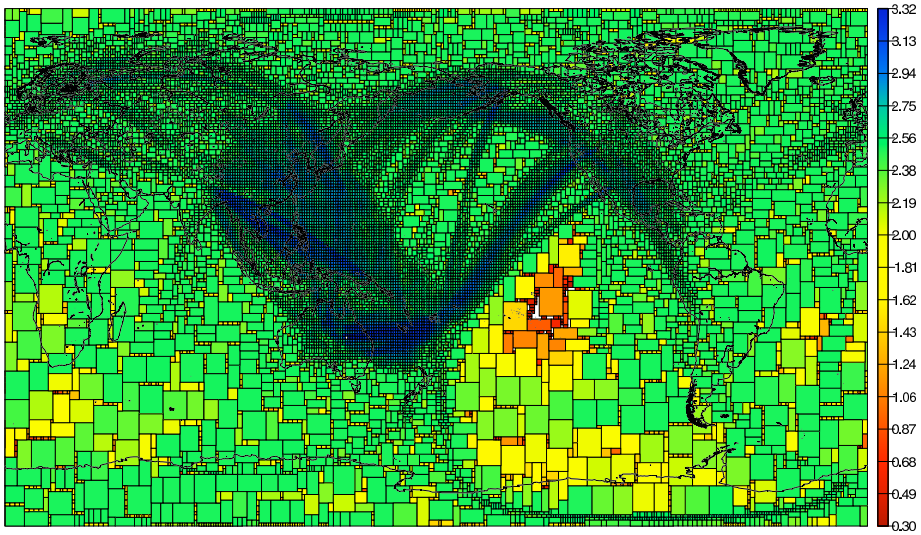


FIG. 2.7 – Maillage irrégulier d'une couche (17 827 cellules agrégées).

La figure 2.7 est une illustration d'un maillage irrégulier obtenu en choisissant un objectif classique, une densité de rai de 2.5 ($10^{2.5}$ rais par cellule). Typiquement, le maillage irrégulier a un nombre de cellules inférieur au maillage initial d'un ordre de grandeur. La qualité de la solution produite est mesurée par la distribution des scores des cellules. Dans l'exemple présenté, 40% des cellules ont le score optimal de 2.5 et 90% ont un score supérieur à 2, ce qui est jugé satisfaisant.

2.4.2 Parallélisation du mailleur adaptatif

La méthode de construction présentée ci-dessus nécessite beaucoup de mémoire pour charger l'ensemble des informations associées aux rais qui traversent les cellules du maillage. De plus, l'exploration exhaustive des configurations de cellules agrégées, ainsi que la détermination des scores associés, sont très consommatrices en ressources de calcul. La parallélisation doit améliorer ces deux aspects.

La parallélisation repose sur la subdivision d'une couche du maillage régulier initial en un ensemble de sous-domaines, chacun pris en charge par un processus. Dans chaque sous-domaine, le processus applique l'algorithme décrit dans la section précédente. Le maillage adaptif final est l'union des maillages produits par sous-domaines.

La difficulté réside dans le traitement des zones de jointures des sous-domaines. Un processus doit connaitre le bord des sous-domaines voisins, car des cellules agrégées créées sur ces bords pourraient empiéter sur son propre sous-domaine. Ces ensembles de cellules candidates contrôlées par un autre processus sont donc aussi intégrées dans la liste triée locale. Quand, lors du parcours de liste visant à éliminer les configurations incompatibles, il est rencontré une cellule agrégée dont l'origine est sur un sous-domaine voisin, le processus se met en attente d'une décision de la part du processus contrôlant cette cellule (communication bloquante). Symétriquement, le processus contrôlant la cellule envoie un message (communication non bloquante) au(x) processus concerné(s) dès qu'il a décidé de garder ou supprimer cette cellule agrégée.

Les tests de performances que nous avons menés sur la méthode parallélisée ont montré une faible extensibilité de la méthode, avec un tassement des accélérations entre 12 et 16 processeurs. Il semble que les attentes bloquantes utilisées pour gérer la cohérence des listes soient pénalisantes pour les temps d'exécution. Néanmoins, la parallélisation reste profitable dans la mesure où des maillages plus importants peuvent être traités sans limite de mémoire.

2.5 Conclusion

Le fait que chaque étape de la tomographie ne permette (ou ne nécessite) pas de paralléliser avec la même efficacité le calcul en question, implique que des moyens de calcul différents peuvent être employés à chaque étape. Nous avons vu qu'avec les technologies actuelles, le tracé de rais peut bénéficier de l'utilisation de ressources de calcul distribuées, y compris si elles sont placées sur des sites géographiquement distants (de l'ordre de 10 ms de latence). En revanche, le mailleur n'a pas la même extensibilité et il est judicieux de l'exécuter sur une seule machine parallèle. Pour la mise en œuvre de l'ensemble des traitements, une perspective de travail est donc l'intégration des différents codes de façon à simplifier l'enchaînement des traitements pour l'utilisateur, car cette tâche est particulièrement ardue quand les traitements ont eu lieu sur différents sites. Les recherches sur les *workflows* appliqués aux grilles de calcul apportent des solutions à ce type de problème. Nous évoquons nos perspectives en la matière dans le chapitre 7.

2.6 Résumé des contributions

L'objectif de ce travail inter-disciplinaire était de concevoir et développer des programmes pour réaliser une tomographie sismique globale de la Terre, en utilisant le maximum de données. Nous avons proposé un ensemble d'outils, dont les deux principaux sont le tracé de rais et l'outil de construction de maillages adaptatifs. Ces outils ont été développé sous une licence libre (GPL) et sont librement téléchargeables[6]. Ils ont été compilés et fonctionnent sur un grand nombre de systèmes UNIX (Linux, MacOsX, IRIX, AIX, SunOS).

L'extensibilité de la parallélisation du tracé de rais a été montrée sur machine parallèle homogène [11]. La grande portabilité du code a également permis de l'évaluer dans de nombreuses configurations, en particulier il a fait l'objet d'études de performances sur des grilles à l'échelle nationale, d'abord en 2002 puis 2003 [9, 6] avec Globus, et ensuite en 2006 lorsque nous avons pu disposer d'un grand nombre de processeurs [7] avec la plateforme Grid'5000.

Nous avons conçu une méthode de construction de maillage adaptatif adaptée aux exigences de la géophysique, et une parallélisation en a été faite [10]. Bien que moins extensible que le tracé de rai, la génération parallèle du maillage adaptatif nous a permis de dépasser les limites mémoires qui nous bloquaient sur une machine séquentielle. En fin de compte, nous avons pu réaliser une tomographie globale de la Terre utilisant le jeu de données complet (sismicité 1965-1995), dont les résultats sont présentés dans la thèse de Marc Grunberg [8].

Bibliographie

[1] Frank Cappello et al. Grid'5000 : A large scale, reconfigurable, controlable and monitorable grid platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing Grid'2005*, November 2005.

[2] H. Philip Crotwell, Thomas J. Owens, and Jeroen Ritsema. The taup toolkit : Flexible seismic travel-time and ray-path utilities. *Seismological Research Letters*, 70 :154–160, 1999.

[3] E. R. Engdahl, R. D. Van der Hilst, and R. P. Buland. Global teleseismic earthquake relocation with improved travel times and procedures for depth determination. *Bulletin of Seismol. Society Amer.*, 1(88) :pp–722–743, 1998.

[4] Ian Foster and Nicholas T. Karonis. A grid-enabled MPI : Message passing in heterogeneous distributed computing systems. In *Proc. of the 1998 ACM/IEEE conference on Supercomputing*, Nov. 1998.

[5] Ian Foster and Carl Kesselman. Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, 1997.

[6] Stéphane Genaud and Marc Grunberg. Calcul de rais en tomographie sismique : exploitation sur la grille. *Technique et Science Informatiques*, 24(5) :591–608, 2005. Numéro thématique RenPar'15.

[7] Stéphane Genaud, Marc Grunberg, and Catherine Mongenet. Experiments in running a scientific mpi application on grid'5000. In *4th High Performance Grid Computing International Workshop, IPDPS conference proceedings*. IEEE, March 2007. received the 'Intel Best Paper Award'.

[8] Marc Grunberg. *Conception d'une méthode de maillage 3D parallèle pour la construction d'un modèle de Terre réaliste par la tomographie sismique*. PhD thesis, Université Louis Pasteur, Strasbourg, September 2006.

[6] `http://renass.u-strasbg.fr/ray2mesh/`

[9] Marc Grunberg and Stéphane Genaud. Calcul de rais en tomographie sismique : exploitation sur la grille. In *Renpar2003*, pages 179–186, La Colle sur Loup, October 2003. INRIA.

[10] Marc Grunberg, Stéphane Genaud, and Catherine Mongenet. Parallel adaptive mesh coarsening for seismic tomography. In *SBAC-PAD 2004, 16th Symposium on Computer Architecture and High Performance Computing*. IEEE Computer Society Press, October 2004.

[11] Marc Grunberg, Stéphane Genaud, and Catherine Mongenet. Seismic ray-tracing and Earth mesh modeling on various parallel architectures. *The Journal of Supercomputing*, 29(1) :27–44, July 2004.

[12] C. Paige and M. Saunders. LSQR : An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8 :43–71, March 1982.

[13] Rowbotham Peter S. and Pratt R. Gerhard. Improved inversion through use of the null space. *Geophysics*, 62(3) :869–883, May 1997.

[14] Malcolm Sambridge and Rado Faletic. Adaptive whole earth tomography. *Geochem., Geophys., Geosyst*, 4(3), March 2003.

[15] Hrvoje Tkalcic, Barbara Romanowicz, and Nicolas Houy. Constraints on D" structure using PKP(AB-DF), PKP(BC-DF) and PcP-P traveltime data from broad-band records. *Geophys. J. Int.*, 148 :599–616, 2002.

[16] R. D. van der Hilst, S. Widiyantoro, and E. R. Engdahl. Evidence for deep mantle circulation from global tomography. *Nature*, 386 :578–584, 1997.

Chapitre 3

Equilibrage de charge statique

3.1 Contexte

Lorsque nous avons proposé le projet *Transformations et Adaptation pour la Grille*[1], nous avions comme objectif de comprendre quel type de modifications il fallait apporter au code source d'une application scientifique *existante* pour pouvoir l'exécuter dans des environnements distribués et hétérogènes. Les programmes parallèles utilisant MPI [19] nous semblaient les plus représentatifs du domaine, et c'est pourquoi nous nous sommes concentrés sur deux applications de ce type. Nous avons travaillé d'une part sur le code RAY2MESH décrit au chapitre précédent, et d'autre part sur le code Vador [10], qui modélise la physique des plasmas par une résolution des équations de Vlasov. Dans ces codes, on retrouve fréquement exprimée de manière implicite l'hypothèse que les processeurs et le réseau d'interconnexion sont homogènes. Par exemple, une parallélisation par décomposition de domaines imposera des domaines de tailles égales. Notre proposition de projet avait ciblé les primitives MPI utilisées de manière caractéristique pour distribuer une charge de travail. Par exemple, la primitive `MPI_Scatter` permet de découper un vecteur présent dans la mémoire d'un processus maître, en p parties de taille égale, et d'envoyer une partie distincte à chacun des p processus du communicateur. La question que nous nous posions était alors de savoir s'il était possible de localiser ces distributions dans le code source, de calculer *statiquement* un équilibrage de charge avant exécution, dépendant des processeurs utilisés, et de modifier la distribution originale par une distribution équilibrée. Pour reprendre l'exemple précédent, on peut substituer à `MPI_Scatter` une instruction `MPI_Scatterv`, qui permet de distribuer p parties de tailles inégales, c'est-à-dire réaliser une distribution paramétrée.

Dans ce chapitre, nous montrons nos résultats en matière d'équilibrage statique qui visent à améliorer le temps d'exécution. Nous nous intéressons à ce seul critère comme indicateur de la performance, car c'est ainsi qu'un utilisateur de cluster juge le plus fréquemment son programme (bien que sur un cluster surchargé, le temps de restitution peut devenir prépondérant). Il ne s'agit pas pour un programme exécuté dans un environnement hétérogène distribué, de rivaliser en général avec ce même programme exécuté sur un cluster. En revanche, les améliorations substantielles qu'on peut apporter à un programme pour l'adapter à un tel environnement (dont l'équilibrage de charge est un élément indispensable) peuvent l'amener à un niveau de performances acceptable. Notons que dans la notion de performance acceptable rentre très souvent aussi le coût économique réduit en raison de la mutualisation des matériels.

[1] Projet financé par l'Action Concertée Incitative (ACI) GRID, 2001.

Ici, l'objectif est de montrer un point particulier du travail de modélisation effectué dans le cadre du projet. Nous discutons des solutions au problème de l'équilibrage de charge, quand il peut être réalisé par une distribution adéquate des données. Ces données doivent être indépendantes, et on doit connaître le coût unitaire de traitement d'une donnée. Ce problème est très largement répandu, et l'approche qui consiste à calculer un équilibrage statique est utilisée dans des contextes variés. Cela va du partitionnement de données pour des traitements parallèles de la vidéo [2] à la recherche du nombre optimal de processeurs pour des algorithmes d'algèbre linéaire [3]. Elle est incontestablement plus difficile à mettre en œuvre qu'un équilibrage dynamique, mais elle fait économiser la surcharge liée aux multiples envois qui sont inhérents à un équilibrage de charge dynamique en plusieurs tournées.

3.2 Introduction

Dans le chaptire précédent, nous avons décrit la parallélisation du tracé de rais (section 2.3). Cet exemple est caractéristique d'un schéma de distribution *maître-esclave* : un maître possède un ensemble de données et un calcul peut être fait sur chaque donnée de manière indépendante. Dans un contexte hétérogène, l'objectif est de distribuer à chaque esclave un nombre de données adéquat avec sa puissance de façon à équilibrer la charge, et ainsi minimiser le temps d'exécution. Remarquons que dans l'exemple du tracé de rais, ce nombre est ajusté *dynamiquement* : le maître distribue les données par blocs, donc en plusieurs tournées, à chaque fois qu'un esclave réclame du travail. Nous nous intéressons ici au cas particulier dans lequel il n'y a qu'une seule tournée, car on détermine à l'avance le nombre de données, si possible optimal, à envoyer à chaque esclave pour toute l'exécution.

Nous modélisons ce schéma de distribution à partir du comportement observé dans nos expériences. La distribution, faite avec `MPI_Scatter`, se traduit par un schéma de distribution tel que l'illustre la figure 3.1.

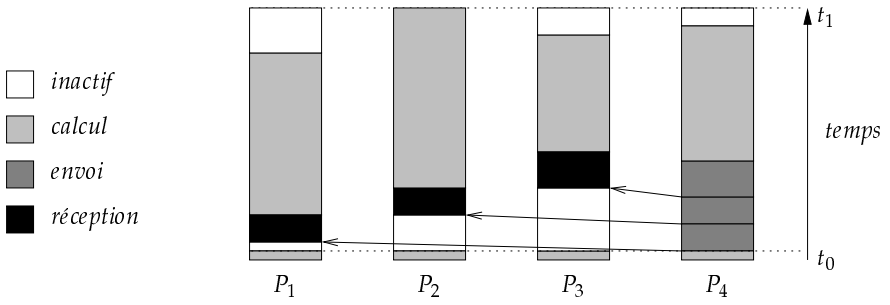


FIG. 3.1 – Une opération de distribution suivie d'une phase de calcul.

Tous les esclaves se mettent en attente bloquante des donnnées, et le maître (P_4 dans l'exemple) envoie les données tour à tour à chacun des esclaves. Ce comportement s'explique par l'implémentation de MPICH-G2 utilisée. Les auteurs de cette bibliothèque ont montré que quand les communications avaient une forte latence, il est préférable de procéder à une communication de type arbre plat [16], au lieu d'un arbre binomial, habituellement utilisé pour optimiser une communication collective comme celle-ci dans un réseau à faible latence. Ce résultat avait été aussi montré précédemment pour MagPIe [17].

Notons que les communications bloquantes interdisent un éventuel recouvrement entre calculs et communications : un processeur doit avoir entièrement terminé ses communications avant de

pouvoir commencer ses calculs. Cette hypothèse correspond à la sémantique de `MPI_Scatter`, dont on ne peut sortir que quand les données ont été envoyées et reçues. Enfin, on note que le processeur qui possède les données (le maître) prend aussi part au calcul après avoir distribué les données.

3.3 Calcul de la distribution des données

3.3.1 Formulation du problème

Il s'agit de traiter n données sur p processeurs, appelés $P_1, \ldots, P_p$. Nous cherchons une distribution $n_1, \ldots, n_p$ de ces données sur ces processeurs de façon a minimiser le temps total d'exécution. Nous faisons l'hypothèse que nous connaissons :

- $T_{comm}(i, x)$, le temps nécessaire à P_i pour recevoir x données du processeur maître ;
- $T_{calc}(i, x)$, le temps nécessaire à P_i pour traiter x données.

Concernant les capacités de communication, nous faisons l'hypothèse qu'à un instant donné, un processeur peut être impliqué dans au plus deux communications : une émission et une réception (modèle un-port). Le maître envoie donc ses données successivement aux processeurs, à P_1, puis à P_2, et ainsi de suite, jusqu'à P_p. Comme le processeur maître ne peut commencer à traiter son lot de données *qu'après* avoir envoyé les autres données à tous les autres processeurs (car il n'y a pas de recouvrement entre les calculs et les communications), il sera toujours le dernier processeur : P_p. Pour évaluer le temps d'exécution d'un processeur P_i dans ces conditions, on établit qu'il ne peut commencer à recevoir ses données qu'après que les processeurs $P_1, \ldots, P_{i-1}$ aient été servis, ce qui prend un temps $\sum_{j=i}^{i-1} T_{comm}(j, n_j)$, puis que les données envoyées par le maître lui arrivent, ce qui prend $T_{comm}(i, n_i)$. Enfin, P_i traite ses données en $T_{calc}(i, n_i)$. Ainsi, P_i termine son exécution au temps :

$$T_i = \sum_{j=1}^{i} T_{comm}(j, n_j) + T_{calc}(i, n_i). \tag{3.1}$$

Le temps T, nécessaire pour traiter complètement l'ensemble des n données est donc :

$$T = \max_{1 \leq i \leq p} T_i = \max_{1 \leq i \leq p} \left(\sum_{j=1}^{i} T_{comm}(j, n_j) + T_{calc}(i, n_i) \right), \tag{3.2}$$

et nous cherchons une distribution $n_1, \ldots, n_p$ minimisant cette durée.

3.3.2 Méthodes de résolution

Nous avons étudié différentes méthodes pour résoudre le problème.

La première approche est très générale et suppose uniquement que les fonctions de coûts sont positives ou nulles. Elle consiste en un algorithme de programmation dynamique qui donne une solution optimale, dont la complexité est en $O(n^2 p)$. Malgré des optimisations (on suppose que les fonctions de coûts sont croissantes avec les données) qui peuvent ramener à une complexité dans le meilleur cas de $O(n \log(n) p)$, le temps de calcul de la solution reste prohibitif. Sur l'exemple du tracé de rai, pour la sismicité de l'année 1999 qui correspond à environ 800 000 données, le calcul optimisé de la distribution optimale pour 16 processeurs prenait six minutes.

La deuxième solution simplifie le problème en faisant l'hypothèse que les fonctions de coûts sont affines. Le problème revient alors à résoudre le programme linéaire :

$$\begin{cases} \text{Minimiser } T \text{ tel que} \\ \forall i \in [1;p],\ n_i \geq 0, \\ \sum_{i=1}^{p} n_i = n, \\ \forall i \in [1;p],\ T \geq \sum_{j=1}^{i} T_{comm}(j,n_j) + T_{calc}(i,n_i). \end{cases} \tag{3.3}$$

Cependant, trouver une solution en nombres entiers à ce problème est très coûteux en temps. Pour contourner le problème, nous avons montré qu'une résolution en rationnels, suivi d'un arrondi pour obtenir une solution en entiers est intéressante car d'une part, le temps de calcul est très court, et d'autre part on peut borner le résultat trouvé T par rapport à l'optimal T_{opt} :

$$T_{opt} \leq T \leq T_{opt} + \sum_{j=1}^{p} T_{comm}(j,1) + \max_{1 \leq i \leq p} T_{calc}(i,1).$$

La dernière approche simplifie encore le problème en supposant des fonctions de coûts linéaires, c'est-à-dire qu'il existe des constantes $\lambda_i, \mu_i \geq 0$ telles que $T_{comm}(i,n) = \lambda_i n$ et $T_{calc}(i,n) = \mu_i n$. On se retrouve alors dans le cadre des tâches divisibles, connu sous le nom de *divisible load theory* (DLT) [8]. Une tâche divisible est une tâche qui peut être divisée et répartie sur un nombre quelconque de processeurs. Initialement développé pour des réseaux de capteurs, les résultats produits dans ce cadre de travail ont été publiés au début dans des journaux comme IEEE *Transactions on Aerospace and Electonic Systems*, et ce n'est que quelques années plus tard qu'on s'est rendu compte que les résultats étaient assez généraux pour modéliser des systèmes parallèles ou distribués. La DLT était déjà riche de nombreux résultats d'ordonnancements pour différentes topologies de réseaux. Ainsi, la topologie correspondante à notre problème est un *single-level tree network without front-end processors* [7] dans la DLT. Ce cadre permet de donner une expression analytique du problème posé. D'abord, on montre que le temps d'exécution optimal est obtenu quand tous les processeurs qui participent au calcul terminent en même temps. Ce résultat a été montré initialement par Robertazzi [20, 7]. L'argument est que si un processeur terminait après les autres, l'excès de charge qui provoque ce temps supplémentaire pourrait être réparti sur les autres processeurs, et on pourrait ainsi terminer plus tôt. Ensuite, en utilisant l'égalité des temps de fin sur chaque processeur, on déduit que la durée de l'exécution est

$$t = \frac{n}{\sum_{i=1}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j}} \tag{3.4}$$

et le processeur P_i reçoit

$$n_i = \frac{1}{\lambda_i + \mu_i} \cdot \left(\prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j} \right) \cdot t \tag{3.5}$$

données à traiter.

3.4 Ordre des communications

Dans le problème exposé ci-dessus, on supposait que l'ordre dans lequel le maître envoie aux processeurs est fixé (par exemple le code de la primitive `MPI_Scatter` envoie dans une boucle allant du processus de rang 1 à p). On peut donc s'interroger sur l'influence qu'aurait une permutation dans l'ordre des envois.

L'ordre a effectivement une influence, et un résultat que nous avons obtenu dans le cadre des tâches divisibles est que le temps d'exécution le plus court est obtenu en effectuant les envois vers

les processeurs dans l'ordre décroissant de leurs bandes passantes avec le maître. Les vitesses de calcul des processeurs n'ont donc pas d'influence.

3.5 Variantes du problème

Des résultats similaires ont été trouvés pour des variantes de notre problème. La variante où le processeur maître est capable de calculer pendant l'envoi est appelé *single-level tree network with front-end processors* dans la DLT. Pour ce problème, il a été montré que trouver une solution optimale implique que tous les processeurs participent et terminent en même temps, et que les envois se fassent par bandes passantes décroissantes [18, 5, 6]. Une autre variante est celle où le maître n'est pas contraint à un seul envoi par esclave mais peut faire plusieurs tournées. Dans le cadre de fonctions de coûts linéaires, Beaumont, Legrand et Robert ont proposé un algorithme glouton pour déterminer la distribution qui maximize le nombre de données traitées en un temps donné [4]. Enfin, dans le cas où les esclaves renvoient des résultats au maître, un algorithme de programmation dynamique a aussi été proposé [1].

3.6 Évaluation

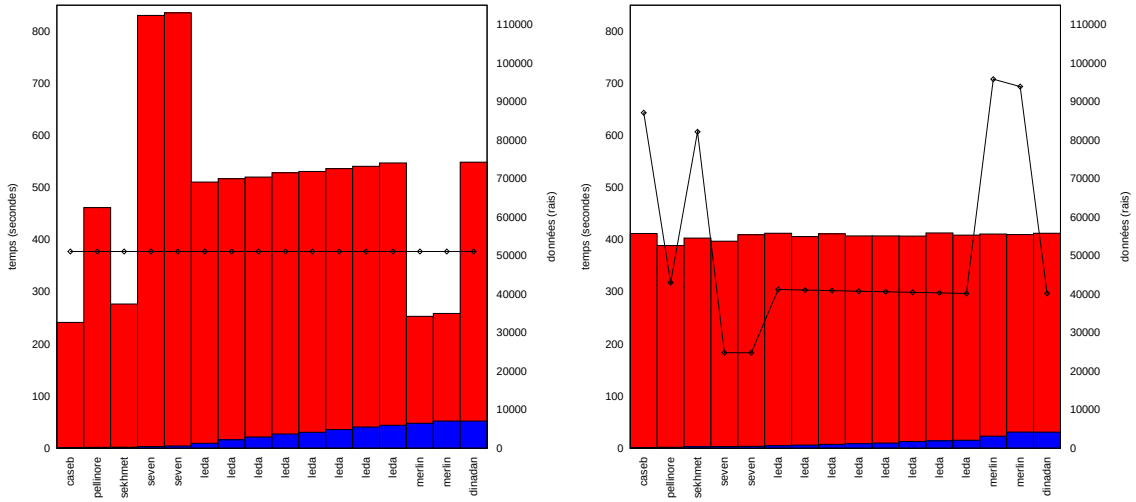
Nous avons évalué ces résultats théoriques sur le code de tracé de rai. La plateforme expérimentale est la grille dont nous avons parlé en section 2.3.3, p. 19. Les ressources de calcul (16 processeurs) utilisées dans l'expérience sont comme précédemment prises sur deux sites : principalement des PC à Strasbourg où se trouve le processus maître possédant les données, et huit processeurs d'une SGI Origin à Montpellier (leda). Concernant les données, le jeu le plus gros dont nous disposions à l'époque comportait environ 800 000 données. Dans un premier temps, nous avons exécuté l'application sans tenir compte de l'hétérogénéité des processeurs : ils reçoivent tous une quantité égale de données. En revanche, en ce qui concerne l'ordre des processeurs, nous choisissons le cas le plus favorable en les ordonnant par bandes passantes décroissantes. Une exécution dans ces conditions est celle de la figure 3.2(a). Les barres du haut (rouge) indiquent le temps passé à calculer, les barres du bas (bleu) le temps d'attente ou de communication. Le déséquilibre dû à la différence de puissance des processeurs est important, le dernier calcul terminant après 835s.

Une distribution optimale est ensuite calculée en rationnels puis arrondie. En gardant l'ordre d'envoi par bandes passantes décroissantes, nous obtenons l'exécution de la figure 3.2(b). La différence entre les temps de fin est maintenant de 6% de la durée totale (les calculs prennent fin entre 388s et 421s). Pour vérifier l'influence de l'ordre d'envoi, l'exécution de la figure 3.3 met en œuvre l'ordre inverse.

Le résultat donne effectivement un moins bon équilibrage, les temps variant de 12% (fins entre 420s et 469s). L'essentiel de la différence vient du temps passé par les processeurs à attendre le début effectif des communications.

3.7 Conclusion

Les résultats obtenus dans l'expérience réelle sont bons, les temps de fins étant sous-estimés d'environ 2% seulement, mais pas optimaux. Deux raisons expliquent la distortion entre le résultat prévu et le résultat observé. La première est que les coûts des calculs et des communications ne sont que des prédictions. Dans l'expérience, l'erreur de prédiction s'est révelée plutôt faible pour les communications locales (8% du temps total des communications), tandis que l'erreur sur les



(a) Distribution uniforme (b) Equilibrée, BP décroissantes

FIG. 3.2 – Exécutions uniformes et équilibrées

communications longues distances a atteint ici 79%. Nous avions remarqué par ailleurs une variabilité importante et récurrente du débit sur ces communications à l'époque. En dépit des erreurs de prédictions, il reste souvent possible de suivre la règle qui consiste à ordonner les envois par ordre décroissant des bandes passantes, ce qui apporte un gain significatif.

La deuxième raison concerne l'évaluation du coût du calcul. D'abord, il faut noter que nous avons simplifié le problème en considérant que le calcul d'une donnée prenait le même temps, quelle que soit la donnée. Ceci est vrai en moyenne dans notre cas, car le nombre de rai est suffisament grand. Si le nombre de données était faible, il faudrait reconsidérer la méthode. Ensuite, il existe des variations dans la charge des ressources de calcul. Au total, l'erreur relative est inférieure à 2% sur les processeurs d'Origin alors qu'elle pouvait parfois atteindre 9% sur les PC. La différence ici est dûe au fait que les processeurs des PC pouvaient être partagés avec d'autres utilisateurs alors qu'un système de réservation nous permettait un usage exclusif des processeurs des machines parallèles.

En conclusion, l'approche d'équilibrage statique de la charge présentée ici est envisageable dans notre contexte de programmes parallèles pour grilles, mais difficile à mettre en œuvre. Il faut être capable, non seulement d'insérer la phase de calcul avant l'exécution, mais aussi de paramétrer le calcul avec des valeurs pertinentes des coûts de communication et de calcul, qu'il convient donc d'acquérir dynamiquement. Cette acquisition peut se faire par exemple avec un outil de monitoring comme NWS [21], mais l'expérience a montré la difficulté d'installer et de maintenir un tel système de sondes. De plus, cette approche est mise en défaut si des variations importantes et inattendues de la charge apparaissent. Au vu de ces difficultés dans notre contexte, l'équilibrage dynamique en plusieurs tournées apparaît compétitif et bien plus simple à mettre en œuvre. Bien sûr, l'équilibrage dynamique nécessite de ré-écrire la partie de code gérant la distribution des données. Le code de gestion de l'équilibrage peut ne pas être séparé proprement du code applicatif (par exemple en MPI), mais cet inconvénient peut être applani en recourant à une librairie come MW [14] qui facilite l'écriture d'applications maître-esclaves.

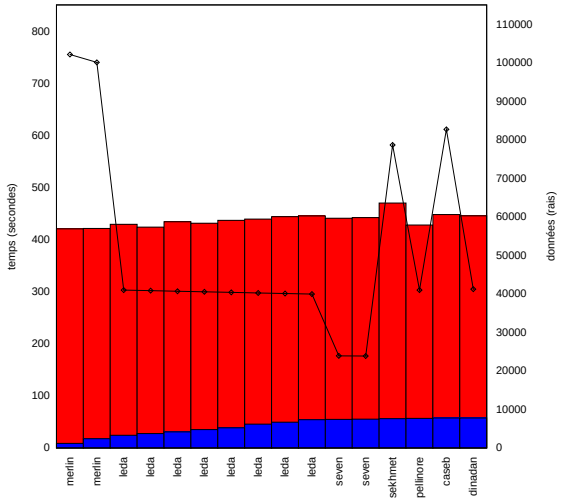


FIG. 3.3 – Exécution équilibrée, BP croissantes

3.8 Résumé des contributions

Parallèlement aux développements des codes parallèles scientifiques, nous avons avons identifié les éléments qu'il fallait améliorer impérativement pour obtenir des performances acceptables en environnement hétérogène. Nous avons étudié pour cela des techniques d'équilibrage de charge, et proposé pour deux techniques différentes, en fonction de la granularité des données. Dans le cas d'une granularité fine (c'est l'exemple présenté dans ce chapitre), nous avons discuté le calcul d'une distribution statique des données. Nous avons d'abord proposé une approximation en rationnels [9]. Puis l'étude à été approfondie dans une partie de la thèse d'Arnaud Giersch [13], où une solution optimale avec des fonctions très générales de coûts est proposée, ainsi qu'un théorème sur l'ordre optimal d'envoi des données selon les bandes passantes décroissantes [11, 12].

Bibliographie

[1] Francisco Almeida, Daniel González, and Luz Marina Moreno. The master-slave paradigm on heterogeneous systems : A dynamic programming approach for the optimal mapping. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004)*, pages 266–272, A Coruña, February 2004. IEEE.

[2] Deniz Turgay Altilar and Yakup Paker. Optimal scheduling algorithms for communication constrained parallel processing. In *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference*, volume 2400 of *LNCS series*, pages 197–206, Paderborn, August 2002. Springer.

[3] Jorge G. Barbosa, João Tavares, and Armando Jorge Padilha. Linear algebra algorithms in heterogeneous cluster of personal computers. In *9th Heterogeneous Computing Workshop (HCW'2000)*, pages 147–159, Cancun, May 2000. IEEE.

[4] Olivier Beaumont, Arnaud Legrand, and Yves Robert. A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs. In *17th International Symposium on Computer and Information Sciences (ISCIS XVII)*, pages 115–119, Orlando, October 2002. CRC Press.

[5] Olivier Beaumont, Arnaud Legrand, and Yves Robert. Optimal algorithms for scheduling divisible workloads on heterogeneous systems. In HCW 2003 [15], page 98b. in conjunction with IPDPS.

[6] Olivier Beaumont, Arnaud Legrand, and Yves Robert. Scheduling divisible workloads on heterogeneous platforms. *Journal of Paralell Computing*, 29(9) :1121–1152, September 2003.

[7] Veeravalli Bharadwaj, Debasish Ghose, Venkataraman Mani, and Thomas G. Robertazzi. *Scheduling divisible loads in parallel and distributed systems*. Wiley, 1996.

[8] Veeravalli Bharadwaj, Debasish Ghose, and Thomas G. Robertazzi. Divisible load theory : A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1) :7–17, January 2003.

[9] Romaric David, Stéphane Genaud, Arnaud Giersch, Benjamin Schwarz, and Éric Violard. Source code transformations strategies to load-balance grid applications. In *3rd International Workshop on Grid Computing (GRID 2002)*, volume 2536 of *LNCS series*, pages 82–87, Baltimore, November 2002. Springer.

[10] Francis Filbet, Eric Sonnendrücker, and Pierre Bertrand. Conservative numerical schemes for the Vlasov equation. *Journal of Comput. Phys.*, 172 :166–187, 2001.

[11] Stéphane Genaud, Arnaud Giersch, and Frédéric Vivien. Load-balancing scatter operations for grid computing. In HCW 2003 [15], page 101a. in conjunction with IPDPS.

[12] Stéphane Genaud, Arnaud Giersch, and Frédéric Vivien. Load-balancing scatter operations for grid computing. *Journal of Paralell Computing*, 30(8) :923–946, August 2004.

[13] Arnaud Giersch. *Ordonnancement sur plates-formes hétérogènes de tâches partageant des données*. Thèse de doctorat, Université Louis Pasteur, Strasbourg, France, December 2004.

[14] Jean-Pierre Goux, Sanjeev R. Kulkarni, Jeff Linderoth, and Michael Yoder. An enabling framework for master-worker applications on the computational Grid. In *9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 43–50, Pittsburgh, August 2000. IEEE.

[15] *12th Heterogeneous Computing Workshop (HCW'2003)*, Nice, April 2003. IEEE. in conjunction with IPDPS.

[16] Nicholas T. Karonis, Bronis R. de Supinski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 377–384, Cancun, May 2000. IEEE.

[17] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe : MPI's collective communication operations for clustered wide area systems. *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 34(8) :131–140, August 1999.

[18] Hyoung Joong Kim, Gyu-In Jee, and Jang Gyu Lee. Optimal load distribution for tree network processors. *IEEE Transactions on Aerospace and Electonic Systems*, 32(2) :607–612, April 1996.

[19] MPI Forum. MPI : A message passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, June 1995.

[20] Thomas G. Robertazzi. Processor equivalence for a linear daisy chain of load sharing processors. *IEEE Transactions on Aerospace and Electonic Systems*, 29(4) :1216–1221, October 1993.

[21] Richard Wolski, Neil T. Spring, and Jim Hayes. The network weather service : a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6) :757–768, October 1999.

Deuxième partie

P2P-MPI

Chapitre 4

P2P-MPI

Dans ce chapitre, nous expliquons les motivations et les idées à l'origine de P2P-MPI, commencé en 2004 avec le DEA de Choopan Rattanapoka. Les chapitres suivants 5 et 6 sont des illustrations des applications que nous avons developpées et mises en œuvre en utilisant ce logiciel.

4.1 Contexte

Dans les chapitres précédents, nous avons évoqué des codes scientifiques, écrits en langage C, C++ ou Fortan. L'universalité de ces langages de programmation était à priori un gage de portabilité, qui aurait dû faciliter leur déploiements dans des environnements hétérogènes du point de vue logiciel. De surcroît, pour mener à bien notre évaluation, nous avions choisi le logiciel Globus toolkit [24], réputé le plus mature et ayant incontestablement bénéficié d'un effort massif de développement. Dans cet environnement, nous pouvons disposer immédiatement d'une implémentation MPI fonctionnelle, la bibliothèque MPICH-G2 [41], basée sur l'implémentation MPICH [36].

Dans la réalité, sur notre grille test décrite en section 2.3.3, p. 19, constituée soit d'ordinateurs individuels partagés, soit de machines parallèles sous le contrôle d'un gestionnaire de batchs (LSF), la réalisation des expériences s'est souvent heurtée à des obstacles pratiques d'importance. D'une part, l'hétérogénéité des systèmes provoque de multiples complications qui augmentent très fortement le risque qu'une exécution d'un programme MPI finisse en erreur. A titre d'andecdote, nous avons découvert que nous étions les premiers à tenter d'exécuter un programme MPICH-G2 utilisant simultanément des processus tournant sur Linux et un autre système Unix : la communication entre ces différents systèmes était en effet impossible à cause d'un bug[1], pour lequel nous avons fourni un correctif.

D'autre part, l'absence de meta-scheduler capable de prendre en charge les ressources appartenant à plusieurs domaines administratifs limite drastiquement le nombre de machines qu'on peut utiliser. En effet, dans un programme MPI, les processus doivent démarrer simultanément. Or, chaque machine contrôlée par un gestionnaire de batchs démarre les processus mis en file d'attente après un délai arbitrairement long, et l'exécution a une durée limitée. Plus le nombre de machines impliquées est grand, moins on a de chances d'obtenir une plage de temps d'exécution commune aux différents machines. Ceci explique le faible nombre de machines utilisées dans les expériences avant l'utilisation de Grid'5000.

Ces difficultés m'ont amené à repenser les besoins des utilisateurs qui veulent exécuter des programmes parallèles de ce type. J'ai proposé en septembre 2003 un stage de master recherche dont

[1] `http://www-unix.globus.org/mail_archive/mpich-g/2002/02/msg00004.html`

l'objectif était de concevoir un environnement logiciel permettant d'exécuter des programmes parallèles dans le contexte où les ressources sont hétérogènes et plutôt volatiles. Pour ces raisons, le cahier des charges comprenait la découverte automatique de ressources, la tolérance aux pannes, et la virtualisation (indépendance par rapport à l'OS). Avec Choopan Rattanapoka, nous avons proposé à l'issue de son stage de master, une solution répondant à ces contraintes, et dans laquelle beaucoup des difficultés évoquées précédemment disparaissent. Cet environnement est à la fois une bibliothèque de communication avec laquelle on peut développer des programmes de type MPI et un intergiciel qui gère les ressources de calcul. Le logiciel, baptisé P2P-MPI[2], a été mis à disposition publiquement sous une licence GPL dès 2005. Nous avons ensuite pu l'utiliser dans des collaborations avec des collègues pour l'évaluer concrètement.

4.2 Introduction

Pour poursuivre notre objectif initial qui est l'évaluation du comportement des programmes parallèles dans des environnements distribués hétérogènes, nous n'avions donc pas besoin de toutes les fonctionnalités offertes par un toolkit comme Globus. Par exemple, la gestion de la sécurité n'est pas dans notre champ d'intérêts immédiats.

Avec P2P-MPI, nous nous concentrons sur des grilles qui peuvent être constituées au sein de communautés dont les membres sont bien identifiés. Concrètement, les ressources de calcul peuvent être prises parmi les machines d'un laboratoire, et aller jusqu'à une communauté de type *campus grid*. Dans cette hypothèse, la plupart des ressources sont des machines personnelles. Nous pensons qu'une approche de type *volunteer computing* peut permettre de mettre à disposition un nombre suffisant de ressources correspondant à beaucoup d'applications parallèles. Des projets comme BOINC [3] ont démontré le succès que pouvait rencontrer une telle approche. Pour cela, il faut que l'utilisateur puisse s'« approprier » le logiciel, ce qui implique au minimum qu'il puisse être autonome pour l'utiliser. P2P-MPI repose sur deux idées clés qui ont orienté tous nos choix de conception. On souhaite :

i) un fonctionnement en mode utilisateur, et
ii) un minimum de tâches de configuration.

Le point i) implique que tout utilisateur d'une machine peut installer et utiliser le logiciel sur cette machine. Ne pas avoir besoin de droit d'administration système facilite l'installation, l'administration et la maintenance du logiciel. Le point ii) amène à éliminer les traditionnels fichiers de configuration lorsqu'il s'agit de mémoriser des informations de nature dynamique.

Pour répondre à ces contraintes, nous avons choisi pour le développement du projet, les orientations suivantes.

- **Virtualisation** : tout l'environnement est écrit en Java et les programmes parallèles doivent aussi être écrits en Java. Ce choix simplifie considérablement les problèmes liés à l'hétérogénéité des systèmes d'exploitation. La virtualisation, qu'elle soit au niveau du système ou de l'exécution du code, est une approche de plus en plus utilisée aujourd'hui.
- **Pair-à-pair** : le système d'information gérant les ressources est organisé en pair-à-pair (P2P). Un utilisateur démarrant le logiciel signale de ce fait sa présence dans le réseau P2P. A chaque fois qu'un utilisateur veut exécuter un calcul parallèle, l'intergiciel va dynamiquement découvrir des resources disponibles, c'est-à-dire d'autres pairs connectés.
- **MPJ** : la bibliothèque de communication que nous avons implémentée suit la recommandation Message Passing for Java (MPJ) [18], qui est une adaptation de MPI pour Java. Nous avons ainsi la même richesse d'expression du parallélisme qu'avec MPI.

[2] `http://www.p2pmpi.org`

– **Réplication** : la tolérance aux pannes est basée sur la réplication des exécutions. Un processus peut avoir un ensemble de copies qui s'éxecutent parallèlement, et sont capables de prendre le relais en cas de panne.

4.3 Travaux connexes

Chacune des orientations ci-dessus aborde un thème de recherche différent. En dehors de l'adoption de Java comme moyen de virtualisation, les autres approches font l'objet de travaux connexes : la gestion des ressources de grille selon un réseau pair-à-pair est une évolution majeure et faisant consensus pour passer à l'échelle, le travail sur la standardisation d'une adaptation pour Java a permis à plusieurs implémentations de voir le jour, et enfin les travaux existants sur la tolérance aux pannes ont été approfondis pour les programmes à passage de message.

4.3.1 Intergiciels et structuration P2P

Aujourd'hui, il est communément reconnu que des approches complètement distribuées peuvent être envisagées pour gérer les ressources d'une grille, et ainsi contourner les limitations d'un serveur central, de serveurs répliqués ou d'une architecture hiérarchique. Foster et Iamnitchi [25], et Talia et Trunfio [64] ont montré les points communs entre la thématique du P2P et celle des grilles. Ces points communs sont le dynamisme, la grande échelle et l'hétérogénéité des ressources. Transposer l'interrogation du système d'information d'une grille dans un système P2P revient à faire une requête de découverte de ressources. Les avantages par rapport aux solutions centralisées ou hiérarchiques sont donc, en particulier, l'autonomie des resources, permettant de simplifier la maintenance, et l'extensibilité des opérations de découverte ou de recherche. La contrepartie est que la réalisation de certains services de grille sont plus difficiles à mettre en œuvre. Une requête complexe du système d'information, comme par exemple une recherche sur plusieurs critères comme le propose depuis longtemps le match-making de Condor [42], qui permet de sélectionner des ressources sélectionnées vérifiant des propriétés (mémoire, CPU, version système, etc), ou encore une recherche partielle sur un nom, ou dans un intervalle de valeurs, est plus complexe à réaliser qu'avec un annuaire centralisé.

En effet, aucune approche simple de structuration d'un réseau P2P ne répond simultanément à tous les besoins potentiels d'un système d'information de grille. On distingue habituellement les réseaux P2P selon qu'ils sont *structurés* ou *non-structurés*.

Dans le premier cas, les pairs sont organisés d'une manière régulière dans une topologie logique, qui peut être un anneau (Chord [63], Pastry [56], Tapestry [75]), un hypercube (CAN [51]) ou même un diagramme de Voronoï (VoroNet [9]). L'insertion et la consultation d'éléments utilisent quasiment toujours une table de hachage distribuée (DHT), ou plus rarement une structure de type arbre préfixe (P-Grid [1]). Les DHT permettent d'une part de réaliser simplement un équilibre de charge en choisissant une fonction de hachage qui répartit uniformément les objets dans l'espace des nœuds. D'autre part, elle permet un routage efficace, en général en $O(\log n)$ sauts pour atteindre le nœud sur lequel on doit stocker ou récupérer un objet. Cependant, les objets doivent être identifiés de manière unique. La contrepartie est la rigidité des opérations de consultation puisque les requêtes complexes évoquées plus haut, comme la recherche sur une partie de l'identifiant, sur un intervalle de valeurs, ou sur plusieurs critères ne sont pas possibles directement. Un autre inconvénient des DHT est que la répartition équilibrée des objets dans l'espace logique, ne tient pas compte de la topologie physique sous-jacente du réseau.

Dans le cas des réseaux non-structurés, dont Gnutella [4] est le représentant le plus connu, la connectivité aux voisins est quelconque, et la recherche d'un élément se fait le plus souvent par innondation (*flooding*), c'est-à-dire une diffusion de la requête à tous ses voisins, et récursivement,

avec un compteur (*time to live*) limitant le nombre de noeuds visités. L'inondation génère beaucoup de messages, mais ce coût peut être amoindri par l'utilisation de marches aléatoires [35], qui limite la diffusion à un seul des voisins. L'avantage déterminant de l'innondation est qu'elle autorise des requêtes complexes, et que celles-ci sont traitées assez simplement puisqu'il suffit de les diffuser et de les exécuter sur chacun des noeuds visités.

Dans les intergiciels de grille mettant en œuvre une organisation P2P des ressources, les choix diffèrent largement selon les fonctionnalités et l'efficacité envisagées. De multiples intergiciels ne demandent pas plus que de pouvoir se reposer sur une couche logicielle capable de gérer les arrivées et départs de pair, capable de faire communiquer des pairs du réseau P2P, et prenant en charge des requêtes simples. Le projet JXTA [67], grâce à une séparation claire des protocoles et des implémentations en plusieurs langages, a connu un succès important, et a servi de base à de nombreux projets, comme par exemple P3 [60], JNGI [72], Japanelo [66], ou JuxMem [5].

D'autre part, certains intergiciels pré-existants ont greffé une couche P2P, limitée à la gestion de leurs besoins spécifiques. Par exemple, dans ProActive [15], la couche P2P doit permettre de découvrir un nombre donné de ressources qui doivent être co-allouées. Pour cela, le mécanisme adopté est une innondation dans un réseau non-structuré : à partir d'un pair p, la requête de réserver n pairs est diffusée à tous les voisins immédiats de p, qui retournent à p l'information qu'ils acceptent la réservation. En retour, p confirme la réservation par un message contenant le nombre n' de ressources restant à réserver, et les voisins de p diffuse la requête récursivement pour n' ressources. Parmi les limites de ce système, on note le nombre important de messages envoyés, les ressources réservées de manière inutile, et qu'il n'est tenu aucun compte de la proximité physique des ressources réservées.

Quand on souhaite un système plus général, cumulant les avantages des réseaux structurés et non-structurés, la conception de la couche P2P devient plus complexe. Les propositions avec cette ambition utilisent simultanément plusieurs structurations du réseau. Si on prend la problématique de la proximité physique des pairs[3], largement étudiée dans le cadre des réseaux structurés, l'approche la plus fréquente consiste en une DHT hiérarchique [27, 73, 74], qui superpose à la DHT usuelle un ensemble de *supernodes* qui sont des représentants d'un sous-ensemble de noeuds ayant une forme de localité physique dans le réseau. Le projet Bamboo [54] a le même objectif. Lors de la maintenance d'une DHT, il existe une flexibilité dans le choix des nœuds voisins et dans celui des choix distants, et Bamboo choisit l'ensemble des nœuds voisins d'un pair en favorisant la localité.

Cet effort est pourtant peu représenté dans les intergiciels de grilles. Dans le domaine des intergiciels capables de déployer des programmes parallèles (toute paire de noeuds est capable de communiquer directement), seuls Zorilla [22] (basé sur le projet Ibis, voir section 4.3.3), Vigne [55, 39], et Vishwa [53] intègrent une forme de proximité réseau. Pour cela, Zorilla et Vigne utilisent deux types de réseaux P2P. Zorilla utilise pour le service de découverte et d'allocation, un mécanisme d'innondation dans un réseau non-structuré, mais les relations de voisinages sont données par l'intermédiaire d'une DHT Bamboo, assurant ainsi une localité entre pairs. De même, Vigne, pour localiser une ressource de manière efficace, utilise les algorithmes de routage de Pastry et ceux de Bamboo pour la maintenance de la DHT, avec toutefois un critère simpliste de localité basé sur la forme du nom DNS de l'hôte. A côté de ce réseau structuré, il construit un autre réseau non-structuré selon le protocole Scamp [26], qu'il utilise pour la découverte, avec possibilité d'exprimer des requêtes complexes. Vishwa propose également une architecture en deux couches. Une couche haute gère la localité grâce à des serveurs zonaux (sortes de supernodes) : quand un pair s'inscrit auprès de ce serveur zonal, celui-ci lui retourne prioritairement des voisins proches. A l'intérieur d'une zone, la couche sous-jacente, dite de reconfiguration, est un réseau structuré qui assure le routage et la du-

[3] la métrique utilisée est le *stretch*, c'est-à-dire le ratio de la latence d'une opération de consultation, sur le RTT (réseau physique) entre le nœud à l'origine de la requête et le nœud destination.

plication des informations stockées par des algorithmes à la Pastry. Le routage entre zones utilisant des algorithmes similaires à Chord. Enfin, la communication des applications repose sur des canaux de communication Distributed Pipes [40].

De nombreuses propositions ont aussi été faites pour pouvoir exprimer des requêtes complexes dans un réseau structuré. Les recherches multi-critères se font souvent au prix de l'utilisation d'une DHT par critère, comme par exemple dans Mercury [10] ou SWORD [46], ou avec une large redondance de l'information [68]. Une alternative est d'ajouter une structure logique supplémentaire adaptée au dessus du réseau structuré. Plusieurs auteurs ont proposé d'ajouter un arbre lexicographique (ou de préfixes, ou *trie*) au dessus d'une DHT [49, 20, 16]. On peut illustrer l'intérêt de l'approche pour les intergiciels de grilles basés sur la publication de services, offrant un modèle de type GridRPC [59]. Par exemple, l'intergiciel DIET [17] a évolué d'une structuration hiérarchique à une structuration P2P des master agents (les interfaces avec le client). Pour résoudre le problème de la découverte et de l'insertion des services dans cette infrastructure distribuée, les auteurs proposent une telle structure d'arbre préfixe distribuée [16], placée directement au-dessus d'un annuaire ou d'une DHT. La structure d'arbre préfixe est naturellement adaptée à la recherche sur une partie du début du nom d'un service, et permet de refléter la proximité sémantique des services dont le nom est proche. Des recherches multi-critères sont également possibles en explorant simultanément plusieurs branches de l'arbre.

4.3.2 Gestion des pannes

La gestion des pannes couvre deux aspects dans P2P-MPI. D'une part, la tolérances aux pannes, et d'autre part la détection des pannes. Nous examinerons les techniques existantes de détection de panne dans la section 4.6, qui explique comment nous les avons adaptées pour notre projet. Pour la gestion des pannes en revanche, notre aproche, présentée en section 4.5, est totalement différente des propositions habituelles que nous présentons ici.

La tolérance aux pannes pour MPI est quasiment toujours traitée par une solution de type *rollback and recovery*. De nombreux protocoles de ce type ont été proposés. Ces protocoles sont basés soit sur un *checkpoint* coordonné, soit sur du *message logging* complété par des checkpoints asynchrones. Dans le checkpoint coordonné, un processus coordonnateur ordonne à tous les processus de faire une image de leur état courant, l'ensemble de ces états formant un point de reprise possible après une panne. Cette approche est simple à réaliser mais induit un surcoût important car il faut synchroniser les processus avant qu'ils ne procèdent au checkpoint.

Dans les protocoles basés sur du message logging, les processus enregistrent les évènements non-déterministes (e.g arrivées de messages) sur un support fiable. Après une panne, chaque processus rejoue les évènements pour revenir à son état précédant la panne. On distingue trois types de messages logging [2].

- Le *pessimistic logging*, dans lequel la réception du message est bloquée jusqu'à ce que le message reçu ait bien été sauvegardé sur le support fiable. En cas de panne, le mécanisme de reprise est simple : chaque processus repart du dernier checkpoint et ré-exécute les communications à partir des enregistrements du support fiable.
- L'*optimistic logging*, dans lequel un message reçu par un processus, est envoyé vers le support fiable de manière asynchrone (afin d'accroître la performance) pour le sauvegarder. La contrepartie est que la reprise en cas de panne est beaucoup plus compliquée : entre le dernier message sauvé et le moment de la panne, l'exécution peut avoir modifié l'état d'autres processus. Il est alors nécessaire de remonter au dernier état cohérent, qui peut être le début de l'exécution dans le pire des cas (effet domino).

– Enfin, le protocole *causal* emprunte aux deux approches précédentes en véhiculant les messages de sauvegarde avec les messages normaux jusqu'à ce qu'ils soient sauvés.

Citons quelques unes des implémentations de ces différentes approches. L'une des premiers projet, CoCheck [62], ainsi que la distribution bien connue LAM/MPI [57] proposent une bibliothèque MPI avec tolérance aux pannes basée sur du checkpointing coordonné. Dans la proposition de MPI-FT [43], tout le traffic est mémorisé en le faisant transiter par un processus de monitoring, ce qui revient à faire du pessimistic logging. Des travaux plus récents ont étudié des approches mixtes ou alternatives. Parmi celles-ci, MPICH-V, greffée sur l'implémentation MPICH. Le première version [12] propose un checkpoint non-coordonné avec pessimistic message logging. Le surcoût s'avère être important avec cette solution, en raison de l'activité d'enregistrement des messages reçus vers le support fiable, qui diminue de moitié la bande passante disponible. Dans une version suivante, MPICH-V2 [14], les même auteurs proposent de séparer l'enregistrement des messages en deux parties : le message lui-même est sauvegardé localement, et seuls la date et l'identifiant du message sont envoyés sur le support fiable. La performance se rapproche de celle observée sans tolérance aux pannes, excepté quand il y a beaucoup de petits messages. Avec une perspective différente, FT-MPI [23] offre un cadre logiciel pour la détection des pannes, et laisse l'utilisateur programmer dans son application, le comportement à adopter en cas de panne. Des primitives supplémentaires sont fournies pour gérer la panne, comme par exemple la réduction du communicator quand un processus est défaillant. Par contre, FT-MPI ne fournit aucun moyen de reprise, comme du checkpointing par exemple. Notons que l'environnement de meta-computing H2O [21], qui permet d'utiliser FT-MPI à cheval sur plusieurs domaines administratifs, est l'un des rares projets qui tentent de traiter les exécutions MPI utilisant des réseaux longues distances.

Le seul projet à notre connaissance qui a proposé de la réplication pour la tolérance aux pannes en MPI est MPI/FT [8]. La détection des pannes est faite par des threads additionnels de surveillance, qui envoient des battements de cœur et votent pour parvenir à un consensus sur l'état des processus. Les pannes sont tolérées grâce à la réplication, qui est adaptée selon le modèle d'application (e.g master-slave, SPMD, ...). La différence majeure par rapport à notre approche est que le protocole est basé sur un coordonnateur à travers lequel tous les messages transitent. Ce schéma présente de forts risques de ne pas être extensible.

4.3.3 Bibliothèque de communication

Nous avons choisi Java pour développer l'intergiciel, dans le but de simplifier les problèmes liés à l'hétérogénité logicielle. Pour profiter pleinement de cette caractéristique, nous devons donc aussi développer une bibliothèque de communication en Java « pur » (c'est-à-dire sans utiliser JNI). Concernant ses fonctionnalités et sa sémantique, on souhaite avoir les mêmes que MPI [61], le standard de-facto pour le modèle de programmation à passage de messages. Cependant MPI ne définit une API que pour C/C++/Fortran, et pas pour Java. Devant ce manque, un groupe a travaillé dans le cadre du Java Grande Forum, pour adapter MPI à Java, ce qui a débouché sur la proposition *Message Passing for Java* (MPJ) [18].

La définition de MPJ a permis à plusieurs implémentations de voir le jour, dont MPJ Express [7] et MPJ/Ibis [11], apparues approximativement au même moment que P2P-MPI. MPJ Express est une implémentation performante qui offre des communications TCP ou myrinet. En revanche, elle ne fournit pas d'outils particuliers pour exécuter des applications sur des grilles. MPJ/Ibis est une autre implémentation, qui s'appuie sur le système Ibis [70]. Ibis désigne toute une pile logicielle, dont l'une des couches, la Portability Layer (IPL), fournit une interface orientée objet à des primitives de communication. Différents modèles de programmation ont été développés dans le cadre d'Ibis (comme par exemple le modèle de programmation de type *fork-and-join* Satin [69]). MPJ/Ibis est l'un d'eux et repose, commme les autres modèles de programmation, sur la couche IPL pour

les communications. Comme le projet Ibis ambitionne d'aborder tous les aspects du déploiement de programmes sur grille, le module Zorilla (dont nous avons parlé en section 4.3.1) a été développé récemment pour traiter le problème de la gestion des ressources. Ce module permet la découverte de ressources et le déploiement d'applications dans un mode P2P. P2P-MPI partage avec Ibis l'objectif de simplifier la tâche de l'utilisateur pour le déploiement de ses applications sur grille. Cependant, MPJ/Ibis ne fournit pas de solutions pour la tolérance aux pannes. Ce point d'ailleurs n'est, à notre connaissance, présent dans aucune autre implémentation MPJ en dehors de P2P-MPI. Pour gérer la tolérance aux pannes, nous verrons que nous devons modifier un certain nombre de structures de données internes à la bibliothèque, ce qui rend difficile la ré-utilisation de code des autres implémentations MPJ.

4.4 L'architecture de P2P-MPI

L'objectif visé par P2P-MPI est la simplicité d'utilisation de l'environnement. Cet objectif nous a fait privilégier une conception intégrée de toutes les fonctions. Concrètement, tout l'intergiciel et la bibliothèque de communication tiennent en une archive Java et quelques scripts. Le développement de programmes parallèles ne nécessite qu'un compilateur Java.

Pour exécuter un programme, un utilisateur doit avoir démarré l'intergiciel sur une machine qui devient alors un pair, prêt à partager son processeur, ou utiliser ceux d'autres pairs. Le lancement de l'intergiciel démarre les différentes fonctionnalités de P2P-MPI. Dans le schéma des différentes couches logicielles représentées en figure 4.1, les fonctionnalités propres à P2P-MPI sont présentées en grisé.

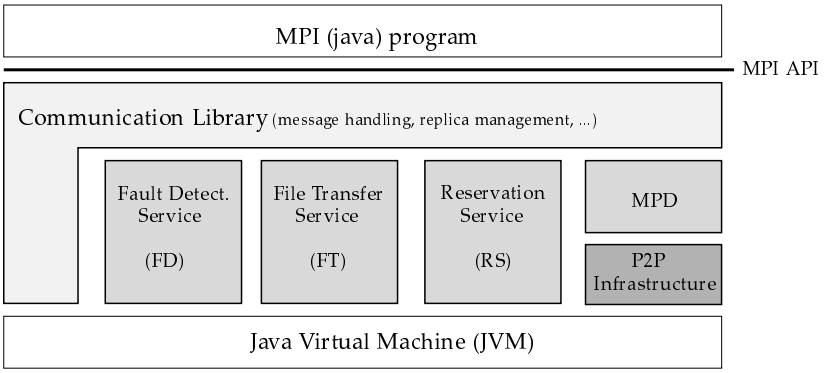


FIG. 4.1 – L'architecture de P2P-MPI

La couche la plus haute est la bibliothèque de communication, qui fournit une API MPJ aux programmes Java de l'utilisateur. Cette couche de communication s'appuie sur une couche de type *intergiciel*, qui fournit différents services. Il y a :

- le service de détection des pannes (FD),
- le service de transfert de fichiers (FT),
- le service de réservation (RS),
- le service de découverte (MPD)[4]

[4]multi-purpose daemon, en référence au MPD de la distribution MPICH [37].

Ce dernier service (MPD), est l'interface en charge de la sélection des ressources. Quand l'utilisateur local demande l'exécution de son programme avec vingt processus, c'est le MPD qui a la charge de les trouver et de fournir les informations de contact nécessaires pour ces ressources.

Pour cela, il s'appuie sur une couche inférieure de gestion des ressources. La gestion des ressources consiste ici à attribuer des identificateurs aux ressources, les localiser, déterminer leurs états et leurs disponibilités, etc. Nous qualifions ce niveau de couche *infrastructure* car la façon dont les ressources sont gérées dépend fortement du paradigme adopté, centralisé ou décentralisé. Comme annoncé, nous avons opté ici pour une approche pair-à-pair, réputée plus extensible qu'un annuaire centralisé.

4.4.1 La couche infrastructure

Le rôle du module de gestion de l'infrastructure est de maintenir, et de tenir à disposition, des informations pertinentes et actualisées concernant les ressources. Au moment de la conception de P2P-MPI, il était clair pour nous que ce n'était pas notre rôle d'innover dans le domaine des réseaux P2P. Nous n'avions alors que peu de recul sur les travaux de recherche qui visaient à améliorer les réseaux P2P structurés (voir section 4.3.1), alors en plein foisonnement. Nous cherchions une bibliothèque générique permettant de gérer un réseau P2P, avec une implémentation stable. De fait, le choix était restreint en 2004. Nous avons alors choisi JXTA [67], qui est un modèle hybride entre les réseaux stucturés et non-structurés. Ce choix s'est révélé erroné vis-à-vis de nos besoins, comme nous l'expliquons ci-dessous. Un argument séduisant de JXTA sur le plan du développement logiciel, est la séparation entre la définition des protocoles, et les implémentations existantes, disponibles dans plusieurs langages. De plus, l'entreprise Sun apportait son soutien à des implémentations en Java et en C, garantissant ainsi une plus grande stabilité de l'implémentation et de son API.

Un pair en JXTA, signale sa présence au moyen d'un *advertisement* (un petit message descriptif en XML). Les advertisements sont stockés et récupérés dans une table de hachage distribuée (DHT) maintenue par des pairs au rôle spécifique, appelés *rendezvous*. Chaque rendezvous maintient une liste des autres rendezvous connus. Cette liste est appelée la *rendezvous Peer View* (RPV). Quand un rendezvous doit stocker un advertisement, il applique une fonction de hashage pour déterminer vers quel rendezvous de sa RPV l'advertisement doit être dirigé pour y être stocké (il est aussi répliqué sur les deux rendezvous voisins dans sa RPV). La consultation utilise la même fonction de hachage pour savoir quel rendezvous possède l'advertisement.

Cependant, JXTA s'est montré inadapté à nos besoins pour plusieurs raisons. Premièrement, la DHT a été conçue pour être extensible à un très grand nombre de pairs et les auteurs ont choisi pour cela une coordination des rendezvous qualifiée de *loosely-coupled*. Si un rendezvous disparaît du réseau, le système n'essaye pas de corriger l'incohérence des RPV au plus vite, au contraire de ce que font des systèmes structurés avec DHT comme Chord [63] ou CAN [51]. Si la consultation d'un advertisement échoue en raison d'une telle incohérence, un mécanisme de parcours systématique des rendezvous est prévu pour retrouver l'advertisement. La conséquence est que le système ne peut garantir de trouver un advertisement dans un temps donné. L'étude expérimentale [6] faite dans un environnement comparable au nôtre (en terme de stabilité des rendezvous) montre que les rendezvous n'ont que très rarement une RPV cohérente. Deuxièmement, la capacité de JXTA à passer les pare-feux en utilisant des pairs *relay* n'est pas un argument dans notre contexte visant la performance, en raison du goulot d'étranglement que constituent ces pairs relay. Troisièmement, JXTA fait abstraction du réseau physique et il est impossible de dire quelle proximité ont les pairs. C'est pourtant une information importante pour améliorer la performance des programmes à passage de messages.

Nous avons évoqué les travaux autour de cette question dans la section 4.3.1. Cependant, aucune implémentation stable et sans dépendance n'était disponible quand nous avons choisi d'intégrer des

informations de proximité. Pour ces raisons, nous avons remplacé JXTA par notre propre module de gestion de l'infrastructure en mode pair-à-pair. Les avantages par rapport à JXTA sont la découverte exhaustive des ressources, la vitesse de la découverte, et l'obtention d'informations concernant la latence réseau entre les pairs. Du point de vue de l'utilisateur, il n'y a quasiment aucun changement : le concept de rendezvous est simplement remplacé par celui de supernode, tel qu'on le retrouve dans gnutella [4]. Quand un pair veut rejoindre le réseau, il doit connaître un supernode, auprès duquel il s'enregistre (son identifiant est son IP), et récupère la liste des pairs connus qu'il stocke dans un cache local. Le supernode enregistre les arrivées et départs de pairs, et pour chaque pair, mémorise ses ports de services et la date de leur dernier contact. Périodiquement, les pairs contactent le supernode pour récupérer une liste à jour des autres pairs du réseau, et actualisent ainsi leur copie locale. Dans la liste locale, une latence réseau est associée à chaque pair. Cette mesure est prise périodiquement par le MPD en envoyant un message vide vers chaque machine concernée.

Notre développement s'est limité à un système avec un unique supernode, l'objectif n'étant pas de démontrer l'extensibilité d'un tel système. Cependant, dans nos expériences ayant impliqué jusqu'à 600 pairs, le système n'a montré aucun signe de surcharge.

4.4.2 La couche intergicielle

Le rôle de la couche intergicielle est de gérer les besoins du programme à exécuter. Ses principales attributions sont i) la sélection, la réservation et l'allocation de ressources adéquates, puis ii) la surveillance du déroulement de l'exécution.

La tâche i) est assez complexe, car elle nécessite pour des programmes de type MPI une co-allocation des ressources : tous les processus composant le programme parallèle doivent démarrer en même temps. Nous avons proposé le schéma de co-allocation, illustré par la figure 4.2 qui montre comment les différents services de cette couche coopèrent dans cette tâche.

Dans ce diagramme, on suppose que l'utilisateur a déjà démarré P2P-MPI, et sa présence dans le réseau P2P est matérialisée par le MPD lancé. Le démarrage du logiciel lance aussi les services FT, FD et RS.

❶ **Requête d'exécution** Le demandeur invoque, pour exécuter un programme : `p2pmpirun -n` *n* `-r` *r* `-a` *alloc* `prog`. Seul le nombre *n* de processus est obligatoire. Les autre arguments sont optionnels : *r* le degré de réplication pour la tolérance aux pannes (c.f § 4.5), et *alloc* qui indique quelle stratégie utiliser pour allouer les processus. Ensuite le processus de rang 0 de l'application est démarré sur la machine locale.

❷ **Requête de découverte** L'application contacte le MPD local et lui demande de trouver suffisamment de pairs pour exécuter ce programme nécessitant $n \times r$ processus.

❸ **Découverte et réservation** Le MPD sélectionne un sous-ensemble de pairs de son cache local et mandate son RS pour qu'il leur envoie une demande de réservation. Le RS retourne ensuite le résultat au MPD. Si les pairs réservés ne sont pas en nombre suffisant, le MPD ré-interroge le supernode à la recherche de nouveaux pairs pour étendre la réservation. Si les réservations obtenues sont insuffisantes, le MPD arrête et signale l'échec de la requête.

❹ **Enregistrement** La réservation effectuée, le MPD local contacte chacun des MPD sur les hôtes réservés. Il leur communique le nom de l'application, le rang attribué dans le communicator, et l'IP et le port du processus de rang 0, afin qu'ils puissent le contacter.

❺ **Prise de contact** chaque MPD distant utilise cette dernière information pour communiquer les numéros de port utilisés par ses services FT et FD au processus de rang 0.

❻ **Transfert des fichiers** Les programmes exécutables et les fichiers d'entrée sont téléchargés via le FT.

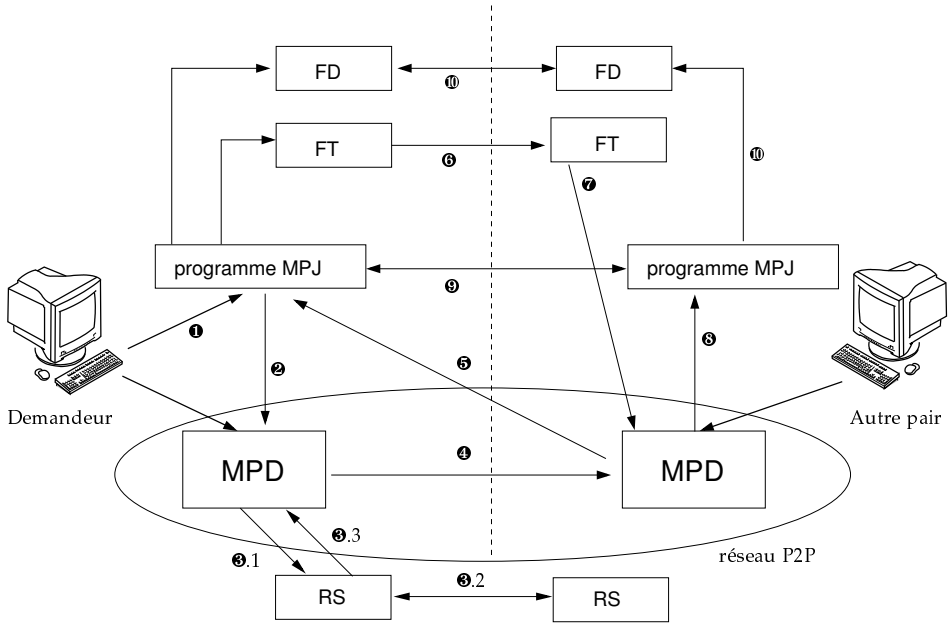


FIG. 4.2 – Protocole pour la co-allocation de ressources.

❼ **Notification** Le transfert terminé, le FT en réception notifie le MPD local que l'exécutable est présent.

❽ **Démarrage distant** Le MPD démarre un processus exécutant le programme téléchargé.

❾ **Création du communicator** Tous les processus envoient leur IP associée à leur rang attribuée par le processus de rang 0, qui construit une table des ces associations, et la renvoie à tous les processus.

❿ **Surveillance de l'exécution** Les processus s'enregistrent auprès de leur FD, qui surveille leur bon fonctionnement, et les informe si une panne survient sur un processus distant.

Les étapes du protocole précédent sont plus ou moins complexes. Parmi celles-ci, l'étape ❸ cache l'ordonnancement, c'est-à-dire dire *où* et *quand*, les processus vont s'exécuter. Pour les programmes MPI, il n'y a que peu de lattitude sur le *quand* : les processus doivent s'exécuter simultanément. Concernant le *où*, il s'agit de trouver un « bon » sous-ensemble des processus.

Ce problème peut être formulé sous la forme d'un problème de *partionnement de graphe*, dans lequel les arrêtes sont pondérées par les volumes des communications. Ces problèmes étant NP-complet [28], nous avons travaillé sur une heuristique de placement des processus qui minimise les temps de communication inter-processus [13]. Concrètement, il faut disposer des mesures de bande passante entre tous les pairs, et il faut enregistrer lors d'une première exécution, le volume de toutes les communications ayant eu lieu au cours du programme. Un algorithme de clustering est ensuite appliqué pour former des groupes de processus communiquant le plus. Ensuite, les groupes de machines ayant la meilleure connectivité sont construits par un clustering similaire. Enfin, la dernière phase consiste à placer les groupes de processus sur les groupes de machines. Cependant, l'acquisition des mesures de communication et de connectivité réseau sont bien trop difficiles dans la pratique pour que l'approche ait un intérêt réel.

Pour conserver la simplicité d'utilisation, nous restreignons nos critères de choix, dans les versions publiques de P2P-MPI, à la latence réseau et au partage de la mémoire. Précisons que le propriétaire d'une machine peut configurer deux paramètres : le nombre J d'applications que la machine peut exécuter simultanément, et le nombre P de processus exécutés au maximum pour cette application. Par exemple J=2 et P=1 permettrait à deux utilisateurs distincts d'exécuter simultanément un des processus de leur application sur cette machine. J=1 et P=2 permet à une même application d'avoir deux de ses processus sur cette machine (cette configuration est pertinente pour une machine dual-core). L'utilisateur qui lance l'exécution influence l'allocation à travers un choix simple :

- il peut demander à *concentrer* les processus en utilisant le maximum de ressources déclarées sur une même machine. La localité des processus hébergés sur cette machine est maximale mais ils doivent partager la mémoire.
- il peut demander à *étendre* l'ensemble des machines utilisées pour tenter de n'avoir qu'un processus par machine.

Dans les deux cas, on priviligie d'abord la proximité réseau. Une liste de machines, classée par latence croissante, est d'abord réservée selon ce critère, puis la procédure d'allocation prend place. Dans la première stratégie, on parcourt la liste à partir du début, et pour chaque machine, on alloue les J processus disponibles sur cette machine, jusqu'à ce que tous les processus soient alloués. Dans la deuxième stratégie, on alloue un seul processus sur chaque machine, et si on atteint la fin de la liste des machines disponibles, on recommence en repartant du début.

4.4.3 La couche bibliothèque de communication

La bibliothèque de communication implémente MPJ [18]. L'implémentation ne fait que du TCP, mais propose deux variantes correspondant à deux objectifs d'utilisation. L'objectif initial était d'utiliser P2P-MPI à large échelle, couvrant potentiellement des domaines administratifs différents, tout en ayant des communications dont la performance soit compétitive par rapport à des modèles comme RMI ou Corba par exemple. Nous avons donc implémenté les communications avec des sockets Java TCP, en faisant le choix de n'ouvrir qu'une connexion à la fois. Ceci permet de ne gérer l'ouverture que d'un port dans les pare-feux.

Plus récemment, une version n'imposant pas de restriction sur le nombre de ports ouverts simultanément a été developpée. Elle utilise la classe Java `nio` qui permet de surveiller les évènements se produisant sur un vecteur de descripteur de fichiers. Cette implémentation est largement plus performante, mais le nombre de ports ouverts doit être aussi grand que le nombre de processus utilisés.

Les deux implémentations utilisent des algorithmes connus pour optimiser les communications collectives. Des détails concernant ces techniques sont donnés dans [52], et nous ne donnons ici qu'un résumé des algorithmes utilisés (Table 4.1).

Primitive	Algorithme	Primitive	Algorithme
`Allgather`	Gather puis Bcast	`Gather`	Flat tree
`Allgatherv`	Gatherv puis Bcast	`Gatherv`	Flat tree
`Allreduce`	Butterfly[48] ou Reduce puis Bcast	`Reduce`	Binomial tree ou flat tree
`Alltoall`	Asynchronous rotation	`Reduce_scatter`	Reduce puis Scatterv
`Alltoallv`	Asynchronous rotation	`Scatter`	Flat tree
`Barrier`	4-ary tree	`Scatterv`	Flat tree
`Bcast`	Binomial tree		

TAB. 4.1 – Algorithmes utilisés pour les communications collectives

Une perspective de travail concernant ces communications collectives serait de paramétrer les algorithmes utilisés, ou d'en changer, en fonction de la topologie du groupe des processus impliqués, afin d'en optimiser la performance. Ces optimisations sont depuis longtemps présentes dans les implémentations MPI ([65],[47]) mais ne concernent que les clusters (ou en tous cas quand la latence est faible). Elles sont moins efficaces que les stratégies naïves en présence de liens réseaux longues distances, par exemple sur une exécution multi-clusters. L'optimisation des opérations collectives dans ce contexte n'a montré des résultats probants que dans certains cas [44]. Cependant, si des progrès étaient accomplis dans ce domaine, la couche infrastructure de P2P-MPI serait à même de renvoyer des informations concernant la localité des processus, pour faciliter le paramétrage de ces algorithmes.

4.5 Tolérance aux pannes

Comme annoncé en introduction, nous avons proposé dans P2P-MPI une approche originale basée sur la réplication pour la tolérance aux pannes de programmes à passage de messages. Rappelons qu'en MPI, la panne d'un seul processus entraîne la panne de tout le programme. L'idée de la réplication est d'exécuter des copies de secours de tous ou certains processus. En cas de panne, le programme peut continuer tant qu'au moins une copie de secours fonctionne. Cette approche ne garantit pas que l'exécution termine toujours[5], c'est pourquoi nous parlons plutôt de la *robustesse* d'un programme ainsi répliqué. Nous examinons par la suite la probabilité de défaillance en fonction des occurences de pannes.

Le choix de cette approche est motivé par la simplicité d'utilisation offerte à l'utilisateur : un seul argument sur la ligne de commande permet de spécifier le taux de réplication, c.f section 4.4.2. Alors que l'approche *checkpoint and restart* repose sur la présence d'un support fiable (il faut donc l'avoir à disposition et le configurer), la réplication s'intègre parfaitement dans le paradigme pair-à-pair.

Une des objections évidentes qu'on peut opposer à cette approche est la gaspillage des ressources. Pour un taux de réplication de deux, soit les copies s'exécutent sur des processeurs différents des processus originaux, et dans ce cas, l'exécution demande deux fois plus de ressources, soit les copies s'exécutent sur les mêmes processeurs, et alors le temps d'exécution est au moins multiplié par deux.

4.5.1 Protocole de réplication

Hypothèses Nous considérons que la gestion des pannes se fait dans le contexte suivant.

- Les pannes sont de type *fail-stop* (un processus défaillant arrête définitivement toute activité). Ceci couvre les pannes où i) le processus lui-même tombe en panne (par exemple après une division par 0), ii) la panne de l'ordinateur sur lequel il tourne, iii) la défaillance du détecteur de fautes qui surveille le processus, qui ne peut plus confirmer la vivacité du processus.
- Nous considérons un système distribué *partiellement synchrone* : i) les horloges des différents ordinateurs dérivent de la même manière durant une exécution, ii) il n'y a pas d'horloge globale, iii) les communications délivrent les messages dans un temps fini.
- Les liens réseaux sont fiables, c'est-à-dire qu'il n'y a pas de perte de message.

Les hypothèses sur la fiabilité du réseau sont justifiées par l'utilisation de TCP, qui est fiable, et par le fait que l'intergiciel vérifie au démarrage que les ports TCP nécessaires sont ouverts dans les pare-feux.

[5] Notons qu'avec une approche *checkpoint and restart*, des erreurs répétitives peuvent aussi empêcher toute progression de l'exécution

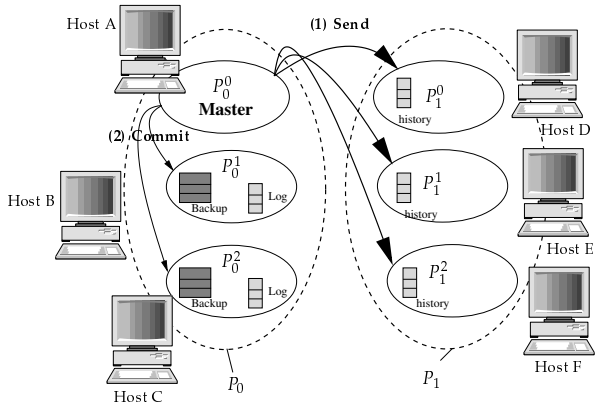


FIG. 4.3 – Une message envoyé du processus logique P_0 au processus logique P_1.

Principe Le principe de la réplication est schématisé dans la figure 4.3, dans laquelle on a représenté deux copies de secours par processus, et une communication entre deux processus. Pour le programmeur, la réplication est complètement transparente, et un envoi d'un processus P_0 à P_1 s'écrit comme d'habitude. L'implémentation gère la cohérence entre les différents réplicas d'un processus. On appelle *processus logique* l'ensemble des processus qui exécutent un même code. A l'intérieur d'un processus logique, l'un des processus est élu pour avoir un rôle de maître.

Dans la littérature, la *réplication active* [58] consiste, lors d'un envoi de message, à envoyer directement à tous les réplicas du processus destinataire une copie de ce message. Notre protocole est comparable à cette stratégie, à l'exception que nous limitons le nombre de messages en imposant que seul le maître d'un processus logique communique avec les processus destinataires.

Quand un replica atteint une instruction d'envoi de message, deux cas de figures se présentent :
- si c'est le maître, il envoie le message à tous les processus du processus logique destinataire. Une fois le message envoyé, il notifie ses réplicas que le message a été correctement envoyé (commit) en envoyant l'identifiant du message[6](MID). Les MIDs sont stockés dans une *log* sur chaque réplica.
- si ce n'est pas le maître, il regarde d'abord dans sa log. Si le MID est déjà présent, le maître a déjà envoyé le message, et le réplica continue l'exécution. Sinon, le message à envoyer est stocké en mémoire (*backup*), et l'exécution continue. Quand un réplica reçoit un commit, il écrit le MID associé dans sa log, et si le message correspondant au MID a été stocké, il l'enlève simplement de la backup.

Protocole de reprise Quand une panne est détectée, tous les autres processus en sont informés rapidement (voir section 4.6), et l'action de reprise est la suivante. Si le processus défaillant est un réplica qui n'est pas maître, tous les processus mettent simplement à jour leur communicateur afin de ne plus envoyer de message à ce processus. Si c'est un processus maître, un nouveau maître est d'abord élu parmi les réplicas restants dans le processus logique. Ensuite, il vérifie la backup. Si elle n'est pas vide, cela signifie que des messages n'avaient pas été envoyés par le maître, ou qu'il n'avait pas eu le temps de les valider par un commit. Le nouveau maître ré-envoie ces messages. Si ces messages avaient déjà été reçus (cas du commit non parvenu), le destinataire du message ignore

[6]Chaque processus peut fabriquer indépendamment un identifiant unique à partir de l'ordre d'envoi du message.

simplement le message en double grâce à l'historique des MIDs.

La cohérence de ce type de protocole repose sur la *diffusion atomique* (ou *atomic braodcast*, ou encore *total order broadcast*). De manière générale, une diffusion atomique n'est pas possible dans un système asynchrone, mais nos hypothèses restrictives permettent de l'émuler. Hadzilacos et Toueg ont formellement spécifié la diffusion atomique d'un message m à l'aide des deux primitives *broadcast(m)* et *deliver(m)* [38]. Nous avons montré que les quatres conditions nécessaires pour la diffusion atomique étaient respectées par notre protocole dans presque tous les cas.

Il y a cependant une exception liée à l'usage des spécificateurs `MPI_ANY_SOURCE` et `MPI_ANY_TAG`. Ces spécificateurs spéciaux permettent de dire respectivement, qu'on accepte la réception d'un message provenant de n'importe quel processus, ou avec n'importe quel tag. Or, la propriété *total order* de la diffusion atomique stipule que tous les processus doivent recevoir les messages dans le même ordre. Donc de manière générale, on ne peut garantir la validité de la réplication s'il est fait usage de ces spécificateurs. La raison est qu'après une panne, un réplica reprenant le rôle de maître peut avoir reçu des valeurs dans un ordre autre que celui du maître.

Cependant, on peut argumenter que l'usage de tels spécificateurs dans la pratique, sert souvent à des calculs qui ne dépendent pas de l'ordre de réception des valeurs. Un exemple typique pourrait être celui de la réduction sommation de valeurs provenant d'autres processus : on profite du fait que cette opération est commutative, associative, et possède un élément neutre pour additionner les valeurs dans n'importe quel ordre au fur et à mesure qu'elles sont reçues.

4.5.2 Évaluation quantitative de la robustesse

Pour évaluer l'effet d'un taux de réplication donné sur la robustesse de l'application, c'est-à-dire sa probabilité de panne après un certain nombre de pannes de ses processus, nous devons décider d'un modèle de panne. Récemment, le travail de Nurmi et al. [45] a étudié les pannes dans des environnements tels que ceux que nous visons. Cette étude montre que c'est avec une loi de Weibull qu'on modélise le mieux la distribution des pannes. En utilisant leur formulation, la probabilité qu'une machine tombe en panne avant le temps t est :

$$Pr([0,t]) = 1 - e^{-(t\lambda)^{\delta}} \tag{4.1}$$

où $\lambda > 0$ est le taux de panne, et $\delta > 0$ le paramètre de forme. Dans ce même article, les auteurs montrent comment calculer λ and δ à partir des observations.

D'autre part, rappelons que l'application est composée de processus, et que la défaillance de n'importe lequel d'entre eux fait tomber en panne l'application en entier. On considére les pannes de processus comme des évènements indépendants, qui peuvent se produire de manière équiprobable sur les machines. Notons $f(t)$ la probabilité qu'une machine tombe en panne avant la date t. Pour une application à p processus sans réplication la probabilité qu'elle tombe en panne est :

$$\begin{aligned} P_{app(p)} &= \text{probabilité que 1, ou 2, } \ldots \text{, ou } p \text{ processus défaille} \\ &= 1 - (\text{probabilité qu'aucun processus ne défaille}) \\ &= 1 - (1 - f(t))^p \end{aligned}$$

Avec un taux de réplication r, l'application tombe en panne si les r réplicas d'au moins un processus logique sont défaillants. La probabilité que les r réplicas tombent en panne est $(f(t))^r$. Donc, par le même principe que ci-dessus, la probabilité de panne d'une application à p processus avec un taux de réplication r est :

$$P_{appr(p,r)} = 1 - (1 - f(t)^r)^p \tag{4.2}$$

Et en instanciant $f(t)$ avec le modèle réaliste de pannes de l'Eq. (4.1), on a :

$$P_{appr(p,r)} = 1 - (1 - (1 - e^{-(t\ \lambda)^{\delta}})^r)^p \tag{4.3}$$

On peut donc évaluer la probabilité de panne de l'application, dépendant du temps estimé d'exécution. Cependant, la réplication ajoute un surcoût au temps d'exécution, en raison des communications supplémentaires nécessaires. L'exécution étant plus longue, la probabilité de panne augmente, et il est donc légitime de se demander si, pour un nombre de processus fixé, il existe un taux de réplication optimal. Nous avons traité cette question [29] en modélisant le temps d'exécution du programme parallèle avec réplication par une loi d'Amdahl étendue tenant compte du surcoût de la réplication. Dans ce modèle, la probabilité de panne est une fonction convexe dont le minimum indique le taux optimal de réplication. Pour des paramètres réalistes, le taux de réplication fait décroître rapidement la probabilité de panne, et l'optimal n'est atteint qu'avec des valeurs déraisonnables de taux de réplication. Une question duale à laquelle on peut répondre avec ce modèle est le taux de réplication nécessaire pour que la probabilité de panne ne dépasse une valeur donnée.

4.6 Détection des pannes

La détection des pannes est un élément essentiel de la gestion des pannes, et en particulier pour la réplication. Par exemple, les processus doivent être informés en un temps défini de la défaillance d'un autre processus pour entreprendre les actions de reprise décrites précédemment (section 4.5.1). C'est le service de détection des pannes (FD, introduit en section 4.4), présent sur tous les noeuds, qui informe le ou les processus applicatifs locaux d'une défaillance d'un processus distant.

4.6.1 Gossiping

Les travaux de recherche sur la détection des pannes ont beaucoup progressé à partir du moment où il a été proposé de considérer le détecteur de panne comme un acteur à part entière des systèmes distribués, notamment par le travail fondateur de Chandra et Toueg [19]. Depuis, de nombreux protocoles ont été proposés et étudiés. Ceux qui nous intéressent particulièrement sont les protocoles basés sur du *gossiping*, inspirés du service de détection de panne proposé par Renesse, Minsky et Hayden [71]. Le principe du *gossiping* est que chaque élément du système distribué à surveiller est accompagné d'un détecteur local. Ces détecteurs locaux s'échangent de manière périodique, l'état de vitalité de la ressource qu'ils ont en responsabilité, mais aussi leurs connaissances les plus récentes de l'état de vitalité concernant les ressources distantes. Chaque détecteur matérialise cette connaissance globale par une table associant à chaque ressource un compteur de battements de cœur. Lors d'un échange, un détecteur reçoit une ou plusieurs de ces tables, et garde pour chaque ressource, le battement de cœur le plus élevé communiqué, ou le dernier battement de cœur observé s'il s'agit de sa propre ressource. Il ré-émet ensuite la table vers un certain nombre d'autres détecteurs.

Sur chaque détecteur, quand la différence dans le nombre de battements de cœur entre une ressource distante et la ressource locale dépasse un seuil, une panne est suspectée. Le détecteur rentre alors dans un consensus pour avoir confirmation que les autres détecteurs ont également suspecté la panne. S'il y a consensus sur la panne, chaque détecteur peut en informer sa ressource locale.

Dans la pratique, la difficulté est de choisir une bonne valeur de seuil : une grande valeur rend le temps de détection long, une petite valeur provoque de nombreux faux positifs. Le choix du seuil est lié à la stratégie de routage des messages de gossip, car il faut s'assurer qu'un message donnant une information de l'état exact de vitalité d'une ressource ait le temps de parvenir à tous les détecteurs. Nous avons utilisé dans nos premières implémentations de P2P-MPI, le protocole de [71],

basé sur une diffusion des messages vers des destinations aléatoires à chaque étape. L'inconvénient est alors qu'on ne peut dire que de manière probabiliste combien d'étapes sont nécessaires pour qu'un message atteigne tous les détecteurs. Nous avons donc étudié plusieurs variantes avec un routage déterministe [50], et nous avons opté pour la variante *Binary Round-Robin* (BRR) qui donnait un temps de détection le plus rapide. Cependant, les informations dans ce protocole sont transmises sans aucune redondance, ce qui fait que la panne de quelques détecteurs peut rapidement compromettre le système de détection tout entier (le système détecte des pannes qui n'en sont pas). Nous avons proposé une variante considérablement plus robuste, en particulier pour les systèmes de petite taille, tout en conservant un temps de détection court. Pour accroître la robustesse, notre protocole (DBRR) remplace les échanges à sens unique par des échanges à double-sens entre les détecteurs.

Ces deux protocoles sont disponibles dans les distributions de P2P-MPI, et l'un ou l'autre peut être choisi par l'utilisateur, et changé à chaque exécution. Le temps de détection est de la forme :

$$T_{detect} = \alpha \cdot \lceil log_2(n) \rceil \cdot T_{gossip} + T_{consensus} \tag{4.4}$$

avec α=2 pour BRR ou α=3 pour DBRR, et où n est le nombre de ressources, T_{gossip} est la périodicité d'échange des messages de gossip, et $T_{consensus}$ est le temps nécessaire au consensus. Notre procédure de consensus repose sur un message envoyé à la ressource suspectée. Sans réponse dans un délai imparti ($T_{consensus}$), la panne est confirmée. Les valeurs par défaut –celles qui se sont montrées pertinentes à l'usage– sont T_{gossip}=500ms et $T_{consensus}$=500ms. Il faut donc théoriquement 10,5s (BRR) et 15,5s (DBRR) pour détecter une panne dans un système à 1 000 nœuds.

Ceci a été confirmé par une expérience sur la plateforme Grid'5000. Dans cette expérience, des pairs sont répartis de manière équitable sur trois sites (Nancy, Rennes et Nice), puis on sélectionne au hasard une machine hébergeant un pair, et tous les processus Java y sont tués à une date t_1. On regarde ensuite à quelle date t_2 chaque pair est notifié de la panne. La figure 4.4 trace la moyenne des différences $t_2 - t_1$ sur l'ensemble des pairs. Le temps théorique est aussi tracé pour comparaison. On constate que les temps de détection observés sont très proches du temps théorique dans tous les cas. D'autre part, les latences inégales dues aux trois sites géographiquement éloignés utilisés, ne semblent pas gêner l'extensibilité du système.

4.7 Résumé des contributions

Sur le plan des environnements logiciels, P2P-MPI apporte à la fois une implémentation MPJ et un intergiciel dédié à son utilisation. Ce logiciel est disponible publiquement sous une licence GPL. L'environnement a été présenté pour la première fois dans [30][7]. Comme nous l'avons vu, ce travail touche à plusieurs domaines de recherche. Tout d'abord, l'intégration de la bibliothèque de communication dans un intergiciel, les services qu'il est nécessaire de proposer dans l'intergiciel, et leur architecture a été proposée et évaluée dans [30, 31]. Ensuite, Le problème de la co-allocation des ressources dans un tel intergiciel a été étudié dans [33]. Nous avons montré expérimentalement que les stratégies d'allocation proposées, qui ont pour objectif de sélectionner les pairs les plus proches en terme de latence réseau fonctionnent bien jusqu'à au moins 600 processus. Nous avons proposé la réplication comme une approche possible au problème de la tolérance aux pannes. Une évaluation de la robustesse des applications répliquées est proposée dans [32], puis dans [29], où nous étudions la question du taux optimal de réplication à l'aide d'un modèle du temps d'exécution d'un programme répliqué. L'étude du surcoût de la réplication a été approfondie dans [34], où son impact est étudié sur plusieurs types d'applications.

[7] à la même conférence était aussi présenté pour la première fois MPJ/Ibis [11].

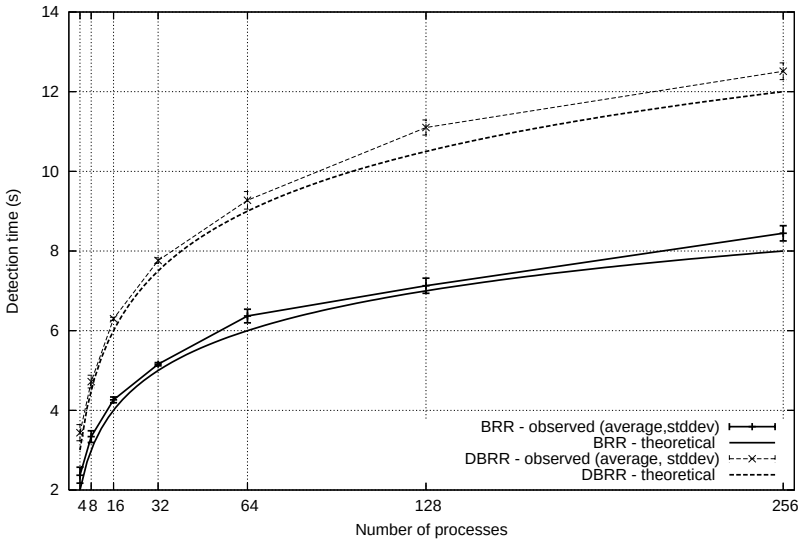


FIG. 4.4 – Temps de détection d'une faute avec BRR et DBRR

Bibliographie

[1] Karl Aberer. P-grid : A self-organizing access structure for p2p information systems. In *Cooperative Information Systems, 9th International Conference, CoopIS 2001, Trento, Italy, September 5-7, 2001, Proceedings*, pages 179–194, 2001.

[2] Lorenzo Alvisi and Keith Marzullo. Message logging : Pessimistic, optimistic, and causal. In *Proceeding of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 229–236, 1995.

[3] David P. Anderson. Boinc : A system for public-resource computing and storage. In Rajkumar Buyya, editor, *5th International Workshop on Grid Computing (GRID 2004)*, pages 4–10. IEEE Computer Society, 2004.

[4] Andy Oram. *Peer-to-Peer : Harnessing the Power of Disruptive Technologies*, chapter Gnutella, pages 94–122. O'Reilly, May 2001.

[5] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. Juxmem : An adaptive supportive platform for data sharing on the grid. *Scalable Computing : Practice and Experience*, 6(3) :45–55, 2005.

[6] Gabriel Antoniu, Loïc Cudennec, Mike Duigou, and Mathieu Jan. Performance scalability of the JXTA P2P framework. In *Proc. 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, USA, March 2007.

[7] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express : Towards thread safe Java HPC. In *CLUSTER*, pages 1–10. IEEE, 2006.

[8] Rajanikanth Batchu, Yoginder S. Dandass, Anthony Skjellum, and Murali Beddhu. MPI/FT : A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4) :303–315, 2004.

[9] Olivier Beaumont, Anne-Marie Kermarrec, Loris Marchal, and Etienne Riviere. Voronet : A scalable object network based on voronoi tessellations. In *21th International Parallel and Distributed*

Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA, pages 1–10, 2007.

[10] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury : supporting scalable multi-attribute range queries. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 30 - September 3, 2004, Portland, Oregon, USA*, pages 353–366, 2004.

[11] Markus Bornemann, Rob V. van Nieuwpoort, and Thilo Kielmann. MPJ/Ibis : A flexible and efficient message passing platform for java. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666 of *Lecture Notes in Computer Science*, pages 217–224. Springer, 2005.

[12] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djailali, Gilles Fedak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri, and Anton Selikhov. MPICH-V : Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *SuperComputing 2002*, pages 1–18, Baltimore, USA, November 2002.

[13] Ghazi Bouabene. Sélection de pairs et allocation de tâches dans p2p-mpi. Master's thesis, Université Louis Pasteur de Strasbourg, June 2006.

[14] Aurélien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. (mpich-v2) : a fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging. In *SuperComputing 2003*, pages 242–250, Phoenix USA, November 2003.

[15] Denis Caromel, Alexandre di Costanzo, and Clement Mathieu. Peer-to-peer for computational grids : mixing clusters and desktop machines. *Parallel Computing*, 33(4-5) :275–288, May 2007.

[16] Eddy Caron, Frederic Desprez, and Cédric Tedeschi. Enhancing computational grids with peer-to-peer technology for large scale service discovery. *J. Grid Comput.*, 5(3) :337–360, 2007.

[17] 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 *LNCS*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[18] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPJ : MPI-like message passing for Java. *Concurrency : Practice and Experience*, 12(11) :1019–1038, September 2000.

[19] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, 1996.

[20] Anwitaman Datta, Manfred Hauswirth, Renault John, Roman Schmidt, and Karl Aberer. Range queries in trie-structured overlays. In Germano Caronni, Nathalie Weiler, Marcel Waldvogel, and Nahid Shahmehri, editors, *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005), 31 August - 2 September 2005, Konstanz, Germany*, pages 57–66. IEEE Computer Society, 2005.

[21] David Dewolfs, Jan Broeckhove, Vaidy S. Sunderam, and Graham E. Fagg. Ft-mpi, fault-tolerant metacomputing and generic name services : A case study. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 133–140. Springer, 2006.

[22] Niels Drost, Rob V. van Nieuwpoort, and Henri Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops (CCGRID'06)*, pages 14–21. IEEE, 2006.

[23] Graham Fagg and Jack Dongarra. FT-MPI : Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *EuroPVM/MPI User's GroupMeeting 2000*, pages 346–353. Springer-Verlag, Berlin, Germany, 2000.

[24] Ian Foster and Carl Kesselman. Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, 1997.

[25] Ian T. Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II, Second International Workshop, IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.

[26] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Computers*, 52(2) :139–149, 2003.

[27] Luis Garcés-Erice, Ernst W. Biersack, Keith W. Ross, Pascal Felber, and Guillaume Urvoy-Keller. Hierarchical peer-to-peer systems. *Parallel Processing Letters*, 13(4) :643–657, 2003.

[28] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[29] Stéphane Genaud, Emmanuel Jeannot, and Choopan Rattanapoka. Fault management in P2P-MPI. *International Journal of Parallel Programming*, 37(5) :433–461, August 2009. Extended version of [32].

[30] Stéphane Genaud and Choopan Rattanapoka. A peer-to-peer framework for robust execution of message passing parallel programs on grids. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings*, volume 3666 of *Lecture Notes in Computer Science*, pages 276–284. Springer, 2005.

[31] Stéphane Genaud and Choopan Rattanapoka. P2p-mpi : A peer-to-peer framework for robust execution of message passing parallel programs on grids. *Journal of Grid Computing*, 5(1) :27–42, 2007.

[32] Stéphane Genaud and Choopan Rattanapoka. Fault management in p2p-mpi. In *Proceedings of International Conference on Grid and Pervasive Computing, GPC'07*, Lecture Notes in Computer Science, pages 64–77. Springer, May 2007.

[33] Stéphane Genaud and Choopan Rattanapoka. Large-scale experiment of co-allocation strategies for peer-to-peer supercomputing in p2p-mpi. In *5th High Performance Grid Computing International Workshop, IPDPS conference proceedings*. IEEE, April 2008.

[34] Stéphane Genaud and Choopan Rattanapoka. Evaluation of replication and fault detection in P2P-MPI. In *6th High Performance Grid Computing International Workshop, IPDPS conference proceedings*. IEEE, May 2009.

[35] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random walks in peer-to-peer networks. In *INFOCOM*, 2004.

[36] 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.

[37] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation Series. MIT Press, 2nd edition edition, 1999.

[38] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, 1994.

[39] Emmanuel Jeanvoine, Christine Morin, and Daniel Leprince. Vigne : Executing easily and efficiently a wide range of distributed applications in grids. In *Proceedings of Euro-Par 2007*, pages 394–403, 2007.

[40] Binu K. Johnson, R. Karthikeyan, and D. Janaki Ram. Dp : A paradigm for anonymous remote computation and communication for cluster computing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10) :1052–1065, 2001.

[41] N. Karonis, B. Toonen, and Ian Foster. MPICH-G2 : A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5) :551–563, 2003.

[42] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

[43] Soulla Louca, Neophytos Neophytou, Arianos Lachanas, and Paraskevas Evripidou. MPI-FT : Portable fault tolerenace scheme for MPI. In *Parallel Processing Letters*, volume 10, pages 371–382. World Scientific Publishing Company, 2000.

[44] Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Ryousei Takano, and Yutaka Ishikawa. Efficient mpi collective operations for clusters in long-and-fast networks. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona, Spain*. IEEE, 2006.

[45] Daniel Nurmi, John Brevik, and Richard Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 432–441. Springer, 2005.

[46] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Scalable wide-area resource discovery. Technical Report Technical Report CSD04-1334, Berkeley, CA, USA, University of California, 2004.

[47] Jelena Pjesivac-Grbovic, George Bosilca, Graham E. Fagg, Thara Angskun, and Jack Dongarra. Mpi collective algorithm selection and quadtree encoding. *Parallel Computing*, 33(9) :613–623, 2007.

[48] Rolf Rabenseifner and Jesper Larsson Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 36–46. Springer, 2004.

[49] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Brief announcement : Prefix hash tree. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, July 2004.

[50] Sridharan Ranganathan, Alan D. George, Robert W. Todd, and Matthew C. Chidester. Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing*, 4(3) :197–209, 2001.

[51] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, University of Berkeley, Berkeley, CA, 2000.

[52] Choopan Rattanapoka. *P2P-MPI : A Fault-tolerant Mesasges Passing Interface Implementation for Grids*. PhD thesis, University Louis Pasteur, Strasbourg, April 2008.

[53] M. Venkateswara Reddy, A. Vijay Srinivas, Tarun Gopinath, and D. Janakiram. Vishwa : A reconfigurable p2p middleware for grid computations. In *International Conference on Parallel Processing (ICPP 2006), 14-18 August 2006, Columbus, Ohio, USA*, pages 381–390. IEEE Computer Society, 2006.

[54] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *ATEC'04 : Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 127–140, Berkeley, CA, USA, 2004. USENIX Association.

[55] Louis Rilling. Vigne : Towards a self-healing grid operating system. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings*, pages 437–447, 2006.

[56] Antony I. T. Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[57] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework : System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4) :479–493, Winter 2005.

[58] Fred. B. Schneider. *Replication Management Using the State Machine Approach*, chapter 7, pages 169–195. ACM Press, 1993.

[59] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig A. Lee, and Henri Casanova. Overview of gridrpc : A remote procedure call api for grid computing. In *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, volume 2536 of *Lecture Notes in Computer Science*, pages 274–278. Springer, 2002.

[60] Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. P3 : P2P-based middleware enabling transfer and aggregation of computational resource. In *5th Intl. Workshop on Global and Peer-to-Peer Computing, in conjunc. with CCGrid05*, pages 259–266. IEEE, May 2005.

[61] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI : The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[62] Georg Stellner. CoCheck : Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, pages 526–531, 1996.

[63] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[64] Domenico Talia and Paolo Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7(4) :94–96, 2003.

[65] Rajeev Thakur, R. Rabenseifner, and William Gropp. Optimization of collective communication operation in mpich. *International Journal of High Performance Computing Applications*, 19(1) :49–66, February 2005.

[66] Niklas Therning and Lars Bengtsson. Jalapeno : secentralized grid computing using peer-to-peer technology. In *Proceedings of the Second Conference on Computing Frontiers, 2005, Ischia, Italy, May 4-6, 2005*, pages 59–65. ACM, 2005.

[67] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz and Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul and Bill Yeager. Project jxta 2.0 super-peer virtual network, May 2003. http ://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf.

[68] Peter Triantafillou and Theoni Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In *Databases, Information Systems, and Peer-to-Peer Computing, First International Workshop, DBISP2P, Berlin Germany, September 7-8, 2003, Revised Papers*, pages 169–183, 2003.

[69] Rob van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Satin : Efficient parallel divide-and-conquer in java. In *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings*, volume 1900 of *Lecture Notes in Computer Science*, pages 690–699. Springer, 2000.

[70] Rob van Nieuwpoort, Jason Maassen, Rutger F. H. Hofman, Thilo Kielmann, and Henri E. Bal. Ibis : an efficient java-based grid programming environment. In José E. Moreira, Geoffrey Fox, and Vladimir Getov, editors, *Java Grande*, pages 18–27. ACM, 2002.

[71] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Middleware '98*, page 55, 1998.

[72] Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, and Ilya Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, pages 1–12. Springer, 2002.

[73] Zhichen Xu, Mallik Mahalingam, and Magnus Karlsson. Turning heterogeneity into an advantage in overlay routing. In *INFOCOM*. IEEE Computer and communication Societies, 2003.

[74] Ben Y. Zhao, Yitao Duan, Ling Huang, Anthony D. Joseph, and John Kubiatowicz. Brocade : Landmark routing on overlay networks. In *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, volume 2429 of *Lecture Notes in Computer Science*, pages 34–44. Springer, 2002.

[75] Ben Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry : An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Berkeley, University of California, April 2001.

Chapitre 5

Application de Boosting

5.1 Contexte

Ce chapitre expose le travail de parallélisation d'une application d'apprentissage automatique (machine learning) mené entre 2006 et 2008 en collaboration avec Virginie Galtier et Stéphane Vialle à Supélec, campus de Metz. L'application a initialement été proposée par Olivier Pietquin, également en poste à Supélec. Une version séquentielle de l'application a ensuite été développée par Virginie Galtier, puis elle a été parallélisée [8] en utilisant JavaSpace [4]. Ce modèle de programmation, utilisé pour la première fois dans le langage Linda [9], est plus abstrait que le modèle à passage de messages, car les communication ne sont pas explicites. Les communications se font à travers une mémoire distribuée virtuelle[1] les processus ne peuvent qu'écrire, lire ou prendre (consommer) une valeur posée sur cet espace mémoire commun. Le regain d'intérêt pour ce modèle de programmation vient sans doute en partie des technologies Java et Jini utilisées, permettant de faire évoluer assez simplement des programmes Java existants vers une version répartie.

Pour comparer les deux modèles de programmation sur cette application, en terme d'expressivité et de performance des exécutions, j'ai réalisé une parallélisation à l'identique en passage de messages, avec MPJ [2]. La partie la plus importante du travail a ensuite consisté à évaluer les différences entre ces deux versions. Nous les avons testées en particulier sur des clusters et sur Grid'5000 [1].

5.2 Introduction

Les méthodes de boosting constituent une famille d'algorithmes d'apprentissage automatique qui construisent des modèles de classification fondés sur la combinaison d'apprenants dits "faibles". L'algorithme de boosting le plus utilisé s'appelle AdaBoost [5]. C'est un algorithme itératif, qui à chaque itération sélectionne le classifieur faible qui minimise l'erreur de classification. Un classifieur faible est habituellement un apprenant qui prend une décision rapidement et dont on attend un taux de réussite d'un peu plus de 50% sur un jeu de données d'apprentissage. Pour améliorer la performance de la sélection des classifieurs faibles, les exemples d'apprentissage sont affectés d'un poid qui change d'une itération à l'autre. L'erreur d'apprentissage d'un classifieur faible à une itération donnée est la somme pondérée des exemples mal classés. Le classifieur ayant la plus petite erreur d'apprentissage est sélectionné, et les poids des exemples qu'il a mal classé sont augmentés. De cette façon, l'algorithme avantage les classifieurs qui répondent avec succès sur les exemples précédemment mal classés. L'algorithme termine quand le classifieur fort, qui est une combinaison

[1] Aussi appelée dans la littérature, tuple space [9], blackboard, ou ici JavaSpace.

linéaire des classifieurs faibles retenus au cours des itérations, obtient un taux de succès dépassant un seuil donné (typiquement 95%).

Le but de la parallélisation de l'application est de permettre à des chercheurs en traitement du signal de tester rapidement les performances de nouveaux classifieurs entrainés par l'algorithme Adaboost. L'application est générique dans la mesure où le classifieur est implanté par une classe spécialisée, interchangeable selon le domaine d'application.

Dans ce travail, l'application est la détection de visages "Viola-and-Jones" [15]. Cette application se caractérise par de très nombreux attributs très simples basés sur des filtres de Haar (réponses binaires ne nécessitant que l'addition des intensités de pixels voisins), qui sont utilisés pour détecter des visages en temps réel dans une séquence d'images. Pour la phase d'apprentissage, un très grand nombre de ces filtres simples (ici 134 736 filtres) sont calculés sur chacun des éléments du jeu de données (ici, 8 500 images 24x24-pixels, dont la moitié représentent des visages, l'autre moitié autre chose). Pour calculer le taux d'erreur d'un classifieur faible (un filtre), on l'applique sur chacune des images et on compte le nombre de fois où il se trompe. Le processus itératif d'apprentissage s'arrête quand le classifieur fort construit a une marge d'erreur inférieure à 3% sur ce jeu de données.

5.3 Parallélisation

De manière surprenante, les travaux de parallélisation d'Adaboost sont peux nombreux. L'obstacle habituellement pointé (par exemple par Margineantu et Dietterich [13]), est le calcul des poids associés aux données, qui dépend de l'itération précédente, et qui impose donc la séquentialité des itérations. L'objectif recherché dans les travaux connexes, n'est souvent pas la minimisation du temps de la phase d'apprentissage. Par exemple, dans [11], les auteurs proposent une méthode de boosting parallèle qui opère sur une base de données distribuée. L'objectif est de résoudre le problème de données qui ne peuvent pas tenir en mémoire, en distribuant le processus d'apprentissage sur des jeux de données éclatés. Un autre exemple ([12]) qui suit le même objectif, propose d'entrainer plusieurs apprenants de base en même temps. Ce travail propose des méthodes heuristiques pour trouver un ensemble de distributions statistiques qui peuvent être générées de manière indépendante avant le processus d'apprentissage.

Quant à l'accélération de la phase d'apprentissage, un article récent [14] tente de tirer le maximum de parallélisme en contournant la dépendance d'une itération sur l'autre. Pour cela, le calcul d'une itération utilise une estimation du poids des exemples. Un modèle de l'évolution de la distribution des poids des exemples est d'abord calculé pendant plusieurs itérations séquentielles. Les calculs sont ensuite menés en parallèle, et au passage à l'itération suivante, les poids des exemples sont générés à partir du modèle. Cependant, il est difficile de déterminer le nombre d'itérations séquentielles nécessaires pour que cet algorithme converge comme AdaBoost.

La parallélisation que nous proposons ne modifie pas les hypothèses de l'algorithme Adaboost séquentiel, mais montre que sur l'application à la détection de visages, nous obtenons des accélérations très satisfaisantes sur un cluster en raison du parallélisme massif que l'on peut extraire. En considérant que chaque processeur peut avoir accès à la base des images, l'application de chaque filtre sur une image peut être calculée de manière indépendante.

Pour exploiter cette propriété, la parallélisation en JavaSpace suit le schéma suivant. Un processus maître coordonne la répartition des calculs des filtres sur des processus esclaves : il écrit dans le JavaSpace un ensemble d'identifiants de filtres qui doivent être traités, et les esclaves viennent consommer ces ensembles d'identifiants dès qu'ils sont inactifs. Quand un esclave a pris un ensemble, il en calcule tous les filtres. Comme au chapitre 2, le calcul de chacun des filtres ne prend pas exactement le même temps, mais le temps moyen de traitement d'un ensemble de filtres varie de manière non significative.

Homogène Dans la version initiale du programme destinée à un cluster, avec P processus esclaves ayant des CPUs homogènes, le maître divise simplement les N filtres en P ensembles contenant N/P filtres (et affecte les filtres restants au dernier esclave). Il faut ensuite déterminer lequel de ces filtres se trompe le moins souvent. Ceci revient à réaliser une *élection distribuée*. Avec JavaSpace, les esclaves ne pouvant communiquer entre eux, l'algorithme consiste à déterminer sur chaque esclave le minimum local, à centraliser les minima locaux sur le maître, puis diffuser à partir du maître le minimum des minima locaux. Ce filtre qui produit la plus petite erreur est intégré par le maître au classifieur fort, et l'esclave qui le possède le supprime de son ensemble de filtres. Ensuite les poids des exemples sont mis à jour, et une nouvelle itération débute avec les filtres restants.

Le passage d'une itération à l'autre impose donc une double synchronisation, car le maître doit avoir reçu tous les minima locaux pour décider du minimum à diffuser, et les esclaves attendent tous cette information du maître pour enchaîner l'itération suivante. Deux difficultés apparaissent donc clairement : un mauvais équilibrage de charge des esclaves provoquerait de l'inactivité à chaque fin d'itération, et la diffusion du minimum global par le seul maître peut devenir un goulot d'étranglement. Pour essayer de comprendre si une telle application programmée en JavaSpace a une extensibilité limitée, une version utilisant MPJ à été développée. De plus, nous souhaitons savoir comment ces deux versions se comportent dans un environnement distribué hétérogène à large échelle géographique.

Hétérogène Pour gérer l'hétérogénéité, nous adoptons un équilibrage de charge dynamique dans la première itération. L'utilisateur règle la granularité de l'équilibrage de charge en fixant la taille S ($S \leq N/P$) des ensembles de filtres distribués par le maître. Selon le même principe que précédemment (dès qu'un processeur devient inactif, il réclame un nouvel ensemble à calculer), ceci permet aux esclaves les plus rapides de calculer plus d'ensemble de filtres que les esclaves les plus lents.

Par rapport à l'application présentée p. 18, la différence est la forme itérative de l'algorithme. A la fin de la première itération, chaque esclave possède un nombre différent de filtres. Cette distribution est ensuite conservée durant toutes les itérations suivantes. Cette stratégie peut être mise en défaut par des surcharges ponctuelles apparaissant sur certains processeurs après la première itération. Cependant, nous ne considérons pas cette éventualité car l'utilisateur dispose de ressources hétérogènes mais en accès exclusif, et la brieveté des exécutions (quelques minutes) réduit cette probabilité.

5.4 Différences entre JavaSpace et MPJ

Pour une comparaison équitable, la version MPJ a été écrite en suivant la même structure de programme que la version JavaSpace. Il y a un processus maître qui a la fonction exclusive de distribuer les données et collecter les résultats.

En JavaSpace, le maître communique aux esclaves le travail à faire en écrivant une valeur typée dans le JavaSpace. Simultanément, tous les esclaves sont en attente bloquante d'une valeur de ce type. Dès qu'un appariement est possible, il est réalisé, ce qui rend l'affectation des valeurs aux esclaves non déterministe et efficace (premier esclave prêt, premier servi). On peut traduire cette affectation non-déterministe en MPJ, en programmant le maître pour qu'il opère des réceptions asynchrones acceptant des requêtes de n'importe quel esclave (`MPI_ANY_SOURCE`). Le maître peut déterminer l'origine d'une demande de travail et y répondre immédiatement.

Concernant la diffusion d'une information (le maître informe tous les esclaves du meilleur filtre globalement), on procède également en JavaSpace par l'écriture de cette valeur dans le JavaSpace, qui est lue par tous les esclaves bloqués en attente de cette valeur. En MPJ, on peut préciser sémantiquement qu'on fait une diffusion, par l'utilisation d'une primitive spécifique (`MPI_Bcast`). Ceci

permet à l'implantation sous-jacente d'optimiser les envois, par exemple avec un arbre binomial sur un réseau à faible latence.

Le code JavaSpace est légèrement plus concis, et plus abstrait que son pendant MPJ. Par exemple, l'esclave dans ses communications avec le maître, peut recevoir deux types de réponses : il s'agit soit de nouveaux filtres à calculer, soit du meilleur classifieur de l'itération. En JavaSpace, ceci s'exprime naturellement : le maître écrit tous les filtres à calculer dans le JavaSpace et les esclaves consomment les valeurs de ce type tant qu'il en existe (`takeIfExists`). Ensuite, ils se mettent en attente du meilleur classifieur. En MPJ, la communication entre le maître et les esclaves se fait sans stockage intermédiaire : le maître envoie du travail au fur et à mesure qu'il reçoit des demandes. Ainsi, l'esclave ne sait pas quand il reçoit un message, si le maître a encore du travail à fournir, ou si le message contient à la place le meilleur classifieur de l'itération. Il faut donc différencier le type de message à l'aide de tags distincts. Ceci oblige l'opération de réception à attendre en surveillant n'importe quel tag, puis a se brancher sur des traitements différents selon les cas. Ce code est naturellement moins lisible que celui en JavaSpace.

5.5 Évaluation

L'objectif de l'évaluation est de comprendre quelles sont les limites à l'extensibilité d'une telle application, et quels environnements sont envisageables. L'implémentation de JavaSpace utilisée est celle du Jini Starter Kit de Sun, baptisée *outrigger*, et l'implémenation MPJ est P2P-MPI.

L'extensibilité de l'application est d'abord testée sur un cluster (nœuds bi-core Xeon-3075 2.66GHz, 4GB RAM) avec 64, 128 et 256 processus. La figure 5.1 montre l'accélération par rapport au temps séquentiel (11598s, plus de 3h), obtenue avec la version homogène, et avec la version hétérogène utilisant la granularité qui donne le meilleur temps. Avec des accélérations entre 174 et plus de 200 pour 256 cœurs utilisés, les deux environnements s'avèrent être tout à fait satisfaisants pour traiter ce problème sur un cluster.

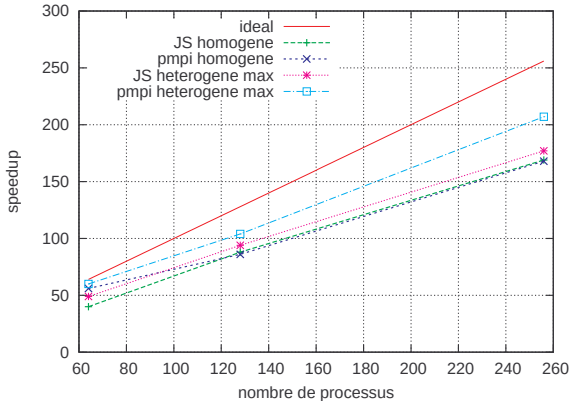


FIG. 5.1 – Accélérations sur un cluster

Ensuite, nous essayons de mettre en évidence les effets liés à la latence réseau, en divisant les processus de calcul en trois parties sur des sites géographiques différents. Nous utilisons à nouveau la plate-forme Grid'5000 [1], et nous plaçons 40 processus esclaves sur des clusters ayant des noeuds (presque) identiques, à Sophia, Nancy et Rennes (Figure 5.2 gauche). Le maître (et le service JavaSpace quand c'est une exécution JavaSpace) est sur un quatrième site (Lyon). Les RTT (deux fois la latence) des clusters de calcul vers le site maître sont respectivement 6.8, 10.2 et 12ms. Tous les sites

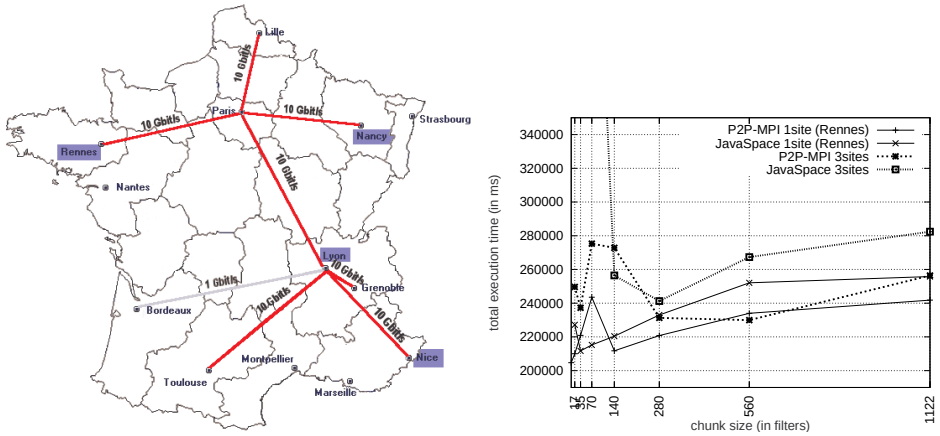


FIG. 5.2 – Temps d'éxecution sur Grid'5000 trois sites

sont reliés en 10 Gbps.

Les courbes de droite de la figure 5.2 montrent les temps d'exécution pour JavaSpace et P2P-MPI en fonction de la granularité d'équilibrage choisie. La granularité la plus grosse correspond à un découpage du jeu de données en autant de blocs qu'il y a d'esclaves. Les courbes en pointillés épais donnent les temps pour l'exécution multi-sites. Pour comparaison, on trace aussi la performance obtenue sur un seul de ces clusters avec 120 esclaves (lignes pleines).

On observe plusieurs éléments intéressants. D'abord, de manière attendue, le choix de la granularité influe beaucoup plus la performance finale dans l'exécution multi-sites que sur un seul cluster. Cependant, la deuxième observation est que presque tous les temps d'exécution sont comparables du point de vue de l'utilisateur, entre 4 et 5 mn. Il y a une exception, quand la granularité de l'exécution multi-sites avec JavaSpace devient trop faible, sur laquelle nous allons revenir. La conclusion est que l'application est plutôt bien adaptée à un environnement avec des ressources de calcul distribuées, car la granularité qui donne des résultats d'équilibrage acceptables, est assez facile à régler. Les communications sont peu volumineuses, principalement composées des petits messages qui contiennent l'identité d'un meilleur classifieur.

Cependant, on observe qu'un mauvais choix de granularité peut amener l'effondrement des performances, comme c'est le pour des blocs de taille inférieure à 70 filtres dans l'exécution JavaSpace. Dans ce cas, le temps d'exécution est de 550s, soit environ le double des autres temps. Pour comprendre la raison d'un tel comportement, il faut examiner l'activité individuelle des esclaves.

Dans les graphes des figures 5.3 et 5.4, on trace le temps que chaque esclave (identifié par un numéro sur l'axe des abscisses) a passé à calculer (barres vertes, zone du bas) et à communiquer (barres bleues, zone supérieure). Ces temps sont à lire sur l'échelle verticale à gauche. On superpose le nombre de filtres affectés à chaque esclave, représenté par la courbe rouge (échelle verticale à droite).

Avec ces mesures, nous pouvons définir un indicateur pour caractériser l'équilibrage de charge. Cet indicateur est le ratio du temps de calcul le plus long sur la moyenne des temps de calculs.

L'exécution sur un seul cluster, vue sous cette angle, est montrée figure 5.3. Les indicateurs montrent des équilibrages qui sont acceptables, avec $1,11$ et $1,23$ pour P2P-MPI et JavaSpace respectivement. P2P-MPI passe moins de temps à attendre que JavaSpace, qui est désavantagé par son

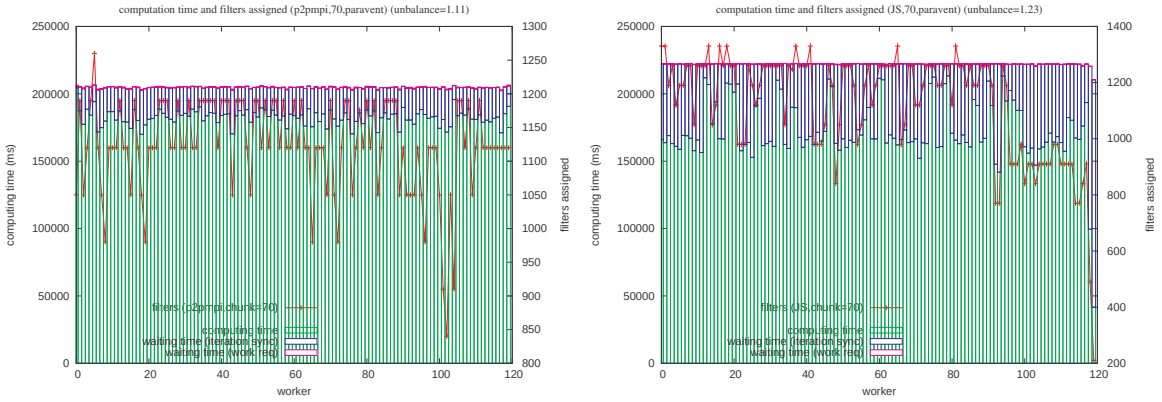


FIG. 5.3 – Comportement comparé sur un cluster, granularité=70 filtres

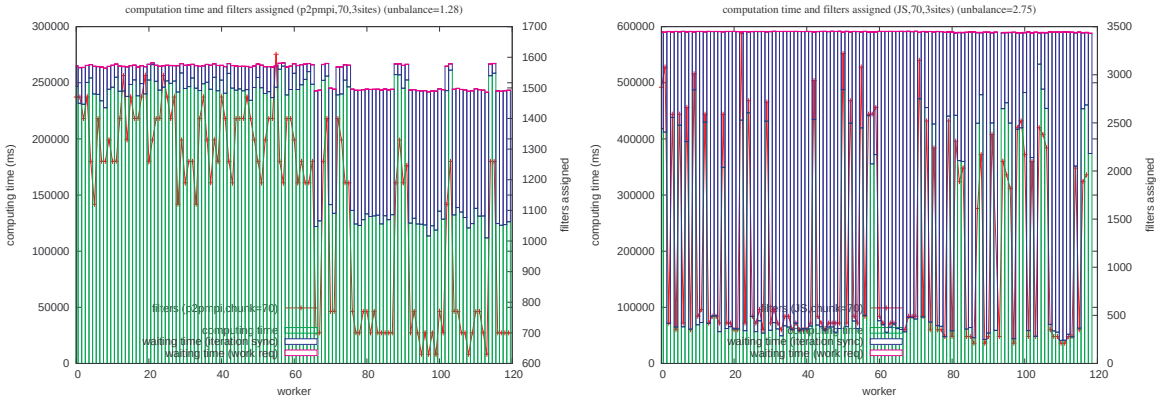


FIG. 5.4 – Comportement comparé en multi-sites, granularité=70 filtres

dernier processus qui est étrangement sous-chargé[2]. En revanche, dans l'exécution multi-sites présentée sur la figure 5.4, on observe un déséquilibre extrêmement important pour JavaSpace, avec un indicateur de 2,75. Certains esclaves n'ont reçu que 210 filtres à calculer par exemple, tandis que d'autres en ont eu plus de 3 000. Par comparaison, l'exécution avec P2P-MPI avec une telle granularité donne aussi les moins bonnes performances, mais l'équilibre de charge reste à 1,28.

Notre explication de ce cas pathologique est que les esclaves accèdent au JavaSpace de manière non équitable quand il y a trop de communications longue distance. L'inéquité est une conséquence de la contention des communications : les communications initiées avec des RTT les plus faibles (les esclaves à Sophia) sont plus réactifs. Sur la figure, le dernier tiers des esclaves est localisé à Sophia, dont le RTT au site maître est presque deux fois inférieur aux autres sites. Les esclaves localisés à Sophia obtiennent effectivement plus de travail que les esclaves des autres sites. La conséquence est que beaucoup de ces processeurs sont surchargés (certains calculent pendant environ 500s) tandis que d'autres sont en famine.

Bien que le schéma de communication soit le même dans le programme Javaspace ou MPJ, à savoir tous les esclaves vers le maître, ou le maître vers tous les esclaves, la gestion des communication diffère dans les deux environnments. JavaSpace utilise RMI, et créé un thread de service par communication. En revanche, P2P-MPI utilise la classe Java `nio`, qui fournit l'équivalent de l'instruction

[2]Ce phénomène a été observé plusieurs fois dans des conditions variées.

`select` de la *libc*. Cette classe fournit un mécanisme pour détecter des évènements survenant sur un ensemble de descripteurs. La gestion de l'évènement n'est pas bloquante pour les entrés/sorties en cours, ce qui permet de ne pas créer de threads pour le traitement des évènements.

5.6 Conclusion et perspectives

Ce travail avait pour objectif de révéler les différences remarquables entre deux modèles de programmation. Nous avons réalisé la première étape, dans laquelle nous avons comparé une version MPJ calquée sur la version JavaSpace. Dans cette version, le seul avantage de MPJ est de pouvoir exprimer sémantiquement une diffusion du maître vers les esclaves. Cette distinction permet d'optimiser cette opération collective (mais qui n'est efficace que dans un environnement avec faible latence [10]).

Une deuxième étape du travail devrait être la ré-écriture partielle de la version MPJ en utilisant la plus grande expressivité du modèle des communications. En effet, sans la contrainte de ne pas pouvoir faire communiquer les esclaves entre eux, on mettrait en place en MPJ des communications évitant le goulot d'étranglement du maître. Par exemple, déterminer quel esclave détient le meilleur classifieur faible à une itération revient à un problème d'élection de leader dans un système distribué asynchrone. Pour résoudre le problème, on pourrait simplement utiliser la primitive de communication collective `MPI_Allreduce`. Si l'implémentation de cette primitive n'est pas optimisée, ou si son implémentation ne convient pas à la topologie du réseau utilisé, on peut coder explicitement un des algorithmes bien connus d'élection de leader, sur un anneau logique [3] ou sur un graphe quelconque [6].

5.7 Résumé des contributions

Nous avons parallélisé l'algorithme AdaBoost, appliqué à la détection de visages. La parallélisation destinée initialement à un environnement d'exécution homogène (cluster) à été étendue pour s'adapter à l'hétérogène, en mettant en place un équilibrage de charge dans la première itération. Nous avons proposé cette parallélisation dans deux modèles de programmation différents : JavaSpace et MPJ. Ces deux versions ont été testées avec les implémentations *outrigger* pour Javaspace et P2P-MPI pour MPJ. L'évaluation faite sur cluster montre l'extensibilité de l'application parallèle jusqu'à 256 processus dans les deux modèles de programmation. L'équilibrage de charge est satisfaisant sur un cluster ou l'hétérogénéité des processeurs est émulée. L'évaluation faite sur Grid'5000 établit les limites à ne pas dépasser dans la finesse de la granularité utilisée pour l'équilibrage de charge. Cette évaluation démontre aussi la faisabilité d'utiliser de l'ordre d'une centaine de processus répartis sur quatre sites géographiques pour cette application. Ces résultats sont publiés dans l'article [7].

Bibliographie

[1] Franck Cappello et al. Grid'5000 : a large scale and highly reconfigurable grid experimental testbed. In *6th IEEE/ACM International Conference on Grid Computing (GRID 2005)*, pages 99–106, 2005.

[2] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPJ : MPI-like message passing for java. *Concurrency : Practice and Experience*, 12(11), September 2000.

[3] Ernest J. H. Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5) :281–283, 1979.

[4] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practive*. Pearson Education, 1999.

[5] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the International Conference on Machine Learning*, pages 148–156, July 1996.

[6] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1) :66–77, 1983.

[7] Virginie Galtier, Stéphane Genaud, and Stéphane Vialle. Implementation of the adaboost algorithm for large scale distributed environments : Comparing javaspace and mpj. In *Fifteenth International Conference on Parallel and Distributed Systems (ICPADS'09)*. IEEE, December 2009.

[8] Virginie Galtier, Olivier Pietquin, and Stéphane Vialle. Adaboost parallelization on PC clusters with virtual shared memory for fast feature selection. In *IEEE International Conference on Signal Processing and Communication*, pages 165–168, November 2007.

[9] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, pages 80–112, January 1985.

[10] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe : MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8) :131–140, 1999.

[11] Aleksandar Lazarevic and Zoran Obradovic. The distributed boosting algorithm. In *7th ACM SIGKDD international conference on Knowledge discovery*, pages 311–316. ACM Press, 2001.

[12] Fernando Lozano and Pedro Rangel. Algorithms for Parallel Boosting. In *Proceedings of the 4th international conference on Machine Learning and Applications (ICMLA'05)*. IEEE Press, 2005.

[13] Dragos D. Margineantu and Thomas G. Dietterich. Pruning adaptive boosting. In *ICML '97 : Proceedings of the Fourteenth International Conference on Machine Learning*, pages 211–218, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[14] Stefano Merler, Bruno Caprile, and Cesare Furlanello. Parallelizing adaboost by weights dynamics. *Computational Statistics & Data Analysis*, 51(5) :2487 – 2498, 2007.

[15] P. Viola and M. Jones. Robust real-time object detection. In *2nd International Workshop On Statistical And Computational Theories Of Vision Modeling, Learning, Computing, And Sampling*, July 2001.

Chapitre 6

Application de Clustering

6.1 Contexte

Nous présentons ici un travail mené en collaboration avec Pierre Gançarski et Alexandre Blansché de l'équipe de fouille de données du LSIIT[1]. Nos collègues ont développé une méthode de classification non-supervisée, implémentée au sein d'une plateforme dédiée à la classification d'images de télédétection. Le langage de développement pour la plateforme est Java. Pour accélérer les temps de réponse de la nouvelle méthode, Guillaume Latu et moi même avons entamé un travail de parallélisation de la méthode. L'étude puis l'implémentation, qui ont exploré différentes modes de parallélisation, partant d'une approche par threads jusqu'à la parallélisation présentée dans ce chapitre, a duré un an et demi entre 2005 et 2006. La collaboration s'est articulée autour d'un étudiant de master, Damien Vouriot, qui tout au long de l'année scolaire, puis de son stage, a finalisé la programmation des versions parallèles de l'application.

6.2 Introduction

Nous nous intéressons ici à une méthode de classification *non-supervisée*, aussi appelée *clustering*. C'est un problème d'optimisation dont l'objectif est de partitionner un ensemble de données, de telle sorte que les données de chaque sous-ensemble soient *similaires* entre elles. La mesure de similarité est souvent définie par une distance. Un algorithme de clustering n'a besoin d'aucune information *à priori* sur les données à classifier, à part (dans le plupart des approches) le nombre de sous-ensembles désirés. Un panorama complet des méthodes de classification non-supervisées est présenté dans [2, 20]. Une illustration du clustering appliqué à des images est présentée sur la figure 6.1. La partie gauche est une image satellite à classer, et la droite représente quatre images associées aux quatre clusters extraits. Un expert peut facilement identifier la nature des éléments (ou encore les *objets*, ici des bâtiments et routes, de la végétation, de la terre, de l'eau) que la classificiation a établi selon les valeurs spectrales des pixels.

Dans la pratique, il est très fréquent que l'exploitation des applications de classification basées sur des algorithmes génétiques ne donne des résultats qu'après plusieurs heures de calcul. La parallélisation des algorithmes utilisés semble donc une idée naturelle pour les accélérer. Plusieurs parallélisations ad-hoc ont été proposées [7, 9, 16] pour des méthodes classiques de clustering (par exemple *K*-means). Ces parallélisations sont basées sur du passage de messages implémentées avec MPI [1].

[1] Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection, UMR 7005 CNRS - Université de Strasbourg.

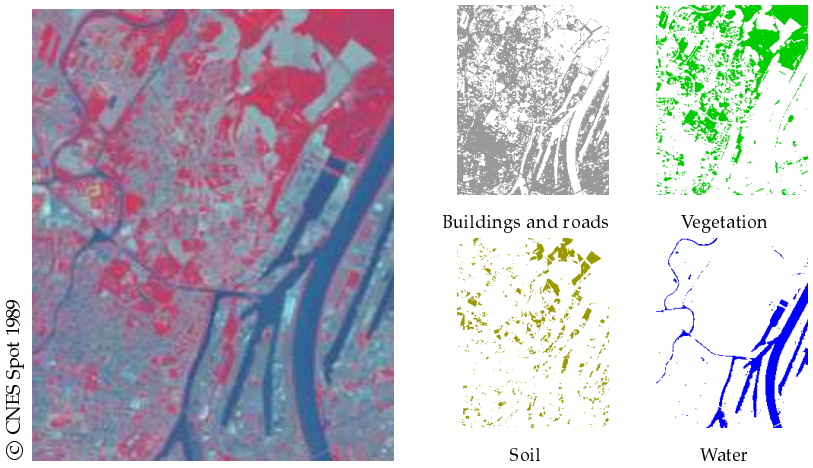


FIG. 6.1 – Image de télédétection de Strasbourg : données SPOT 4 data avec trois canaux en résolution standard (200×250 pixels - 20 mètres/pixel) (à gauche) et les quatre clusters extraits (à droite).

Dans ces études, les tests de performances menés sur des machines parallèles ou clusters montrent qu'on peut obtenir des accélérations quasiment linéaires.

Il est donc intéressant de comprendre si de tels gains peuvent être obtenus sur des méthodes plus complexes, comme celle à laquelle nous allons nous intéresser ici, qui sont basées sur la *pondération des attributs*.

Aujourd'hui, on souhaite traiter des objets décrits par un nombre croissant d'attributs (par exemple dans le domaine de l'imagerie, le nombre important de bandes de fréquence dans une image hyper-spectrale), qui peuvent pour certains être non-pertinents, bruités ou corrélés avec d'autres. Une approche possible pour faire face à ce problème est de pondérer chaque attribut pour juger de son importance. La pondération automatique des attributs est un problème en soi. Utiliser un vecteur global de poids pour les attributs n'est pas satisfaisant car la pertinence d'un attribut peut varier selon les classes extraites [14, 15, 18]. Des méthodes proposées récemment utilisent donc un vecteur de poids local à chaque cluster ([10, 8]).

Nos collègues de l'équipe de fouille de données, ont proposé une méthode de clustering originale baptisée MACLAW (a Modular Approach for Clustering With Local Attributes Weighting) [3]. Elle vise également à construire des vecteurs de poids locaux pertinents. C'est une méthode itérative qui optimise les vecteurs de poids locaux individuellement à l'aide d'une heuristique adaptée (par exemple K-means ou EM) et qui évalue ensuite le résultat global, c'est-à-dire, la classification obtenue si on utilise ces poids locaux, et itère jusqu'à ce que le résultat global atteigne le seuil de satisfaction. D'autres travaux suivent le même schéma, en utilisant l'algorithme du K-means pondéré pour calculer les poids locaux mais utilisent un heuristique de type hill-climbing pour progresser d'une itération à l'autre [6].

La méthode MACLAW La méthode MACLAW, elle, utilise une approche évolutionnaire pour explorer l'espace des solutions. Dans cette approche itérative, on associe une population d'individus pour les cluster que l'on souhaite extraire (supposons qu'il y en a K), et on fait évoluer indépendamment ces populations (co-évolution). Chaque itération comporte trois phases.

Phase 1 Chaque individu porte un vecteur de poids locaux et procède à une extraction de cluster paramétrée par ses poids. La méthode utilisée ici est un K-means pondéré mais d'autres

pourraient être choisies. Toutes les extractions, dans tous les populations, se font de manière indépendante à une itération. Dans chaque population $k(1 \leq k \leq K)$, on retient l'individu I_k^g (à l'itération g) qui produit le cluster de meilleure qualité. La qualité est le produit de deux critères. Le premier critère est l'*inertie* interne du cluster, c'est-à-dire la somme des distances pondérées des objets appartenant au cluster jusqu'au centre du cluster. Le deuxième critère indique la complémentarité du cluster extrait avec les clusters extraits dans les autres populations à l'étape précédente (une partition des données donnera la qualité maximale, tandis que que beaucoup d'objets communs à d'autres clusters ou des objets non classés feront baisser ce score).

Phase 2 La meilleure solution globale est déterminée en évaluant la qualité de chaque clustering que l'on peut former à partir des combinaisons des meilleurs individus de cette génération et de la précédente. Il faut évaluer la qualité (mêmes critères qu'à la phase 1) de chacun des clusters de l'ensemble $\{(I_1^g, I_2^g, \ldots, I_K^g), (I_1^{g-1}, I_2^g, \ldots, I_K^g), (I_1^g, I_2^{g-1}, \ldots, I_K^g), \ldots, (I_1^{g-1}, I_2^{g-1}, \ldots, I_K^{g-1})\}$.

Phase 3 La dernière phase est une étape de *reproduction* au sens des algorithmes génétiques, qui a pour but de diversifier les individus. La reproduction n'implique que les individus d'une même population (pas de croisement inter-population).

6.3 Parallélisation

L'analyse de complexité de la méthode dont nous avons schématisé le fonctionnement ci-dessus a mis en évidence une complexité élevée pour les phases 1 et 2. Asymptotiquement, la complexité de la phase 1 est en $O(K^3)$, celle de la phase 2 en $O(2^K)$, et celle de la phase 3 en $O(K)$.

Cette complexité élevée est acceptable au regard du nombre modéré de clusters en jeu dans les applications visées, avec de 5 à 20 clusters au maximum. Dans nos évaluations de l'application parallélisée, on nous a demandé de tester avec 4, 8 ou 16 clusters. L'enjeu de la parallélisation est ici de permettre des durées d'exécution qui sont inférieures à la demi-heure. Avec des durées d'exécution assez courtes, il est possible d'utiliser le logiciel en mode « essai-erreur », soit pour de l'exploitation, soit pour mettre au point la méthode elle même par l'essai de différents réglages.

Environnement utilisateur Dans cette collaboration, un souhait important de l'utilisateur était de conserver un environnement applicatif aussi proche que possible de celui qu'il est habitué à manipuler : il s'agit d'une interface graphique permettant de choisir les images, les algorithmes utilisés, les paramètres, etc. Pour accélérer les traitements tout en conservant cet environnement, le premier travail mené a été une parallélisation à base de threads, avec l'objectif d'utiliser une seule machine multi-coeurs. Ce travail a permis d'identifier les traitements indépendants. Les accélérations obtenues n'ont cepedant pas rempli les objectifs d'accélération fixés. Un deuxième travail a donc été de développer une version parallèle à base de passage de messages. Habituellement, passer à une application de ce type suppose d'abandonner l'environnement graphique conçu pour un ordinateur de bureau, car il faut se connecter à la frontale du cluster, transférer ses fichiers, soumettre ses tâches au gestionnaire de jobs, etc. Dans le cas où l'on tente de masquer ces manipulations en les automatisant, on hérite de scripts ad-hoc difficiles à maintenir. Avec P2P-MPI, nous pouvons au contraire conserver une bonne intégration à l'environnement (Java) existant : quand l'utilisateur sélectionne le mode d'exécution parallèle, la découverte des ressources disponibles se fait directement à partir de la machine de l'utilisateur, les programmes et fichiers sont transférés automatiquement, et le résultat final est affiché graphiquement exactement comme avec le mode d'exécution séquentielle.

Stratégie de parallélisation Comme nous l'avons décrit précédemment, dans la phase 1, chaque individu procède indépendamment à une extraction de cluster, ce qui correspond en l'occurrence à l'exécution d'un K-means. Par conséquent, la stratégie naturelle de parallélisation consiste à distribuer les individus avec les calculs associés sur les processeurs disponibles. Ce choix fait, il reste deux alternatives pour distribuer les individus sur les processeurs :

(i) Distribuer des populations entières à un processeur, avec l'idée de minimiser le coût des commmunications intra-population. En effet, à la fin de la phase 1, il faut déterminer le meilleur individu d'une population, ce qui demande de comparer les scores des individus d'une même population.

(ii) Distribuer les individus sans tenir compte de la population à laquelle ils appartiennent (une population peut être répartie sur plusieurs processeurs). L'avantage est alors de pouvoir équilibrer la charge de calcul à la granularité de l'individu, et non de la population.

La figure 6.2 schématise le fonctionnement de MACLAW parallélisée avec le choix de distribution que nous avons fait. Dans cette figure, on a six processeurs et quatre populations. La deuxième alternative donne beaucoup de flexibilité. En particulier, le nombre assez faible de clusters souhaités (inférieur à 20) limiterait le nombre de processeurs utilisables dans la première alternative. Le surcoût de communications à payer par rapport à la première alternative est dû à l'élection du meilleur individu par population : on élit un processeur référence pour chaque population, afin qu'il coordonne les élections partielles nécessaires pour trouver le meilleur individu de la population (Phases 1(b) et 1(c)) sur la figure 6.2).

6.4 Évaluation

Nous avons d'abord mesuré les performances de l'application sur un cluster unique pour évaluer l'extensibilité et les accélérations obtenues par la parallélisation. Les résultats sont présentés sur la figure 6.3 pour K=4, 8 et 16 clusters, avec à gauche les temps de calcul, et à droite les accélérations. Le test correspond au traitement d'une image comme celle de la figure 6.1, avec les paramètres suivants : on a 20 individus par population, chaque pixel de l'image a trois composantes, l'image a 50000 pixels, et on itère sur 20 générations. Le temps de calcul ne dépend pas de l'image, mais seulement de ces paramètres, que nous avons choisis représentatifs de l'utilisation habituelle de la méthode.

Ces tests illustrent d'abord les temps d'exécution qu'on observe fréquemment : le traitement prend 24 minutes pour quatre clusters, 85 minutes pour huit clusters et plus de 8 heures pour seize clusters. Pour une telle instance de problème, on voit immédiatement l'intérêt pratique de la parallélisation, la durée d'exécution étant ramenée à 12 minutes avec 60 processeurs.

Concernant les performances dans l'ensemble de l'application parallèle, il faut distinguer selon le nombre de clusters cherchés. Pour quatre clusters, l'accélération est linéaire jusqu'à 16 processeurs, puis décroît lentement pour se stabiliser à 30-32 processeurs et chute brutalement après 48 processeurs. La courbe est semblable pour huit clusters, mis à part qu'on observe une accélération linéaire jusqu'à 24 processeurs, qui chute à 48. Pour 16 clusters, l'accélération est linéaire jusqu'à 60 processeurs, avec une accélération de 47 en utilisant 60 processeurs.

Ceci s'explique par la complexité de chacune des phases : pour K=4 et K=8, la phase 1 est celle qui domine largement, la phase 2 ne nécessitant que relativement peu de calculs pour ce nombre de clusters. Dans ces cas, l'extensibilité de l'application est principalement limitée par deux facteurs. D'une part, des déséquilibres dans les calculs apparaissent quand le nombre de processeurs augmente car on utilise peu d'individus (par exemple pour K=4, il y a 80 individus au total, ce qui

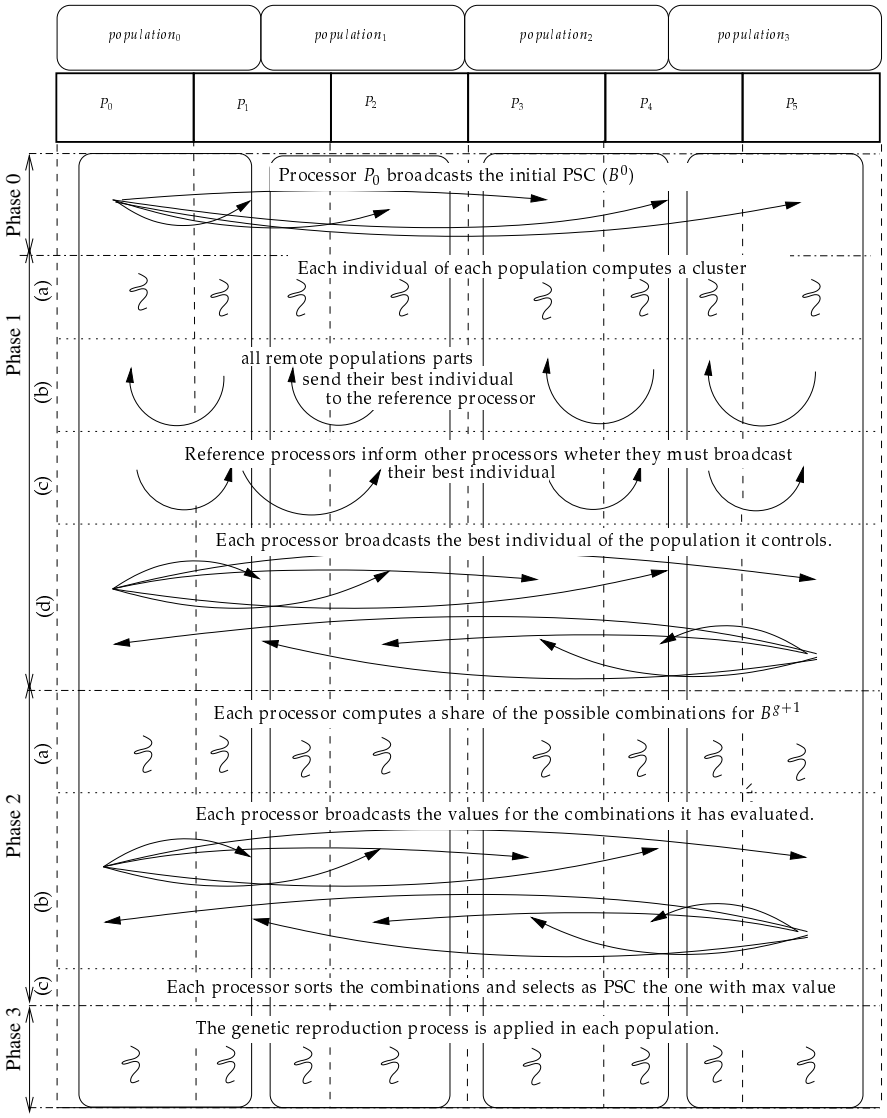


FIG. 6.2 – Execution de MACLAW parallèle avec 6 processus et 4 populations.

fait que si on utilise plus de 40 processeurs, la charge par processeur est soit de un, soit de deux individus). D'autre part, le nombre de communications est proportionnel avec le nombre de processus impliqués, et pour ces cas, il n'y aucune accélération dans la phase 2 car les communications et synchronisations (phase 2(b)) sont importantes par rapport au faible nombre de calculs. Cependant, comme la phase 1 qui prend la majeure partie du temps a une bonne accélération, on obtient au final des temps d'exécution acceptables : par exemple, 120 s avec 16 processeurs pour K=4, ou 246 s avec K=8 sont des bons compromis entre durée et ressources utilisées.

Pour K=16, les temps pris par les deux phases sont à peu près égaux car le nombre de calculs de la phase 2 qui augmente exponentiellement rattrape celui de la phase 1. Par conséquent, le surcoût des communications évoqué précédemment devient très petit au regard du coût des calculs, ce qui explique la linéarité des accélérations jusqu'à 60 processeurs. Pratiquement, le clustering sur de telles instances peut se faire dans des durées de l'ordre de la dizaine de minutes, alors qu'il faut

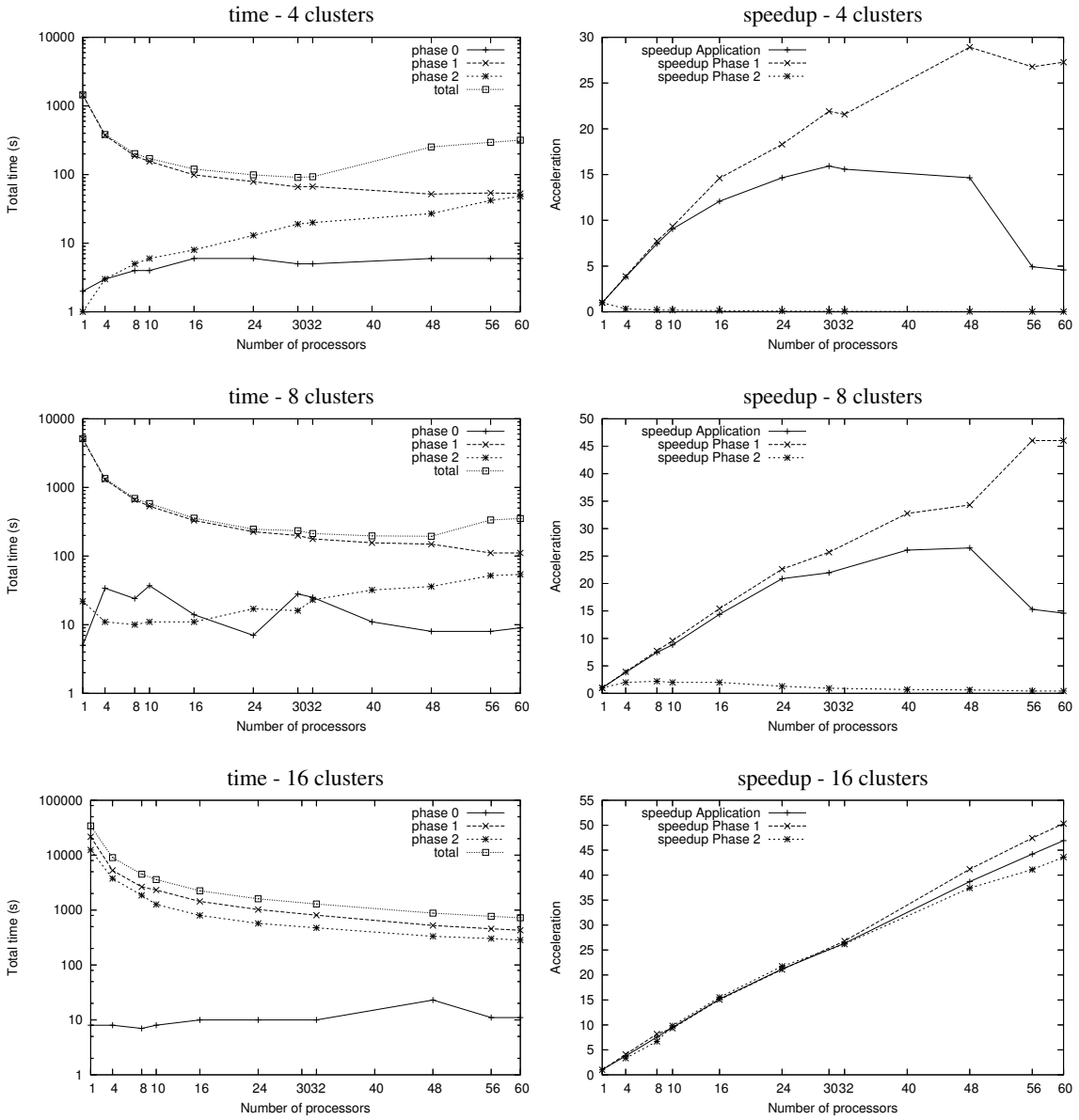


FIG. 6.3 – Performances de l'extraction de 4, 8 et 16 clusters.

plusieurs heures en séquentiel, ce qui rend la méthode utilisable dans bien plus de cas.

Ensuite, nous avons étendu l'étude à des configurations avec plusieurs clusters sur Grid'5000 [5]. Nous nous limitons au traitement de seize clusters, les cas avec quatre ou huit clusters ne permettant pas des accélérations intéressantes sur cet exemple au delà de vingt à trente processeurs. Ces tests multi-sites impliquent à chaque fois deux ou trois sites parmi les sites de Lille, Nancy, Sophia ou

Bordeaux. Nous combinons à cela aussi deux types de configurations : soit avec des processeurs tous identiques, soit avec trois types de processeurs[2].

Pour avoir un ordre de grandeur de la dégradation des performances, nous faisons la moyenne des temps de toutes les exécutions multi-sites. La moitié de ces exécutions utilisent des processeurs hétérogènes, l'autre moitié uniquement des processeurs identiques. Cette performance moyenne, ainsi que les écarts-types, sont présentés en figure 6.4, avec la performance sur un cluster unique pour comparaison.

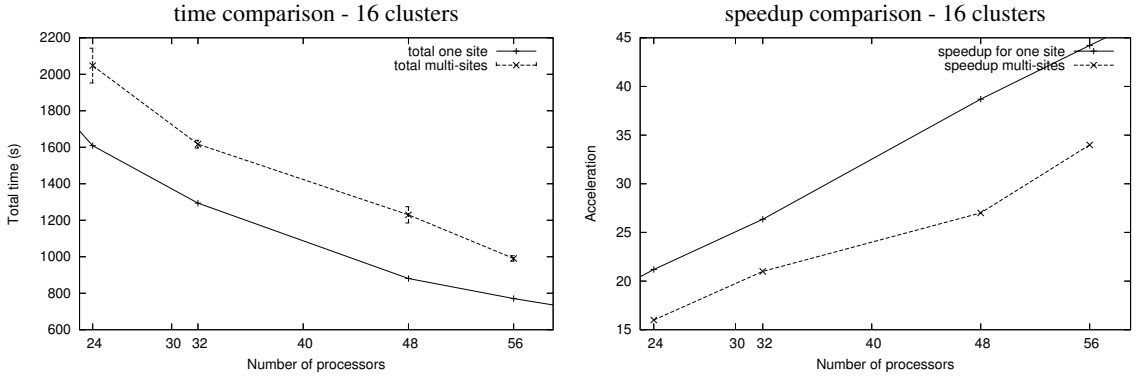


FIG. 6.4 – Performance pour K=16 sur un cluster et plusieurs clusters.

Il faut noter que le code parallèle n'intègre aucune gestion de l'hétérogénité des unités de calcul. Cette gestion de l'hétérogénité n'était pas prioritaire et a été laissée à un travail futur. En effet, les utilisateurs avaient décidé d'acheter des ressources de calcul homogènes dédiées aux applications de la plateforme, en espérant pouvoir bénéficier de machines supplémentaires de manière occasionnelle. Concernant l'influence de l'hétérogénéité, c'est le processeur le plus lent qui donne la cadence en raison de notre répartition de charge statique. De la même manière, les temps de synchronisation entre phases et itérations sont dépendants des performances des communications et l'utilisation de liens réseau longue-distance les allonge.

L'observation globale est que le passage d'une exécution sur un cluster unique à plusieurs clusters conduit ici à un surcoût d'environ 30%, assez constant quel que soit le nombre de processeurs utilisés. Ainsi, les accélérations obtenues en multi-sites suivent la courbe des accélérations sur un seul cluster à une constante près. En revanche, utiliser deux ou trois sites ne change pas les performances de l'application de manière significative. La moitié du surcoût est liée à l'utilisation des processeurs moins rapides, et l'autre moitié aux communications moins performantes. Nous retenons donc qu'utiliser un tel environnement, que l'on pourrait qualifier de "modérément hétérogène", reste compétitif (par exemple une accélération de 21 sur 32 processeurs) par rapport à l'application séquentielle.

6.5 Conclusion

Ce travail avait pour but d'accélérer l'application de clustering conçue par l'équipe de fouille de données, ainsi que par des géographes utilisateurs de cette application pour des images satel-

[2]des dual-core AMD Opteron 2.0 GHz (les processeurs utilisés sur un cluster unique, ou en multi-sites avec processeurs identiques), des Opteron 2.2GHz et des Intel Xeon EM64T 3GHz. Les Xeon EM64T 3GHz sont environ 25% moins rapides que les Opteron 2.0 GHz

litaires. La tentative de multi-threading de l'application a rapidement montré ses limites, et il a fallu paralléliser le code selon un paradigme de passage de messages pour utiliser des dizaines de processeurs en parallèle et obtenir des temps d'exécution réduits de manière significative. Cette évaluation montre de bonnes performances à la fois sur une architecture homogène (cluster), et sur des configurations variées utilisant des processeurs hétérogènes répartis sur deux ou trois sites éloignés géographiquement (Grid'5000).

Du point de vue de l'efficacité de la parallélisation, les accélérations mesurées sont très bonnes en raison de l'augmentation exponentielle des calculs avec le nombre de clusters à extraire. Du point de vue utilisateur, le traitement de certaines instances de problèmes (par exemple 16 clusters) redevient accessible, alors que le temps d'exécution avoisine les huit heures en séquentiel. L'autre avantage pour l'utilisateur est la très bonne intégration de P2P-MPI à n'importe quel programme Java. Le déclenchement de la découverte dynamique d'autres processeurs disponibles, le transfert des fichiers et programmes, la coordination des différents processus formant le communicator est fait à l'intérieur du code applicatif. Ceci permet à l'utilisateur de n'utiliser que son ordinateur personnel, et de déclencher les calculs à partir de celui-ci. Suite à cette collaboration, l'équipe fouille de données a acheté quatre noeuds bi-cœurs, intégrés dans un cluster existant, qui ont pour vocation de faire tourner en permanence des pairs P2P-MPI pour maintenir ainsi une infrastructure minimale de calcul.

Cette utilisation de P2P-MPI dans un cluster ouvre la perspective d'un travail important à faire pour intégrer de manière élégante deux conceptions contradictoires du partage de la ressource de calcul. Avec P2P-MPI, l'utilisateur décide combien de processus peuvent être exécutés simultanément, et le système répond instantanément oui ou non à une requête d'exécution. Dans le cas des équipements de calcul mutualisés comme les clusters, c'est habituellement un gestionnaire de jobs qui contrôle l'utilisation de la ressource, et son objectif est de maximiser cette utilisation. Quand une requête est envoyée vers ce gestionnaire, elle est systématiquement acceptée et placée dans une file d'attente pour une exécution à une date ultérieure dont on ne peut donner qu'une prédiction. Cette prédiction ne peut d'ailleurs raisonnablement être faite que par un service ayant accès à toutes les informations possédées par le gestionnaire. En effet, la durée maximum d'un job annoncée par l'utilisateur est une sur-estimation de la durée réelle, et on peut utiliser un historique des applications exécutées pour approcher la réalité [19]. Ensuite, la date d'exécution peut être prédite connaissant la politique d'ordonnancement des tâches, comme proposé dans [11] par exemple.

Nous avons fait des tentatives infructueuses de modification de la procédure de découverte des ressources pour tenir compte de la présence de gestionnaires de ressources. Ces tentatives n'ont pas été satisfaisantes car la prédiction de la date d'exécution de la tâche nous était indisponible, obligeant à fixer des timeouts trop longs pour être acceptables. De plus, la réactivité de ces gestionnaires, qui se mesure en dizaines de secondes (c'est la période de ré-ordonnancement des tâches dans les files d'attente), demanderait d'introduire des exceptions dans la gestion de la détection des pannes. Une fonctionnalité de type *advanced reservation* disponible dans certains gestionnaires (comme GridEngine [13], OAR [4], ou KOALA [17]) permettrait de proposer un mode de soumission programmé utilisant à la fois des équipements de calcul munis d'un gestionnaire de tâches et des ordinateurs individuels.

6.6 Résumé des contributions

Nous avons proposé une parallélisation en MPJ de la méthode de clustering MACLAW [3], qui s'intègre complètement dans la plateforme exsitante en Java de nos collègues de l'équipe de fouille de données. L'étude de la méthode en vue de sa parallélisation a permis d'en établir précisément sa complexité, et de mettre en évidence, en fonction du nombre de clusters à trouver, l'extensibilité

maximale qu'on pouvait espérer. La solution proposée permet avant tout de réduire les temps de traitement, prohibitifs en séquentiel pour des grandes instances de problème, et de rendre ainsi la méthode utilisable dans de plus nombreux cas. Ces résultats sont publiés dans l'article [12].

Bibliographie

[1] MPI : A message passing interface standard, version 1.1. Technical report, University of Tennessee, Knoxville, TN, USA, jun 1995.

[2] P. Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[3] A. Blansché and P. Gançarski. MACLAW : A modular approach for clustering with local attribute weighting. *Pattern Recognition Letters*, 27(11) :1299–1306, 2006.

[4] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*. IEEE, 2005.

[5] F. Cappello et al. Grid'5000 : A large scale, reconfigurable, controlable and monitorable grid platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing Grid'2005*, November 2005. http ://www.grid5000.org.

[6] E.Y. Chan, W.K. Ching, M.K. Ng, and J.Z. Huang. An optimization algorithm for clustering using weighted dissimilarity measures. *Pattern Recognition*, 37 :943–952, 2004.

[7] I.S. Dhillon and D.S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, pages 245–260. Springer-Verlag, 2000.

[8] C. Domeniconi, D. Gunopulos, S. Ma, B. Yan, M. Al-Razgan1, and D. Papadopoulos. Locally adaptive metrics for clustering high dimensional data. *Data Mining and Knowledge Discovery*, 14(1) :63–97, 2007.

[9] G. Forman and B. Zhang. Linear speedup for a parallel non-approximate recasting of center-based clustering algorithms, including k-means, k-harmonic means, and em. In *ACM SIGKDD Workshop on Distributed and Parallel Knowledge Discovery, KDD-2000*, 2000.

[10] J. H. Friedman and J. J. Meulman. Clustering objects on subsets of attributes. *Journal of the Royal Statistical Society*, 66(4) :815–849, 2004.

[11] Jean-Sébastien Gay and Yves Caniou. Simbatch : an api for simulating and predicting the performance of parallel resources and batch systems. Technical Report RR2006-32, LIP, October 2006. Also available as INRIA Research Report 6040.

[12] Stéphane Genaud, Pierre Gançarski, Guillaume Latu, Alexandre Blansché, Choopan Rattanapoka, and Damien Vouriot. Exploitation of a parallel clustering algorithm on commodity hardware with P2P-MPI. *Journal of SuperComputing*, 1(43) :21–41, 2008.

[13] W. Gentzsch. Sun grid engine : Towards creating a compute power grid. In *CCGRID '01 : Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35. IEEE Computer Society, 2001.

[14] R. Gnanadesikan, J. R. Kettenring, and S. L Tsao. Weighting and selection of variables for cluster analysis. *Journal of Classification*, 12(1) :113–136, 1995.

[15] N. Howe and C. Cardie. Examining locally varying weights for nearest neighbor algorithms. In *ICCBR*, pages 455–466, 1997.

[16] C. Kruengkrai and C. Jaruskulchai. A parallel learning algorithm for text classification. In *Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2002).*, July 2002.

[17] H. H. Mohamed and D. H. J. Epema. Experiences with the koala co-allocating scheduler in multiclusters. In *CCGRID'05 : Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 784–791. IEEE Computer Society, 2005.

[18] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data : A review. *SIGKDD Explorations, Newsletter of the ACM Special Interest Group on Knowledge Discovery and Data Mining*, 6(1) :90–106, 2004.

[19] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times with historical information. *J. Parallel Distrib. Comput.*, 64(9) :1007–1016, 2004.

[20] R. Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3) :645–678, 2005.

Chapitre 7

Synthèse et Projet de Recherches

7.1 Résumé des contributions

La façon dont j'ai mené mes recherches sur la thématique des grilles de calcul a été assez constante. Depuis le démarrage du projet *Transformations et Adaptations pour les Grilles*, nous sommes partis de l'étude d'applications existantes, en nous intéressant au cas particulier des programmes parallèles à passage de messages, pour comprendre quels étaient les obstacles majeurs à leur exploitation sur des grilles.

Équilibrage de charge De manière constante, nous avons retrouvé la problématique de l'équilibrage de charge inhérente à l'hétérogénéité des processeurs et des réseaux. Le tracé de rais dans l'application de géophysique au chapitre 2 et l'application adaboost au chapitre 5 sont deux exemples de traitements basés sur des tâches indépendantes. Pour ce cas des tâches indépendantes, nous avons étudié des solutions d'équilibrage statique au chapitre 3. Cependant, nous avons montré qu'une solution dynamique de type maître-esclave est i) générique, dans la mesure où elle peut être utilisée dans de nombreux cas, comme par exemple pour ajuster la distribution des données dans la première itération d'adaboost, et ii) robuste, car elle s'adapte naturellement à un changement soudain de charge, contrairement à une approche statique.

Expérimentation Depuis le début, j'ai également la conviction que l'expérimentation est une composante essentielle de ce domaine de recherche. En effet, la complexité résultante de l'amoncellement des couches logicielles et matérielles rend extrêmement difficile la modélisation réaliste des programmes dans un tel contexte, et il est délicat de prédire des résultats sans pouvoir les vérifier expérimentalement. L'outil Grid'5000 [11] que nous avons utilisé dès qu'il a été disponible est à cet égard extrêmement précieux. Tout au long des chapitres, nous avons relaté les évaluations expérimentales que nous avons systématiquement réalisées. Comme il est difficile de prédire grâce à un modèle les performances des programmes, l'expérimentation reste souvent le seul moyen d'acquérir une expérience de ce qui peut donner des résultats intéressants dans la réalité, ou au contraire n'avoir aucun intérêt. Dans le chapitre 2, nous avons montré que les performances des réseaux longue distance avant 2003 n'étaient pas suffisantes pour envisager l'utilisation de ressources de calcul non locales dans l'application de tracé de rais, alors qu'en 2006, cette même application pouvait être distribuée sur cinq sites en France et afficher une courbe d'accélération linéaire jusqu'à environ 500 processeurs. De même, il est difficile de dire quel niveau de contention maximum un modèle de programmation comme celui utilisé dans JavaSpace peut raisonnablement supporter.

Nouvelles voies pour les intergiciels J'ai acquis avec mon équipe une expérience que je considère assez originale au niveau national, qui a trait à la mise en œuvre d'applications parallèles sur grille. Nous l'avons d'abord expérimenté avec MPICH-G2 [18] et Globus [19]. Cette expérience m'a fait comprendre certaines des limitations de ces intergiciels, en particulier les innombrables problèmes liés à la maintenance de l'installation logicielle et à l'absence de tolérance aux pannes. Ce constat a orienté mes recherches pour proposer une nouvelle voie pour les intergiciels. Le prototype d'intergiciel que nous avons proposé, P2P-MPI[1], repose sur i) qui permet l'exécution de programmes parallèles Java, ii) qui simplifie les problèmes de maintenance du système d'information concernant les ressources de calcul disponibles en proposant une organisation en mode pair-à-pair, et iii) qui autorise la tolérance aux pannes par de la réplication.

Tout au long de la conception et du développement de l'intergiciel, nous avons continué à travailler sur des applications, dans l'optique d'évaluer les bénéfices que les utilisateurs peuvent tirer d'un tel environnement d'exploitation. Au bilan, P2P-MPI s'est révélé être un cadre très intéressant pour collaborer avec des utilisateurs souhaitant paralléliser des programmes Java.

Perspectives J'inscris mes perspectives dans la lignée de mes travaux actuels. Je suis engagé dans deux projets d'ANR et mon détachement à l'INRIA m'a permis de multiplier les contacts avec d'autres chercheurs concernant des thématiques connexes à mes travaux en cours. Dans le reste de ce chapitre, je dégage deux directions générales vers lesquelles je souhaite diriger mes efforts.

La première est la simulation et la modélisation. Bien que l'approche expérimentale dans notre communauté ait fait un pas de géant avec l'arrivée de Grid'5000, je crois que l'expérimentation doit mener à la construction de modèles. En effet, quelle que soit la facilité de mise en œuvre des expériences, le nombre d'expériences nécessaires pour établir scientifiquement un résultat est généralement élevé, et donc coûteux en temps. Grâce à la simulation, on peut discerner beaucoup plus vite les cas intéressants, et diminuer ainsi le nombre de vérifications expérimentales à réaliser.

La deuxième direction concerne la nature des applications auxquelles je veux m'intéresser. Dans les travaux décrits ici, il n'est question que d'applications monolithiques. Or, les grilles se prêtent naturellement à des applications composées de modules, en particulier les *workflows*. On ajoute potentiellement beaucoup de complexité en traitant des workflows, mais étant donné que ce type d'application correspond à de nombreux besoins réels, on trouve depuis assez longtemps des solutions imparfaites mais fonctionnelles. Je distingue deux domaines dans lesquelles on pourrait faire progresser les solutions existantes. Le premier est celui de la gestion des données, et sur le long terme, j'aimerais concevoir un service de gestion de données capable de coopérer de manière plus efficace avec les applications. Le deuxième concerne l'allocation des ressources. La complexité croissante des ressources de calcul (de plus en plus spécialisées et hiérarchisées) nécessite d'intégrer cette caractéristique à la localisation des ressources. Sur le long terme, l'objectif pourrait être celui d'un intergiciel capable à la fois de détecter très rapidement les arrivées/départs de nouvelles ressources, d'évaluer les capacités de la ressource à exécuter une catégorie de programme, et de prendre en charge de bout en bout une requête d'exécution.

7.2 Projet de Recherches

L'évolution actuelle des matériels et systèmes de traitement de l'information va vers une complexité croissante : les systèmes sont de plus en plus hétérogènes, les matériels de plus en plus spécialisés. D'une part, on observe la généralisation de la présence de GPGPU (General-Purpose computation on Graphics Processing Units) capables de performances largement supérieures aux

[1] `http://www.p2pmpi.org`

CPU généralistes sur des problèmes particuliers, le nombre croissant de cœurs dans chaque machine individuelle, ou le nombre croissant de clusters [35]. D'autre part, les équipements sont de plus en plus accessibles à travers le réseau depuis que beaucoup de problèmes de sécurité ont été résolus par l'emploi de connexions sécurisées ou de VPN. On peut aujourd'hui envisager également de recourir à des tiers commerciaux pour fournir des ressources, qu'elles soient de type IaaS (Infrastructure as a Service) (e.g Amazon EC2), PaaS (Platform as a Service) (e.g Google App Engine) ou SaaS (Software as a Service), que ce soit pour du calcul, du stockage ou un service logiciel. Au final, l'éventail de possibilités pour déployer une application est devenu très large. Le corollaire est la complexité du déploiement qui va de pair avec cette multiplicité des matériels. Je pense qu'un travail important doit être fait pour que les intergiciels puissent aider l'utilisateur à déployer ses applications, en lui fournissant un panorama des ressources accessibles, en le guidant dans ses choix (en prenant en compte les besoins de son application vis-à-vis des spécificités des matériels), et en prenant en charge le déploiement après avoir procédé à la co-allocation des ressources.

Les perspectives de travail pour approcher un tel objectif sont multiples. D'abord, il est extrêmement difficile de reconstituer la complexité de tels environnements. C'est pourquoi les travaux préliminaires sont souvent testés à l'aide de modèles de l'environnement, ou à l'aide de simulateurs. Adapter un simulateur existant à cette évolution des systèmes est donc une piste de recherche en soi (section 7.2.1). Une autre perspective concernant les applications complexes est celle de la gestion de leurs données : utiliser différents matériels à différents endroits implique de déplacer les données à traiter. Étant donné la masse croissante de données dans les applications scientifiques, ce problème est crucial (section 7.2.2). Enfin, l'évolution de la complexité des environnements énoncée en exergue demande des intergiciels capables d'allouer les ressources de manière efficace, afin de permettre aux applications d'exploiter la diversité des ressources. Cet objectif implique de progresser à la fois sur la problématique de la découverte des ressources de calcul et de leur interconnexion, et sur les heuristiques d'allocations de ces ressources (section 7.2.3).

7.2.1 Simulation et Modélisation

Les expérimentations que nous avons menées dans ce travail sur les grilles sont fastidieuses. L'apparition de l'outil Grid'5000 a considérablement élargi les possibilités d'expérimentation dans un environnement réel, tout en permettant un protocole expérimental plus rigoureux car l'utilisateur peut sélectionner le matériel voulu, et installer le système d'exploitation de son choix. Le caractère reproductible des expériences a donc été considérablement amélioré avec Grid'5000, mais reste néanmoins imparfait car le réseau reliant les sites ainsi que les clusters sont régulièrement renouvelés. Les expériences doivent donc se succéder sur une période relativement courte pour obtenir une série de résultats comparables. Ceci n'est pourtant pas facile à mettre en œuvre car l'accès à l'outil au moment souhaité n'est pas toujours simple dans la pratique. La simulation présente donc un grand intérêt pour compléter l'expérimentation réelle. Elle peut permettre de tester de nombreux scénarios et de ne faire des expériences réelles que pour les cas les plus intéressants.

Récemment, le projet d'ANR USS-SimGrid[2] (2009-2011) porté par Martin Quinson s'est fixé l'objectif d'étendre les capacités du simulateur SimGrid [12]. SimGrid repose sur noyau de simulation à événèments discrets, qui calcule la date à laquelle se produisent les événements d'une exécution distribuée d'après un modèle du programme et un modèle de comportement de la plate-forme. La plate-forme désigne les ressources de calcul et leur interconnexion (topologie, performance des liens réseaux, performance des CPU,...). Le comportement des communications réseaux est approximé par des modèles analytiques des performances de TCP ou d'un simulateur au niveau paquet (GTNets). Pour modéliser les programmes, plusieurs *interfaces* au noyau du simulateur sont disponibles.

[2] `http://uss-simgrid.gforge.inria.fr/`

Une interface définit les primitives de calcul et de communication utilisables pour modéliser le programme. Une des limitations des interfaces existantes est qu'elles ne proposent que des communications point à point bloquantes.

SMPI Il est donc difficile de simuler de nombreuses applications réelles basées sur du parallélisme à passage de message. L'une des extensions prévue est de faire aboutir SMPI, une nouvelle interface capable de simuler des programmes MPI sans modification des codes sources, dont les bases ont été jetées par Henri Casanova et Mark Stilwell (Univ. Hawaï). Mes premiers efforts pendant l'été 2009 ont abouti à l'implémentation d'un bon nombre de primitives MPI dont sept opérations de communications collectives. L'implementation utilise les mêmes algorithmes qu'OpenMPI [21], une des implémentations de référence de MPI. En résumé, un programme P exécuté à travers une implémentation I sur une plate-forme matérielle M donne lieu à une exécution E. Notre simulation de P à travers une implémentation I' et un modèle M' de la plate-forme donne lieu à une exécution E'. Plus E et E' sont "proches", plus la simulation est réaliste.

Il y a deux perspectives à ce travail. La première est évidemment de simuler à l'aide d'un seul ordinateur, l'exécution d'un programme parallèle sur une plate-forme virtuelle. Le premier prototype de SMPI devrait être achevé à court terme. En revanche, il est probable que les premiers tests de validation des simulations obtenues engendreront beaucoup de questions et de travail pour affiner le simulateur. Il faudra travailler sur I', voire sur l'interaction de I' et M' pour que E' s'approche de E.

Cette partie nécessitera de comprendre les facteurs clés qu'il faut capturer dans la simulation pour la rendre réaliste. Ceci passe par une bonne expertise du comportement des programmes ainsi que des implémentations MPI utilisées. De ce point de vue, mon expérience est variée. Il y a d'abord les programmes MPI de la suite Ray2mesh (c.f. Chapitre 2) dont les benchmarks ont été faits à la fois sur différents types de supercalculateurs (machines cc-NUMA, clusters) et dans des environnements hétérogènes constitués de plusieurs clusters reliés par des backbones. Je compte également profiter de l'expérience acquise dans la mise au point de P2P-MPI pour sa partie bibliothèque de communication.

Avec SMPI, nous aurons une évaluation macroscopique de la fidélité de la simulation (au niveau d'une application MPI entière), alors que jusqu'à présent, les tests de validation du simulateur sont plutôt unitaires ou sur des exemples simples. Cette évaluation devrait permettre de confirmer la qualité du simulateur ou de donner des pistes pour l'améliorer.

La deuxième perspective inscrite dans le projet d'ANR est l'extrapolation d'exécutions MPI. L'idée est de capturer une trace d'exécution réelle à l'aide d'un outil de profiling existant, la convertir en événements discrets compatibles avec le simulateur pour ensuite l'injecter dans le simulateur. À l'aide du simulateur, l'utilisateur peut rejouer l'exécution réelle, l'interrompre à tout moment et lancer la simulation à partir de l'état atteint. L'intérêt est donc de minimiser les divergences induites par la simulation, par rapport à l'exécution réelle. Ceci peut servir par exemple à comparer de manière beaucoup plus précise les différences entre l'exécution réelle et la simulation d'un fragment de programme qui se déroule vers la fin de l'exécution. On peut aussi imaginer l'évaluation de scénarios de type *what if* : on stoppe le rejeu de la trace réelle, on choisit de modifier les caractéristiques de la plate-forme (par exemple saturation d'un lien réseau) à partir de cet instant, et on lance la simulation dans ces nouvelles conditions virtuelles.

Modélisation des communications TCP Dans le cadre des environnements hétérogènes avec des communications TCP à forte latence, typiques des grilles, un des facteurs-clé est la modélisation des communications. Les modèles réseaux analytiques disponibles dans SimGrid (bien plus utilisables que ceux basés sur la simulation au niveau paquet) ne prennent pas en compte certains détails importants si on considère une communication individuelle (par exemple, la phase *slow-start* de TCP

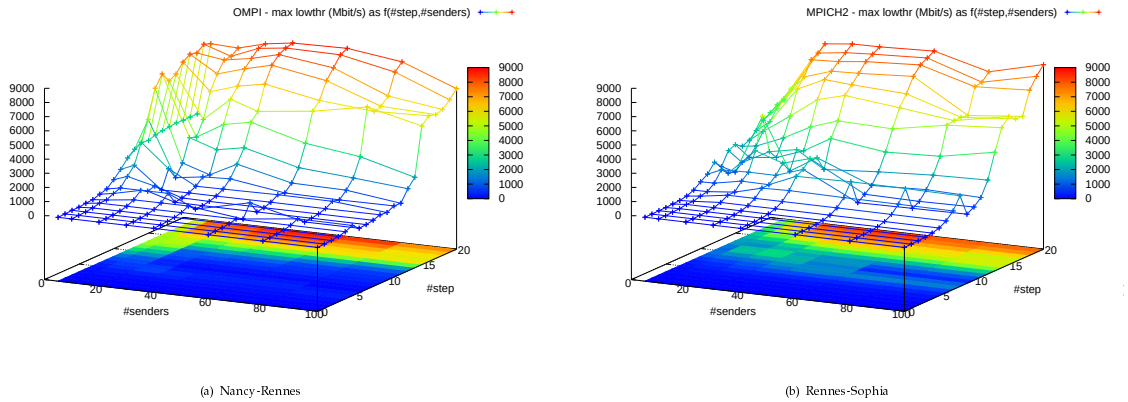
n'est pas prise en compte). En collaboration avec Jean-Jacques Pansiot de l'équipe réseau du LSIIT, nous avions engagé en 2006 des travaux sur la modélisation des communications TCP au niveau applicatif à travers le travail de master de Constantinos Makassikis. Nous avons conclu qu'une modélisation analytique très fine du comportement d'un flux TCP (comme le modèle de Padhye [32] qui fait référence) est très difficilement généralisable à des flux multiples et des topologies de réseau hiérarchiques. Une perspective prometteuse serait donc de construire une modélisation à un niveau intermédiaire, permettant de prédire les coûts des communications à partir de paramètres que l'application peut inférer directement : par exemple la latence, ou le temps incompressible entre deux messages, alors que l'application ne peut connaitre le taux de perte des paquets qui est un des paramètres de la formule de Padhye. Les modèles LogP [14], PLogP [26, 25], LogGP [4], ou Hockney [24] respectent justement cette contrainte. En revanche, s'ils sont satisfaisants quand l'application utilise un cluster unique, ils sont défaillants pour plusieurs clusters reliés par un *backbone*.

Je l'ai constaté durant l'année 2008, après avoir mené une campagne massive de mesures des performances des opérations collectives alltoall et alltoallv entre plusieurs clusters sur Grid'5000. L'objectif du travail en collaboration avec Emmanuel Jeannot et Antonio Grassi (internship) était de construire des algorithmes plus performants pour ces opérations (les versions optimisées de ces primitives, basées sur des arbres de diffusion, ont des performances inférieures à une approche naïve où la communication s'effectue simultanément entre chaque paire de processus). Nous avons testé deux idées pour construire des algorithmes alternatifs. La première idée est l'agrégation des messages individuels en messages plus longs pour traverser les backbones avec une redistribution des messages individuels à l'arrivée sur les clusters destination. La deuxième idée est de restreindre le nombre de processus émettant à l'extérieur du cluster, ceci afin de réguler le nombre de flux TCP entrants sur le backbone et ainsi éviter la congestion. Ces deux intuitions n'ont pas été fructueuses par rapport à l'approche naïve : la gestion de la congestion au niveau TCP est suffisamment efficace et on ne gagne quasiment rien à essayer de réguler les flux. La figure 7.2.1 illustre la difficulté de modéliser le comportement d'une opération alltoall naïve [3]. On trace sur l'axe vertical le débit obtenu quand on transfère une certaine quantité D de données, en utilisant plus ou moins d'émetteurs (et donc de flux TCP simultanés) : chaque émetteur transmet son buffer individuel de taille b, telle que $b^2=D$. On fait également varier D sur l'axe intitulé *#steps*. Sur la figure 7.1(a) les processus émetteurs sont à Nancy, les récepteurs à Rennes, et sur la figure 7.1(b) les processus émetteurs sont à Rennes et les récepteurs à Sophia. On se rend compte dans les deux cas que la fragmentation des messages n'est globalement pas pénalisante sauf quand on atteint un grand nombre d'émetteurs (à partir de 80 émetteurs, c'est-à-dire que D est transmise en 6400 messages). D'autre part, la fragmentation optimale change en fonction de D, ce qui rend difficile le choix d'une stratégie.

Après avoir mené de nombreuses expériences sur ce thème, nous avons évalué les prédictions du modèle PlogP par rapport à nos mesures. Il en ressort que le modèle n'arrive pas à prédire la contention générée par les opérations collectives comme alltoall en présence d'un backbone. Néanmoins, j'ai commencé à modifier le modèle pour qu'il corresponde au mieux à toutes les données collectées (nombre d'emetteurs et récepteurs variables, cinq paires de sites Grid'5000) et les résultats préliminaires sont encourageants. La perspective est donc de dériver un modèle qui tienne compte de l'opération MPI utilisée, et qui fournisse une estimation à partir d'un grand nombre d'observations et du modèle des communications de PLogP. Il pourrait être la base d'un nouveau modèle des coûts de communication pour SMPI.

Finalement, ces travaux s'inscrivent dans un effort collectif, bien structuré au niveau national et même international. Au niveau national, la modélisation du réseau pour les applications sur grille est un thème important de l'équipe Reso, au centre INRIA Grenoble Rhône-Alpes (Pascale Primet) avec laquelle nous collaborons. La perspective de modélisation décrite précédemment pourrait bé-

[3] Il ne s'agit en fait que d'un "demi" alltoall, les émetteurs ne recevant pas pour ne pas polluer la mesure.



(a) Nancy-Rennes (b) Rennes-Sophia

FIG. 7.1 – Débits maximums observés dans un alltoall non-optimisé, pour 5 à 100 émetteurs et autant de récepteurs sur le site distant, et des données totalisant de 24KB à 24GB (pour chaque point, 33 mesures prises du 6 au 27/05/2008).

néficier de leur expertise, en particulier pour mieux instrumenter les expériences (au niveau paquet TCP) et comprendre plus finement certains phénomènes. Ce travail est à mener dans la lignée de ceux engagés pour mettre en avant les éléments importants dans la performance de TCP dans les applications MPI sur la grille [23].

Pour la simulation, SimGrid a bénéficié au niveau national de deux actions spécifiques de l'INRIA pour pérériser son développement logiciel, et du projet d'ANR USS-SimGrid, pour développer ses perspectives scientifiques. Cet outil rentrera également dans l'un des volets de l'action d'envergure Héméra de l'INRIA, adossée à l'action de développement technologique Aladdin qui supporte la plate-forme Grid'5000. Au niveau international, outre la contribution d'Henri Casanova à Hawaï, les contributeurs extérieurs au code de SimGrid sont de plus en plus nombreux, faisant de cet outil l'un des simulateurs pour applications réparties les plus connus au plan international.

7.2.2 Gestion des données

Ordonnancement des mouvements de données La gestion des données est un problème majeur dans le domaine des grilles. Les protocoles de transfert et de stockage des données sont nombreux mais sont souvent liés aux besoins d'applications spécifiques. La solution GridFTP [5] conçue pour s'intégrer avec Globus, consiste à faire tourner un démon sur la ressource de stockage, qui exporte le système de fichier et en contrôle les accès distants. Un service de gestion des réplicas (RLS) peut répertorier les différents exemplaires des fichiers et fournit un service de nommage logique des fichiers. Plus proche de l'idée d'un service de partage des données, le *storage resource broker* (SRB) [8] développé depuis 1995 est un service qui fournit une vision unifiée de systèmes de fichiers ou de bases de données répartis sur plusieurs sites à l'aide d'un gestionnaire de meta-données. L'approche est aussi de type client/server : le client se connecte à un ou plusieurs SRB masters à une adresse connue avant de pouvoir transférer des données. Les données sont représentées de manière abstraite dans des collections hiérarchiques, dont les éléments sont potentiellement stockés sur des ressources hétérogènes et physiquement distantes.

Ces solutions très utilisées nécessitent des ressources de stockage stables et des paramètres de configuration à jour. Elles sont pertinentes pour des grilles institutionnelles mais beaucoup moins dans un environnement plus volatile. Dans ce cas, les infrastructures pair-à-pair sont clairement

supérieures. Les protocoles éprouvés de partage de fichiers en P2P, comme *bittorrent*, sont donc des candidats naturels comme support à un service de gestion des données. Des études récentes, comme [3], s'intéressent à comparer l'efficacité de différentes approches, client/serveur ou P2P, pour la gestion des données dans une grille de calcul. Il me semble donc naturel qu'à l'avenir, on perde la notion de localité des données, et que le système puisse trouver quelles sont les ressources physiques les plus adaptées pour satisfaire une requête. On pourra dire que ces systèmes sont adoptés le jour où les utilisateurs n'iront pas, préalablement à un calcul, copier toutes les données sur le site où se déroule le calcul.

Dans ce sens, le projet JuxMem [6] a proposé l'idée originale d'une mémoire partagée virtuelle, avec une persistance des données, physiquement réparties sur des pairs JXTA. De manière similaire, mais en adoptant une approche plus globale, BitDew [17] unifie l'utilisation de plusieurs protocoles (http, ftp ou *bittorrent*) derrière son API de gestion de données. Il est proposé en plus, un service ordonnançant des mouvements des données pour satisfaire les dépendances entres données. Dans P2P-MPI, la version publiée en septembre 2009 intègre un cache de données dont la taille est fixée par le propriétaire de la ressource. Ceci évite, dans le cas où les mêmes pairs sont sélectionnés au cours d'exécutions successives utilisant les mêmes données, de télécharger à chaque fois ces données. Même si cette fonctionnalité est mineure, elle remet en question les stratégies d'allocation des ressources. En effet, après quelques exécutions, les pairs sélectionnés pouvant varier d'une exécution à l'autre, la totalité ou une partie (en raison de la politique de remplacement du cache) des données nécessaires peuvent se retrouver disséminées à de nombreux endroits. Lors de l'allocation des processus de calcul, une stratégie intelligente devrait évaluer s'il est intéressant d'utiliser les ressources disposant déjà des données dont dépendent les calculs.

Cette notion de dépendance entre données, ou entre données et calculs, est très importante et peu exploitée dans les logiciels existants. La vision la plus précise des dépendances de l'application par rapport aux données est fournie par le code source. Dans des cas favorables, le modèle de programmation permet d'exprimer la sémantique de l'accès aux données. Un tel exemple est l'extension MPI IO, intégrée au standard MPI-2 en 1997. Un travail intéressant serait de comprendre si cette API est appropriée à la grille, et comment ses primitives doivent être implémentées pour supporter la latence réseau et l'hétérogénéité. Des travaux préliminaires ont été récemment engagés par Fukuda et Miyauchi [20] pour implémenter les primitives de MPI IO dans leur intergiciel de grille. Dans d'autres approches, comme par exemple *Data Handover* [22] proposée par Jens Gustedt, les données mémoires partagées par les processus répartis sont identifiées de manière explicite par le programmeur (contrairement à MPI où un message reçu ou envoyé transite dans une zone mémoire banalisée du point de vue du programme), et celui ci donne des indications sur la façon dont il va les utiliser (lecture-seule, écriture exclusive, etc). Cette approche peut-être étendue aux données provenant de fichiers. L'analyse du programme donnerait alors des informations intéressantes pour le placement des fichiers.

En conclusion de ce point, il y a à mon sens de nombreuses perspectives d'amélioration de la gestion des données par inspection des codes sources, ou par ajout de directives. Un premier travail peut être de définir une API qui soit à la fois pertinente pour le programmeur en lui permettant de manipuler des fichiers (locaux, distants, ou répliqués) de manière abstraite, et pertinente pour le système de gestion de données sous-jacent, qui doit avoir suffisamment d'informations pour placer et déplacer les données aux endroits nécessaires pour accroître la performance. L'idée est d'anticiper les accès aux données distantes, afin de les pré-acheminer à l'endroit où s'exécute le processus qui en a besoin. Cette analyse devrait s'appuyer sur un service annexe de gestion des données, utilisable par toutes les applications. On rejoint l'idée développée dans le projet Internet Backplane Protocol (IBP) [9], qui permet d'une part à différentes applications d'écrire et lire des données dans un entrepôt,

et d'autre part à une application d'ordonner à IBP de démarrer des transferts asynchrones d'un serveur vers un autre. Notons toutefois qu'IBP s'affranchit de nombreuses contraintes : l'écriture de données peut être refusée par le serveur, on ne peut modifier des données écrites, et les données peuvent être temporairement inaccessibles (comportement *best effort*). Plus récemment mais avec le même objectif et la même optique d'accès transparent aux données, il a été proposé de fournir à Grid-RPC un modèle de gestion des données [7]. Comparées à IBP, les spécifications du modèle offrent des garanties supérieures : persistance, tolérance aux pannes (par réplication), cohérence (les données peuvent être modifiées, donc le système doit assurer leur cohérence si elles sont répliquées). Ce type de projet représente donc une base intéressante pour travailler sur l'ordonnancement des mouvements de données par analyse des codes sources.

Ordonnancement des workflows dirigé par les données Mettre en évidence des dépendances vis-à-vis des données n'est pas toujours facile. En particulier, si l'application est composée de plusieurs codes, écrits dans des langages différents, voire composée de codes patrimoines, l'approche précédente ne peut pas s'appliquer. Dans ce cas, les dépendances entre les différents codes de calcul et les données en entrée et sortie sont habituellement représentées par un *workflow*. Les forts besoins en la matière exprimés dans différentes communautés scientifiques ont amené à maturité plusieurs outils de gestion de workflow (e.g Triana [34], Kepler [29] ou Taverna [31]).

L'utilisateur modélise son workflow et le confie ensuite à un moteur d'exécution chargé de l'exécuter. Pour cela, le moteur soumet les tâches au gestionnaire de tâches sous-jacent qui est le meta-ordonnanceur de la grille. L'exécution du workflow est un problème difficile à la fois d'un point de vue théorique et pratique. D'un point de vue théorique, il s'agit d'un problème d'ordonnancement, et des heuristiques classiques d'ordonnancement statiques sont proposées (par exemple Pegasus [15] propose HEFT, min-min, round-robin et random). La qualité de ces ordonnancements dépend bien sûr de l'exactitude des informations fournies (temps d'exécution de la tâche, disponibilité des données, ...). D'un point de vue pratique, la difficulté est que chaque plate-forme d'exécution possède son propre intergiciel muni d'un gestionnaire de tâches, avec des capacités ou des politiques d'ordonnancement différentes. Par conséquent, il est difficile de piloter en général l'ordonnancement à partir du moteur d'exécution du workflow.

La perspective que j'entrevois pour améliorer l'ordonnancement des workflows est, comme dans le paragraphe précédent, liée à l'observation de l'utilisation des données. L'ordonnancement devrait prendre en compte le temps d'accès aux données comme il le fait avec les temps de calcul des tâches. Un fichier de données consulté en lecture seulement durant tout le workflow pourrait être répliqué à plusieurs endroits pour que plusieurs tâches puissent l'utiliser. Des transferts de données pourraient aussi être anticipés pour précéder leur utilisation, ou au contraire, certaines tâches peuvent calculer alors qu'une partie seulement du fichier de données est présente (streaming).

La problématique qui consiste à considérer les données comme un élément essentiel de l'application, permettant entre autres de guider le placement des tâches, mérite des investigations supplémentaires. Une première contribution à été faite par les chercheurs qui ont proposé Stork [27], qui considère les opérations sur les données comme des tâches à part entière et les intègre dans son ordonnancement. Cependant, ce type d'approche doit être repensé pour s'intégrer dans un service de gestion des données qui comprendrait par défaut de la réplication.

7.2.3 Déploiement des applications

Nous avons souligné en préambule l'élargissement considérable de l'éventail des ressources utilisables. Cette évolution modifie profondément la problématique du déploiement d'une application : sans l'aide d'un intergiciel, l'utilisateur doit connaître en permanence les spécificités des ressources (par exemple la taille mémoire, le nombre de cœurs d'un PC), le temps moyen d'attente dans le

batch scheduler de tel cluster, la possibilité de faire communiquer telle machine avec telle autre sur ce port, la capacité disque de telle autre machine, etc. On peut naturellement espérer que des intergiciels viendront aider l'utilisateur, en permettant la maîtrise des aspects dynamiques de l'environnement (apparition/disparition des ressources, taux d'occupation, ...) et en les guidant dans le choix des ressources les mieux adaptées pour exécuter telle ou telle partie de l'application.

Découverte de ressources La première problématique liée à cet objectif est la découverte des ressources et de la topologie du réseau connectant ces ressources. Concernant la découverte des ressources à l'intérieur d'un même sous-réseau ou d'un réseau privé, des standards industriels pourraient s'imposer et apporter des solutions assez rapidement. En particulier, l'industrie du loisir multimédia pourrait rapidement arriver à un consensus, comme par exemple l'Universal Plug and Play (UPnP). Les équipements domestiques récents (e.g disques durs multimédia) sont en effet conçus pour être interconnectés dans un réseau privé, et il faudra obligatoirement automatiser la découverte et la connexion pour rendre ces technologies accessibles à tous. Dans notre domaine, à ma connaissance, un seul travail de recherche a étudié l'adéquation et les performances des protocoles de type *publish/subscribe* pour les grilles de calcul. Le travail étudie le protocole ZéroConf, à travers son implémentation Bonjour [1]. Les auteurs ont montré que le protocole pouvait même être utilisé pour construire et coordonner un environnement d'exécution [2].

Cependant, l'immense majorité des travaux ignorent les problèmes de connectivité des ressources d'un sous-réseau à un autre. C'est pourtant une composante essentielle des systèmes aujourd'hui. Si on exclut les grilles institutionnelles pour lesquelles les procédures d'installation sont très calibrées, il est très difficile et fastidieux de découvrir la façon dont chaque ressource peut communiquer avec les autres. Pourtant, c'est aujourd'hui la norme d'avoir à utiliser un VPN pour accéder à un cluster. Les pratiques concernant l'usage des réseaux se sont complexifiées : les techniques de pare-feux pour filtrer de nombreux ports, de NAT permettant de multiplier les adresses disponibles, ou de *multi-homing* pour accroître l'accessibilité, sont quelques exemples qui rendent difficile le déploiement des applications distribuées. Il existe des techniques pour contourner ces obstacles (par exemple ICE, STUN, ...) mais les propositions ne sont que des réponses partielles au problème. A notre connaissance, seul *SmartSockets* [30] du projet Ibis a tenté d'unifier ces propositions en une seule solution logicielle, apte à découvrir la connectivité des ressources (un graphe orienté) et à mettre en place certains mécanismes pour que deux ressources qui ne peuvent pas communiquer directement puissent le faire à travers un lien logique (par exemple, en utilisant un noeud intermédiaire, en utilisant du *reverse routing*, ou des tunnels). Cependant, le graphe orienté de connectivité obtenu n'apporte pas d'information sur le coût d'emprunter tel ou tel lien. En raison des techniques employées, les liaisons logiques construites peuvent afficher des performances considérablement dégradées. Il est donc nécessaire de valuer ce graphe pour pouvoir y déployer de manière pertinente une application distribuée. Ensuite, la perspective de travail consiste à trouver des heuristiques efficaces de placement du graphe d'application sur le graphe des ressources. Nous avons évoqué en section 4.4.2 les travaux initiés en la matière dans le travail de master [10] portant sur la co-allocation de tâches communicantes sur une plate-forme hétérogène. Un travail très important est nécessaire pour mieux comprendre et évaluer l'efficacité de ces heuristiques dans les environnements complexes d'aujourd'hui.

Stratégies d'allocation des ressources Nous avons souligné l'hétérogénéité croissante des ressources de calcul. Certaines applications pourront être avantageusement prises en charge par une seule machine multi-cœurs, ou une machine multi-GPGPU, tandis que d'autres tireront plus de profit d'un petit laps de temps, éventuellement en mode *best-effort*, accordé sur un cluster avec beaucoup de processeurs. Pour pouvoir intégrer des décisions de cette nature dans le mécanisme d'allocation de ressources, il faut travailler sur deux plans.

D'une part, il faut mieux comprendre et modéliser les performances des machines multi-cœurs, notamment les effets de la contention mémoire. De nombreuses équipes de recherche travaillent ou orientent leurs travaux dans la compilation et l'optimisation, et donc la compréhension, des performances qu'on peut tirer des programmes sur des ressources de calcul spécialisées. Je participe à l'Action COST IC0805 *Open Network for High Performance Computing on Complex Environments* initiée par Emmanuel Jeannot, démarrée en mai 2009. En particulier, un des quatre groupes de travail s'intéresse à la conception, au développement et à la validation de bibliothèques logicielles efficaces avec un focus particulier sur les GPUs, les processeurs multi-coeurs et les réseaux hétérogènes. J'ai également ré-intégré l'équipe ICPS qui compte maintenant l'équipe INRIA CAMUS dont l'objectif est l'optimisation des programmes pour les multi-cœurs. Ce cadre est propice à des collaborations, avec comme perspective la possibilité de ré-utiliser des résultats des travaux de ces équipes pour la stratégie d'allocation.

D'autre part, pour exploiter des tranches de temps réduites qui peuvent apparaître de manière fugace sur des clusters, il faut rendre les intergiciels plus réactifs. Le projet SPADES (2009-2011)[4] auquel je participe a justement cet objectif de localiser rapidement des disponibilités sur des machines peta-scale, tout en permettant la co-allocation des ressources. Le succès commercial de l'*utility computing* (comme l'Elastic Cloud d'Amazon) qui consiste à louer ponctuellement des ressources de calcul ou de stockage, fait que ces solutions complémentaires devront peut être être prises en compte également à l'avenir. Dans le futur, on peut imaginer allouer à la volée de telles ressources, de manière très rapide, pour apporter un complément de puissance de calcul. Les deux clés pour faire progresser les intergiciels dans cette voie sont la gestion efficace de la dynamicité des environnements, et une meilleure standardisation des protocoles permettant à des intergiciels différents de coopérer. De ce point de vue, les spécification DRMAA (Distributed Resource Management Application API) et SAGA (Simple API for Grid Applications) établies dans le cadre de l'Open Grid Forum sont des avancées significatives.

Pour conclure sur le déploiement des applications, ma conviction est que l'intergiciel devra masquer la complexité des matériels et guider l'utilisateur dans le choix des ressources. Dans P2P-MPI par exemple, nous sélectionnons automatiquement les machines les plus proches de celle ayant soumis une requête, car c'est un paramètre que le système connaît et qui ne peut qu'accroître la performance. En revanche, nous demandons le concours de l'utilisateur pour savoir si son application peut bénéficier de l'allocation prioritairement sur des machines multi-cœurs. L'utilisateur peut donner des informations très utiles en caractérisant son application : communique-t'elle beaucoup, de combien de RAM a-t'elle besoin, fait-elle beaucoup d'accès mémoire, etc. Cette caractérisation des applications, déjà utilisée dans l'ordonnancement fait par ApLeS [13] il y a dix ans, est un problème en soi, et les critères de la caractérisation doivent évoluer pour refléter l'évolution des technologies et ce qui a le plus d'impact sur les performances.

Si on généralise cette caractérisation aux applications plus complexes comme les workflows ou les couplages de codes, l'aide que peut apporter un intergiciel est également cruciale. Dans ce contexte, qui peut par exemple impliquer des codes patrimoines développés avec des techniques différentes, la programmation par composants, qui permet de déployer des codes écrits selon des modèle de programmation différents, semble particulièrement adaptée. Dans cette approche, une couche logicielle permet d'encapsuler les codes dans des composants qui peuvent communiquer entre eux à travers des ports. Des propositions ont été faites pour les grilles de calculs, en montrant comment on pouvait encapsuler des codes parallèles [33, 16, 28]. Dans un contexte de matériels de plus en plus spécialisés, cette approche semble prometteuse pour cacher la complexité interne des composants. Une aide importante de l'intergiciel pourrait être de découvrir et proposer des ressources capables d'exécuter tel ou tel type de composant.

[4] `http://graal.ens-lyon.fr/SPADES/SPADES`

Utiliser simultanément les informations que peut fournir l'utilisateur (qui ne sont pas toutes quantitatives, ou pas quantifiables de manière exacte), ainsi que les informations provenant de la découverte dynamique des ressources disponibles de la plate-forme au sein d'un système de sélection des ressources est un vrai défi qu'il est nécessaire de relever pour exploiter plus efficacement les futurs environnements distribués hétérogènes.

Bibliographie

[1] Heithem Abbes, Christophe Cérin, , Jean-Christophe Dubacq, and Mohamed Jemni. Étude de performance des systèmes de découverte de ressources. In *Renpar'18*, Suisse, Fribourg, 11-13 février 2008.

[2] Heithem Abbes, Christophe Cérin, and Mohamed Jemni. Bonjourgrid as a decentralised job scheduler. In *IEEE APSCC-08*, December 2008.

[3] Samer Al-Kiswany, Matei Ripeanu, Adriana Iamnitchi, and Sudharshan Vazhkudai. Beyond music sharing : An evaluation of peer-to-peer data dissemination techniques in large scientific collaborations. *Journal of Grid Computing*, 7(1) :91–114, March 2009.

[4] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP : incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM New York, NY, USA, 1995.

[5] William E. Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The globus striped gridftp framework and server. In *SC*, page 54. IEEE Computer Society, 2005.

[6] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. Juxmem : An adaptive supportive platform for data sharing on the grid. *Scalable Computing : Practice and Experience*, 6(3) :45–55, September 2005.

[7] Gabriel Antoniu, Eddy Caron, Frédéric Desprez, Aurélia Fèvre, and Mathieu Jan. Towards a transparent data access model for the gridrpc paradigm. In S. Aluru et al. (Eds), editor, *HiPC'2007. 14th International Conference on High Performance Computing.*, number 4873 in LNCS, pages 269–284. Springer Verlag, December 2007.

[8] Chaitanya K. Baru, Reagan W. Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON*, page 5. IBM, 1998.

[9] Alessandro Bassi, Micah Beck, Terry Moore, James S. Plank, D. Martin Swany, Richard Wolski, and Graham E. Fagg. The internet backplane protocol : a study in resource sharing. *Future Generation Comp. Syst.*, 19(4) :551–561, 2003.

[10] Ghazi Bouabene. Sélection de pairs et allocation de tâches dans p2p-mpi. Master's thesis, Université Louis Pasteur de Strasbourg, June 2006.

[11] Franck Cappello et al. Grid'5000 : a large scale and highly reconfigurable grid experimental testbed. In *6th IEEE/ACM International Conference on Grid Computing (GRID 2005)*, pages 99–106, 2005.

[12] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid : a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*. IEEE Computer Society Press, March 2008.

[13] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The apples parameter sweep template : User-level middleware for the grid. In *Proceedings of SuperComputing*, 2000.

[14] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP - a practical model of parallel computing. *Communication of the ACM*, 39(11) :78–85, 1996.

[15] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia C. Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus : A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3) :219–237, 2005.

[16] Alexandre Denis, Christian Pérez, and Thierry Priol. Padicotm : an open integration framework for communication middleware and runtimes. *Future Generation Comp. Syst.*, 19(4) :575–585, 2003.

[17] Gilles Fedak, Haiwu He, and Franck Cappello. Bitdew : a programmable environment for large-scale data management and distribution. In *SC*, page 45. IEEE/ACM, 2008.

[18] Ian Foster and Nicholas T. Karonis. A grid-enabled MPI : Message passing in heterogeneous distributed computing systems. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (SC'98)*. IEEE Computer Society Press, November 1998.

[19] Ian Foster and Carl Kesselman. Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, 1997.

[20] Munehiro Fukuda and Jumpei Miyauchi. An implementation of parallel file distribution in an agent hierarchy. *The Journal of Supercomputing*, 47(3) :255–285, 2009.

[21] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI : Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[22] Jens Gustedt. Data handover : Reconciling message passing and shared memory. In José Luiz Fiadeiro, Ugo Montanari, and Martin Wirsing, editors, *Foundations of Global Computing*, number 05081 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[23] Ludovic Hablot, Olivier Glück, Jean-Christophe Mignot, Stéphane Genaud, and Pascale Vicat-Blanc Primet. Comparison and tuning of mpi implementations in a grid context. In *Proceedings of CLUSTER 2007*, pages 458–463, 2007.

[24] RW Hockney. The communication challenge for MPP : Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3) :389–398, 1994.

[25] Thilo Kielmann, Henri Bal, and Kees Verstoep. Fast measurement of LogP parameters for message passing platforms. In *Proceedings of the 4th Workshop on Runtime Systems for Parallel Programming*, LNCS Vol. 1800, pages 1176–1183, 2000.

[26] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe : MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8) :131–140, August 1999.

[27] Tevfik Kosar and Miron Livny. Stork : Making data placement a first class citizen in the grid. In *ICDCS*, pages 342–349. IEEE Computer Society, 2004.

[28] Dawid Kurzyniec, Vaidy S. Sunderam, and Mauro Migliardi. Pvm emulation in the harness metacomputing framework - design and performance evaluation. In *2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, pages 282–283. IEEE Computer Society, May 2002.

[29] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation : Practice and Experience*, 18(10) :1039–1065, 2006.

[30] Jason Maassen and Henri E. Bal. Smartsockets : solving the connectivity problems in grid computing. In Carl Kesselman, Jack Dongarra, and David W. Walker, editors, *HPDC*, pages 1–10. ACM, 2007.

[31] Thomas M. Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, R. Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna : a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17) :3045–3054, 2004.

[32] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. Modeling tcp reno performance : a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2) :133–145, 2000.

[33] Christophe René and Thierry Priol. Mpi code encapsulating using parallel corba object. In *HPDC*, 1999.

[34] Ian J. Taylor, Matthew S. Shields, Ian Wang, and Roger Philp. Distributed p2p computing within triana : A galaxy visualization test case. In *IPDPS*, page 16. IEEE Computer Society, 2003.

[35] Jie Wu, Earl C. Joseph, Richard Walsh, and Steve Conway. Worldwide technical computing server 2009–2013 forecast. Technical Report Doc #217232, IDC, 2009. http ://www.idc.com/getdoc.jsp ?containerId=217232.