HAL
open science

Expérimentation sur les nouvelles architectures : des processeurs multi-cœurs aux grilles de calcul

Brice Videau

To cite this version:

Brice Videau. Expérimentation sur les nouvelles architectures : des processeurs multi-cœurs aux grilles de calcul. Calcul parallèle, distribué et partagé [cs.DC]. Université Joseph-Fourier - Grenoble I, 2009. Français. NNT : . tel-00924386

HAL Id: tel-00924386

https://theses.hal.science/tel-00924386

Submitted on 6 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ JOSEPH FOURIER

Numéro attribué par la bibliothèque
|__/__/__/__/__/__/__/__/__/__/

THÈSE

pour obtenir le grade de

DOCTEUR DE L'Université Joseph Fourier

Spécialité : Informatique

préparée au Laboratoire d'Informatique de Grenoble
dans le cadre de ***l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique***

présentée et soutenue publiquement

par

Brice VIDEAU

le 28 septembre 2009

Titre :

Expérimentation sur les nouvelles architectures : des processeurs multi-cœurs aux grilles de calcul

JURY

M. RAYMOND NAMYST,	Rapporteur
M. THIERRY PRIOL,	Rapporteur
M. JEAN-CLAUDE FERNANDEZ,	Président du jury
M. MARTIN QUINSON,	Examinateur
M. JEAN-FRANÇOIS MÉHAUT,	Directeur de thèse
M. OLIVIER RICHARD,	Directeur de thèse

résumé : Les besoins en puissance de calcul ne cessent d'augmenter. Pour répondre à ces besoins, de nouvelles architectures sont apparues. Les grilles et grappes de calcul, qui permettent d'agréger la puissance de plusieurs machines et les processeurs multi-cœurs qui permettent d'offrir plus de puissance sans nécessité d'augmenter la fréquence et la consommation énergétique du processeur.
Cependant, l'étude de ces nouvelles architectures n'est pas aisée. Pour l'étude des grilles de calcul, nous disposons de plates-formes dédiées. Néanmoins, la conduite d'expérience sur ces plates-formes complexe est une tâche difficile à mettre en œuvre. Dans le cas des processeurs multi-cœurs, leur comportement est mal connu.
Dans cette thèse nous proposons deux outils dédiés à l'étude des nouvelles architectures. Le premier, Expo, est un logiciel de conduite d'expérience sur grille dédiée à l'expérimentation. Expo permet d'automatiser en partie la conduite d'une expérience, tout en essayant de garantir son bon déroulement. Expo propose également un langage concis de description d'expérience adapté à la problématique des grilles. Le second outil, PaSTeL, est dédié à l'étude de l'adéquation entre un support exécutif et une architecture. Hautement paramétrable, il permet d'étudier les multiples facettes d'une solution de parallélisation s'exécutant sur une architecture donnée.
Les deux outils ont été validés au cours de leur développement, mais également lors d'une campagne expérimentale visant à étudier le comportement d'un moteur de vol de travail sur différentes architectures multi-cœurs.

abstract : Needs in computing power are increasing steadily. In order to meet those needs, new architectures have appeared. Computing grids and clusters are gathering the computing power of several machines, and multi-core processors are offering more computing power without the need to increase the frequency and electrical power usage.
Nonetheless, The study of those new architectures is not an easy task. In order to study computing grids researchers have dedicated platforms. But the conduct of experiments on such platforms is a difficult process. In the case of multi-core processors, their behavior is not well known.
In this thesis we put forward two tools dedicated to the study of new architectures. The first one, Expo, is an experiment management software for grid architectures. Expo allows for automated experiment conduct while still trying to guarantee its sound unrolling. Expo also provides a language to describe experiments on grid infrastructures. The second one, PaSTeL, is dedicated to the study of interactions between architectures and execution strategies. Highly tunable, it grants user with means to study in depth the behavior of a parallel solution running on a given architecture.
Both tools have been validated during the development phase, but also during an experimental study of a work stealing engine on several multi-core architectures.

remerciements : Je tiens à remercier ici toutes les personnes qui m'ont aidé dans la réalisation de ce travail. Je pense plus particulièrement à mes directeurs de Thèse, Jean-François Méhaut et Olivier Richard, pour leur patience et leur bienveillance à mon égard. Ils ont, chacun à leur manière, su me motiver et me diriger dans les moments difficiles. Je tiens également à remercier les rapporteurs de ce travail, Raymond Namyst et Thierry Priol, d'avoir accepté de relire ma thèse. Leur analyse a grandement contribué à améliorer le résultat final. Finalement, je remercie les examinateurs, Jean-Claude Fernandez et Martin Quinson, d'avoir accepté de présider et de venir assister (parfois de loin) à ma soutenance.

J'ai bien entendu une pensée pour tout les membres du laboratoire (qui fut jusqu'à récemment) ID, et qui ont su transformer cette aventure en un véritable bonheur. Les innombrables discussions que j'ai pu avoir avec chacun d'entre eux ont réellement enrichi mon horizon scientifique. Je remercie le groupe de travail en théorie de jeux qui a su m'initier à cette science nouvelle pour moi. Pour conclure, j'ai une pensée particulière pour Erik Saule, coauteur d'une partie de mes travaux, c'était un réel plaisir de travailler avec lui.

Table des matières

Table des matières **viii**

1 Introduction **1**
1.1 Objectifs 2
1.2 Plan 2

2 L'expérimentation sur plates-formes d'expérimentation distribuées **5**
2.1 Les outils d'analyse 6
2.1.1 Modélisation mathématique 6
2.1.2 Simulation 7
2.1.3 Émulation 9
2.1.4 Tests sur plates-formes et environnements réels 12
2.2 Problématiques de l'expérimentation sur plate-forme réelle 13
2.2.1 Reproductibilité 13
2.2.2 Analyse 16
2.2.3 Rejouabilité 17
2.2.4 Archivage 18
2.2.5 Efficacité 19
2.2.6 Facilité d'utilisation 20
2.2.7 Évolutivité 21
2.3 Quelques gestionnaires d'expériences 21
2.3.1 Quelques plates-formes dédiées à l'expérimentation . . 22
2.3.2 PluSH 23
2.3.3 DART et Emulab 24
2.3.4 CLIF 24
2.3.5 ZENTURIO 24
2.3.6 Weevil 24
2.3.7 NXE 25
2.3.8 Conclusion 25

3 L'exploitation des processeurs multi-cœurs 27
3.1 Pourquoi les architectures multi-cœurs ? 27
3.1.1 Augmenter la puissance sans augmenter la consommation électrique 27
3.1.2 L'approche multi-cœur 28
3.2 Des machines très différentes 30
3.2.1 Quelques exemples d'architectures 30
3.2.2 Récapitulatif 36
3.3 Comment exploiter le parallélisme d'un processeur multi-cœurs ? 36
3.3.1 Solutions système 37
3.3.2 Solutions langage 38
3.3.3 Programmation multi-cœurs et bibliothèques 42
3.3.4 STAPL et PSTL 44
3.3.5 De nombreuses solutions 44
3.4 Comment étudier les plates-formes multi-cœurs 45
3.4.1 Quels critères étudier ? 45
3.4.2 Un banc d'essai pour architectures multi-cœurs 46
3.4.3 Conclusion 46

4 PaSTeL : un banc d'essai pour architectures multi-cœurs 49
4.1 Support d'exécution 50
4.1.1 Langage de programmation 51
4.1.2 Support système 52
4.1.3 Type de synchronisation 52
4.1.4 Allocation des ressources 53
4.1.5 Modèle exécutif 54
4.1.6 Conclusion 56
4.2 Exemple d'utilisation de PaSTeL 56
4.2.1 Implantation d'un sous-ensemble de la STL 56
4.2.2 Vol de travail de PaSTeL 57
4.2.3 Implantation de l'interclassement de deux tableaux . . 59
4.3 Modélisation de PaSTeL 67
4.3.1 Modélisation des threads 67
4.3.2 Modélisation de la boucle de travail 70
4.4 Analyse des performances 71
4.4.1 Plates-formes expérimentales et méthodologie 71
4.4.2 Évaluation brute du moteur 72
4.4.3 Validation de la modélisation 79
4.5 Discussion et conclusion 80

5 Expo : un outil de conduite d'expériences pour grilles 83
5.1 Une étude de cas 84
5.1.1 Le programme *mput* 84
5.1.2 Méthodologie 85
5.1.3 Détails expérimentaux 86
5.1.4 Plan expérimental 87
5.1.5 Résultats 89
5.2 Modélisation de la conduite d'expérience 91
5.3 Le moteur de conduite d'expérience *Expo* 93
5.3.1 Une architecture client serveur 93
5.3.2 Module d'exécution de commande 94
5.3.3 Lanceur parallèle 96
5.3.4 Module de réservation 96
5.3.5 Module d'expérience 96
5.3.6 Module de génération de commandes et langage de description d'expériences 97
5.4 Cas d'utilisation 98
5.4.1 Cas simple 98
5.4.2 Évaluation d'un outil de diffusion de fichiers 99
5.5 Conclusion 100

6 Expériences : évaluation de PaSTeL à l'aide d'Expo 105
6.1 Objectifs expérimentaux 105
6.1.1 Choix des paramètres 106
6.1.2 Paramètres écartés 108
6.1.3 Plan expérimental 109
6.2 Conduite expérimentale 110
6.2.1 Compilation 110
6.2.2 Script Expo 110
6.2.3 Format de sortie 112
6.2.4 Déroulement de l'expérience 112
6.3 Analyse des résultats 112
6.3.1 Comportement général sur Bordereau 114
6.3.2 Comportement général sur Genepi 116
6.3.3 Efficacités comparées 119
6.3.4 Influence du grain 121
6.4 Discussions et conclusions 124

7 Conclusion et perspectives 127
7.1 Perspectives à court terme 128
7.2 Perspectives à moyen terme 128

Bibliographie **139**

Annexes **141**

Table des figures

2.1 Les types d'expériences 6
2.2 Simulateur 7
2.3 Émulateur 9
2.4 Expérience fictive 16
2.5 Expérience fictive 17
2.6 Récapitulatif des moteurs d'expérience 26

3.1 Consommation et fréquence 30
3.2 Un ordinateur classique 31
3.3 Un ordinateur multiprocesseur SMP 32
3.4 Un ordinateur multiprocesseur NUMA 32
3.5 Un ordinateur avec processeur multi-cœurs 33
3.6 Un ordinateur avec processeur multi-cœurs dont le cache est partagé 34
3.7 Un ordinateur avec processeur multi-cœurs et interface mémoire intégrée 35
3.8 Un ordinateur avec processeur multi-cœurs et une structure de cache hiérarchique 35
3.9 Exemple d'utilisation d'OpenMP 38
3.10 Exemple d'utilisation de Fortress 39
3.11 Exemple d'utilisation de Java 7 40
3.12 Exemple d'utilisation de Cilk 41
3.13 Exemple d'utilisation de Cilk++ 41
3.14 Exemple d'utilisation d'Athapascan 42
3.15 Exemple d'utilisation de TBB 43

4.1 PaSTeL : un outil pour étudier le support d'exécution et l'architecture 50
4.2 Support d'exécution modulaire de PaSTeL 51
4.3 Allocation de ressources 53
4.4 Exemple d'utilisation de PaSTeL 54

4.5 Modèle de threads 55
4.6 Moteur de vol de travail de PaSTeL 58
4.7 Interface de programmation de PaSTeL 60
4.8 Algorithme *merge* de la STL 61
4.9 Données globales de l'algorithme *merge* 62
4.10 Données locales de l'algorithme *merge* 62
4.11 Procédure CalculerMorceau de *merge* 64
4.12 Fonction TravailLocal de *merge* 65
4.13 Fonction MiseAJourEtatGlobal de *merge* 65
4.14 Fonction TravailGlobal de *merge* 65
4.15 Procédure Vol de *merge* 66
4.16 Modèle de thread de PaSTeL. 68
4.17 Boucle de travail de PaSTeL 70
4.18 Algorithme *merge* sur *portable* 73
4.19 Algorithme *stable_sort* sur *portable* 74
4.20 Algorithme *min_element* sur *portable* 74
4.21 Algorithme *min_element* sur *portable*, détails 75
4.22 Algorithme *merge* sur *serveur* 76
4.23 Algorithme *stable_sort* sur *serveur* 77
4.24 Algorithme *min_element* sur *serveur* 77
4.25 Algorithme *min_element* sur *serveur*, détails 78
4.26 Performances relatives du mécanisme de prédiction. 81

5.1 Le pipeline de *mput* 85
5.2 Processus expérimental suivi 85
5.3 Plan factoriel fractionnaire d'expérience 88
5.4 Résultats de l'étude de cas 89
5.5 Une expérience pilotée par un moteur 92
5.6 Utilisation d'*Expo* sur grille légère 94
5.7 Architecture client serveur modulaire 95
5.8 Le module d'exécution de commandes 95
5.9 Liste des commandes disponibles dans *Expo* assortie d'une brève description 97
5.10 Une utilisation simple d'*Expo* dans le contexte de Grid'5000 . 98
5.11 Programme en langage dédié pour la réalisation de l'évaluation de l'outil de diffusion *Kastafior* 101
5.12 Résultat du temps de diffusion de fichier avec *Kastafior* en fonction de la taille du fichier et du nombre de machines en réception. La moyenne (reportée en seconde) et l'écart-type sont obtenus sur 10 évaluations. 102

6.1 Matériel utilisé 107
6.2 Plan expérimental : récapitulatif 109
6.3 Script Expo 111
6.4 Format de sortie de l'expérience 113
6.5 Agrégation des résultats 114
6.6 Algorithme *merge* sur Bordereau 115
6.7 Performances sur Bordereau 116
6.8 Algorithme *merge* sur Genepi 117
6.9 Performances sur Genepi 118
6.10 Efficacité de *merge* 120
6.11 Efficacité de *stable_sort* 122
6.12 Influence du grain Genepi 123
6.13 Influence du grain Bordereau 125

7.1 Algorithme *min_element* sur Bordereau 142
7.2 Algorithme *merge* sur Bordereau 143
7.3 Algorithme *stable_sort* sur Bordereau 144
7.4 Algorithme *min_element* sur Genepi 145
7.5 Algorithme *merge* sur Genepi 146
7.6 Algorithme *stable_sort* sur Genepi 147
7.7 Efficacité de *merge* 148
7.8 Efficacité de *stable_sort* 149
7.9 Efficacité de *min_element* 150

Chapitre 1

Introduction

Les besoins des utilisateurs en puissance de calcul ne cessent d'augmenter. Les modèles scientifiques de plus en plus complexes, en météorologie ou en sismologie par exemple, nécessitent des machines de plus en plus performantes. Mais, même les utilisateurs occasionnels ont dans une moindre mesure besoins de performances améliorées, pour lire des vidéos en haute définition ou pour utiliser des systèmes d'exploitation toujours plus exigeants.

À ceci est venu s'ajouter un problème physique : la consommation électrique des processeurs augmente plus vite que leur puissance de calcul. Ceci est notamment vrai quand on augmente leur fréquence de fonctionnement. Tous ces paramètres ont dirigé l'informatique dans deux directions orthogonales mais complémentaires.

Les super-calculateurs ont progressivement été abandonnés au profit de grappes et de grilles de calcul. En effet, ces ensembles d'ordinateurs grand public inter-connectés par un réseau, présentent un rapport performance sur prix plus intéressant que les machines dédiées. Dans le même temps, pour éviter de continuer à augmenter la fréquence des processeurs, les constructeurs ont commencé à produire des processeurs possédant plusieurs unité d'exécution distinctes. C'est ce que l'on appelle des processeurs multi-coeurs. Tout d'abord de simples processeurs mis côte à côte, les processeurs multi-coeurs ont évolué vers des structures plus complexes permettant une meilleure communication entre les coeurs.

Plus récemment encore, de nouveaux acteurs sont arrivés sur le marché du calcul haute performance : les constructeurs de cartes graphiques ont fabriqué des processeurs qui, bien que principalement dédiés à une utilisation de rendu, peuvent aussi servir de co-processeurs généralistes. Ces processeurs sont souvent accompagnés d'une mémoire dédiée et présentent des performances impressionnantes.

1.1 Objectifs

Pour exploiter efficacement ces nouvelles technologies, il est nécessaire de les étudier avec précision. En effet, le comportement de systèmes aussi complexes peut être difficile à prévoir. Ils peuvent tout à fait présenter des comportements inattendus.

Aussi, pour pouvoir étudier ces nouveaux systèmes, il peut-être utile de développer des outils adaptés. Nous avons concentré nos efforts sur deux aspects de ces nouvelles plates-formes. Tout d'abord sur la conduite d'expérience sur grille légère, c'est à dire des ensembles quasi-homogènes de machines dédiées à l'expérimentation. Notre attention s'est également portée sur l'étude des architectures multi-cœurs.

Nous nous sommes tout d'abord attachés à définir les besoins expérimentaux de ces nouveaux systèmes et à répertorier les solutions existantes. Cette étude nous a amenés à proposer deux outils dédiés à l'expérimentation. Chacun de ces outils a été validé pour vérifier sa bonne adéquation avec les besoins définis précédemment.

1.2 Plan

Le Chapitre 2 présente la problématique de l'expérimentation sur plate-forme distribuée. Les outils associés sont également étudiés. Une place particulière est réservée aux moteurs de conduite d'expériences existants, ainsi qu'aux plates-formes pour lesquelles ils ont été créés. L'adéquation de ces moteurs à la conduite d'expérience sur grille légère est évaluée dans ce même chapitre.

Le Chapitre 3 présente en premier lieu les enjeux des architectures multi-cœurs. Différentes architectures classiques de processeurs multi-cœurs sont ensuite répertoriées. Les différents types d'outils qui permettent d'exploiter ces architectures sont également présentés, ainsi que les exemples qui nous ont semblé pertinents.

Le chapitre 4 présente PaSTeL, un outil dédié à l'étude de l'adéquation entre les supports exécutifs et les architectures qui peuvent les accueillir. Les choix de conception de cet outil sont justifiés, et l'outil est validé au cours d'une étude expérimentale. Cette étude comporte une comparaison de ses performances avec celle de bibliothèques dédiées aux architectures multi-cœurs. Ces travaux ont donné lieu à deux publications en collaboration avec Erik Saule, l'une dans RenPar 2008 [SV08] et l'autre dans MuCoCoS'09 [VSM09]. Un rapport de recherche INRIA a également été publié [VSM08]. *Multi-Core Computing Systems* est un workshop qui s'est tenu

à Fukoaka, Japon dans le cadre de la conférence CISIS'09.

Le Chapitre 5 présente Expo, un moteur de conduite d'expériences pour grilles légères. Les besoins associés à la conception d'un tel outil sont d'abord définis au travers d'une étude expérimentale. La construction de l'outil, ainsi que les différents choix effectués au cours du développement, sont présentés. Le comportement de l'outil est ensuite validé expérimentalement. Ces travaux ont donné lieu à deux publications, l'une dans MetroGrid 2007 [VTR07] en collaboration avec Corinne Touati et l'autre dans CFSE'6 [VR08]. *Metrology for Grid Networks* (MetroGrid) est un workshop qui s'est tenu à Lyon dans le cadre de la conférence GridNets 2007.

Le Chapitre 6 présente une campagne expérimentale mettant en jeu à la fois PaSTeL et Expo. Cette campagne vise à caractériser le comportement de certains algorithmes fournis par PaSTeL, en fonction de nombreux paramètres différents. Cette expérience est rendue possible par l'utilisation d'Expo, qui permet d'en simplifier la mise en œuvre et d'assurer son bon déroulement. Les résultats de ces expériences sont également étudiés.

Chapitre 2

L'expérimentation sur plates-formes d'expérimentation distribuées

La taille des infrastructures informatiques n'a cessé de croître ces dernières années. Pour les grilles et les grappes de calcul, cela est dû à l'augmentation des besoins qui est supérieure à l'augmentation de la puissance d'une simple machine. Les grilles ou grappes de plusieurs milliers de nœuds sont devenues courantes [Top] et d'autres, encore plus grandes, sont prévues. Pour Internet, cela est simplement dû à l'arrivée continue de nouveaux utilisateurs. Conjointement, les applications pair-à-pair ou distribuées ont vu leur nombre multiplié. Les entreprises ont besoin de systèmes toujours plus performants pour répondre aux demandes toujours croissantes de leurs clients. Les réseaux de serveurs se sont donc également agrandis.

Pour exploiter efficacement ces infrastructures, il est nécessaire d'étudier ces grandes plates-formes distribuées [Tic98]. Pour ce faire, des instruments de grande taille ont été conçus. Par exemple, Grid'5000 [CDD+05], un réseau de grappes de calcul réparties sur le territoire français, a été créé dans ce but. La conduite d'expériences sur ce type d'architecture est un défi car le nombre de nœuds, les différentes configurations matérielles et logicielles ainsi que les différentes topologies de réseaux sont autant de paramètres à prendre en compte.

Ce chapitre débute par une présentation des différents outils d'analyse dont dispose l'informaticien pour étudier le comportement d'un système informatique. Il se poursuit par la présentation de la problématique de la conduite d'expériences sur une grande plate-forme distribuée (Section 2.2). Ensuite, les principales plates-formes expérimentales ainsi que leur moteur d'expériences seront étudiées (Section 2.3).

2.1 Les outils d'analyse

L'outil d'analyse que va choisir un chercheur pour étudier un problème va conditionner toute sa démarche scientifique. On peut distinguer quatre types d'outils d'analyse différents en informatique :

— La modélisation mathématique
— La simulation
— L'émulation
— Les tests sur systèmes réels

Chacun de ces types offre un compromis entre réalisme et coût de réalisation (cf figure 2.1).

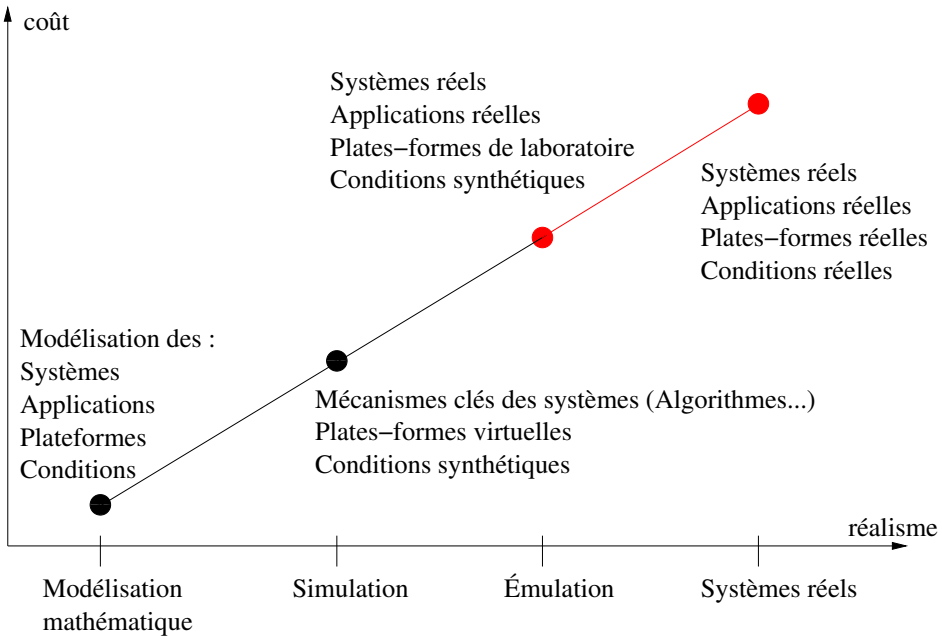


FIGURE 2.1 – Les types d'expériences

Une première problématique sera de choisir l'outil le plus adapté aux besoins de l'expérimentateur.

2.1.1 Modélisation mathématique

La modélisation consiste, comme son nom l'indique, à obtenir un modèle du système étudié sous forme d'équations ou d'autres objets mathématiques comme des files d'attente.

La modélisation mathématique permet d'étudier à faible coût une architecture ou des modèles qui ne sont pas encore implémentés, et permet d'éli-

miner un certain nombre de défauts avant de se lancer dans des processus plus coûteux. Mais la modélisation se prêtant relativement mal à l'expérimentation, nous ne la traiterons pas dans cette thèse.

2.1.2 Simulation

La simulation est une technique qui consiste à représenter le monde réel à l'aide d'un programme informatique (cf figure 2.2). En d'autres termes à remplacer le système réel par un système plus simple qui doit montrer un comportement proche du système réel.

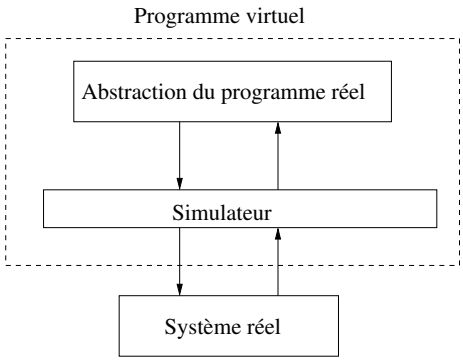


FIGURE 2.2 – Simulateur

La difficulté d'une simulation est qu'elle doit être suffisamment proche du réel pour qu'il soit possible d'en tirer des conclusions sur l'objet simulé. Les modèles doivent donc prendre en compte les détails des protocoles et des systèmes, sinon les conclusions de l'expérience ne seront pas applicables à la réalité.

Principe

Pour réussir à représenter le monde réel, il est possible d'employer plusieurs types de simulation, qui peuvent être plus ou moins bien adaptées aux problèmes que l'on voudra résoudre. Il existe trois types principaux de simulations :

— la simulation par événements discrets
— la simulation par agents
— la simulation continue

La simulation par événements discrets consiste à représenter un système comme un ensemble d'événements qui se produisent à des dates précises. Le système construit alors des files d'événements triés par l'instant où ils se

produisent. Le système exécute ensuite les événements un par un et ajoute les nouveaux événements générés. L'avantage de cette technique est qu'il n'est pas nécessaire de simuler les événements en temps réel. Il est donc possible de simuler le système aussi vite que nous le permet la machine de simulation.

La simulation par agents est un cas particulier de simulation par événements discrets. Le système est représenté par un ensemble d'agents qui interagissent entre eux. L'état d'un agent au temps $t + 1$ se déduit de son état au temps t et des interactions avec les autres agents par un ensemble de règles. L'avantage de cette méthode est qu'elle peut être très simple à concevoir et à implémenter. L'inconvénient est que son coût en temps de calcul peut rapidement devenir prohibitif.

Une simulation continue repose sur un ensemble d'équations différentielles auxquelles obéit le système au cours du temps. Si, au fur et à mesure de la simulation, on résout l'ensemble des équations avec un pas de temps suffisamment petit, on obtient, en sortie, un résultat continu. Ce type de simulation permet de simuler des systèmes complexes tels que le comportement mécanique de véhicules. Par contre, il est extrêmement coûteux en temps de calcul.

Comme les ordinateurs sont en grande partie des machines à états, et que les événements qui s'y produisent sont principalement discrets, ce sont les deux premiers types de simulation qui nous intéresseront par la suite.

Classiquement, les événements extérieurs qui sont utilisés au cours de la simulation peuvent être soit des traces d'événements existants, soit générés aléatoirement en fonction de la probabilité de l'événement.

Exemples

SimGrid SimGrid [LMC03] est un environnement de simulation basé sur des agents, et qui permet d'implémenter des simulateurs mixtes, à agents et à événements discrets. Il présente l'avantage d'être conçu pour simuler des programmes distribués et entre donc tout à fait dans le cadre de notre problématique.

Plusieurs interfaces de programmation ont été développées, pour répondre aux différents types de simulation que l'on pourrait vouloir réaliser, notamment une interface pour les programmes MPI, une pour les programmes à agents, et une générique. Il s'agit donc d'un programme très souple.

Network Simulator Network Simulator 2 (NS2) [BBE+99] est un simulateur qui utilise des événements discrets pour la simulation de protocoles réseaux. On peut donc simuler les protocoles classiques, UDP ou TCP par exemple, sur différents types de réseaux (sans fils, câblés etc...). La précision,

et donc le coût de la simulation, vont dépendre de la granularité désirée par l'expérimentateur. C'est le principal défaut de NS2 : il passe relativement mal à l'échelle. Pour permettre de réaliser des expériences plus conséquentes, il va falloir diminuer la granularité.

2.1.3 Émulation

"Une simulation utilisant du matériel ou un progiciel est appelée émulation." [Jai91a] Un émulateur duplique les fonctionnalités d'un système à l'aide d'un système différent, de manière à ce que le second système semble se comporter comme le premier. L'émulation consiste donc à exécuter une application réelle dans un environnement synthétique comme présenté dans la figure 2.3.

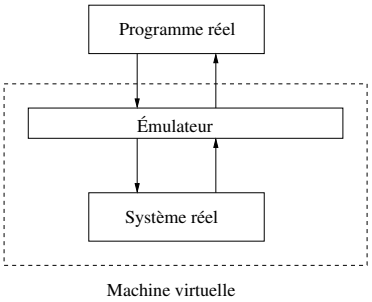


FIGURE 2.3 – Émulateur

Principe

Le but de l'émulation est d'exécuter le programme à tester sur un support qui n'est pas le support cible du programme. Par exemple, lors du développement d'un programme pour internet il n'est pas toujours possible ou souhaitable de réaliser des tests "grandeur nature". Il est possible de les réaliser en émulant le réseau internet sur le réseau disponible pour faire l'expérience. De même, l'absence de machine disponible utilisant un certain système d'exploitation est un problème qui peut être résolu en émulant une machine utilisant ce système. Si le problème vient du nombre de machines disponibles il est possible d'émuler plusieurs machines sur une seule si sa puissance est suffisante.

La difficulté de l'émulation consiste à reconstituer avec suffisamment de réalisme l'environnement dans lequel va s'exécuter l'application testée. Pour

cela, il faut pouvoir s'accommoder des limitations des logiciels existants. En effet, il est très coûteux de développer un émulateur.

L'inconvénient de l'émulation de machines est le coût du repliement, chaque machine émulée consommant de la mémoire. Ce coût en mémoire peut rapidement devenir prohibitif. Un certain nombre d'émulateurs tentent donc de diminuer ce coût en partageant au maximum les ressources (bibliothèques partagées entre les systèmes émulés...).

On peut répartir les émulateurs en plusieurs classes en fonction du type de système émulé.

Exemples d'émulateurs de réseau

Les émulateurs de réseau sont très nombreux et basés sur différentes approches. Ils sont principalement utilisés pour développer des applications distribuées. Ils permettent d'émuler une vaste gamme de conditions de réseau. On va donner ici les principaux émulateurs réseau disponibles en mettant en avant leurs caractéristiques principales.

Dummynet Dummynet [Riz97] permet d'émuler avec précision une topologie réseau simple, des pertes de paquets, la latence des communications, la bande passante allouée aux différents brins du réseau. Comme Dummynet intercepte les communications entre les couches transport et réseau, il est complètement transparent pour l'utilisateur. Les paquets interceptés sont traités en passant par un ensemble de files d'attente où ils peuvent être mélangés, retardés, perdus ou acheminés à l'étape suivante à un certain débit. On peut donc émuler un réseau relativement important sur une seule machine. Il est bien entendu également possible d'utiliser plusieurs machines.

ModelNet ModelNet [VYW+02] vise à émuler des topologies réseau complexes. Pour cela il divise les machines en deux classes :

— les machines utilisateurs, sur lesquelles tournent les programmes à tester,

— les machines serveurs, qui émulent la topologie du réseau, lien par lien.

Les machines utilisateurs sont habituellement émulées par des stations de travail connectées aux serveurs par un réseau haut débit. Les machines serveurs sont, pour leur part, émulées par une grappe de calcul interconnectée par un réseau très haut débit. ModelNet étant basé sur Dummynet, concernant les caractéristiques des liens et les routeurs du réseau, les capacités d'émulation sont similaires. Il apparaît que c'est une solution plus lourde que Dummynet, mais qui, en contrepartie, offre des résultats plus complets. Pour compenser cette complexité ModelNet rassemble les nœuds dédiés à l'émulation du

réseau dans un ensemble appelé cœur et les nœud dédiés à l'émulation des clients dans un ensemble appelé périphérie. Pour optimiser l'émulation du réseau, ModelNet effectue une phase dite de *distillation* qui permet d'obtenir un compromis entre le passage à l'échelle et la précision de la simulation. En pratique cette phase permet de limiter le nombre de serveurs que les paquets doivent traverser pour arriver à leur destination.

NetBed NetBed [WLS+02] est une solution un peu particulière car elle se situe à l'interface entre les émulateurs de réseau et les émulateurs de machines. Elle permet d'émuler, sur un réseau physique, un ensemble de machines, chaque nœud du réseau émulant une ou plusieurs machines. Mais NetBed permet en outre de simuler certaines parties des expériences, ou d'ajouter des nœuds physiques réels tels que des portables utilisant une interface sans fil. Pour gérer cet ensemble, une interface unique permet de gérer de la même manière les nœuds simulés, les nœuds émulés et les nœuds physiques. Comme l'infrastructure est assez large, elle est partagée par plusieurs utilisateurs. Le système gère donc les différents utilisateurs et les expériences sont exécutées quand le nombre de ressources disponibles devient suffisant.

P2PLab P2PLab [NR08] est une solution d'émulation légère destinée à l'étude des protocoles pair-à-pair. P2PLab combine l'émulation et la virtualisation pour obtenir un repliement maximal. La virtualisation s'effectue au niveau des processus, en associant à chacun une adresse IP différente, ce qui permet d'exécuter de nombreux nœuds virtuels (de l'ordre de la centaine) sur une seule machine. La topologie réseau est émulée via Dummynet et permet de régler des paramètres de débit et de latence entre différents sous-ensembles.

Ces échelles sont nécessaires car les protocoles pair-à-pair sont prévus pour supporter plusieurs dizaines de milliers de pairs. P2PLab a notamment été utilisé pour évaluer le protocole BitTorrent.

Exemples d'émulateurs de machine

Les émulateurs de machines permettent, sur une machine hôte, d'émuler d'autres machines, qualifiées de virtuelles. Si l'émulateur est performant, il peut alors exécuter différents systèmes d'exploitation, comme si l'on disposait en fait de plusieurs machines. Cela permet non seulement de réaliser des expériences entres les machines virtuelles, mais également de faire du développement beaucoup plus facilement. En effet, l'utilisateur a un contrôle total sur le système émulé.

Ce type d'émulateur utilise fréquemment une technique de virtualisation, qui consiste à partager les ressources d'une machine entre plusieurs environnements d'exécution. La virtualisation peut se faire à plusieurs niveaux (BIOS, noyau, ...). Les principaux outils de virtualisation disponibles seront présentés ici.

UML User Mode Linux [Dik00] permet de lancer plusieurs noyaux Linux sur une machine fonctionnant sous Linux. Chaque noyau peut donc être configuré différemment. Ces noyaux sont considérés comme des processus indépendants par le système hôte. C'est donc lui qui assure l'équité entre les différentes machines virtuelles.

Xen Xen [BDF+03] permet d'exécuter plusieurs systèmes d'exploitation en parallèle. Pour cela, il créé plusieurs machines virtuelles. A chacune de ces machines est allouée une partie des ressources disponibles. Xen permet de choisir différentes politiques de partage entre les machines virtuelles, et de ré-allouer dynamiquement les ressources.

Vserver Vserver [Pro] permet de créer plusieurs serveurs privés virtuels indépendants. Ces serveurs ne partagent que les ressources physiques. L'utilisateur d'un serveur privé virtuel ne peut en sortir pour affecter par exemple un autre serveur virtuel tournant sur la même machine. L'indépendance mémoire des serveurs est également assurée. L'intérêt de Vserver est sa légèreté. En effet, un seul noyau linux est utilisé par les différents serveurs. Seuls les applications et les systèmes de fichiers sont dupliqués. Il n'est donc pas nécessaire d'instancier un système d'exploitation complet pour chaque serveur.

2.1.4 Tests sur plates-formes et environnements réels

Le test sur système consiste à étudier une application réelle dans un environnement réel.

La difficulté des tests sur système réel est pratiquement à l'opposé de celle de la simulation. L'expérimentateur va souvent tenter de faire les mesures dans un cas le plus proche possible de l'idéal pour limiter les facteurs externes. Le but va être d'éviter que les mesures soient perturbées. Par exemple, une machine qui fonctionne mal peut fausser toute une série de mesures. Et même si le but consiste justement à étudier l'influence d'une machine en panne, l'expérimentateur devra surveiller qu'une seule machine est en panne, pour être certain de la qualité des observations.

La difficulté des tests sur système réel consiste donc principalement à s'assurer que les observations correspondent bien au comportement du système étudié, et ne sont pas dues à un effet parasite [DRT04]. Il faut donc évaluer l'impact de l'environnement où s'exécute le test, et pouvoir recréer des environnements connus pour avoir des résultats reproductibles.

C'est sur les tests de systèmes réels, en insistant sur les plates-formes distribuées que nous allons centrer la suite du chapitre.

2.2 Problématiques de l'expérimentation sur plate-forme réelle

La conduite d'expérience sur une plate-forme distribuée présente de nombreux défis. En effet, les problèmes classiques de l'expérimentateur sont présents, mais de nouveaux écueils apparaissent. Ils sont liés à la taille et à la nature de la plate-forme.

2.2.1 Reproductibilité

La reproductibilité est l'un des principes fondamentaux de la méthode scientifique, et se réfère à la capacité d'une expérience scientifique à être reproduite précisément par un expérimentateur indépendant.

La reproductibilité est le premier problème qui devrait occuper l'esprit de l'expérimentateur. Malheureusement, et particulièrement en informatique, c'est souvent celui qui est le moins bien traité.

Ce point est particulièrement capital car c'est lui qui peut garantir la qualité de l'étude expérimentale.

Dans le cadre de l'utilisation d'un instrument complexe, par exemple un ordinateur, pour faire une expérience il existe de nombreux paramètres cachés qui peuvent intervenir. Un ordinateur est un assemblage de composants logiciels et matériels qui interagissent entre eux.

Plusieurs centaines de logiciels s'exécutant simultanément sur la machine consomment des ressources matérielles. Cette consommation n'est à priori pas connue et peut être importante. Elle a un impact sur les performances de la machine, impact qui peut varier en fonction du type de ressources utilisées.

De même le matériel est soumis à des pannes. Ces pannes ne se caractérisent pas forcément par des interruptions de service mais peuvent également se traduire par des baisses de performances d'un ou plusieurs périphériques.

Lors d'une étude de performance, ces paramètres cachés peuvent avoir une influence non négligeable sur le résultat, soit en introduisant une varia-

bilité statistique qui n'a pas lieu d'être, soit en impactant suffisamment les performances pour fausser les conclusions de l'étude.

Il n'existe pas de solution idéale pour régler ce problème. Néanmoins, un certain nombre de précautions peuvent permettre de s'affranchir de la majorité des cas problématiques. Une solution possible est l'étalonnage de la plate-forme d'expérience. L'étalonnage consiste à obtenir des valeurs de référence des performances de la machine au moyen de programmes de test ou *benchmarks*. Avant de conduire une expérience, l'expérimentateur vérifie que la machine présente un comportement comparable à celui constaté lors de l'étalonnage.

Nous allons maintenant présenter quelques programmes de test classiques qui peuvent être utilisés pour étalonner différents composants d'une machine.

Benchmarks pour système de stockage

IOzone IOzone [NC] est un benchmark très complet pour système de fichiers. Il permet d'évaluer les performances en écriture, lecture, réécriture, etc. Il prend également en compte la taille des paquets écrits sur le disque. Le temps nécessaire pour rassembler ces informations est important. Cependant, il est possible d'envisager son utilisation comme benchmark disque avant une manipulation, en réduisant les tests effectués au strict nécessaire.

En tout cas, il devrait permettre de repérer un disque défectueux facilement.

Bonnie Bonnie est un benchmark pour système de fichier Unix. Il est maintenu depuis plus de 20 ans. Deux versions sont actuellement utilisables sur des machines modernes : Bonnie++ [Cok] et Bonnie64 [Bra]. Ces benchmarks sont des alternatives valables à IOzone qui est plus récent.

Benchmark pour réseau

netperf Netperf [Jona] est un programme permettant de tester de nombreux aspects d'une connexion réseau. Il permet d'utiliser des flux différents (TCP ou UDP par exemple). Il mesure également l'utilisation processeur au cours de son exécution. Il est basé sur une architecture client serveur : pour effectuer une mesure, un client se connecte à un serveur distant et envoie ou reçoit des données. Une fois installé, il est donc très facile d'utilisation.

netspec NetSpec [Jonb] est une infrastructure servant à évaluer les performances d'une architecture réseau soumise à une charge complexe. Contrai-

rement à Netperf qui s'axe sur une communication point à point, NetSpec utilise un ensemble de serveurs pour générer une topologie arbitraire.

Benchmark pour processeur et mémoire

SPEC CPU2006 SPEC [Cor] est l'acronyme de *Standard Performance Evaluation Corporation*. Ce groupe publie des benchmarks orientés vers le calcul haute performance. Les plus connus font partie de la suite de benchmarks CPU2006, et sont SPECINT et SPECFP qui mesurent respectivement les performances de la machine lors de calcul sur des entiers et sur des nombres à virgule flottante. Ces benchmarks sont assez longs et donc peu adaptés à une utilisation régulière. De plus, ils ne sont pas disponibles librement.

Benchmark pour machine parallèle

High Performance Linpack Benchmark HPL Benchmark [PWDC] est un benchmark de référence pour le calcul en mémoire distribuée. Il est utilisé pour classer les machines du TOP500 [DMS94], le classement des 500 machines les plus performantes du monde. Ce classement est disponible sur Internet [Top]. Il donne une mesure synthétique de la performance d'une plate-forme distribuée. Cette mesure permet donc, en partie, d'évaluer l'état de *santé globale* de la machine. Il ne permet pas d'évaluer individuellement les composants de la plate-forme.

NAS Parallel Benchmarks NPB [Sup] est un ensemble de benchmarks pour super calculateur, proposé par la NASA. Ces benchmarks sont destinés à couvrir un large spectre d'applications et de problèmes. Ils concernent la résolution d'équations différentielles, le tri de tableau, le calcul rapide de transformées de Fourier.

Les NPB ont pour ambition de représenter les performances que pourra obtenir l'utilisateur final de la plate-forme testée. Actuellement, en version 3, les NPB offrent des implantations de référence des algorithmes pour éviter que les vendeurs ne présentent des versions optimisée aux performances difficilement atteignables par les utilisateurs finaux.

Il existe également une version des NAS Parallel Benchmarks dédiée au grilles.

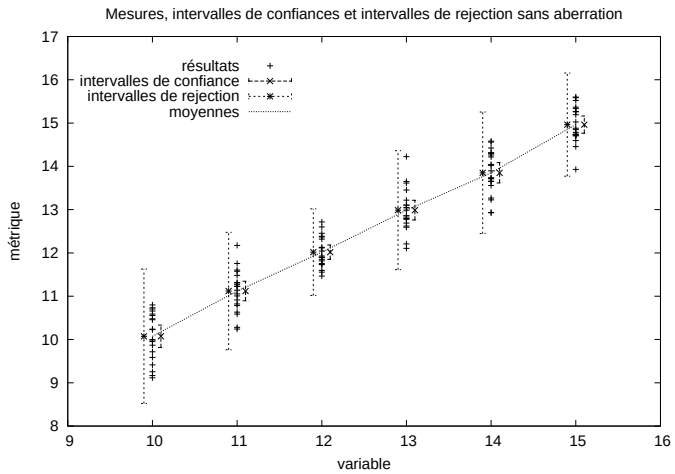


FIGURE 2.4 – Une expérience fictive avec des résultats distribués normalement.

2.2.2 Analyse

L'analyse est ici définie comme l'ensemble des traitements statistiques que vont subir les résultats d'une expérience scientifique.

L'analyse des résultats est intimement liée à la notion de reproductibilité. Dans le cadre d'une étude expérimentale sur une plate-forme réelle, il n'est pas suffisant d'exécuter une seule fois chaque expérience. En effet, si l'on répète une expérience, les résultats ne sont jamais exactement identiques.

Ces variations d'une expérience à l'autre sont très riches d'enseignements sur le comportement du système testé. Cependant, il faut savoir si ces variations sont dues à une variable cachée (cf 2.2.1) ou à un comportement non déterministe du système.

La moyenne ne suffit donc pas pour caractériser les performances d'un système. Pour caractériser les variations autour d'une moyenne, on utilise généralement des intervalles de confiance. Ces outils mathématiques sont décrits dans [Jai91b].

La Figure 2.4 présente des mesures dont les variations sont distribuées selon un loi gaussienne. Les intervalles de confiance à 95% sont représentés centrés autour de la moyenne. Quand deux intervalles de confiance de deux résultats se recouvrent, il n'est pas possible de déterminer avec certitude que les deux résultats sont significativement différents.

Sur la Figure 2.5, un point sort nettement de l'intervalle de confiance. Il

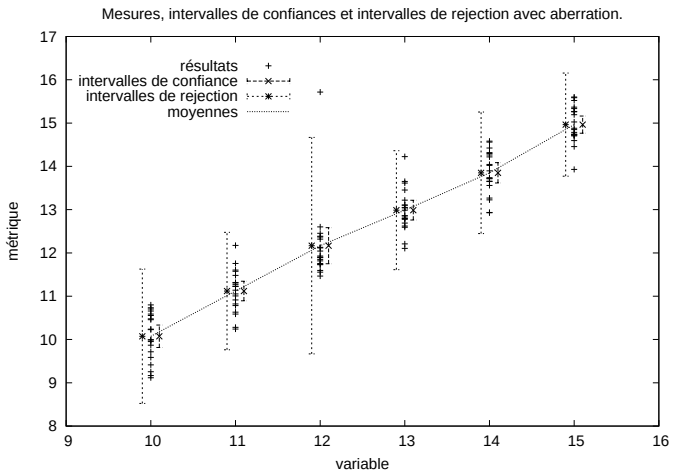


FIGURE 2.5 – Une expérience fictive avec des résultats distribués normalement et une valeur aberrante.

sort de l'intervalle de rejet. Cet intervalle est ici déterminé comme six fois plus large que l'intervalle de confiance. Ce point n'est pas a priori statistiquement significatif et doit être supprimé car il fausse les autres critères comme la moyenne ou la variance. Une fois supprimé ce point il est possible de recalculer la moyenne et les nouveaux intervalles de confiance. Lors de la réalisation d'une étude expérimentale, ce type de point peut apparaître quand le système a été perturbé. Il est donc parfois possible d'éliminer l'influence de variables cachées de cette manière.

2.2.3 Rejouabilité

La rejouabilité consiste à être capable de reproduire fidèlement une expérience et son environnement. En informatique cela revient à reproduire une expérience sur les mêmes ressources avec un environnement et des événements identiques. Alors que la reproductibilité consiste à mettre en place des conditions permettant d'obtenir des mesures représentant fidèlement le comportement du programme dans un environnement donné, la rejouabilité consiste à être également capable de reproduire l'environnement d'une expérience.

La rejouabilité d'une expérience est aussi un point critique. En effet, il n'est jamais possible de s'extraire complètement de l'environnement d'exé-

cution d'une expérience. Si un expérimentateur veut reproduire l'expérience d'un confrère, il va utiliser le matériel à sa disposition. En supposant qu'il dispose bien du même matériel, un certain nombre de facteurs peuvent encore influer sur le résultat de l'expérience. Par exemple la version du système d'exploitation, la version du compilateur ou les versions des bibliothèques utilisées. De plus, une bonne rejouabilité peut permettre de reproduire des conditions particulières permettant de générer des événements particuliers comme des fautes dans des programmes.

Malheureusement, ces quelques informations ne sont pas toujours suffisantes. De plus, la complexité des outils informatiques croissant sans cesse, l'environnement devient de plus en plus complexe. De nombreux facteurs essentiels peuvent néanmoins être négligés car ils doivent être considérés comme évidents ou sans influence.

La démarche scientifique est basée sur la capacité de reproduire fidèlement une expérience pour se convaincre soi-même de la véracité des résultats. Dans le cadre d'une démarche scientifique, et au vu de la complexité croissante de l'environnement des expériences en informatique, la nécessité de se doter d'outils pour analyser et sauvegarder les données d'un environnement apparaît clairement.

La rejouabilité est abordée dans le cadre de la conduite d'expériences de réseau dans [ESL07]. Les auteurs présentent un outil de conduite d'expérience qui permet de rejouer des expériences sur la plate-forme Emulab qui sera présentée en Section 2.3.1.

2.2.4 Archivage

Comme vu dans 2.2.3, la capacité de reproduire une expérience *à l'identique* est fondamentale. Cependant, les expériences devenant de plus en plus complexes, ces expériences peuvent prendre énormément de temps. Il n'est donc pas possible de toutes les reproduire.

Une alternative consiste à obtenir suffisamment d'informations pour se convaincre de la qualité de la démarche et pour pouvoir éventuellement reproduire une partie d'une expérience.

Pour ce faire, il est nécessaire d'enregistrer un maximum d'informations à chaque étape de l'expérience. Le simple enregistrement de l'environnement ne suffit pas. Quand l'expérience se déroule sur une plate-forme distribuée, la quantité d'informations à enregistrer croît dramatiquement.

En effet, il faut savoir quelles machines ont été utilisées au cours de l'expérience, le rôle qu'elles ont joué, la manière dont se sont enchaînées les commandes et les éventuels problèmes rencontrés. De même, les résultats bruts doivent être conservés pour que les traitements qu'ils ont subis puissent être

vérifiés. Ces résultats ne sont pas uniquement les métriques recherchées, mais également les entrées et sorties du programme et les fichiers générés au cours de l'exécution. De même, conserver les sources du programme utilisé, les commandes de compilation, et les binaires compilés paraît souhaitable.

La provenance des résultats est aussi un paramètre important [BKT00]. Il est capital de savoir qui à réalisé une expérience et d'où viennent les résultats associés. En effet toutes les sources ne sont pas forcément fiables.

Toutes ces informations peuvent prendre beaucoup de place et s'avérer complexes à structurer. Certaines parties d'une expérience pouvant être reproduites si un problème survient, il existe une notion de version ou de répétition associée à une mesure. De même, l'expérimentateur peut reproduire une expérience à l'identique sur différents systèmes pour les comparer. Beaucoup d'informations sont alors identiques pour les deux expériences, et ce sont les différences qui deviennent capitales.

Pour cette problématique également, le développement d'outil paraît indispensable. Par exemple, perfbase [Wor05] propose d'archiver les résultats d'expériences dans une base de donnée SQL. Un système de requête est utilisé pour faciliter le travail d'analyse et d'interprétation des données. PPerfGrid [HBMK05] est utilisé pour comparer les performances d'intergiciels sur grille. Il permet de mettre en relation des expériences utilisant à la fois des paramètres différents et des machines différentes. PPerfGrid met également en place un système de sonde pour obtenir des informations de performance au cours de l'exécution. PPerfGrid est un logiciel propriétaire.

2.2.5 Efficacité

L'efficacité d'une expérience est sa capacité à maximiser la quantité d'informations obtenues en minimisant le nombre et le coût des manipulations nécessaires pour la réaliser.

Bien qu'une plate-forme distribuée permette, dans une certaine mesure, de paralléliser les expériences l'utilisateur ne dispose pas pour autant de ressources illimitées. De plus, si l'on veut étudier en détail une application parallèle, de très nombreux paramètres sont à prendre en compte.

Il en résulte un problème de dimensionnement de l'expérience. Le nombre d'expériences nécessaires pour explorer de manière systématique un espace d'état croît exponentiellement avec le nombre de paramètres à étudier et géométriquement avec le nombre de valeurs que peuvent prendre ces paramètres. Dans de nombreux cas, notamment quand l'expérience est distribuée, le nombre d'expériences à réaliser est bien trop important.

C'est pour cela qu'il est nécessaire d'effectuer des expériences préliminaires afin de discriminer entre les paramètres qui ont une influence sensible

sur le déroulement de l'expérience et ceux qui n'ont qu'une influence négligeable.

Il existe des outils expérimentaux pour étudier un système contenant de nombreux paramètres. Par exemple la conduite de plans factoriels d'expériences permet d'obtenir un maximum d'informations en un minimum d'expériences. Les différents types de plans expérimentaux et les méthodes pour les analyser sont exposés dans [Jai91c]. Des exemples d'utilisation de ces plans peuvent être trouvés dans [JM96] et [Jac96].

2.2.6 Facilité d'utilisation

La condition d'acceptation de tout outil informatique est qu'il permette de transformer une tâche laborieuse en une tâche moins laborieuse. La conduite d'une expérience qui réponde aux différentes problématiques soulevées dans les sections précédentes est une tâche très laborieuse.

Par exemple, l'archivage nécessite la gestion de nombreux flux de données qui correspondent aux entrées et sorties des programmes. La rejouabilité nécessite de conserver de nombreuses données sur l'état de la plate-forme mais également de maintenir une structure hiérarchique contenant les informations d'exécution de l'expérience. La reproductibilité impose la mise en place de protocoles de tests et d'analyse de performances.

Il est donc nécessaire de mettre en place des solutions pour simplifier la conduite de la démarche expérimentale.

Une de ces solutions doit être l'automatisation d'un certain nombre de tâches. Néanmoins, cette automatisation ne doit pas être opaque. En effet, un faux sentiment de sécurité serait un vrai problème. Par exemple, l'exécution avant une expérience d'un ensemble de tests ne doit pas faire oublier que cet ensemble est forcément limité. Il peut ne pas convenir à l'expérience de l'utilisateur car il ne couvre pas la ressource utilisée au cours de cette expérience. Il peut être plus avantageux de fournir à l'utilisateur un ensemble de tests qu'il utilisera en fonction de ses besoins.

Une autre solution est la mise en place d'une interface adaptée à l'expression des besoins expérimentaux. Cette interface doit être à la fois suffisamment concise pour que le gain de temps soit conséquent, et assez souple pour ne pas limiter l'expérimentateur dans ses expériences.

La solution la plus utilisée consiste à intégrer un langage dédié au flux de travail que représente l'expérience. Les systèmes de *workflow* dédiés aux grilles de calcul sont comparés dans [YB05]. Nous présentons ici ceux qui nous ont paru les plus pertinents.

AGWL AGWL [FQH05] est le langage de workflow utilisé par l'environnement Askalon [FJP+05], un gestionnaire d'expériences pour grille Globus. AGWL utilise XML pour décrire les workflows. Les workflows écrits en AGWL se décomposent en une quinzaine d'activités qui permettent de décrire des workflows itératifs, parallèles ou séquentiels. Les workflows sont ensuite compilés par Askalon pour être exécutés.

XSCUFL Taverna [OAF+04] est le système d'édition et de gestion de workflow du projet myGrid [SRG03]. Le projet myGrid est prévu pour simplifier la conduite d'expériences de bio-informatique sur la grille. Taverna permet de gérer les expériences à travers une interface graphique. De par le type d'expériences visées, c'est à dire l'exécution de scripts d'analyse, Taverna n'a pas besoin de se soucier de rejouabilité ou de reproductibilité. Le langage de workflow sous-jacent est XSCUFL *Simple Conceptual Unified Flow Language* basé également sur XML.

2.2.7 Évolutivité

Un dernier point à prendre en compte est l'évolutivité de ces solutions. Les plates-formes expérimentales sont sujettes à de nombreuses évolutions. Par exemple, le gestionnaire de ressources de Grid'5000 a été récemment mis à jour et sa rétro-compatibilité n'est pas assurée. Il faut donc pouvoir s'adapter facilement à ces changements.

De même, il faut s'adapter aux besoins des différents utilisateurs, car ce qui satisfera les uns ne contentera pas forcément les autres. Pour certains, l'archivage est capital alors que pour d'autres, l'absence de perturbations prime. L'archivage pouvant perturber les mesures, ces critères peuvent s'avérer antagonistes.

Il est nécessaire pour un gestionnaire d'expériences d'être modulaire afin de s'adapter à la plate-forme et aux besoins des utilisateurs. Il doit aussi pouvoir être aisément modifié par un expérimentateur qui en ressentirait le besoin.

2.3 Quelques gestionnaires d'expériences

La conduite d'expériences a connu un grand développement conjointement aux grilles de calcul et aux grands systèmes distribués tels qu'Internet. En effet, comme décrit dans la Section 2.2, réaliser des expériences automatiques sur une plate-forme aussi complexe et volatile est difficile, mais

réaliser les mêmes expériences sans un moteur l'est encore plus. Chaque type de plate-forme possède donc son propre gestionnaire d'expériences.

2.3.1 Quelques plates-formes dédiées à l'expérimentation

Tout d'abord, il est nécessaire de présenter les plates-formes dédiées à l'expérimentation. Ce sont des instruments informatiques de grande taille dont le but est l'étude du comportement de différents systèmes informatiques. En ce sens, ils peuvent être comparés (dans une moindre mesure pour l'instant) à des instruments comme le Large Hadron Collider en physique. Ces instruments sont souvent distribués et composés de plusieurs centaines, voire plusieurs milliers, de nœuds.

Grid'5000

Grid'5000 [CDD+05] est une plate-forme expérimentale à grande échelle. Elle est reconfigurable en profondeur, possède une hétérogénéité mesurée et une puissante infrastructure de contrôle et de supervision. Elle cible l'étude d'intergiciels distribués pour grille ainsi que les systèmes pair-à-pair et les applications réparties. Elle est composée d'environ 5000 cœurs qui sont repartis en grappes sur toute la France et interconnectés par un réseau haut débit à 10 Gbit/s.

PlanetLab

PlanetLab [BBC+04] est un réseau d'environ 800 nœuds répartis sur toute la planète. Sa principale utilisation est la mise au point de nouvelles applications distribuées comme des applications pair-à-pair, ou destinées à développer de nouveaux protocoles réseau. PlanetLab utilise des machines virtuelles pour confiner les utilisateurs qui exploitent la même machine. Ces machines virtuelles sont appelées *slices* ou tranches.

Les particularités de PlanetLab et leur impact sur les pratiques expérimentales sont présentées dans [SPBP06]. L'objectif de PlanetLab n'est pas la reproductibilité, et encore moins la rejouabilité des expériences. En effet, les conditions du trafic réseau entre les machines partagées entre plusieurs utilisateurs, vont avoir une influence sur les résultats d'une expérience. Cependant, cet aspect est également une force de PlanetLab qui présente à l'utilisateur un environnement complexe et changeant censé représenter Internet. Il permet notamment de tester les programmes dans une large gamme de conditions, et donc d'évaluer leur robustesse.

Emulab

Emulab [WGN+02] est un banc d'essai utilisé pour développer des applications distribuées et des protocoles réseau. Emulab utilise l'émulation pour simuler des réseaux de grande taille avec des caractéristiques précises. Tout organisme peut installer Emulab sur des machines disponibles. Il existe des installations d'Emulab de plusieurs centaines de nœuds.

Emulab permet d'émuler une topologie arbitraire et permet d'obtenir un environnement réseau aux caractéristique précise. Les utilisateurs peuvent avoir un accès total aux machines qu'ils ont réservé. Il leur est donc possible de déployer le système d'exploitation de leur choix et de le configurer comme bon leur semble.

2.3.2 PluSH

PluSH [ATSV06] est un gestionnaire d'expériences pour PlanetLab. PluSH fonctionne grâce à une architecture client / serveur. Chaque machine utilisée dans une expérience accueille le client qui exécutera les commandes envoyées par le serveur. Les machines ne sont pas limitées à PlanetLab, le client pouvant être installé automatiquement sur n'importe quelle machine de type UNIX.

La description de l'expérience se fait à travers un fichier XML qui spécifie les ressources désirées, les différents logiciels à installer sur ces ressources ainsi que les différentes commandes qui seront exécutées. Ce fichier peut être généré via une interface graphique appelée NEBULA [Alb07] qui sert aussi à visualiser le déroulement d'une expérience sur PlanetLab.

Bien que destiné à des plates-formes différentes de PlanetLab, PluSH a été conçu avec PlanetLab comme trame de fond et cela se ressent dans son utilisation et son architecture. Par exemple, chaque ressource de l'expérience doit utiliser un client PluSH. Étant donné les programmes d'exécution à distance qui existent, cette restriction paraît superflue. De plus, ajouter un serveur sur une machine peut modifier son comportement. Ceci n'est pas important dans le contexte de PlanetLab où obtenir une des ressources ne garantit pas l'exclusivité sur une machine physique. Dans le contexte d'une grille légère, cela peut s'avérer plus gênant. Les expériences visées ne sont pas les mêmes. De même, l'installation du client prend un certain temps, et si on enchaîne des expériences courtes, le coût peut devenir prohibitif. Dans le contexte de PlanetLab où la majorité des expériences sont des expériences de *réseau*, et sont donc d'une certaine durée, ce n'est pas un handicap.

2.3.3 DART et Emulab

DART [Chu04] (*Distributed Automated Regression Testing*) est un gestionnaire d'expériences pour Emulab [WGN+02]. Comme son nom l'indique, il cible plus particulièrement les tests de non-régression d'applications distribuées. DART a été conçu pour des grappes, et n'est implanté que sur Emulab, ce qui le rend difficile à évaluer sur d'autres plates-formes.

Un nouvel outil de conduite d'expériences est présenté dans [ESL07] mais il est encore une fois conçu pour Emulab et les expériences de réseau. Cependant, il met l'accent sur la rejouabilité des expériences ce qui est un des enjeux fondateurs de la plate-forme.

2.3.4 CLIF

CLIF [Dil09] est un outil d'étude pour les plates-formes distribuées. Il est plus particulièrement destiné aux tests de montée en charge. Pour ce faire il utilise le modèle à composants logiciels Fractal [BCS]. Fractal permet de déployer les différents composants de CLIF. Ces composants sont par exemple des sondes, des serveurs d'injection, des consoles de commandes et des composants destinés à l'archivage des résultats.

L'une des originalités de CLIF est l'automatisation de l'injection de charge via la détection automatique des changements de régime. Ceci est rendu possible par la communication entre les différents composants.

2.3.5 ZENTURIO

ZENTURIO [PF02] est le gestionnaire d'expériences pour grilles de type Globus [FK97]. Composé d'une douzaine de modules, il est très complexe à installer et utiliser. De plus, il gère la compilation des programmes pour les paramétrer. Il se situe donc au niveau des sources de l'application. Il est également lié à l'intergiciel de gestion de grilles Globus, qui n'est pas le gestionnaire de ressources utilisé sur Grid'5000 et il ne vise pas les mêmes expériences : il est plus orienté vers les tests d'application que vers les tests d'intergiciels. Néanmoins, ZENTURIO offre une infrastructure complète pour tester des applications distribuées et un langage impératif de description d'expériences très précis.

2.3.6 Weevil

Weevil [WRCW05] est un gestionnaire d'expérimentations orienté vers l'évaluation de systèmes distribués. Il vise l'étude de systèmes plus complexes

qu'une simple architecture client et serveur. Pour cela, Weevil intègre un générateur automatique de charge de travail. Il se charge de déployer les applications, de générer la charge de travail et ensuite de récupérer les fichiers produits.

La description des expériences se fait dans un langage impératif utilisant l'expansion de macros. Les expériences sont ensuite compilées pour être exécutées par Weevil.

Weevil n'est pas destiné à une plate-forme particulière et il a été validé sur des grappes de machines et sur PlanetLab.

2.3.7 NXE

NXE [GP08] Network eXperiment Engine est un outil qui peut être rapproché de Weevil quand à ses objectifs. Cependant, il est destiné à Grid'5000 et la génération de charge se fait avec l'aide d'*iperf*.

Les expériences sont décrites en XML. NXE se charge de la configuration et de la réservation des ressources. Il initialise ensuite la topologie décrite et exécute l'expérience.

NXE aide également l'utilisateur en agrégeant les résultats, en calculant les métriques et en générant les graphiques associés.

2.3.8 Conclusion

La conduite d'expérience sur une plate-forme distribuée comme Grid'5000 confronte l'expérimentateur à de nombreuses problématiques différentes. De nombreux outils existent déjà et peuvent permettre de résoudre les problématiques prises indépendamment. Cependant, l'assemblage de toutes ces solutions en un unique outil aboutit systématiquement à une spécialisation.

Le Tableau 2.6 présente les spécialisations des différents moteurs présentés dans la section précédente. Il est notable qu'au niveau de la description d'expériences deux choix sont généralement faits : le langage impératif compilé (qu'il soit dédié ou non) et le langage dérivé de XML, qui lui, est interprété. Il n'existe pas de solution utilisant par exemple un langage dédié impératif qui soit interprété.

Ces différents moteurs d'expérience proposent tous des mécanismes évolués de contrôle de l'expérience. Mais, en l'état, aucun n'a été conçu pour régler les problèmes inhérents aux expériences conduites sur Grid'5000. En effet, les applications visées ne sont pas les mêmes et la reproductibilité visée des expériences n'est pas la même. C'est pour pallier ces inadéquations que le moteur d'expérience *Expo* a été conçu. Il est présenté en détails dans le chapitre 5.

Gestionnaire	Plate-forme ciblée	Expériences ciblées	Reproductibilité	Langage de description	Compilation
Weevil	Générique	Système distribué	non	Impératif	oui
PluSH	Planetlab	Système distribué	non	XML	non
Zenturio	Globus	Applications	oui	Impératif	oui
Dart	EmuLab	Applications	non	Impératif	oui
NXE	Grid'5000	Réseau	oui	XML	non

FIGURE 2.6 – Les principales caractéristique des moteurs de conduite d'expérience

De plus, les grilles de calcul ont pris une nouvelle dimension avec l'arrivée des processeurs multi-cœurs, chaque nœud de la grille devenant lui-même une machine parallèle. Pour comprendre en détail le comportement d'une application distribuée sur ces nouvelle plates-formes, il est nécessaire de s'intéresser également aux machine multi-cœurs.

Chapitre 3

L'exploitation des processeurs multi-cœurs

Dans ce chapitre, nous nous intéresserons aux architectures multi-cœurs et aux moyens de les exploiter. Notre objectif à terme est de comprendre comment programmer ces machines efficacement, et de définir un environnement de programmation se prêtant bien à l'expérimentation. Tout d'abord, nous allons expliquer le pourquoi de ces architectures. Nous présenterons ensuite les différents types de machines multi-cœur qui existent. Les environnements de programmation disponibles seront également présentés.

3.1 Pourquoi les architectures multi-cœurs ?

Récemment, les processeurs se sont transformés. Ils sont devenus plus sophistiqués. Les raisons de cette évolution sont présentées dans cette section.

3.1.1 Augmenter la puissance sans augmenter la consommation électrique

Le modèle économique de l'informatique est basé sur la loi de Moore. Cette loi stipule que tous les 18 mois [Sch97] la puissance de calcul des processeurs double. Cette augmentation est principalement due aux progrès des procédés de gravure des processeurs. En effet, une gravure plus fine d'une puce permet d'augmenter sa fréquence, au prix d'une augmentation de la consommation électrique. Plus qu'une loi, c'est un mantra pour l'industrie informatique qui a toujours tenu à suivre la loi de Moore au plus près. En effet, cela justifie le renouvellement du parc informatique tous les trois ans, les nouveaux ordinateurs étant 4 fois plus puissants que les anciens. Les

concepteurs de logiciels ont tiré avantage de cette progression pour écrire du code de plus en plus riche et lourd à exécuter, avec la garantie que bientôt des machines assez puissantes pour le faire tourner ne manqueraient pas d'arriver.

Ce modèle d'évolution des architectures s'est assez bien vérifié pendant plus de 20 ans, les générations de processeurs se succédant régulièrement au fur et à mesure des progrès de la lithographie. Cependant, si les premiers processeurs ne dissipaient pas de grandes quantités de chaleur, cela n'a pas été le cas des générations suivantes. Les premières puces ne disposaient pas de système de dissipation thermique. Ensuite, un dissipateur passif (radiateur) a été ajouté. Lorsque ce dissipateur est devenu insuffisant, les constructeurs lui ont adjoint un ventilateur. La taille de l'ensemble ventilateur et radiateur n'a cessé de croître jusqu'à l'année 2004. À cette date, le moindre processeur consommait au moins 100 watts et la dissipation de toute cette chaleur est devenue problématique. Conjointement, les constructeurs ont intégré du parallélisme à l'intérieur du processeur pour rendre ces dernier plus efficaces. Ceci a été fait par le biais de l'utilisation de pipeline d'instructions, de parallélisme d'instructions (ILP) ou d'architectures superscalaires. Mais ces méthodes ont rapidement trouvé leurs limites.

Confrontés à l'impossibilité d'augmenter facilement la fréquence des processeurs, les constructeurs n'ont eu qu'un choix possible pour maintenir le modèle économique : effectuer encore plus d'opérations à la même fréquence. Il existe deux moyens d'y parvenir, optimiser l'architecture pour la rendre plus efficace et effectuer du travail en parallèle. Bien entendu les constructeurs ont choisi les 2 voies simultanément. Les processeurs sont devenus plus efficaces, et, de plus, ils sont devenus polycéphales. En fait un processeur physique contient maintenant plusieurs processeurs appelés cœurs. Au début ces cœurs n'étaient que deux processeurs accolés et intégrés dans un même emballage, mais ensuite les architectures sont devenues plus complexes pour permettre une meilleure communication entre les cœurs.

3.1.2 L'approche multi-cœur

L'intérêt du processeur multi-cœurs ne s'arrête pas là. Pour le comprendre, il faut regarder plus en détail la relation qui existe entre la fréquence de fonctionnement du processeur, le voltage d'entrée nécessaire à ce fonctionnement et la dissipation thermique que l'on peut y associer.

En effet, la relation entre fréquence et consommation n'est linéaire que si l'on considère que la tension ne varie pas. Pour obtenir une fréquence plus importante il est nécessaire d'augmenter la tension de fonctionnement. La relation entre le couple fréquence/tension et la consommation est alors cubique.

La fréquence de fonctionnement d'un processeur est définie par la fréquence d'horloge du bus multiplié par un coefficient. Ce coefficient est appelé *coefficient multiplicateur*. Ce coefficient peut varier et permet donc de modifier la fréquence d'un processeur en cours de fonctionnement.

En prenant pour exemple la dernière génération de processeurs Intel, l'architecture baptisée *Nehalem*, on obtient le modèle thermique (TDP : *Thermal Design Power*) suivant :

$$TDP[coeff] = \left(\frac{coeff}{coeff_{max}}\right)^3 * TDP_{core} + TDP_{uncore} \tag{3.1}$$

Pour le processeur à 2.93 Ghz possédant 4 cœurs, le coefficient $coeff$ peut varier de 12 à $coeff_{max} = 22$, la fréquence externe étant de 133 MHz. $TDP_{uncore} = 20Watts$ est un facteur constant qui représente la consommation des éléments dont la fréquence ne varie pas. $TDP_{core} = 110Watts$ représente la consommation maximale de la partie dont la fréquence peut varier. Lorsque le coefficient du processeur est réglé sur 12 qui correspond à la fréquence la plus basse, le processeur consomme environ $40Watts$, alors que lorsque le coefficient est à 22 le processeur consomme $130Watts$, alors qu'il fournit moins de 2 fois plus de puissance de calcul. Ce modèle considère que tous les processeurs fonctionnent à la même fréquence. Maintenant, en considérant que les coefficients des cœurs peuvent varier indépendamment :

$$\begin{aligned} &TDP[coeff_1, coeff_2, coeff_3, coeff_4] = \\ &\quad \left(coeff_1^3 + coeff_2^3 + coeff_3^3 + coeff_4^3\right) \frac{TDP_{core}}{4 * coeff_{max}^3} + TDP_{uncore} \end{aligned} \tag{3.2}$$

S'il n'est possible d'exploiter qu'un cœur, il faut alors utiliser une fréquence maximale pour ce dernier, en laissant les autres à basse fréquence. Cela donne un consommation électrique de $60Watts$. Un quart de la puissance de calcul maximale de la puce est obtenue. Cependant, s'il est possible d'utiliser deux coeurs, le quart (un peu plus, en fait) de la puissance de calcul devient accessible pour $20Watts$ de moins. S'il est possible d'exploiter les quatre cœurs, alors la moitié de la puissance de calcul de la puce est disponible pour ces mêmes $20Watts$. Il devient plus intéressant d'utiliser plusieurs cœurs à basse fréquence qu'un cœur à haute fréquence. Il n'est en revanche pas intéressant de l'utiliser à plein régime et de l'arrêter et ce, même si seulement la moitié de la puissance de calcul est nécessaire. L'arrêt du processeur réduit sa consommation à $0Watts$, mais en moyenne on consomme dans ce cas $65Watts$ alors que si on utilise les quatre cœurs à mi-régime, seuls $40Watts$ sont consommés. Ces exemples sont résumés dans le Tableau 3.1.

Nombre de cœurs utilisés	4	1	1	2	4
Coefficient des cœurs utilisés	22	22	22	12	12
Coefficient des cœurs inutilisés		22	12	12	
Puissance de calcul	1	0.25	0.25	0.27	0.55
Consommation	$130W$	$130W$	$60W$	$40W$	$40W$
Efficacité	1	0.25	0.54	0.89	1.77

FIGURE 3.1 – Quelques couples de fréquence de fonctionnement et de consommation dans un processeur.

L'efficacité est le rapport entre la puissance de calcul et la consommation, ramené à l'efficacité du processeur utilisant 4 cœurs à plein régime.

Le pari du multi-cœurs consiste en ceci : avec une bonne parallélisation on doit pouvoir obtenir des processeurs qui consomment peu tout en restant performants. Mais, pour ce faire, il faut obtenir une bonne parallélisation, c'est à dire une exploitation efficace de plusieurs cœurs et ceci même dans le cas de petites tâches.

3.2 Des machines très différentes

Cependant, il n'existe pas un unique processeur multi-cœurs. En réalité, il en existe de nombreuses versions différentes. Et chacune de ces versions présente un comportement différent. Dans cette section, nous allons donc observer les principales différences que l'on peut noter entre les différents processeurs multi-cœurs.

3.2.1 Quelques exemples d'architectures

Dans cette section, nous allons présenter les différents type d'architectures que le programmeur d'une application peut-être amené à rencontrer.

Un couple processeur mémoire classique

La Figure 3.2 présente schématiquement un ordinateur utilisant un processeur mono-cœur. Cet ordinateur est composé d'un processeur et d'une

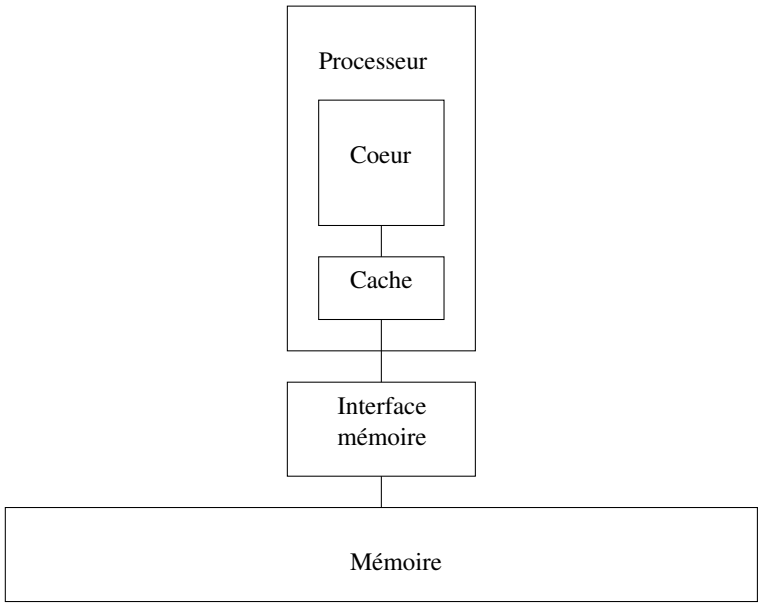


FIGURE 3.2 – Un ordinateur classique

mémoire reliés ensemble par une interface appelée bus mémoire. Le processeur est composé du cœur qui exécute les instructions et d'une petite mémoire appelée *cache* qui permet d'accélérer certains accès à la mémoire. Ce modèle a été à la base des ordinateurs de bureaux pendant près de 20 ans.

Exemple : Pentium III [Inte], Atom [Inta].

Un ordinateur multiprocesseur SMP

La Figure 3.3 présente un ordinateur multiprocesseur. Dans ce cas plusieurs processeurs sont reliés à la mémoire par une même interface. Le rôle joué par chaque processeur est symétrique. On appelle ces architectures SMP pour *Symmetric MultiProcessing*. L'accès à la mémoire est uniforme pour chaque processeur. Si les processeurs étaient différents, ou ne jouaient pas le même rôle, on appellerait cette architecture ASMP pour *ASymetric MultiProcessing*.

Exemple : Pentium III Xeon [Intf].

Un ordinateur multiprocesseur NUMA

La Figure 3.4 présente un ordinateur multiprocesseur qui possède plusieurs bancs mémoires. On peut remarquer que les parties droite et gauche sont identiques à la figure précédente. On peut donc considérer que l'on a

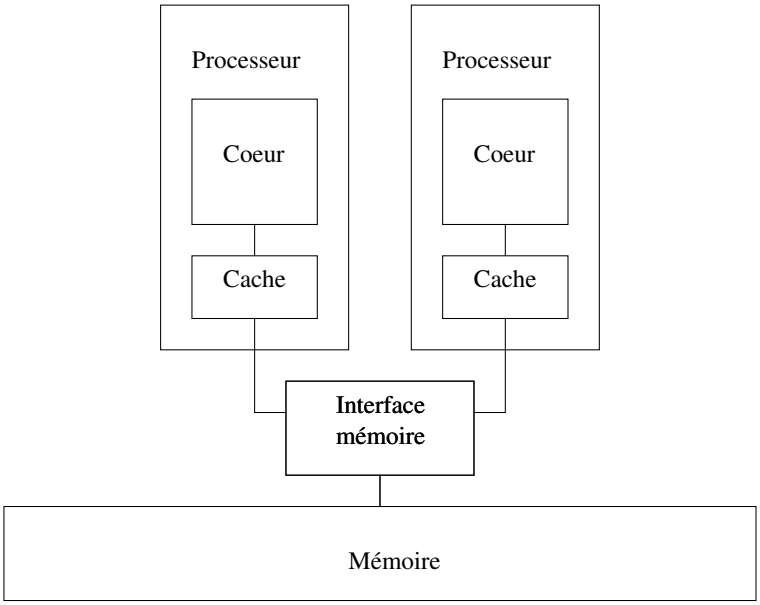


FIGURE 3.3 – Un ordinateur multiprocesseur SMP

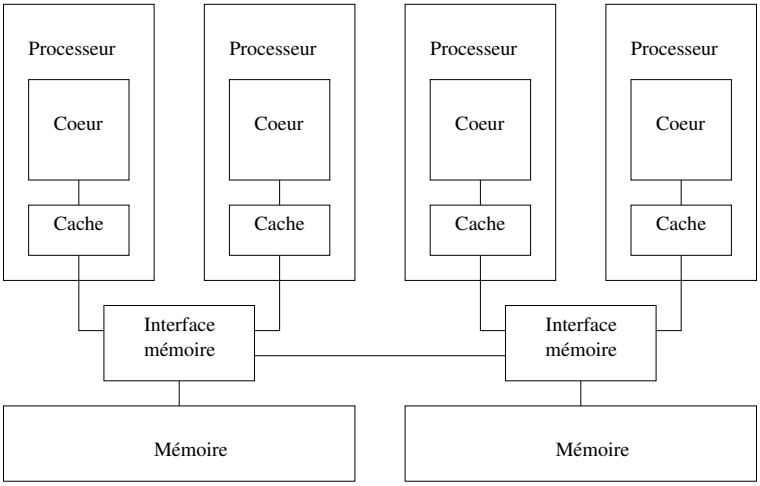


FIGURE 3.4 – Un ordinateur multiprocesseur NUMA

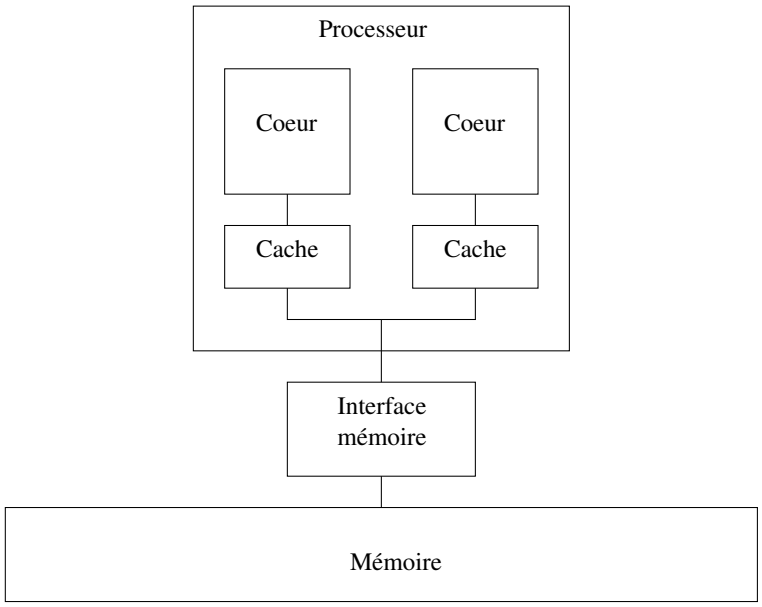


FIGURE 3.5 – Un ordinateur avec processeur multi-cœurs

ensembles SMP reliés ensembles. Par contre, l'accès à la mémoire de droite par l'ensemble de gauche nécessite de traverser 2 interfaces mémoires au lieu d'une. L'accès à la mémoire n'est donc pas uniforme. On appelle ces architectures NUMA pour *Non-Uniform Memory Access*.

Un ordinateur avec processeur multi-cœurs

La Figure 3.5 présente un ordinateur doté d'un processeur multi-cœur simple. Chaque cœur possède son propre cache. L'ensemble est connecté à l'interface mémoire. Ce modèle est a priori plus performant que le modèle SMP car la connexion entre les coeurs se fait directement plutôt qu'à travers l'interface mémoire.

Exemple : Pentium D [Intd].

Un cache partagé

La Figure 3.6 présente un ordinateur doté d'un processeur multi-cœurs possédant un cache partagé. Cela permet d'améliorer les performances quand les cœurs travaillent sur les mêmes données. Cependant, il faut que le cache soit bien dimensionné, car il ne doit pas souffrir de contention lors de lecture ou d'écriture et être suffisamment grand pour que les cœurs n'effacent pas mutuellement leurs données.

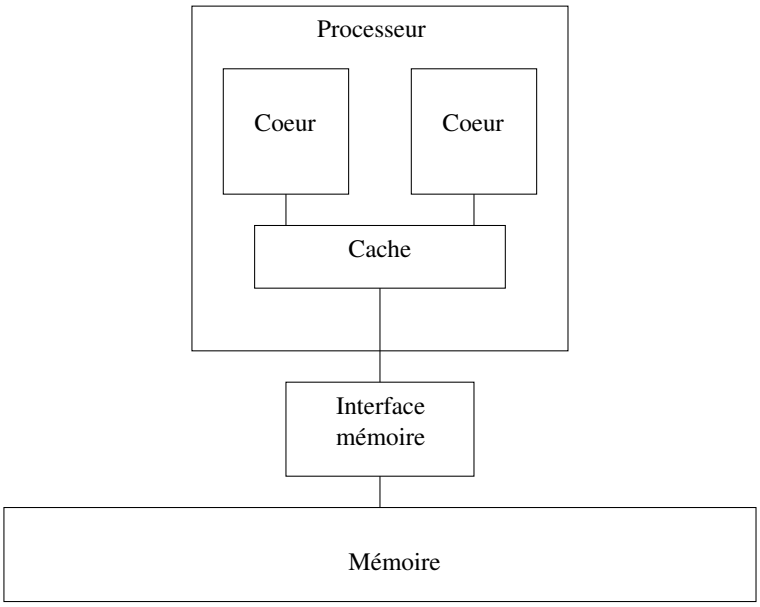


FIGURE 3.6 – Un ordinateur avec processeur multi-cœurs dont le cache est partagé

Exemple : Core 2 Duo [Intb].

Une interface mémoire intégrée

La Figure 3.7 présente un ordinateur dont l'interface mémoire est intégrée au processeur. Cela permet de réduire sensiblement la latence d'accès à la mémoire et donc d'améliorer les performances. De plus, il devient inutile d'utiliser le composant externe qui faisait la liaison entre le processeur et la mémoire.

Exemple : Opteron de seconde génération [AMDa].

Une structure de cache hiérarchique

La contention sur le cache peut devenir un problème si l'on augmente de manière trop importante le nombre de cœurs partageant le même cache. Un moyen de régler ce problème consiste à adjoindre à chaque processeur son propre cache qui absorbe une partie des requêtes et limite donc l'accès au cache principal [HIM+05]. Cette architecture est représentée sur la Figure 3.8.

Cette architecture permet également de désactiver le cache d'un cœur inutilisé, pour économiser une partie de l'énergie.

Exemple : Opteron de troisième génération [AMDb], Core i7 [Intc].

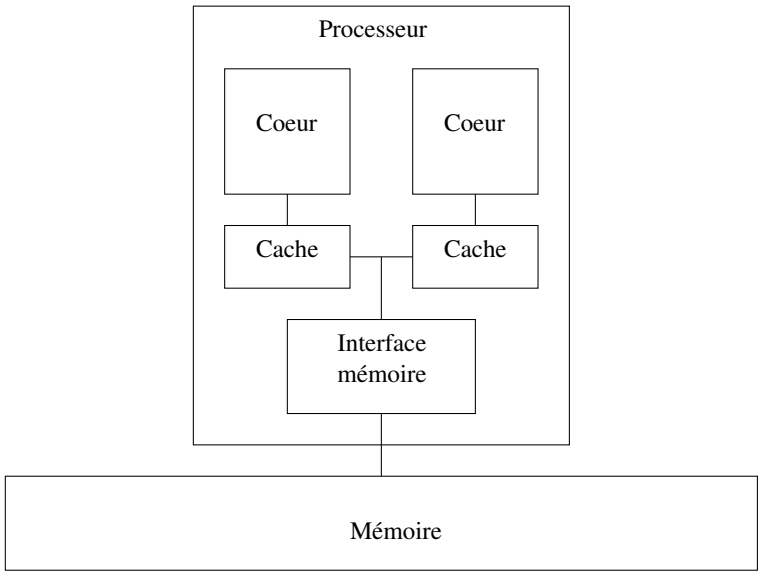


FIGURE 3.7 – Un ordinateur avec processeur multi-cœurs et interface mémoire intégrée

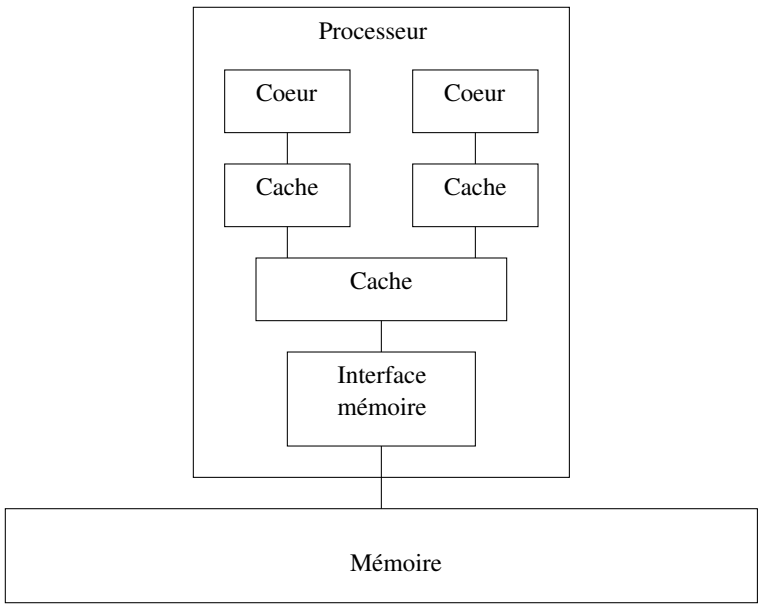


FIGURE 3.8 – Un ordinateur avec processeur multi-cœurs et une structure de cache hiérarchique

3.2.2 Récapitulatif

Il est bien entendu fréquent de voir combinées les différentes architectures présentées ici, par exemple des ordinateurs simultanément NUMA et multi-cœurs. Les performances de ces machines vont dépendre des performances du ou des cœurs, mais également des différents maillons de la chaîne qui relient les cœurs à la mémoire.

Dans une plate-forme comme Grid'5000, presque toutes les architectures présentées ici sont utilisées.

3.3 Comment exploiter le parallélisme d'un processeur multi-cœurs ?

Malheureusement, en faisant le choix de l'architecture multi-cœurs, les constructeurs se sont déchargés d'une partie du travail. En effet, ils fournissent une technologie qui n'est pas facile à utiliser pour un informaticien non spécialisé dans la programmation parallèle. Car, de manière générale, les utilisateurs n'effectuent pas un grand nombre de tâches en parallèle. Si deux cœurs sont assez facilement justifiables, quatre ou plus le sont difficilement. Or des processeurs à 4 cœurs et plus commencent à arriver sur le marché, et certains constructeurs présentent déjà des prototypes à 80 cœurs [Rat]. Il est fort probable de disposer sous peu de processeurs généralistes contenant plusieurs centaines de cœurs [SCS+08] [Bor07]. Dans les faits, les adaptateurs graphiques modernes embarquent des processeurs constitués de centaines de cœurs spécialisés [NVI].

Comme les utilisateurs manquent d'applications à faire tourner en parallèle, il va être nécessaire de transformer autant que faire se peut les applications séquentielles en applications parallèles. Mais pour y parvenir efficacement il est nécessaire de bien connaître les processeurs et les architectures disponibles.

Or tout les processeurs multi-cœurs ne présentent pas les mêmes caractéristiques. Cela est vrai d'une marque à l'autre, mais c'est également le cas entre les différents modèles d'un même constructeur. Ces caractéristiques influent directement sur les performances des applications qui utilisent ces processeurs. Certaines architectures sont même tellement différentes qu'il faudrait les programmer différemment pour parvenir à les exploiter efficacement.

Pour exploiter les machines parallèles, différentes solutions ont été utilisées. Ces solutions peuvent être séparées en trois classes. Une première classe de solutions de niveau système offre aux couches supérieures des abstractions de flots d'exécutions. Une seconde de niveau langage permet de décrire le pa-

rallélisme dans le langage de programmation utilisé. Finalement une troisième de niveau bibliothèque qui fournit certain nombre de services rassemblés au sein d'une bibliothèque.

3.3.1 Solutions système

Pour permettre à un développeur d'application d'exploiter le parallélisme d'une machine, il est nécessaire que le système sous-jacent offre une abstraction du parallélisme, sinon, le programmeur serait contraint de développer un pilote pour utiliser spécifiquement le matériel ciblé.

Parallélisme de processus

La première solution utilisé pour offrir du parallélisme au sein d'un système d'exploitation est le parallélisme de processus [SGG04]. Les processus sont des flots d'exécution indépendants qui ne partagent pas directement de la mémoire. Pour communiquer, ils utilisent soit des messages, soit des zones de mémoire auxquelles les différents processus peuvent accéder via le système d'exploitation.

Le passage de message est particulièrement bien adapté aux machines qui ne peuvent pas aisément partager de la mémoire, comme les grappes de machines ou les machines à mémoire distribuée.

Parallélisme de thread

Une autre solution consiste à offrir du parallélisme de threads. Les threads d'un même processus partagent une partie de leur mémoire. Il n'est donc pas nécessaire de les faire communiquer via des messages.

Il existe différentes normes de threads. Dans le monde UNIX et ses dérivés, l'interface standard couramment admise est la norme POSIX [IEE]. Dans Linux, une des premières implantations de la norme a été faite sous forme de bibliothèque [Mue93]. Actuellement elle est supplantée par la NPTL [DM03] *Native POSIX Thread Library*. Solaris utilise des threads un peu différents, les LWP [KVE+92] pour *Light Weight Process* et propose, au dessus, des threads POSIX.

Il existe également des bibliothèques de threads qui s'éloignent de la norme POSIX. Par exemple, la bibliothèque Marcel [Dan04] permet d'ordonnancer des threads en espace utilisateur et d'obtenir de meilleures performances qu'avec l'ordonnanceur du noyau. Une extension de Marcel [Thi07] permet d'exprimer les affinités enter les threads pour les répartir efficacement

```
int fib_parallel(int n){
        int x, y;
        if (n < 2) return n;
        #pragma omp task default(none) shared(x,n)
        {
                x = fib_parallel(n-1);
        }
        y = fib_parallel(n-2);
        #pragma omp taskwait
        return (x+y);
}
```

FIGURE 3.9 – Exemple d'utilisation d'OpenMP 3 : le calcul du n-ième terme de la suite de Fibonacci

sur une machine NUMA par exemple. Il faut noter cependant que Marcel offre une couche de compatibilité Posix.

3.3.2 Solutions langage

Une autre manière d'exploiter le parallélisme consiste à intégrer un support du parallélisme dans le langage de programmation.

High Performance Fortran

High Performance Fortran ou HPF [Lov93] est une extension de Fortran 90 qui permet d'exploiter le parallélisme d'une machine. Il permet d'exploiter différents types d'architectures via un compilateur spécifique. Par exemple, pour une machine multiprocesseur, il génère un code utilisant plusieurs threads et communiquant par passage de messages.

HPF est désormais tombé en désuétude et est supplanté par des extensions comme OpenMP.

OpenMP

OpenMP [DM98] est une extension du Fortran, du C et du C++. Il permet d'indiquer au compilateur les parties qui peuvent être parallélisées en spécifiant notamment les contraintes mémoire. OpenMP est assez simple et laisse une certaine latitude au programmeur quand à la manière de paralléliser son application. OpenMP existe aujourd'hui en version 3 [Boa]. La

```
fibonacci(n: ZZ) : ZZ =
        case n of
                0:1 => n
                else => fibonacci(n-1) + fibonacci(n-2)
        end
```

FIGURE 3.10 – Exemple d'utilisation de Fortress : le calcul du n-ième terme de la suite de Fibonacci

Figure 3.9 présente l'utilisation d'OpenMP pour décrire un calcul parallèle de la suite de Fibonacci.

Il existe de nombreuses implantations d'OpenMP. Ces implantations sont liées à la fois au compilateur utilisé et au système d'exploitation. Leurs performances sont très variables. En effet, les performances dépendent du parallélisme sous-jacent utilisé, et de la manière dont il est exploité. Par exemple GOMP [Nov06], le support OpenMP de GCC utilise des threads POSIX. En utilisant les threads Marcel et l'ordonnanceur hiérarchique présentés plus haut à la place des threads POSIX, il est possible d'obtenir des gains de 20 à 40% suivant les applications [TBG+08] sur des plates-formes NUMA. Cela démontre l'intérêt d'avoir des couches logicielles en adéquation pour espérer exploiter efficacement les machines parallèles modernes.

Fortress

Fortress [ACH+08] est un langage qui a l'ambition de remplacer Fortran. C'est un langage orienté composant avec un support natif du parallélisme. Ce support se fait à travers la parallélisation de boucles ou la parallélisation implicite des expressions. Le langage permet également de gérer les données partagées en mémoire.

La Figure 3.10 présente le calcul d'un terme de la suite de Fibonacci en utilisant Fortress. Le langage génère au besoin du parallélisme lors de l'évaluation de *fibonacci(n-1) + fibonacci(n-2)*.

Pour l'instant l'implantation de référence de Fortress est faite via la machine virtuelle Java. Cela limite les performances de Fortress, et il faudra attendre une version réellement compilée pour pouvoir l'utiliser dans le cadre du calcul hautes performances.

Java

Java [GJSB05] possède un support natif des threads, ainsi que des mécanismes de synchronisation intégrés au langage via l'utilisation de quelques

```
public class Fibonacci extends RecursiveAction {
        private final int number;
        private int result;

        public Fibonacci(int number) {
                this.number = number;
        }

        public int getResult() {
                return result;
        }

        public void compute() {
                int n = number;
                if (n < 2) {
                        result = n;
                        return;
                }
                final Fibonacci f1 = new Fibonacci(n - 1);
                final Fibonacci f2 = new Fibonacci(n - 2);
                coInvoke(f1, f2);
                result = f1.getResult() + f2.getResult();
        }
}
```

FIGURE 3.11 – Exemple d'utilisation de Java 7 : le calcul du n-ième terme de la suite de Fibonacci

```
cilk in fib(int n){
        if(n<2) return n;
        else {
                int x,y;
                x = spawn fib(n-1);
                y = spawn fib(n-2);
                sync;
                return(x+y);
        }
}
```

FIGURE 3.12 – Exemple d'utilisation de Cilk : le calcul du n-ième terme de la suite de Fibonacci

```
int fib_parallel(int n)
{
        int x, y;
        if (n < 2) return n;
        x = cilk_spawn fib_parallel(n-1);
        y = fib_parallel(n-2);
        cilk_sync;
        return (x+y);
}
```

FIGURE 3.13 – Exemple d'utilisation de Cilk++ : le calcul du n-ième terme de la suite de Fibonacci

mots clefs. Il est donc très facile de développer du code parallèle à base de threads en Java.

Dans la prochaine version, numérotée 7, il existera en plus un support pour le parallélisme de type Fork/Join. Ce support est illustré au travers de la Figure 3.11.

Cilk et Cilk++

Cilk [BJK+95] est une extension du C qui permet d'exprimer le parallélisme d'une application à travers quelques mots clefs. Ces mots clefs désignent des fonctions qui peuvent êtres exécutées en parallèle. La Figure 3.12 présente l'utilisation de Cilk pour décrire un calcul parallèle de la suite de Fibonacci. Le compilateur de Cilk se charge de traduire un programme Cilk en

```
struct fibonacci {
        void operator()( int n , a1::Shared_w<int> res ) {
                if( n < 2 ) {
                        res.write(n) ;
                } else {
                        a1::Shared<int> res1;
                        a1::Fork<fibonacci>()(n-1,res1);
                        a1::Shared<int> res2;
                        a1::Fork<fibonacci>()(n-2,res2);
                        a1::Fork<add>()( res1, res2, res );
                }
        }
};
```

FIGURE 3.14 – Exemple d'utilisation d'Athapascan : le calcul du n-ième terme de la suite de Fibonacci

un programme C qui est ensuite compilé normalement et lié avec un support exécutif chargé de gèrer l'allocation des tâches sur des threads de travail.

Le support exécutif de cilk utilise le vol de travail [BL99] pour distribuer les tâches sur les threads. Pour limiter le surcoût et garantir un meilleur passage à l'échelle, le coût du vol de travail est reporté sur les opérations de vol qui sont peu nombreuses.

Récemment, une version pour le C++ a été développée. Elle est baptisée Cilk++ [Art]. En plus d'une simplification du langage, Cilk++ ajoute la notion de boucle parallèle. La Figure 3.13 présente le même programme (cette fois-ci en Cilk++).

3.3.3 Programmation multi-cœurs et bibliothèques

Une troisième possibilité qui permet d'offrir du parallélisme est d'écrire une bibliothèque externe. On peut distinguer deux types de bibliothèques. Tout d'abord celles que nous appelleront intergicielles, qui offrent des outils pour construire des algorithmes parallèles, et celles qui offrent des algorithmes déjà écrits. Ces dernières se basent souvent sur les premières, ou alors directement sur les solutions présentées précédemment. Les bibliothèques présentées ici sont principalement orientées vers les architectures à mémoire partagée.

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) : n(n_), sum(sum_) {}
    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long x, y;
            FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
            FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
            set_ref_count(3);
            spawn( b );
            spawn_and_wait_for_all( a );
            *sum = x + y;
        } return NULL;
    }
};
```

FIGURE 3.15 – Exemple d'utilisation de TBB : le calcul du n-ième terme de la suite de Fibonacci

KAAPI et Athapascan

KAAPI [GBP07] est un support exécutif qui permet d'exécuter des programmes écrits en Athapascan [GCRD98]. Athapascan est un langage qui permet de décrire des tâches qui doivent être exécutées en parallèle et le statut de données qu'elles utilisent, privé ou partagé. Les dépendances entres tâches sont construites pendant l'exécution. Ceci évite une phase de compilation séparée comme c'est le cas pour Cilk. La Figure 3.14 illustre l'utilisation d'Athapascan au travers de l'exemple classique du calcul d'un terme de la suite de Fibonacci.

KAAPI permet l'exécution de ces programmes sur des machines distribuées mais également sur des machines multi-cœurs. Le moteur de KAAPI est adaptatif, il permet ainsi d'éviter de générer trop de parallélisme et donc limite son surcoût. KAAPI utilise des tâches ordonnancées sur des threads POSIX.

Intel TBB

Intel TBB [KV07] est une bibliothèque conçue et développée par Intel. Elle n'est cependant pas limitée aux processeurs de cette marque. Cette bibliothèque propose à la fois un ensemble de classes et d'interfaces pour décrire des applications parallèles et un exécutif basé sur du vol de travail pour exécuter ces applications. TBB offre un certain nombre de constructions parallèles comme une boucle *for* ou une construction *while*. Il est également possible de décrire des application de type *fork/join*. Cette utilisation est illustrée sur la Figure 3.15

MCSTL

MCSTL [SSP07] est une implantation parallèle de la STL pour les architectures multi-cœurs utilisant openMP pour la parallélisation.

La STL [SL95] est un ensemble de structures de données et d'algorithmes séquentiels écrits en C++. Par exemple, il existe des structures de tableaux et de tables de Hash, ainsi que des algorithmes de tri ou de recherche de minimum sur ces structures. La STL fait maintenant partie du langage C++.

Les algorithmes *embarassingly parallel* utilisent un moteur de vol de travail tandis que les autres utilisent des algorithmes parallèles *ad hoc*. MCSTL est à notre connaissance la seule STL qui vise directement les machines à mémoire partagée multi-cœurs.

3.3.4 STAPL et PSTL

STAPL [AJR+01] est une implantation de la STL pour plates-formes parallèles et distribuées. Elle est basée sur la notion de containers distribués, de vues sur ces containers et d'algorithmes parallèles. Une vue représente un niveau intermédiaire entre le container et l'élément. STAPL propose des itérateurs sur les vues, pour accéder aux éléments.

STAPL offre une version distribuée de nombreux types de containers et d'algorithmes de la STL. STAPL propose également des services de communication ainsi que des diagrammes de dépendance de tâches correspondant aux algorithmes classiques.

Une autre bibliothèque basée sur des containers distribués est PSTL [JG97].

3.3.5 De nombreuses solutions

Les solutions présentées permettent d'exploiter, à un niveau plus ou moins fin le parallélisme d'un machine. Cependant, Les machines multi-cœurs sont bien différentes des machines à mémoire partagée qui existaient jusqu'alors.

Aucune de ces approches ne cherche spécifiquement à obtenir des accélérations sur des tâches de petite taille exécutées sur ces architectures. Or cela nous semble être un des enjeux du pari multi-cœurs.

3.4 Comment étudier les plates-formes multi-cœurs

Pour savoir quelle finesse de parallélisation les nouvelles architectures permettent il est nécessaire de les étudier avec précision. En effet, comme montré dans la section 3.3.2, l'adéquation entre les différentes couches logicielles permet des gains de performances non négligeables. Il est donc nécessaire d'étudier toute la structure simultanément. Ceci va engendrer de nombreux paramètres différents qui sont présentés ci-après.

3.4.1 Quels critères étudier ?

Les différents processeurs multi-cœurs, grand public ou pour serveur d'entrée de gamme, qui existent sont assez proches sur le plan technique. Néanmoins, des détails d'architecture peuvent avoir une grande influence sur les performances. De plus, considérer uniquement le processeur ne suffit pas pour caractériser les performances d'une plate-forme utilisant ces processeurs. En effet, l'environnement dans lequel ce processeur est plongé (carte mère, mémoire ou éventuelle carte fille) joue un rôle capital.

Le nombre de paramètres à prendre en compte est donc très grand, et le nombre d'expériences que l'on voudrait pouvoir réaliser croît exponentiellement avec le nombre de paramètres. Voici une liste non exhaustive de ces paramètres :

— le constructeur du processeur,
— la fréquence du processeur,
— le nombre de processeurs disponibles,
— le nombre de cœurs disponibles,
— la technologie de cache,
— la taille du cache,
— la technologie de bus,
— la fréquence de la mémoire,
— etc...

Chacun de ces paramètres peut avoir ou non une influence et aucun ne peut être exclu a priori. De plus, ces paramètres concernent uniquement la plate-forme. Le logiciel qui servira de test va certainement induire de nombreux paramètres supplémentaires :

— les politiques de synchronisation,
— les politiques d'allocation de la mémoire,
— les politiques d'allocation des cœurs,
— les politiques d'ordonnancement,
— la taille des problèmes considérés,
— le nombre de cœurs utilisés,
— la répartition des cœurs utilisés,
— etc...

3.4.2 Un banc d'essai pour architectures multi-cœurs

Face à un si grand nombre de paramètres, les programmes de test applicatifs ne permettent pas d'avoir une vue précise de l'influence de tel ou tel paramètre. Pour ce faire, il est nécessaire de construire un environnement logiciel qui va permettre une étude approfondie de la plate-forme.

Cet environnement logiciel doit respecter un certain nombre de conditions :

— il doit être aisément modifiable pour pouvoir s'adapter à la variété d'architectures mises à notre disposition,
— il doit offrir des résultats reproductibles,
— il doit s'abstraire du système d'exploitation sous-jacent,
— il doit être suffisamment court pour pouvoir être reproduit un grand nombre de fois,
— sans être un test applicatif, il doit néanmoins être représentatif d'une certaine réalité, tant au niveau du travail effectué que des performances présentées.

3.4.3 Conclusion

L'exploitation des architectures modernes, et notamment les architectures multi-cœurs présente un réel défi. De nombreuses solutions existent déjà pour exploiter ces plates-formes. Elles utilisent majoritairement le principe du vol de travail. Mais l'étude précise de l'adéquation entre ces logiciels et les plates-formes cibles paraît difficile tant ces solutions sont complexes.

Il nous semble donc essentiel de proposer un outil pour étudier ces nouvelles plate-formes. Il n'est pas non plus possible de se limiter aux seules machines multi-cœurs. En effet, les machines composées de plusieurs processeurs multi-cœurs sont légion. De même, les machines NUMA doivent également pouvoir être étudiées. En fait, en prenant en compte les effets de cache, une machine composée de plusieurs processeurs multi-cœurs peut être considérée comme une machine NUMA.

Dans le chapitre suivant, nous proposons donc PaSTeL un outil d'étude des plates-formes multi-cœurs qui se veut à la fois performant et modulaire.

Chapitre 4

PaSTeL : un banc d'essai pour architectures multi-cœurs

La construction d'applications parallèles sur des architectures complexes pose un réel défi. Le défi ne consiste pas juste à produire le code de ces applications mais également à évaluer leurs performances et comprendre ces résultats. Ce chapitre propose donc un outil, PaSTeL, pour étudier les différentes stratégies d'implantation d'algorithmes parallèles et l'impact de l'architecture matérielle et logicielle sur ces stratégies. La structure de cet outil est succinctement représentée sur la Figure 4.1.

L'objectif de PaSTeL est d'étudier l'adéquation du support d'exécution avec les différents composants de l'architecture pouvant avoir un impact significatif (et pénalisant) sur les performances. Il peut être utile d'étudier l'influence de l'architecture du processeur, du nombre de cœurs et de processeurs, des effets de cache, de la bande passante mémoire ou des différentes politiques d'allocation de ressources. PaSTeL doit donc facilement s'adapter à ces différentes architectures et adapter sa configuration logicielle à son environnement.

Pour satisfaire ces exigences, nous avons défini un environnement modulaire. Il doit permettre notamment d'utiliser différentes stratégies d'exécution. Par exemple, il doit être possible d'implanter à la fois du vol de travail et de la distribution de travail. Il doit également être possible de comparer les politiques de synchronisation pouvant être utilisées.

PaSTeL propose une interface de programmation pour implanter des algorithmes parallèles au-dessus des stratégies de d'exécution choisies. Cette interface doit être suffisamment expressive pour permettre d'implanter un large spectre d'algorithme.

Au cours du développement de PaSTeL, un certain nombre de modules que nous avons jugés pertinents pour répondre à la problématique ont été

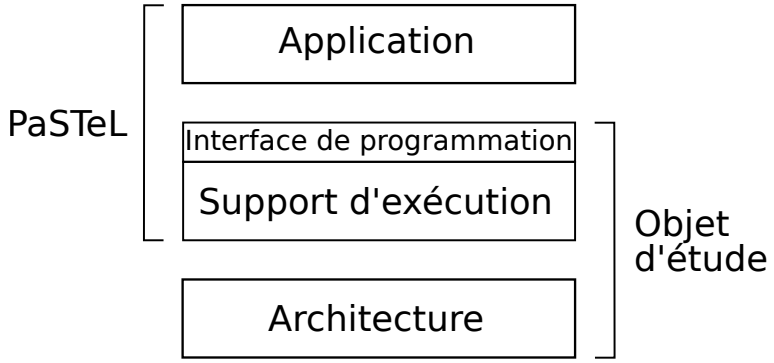


FIGURE 4.1 – PaSTeL : un outil pour étudier le support d'exécution et l'architecture

construits. Pour valider ces modules et leur assemblage, nous avons comparé les performances d'algorithmes implantés sur PaSTeL avec celles des mêmes algorithmes implantés dans d'autres bibliothèques parallèles existantes. La reproductibilité des résultats obtenus a aussi été un critère de validation de la construction. Il est également nécessaire de vérifier que PaSTeL répond à des questions telles que :

— Quelle est la taille des problèmes que l'on peut paralléliser sur une architecture ?
— À l'exécution, quel nombre de cœurs allouer à une tâche parallèle ?

Dans une première section, nous allons présenter le support d'exécution de PaSTeL dans sa dernière version. Dans un second temps, nous présenterons l'utilisation du support d'exécution à travers l'implantation d'un sous-ensemble de la STL. Ensuite, nous nous attacherons à modéliser l'exécution de PaSTeL pour pouvoir prédire ses performances. Finalement, nous validerons le comportement de PaSTeL par une étude comparée de ses performances face à d'autres implantations de la STL.

4.1 Support d'exécution

Le support d'exécution est le cœur de PaSTeL. Il est composé de 4 grands modules représentés sur la Figure 4.2. Le premier module est le support système employé. Il est possible d'envisager du parallélisme de processus en mémoire distribuée, ou du parallélisme de thread, en mémoire partagée. Le second module concerne la manière de synchroniser les processus ou les threads. Il faut noter que suivant le système d'architecture employé toutes les formes de synchronisation ne sont pas possibles. Un troisième module concerne la manière dont PaSTeL gère les ressources qui lui sont attribués.

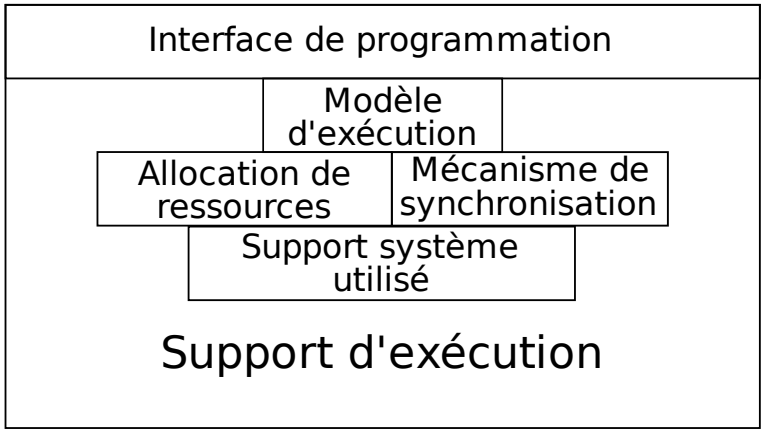


FIGURE 4.2 – Support d'exécution modulaire de PaSTeL

En effet il peut être nécessaire de placer certains threads ou processus sur différents cœurs ou processeurs pour étudier l'influence de leur placement. Enfin, un dernier module concerne le modèle d'exécution choisi. Ce peut être du vol de travail, mais il est possible également d'envisager une répartition à priori du travail entre les threads. L'ensemble de ces modules présentent une interface de programmation à l'utilisateur pour lui permettre d'exploiter les services proposés.

Au cours du développement de PaSTeL, nous avons conçu un certain nombre de modules qui nous ont paru pertinents par rapport à la problématique de l'exploitation et de l'évaluation des plates-formes multi-cœurs. Nous allons donc maintenant présenter ces choix ainsi que leurs motivations, en commençant par le langage de programmation choisi pour développer ces modules.

4.1.1 Langage de programmation

Le langage utilisé pour développer PaSTeL est le C++. Ce langage permet à la fois de gérer le parallélisme d'une application à un niveau proche du système et de structurer cette application via l'utilisation d'objets et de classes. L'autre avantage du C++ est la grande variété de bibliothèques de parallélisme et de synchronisation disponibles. De fait, de nombreuses applications parallèles utilisent ce langage ou sont fondées sur des bibliothèques écrites en C/C++.

4.1.2 Support système

Le support système choisi va imposer des options que nous présenterons ultérieurement. Cependant, dans le cadre d'une étude des architectures multi-cœurs, il nous a semblé opportun de nous limiter d'abord au modèle de mémoire partagée [1]. De plus, pour faciliter la collaboration entre les différents fils d'exécution, nous avons choisi d'utiliser des processus légers. Ces processus présentent également l'avantage d'utiliser moins d'espace mémoire, de pouvoir gérer leur ordonnancement en espace utilisateur et d'alléger les changements de contexte.

Pour ce type de processus, POSIX est la norme la plus répandue et ce sont donc des threads POSIX qui ont été utilisés. Ils présentent l'avantage d'être associés à des threads noyaux sur la plupart des système récents, ce qui permet de l'exploitation efficace de machines multiprocesseurs.

D'autres type de threads pourraient être employés, notamment des bibliothèques comme Marcel.

4.1.3 Type de synchronisation

Le choix du type de synchronisation employé n'est pas anodin. Il existe en effet plusieurs niveaux possibles pour gérer la synchronisation entre les threads. La synchronisation peut s'opérer à l'intérieur du noyau ou dans l'espace de mémoire partagée par les threads ou les deux à la fois.

Dans PaSTeL deux modules ont été développés. Le premier module emploie des verrous ou *mutex* et des conditions POSIX, le second emploie des verrous tournants ou *spinlock*.

Bien que les données d'un verrou POSIX soient stockées en mémoire partagée, les verrous restent dépendants du système d'exploitation. En effet, si un thread se bloque sur un verrou il peut perdre la main au profit du système d'exploitation qui va alors ordonnancer un autre thread disponible. Lorsque le mutex redevient disponible, le thread peut ne pas obtenir immédiatement le processeur. Cela peut induire des latences importantes qui pénalisent les performances.

Dans le cas de l'utilisation de verrous tournants la synchronisation est alors effectuée uniquement en espace utilisateur. On s'affranchit alors du problème précité au prix d'une utilisation inutile de cycle machine. Néanmoins, dans le cadre d'une étude de performance, l'utilisation de spinlock permet d'obtenir une bonne reproductibilité des expériences et de très bonnes performances.

1. Le modèle de programmation à mémoire distribuée ne sera pas abordé dans ce travail.

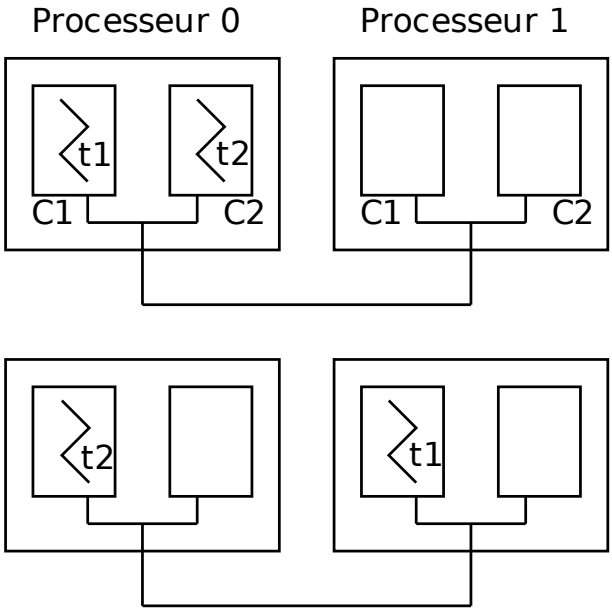


FIGURE 4.3 – Exemple de placements possibles sur une architecture possédant 2 processeurs bi-cœur

4.1.4 Allocation des ressources

Le problème de l'allocation des ressources est au cœur de la problématique des nouvelles architectures. En effet, dans le cas, par exemple, d'une architecture qui posséderait 2 processeurs ayant chacun 2 cœurs, comme représenté sur la Figure 4.3, il existe de nombreuses manières de répartir les threads d'une application parallèle. Si l'application possède 2 threads, alors, il est possible de mettre les deux sur le même processeur (1er cas), d'en mettre un sur chaque processeur (2ème cas) ou même de mettre les deux threads sur le même cœur (non représenté). Au total il existe 16 manières de répartir les threads sur la machine.

L'impact de tel ou tel placement sur les performances va dépendre de l'application et de la manière dont ses threads interagissent. Les différences peuvent être très importantes ou négligeables et les déterminer a priori est un problème difficile.

PaSTeL permet donc d'associer chaque thread à une ou plusieurs unités de calcul. Cela permet d'étudier finement l'influence du placement sur une application. Cette association peut se faire de deux manière différentes. Soit via un fichier de configuration qui décrit l'architecture de la machine, le nombre de threads à créer et l'allocation des threads aux différentes unités de calcul, soit automatiquement en associant un thread à chaque unité de

```
#include <pastel_algorithm>
int main (void)
{
  float tableau[TAILLE];
  /* ..... */
  //std::sort(tableau, tableau + TAILLE);
  pastel::sort(tableau, tableau + TAILLE);
  /* ..... */
}
```

FIGURE 4.4 – Exemple d'utilisation de PaSTeL

calcul dans l'ordre, les unités étant utilisées plusieurs fois si un trop grand nombre de threads sont présents.

Il n'existe pas pour l'instant de module pour étudier directement l'influence du placement des données en mémoire dans PaSTeL. Cependant, les threads de PaSTeL allouent et initialisent eux même les structures de données qui les concernent. Dans le cas d'une politique d'allocation mémoire classique de type *first touch*, ce comportement doit être proche de l'optimal. Pour étudier le placement des autres données, d'autres bibliothèques qui peuvent s'intégrer dans PaSTeL existent.

4.1.5 Modèle exécutif

Le modèle exécutif actuellement implanté dans PaSTeL est basé sur du vol de travail. Le choix de ce modèle s'est imposé de par sa popularité au sein de la communauté du parallélisme en mémoire partagée. Cela met à notre disposition de nombreuses références de comparaisons, tant au niveau des performances que de la simplicité de programmation.

Il existe de nombreuses implantations de vol de travail en mémoire partagée. Plusieurs éléments permettent de caractériser l'implantation du vol de travail dans PaSTeL.

Tout d'abord il existe deux types de threads dans PaSTeL. Une application sera constituée d'un thread principal et d'un ensemble de threads secondaires (cf Figure 4.5. Le thread principal peut par exemple exécuter un programme semblable à celui présenté sur la Figure 4.4. Dans ce programme, l'appel au tri standard du C++ a été remplacé par un appel à la version du tri proposée par PaSTeL. Le deuxième type de threads qui existent dans PaSTeL sont appelés threads secondaires. Leur rôle consiste à participer à

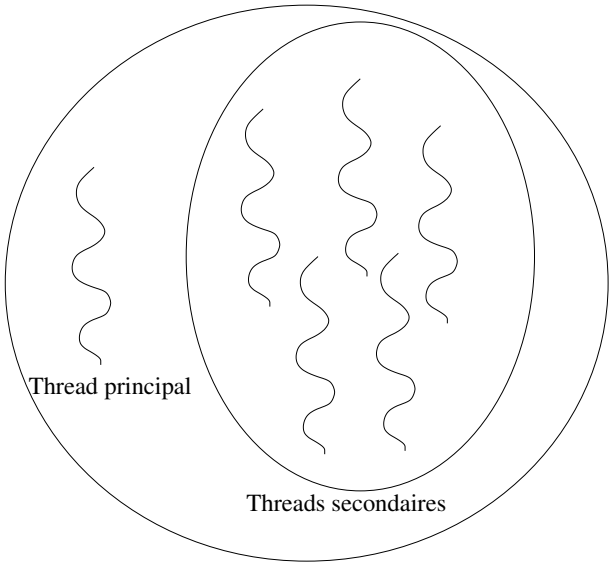


FIGURE 4.5 – PaSTeL est composé d'un thread principal qui exécute le programme principal et de threads secondaires. Les thread secondaires aident le thread principal dans la réalisation de tâches parallèles.

l'exécution des algorithmes parallèles, par exemple ici lors du tri parallélisé du tableau.

Le second point concerne la manière dont les threads collaborent pour réaliser une tâche. Lorsqu'un thread principal rencontre un algorithme parallèle de PaSTeL, il stocke cet algorithme dans une file d'attente et commence ensuite à travailler. Les threads secondaires inactifs scrutent la file pour trouver du travail. Lorsqu'un thread secondaire trouve un algorithme non terminé, il retire l'algorithme de la file, se joint au groupe qui travaille déjà sur cet algorithme, puis remet l'algorithme à la fin de la file pour d'éventuels autres threads disponibles. Le thread va ensuite acquérir du travail en volant un des threads travaillant déjà sur cet algorithme. Ce mécanisme possède plusieurs avantages. Tout d'abord il permet une allocation en tourniquet, ou *round robin*, des threads sur les algorithmes si plusieurs sont disponibles simultanément. Ensuite ce mécanisme facilite la limitation du nombre de threads que l'on veut allouer à un algorithme. Finalement il est assez facile à modéliser si l'on veut réaliser de la prédiction de performances. Ce mécanisme possède néanmoins certains inconvénients comme l'impossibilité de répartir à priori le travail entre les threads, comportement qui n'est en théorie pas souhaitable dans le cadre du vol de travail.

Le troisième point concerne le vol lui-même. En effet PaSTeL utilise un vol synchrone. Un thread qui travaille vérifie périodiquement s'il va être volé.

Si c'est le cas, alors il interrompt son exécution le temps que le ou les voleurs acquièrent du travail. Le surcoût de scrutation étant faible, cela permet de réagir rapidement en cas de vol tout en évitant le surcoût de création de tâche pour réaliser un vol asynchrone. De cette manière, la taille des tâches que l'on peut voler diminue, et on gagne donc en réactivité.

4.1.6 Conclusion

PaSTeL met l'accent sur la réactivité de son support exécutif. Cet approche doit permettre de paralléliser des travaux de petite taille. La section suivante présente l'utilisation qui peut être faite de PaSTeL.

4.2 Exemple d'utilisation de PaSTeL

Cette section présente l'utilisation du support exécutif de PaSTeL pour implanter certains algorithmes de la STL. La STL est la bibliothèque standard du C++. Elle propose des types de données et des algorithmes s'appliquant à ces types. Tout d'abord, nous allons justifier ce choix. Ensuite nous étudierons plus précisément l'interface de programmation proposée par le support exécutif actuel de PaSTeL. Finalement nous décrirons l'implantation de l'algorithme d'interclassement de deux tableaux.

4.2.1 Implantation d'un sous-ensemble de la STL

Il existe de nombreux algorithmes parallèles qu'il serait intéressant d'étudier avec un outil comme PaSTeL. Mais dans le cadre de notre démarche d'évaluation de performances, ces algorithmes doivent présenter un certain nombre de caractéristiques. Notamment, il doit exister des implantations de référence de ces algorithmes. De plus, il faut des algorithmes dont le coût calculatoire est faible pour que les tâche restent de petite taille.

Pour enrichir l'étude, il est intéressant de disposer de plusieurs algorithmes parallèles. En effet, les schémas de vol et d'accès aux données peuvent avoir une influence sur les performances. Cependant, certains algorithmes se prêtent plus ou moins bien à la parallélisation en fonction du modèle d'exécution choisi. La STL propose une gamme d'algorithmes qui reposent sur des itérateurs. Ces algorithmes s'adaptent parfaitement à notre modèle exécutif. De plus, les algorithmes de la STL peuvent (à quelques exceptions près) se classer en six catégories. Ces catégories sont nommées ici à partir de leur représentant le plus caractéristique :

— *find* : recherche le premier élément d'un tableau qui corresponde à un prédicat,
— *min_ element* : recherche la valeur minimum d'un tableau,
— *binary_ search* : recherche si un élément existe dans un tableau,
— *partition* : partitionne un tableau en deux sous-ensembles autour d'un pivot,
— *merge* : interclasse deux tableaux triés en un troisième trié également,
— *sort* : trie un tableau.

Il faut remarquer que les algorithmes de type *for_ each* (application d'une méthode à chaque élément d'un tableau) rentrent dans la catégorie de l'algorithme *min_ element*. Ces différentes classes d'algorithmes exhibent des comportements très différents et nous ont semblé suffisamment variées pour nos besoins expérimentaux.

Ces algorithmes, bien que non triviaux à implanter en parallèle pour certains, ont également le mérite d'être suffisamment simples pour présenter des comportements réguliers lors de l'exécution. Cette régularité s'observe dans le temps de traitement des données et dans le nombre de vols réalisés au cours de l'exécution. Cette régularité simplifie l'interprétation des résultats expérimentaux et la reproductibilité des mesures. Il devrait être aussi possible de prédire les performances de ces algorithmes moyennant un certain nombre de mesures préalables.

Un autre intérêt de la STL est l'existence d'implantations parallèles de la STL ou de bibliothèques parallèles permettant un implantation de ces algorithmes. Nous avons donc à notre disposition des références auxquelles nous comparer, en termes de performances mais également en termes de facilité de programmation et de mise en œuvre.

4.2.2 Vol de travail de PaSTeL

Pour implanter un algorithme de la STL sur le moteur de vol de travail de PaSTeL il est nécessaire de détailler un peu le fonctionnement de ce vol. Cette section présente donc le moteur de vol de travail.

Les algorithmes de PaSTeL dépendent de deux type de données partagées en mémoire. Les données globales (notées g) contiennent l'état global d'exécution de l'algorithme, alors que les données locales (notées l) contiennent l'état d'exécution d'un thread. Les données globales sont initialisées par le thread principal qui appelle l'algorithme (cf Section 4.1.5), alors que les données locales sont initialisées par chaque thread secondaire quand il se joint à l'algorithme. Les données contenues dans ces structures sont protégées par des verrous.

Une fois l'initialisation des données terminée, tous les threads exécutent

```
struct DonneesGlobales {...}
/*Contient les données de l'algorithme*/
struct DonneesLocales {...}
/*Contient les données d'un thread*/

Fonction workstealing(DonneesGlobales g, DonneesLocales l)
tant que TravailGlobal (g) faire
  si non TravailLocal (l) alors
    t = Choisir Victime ()
    Vol (l,t)
  sinon
    tant que TravailLocal (l) faire
      CalculerMorceau (l)
      si Cible de vol alors
        MiseAJourEtatGlobal (g,l)
        Attendre le départ des voleurs
      fin si
    fin tant que
    MiseAJourEtatGlobal (g,l)
  fin si
fin tant que
```

Figure 4.6 – Moteur de vol de travail de PaSTeL

le même programme. Cette fonction principale (*workstealing*) est présentée sur la figure (Figure 4.6).

La procédure est architecturée autour d'une boucle qui se termine quand l'algorithme est terminé. Cette terminaison est détectée par la fonction TRAVAILGLOBAL qui indique s'il reste globalement du travail à réaliser.

S'il reste globalement du travail, alors deux possibilités s'ouvrent :

— soit le thread qui exécute *workstealing* possède une partie du travail restant,

— soit il ne dispose plus de travail.

S'il ne possède pas de travail, mais qu'il en reste globalement, alors il va en voler à un thread qui en possède. Le moteur choisit une victime aléatoirement et le thread vole du travail grâce à la fonction VOL (Figure 4.6, Ligne 5).

Une fois qu'il a du travail localement, et tant qu'il lui en reste, le thread l'exécute par morceau grâce à la fonction CALCULERMORCEAU. La taille des morceaux est appelé grain. Entre chaque morceau, le thread vérifie si l'on tente de le voler. Si c'est le cas il met à jour l'état global pour valider le travail déjà effectué (MISEAJOURETATGLOBAL) puis attend que les voleurs aient terminé leur opération (**Attendre le départ des voleurs**) (Figure 4.6, Lignes 8 à 11).

Une fois son travail local épuisé, le thread met à jour l'état global (MISEAJOURETATGLOBAL) puis la boucle recommence.

On peut remarquer que l'appel à MISEAJOURETATGLOBAL ne semble pas nécessaire avant un vol. Il permet en fait d'accélérer la détection de la terminaison de l'algorithme quand de nombreux threads cherchent simultanément du travail.

4.2.3 Implantation de l'interclassement de deux tableaux

Comme nous l'avons vu dans la section précédente, le vol de travail de PaSTeL fait appel à un certain nombre de fonctions lors de son exécution. Ce sont ces fonctions que l'utilisateur doit implanter s'il veut écrire un algorithme dans PaSTeL. La liste des fonctions et des structures de données dont l'utilisateur a la responsabilité sont représentées sur la Figure 4.7.

Nous allons illustrer l'implantation de ces fonction en utilisant l'algorithme d'interclassement de deux tableaux. Cet algorithme prend deux tableaux triés T_1 et T_2 en entrée, de taille L_1 et L_2 et renvoie un troisième tableau T_3 de taille L_3 contenant les éléments de T_1 et T_2 triés. Le code source de l'algorithme *merge* de la STL est présenté sur la Figure 4.8 (à peine simplifié). Les bornes du premier tableau sont __first1 et __last1, celle du deuxième tableau __first2 et __last2. Le résultat est stocké dans le tableau commençant à __result. Les tableaux sont ensuite parcourus dans

```
struct DonneesGlobales {...}
/*Contient les données de l'algorithme*/
struct DonneesLocales {...}
/*Contient les données d'un cœur*/
bool TravailGlobal (DonneesGlobales g){...}
/*Retourne faux si l'algorithme est terminé*/
bool TravailLocal (DonneesLocales l){...}
/*Retourne faux si le travail du cœur est terminé*/
void CalculerMorceau (DonneesLocales l){...}
/*Traite un bloc de données*/
void MiseAJourEtatGlobal
  (DonneesLocales l, DonneesGlobales g){...}
/*Met à jour l'état global avec l'avancement d'un cœur*/
void Vol (DonneesLocales l, DonneesLocales v){...}
/*transfert tu travail d'un cœur à un autre*/
```

Figure 4.7 – Interface de programmation de PaSTeL

l'ordre en recopiant le plus petit élément dans le tableau résultat. La dernière ligne permet, une fois l'un des tableaux vide, de recopier la fin du dernier tableau. La dernière case du tableau résultat est renvoyée comme valeur de retour de l'algorithme.

Pour écrire une version parallèle de cet algorithme dans PaSTeL il est nécessaire de définir deux structures de données. La première stocke les informations globales de l'algorithme. Elle est présentée sur la Figure 4.9. On retrouve les mêmes paramètres que dans l'algorithme de la STL, avec en plus un compteur $__counter$. Ce compteur est initialisé à $__counter = __last1 - __first1 + __last2 - __first2$. Il indique donc le nombre d'éléments à traiter. Quand ce compteur atteint 0, l'algorithme est terminé.

La structure qui stocke les donnée locales à un thread est sensiblement identique à la structure globale. Elle est présentée sur la Figure 4.10. Au lieu de pointer sur l'intégralité des tableaux, elle pointe sur une sous-partie de chaque tableau. Le compteur dans la structure locale sert à compter le nombre d'éléments que le thread a déjà traité.

La première fonction à implanter ensuite est CalculerMorceau. Cette procédure est présentée sur la Figure 4.11. C'est dans cette procédure que l'interclassement est réalisé par les threads. Elle prend donc en paramètre les données locales d'un thread. La première partie de la procédure consiste à extraire deux sous-tableaux de taille $GRAIN$, au plus. On s'arrête quand l'un des sous-tableaux est vide car on ne peut pas anticiper les valeurs suivantes. A

```
template<typename Iterator>
  Iterator
  merge(Iterator first1, Iterator last1,
        Iterator first2, Iterator last2,
        Iterator result)
  {
    /***/
    while (first1 != last1 && first2 != last2)
      {
        if (*first2 < *first1)
          {
            *result = *first2;
            ++first2;
          }
        else
          {
            *result = *first1;
            ++first1;
          }
        ++result;
      }
    return std::copy(first2, last2,
            std::copy(first1, last1, result));
  }
```

FIGURE 4.8 – Algorithme *merge* de la STL

```
template <typename RandomIterator>
struct MergeData : public WorkstealingData {
  RandomIterator first1;
  RandomIterator last1;

  RandomIterator first2;
  RandomIterator last2;

  RandomIterator result;

  size_t counter;
}
```

FIGURE 4.9 – Données globales de l'algorithme *merge*

```
template <typename RandomIterator>
struct MergeThreadParam : public WorkstealingThreadParam {
  RandomIterator sub_first1;
  RandomIterator sub_last1;
  RandomIterator sub_first2;
  RandomIterator sub_last2;

  RandomIterator sub_result;

  size_t counter;
};
```

FIGURE 4.10 – Données locales de l'algorithme *merge*

chaque appel de procédure, le thread traite donc au plus $2*GRAIN$ éléments. Quand les tableaux ne sont pas presque vides on en traite au moins $GRAIN$. A la fin de la procédure, le compteur est incrémenté du nombre d'éléments traités et les bornes des tableaux d'origine modifiées en fonction. Le grain de l'algorithme est ici défini statiquement. Le code utilisé est très semblable à celui qui figure dans la STL, pour éviter de voir des différences de performance imputables à la réécriture du code.

La fonction TravailLocal est très simple et permet de savoir s'il reste localement du travail à un thread. Cette fonction vérifie donc qu'au moins l'un des tableaux que possède ce thread n'est pas vide (Figure 4.12).

La procédure qui met à jour l'état global de l'algorithme, MiseAJourEtatGlobal, est également très simple (Figure 4.13. Elle décrémente le compteur global de l'algorithme avec celui du thread passé en paramètre. Le compteur du thread est ensuite remis à 0.

La vérification de non terminaison de l'algorithme est donc également très simple. Il suffit de vérifier que le compteur global n'est pas tombé à 0. Cette fonction, TravailGlobal, est présentée sur la Figure 4.14.

La dernière procédure à implanter est le vol de travail entre deux threads. Cette procédure, présentée sur la Figure 4.15, est un peu plus complexes que les précédentes. La fonction vérifie tout d'abord que le thread choisi pour le vol possède suffisamment de travail pour qu'il soit intéressant de le partager. Si ce n'est pas le cas, la totalité du travail est volée. Ce comportement est intéressant car les threads voleurs recommencent plus vite à travailler que les threads volés. Si une partition est possible, la fonction choisit un pivot dans le tableau le plus grand, et recherche la première occurrence de ce pivot dans le second tableau. Le premier élément supérieur au pivot est choisi dans le premier tableau. Cette manière de faire garantit la stabilité de l'interclassement, c'est à dire qu'à valeur égale, les éléments du premier tableau sont toujours avant les éléments du deuxième dans le résultat. Si les éléments sont tous égaux, alors le travail de recopie est également partagé équitablement. Ceci fait, les bornes de tableaux sont mises à jour en fonction. Le nombre d'éléments que chaque thread devra traiter étant connu, la position à laquelle le thread devra stocker le résultat est également connue.

L'utilisation de ce modèle a permis d'implanter 4 classes d'algorithmes dans PaSTeL : *find*, *min_ element*, *merge* et *sort*. Ces 4 classes couvrent 80% des algorithmes de la STL. L'implémentation fonctionne sur toute structure de données avec un accès aléatoire (dans la terminologie STL, la structure doit fournir *RandomAccessIterator*). La sémantique est donc équivalente à celle des tableaux.

```
template <typename RandomIterator>
inline static void
calculerMorceau( MergeThreadParam* param) {
  RandomIterator tmp_last1 =
    (param->sub_first1 + GRAIN < param->sub_last1 ?
     param->sub_first1 + GRAIN : param->sub_last1 );
  RandomIterator tmp_last2 =
    (param->sub_first2 + GRAIN < param->sub_last2 ?
     param->sub_first2 + GRAIN : param->sub_last2 );
  RandomIterator tmp_first1 = param->sub_first1;
  RandomIterator tmp_first2 = param->sub_first2;
  RandomIterator tmp_result = param->sub_result;

  if( tmp_first1 >= tmp_last1 ) {
      tmp_result =
         std::copy(tmp_first2, tmp_last2, tmp_result);
      tmp_first2 = tmp_last2;
    }
  else if ( tmp_first2 >= tmp_last2 ) {
      tmp_result =
         std::copy(tmp_first1, tmp_last1, tmp_result);
      tmp_first1 = tmp_last1;
    }
  else while ( tmp_first1 != tmp_last1
            && tmp_first2 != tmp_last2 ) {
      if( *tmp_first2 < *tmp_first1 ) {
          *tmp_result = *tmp_first2;
          ++tmp_first2;
        }
      else {
          *tmp_result = *tmp_first1;
          ++tmp_first1;
        }
      ++tmp_result;
    }
  param->counter += ( tmp_first1 - param->sub_first1 )
                  + ( tmp_first2 - param->sub_firstb );
  param->sub_first1 = tmp_first1;
  param->sub_first2 = tmp_first2;
  param->sub_result = tmp_result;
}
```

FIGURE 4.11 – Procédure CALCULERMORCEAU de *merge*

```
inline static bool travailLocal( MergeThreadParam* param)
  {
    return ( param->sub_first1 < param->sub_last1 )
        || ( param->sub_first2 < param->sub_last2 ) ;
  }
```

FIGURE 4.12 – Fonction TRAVAILLOCAL de *merge*

```
inline static void
        commitLocalDansGlobal(MergeData* param_g,
                              MergeThreadParam* param)
  {
    param_g->counter -= param->counter;
    param->counter = 0;
  }
```

FIGURE 4.13 – Fonction MISEAJOURETATGLOBAL de *merge*

```
inline static bool travailGlobal(MergeData* param_g)
  {
    return param_g->counter > 0;
  }
```

FIGURE 4.14 – Fonction TRAVAILGLOBAL de *merge*

```
inline static void vol(MergeThreadParam* voleur,
                       MergeThreadParam* vole) {
  if ( ((vole->sub_last1 - vole->sub_first1) +
        (vole->sub_last2 - vole->sub_first2)) > GRAIN){
    voleur->sub_last1 = vole->sub_last1;
    voleur->sub_last2 = vole->sub_last2;
    if( (vole->sub_last2 - vole->sub_first2) >=
        (vole->sub_last1 - vole->sub_first1) ) {
      voleur->sub_first2 = vole->sub_first2
            + (vole->sub_last2 - vole->sub_first2 ) /2;
      voleur->sub_first2 = std::lower_bound( vole->sub_first2,
            vole->sub_last2, *(voleur->sub_first2) );
      voleur->sub_first1 = std::lower_bound( vole->sub_first1,
            vole->sub_last1, *(voleur->sub_first2));
    } else {
      voleur->sub_first1 = vole->sub_first1
            + (vole->sub_last1 - vole->sub_first1 ) / 2;
      voleur->sub_first2 = std::lower_bound( vole->sub_first2,
            vole->sub_last2, *(voleur->sub_first1) );
      voleur->sub_first1 = std::lower_bound( vole->sub_first1,
            vole->sub_last1, *(voleur->sub_first1));
    }
    vole->sub_last1 = vloeur->sub_first1;
    vole->sub_last2 = voleur->sub_first2;
    voleur->sub_result = vole->sub_result
         + (vole->sub_last1 - vole->sub_first1 )
         + (vole->sub_last2 - vole->sub_first2 );
    } else { //on vole tout, les voleurs recommencent
      //a travailler avant les voles
      voleur->sub_last1 = vole->sub_last1;
      voleur->sub_first1 = vole->sub_first1;
      voleur->sub_last2 = vole->sub_last2;
      voleur->sub_first2 = vole->sub_first2;
      vole->sub_last1 = voleur->sub_first1;
      vole->sub_last2 = voleur->sub_first2;
      voleur->sub_result = vole->sub_result;
    }
}
```

FIGURE 4.15 – Procédure VOL de *merge*

4.3 Modélisation de PaSTeL

Tout d'abord, il faut rappeler que le but de PaSTeL n'est pas de proposer un autre moteur d'exécution parallèle hautement performant et efficace. Le but de PaSTeL est de proposer un outil permettant de comprendre le comportement des systèmes modernes pour pouvoir adapter le support exécutif à la plate-forme, et ce, idéalement, pendant l'exécution. Cette section propose une première modélisation de PaSTeL qui permet d'en régler certains paramètres.

L'implantation actuelle de PaSTeL propose deux paramètres principaux. Le premier est le nombre de threads secondaires de calcul que PaSTeL alloue à un algorithme. Le second est la taille des blocs à traiter. Pour estimer des valeurs adéquates de ces paramètres, il faut disposer d'un modèle permettant de prédire les performances de PaSTeL.

4.3.1 Modélisation des threads

La première chose à modéliser est le comportement des threads, par exemple leur date de lancement et de terminaison. L'exécution d'un algorithme dans PaSTeL peut être modélisée simplement. La Figure 4.16 représente un modèle de l'exécution d'un algorithme de PaSTeL qui utilise n threads. Tout d'abord, le thread principal T_1 initialise PaSTeL. Cette étape consomme I unités de temps. Ensuite, T_1 réveille les autres threads. Le réveil des threads se fait en cascade avec une latence fixe W entre chaque réveil. Le thread i met donc $W_i = (i-1)W$ unités de temps pour se réveiller. Une fois réveillé, le thread T_i travail pendant p_i unités de temps. Finalement, le thread principal va effectuer une opération de synchronisation qui dure S unités de temps.

Le temps global de complétion de l'algorithme parallèle est noté T_{par} et le temps de complétion de la même tâche sur un seul cœur (c'est à dire le temps séquentiel) est noté T_{seq}. p_i est le temps que le i-ème thread passe effectivement à travailler sur l'algorithme. On suppose qu'au cours de la phase de travail le parallélisme est parfait et donc que : $T_{seq} = \sum_{i=1}^{n} p_i$. Cette hypothèse néglige le coût des vols et le surcoût de la mécanique de vol. On suppose également que I, W et S sont constants et ne dépendent pas du nombre de threads utilisés. Ces hypothèses seront validées par une étude de performances présentée dans la Section 4.4.

La phase finale de synchronisation ne peut démarrer avant que tout les threads soient réveillés : le temps de complétion est donc $T_{par} = I + \max_i(W_i + p_i) + S$. Comme le parallélisme est supposé parfait, le n-ième thread est utile

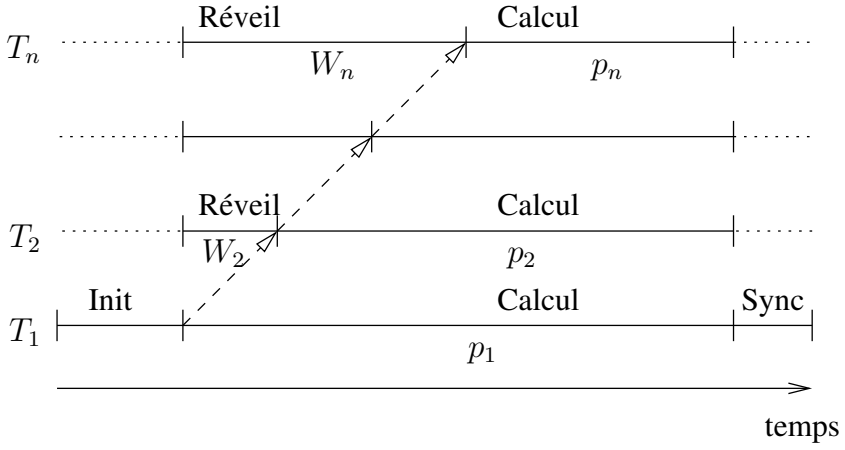


FIGURE 4.16 – Modèle de thread de PaSTeL.

pour peu qu'il ait suffisamment de temps pour travailler, c'est à dire, pour peu que $p_n > 0$. Donc, on minimise T_{par}, en maximisant n et en gardant la contrainte $p_n > 0$.

Tout d'abord, il faut déterminer p_1 en fonction du nombre de threads. p_1 s'exprime comme la somme du temps que chaque thread passe à calculer et à se réveiller :

$$\begin{aligned} n * p_1 &= \sum_{i=1}^{n} p_i + \sum_{i=2}^{n} W_i \\ &= T_{seq} + \frac{n(n-1)}{2} W \\ p_1 &= \frac{T_{seq}}{n} + \frac{n-1}{2} W \end{aligned}$$

Le temps que chaque thread passe à travailler s'exprime par :

$$\begin{aligned} p_i &= p_1 - (i-1)W \\ p_i &= \frac{T_{seq}}{n} + \frac{n-2i+1}{2} W \end{aligned}$$

Comme p_n est positif alors :

$$\begin{aligned} p_n = \frac{T_{seq}}{n} - \frac{n-1}{2}W &> 0 \\ n^2 - n - \frac{2T_{seq}}{W} &< 0 \end{aligned}$$

La résolution de cette équation dans l'espace réel permet de calculer la borne suivante :

$$n < \frac{1}{2} + \frac{1}{2}\sqrt{1 + 8\frac{T_{seq}}{W}}$$

Il est maintenant possible de calculer le nombre optimal (qui minimise T_{par}) de threads que PaSTeL doit utiliser et le temps total d'exécution d'un algorithme dans PaSTeL. Ces deux valeurs sont présentées ci-après :

$$n = \left\lceil \frac{1}{2}\sqrt{1 + 8\frac{T_{seq}}{W}} - \frac{1}{2} \right\rceil \tag{4.1}$$

$$T_{par} = I + \frac{T_{seq}}{n} + \frac{n-1}{2}W + S \tag{4.2}$$

Ce modèle est actuellement utilisé dans PaSTeL dans le but de prédire les performances lors de l'exécution. Les paramètres du modèle, I, W et S sont mesurés lors de l'initialisation de PaSTeL et sont constants si la vitesse des cœurs ne varie pas. En revanche, le temps séquentiel T_{seq} ne peut être déterminé lors de l'appel à PaSTeL. T_{seq} est donc estimé en mesurant le temps d'exécution d'une opération unitaire de l'algorithme en utilisant le compteur de cycle des cœurs. Si la première mesure se révèle trop petite pour obtenir un résultat précis, alors, une seconde mesure est effectuée sur un plus grand nombre d'éléments. Comme la complexité des algorithmes de la STL, est connue il est possible de prédire le temps séquentiel d'exécution T_{seq}.

Comme tous les paramètres sont maintenant connus, il est possible de déterminer si l'utilisation de l'algorithme parallèle (et donc de PaSTeL) va conduire à une accélération intéressante ou non. Une borne inférieur de T_{par} s'exprime facilement : $T_{par} > I + W + S$. Donc si $T_{seq} \leq I + W + S$, alors l'algorithme séquentiel est utilisé. Sinon, PaSTeL calcule le nombre optimal de threads en utilisant l'équation 4.1 et lance l'algorithme parallèle en utilisant n threads. L'équation 4.2 n'est pas directement nécessaire et n'est donc pas utilisée à ce moment.

Pour l'instant, ce mécanisme est implanté dans la classe d'algorithmes de *minimum*, mais peut aisément être étendu à d'autres classes. Le surcoût de

tant que TRAVAILLOCAL (l) **faire**
  CALCULERMORCEAU (l)
  si Cible de vol alors
    MISEAJOURETATGLOBAL (g,l)
    Attendre le départ des voleurs
  fin si
fin tant que

FIGURE 4.17 – Boucle de travail de PaSTeL

la mécanique est faible (quelques centaines de cycles), et montre des performances intéressantes comme il sera montré plus loin.

Néanmoins, si le temps des opérations unitaires est irrégulier, le mécanisme devra être étendu. Une technique similaire pourra quand même être utilisée si la distribution des temps de traitement est connue de l'utilisateur et s'il la communique à PaSTeL, ou alors s'il est possible de la déterminer à l'exécution.

4.3.2 Modélisation de la boucle de travail

Pour pouvoir évaluer correctement la taille des blocs à traiter, il est nécessaire d'étudier précisément la boucle de travail de PaSTeL, rappelée en Figure 4.17. De par l'ajout d'une mécanique de vol de travail, PaSTeL introduit un surcoût par rapport à une exécution purement séquentielle. Si l'on néglige les opérations de vol, alors ce surcoût se compose du test TRAVAILLOCAL (l), du coût de découpage des tableaux dans CALCULERMORCEAU (l) et du test **Cible de vol**. Pour un type de tableau donné et une plate-forme donnée ce coût est fixe pour chaque itération. En réglant donc le nombre d'éléments traités à chaque itération, appelé grain, il est possible de choisir le coût de la mécanique de PaSTeL par rapport au coût de l'algorithme séquentiel. Si le grain est trop faible alors le coût du moteur sera prohibitif, si il est trop important, alors le moteur perdra en réactivité.

Si on note b le surcoût d'une itération de la boucle et B le surcoût total de la boucle, G le grain, N le nombre d'éléments à traiter, T_{seq} le temps de traitement séquentiel de ce tableau et o la fraction du temps que l'on veut bien donner au moteur :

$$B = \frac{N}{G} b$$

$$o = \frac{B}{T_{seq} + B}$$

Il en découle que :

$$\frac{T_{seq}}{B} = \frac{1-o}{o}$$
$$\frac{T_{seq}G}{Nb} = \frac{1-o}{o}$$

Et donc :

$$G = \frac{Nb(1-o)}{T_{seq}o} \tag{4.3}$$

Ou si le surcoût est minimal :

$$G = \frac{Nb}{T_{seq}o} \tag{4.4}$$

Comme stipulé dans la partie précédente, le temps de traitement séquentiel du tableau est évalué à l'exécution. b peut être mesuré à l'initialisation de PaSTeL. Ce mécanisme n'est pas encore implanté dans PaSTeL.

4.4 Analyse des performances

Avant de faire des études plus poussées en utilisant PaSTeL il est nécessaire de vérifier que ses performances sont comparables avec d'autres bibliothèques parallèles existantes. Pour ce faire, MCSTL [SSP07] et Intel TBB [KV07] sont utilisées. MCSTL est une implantation parallèle de la STL basée sur le support OpenMP de gcc. TBB est une bibliothèque de vol ciblée pour les architectures multi-cœurs. Les performances seront étudiées sur deux plates-formes différentes, un ordinateur portable et un serveur de calcul. Ensuite, les performances du mécanisme de prédiction seront évaluées. Mais avant l'étude proprement dite, voici une brève description des plates-formes expérimentales et de la méthodologie suivie.

4.4.1 Plates-formes expérimentales et méthodologie

La première plate-forme utilisée dans cette étude est appelée *portable*. C'est un ordinateur portable équipé d'un processeur Intel Core2 Duo T7100 possédant 2 cœurs cadencés à 1.8 GHz et 2 Go de mémoire vive. Le bus mémoire est cadencé à 667 MHz.

La deuxième plate-forme est appelée *serveur*. C'est un serveur équipé de 8 AMD Opteron 875 possédant 2 cœurs cadencés à 2.2 GHz. Chaque processeur est associé à 4 Go de mémoire vive. On dispose donc de 16 cœurs et de 32 Go de mémoire. Néanmoins seuls 4 processeurs sont utilisés car des tests ont

montré que la bande passante mémoire était saturée lors de tests utilisant plus de 8 cœurs, et que donc les bibliothèques ne passent plus à l'échelle.

Sur *portable*, le noyau utilisé est un Linux 64 bit en version 2.6.22 et le compilateur est g++ 4.2.1. Sur *serveur*, le noyau est également un Linux mais en version 2.6.23 et le compilateur est g++ 4.2.3. Sur chaque plate-forme, les options de compilation utilisées sont : *-02* et *-DNDEBUG* pour désactiver les assertions.

Pour pouvoir comparer PaSTeL et TBB, les même algorithmes ont été implantés dans les deux bibliothèques. Pour cela, nous avons utilisé TBB20 version 20080319 et l'*auto partitioner* disponible dans la bibliothèque. Dans le cas de MCSTL (en version 0.8) les algorithmes étaient déjà disponibles et ont donc été utilisés tel-quels.

4.4.2 Évaluation brute du moteur

Pour valider les performances de PaSTeL nous proposons d'utiliser certains algorithmes de la STL. Trois algorithmes ont été sélectionnés. Ils appartiennent à des classes différentes d'algorithmes de la STL. Le premier algorithme est *min_ element* qui recherche la valeur minimum contenue dans un tableau. Le deuxième est *merge*, l'algorithme d'interclassement de tableaux triés. Le dernier est *stable_ sort*, une variante du tri qui présente la particularité d'être stable.

Les performances de ces trois algorithmes sont d'abord évalués sur *portable* puis sur *serveur*.

Performances sur *portable*

La première expérience présentée ici a pour but de comparer les performances de TBB, MCSTL et PaSTeL dans un environnement dual-cœur : 2 cœurs et donc 2 threads sont utilisés. Les performances de l'implantation séquentielle de la STL sont données comme référence. Les trois algorithmes précédemment présentés ont été exécutés sur *portable*, en utilisant des instances aléatoires de taille variant entre 50 et 50000 éléments. Les instances sont des tableaux d'entiers de type *int*. D'autres types de données simples ont été utilisés, mais les résultats étant similaires ils ne sont pas présentés ici. Le paramètre de grain, la taille des blocs, a été réglé manuellement pour chaque algorithme. Il faut noter que les entiers sont un *pire cas* car ce sont les éléments traités le plus rapidement. Il est également remarquable que ces valeurs soient des bornes supérieures du paramètre, en effet, quand le temps de traitement augmente, par exemple pour des types de données complexes, alors le grain diminue. Ces valeurs sont 50 itérations pour *min_ element*, 100

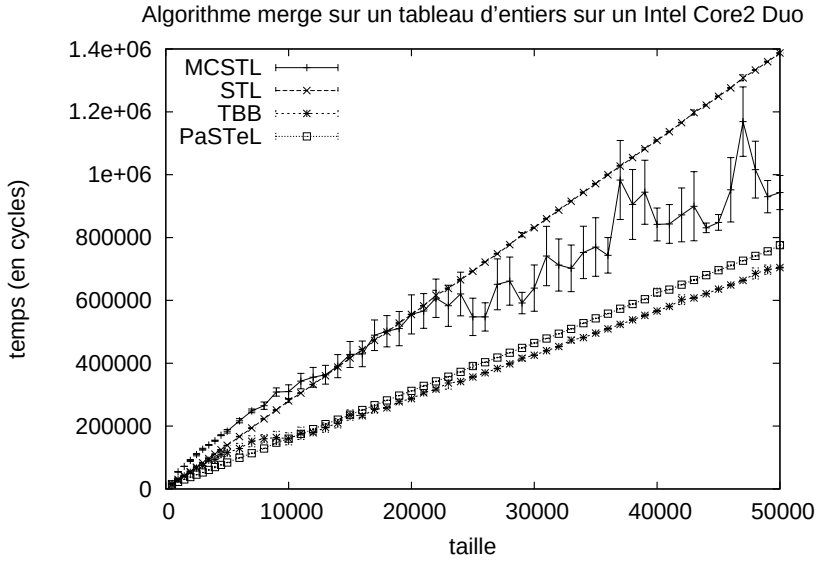


FIGURE 4.18 – Algorithme *merge* sur *portable*

pour *merge* et 400 pour *stable_sort*. Chaque mesure est répétée 20 fois, et chaque mesure est la moyenne de 20 exécutions sur la même instance pour se placer en régime permanent.

Les figures 4.18, 4.19, 4.20 et 4.21 présentent la moyenne des temps d'exécution en cycles et l'intervalle de confiance à 95% en fonction de la taille des données en nombre d'éléments, et ce pour les différents algorithmes. La figure 4.21 présente les résultats de l'algorithme *min_element* sur de très petits jeux de données, de taille comprise entre 0 et 3000 éléments. Cette dernière figure présente également les résultats obtenus en utilisant la prédiction de performance précédemment exposée.

La première information qui apparaît est que MCSTL est plus lent que PaSTeL et TTB sur les algorithmes *min_element* et *merge*. Ceci est particulièrement flagrant sur les tableaux de petite taille comme ceux présentés ici. Il est aussi notable que le surcoût fixe de MCSTL est beaucoup plus important que celui des autres implantations, spécialement sur l'algorithme *min_element*. Le fait que le support OpenMP de gcc soit basé sur des verrous (*mutexes*) ne suffit pas à expliquer la différence de performances. En effet, des expériences utilisant PaSTeL compilé pour utiliser des verrous permet de montrer que le surcoût n'est pas aussi élevé.

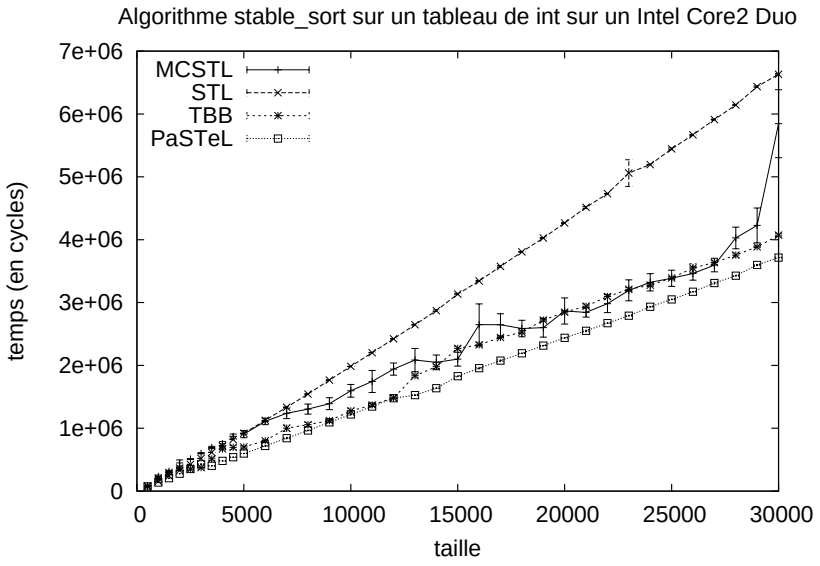


FIGURE 4.19 – Algorithme *stable_sort* sur *portable*

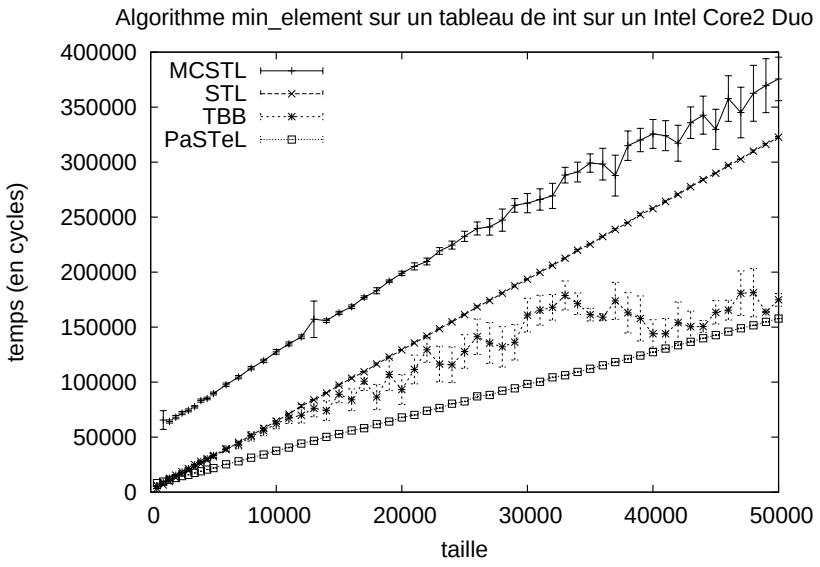


FIGURE 4.20 – Algorithme *min_element* sur *portable*

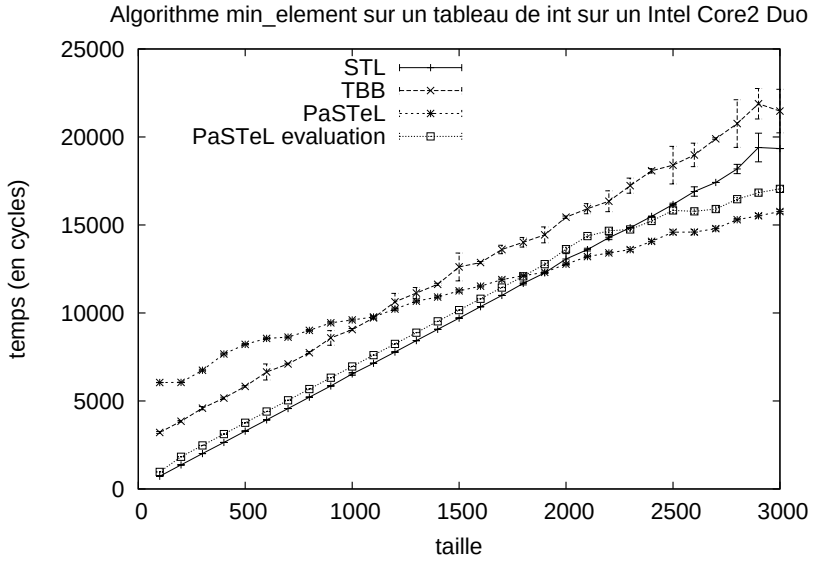


FIGURE 4.21 – Algorithme *min_ element* sur *portable*, détails

TBB pour sa part montre de meilleures performances que MCSTL, et est comparable avec PaSTeL. Effectivement, TBB est légèrement plus efficace que PaSTeL sur l'algorithme *merge* et légèrement moins efficace sur l'algorithme *stable_ sort* (cf. Figures 4.18 et 4.20). Le cas de *min_ element* est particulier car son temps de complétion est très court, moins de 200000 cycles, sur le plus gros tableau considéré. L'*auto partitioner* de TBB n'est pas très efficace sur des tableaux aussi petits alors que sur des données plus importantes l'algorithme *min_ element* de TBB et PaSTeL présente des performances similaires.

Un des objectifs de notre étude est de démontrer qu'avec les architectures multi-cœurs il est possible de construire des algorithmes parallèles qui soient efficaces sur des petits jeux de données. Sur *portable*, PaSTeL montre des accélérations sur des tableaux de moins de 3000 *int*, ou 15000 cycles, sur l'algorithme *min_ element*, de moins de deux fois 800 *int*, ou 20000 cycles, sur l'algorithme *merge* et sur moins de 900 *int*, ou 120000 cycles, sur l'algorithme *stable_ sort*. Définir une *petite* taille est difficile, néanmoins, PaSTeL présente des accélérations pour des tableaux plus petits que MCSTL ou TBB et peut être considéré comme réactif sur cette plate-forme.

Il faut également noter que sur des tableaux de grande taille (> 10000000)

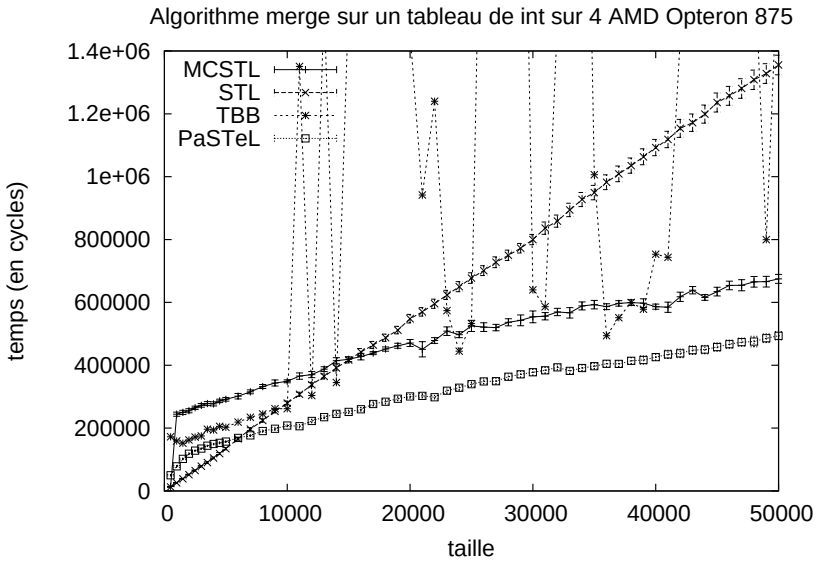


FIGURE 4.22 – Algorithme *merge* sur *serveur*

d'éléments, les différences de performances entre les bibliothèques deviennent négligeables.

Performances sur *serveur*

Les figures 4.22, 4.23, 4.24 et 4.25 présentent les mêmes expériences mais sur la plate-forme *serveur* en utilisant cette fois 8 cœurs et donc 8 threads pour les calcul parallèles. L'objectif est de vérifier le bon comportement de PaSTeL sur une architecture hiérarchique qui utilise plusieurs processeurs. Pour tous ces algorithmes, le grain à été fixé à 400 éléments, cette valeur s'étant avéré satisfaisante pour tous les algorithmes.

Les résultats de TBB sont présentés sans intervalles de confiance, ces derniers étant trop grands. Ceci est dû à des performances décevantes pour certaines exécutions. Des inter-blocages ont parfois été constatés, visiblement dus à une condition de course dan la terminaisons du moteur de thread de TBB. Plusieurs versions de TBB ont été essayées mais le problème a persisté. Néanmoins, quand les problèmes n'apparaissent pas, le comportement de TBB est comparable à celui obtenu sur *portable*.

Les résultats sont similaires à ceux obtenus sur *portable* : PaSTeL continue

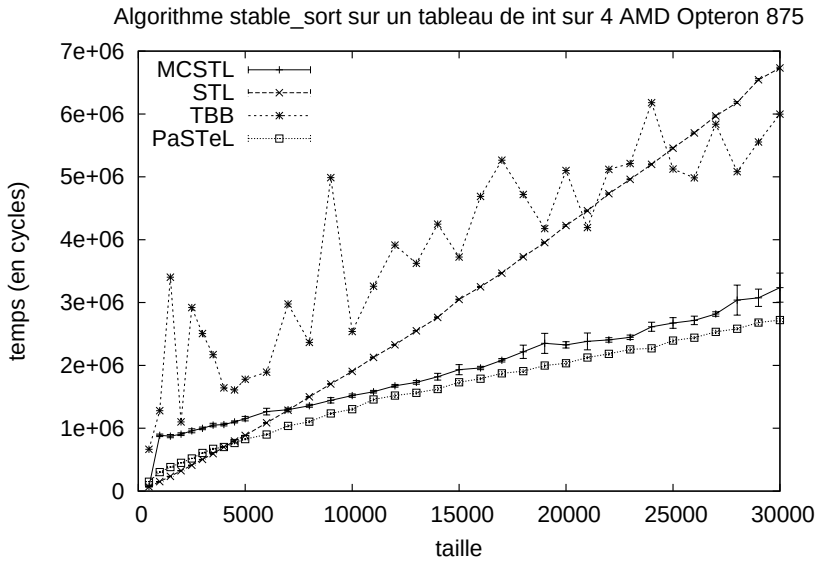


FIGURE 4.23 – Algorithme *stable_sort* sur *serveur*

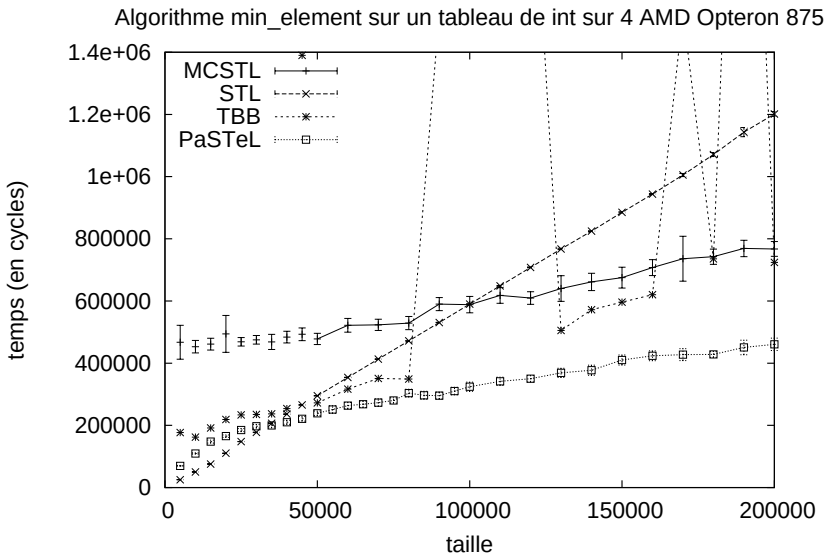


FIGURE 4.24 – Algorithme *min_element* sur *serveur*

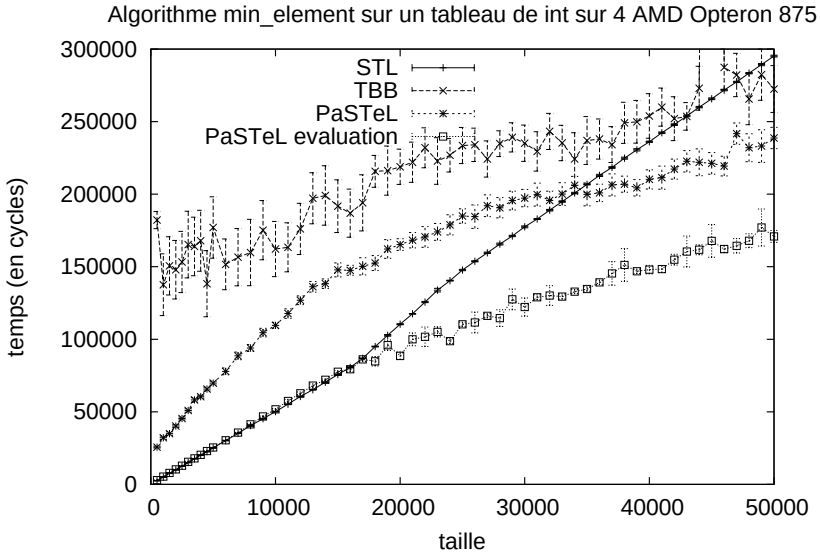


FIGURE 4.25 – Algorithme *min_ element* sur *serveur*, détails

d'obtenir des temps d'exécution inférieurs à MCSTL ou TBB. Le moteur de PaSTeL semble passer correctement à l'échelle avec le nombre de processeurs pour chaque algorithme testé. Cependant, l'algorithme *stable_ sort* sur des tableaux de plus de 1 million d'éléments de type *int* (non présentés ici) présente un comportement différent. La présence de barrières de synchronisation, dans l'implantation de cet algorithme spécifiquement, semble diminuer significativement le passage à l'échelle de cet algorithme. Dans ce cas, MCSTL devient meilleur.

L'algorithme *min_ element* de PaSTeL produit de meilleures performances que le code séquentiel pour des tableaux de plus de 18000 *int* sur cette plateforme. Sur l'architecture Core2, l'implantation parallèle montrait des gains de temps pour des tableaux de plus de 2500 *int*. Dans ce cas, l'architecture Core2 se révèle environ 7 fois plus réactive que l'architecture Opteron 875. Néanmoins, *serveur* possédant plus de processeurs, quand les temps de calcul deviennent plus importants les processeurs de *serveur* permettent de meilleurs temps de calcul. Par exemple, la fusion de tableaux de 50000 *int* sur *serveur* prend 500000 cycles alors que *portable* utilise 800000 cycles pour accomplir la même tâche. Bien sûr les processeurs de *serveur* fonctionnant à une fréquence supérieure à celle de *portable*, les différences en secondes

sont encore plus importantes. Ces chiffres sont donnés à titre indicatif pour montrer la disparité entre les architectures. Ils ne permettent pas d'évaluer la qualité de tel ou tel processeur.

4.4.3 Validation de la modélisation

La modélisation de PaSTeL présentée dans la Section 4.3 est basée sur un certain nombre d'hypothèses. Un moyen de vérifier la validité de ces hypothèses consiste à mener des expériences pour valider le modèle obtenu.

L'utilisation du moteur d'évaluation de performance qui essaie de déterminer le meilleur nombre de threads à allouer à une tâche est présentée sur les Figures 4.21 et 4.25. Ces figures présentent l'utilisation de la prédiction pour l'algorithme *min_ element*.

Sans la prédiction de performance, PaSTeL montre un surcoût sur les tableaux les plus petits car de trop nombreux threads sont assignés à une tâche trop petite. Sur *portable* (Figure 4.21), le surcoût induit par PaSTeL sans prédiction est de 5000 cycles. Ce coût est prohibitif sachant que l'algorithme séquentiel s'exécute en 3000 cycles. La prédiction du moteur de PaSTeL est capable de déterminer que le code séquentiel doit être utilisé pour des tableaux de moins de 2000 *int*, alors que l'implantation parallèle doit être utilisée pour des tableaux de taille supérieure. Dans l'implantation actuelle le surcoût de la prédiction est d'environ 100 cycles si le code séquentiel doit être utilisé, et environ 1200 si code parallèle doit être utilisé. En effet, quand le code parallèle est utilisé, le mécanisme doit calculer le nombre de threads à utiliser, ce qui demande plus de calculs. Cette phase aurait pu être retirée sur *portable*, un code parallèle ne pouvant utiliser plus des 2 cœurs disponibles, mais elle a été conservée pour pouvoir comparer son coût à celui sur *serveur*.

Sur *serveur* (Figure 4.25), l'utilisation de 8 threads par PaSTeL sans prédiction induit un surcoût important pour des tableaux de moins de 35000 *int*. Ce surcoût est d'environ 50000 cycles pour des tableaux de 10000 *int*. Cependant, le mécanisme de prédiction gomme presque entièrement le surcoût pour des tableaux de moins de 20000 *int*. Le surcoût est alors limité à 300 cycles. Pour les tableaux de plus de 20000 *int*, le mécanisme de prédiction est capable de sélectionner un nombre de thread permettant de meilleurs performances que l'algorithme séquentiel ou que PaSTeL utilisant 8 threads. Le mécanisme est donc capable de sélectionner un nombre de threads apportant un gain intéressant en temps de calcul.

Néanmoins, le fait d'être capable de déterminer un nombre intéressant de thread est un résultat qualitatif. Il serait instructif d'obtenir des résultats quantitatifs pour évaluer la précision de cette prédiction. Pour ce faire, trois

types d'exécution sont considérés :

— *Prédiction* est le temps de calcul de l'algorithme de PaSTeL en utilisant le mécanisme de prédiction pour sélectionner le nombre de threads ;

— *Meilleur* est le meilleur temps de calcul obtenu avec PaSTeL en faisant varier le nombre de threads. Cela inclut l'algorithme séquentiel ;

— *Pire* est le pire temps de calcul obtenu avec PaSTeL en faisant varier le nombre de threads.

La Figure 4.26 présente les performances du mécanisme de prédiction relativement à la meilleure performance atteignable sur *serveur* pour des tableaux de différentes tailles. Les résultats sont présentés comme une dégradation par rapport à la meilleure exécution. Trois quantités sont présentées dans la Figure 4.26 : le ratio entre *Meilleur* et *Meilleur*, le ratio entre *Prédiction* et *Meilleur* et le ratio entre *Prédiction* et *Pire*. Évidemment *Meilleur* atteint un ratio de 1 avec lui-même. Pour des tableaux de 50000 éléments, la figure montre que *Pire* n'obtient que 55% des performances de *Meilleur* c'est à dire du nombre optimal de threads. En d'autres termes *Pire* prend 2 fois plus de temps que *Meilleur*.

Les valeurs de *Meilleur/Prédiction* sont toujours supérieures à $0,8$, ce qui signifie que le mécanisme de prédiction ne perd jamais plus de 20% du temps. La plupart du temps, *Meilleur/Prédiction* est supérieur à $0,9$. Ces 20% peuvent paraître importants, mais il faut garder à l'esprit que *Meilleur* a été sélectionné manuellement, et qu'en cas d'erreur le résultat aurait pu être plus proche de *Pire*. Dans tout les cas, *Prédiction* est bien meilleur que *Pire*.

Une observation plus précise de la courbe en fonction de la taille du tableau révèle que *Prédiction* augmente pour des tailles de tableau comprises entre 1 et environ 15000 éléments. En fait, le mécanisme de prédiction démarre l'implantation séquentielle, le meilleur choix. Comme la prédiction possède un coût constant, l'efficacité augmente avec la taille du tableau. Après 15000, la non-monotonie du ratio indique que le mécanisme de prédiction fait quelques erreurs. Dans le cas du ratio de *Pire*, le nombre de threads diminue progressivement de 8 jusqu'à l'utilisation de l'algorithme séquentiel après 35000 éléments.

4.5 Discussion et conclusion

Dans ce chapitre, nous avons présenté PaSTeL notre outil destiné à l'analyse des stratégies de parallélisation. PaSTeL possède un modèle d'exécution basé sur du vol de travail qui présente des performances comparables à celles

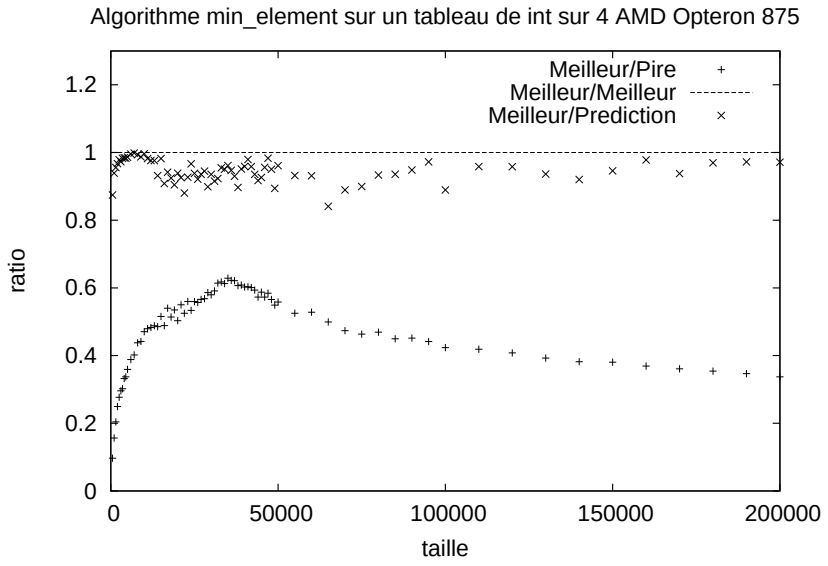


FIGURE 4.26 – Performances relatives du mécanisme de prédiction.

d'autres bibliothèques parallèles, ainsi qu'une large gamme d'algorithmes utilisant ce modèle.

PaSTeL possède également un module de prédiction de performances qui permet de régler le nombre de threads lors de l'exécution. Il est également possible de choisir entre plusieurs méthodes de synchronisation.

Cependant cette souplesse a un coût. En effet, l'implantation d'un algorithme en PaSTeL est beaucoup plus complexe que l'implantation du même algorithme dans une bibliothèque dédiée comme TBB.

Néanmoins, en l'état, PaSTeL offre de nombreuses possibilités expérimentales et une bonne reproductibilité des mesures. Nous allons donc l'utiliser pour évaluer les performances des algorithmes proposés sur de nombreuses configurations différentes, et comparer l'influence des paramètres en fonction du type de machine. Nous disposons à cet effet de nombreuses machines avec des caractéristiques différentes au sein de Grid'5000. Mais au vu du nombre de paramètres envisageables, il va être nécessaire de se doter d'outils pour mener une expérience aussi complexe. Le chapitre suivant présente donc un outil de conduite d'expérience sur grille de calcul.

Chapitre 5

Expo : un outil de conduite d'expériences pour grilles

La construction d'un moteur de conduite d'expériences adapté à une plate-forme particulière présente de nombreuses difficultés. Notamment, il faut s'adapter à la plate-forme cible sans pour autant perdre en généricité. En effet, toute plate-forme est destinée à évoluer, et une spécialisation trop importante limite les capacités d'adaptation d'un programme. Les deux doivent donc pouvoir évoluer de concert.

Un soin particulier a donc été apporté à la construction d'une expérience *type* permettant de définir clairement les besoins de notre moteur de conduite d'expériences. Ce cas d'utilisation est non trivial, dans le sens où il exécute de nombreuses commandes différentes sur de nombreuses machines tout en nécessitant un soin particulier dans l'analyse des résultats. Il met l'accent sur l'aspect mesure de performances et reproductibilité.

Cette expérience nous a permis à la fois de définir les caractéristiques de notre moteur, mais a également mis en avant un certain nombre d'écueils à éviter au cours du processus expérimental. Ces problèmes peuvent être évités en utilisant une méthodologie rigoureuse. Cette méthodologie est présentée et influe sur l'architecture du moteur.

Comme nous souhaitons qu'Expo puisse évoluer, il nous a paru opportun d'utiliser une architecture modulaire. Cette architecture permet de séparer les principales fonctionnalités d'Expo et permet même d'envisager, moyennant le développement de quelques modules, son utilisation sur des plate-formes différentes de notre cible d'origine.

La construction et les fonctionnalités d'Expo sont validées à la fois par des tests unitaires et par des cas d'utilisation. Cette validation est capitale car l'objectif d'un tel moteur est de simplifier l'expérience de l'utilisateur final. Les tests unitaires permettent de garantir que les modules respectent

les contraintes définies tandis que les cas d'utilisation permettent de juger du gain apporté par le moteur au cours d'une expérience classique.

Dans une première section, nous allons présenter l'étude de cas qui a déterminé les besoins d'Expo. Dans un second temps, une modélisation de la conduite d'expériences va être exposée. Ensuite, nous décrirons la construction d'Expo et de ses modules. Finalement nous validerons Expo à travers deux cas simples d'utilisation.

5.1 Une étude de cas

Pour définir précisément les besoins du moteur d'expérience et les enjeux de la démarche, une étude de cas a été conduite sur Grid'5000. Pour être représentatif d'une expérience utilisant cette plate-forme, le cas d'étude a été choisi avec attention. Les critères qui devaient être remplis étaient : le programme utilisé devait être distribué sur un nombre suffisant de nœuds, il devait être intensif sur plusieurs ressources de la plate-forme et si possible être utilisé sur Grid'5000. Le programme *mput* remplissait cet ensemble de conditions et a donc été sélectionné.

5.1.1 Le programme *mput*

La fonction de cette application est la diffusion de fichiers avec des performances de haut niveau. À cette fin, il utilise une chaîne de connexions TCP pour diffuser les fichiers. À l'origine, ces fichiers ne sont présents que sur un nœud appelé nœud source [MR01]. Lors du démarrage, le programme se réplique sur chaque nœud à l'aide d'un lanceur parallèle intégré, Taktuk 2[MRH05]. Ainsi, le lanceur peut utiliser un arbre de déploiement d'arité fixée ou dynamique. L'arité dynamique est obtenue grâce à un algorithme de vol de travail. Une fois tous les nœuds lancés, une communication collective prend place et permet un échange des adresses réseau. Ceci est fait grâce à la couche de communication intégrée au lanceur. Les nœuds se chaînent les uns aux autres par ordre d'adresses réseau. Le programme peut donc être vu comme un pipeline possédant une profondeur égale nombre de nœuds recevant les fichiers. Cette chaîne est représentée sur la Figure 5.1. Le pipeline est alimenté depuis l'émetteur avec les fichiers à transférer, tandis que les autres nœuds écrivent les fichiers sur le disque et transfèrent au nœud suivant. Le programme est une application multi-threads écrite en C++ et représente quelques 2000 lignes de code.

Comme on peut le deviner, ce programme est très sensible. En effet, il possède autant de points de contention potentiels que de nœuds utilisés.

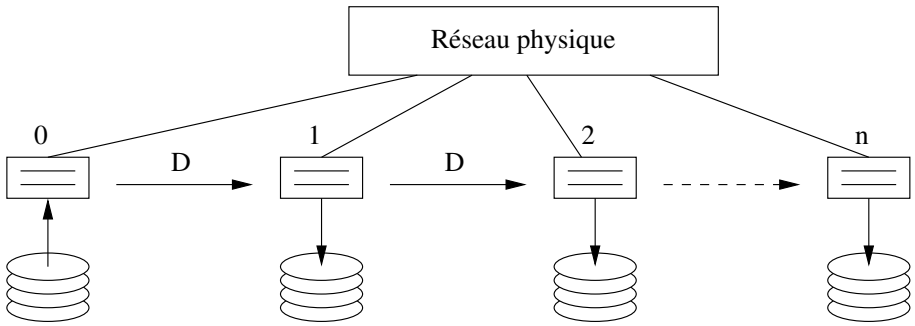


FIGURE 5.1 – Le pipeline de *mput* : les nœuds sont inter-connectés par un réseau à haut débit, le nœud racine (numéroté 0) lit les fichiers sur son disque et les envoie au nœud suivant qui les écrit sur son disque et les envoie au nœud suivant, et ainsi de suite.

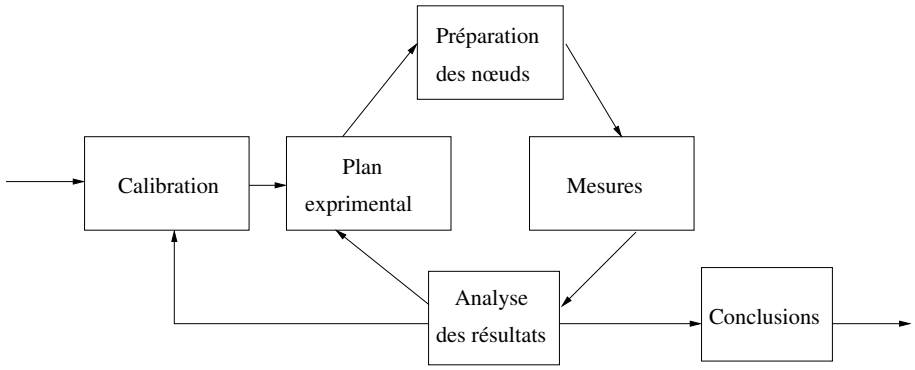


FIGURE 5.2 – Le processus expérimental suivi est itératif et composé de 5 étapes. Quand de nouveaux problèmes sont identifiés, un nouveau programme de test est créé et ajouté à l'ensemble des programmes de calibration. La machine est à nouveau étalonnée.

Les premières causes apparentes de mauvaises performances peuvent être le débit des disques de chaque machine et la bande passante du réseau. Mais il n'est pas possible d'éliminer à priori d'autres paramètres tels que la charge des processeurs ou la taille des fichiers à transférer.

5.1.2 Méthodologie

Pour obtenir de bons résultats, nous proposons d'améliorer de manière itérative la qualité de l'expérience. Les problèmes sont résolus au fur et à mesure jusqu'à ce que la qualité de l'expérience soit jugée satisfaisante.

La méthodologie suivie pour les expériences est représentées sur la Fi-

gure 5.2.

La première étape consiste à obtenir des valeurs de référence des performances de la plate-forme en général et de chaque type de machine disponible en particulier. C'est la phase de calibration. Cette étape permettra de décider si une machine est dans un état compatible avec la conduite d'une expérience ou si elle a besoin d'être réparée et donc doit être exclue des machines utilisées pour cette expérience.

Un plan expérimental est ensuite choisi. Il doit prendre en compte à la fois l'objectif de l'utilisateur et les contraintes de la plate-forme. En effet, le plan dépend de l'architecture et des ressources disponibles.

Pour exécuter le plan, un ensemble de machines plus grand que nécessaire est sélectionné et chaque machine est testée puis comparée aux valeurs de référence. Les machines montrant des performances insatisfaisantes sont éliminées.

Le plan d'expérience est alors exécuté sur l'ensemble des machines restantes. On obtient ainsi des résultats mais également des rapports d'erreur et des détails sur le déroulement de l'expérience. Par exemple, il est possible de savoir quelle machine a exécuté quelle commande et à quelle date. Ces détails *annexes* sont précieux car se sont eux qui vont permettre de comprendre pourquoi une expérience s'est bien ou mal déroulée.

Les résultats sont analysés une première fois pour vérifier le bon déroulement de l'expérience. Deux raisons peuvent expliquer des résultats insatisfaisants. Premièrement, le plan expérimental peut être mauvais et ne rien montrer d'instructif, néanmoins les données sont valides et peuvent être utilisées lors d'expériences futures. Un nouveau plan est alors défini. Deuxièmement, un point important a pu être oublié lors de la phase de calibration et certains nœuds peuvent se trouver dans un état incompatible avec l'expérience, avec pour conséquence des résultats incohérents. Les machines problématiques doivent êtres identifiées, et ceci fait, un nouveau programme de test doit être écrit et ajouté à l'ensemble des tests pour prendre en compte ce nouveau cas de figure. Dans tous les cas, tout ou partie des résultats obtenus sont invalides.

Une fois des résultats satisfaisant obtenus, il est possible de les interpréter et de tirer d'éventuelles conclusions sur les enseignements apportés par l'expérience.

5.1.3 Détails expérimentaux

Certains détails de l'expérience sont dépendants de l'état de la plate-forme à la date de l'expérience. Ces détails ont en partie contraint le plan expérimental et conditionné la manière dont l'expérience a été conduite.

La première chose à remarquer est, qu'à l'heure actuelle, il n'existe pas d'outil pour mesurer précisément l'état global de Grid'5000. Pour cette raison, les expériences présentées dans cette section ont été réalisées la nuit par un expérimentateur seul sur les grappes dans le but d'éviter d'éventuelles interférences réseau.

L'un des avantages de Grid'5000 est son hétérogénéité contrôlée. En effet deux tiers des nœuds sont homogènes et, comme l'impact du matériel n'était pas un paramètre de l'expérience, seuls des nœuds de configuration identique ont été utilisés. Les nœuds utilisés sont des eServer 325 d'IBM configurés de manière identique et utilisant la même distribution Linux, la même version du noyau et la même version du compilateur gcc.

Deux grappes sont utilisées pour cette expérience. Elles sont distantes de 400km, l'une se trouvant sur le site d'Orsay et l'autre sur le site de Lyon. Elles étaient composées respectivement de 216 et 56 nœuds à la date de l'expérience. Les deux grappes sont inter-connectées par un lien Gigabit Renater 3. Dans les grappes, les nœuds sont inter-connectés par un réseau Ethernet Gigabit.

Avant chaque mesure, les nœuds sont testés en utilisant deux programmes de test simples. Le premier éprouve le disque en écrivant un fichier de 1GiB. Le second vérifie le processeur en effectuant le calcul de la fonction de Ackermann pour les paramètres 4 et 1. Le temps de complétion des deux programmes de test est utilisé pour supprimer les nœuds lents de l'ensemble des machines valides. Les nœuds lents sont ensuite étudiés pour comprendre d'où vient le problème. Ce processus peut réduire considérablement le nombre de machines disponibles pour l'expérience : dans ce cas d'étude 50% des nœuds se sont révélés inutilisables pour l'expérience. Cela ne serait plus le cas aujourd'hui car la qualité de la plate-forme a beaucoup progressé.

5.1.4 Plan expérimental

Comme l'expérience réalisée ne doit pas être triviale et doit contenir de nombreux paramètres, un plan d'expérience factoriel fractionnaire avec deux niveaux pour chaque paramètre et avec répétitions est utilisé. L'analyse détaillée de plan expérimentaux fractionnaires avec répétitions peut être trouvées dans [Jai91b], au chapitre 18. Ce type de plan permet d'obtenir une modélisation linéaire de la réponse du système à la variation des paramètres. Cette modélisation ne prend pas en compte les interactions entre les paramètres, mais elle utilise un nombre minimal de manipulations via l'utilisation d'une matrice d'expérience orthogonale. Les répétitions permettent d'améliorer la confiance statistique des résultats.

Le plan utilisé est représenté sur la Figure 5.3. Chaque ligne de la matrice

	I	A	B	C	D	E	F	G
1	+1	+1	+1	+1	+1	+1	+1	+1
2	+1	−1	+1	+1	−1	+1	−1	−1
3	+1	+1	−1	+1	−1	−1	+1	−1
4	+1	−1	−1	+1	+1	−1	−1	+1
5	+1	+1	+1	−1	+1	−1	−1	−1
6	+1	−1	+1	−1	−1	−1	+1	+1
7	+1	+1	−1	−1	−1	+1	−1	+1
8	+1	−1	−1	−1	+1	+1	+1	−1

FIGURE 5.3 – Plan factoriel fractionnaire d'expérience utilisé : les mesures (1..8) et les niveaux (+1,-1) des paramètres (A..G)

correspond à une mesure qui est répétée 8 fois. Pour chaque mesure, les paramètres sont fixés conformément aux niveaux donnés dans la matrice. Les paramètres et leurs différentes valeurs, ainsi que les niveaux correspondants sont :

— A : taille du fichier à diffuser, 10MiB (-1) ou 1GiB (+1),
— B : nombre de nœuds, 4 (-1) ou 32 (+1),
— C : nombre de grappes, 1 (-1) ou 2 (+1),
— D : nombre de processeurs utilisés par nœud, 1 (-1) ou 2 (+1),
— E : arité de l'arbre de déploiement, fixée (-1) ou dynamique (+1),
— F : charge des nœuds, non chargé (-1) ou chargé (+1),
— G : temps de pause entre les mesures, 0s (-1) ou 60s (+1).

I n'est pas un paramètre à proprement parler, c'est la réponse moyenne du système étudié. Soit, dans notre cas le temps d'exécution moyen de l'application. La charge machine, le paramètre F, est créé par un programme de charge, une boucle infinie, qui utilise donc uniquement du temps processeur. Un nœud chargé utilise une instance du programme de charge par processeur utilisé. Si y est la réponse du système étudié, l'influence des paramètres peut être modélisée comme suit :

$$y = q_0 + q_A x_A + q_B x_B + q_C x_C + q_D x_D + q_E x_E + q_F x_F + q_G x_G + e,$$

Où e est l'erreur, q_0 est la réponse moyenne du système, q_i est l'influence du

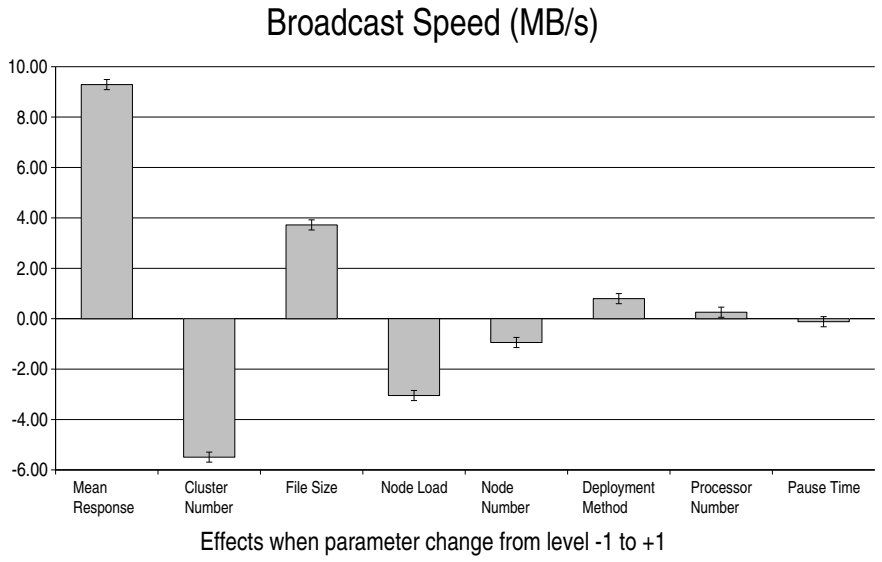


FIGURE 5.4 – Résultats de l'étude de cas : les effets sont triés par influence décroissante sur la vitesse de transfert en MiB/s

paramètre i quand son niveau change de -1 à $+1$. Les x_i sont les niveaux des paramètres correspondant, -1 ou $+1$. Les q_i sont obtenus en résolvant le système d'équations linéaires obtenu grâce aux différentes mesures.

5.1.5 Résultats

Les résultats de l'étude de cas sont présentés sur la Figure 5.4. Trois effets principaux peuvent être identifiés : le nombre de grappes (C), la taille du fichier (A) et la charge des nœuds (F). Dans le meilleur cas (le transfert d'un fichier de 1 GiB sur des nœuds appartenant à une seule grappe et sans charge) le débit mesuré atteint 20 MiB/s alors que le modèle prédit : $21MiB/s = 9(MeanResponse) + 5.5(x_C = -1) + 3.5(x_A = +1) + 3(x_F = -1)$. Dans le pire cas (le transfert d'un fichier de 10 MiB sur des nœuds appartenant à deux grappes différentes avec une charge) le débit chute à 3 MiB/s alors que le modèle prédit -2 MiB/s. Ce résultat peut paraître aberrant mais il faut se rappeler que le modèle choisi est simpliste. Ce modèle est un modèle linéaire simple qui ne tient pas compte des interactions entre les différents paramètres. Une prédiction plus fine serait obtenue en utilisant un modèle plus complexe et en faisant des mesures supplémentaires.

L'impact de la taille du fichier est assez simple à expliquer : avec un fichier

de 10 MiB, la plupart du temps est consacrée au déploiement de l'application. L'impact de la charge montre que le programme nécessite de la puissance de calcul pour gérer le débit qui traverse chaque nœud : 20 MiB/s en entrée, 20 MiB/s en sortie et 20 MiB/s d'écriture sur le disque. Ce n'était pas a priori un facteur limitant, et il est donc bon qu'il ait été conservé dans le plan d'expérience final. L'impact du nombre de grappes est moins compréhensible et va être expliqué dans la section suivante. Il peut néanmoins être considéré comme un artefact. Les résultats, bien qu'intéressants, ne sont pas le but de cette expérience. Le vrai but est l'identification des problème rencontrés. Ils sont présentés ci-après.

Problèmes de conduite d'expériences

Tout d'abord, les mesures se sont révélées difficiles à réaliser sans intervention humaine. Le lanceur parallèle utilisé à l'époque n'étant pas assez mature, on observait de très rares inter-blocages. Il était parfois nécessaire de terminer ces tâches bloquées. Parfois, les performances étaient visiblement trop faibles du fait de défaillances matérielles ou de problèmes de configuration. Il était alors nécessaire d'identifier le coupable et de recommencer les mesures.

Ce processus n'a pas pu être aisément automatisé. Pour identifier les problèmes, il a fallu interpréter les résultats des mesures en continu, et déterminer si une mesure avait échoué ou avait été perturbée. Il a fallu alors évaluer à nouveau l'état des nœuds pour constituer un nouvel ensemble de machines valides. Nous en avons conclu à la nécessité d'inclure dans le script la gestion de programmes de test et l'analyse de leur résultats.

Problème d'hétérogénéité

Le premier problème trouvé a été une partition des nœuds en deux groupes distincts en fonction de leurs performances disque. Certains nœuds présentaient des performances anormalement faibles et ceci, apparemment, de manière aléatoire. Les performances variaient du simple au double et le programme de test pouvait s'exécuter trois fois correctement et présenter un problème la quatrième fois. Une étude attentive du matériel a révélé que les disques étaient de deux marques différentes. Les disques d'une marque présentaient parfois des performances faibles, notamment après une utilisation intensive. Il a donc fallu écarter la moitié des nœuds pour les besoins de cette expérience.

Le problème inverse a été rencontré, c'est à dire un disque présentant des performances supérieures aux autres. Il s'est avéré que le partitionnement

géométrique du disque posait problème. Le disque écrivait donc les données sur une partie plus performante. Ce nœud a été configuré de manière identique aux autres et les performances sont redevenues comparables.

Malgré cette première élimination, pendant les mesures, des performances inférieures à celle attendues était mesurées. Ceci arrivait souvent quand un grand nombre de nœuds était utilisé. L'utilisation du programme mput sans écriture sur le disque nous a aiguillés sur un problème réseau. Une recherche dichotomique sur la chaîne de transfert a permis d'identifier des coupables potentiels, qui variaient selon le jour où était réalisée cette expérience. Il s'est avéré qu'au redémarrage, et de manière aléatoire, les nœuds avaient une chance faible, de l'ordre de 5%, de configurer leur interface réseau dans un mode 100 MiB/s au lieu du mode 1 GiB/s attendu. Ces nœuds ralentissaient le pipeline. Une mise à jour du micro-logiciel est venu corriger ce problème, mais le micro-logiciel n'était pas disponible lors de la campagne d'expérience. Il a donc fallu vérifier les nœuds à chaque redémarrage et notre ensemble de programmes de test s'est enrichi d'un test réseau adapté.

Performances faibles sur deux grappes

Comme montré sur la Figure 5.4, dans la colonne *Cluster Number*, quand deux grappes sont utilisées, les performances s'écroulent. Ce résultat s'est révélé surprenant au premier abord car cet outil de diffusion est habituellement limité par les performances du disque et pas par celles du réseau. La bande passante était en fait limitée par la taille de la fenêtre TCP. Les grappes étant distantes de 400 km, la latence est importante et les fenêtre TCP doivent être agrandies en fonction. Les mesures devraient êtres refaites pour compléter l'expérience. Mais comme on peut le voir sur la Figure 5.3 dans la colonne C seules les mesures 1, 2, 3 et 4 doivent êtres répétées, les autres ne concernent qu'une grappe.

5.2 Modélisation de la conduite d'expérience

Les problèmes rencontrés dans la conduite d'une expérience de taille limitée, 32 nœuds sur 2 grappes, vont empirer quand la taille de l'expérience va croître. Il est nécessaire de développer des outils pour résoudre ces problèmes. Dans l'idéal, ces outils doivent être capables d'exécuter l'expérience sans interaction humaine, gérer les ressources, vérifier l'état de la plate-forme et archiver les traces de l'exécution. Dans ce but, nous proposons un modèle de processus expérimental automatique.

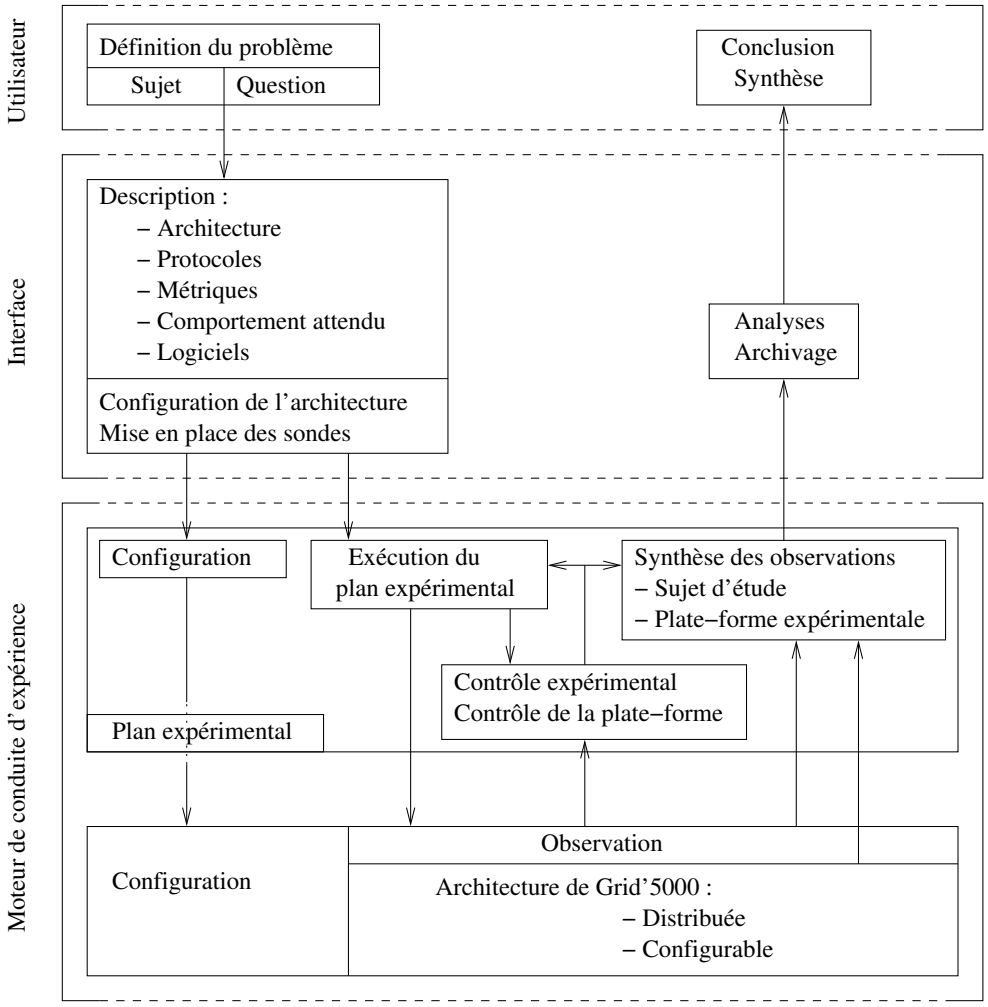


FIGURE 5.5 – Une expérience pilotée par un moteur

Une expérience exécutée automatiquement sur une grille comme Grid'5000 peut être modélisée comme sur la Figure 5.5.

La première étape consiste pour l'utilisateur à définir son problème, c'est à dire l'objet de son étude et la question posée.

Il formule ensuite son expérience dans un langage adapté. Ce langage doit permettre d'exprimer de multiples aspects du processus expérimental. Par exemple quels fichiers sont produits par quelle commande, les valeurs de retour attendues des commandes, les dates d'expiration ou les environnements requis par l'utilisateur. Cette description doit décrire le flot des commandes et, en fonction du résultat de ces commandes, le chemin à suivre dans le flot. Ce programme est un plan expérimental qui sera suivi par le moteur d'expérience.

Ceci fait, le plan expérimental peut être exécuté. Tout d'abord, le moteur prend en charge la configuration de la plate-forme et teste son bon comportement. Si un nombre suffisant de ressources est jugé acceptable, alors le moteur procède à la conduite du plan. Le moteur vérifie le bon comportement des commandes tel que spécifié par l'utilisateur et archive les résultats. Il peut être utile de vérifier périodiquement le statut de la plate-forme, et il peut même devenir nécessaire d'interrompre l'expérience si le nombre de ressources disponibles devient trop faible. Si une erreur survient, en fonction des spécifications de l'utilisateur, le moteur peut, soit exécuter cette commande à nouveau, soit interrompre l'expérience, soit continuer le plan en signalant l'erreur.

Une fois le plan terminé, le moteur rend la plate-forme dans son état d'origine, sauvegarde les fichiers générés ainsi que les sorties des commandes. Un certain nombre d'analyses préliminaires peuvent être effectuées à ce stade. Les résultats sont ensuite fournis à l'utilisateur.

5.3 Le moteur de conduite d'expérience *Expo*

Pour répondre aux différents problèmes rencontrés dans les sections précédentes, un moteur de conduite d'expérience à l'architecture modulaire a été proposée. Cette architecture ainsi que les différents modules qui la composent sont présentés dans les parties suivantes. L'utilisation de ce moteur sera présentée à travers deux cas d'utilisation simples.

5.3.1 Une architecture client serveur

Une expérience pouvant se prolonger sur plusieurs jours, une perte de connexion est toujours possible entre la machine de l'expérimentateur et les

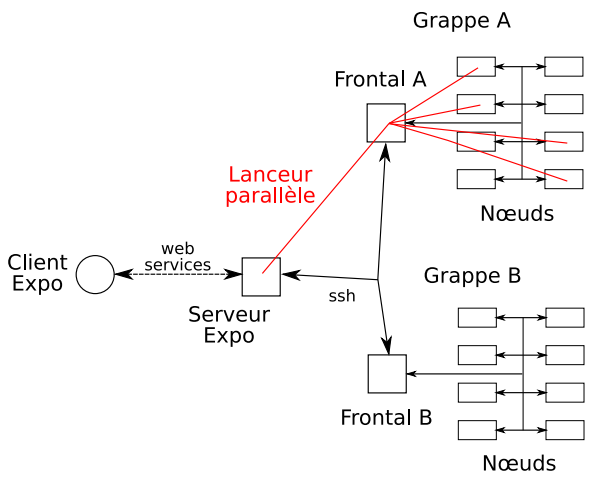


FIGURE 5.6 – Utilisation d'*Expo* sur grille légère

ressources qu'il utilise. Pour pouvoir garantir une exécution asynchrone de l'expérience, une architecture client/serveur a été implantée. Dans un cas d'utilisation classique (cf. Figure 5.6), l'expérimentateur utilise le client du moteur d'expérience sur sa machine personnelle. Il se connecte au serveur qui est chargé de l'exécution des différentes commandes. Lorsque les commandes qu'il veut exécuter sont parallèles c'est le lanceur parallèle qui est utilisé. Les différents modules qui composent le gestionnaire d'expérience s'exécutent soit sur le client, soit sur le serveur. Le développement s'est fait par une approche *bottom up*. Un certain nombre de composants fiables et réutilisables ont donc été créés au cours du développement. Cette architecture est représentée sur la Figure 5.7.

Le premier composant de cette architecture est le module d'exécution de commande. Couplé à un lanceur parallèle, il permet de gérer les différentes commandes d'une expérience distribuée. Le module de réservation permet une communication de haut niveau avec les gestionnaires de ressources. Le module d'expérience permet d'unifier les commandes et les ressources dans une entité logique : une expérience. Ces expériences sont exprimées dans le langage de description d'Expo et les différentes commandes instanciées par le module de génération de commandes.

5.3.2 Module d'exécution de commande

Le module de commande est le cœur du serveur, il est présenté sur la Figure 5.8. Il gère l'exécution asynchrone ou synchrone des commandes qui lui sont envoyées. Il permet d'accéder à différentes informations sur les commandes terminées ou en cours d'exécution grâce à un identifiant unique qui

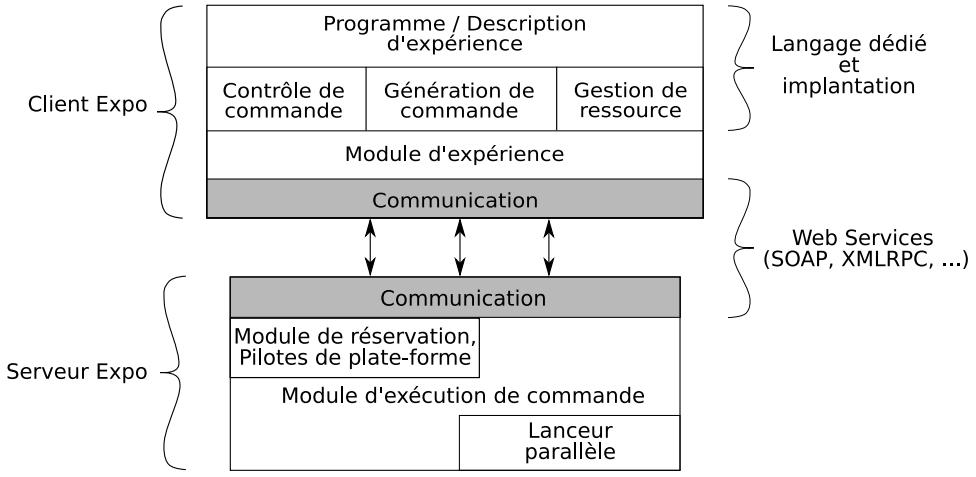


FIGURE 5.7 – Architecture client serveur modulaire

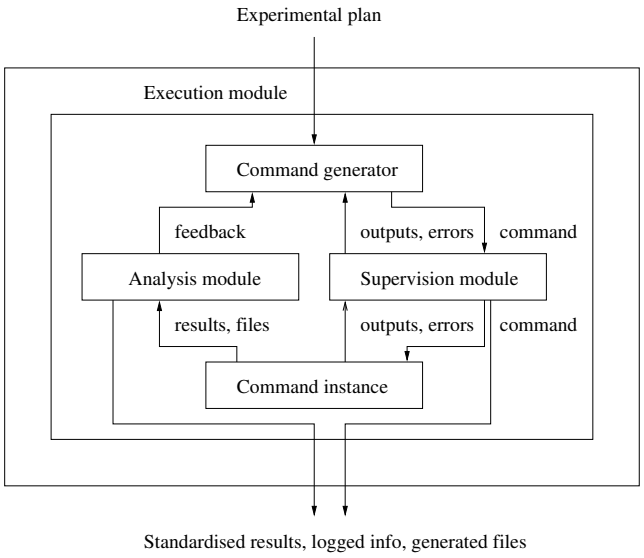


FIGURE 5.8 – Le module d'exécution de commandes

caractérise chaque commande. Les informations disponibles sont :

— l'état de la commande (lancée, terminée, prévue),
— les dates de début et de fin de la commande,
— le statut de retour,
— les flux de sortie et d'erreur (stdout, stderr).

Les flux de sortie peuvent être soit conservés en mémoire, soit directement archivés sur le disque, en fonction des préférences de l'utilisateur. De plus, le module de commande est récursif, et permet d'envoyer des commandes à un serveur *Expo* en passant par un autre serveur *Expo*. La charge peut donc être répartie sur différents serveurs. En effet, la gestion de nombreuses commandes en parallèle peut s'avérer coûteuse.

5.3.3 Lanceur parallèle

Le lanceur parallèle est une brique indispensable d'un moteur d'expérience. Il permet de ne pas déployer trop de serveurs d'exécution *ad hoc* et d'utiliser ceux qui sont nativement installés sur les nœuds, par exemple le serveur *ssh* qui est installé sur toutes les machines de Grid'5000.

Le lanceur parallèle utilisé actuellement par *Expo* est *Taktuk* [MRH05]. C'est un lanceur rapide dont le déploiement se fait par vol de travail. Le temps de déploiement est proportionnel au logarithme du nombre de nœuds cibles. Taktuk correctement configuré permet de structurer les entrées et sorties produites par les commandes qu'il lance en parallèle. Il est donc facile de les analyser ensuite ou de les stocker pour archivage.

5.3.4 Module de réservation

Le module de réservation se base sur le module de commande pour communiquer avec les gestionnaires de ressources. Cette communication se fait par le biais de pilotes spécifiques à la plate-forme cible. De même que pour les commandes, chaque réservation, et donc les ressources qui lui seront associées, possède un identifiant unique. On peut également imaginer se servir de ce module pour se connecter aux outils de supervision de la plate-forme.

5.3.5 Module d'expérience

Le module d'expérience permet de réunir des commandes et des réservations dans une entité logique qui correspond à une expérience. Chaque expérience possède un identifiant unique. Il est donc toujours possible d'ajouter une commande ou des ressources dans une expérience. En combinant ces trois modules : le module d'exécution de commande, le module de réservation, le

Commande	Description
Commandes générales	
check	Vérification des nœuds
task	Exécution d'une tâche
atask	Exécution d'une tâche asynchrone
ptask	Exécution d'une tâche parallèle
patask	Exécution d'une tâche parallèle asynchrone
barrier	Attente des tâches asynchrones en cours d'exécution
copy	Copie d'un fichier d'une machine à une autre
Commandes spécifiques Grid'5000	
oargridsub	Réservation de ressources en utilisant oargrid
oargridconnect	Connexion à des ressources préalablement réservées
kadeploy	Déploiement d'un système d'exploitation sur des noeuds
akadeploy	Déploiement asynchrone d'un système d'exploitation
kadeploy_advancement	Accès à l'avancement des déploiements

FIGURE 5.9 – Liste des commandes disponibles dans *Expo* assortie d'une brève description

module d'expérience, on peut recréer le détail du déroulement d'une expérience. La partie cliente de ce module propose, par exemple, d'archiver ces informations dans un format de données structuré pour une étude ultérieure.

5.3.6 Module de génération de commandes et langage de description d'expériences

Pour permettre un pilotage facile du moteur d'exécution (le module de commande, le module de réservation, le module d'expérience) il est nécessaire de présenter une interface de plus haut niveau à l'utilisateur final. Un langage de description d'expériences a donc été conçu. Ce langage est basé sur Ruby, un langage de script orienté objet. On peut le classer dans la catégorie des *domain specific languages* ou langages dédiés. Une liste des commandes et de leur description est donnée dans la Figure 5.9.

Ce langage présente à l'utilisateur des abstractions :

— de commandes et de commandes parallèles,
— de commandes synchrones et asynchrones,
— de réservations,
— de sections parallèles,

```
1 require 'expo_g5k'
2 oargridsub :res => "gdx:nodes=10,helios:nodes=10"
3 check $all
4 ptask $all.gateway, $all, "date"
5 id, res = ptask $all["gdx"].gateway, $all["gdx"], "sleep 1"
6 res.each { |r| puts r.duration }
7 puts "moyenne : " + res.mean_duration
```

FIGURE 5.10 – Une utilisation simple d'*Expo* dans le contexte de Grid'5000

— de barrières.

Une section parallèle permet d'exécuter le moteur *Expo* en parallèle. Il introduit également la notion d'ensemble de ressources. Ces ensembles de ressources agrègent des ressources dans une unité logique et leur associent des propriétés. Par exemple, on peut réunir les noeuds d'une même grappe dans un même ensemble de ressources en leur associant le même frontal, ainsi que les mêmes propriétés physiques si la grappe est homogène. Ces ensembles peuvent subir diverses transformations, par exemple concaténations et projections. Il est également possible d'itérer sur les éléments de ces ensembles.

5.4 Cas d'utilisation

Pour illustrer le fonctionnement d'*Expo*, deux cas d'utilisation vont être présentés. Un premier cas, purement démonstratif et qui sera donc très simple, et un second qui correspond à une expérience réelle : l'évaluation d'un programme de diffusion de fichiers.

5.4.1 Cas simple

Le code du premier cas d'utilisation est présenté sur la Figure 5.10. Comme cet exemple a été réalisé dans le contexte de Grid'5000, une certaine quantité d'information est préalablement chargée (Ligne 1), par exemple des informations concernant la topologie de la grille (les frontaux des grappes, ...).

Ensuite, on demande 10 nœuds de la grappe *gdx* et 10 nœuds de la grappe *helios* (Ligne 2). Si la demande ne peut aboutir, le programme s'arrête là avec

un message d'erreur.

Les ressources sont ensuite vérifiées, et les nœuds jugés impropres sont supprimés des ressources disponibles (Ligne 3). Par vérifier, on entend soumettre les machines à un ensemble de programmes de test dont on possède les valeurs de référence, et qui permettent de déterminer le bon état d'une machine et de ses principaux services.

On effectue ensuite une tâche parallèle (*ptask*). Cette tâche sera exécutée en passant par la frontale par défaut (*$all.gateway*), et concernera tous les nœuds réservés sauf ceux jugés impropres. Toutes les machines exécutent donc la commande *date* (Ligne 4). Le résultat s'affiche à l'écran.

Une deuxième tâche parallèle est alors lancée, cette fois-ci en se limitant aux machines de *gdx* et en passant par la frontale de *gdx $all["gdx"].gateway*. La commande exécutée est donc *sleep 1* (Ligne 5). On récupère cette fois-ci l'identifiant de la tâche, ainsi que les informations concernant l'exécution de la commande sur les différentes machines.

Pour chaque commande exécutée, on affiche sa durée (Ligne 6). Puis, on affiche la moyenne des temps d'exécution de chaque commande dans la tâche (Ligne 7).

Ces deux dernières lignes permettent d'évaluer la précision de la prise de temps.

5.4.2 Évaluation d'un outil de diffusion de fichiers

Dans cette section, nous présentons le cas de l'évaluation d'un outil de diffusion de fichiers (*Kastafior*) vers plusieurs machines. *Kastafior* fonctionne sur le même principe que *mput* présenté sur la Figure 5.1, mais utilise une version ultérieure du lanceur parallèle. Cette version plus stable permet de pousser l'expérience sur un plus grand nombre de nœuds. La première phase est la mise en place d'une chaîne de connexions TCP entre les machines. Cette mise en place utilise le lanceur parallèle présenté précédemment en Section 5.3.3, lanceur qui sert de couche de lancement et d'auto-diffusion de l'outil. La second phase est la diffusion du fichier via la chaîne. Finalement, un mécanisme de *pipeline* garantit une bonne performance de la diffusion.

Le but de l'évaluation est de tester dans un environnement stable l'évolution du temps de diffusion en fonction du nombre de machines impliquées et de la taille du fichier à transférer. Le programme (Figure 5.11) décrit en langage dédié toutes les étapes nécessaires pour la réalisation complète de l'évaluation.

Comme dans le cas précédent, les premières lignes (1 à 3) réalisent la réservation des nœuds sur une grappe et effectuent le test par défaut sur l'ensemble des nœuds obtenus. La ligne 4 produit le tableau des tailles des

fichiers à transférer, de 2 kilo-octets à 1 giga-octets, avec une taille doublant entre chaque valeur. La ligne 5 exécute le script de création de fichiers (*create_file*) sur le premier nœud qui sera le nœud émetteur. La ligne suivante (6) est la boucle itérant sur l'ensemble des nœuds. Chaque itération rend un ensemble de ressources n dont la taille double à chaque itération. La ligne 7 permet d'écarter le cas où il y a un seul nœud pour l'évaluation. La ligne 8 permet la copie du fichier comportant la liste des nœuds, sur le nœud où sera lancée la commande de diffusion. Les 2 lignes suivantes (9 et 10) réalisent les itérations sur la taille du fichier et la répétition de la mesure pour obtenir une bonne précision statistique. La ligne 11 correspond au lancement de la commande de transfert sur le premier nœud de l'ensemble fourni. Finalement, la ligne 12 affiche le nombre de nœuds, la taille du fichier et le temps de diffusion.

La figure 5.12 présente les résultats obtenus sur la grappe *gdx* du site d'Orsay de la plate-forme Grid'5000. Cette grappe [1] est composée de nœuds bi-processeurs mono-cœur Opteron AMD à 2 Ghz et 2 Goctets de mémoire. Le réseau est un réseau Ethernet 1 Gbits. Le système d'exploitation utilisé est Linux version 2.6.22. Les résultats obtenus montrent un bon comportement de l'outil en fonction du nombre de nœuds (effet de *pipeline*) et une bonne stabilité sur les mesures (écart-type faible) excepté pour les petites tailles de fichier et un faible nombre de machines.

Cette expérience illustre l'utilisation du moteur *Expo* pour un cas réel d'évaluation. Le programme est court avec seulement 15 lignes, et ce, grâce à la disposition de l'ensemble du langage de script Ruby. En comparaison aux projets concurrents comme PluSH, qui utilisent un langage de description généralement de type XML, l'approche par langage dédié permet d'avoir une très grande souplesse lors des phases de mises au point en rendant les modifications et l'extension du programme simple à réaliser.

5.5 Conclusion

Dans ce chapitre nous avons présenté Expo, notre moteur de conduite d'expériences pour plate-forme à grande échelle. Expo est déjà fonctionnel et l'approche par langage dédié permet de décrire succinctement des expériences assez complexes.

Expo n'est pas parfait. Le langage dédié, bien que puissant, n'est pas encore arrivé à maturation. De même, le cœur d'Expo est écrit en Ruby ce qui peut poser des problèmes de performances. De plus il existe certainement

1. Pour plus de précision sur le matériel et le logiciel se reporter aux sections idoines du site http ://www.grid5000.fr

```
1  require "expo_g5k"
2  oargridsub :res => "gdx:nodes=128"
3  check $all
4  file_size = (1..20).collect do |i| 2**i; end
5  task $all.first, "./create_file 20"
6  $all.each_slice_power2 do |n|
7   next if n.resources.length == 1
8   copy n.nodefile, $all.first
9    file_size.each do |size|
10    10.times do |i|
11     id,r = task $all.first,"kastafior -f #{n.nodefile}
12 ____-- -i /tmp/#{size}K.dat -d /tmp/"
13     puts "#nb_nodes: #{n.resources.length}
14 ____file_size #{size} time #{r.duration}"
15    end
16   end
17 end
```

FIGURE 5.11 – Programme en langage dédié pour la réalisation de l'évaluation de l'outil de diffusion *Kastafior*

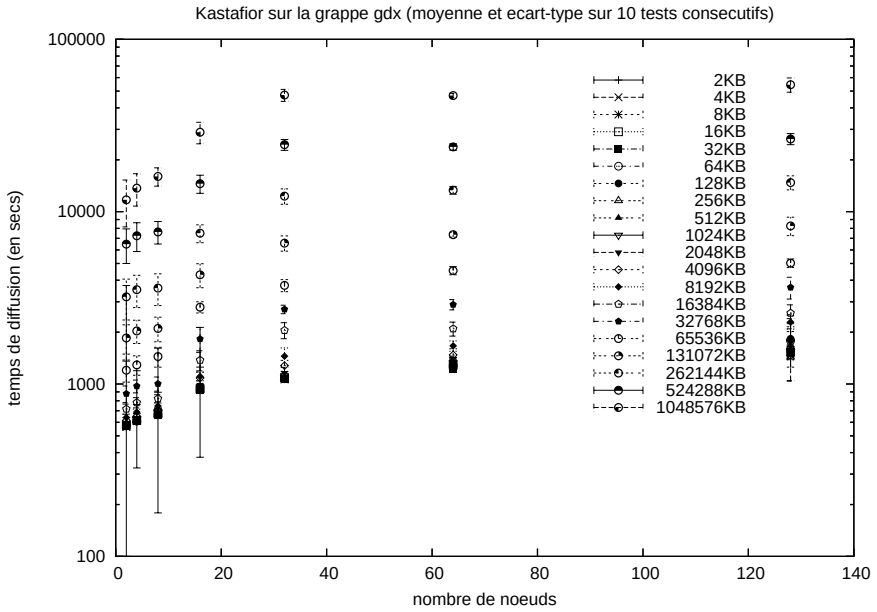


FIGURE 5.12 – Résultat du temps de diffusion de fichier avec *Kastafior* en fonction de la taille du fichier et du nombre de machines en réception. La moyenne (reportée en seconde) et l'écart-type sont obtenus sur 10 évaluations.

d'autres problèmes que nous n'avons pas identifiés notamment sur la gestion des ensembles de ressources et sur leur allocation aux différentes tâches.

Nous allons donc maintenant utiliser Expo pour évaluer le comportement de PaSTeL dans de nombreuses configurations différentes. Cela permettra de valider encore Expo tout en permettant un gain de temps considérable sur la conduite des expériences et l'analyse des résultats. Ce gain de temps augmentera le nombre d'expériences qui pourront être réalisées et améliorera donc la qualité même des résultats.

Chapitre 6

Expériences : évaluation de PaSTeL à l'aide d'Expo

Nous avons développé deux outils dédiés à la conduite d'expérience. Bien qu'ils aient été conçus indépendamment l'un de l'autre, ils vont être utilisés conjointement pour conduire une campagne d'expériences. Cette campagne a pour objectif d'évaluer l'impact de nombreux paramètres sur le comportement de PaSTeL. Elle a également pour but d'étudier l'adéquation entre PaSTeL et Expo et de souligner les éventuels problèmes rencontrés ainsi que les solutions qui ont été apportées. Cela permet de valider plus avant les outils conçus et notre démarche.

Pour cela, nous avons décidé d'étudier simultanément l'influence de nombreux paramètres sur le comportement de PaSTeL. Cela permet à la fois d'obtenir de nombreuses informations sur le comportement de PaSTeL tout en exploitant les capacités d'Expo.

Dans ce chapitre nous allons tout d'abord décrire précisément les objectifs de notre campagne, ainsi que le plan expérimental qui a été conçu. Le déroulement des expériences est ensuite étudié. Dans un troisième temps, les résultats expérimentaux sont analysés et présentés. Finalement, les conclusions de la campagne, ainsi que les enseignements qui en ont été tirés sont discutés.

6.1 Objectifs expérimentaux

Le but final de cette campagne d'expérience est d'étudier finement l'influence de certains paramètres sur le comportement de PaSTeL. La parallélisation des expériences offerte par une plate-forme comme Grid'5000 nous permet d'envisager des expériences comportant un très grand nombre de me-

sures. Cependant, il ne nous a pas été possible d'étudier l'influence de tous les paramètres en raison de contraintes temporelles. Nous en avons donc sélectionné un certain nombre. Cette sélection a dirigé la conception d'un plan expérimental qui nous a paru adapté à la fois à nos critères expérimentaux et aux limites de nos outils.

Il faut rappeler que, si un paramètre n'a pas été sélectionné ici, les expériences ne sont pas à refaire. Il est tout à fait possible de les enrichir de nouvelles mesures pour étudier l'influence de nouveaux paramètres.

6.1.1 Choix des paramètres

Le choix des paramètres qui s'offrent à l'expérimentateur voulant étudier des algorithmes parallèles est particulièrement vaste. Nous allons présenter ici ceux qui ont été retenus, en justifiant ces choix. Nous allons également citer quelques paramètres qui ont été écartés.

Paramètres retenus

Types de plates-formes Il nous a paru important d'étudier l'influence de deux plates-formes différentes. En effet les expériences conduites dans le Chapitre 4 montrent certaines disparités de comportement entre les deux plates-formes étudiées alors. Cependant, ces plates-formes étaient très différentes l'une de l'autre. Un ordinateur portable et un serveur octo-processeur ne sont pas conçus avec les mêmes objectifs de performance ou d'utilisation.

Nous avons donc choisi d'étudier deux plates-formes distinctes dans leur construction mais similaires dans leur destination. Les machines choisies sont présentées sur la Figure 6.1. Pour plus de simplicité elles sont baptisées Bordereau et Genepi, qui est le nom des grappes de calcul où ces machines sont utilisées.

Comme le montre le tableau, les machines possèdent un certain nombre de similitudes. Tout d'abord ce sont des serveurs bi-processeurs. De plus, la mémoire utilisée possède des caractéristiques similaires. Par contre, Bordereau est une machine NUMA alors que Genepi est une machine UMA. Le nombre de cœurs est également différent entre les 2 plates-formes, Genepi possédant des processeurs quadri-cœurs alors que Bordereau possède des processeurs bi-cœurs. La philosophie de conception des machines est également très différente : Bordereau possède un bus mémoire intégré au processeur, et peut donc se contenter d'un cache L2 plus petit. Ce n'est pas le cas de Genepi qui, pour pallier une latence d'accès supérieure à la mémoire, utilise un cache plus gros.

Caractéristiques	Bordereau	Genepi
Modèle	IBM System x3455	Bull R422-E1
Processeurs	2 * AMD Opteron 2455	2 * Intel Xeon E5420 QC
Nombre de cœurs par processeur	2	4
Fréquence	2.6 GHz	2.5 GHz
Cache L2 par processeur	2 MB	12 MB
Répartition cache L2	1 MB par cœur	6 MB partagé entre 2 cœurs
Mémoire	4 GB	8 GB
Bus mémoire	intégré	externe
Modèle de mémoire	DDR2 667 MHz	DDR2 667 MHz
Architecture	NUMA	UMA
Système	debian lenny	debian etch-and-a-half
Noyau	2.6.26	2.6.24
Compilateur	g++ 4.2.4	g++ 4.3.2

FIGURE 6.1 – Matériel utilisé pour la campagne d'expériences

Le système installé est légèrement différent, de même que les compilateurs. Il faut cependant prendre en compte le fait que PaSTeL s'abstrait du système d'exploitation en liant les threads aux processeurs et en utilisant des verrous tournants. De même, nous n'avons pas mesuré de différences de comportement de PaSTeL ou de la STL en utilisant l'une ou l'autre version du compilateur.

Ces différences devraient permettre d'observer des différences de comportement de PaSTeL dues à l'architecture de ces machines.

Algorithmes étudiés Nous avons conservé les 3 algorithmes présentés précédemment pour ces expériences. Leur comportement est déjà connu sur les plates formes *serveur* et *portable*, ce qui peut permettre de découvrir d'éventuelles disparités de fonctionnement. Il faut également rappeler que ces algorithmes sont représentatifs de nombreux algorithmes de la STL. Ces trois algorithmes sont : *min_ element*, *merge* et *stable_ sort*.

Nombre de thread utilisés Le passage à l'échelle d'une application est un critère important car il conditionne en partie l'efficacité de l'algorithme au regard des ressources qui lui sont allouées. Chaque algorithme sera donc

testé en utilisant 1 thread en plus du thread principal, puis 2 threads supplémentaires et, ainsi de suite jusqu'au nombre de cœurs disponibles.

Taille de données Il est souhaitable d'étudier le comportement des algorithmes sur une large gamme de tailles. Cependant, le comportement des algorithmes est plus variable sur les données de petite taille que sur celles de grande taille. Les tailles des données ont donc été réparties de manière logarithmique sur l'intervalle 0 à 2^{27} (134 millions) d'éléments. Cet intervalle a été découpé en 100 points. Les dix premiers points ont été ignorés.

La formule permettant d'obtenir le i-ème point est $2^{\frac{27.0*i}{100.0}}$.

Granularité La granularité est un paramètre important de PaSTeL, car c'est lui qui détermine le surcoût du moteur de vol de travail. Une granularité faible offre une meilleur réactivité et de meilleures performances sur les données de petite taille, alors qu'elle est pénalisante sur des données de grande taille. Les granularité retenues pour l'expérience sont 20, 40, 60, 80, 100, 200, 300, 400, 500, 600, 800 et 1000.

6.1.2 Paramètres écartés

Type des données Le type retenu pour l'expérience est le type entier (*int*). Comme expliqué précédemment, il constitue un pire cas pour PaSTeL de par le temps de traitement individuel très court et du stress qu'il exerce sur le bus mémoire.

Il serait instructif, dans une extension de l'expérience, d'étudier le comportement de PaSTeL sur des types de données au traitement plus long.

Type de synchronisation Le type de synchronisation retenu ici est la synchronisation par verrous tournants (*spin locks*). Ces verrous offrent de meilleures performances et une meilleure reproductibilité des expériences. Cela permet d'avoir une plus grande confiance sur les effets observés. Une extension de l'expérience pourrait être la comparaison avec les verrous bloquants (*mutex*).

Évaluation de performance Le mécanisme d'évaluation de performance de PaSTeL n'a pas été étudié au cours de cette expérience. Il le sera dans une campagne ultérieure.

Paramètres	Valeurs	Nombre	type
Plate-formes	Genepi, Bordereau	2	compilation
Algorithmes	*min_ element*, *merge stable_ sort*	3	compilation
Nombre de threads	Genepi : 1-7 Bordereau : 1-3	7 3	compilation
Taille des données	6-134217728	90	exécution
Granularité	20-1000	12	compilation
Nombre de répétitions	20		exécution
Nombre de mesures	Genepi : 453600 Bordereau : 194400		
Nombre de binaires	Genepi : 252 Bordereau : 108		

FIGURE 6.2 – Plan expérimental : récapitulatif

Influence de l'architecture De nombreux détails architecturaux n'ont pas été la cible de l'expérience non plus. Par exemple, l'influence du placement des threads n'a pas été étudiée. Les threads ont donc été fixés sur les processeurs dans l'ordre de la numérotation du noyau.

De même, nous n'avons pas effectué l'expérience sur deux machines presque identiques dont un seul paramètre, comme la fréquence du processeur, aurait varié.

6.1.3 Plan expérimental

Le plan expérimental reprend les paramètres évoqués dans la section précédente. Ces paramètres sont récapitulés sur la Figure 6.2. Dans ce tableau est aussi explicité le type du paramètre, c'est à dire s'il doit être spécifié à la compilation ou à l'exécution. Les paramètres spécifiés à la compilation conditionnent le nombre de binaires qui ont été employés au cours de la campagne d'expérience. Chaque manipulation a été répétée 20 fois pour pouvoir calculer des intervalles de confiance.

Ce plan expérimental nécessite donc d'effectuer environ 650 000 mesures dont certaines peuvent prendre plusieurs minutes. Dans la section suivante nous allons voir comment ce plan a été mis en œuvre et comment se sont déroulées les expériences.

6.2 Conduite expérimentale

La conduite de la campagne d'expériences s'est faite au moyen d'Expo. Expo n'avait jamais été utilisé pour conduire des expériences de cette taille, et il existait de nombreuses incertitudes quand à son comportement.

La conduite de la campagne se déroule en 3 phases. Une première phase de compilation des différents binaires sur chacune des plates-formes. Ensuite la conduite des mesures à l'aide d'Expo et l'archivage des résultats et finalement le pré-traitement des résultats pour permettre leur interprétation.

6.2.1 Compilation

L'installation de PaSTeL sur Genepi et Bordereau a nécessité un certain nombre d'étape préalables. Il est notamment nécessaire d'installer en amont la bibliothèque libconfig [Lin] et la bibliothèque libnuma [Kle].

La phase de compilation consiste à générer l'ensemble des binaires nécessaires à l'expérience. Pour faciliter l'interprétation des résultats, les binaires sont nommés de manière standard. Les noms correspondent au masque suivant : algorithme + *test* + bibliothèque + type de données + granularité + nombre de threads employés + méthode de synchronisation + utilisation de l'évaluation de performances.

Lors des mesures, le résultat est stocké avec la ligne de commande qui a été employée. Il est donc possible de récupérer à la fois les paramètres spécifiés à la compilation et les paramètres spécifiés à l'exécution. Le travail d'analyse est donc simplifié.

6.2.2 Script Expo

Le script présenté sur la Figure 6.3 est celui qui a été utilisé au cours de la campagne d'expérience. Il effectue à la fois la récupération des ressources, le lancement des manipulations et un archivage sur disque des résultats obtenus. La première étape consiste à se connecter à une réservation *oargrid* au moyen de son numéro passé en paramètre (Ligne 5). Ensuite, la liste des binaires à utiliser pendant l'expérience est obtenue (Ligne 7 à 11). Les Lignes 13 à 17 génèrent les différentes tailles de données qui seront employées. L'archivage des données au format YAML est ensuite traité (Lignes 19 à 25). Cet archivage se fait à la sortie du programme. Les différentes commandes sont ensuite exécutées et les résultats sauvegardés en mémoire (Ligne 27 à 32). Pour paralléliser le déroulement de l'expérience nous avons ici décidé d'effectuer les répétitions sur des machines différentes. Chaque mesure est donc répétée sur chaque machine de la réservation.

```
1  require 'yaml'
2  require 'expo_g5k'
3  require 'date'
4
5  oargridconnect ARGV[0]
6
7  binaries = Array::new
8  task = Task::new("ls pastel/pastel-stable/bin",$all.first)
9  number, result = task.execute
10 binaries = result.first["stdout"].split
11 binaries.sort!
12
13 sizes = []
14 for i in 10..100 do
15   sizes.push( ( 2 ** ( 27.0 * i.fdiv 100.0 ) ).to_int )
16 end
17 sizes.uniq!
18
19 file = File::new("Experiment-#{$client.experiment_number.\
20 experiment_number}-#{DateTime::now}","w")
21 results = []
22 at_exit {
23   results.each { |result| file.puts(YAML::dump(result)) }
24   file.close
25 }
26
27 binaries.each { |binary|
28   sizes.each { |size|
29     task = Task::new("#{binary} #{size} 20", $all.uniq )
30     results.push(task.execute)
31   }
32 }
```

FIGURE 6.3 – Le script Expo utilisé au cours des expériences

Il faut noter que l'archivage pourrait être fait au fur et à mesure, mais cela imposerait d'utiliser un thread pour gérer les sorties. En effet, le temps de traitement des données pour les sauvegarder au format YAML n'est pas négligeable, et un archivage au fur et à mesure ralentit considérablement le déroulement de l'expérience.

6.2.3 Format de sortie

Le format de sortie des expériences est un ensemble d'enregistrements en YAML. Chaque enregistrement correspond à une manipulation. Le format retenu conserve une grande quantité d'informations sur chaque manipulation. Ce format est présenté sur la Figure 6.4.

Le traitement des données dans ce format est particulièrement facilité par l'utilisation de Ruby qui le gère nativement. Le script présenté sur la Figure 6.5 permet de réunir les différentes répétitions d'une mesure au sein d'un même tableau. Le résultat de cette opération est lui-même exporté au format YAML pour subir de futurs traitements. Ces traitements sont par exemple le calcul de la moyenne et des intervalles de confiance sur les résultats.

6.2.4 Déroulement de l'expérience

La conduite de l'expérience s'est déroulée sur plusieurs créneaux. En effet, le nombre important de répétitions ne permettait pas d'effectuer toutes les mesures en une fois. Le script et le serveur Expo ont été exécutés sur la machine frontale du site de Grenoble. Les expériences se sont déroulées sans incident notable.

Le script expérimental a cependant dû être modifié. A l'origine, le script sauvegardait en YAML l'intégralité de la structure *results* (cf Figure 6.3, Ligne 21). Cette approche est très coûteuse en mémoire et l'espace mémoire autorisé par personne connectée sur la machine frontale de Grenoble est de 1 GO. Le script a donc été terminé par le système et l'archivage n'avait pas été effectué. Cependant, les résultats des mesures ont pu être récupérés. En effet, le serveur Expo conserve les informations relatives à chaque commande. Cette mésaventure illustre bien l'intérêt qu'il y a à découpler la partie de génération et d'interprétation des commandes de la partie d'exécution des commandes.

6.3 Analyse des résultats

Dans cette section nous allons présenter une partie des résultats obtenus au cours de la campagne d'expériences. Tout d'abord, nous allons étudier le

```
---
- 2493
- !seq:ExpoResult
  - !map:TaskResult
    stdout: |
      371446

    status: "0"
    host_name: bordereau-29.bordeaux.grid5000.fr
    stderr: ""
    rank: "11"
    command_line: |
      merge_test_pastel_int_40_1_SPIN_LOCK_false 20311 20

    start_time: 2009-04-08 22:59:33.438950 +02:00
    end_time: 2009-04-08 22:59:33.480720 +02:00
  - !map:TaskResult
    stdout: |
      375518

    status: "0"
    host_name: bordereau-20.bordeaux.grid5000.fr
    stderr: ""
    rank: "6"
    command_line: |
      merge_test_pastel_int_40_1_SPIN_LOCK_false 20311 20

    start_time: 2009-04-08 22:59:33.483690 +02:00
    end_time: 2009-04-08 22:59:33.527160 +02:00
```

FIGURE 6.4 – Extrait d'un champ YAML décrivant une manipulation effectuée en parallèle sur 2 machines. Les dates de lancement et de terminaison de chaque commande sont enregistrées, ainsi que les sorties standard et d'erreur. Le nom des machines qui ont exécuté les commandes est également conservé.

```
1  require 'yaml'
2  require 'expolib'
3
4  raise 'missing file name' if not ARGV[0]
5
6  file = File::new(ARGV[0],"r")
7  results = Hash::new { |hash,key| hash[key] = [] }
8  yp = YAML::load_documents(file) { |record|
9    command_number, command_result = record
10   command_result.each { |task|
11     results[task["command_line"]].push(task["stdout"].to_i)
12   }
13 }
14 puts YAML::dump(results)
15 file.close
```

FIGURE 6.5 – Le script permettant l'agrégation des résultats au format YAML

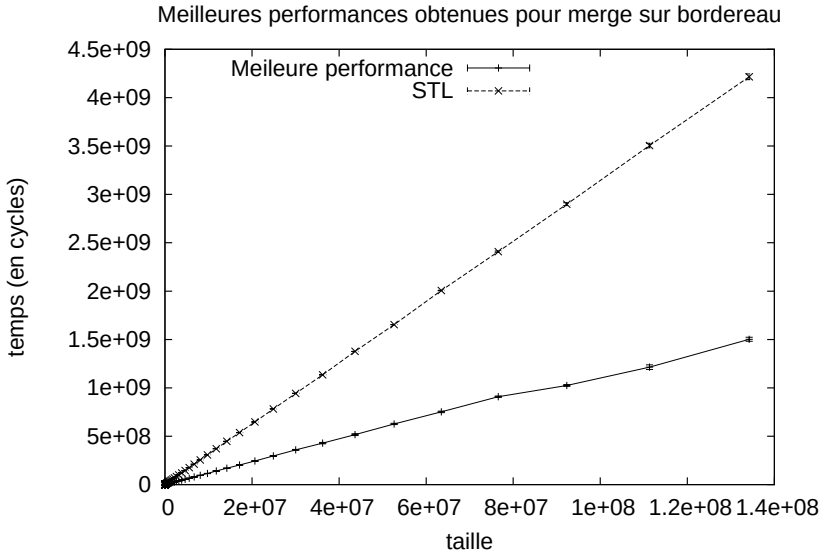
comportement général de PaSTeL sur les différentes plates-formes. Ensuite, nous comparerons l'efficacité des algorithmes de PaSTeL sur Bordereau et Genepi. Enfin nous étudierons l'influence du grain sur les performances des algorithmes.

6.3.1 Comportement général sur Bordereau

Les machines de la grappe de calcul Bordereau sont des serveurs NUMA utilisant 2 processeurs bi-cœurs. Dans cette section, l'influence de la granularité n'est pas présentée. C'est donc la meilleur performance atteinte pour chaque taille qui est utilisée sur les courbes suivantes.

Les courbes présentées ici (Figure 6.6 et en annexes (Figure 7.1 à 7.3) indiquent le temps d'exécution (exprimé en cycles processeur) en fonction de la taille des données à traiter. Elles permettent d'obtenir un aperçu global du comportement des algorithmes de PaSTeL. Le temps d'exécution séquentiel est également présenté. Les intervalles de confiance à 95% sont reportés sur les figures.

La Figure 6.6 présente les performances de l'algorithme *merge*. Deux



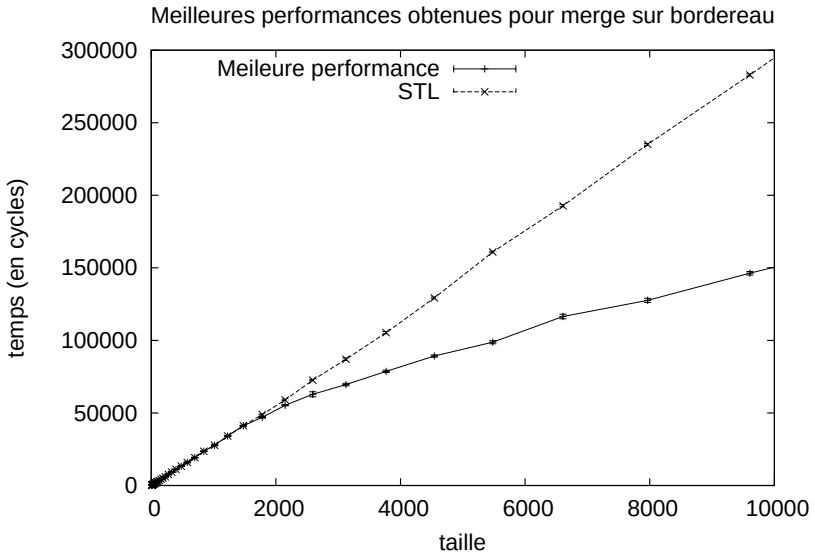


FIGURE 6.6 – Meilleures performances obtenues sur Bordereau avec l'algorithme *merge*

Algorithme	*min_ element*	*merge*	*stable_ sort*
Accélération finale pour 3 threads	2.82	2.81	2.87
Accélération finale pour 2 threads	2.46	2.42	2.57
Accélération finale pour 1 thread	1.90	1.71	1.86
Accélération à partir de (éléments)	7000	2000	1500
coût de l'algorithme séquentiel (cycle/élément) pour 134 M d'éléments	6.5	31.4	449

FIGURE 6.7 – Performances des différents algorithmes sur Bordereau

échelles de tailles sont données : entre 0 et 140 millions d'éléments et entre 0 et 10000 éléments. Les intervalles de confiance sont à peine visibles, ce qui indique une bonne stabilité des mesures. Sur les tableaux de grande taille, le comportement de PaSTeL est globalement linéaire. L'accélération atteinte par PaSTeL pour 134 millions d'éléments est de $2,8$. PaSTeL commence à obtenir des accélérations à partir de 2000 éléments.

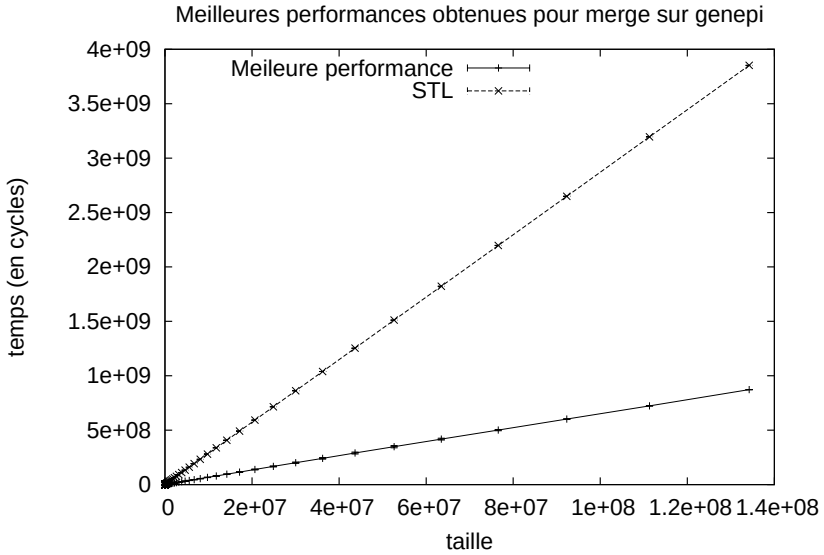
La comportements global des autres algorithmes est similaire. Cependant, les valeurs sont différentes. Ces valeurs sont présentées dans le Tableau 6.7. Les accélérations pour des tailles de tableaux de 134 millions d'éléments sont également présentées en fonction du nombre de threads utilisés en plus du thread principal.

6.3.2 Comportement général sur Genepi

Les machines de la grappe de calcul Genepi sont des serveurs UMA utilisant 2 processeurs quadri-cœurs. Dans cette section l'influence de la granularité n'est pas présentée. C'est donc la meilleure performance atteinte pour chaque taille qui est utilisée sur les courbes suivantes.

Les courbes présentées ici (Figure 6.8) et en annexes (Figure 7.4 à 7.6) sont de même nature que celles présentées pour la plate-forme bordereau.

La comportements global des autres algorithmes est similaire. Cependant, les valeurs sont différentes. Ces valeurs sont présentées dans le Tableau 6.9. Les accélérations pour des tailles de tableaux de 134 millions d'éléments sont également présentées en fonction du nombre de threads utilisés.



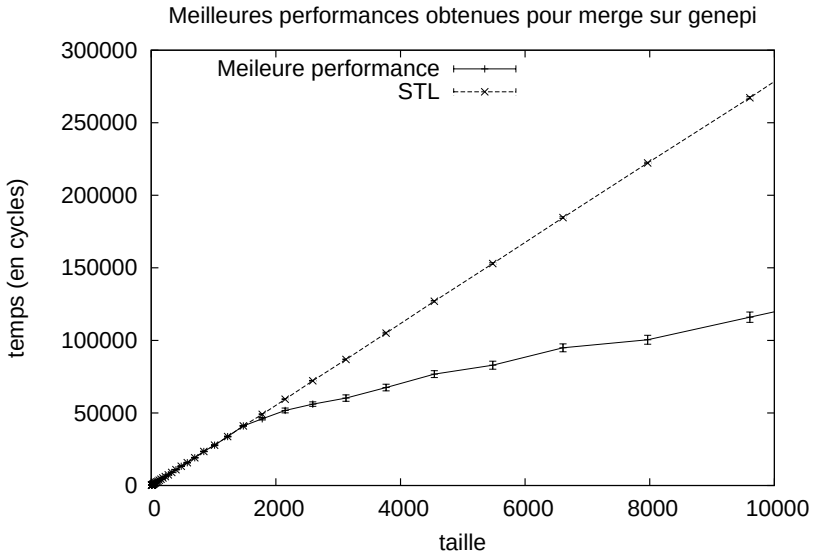


FIGURE 6.8 – Meilleures performances obtenues sur Genepi avec l'algorithme *merge*

Algorithme	*min_ element* g++-4.3	*min_ element* g++-4.1	*merge*	*stable_ sort*
Accélération finale pour 7 threads	4.30	4.28	4.42	5.17
Accélération finale pour 6 threads	4.27	4.23	4.35	4.94
Accélération finale pour 5 threads	4.23	4.17	4.29	4.68
Accélération finale pour 4 threads	4.08	3.95	3.96	4.15
Accélération finale pour 3 threads	3.93	3.74	3.62	3.57
Accélération finale pour 2 threads	3.55	2.86	2.79	2.71
Accélération finale pour 1 thread	3.24	1.99	1.92	1.83
Accélération à partir de (éléments)	4000	6000	2000	1500
coût de l'algorithme séquentiel (cycle/élément) pour 134 M d'éléments	6.75	6.75	28.7	406

FIGURE 6.9 – Performances des différents algorithmes sur Genepi

Les mesures nécessaires au calcul des données de *min_ element* ont été refaites en utilisant des binaires compilés avec une autre version du compilateur. En effet, lorsque l'on utilise un faible nombre de threads, PaSTeL présente un comportement suspect. L'utilisation d'un thread supplémentaire permet d'améliorer les performances d'un facteur $3,2$. Ce n'est pas le cas lorsque le binaire est compilé avec une version antérieure de g++. En revanche, lorsque le nombre de threads devient grand, cette différence s'estompe. Cela laisse supposer qu'un autre facteur limitant entre en jeu, par exemple de la contention mémoire les algorithme utilisant des données au temps de traitement très court et donc très intensif en bande passante mémoire.

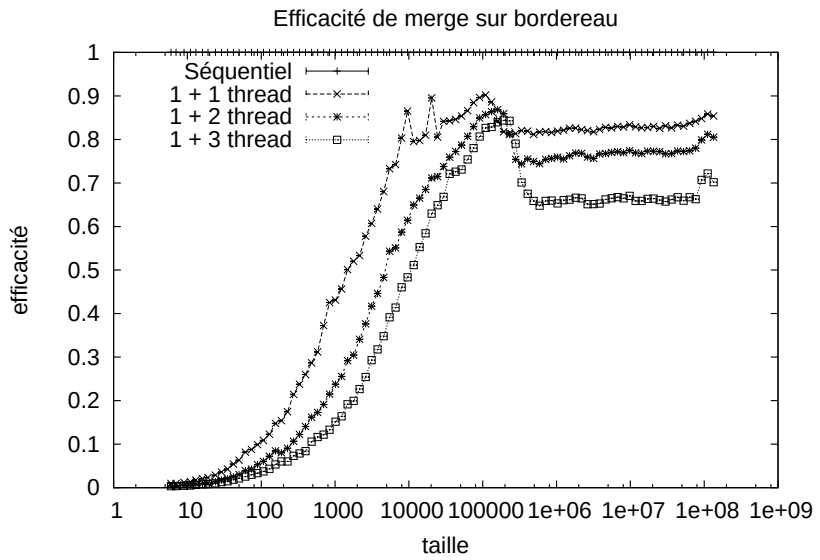
6.3.3 Efficacités comparées

Dans cette section, l'efficacité est définie comme le ratio entre l'accélération et le nombre de ressources employées. Donc, si l'on utilise 2 threads en plus du thread principal : $E = \frac{T_{seq}}{3T_{par}}$, par convention, l'algorithme séquentiel possède une efficacité de 1. Cette valeur représente la fraction des ressources qui est efficacement utilisée. Une efficacité de $0,75$ en utilisant 4 cœurs indique à la fois une accélération de 3 et que la puissance de calcul d'un cœur a été perdue.

La Figure 6.10 présente l'efficacité de l'algorithme *merge* sur Bordereau et Genepi. Comme l'influence du grain n'est pas étudiée ici, c'est la meilleure performance pour chaque couple (nombre de threads et taille des données) qui est retenue. Sur ces graphiques, il apparaît clairement que chaque thread est plus coûteux que le précédent. PaSTeL, sur des problèmes de ce type, ne passe donc pas à l'échelle.

Il apparaît également que, sur cet algorithme, la parallélisation est moins efficace sur Bordereau que sur Genepi. Sur Genepi, jusqu'à 3 threads, l'efficacité est supérieure à $0,9$ à partir de 100000 éléments environ. Sur Bordereau, avec 3 threads, elle ne dépasse que rarement $0,7$. Pour cet algorithme, PaSTeL passe donc mieux à l'échelle sur Genepi que sur Bordereau.

Les graphiques font apparaître un effet de cache assez marqué, les mesures étant effectuées à cache chaud. Sur Bordereau, cet effet induit un pic de performances aux alentours de 130000 éléments. Cette taille correspond à une utilisation mémoire de 520000 éléments (2 tableaux en entrée de 130000 éléments chacun et un tableau en sortie de 260000 éléments). Les éléments étant de type *int* utilisant 4 octets en mémoire, cela correspond à environ 2080000 octets. Ce chiffre est très proche de la taille du cache qui est de 2Mo. Pour Genepi, ce pic se situe aux alentours de 600000 éléments. Le même calcul conduit à une utilisation mémoire d'environ 9600000 octets. Cette taille est inférieure à la taille du cache qui est de 12Mo.



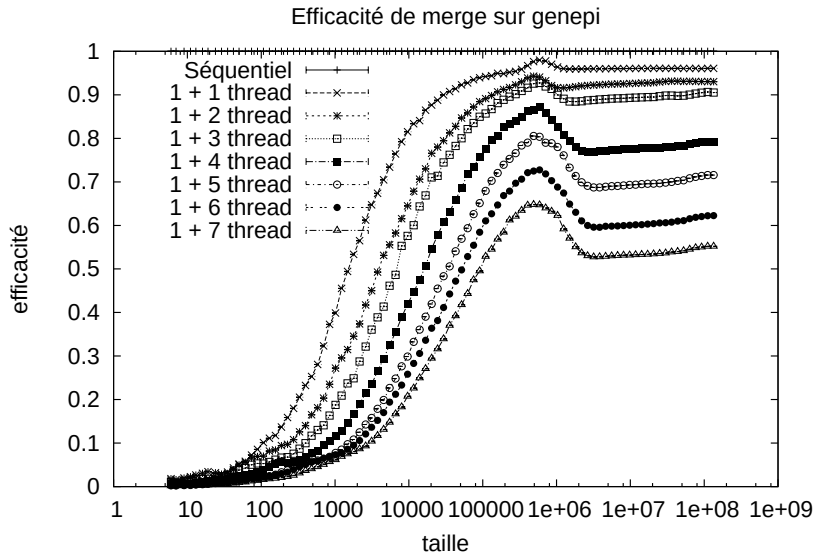


FIGURE 6.10 – Efficacité de l'algorithme *merge* sur Bordereau et Genepi en fonction du nombre de threads utilisés

L'efficacité de l'algorithme *min_ element* présente un comportement similaire. La figure est présentée en annexes 7.9. Elle laisse également bien apparaître le problème de performances rencontré précédemment.

Par contre, l'algorithme *stable_ sort* présente un comportement différent. L'efficacité de cet algorithme est présenté sur la Figure 6.11. L'effet de cache qui était fortement présent n'apparaît plus que sur Bordereau et n'est vraiment visible qu'en utilisant 3 threads en plus du thread principal.

Cependant, ici aussi, Genepi passe mieux à l'échelle que Bordereau pour un nombre de threads allant de 1 à 3. Pour le grandes tailles de données, PaSTeL passe même linéairement à l'échelle jusqu'à 3 threads.

6.3.4 Influence du grain

Dans cette section, l'influence du grain est étudiée comme une dégradation de performance par rapport à la meilleure performance obtenue. La dégradation due à l'utilisation d'un grain de 80 s'exprime donc comme $D = fracT_{80}T_{meilleur}$.

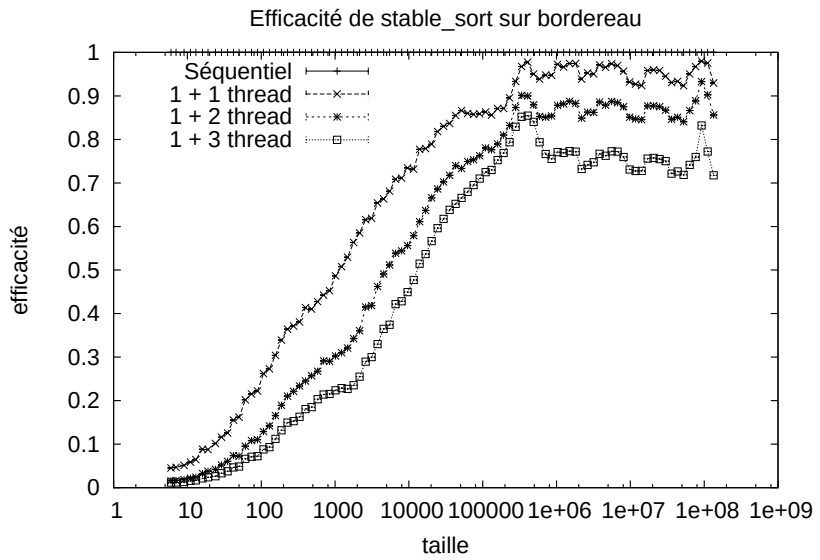
Sur la Figure 6.12 la dégradation de performances en fonction du grain de l'algorithme *merge* sur Genepi est reportée. Le premier graphique présente les résultats en utilisant 1 thread en plus du thread principal, alors que le second présente l'utilisation de 3 threads. Pour plus de lisibilité, les courbes sont présentées sans intervalles de confiance. De même, tous les grains disponibles ne sont pas présentés.

Le premier graphique montre que, pour de petites tailles, un grain de 200 obtient les meilleures performances. Ensuite, c'est le grain de 400 qui présente le plus d'intérêt. Finalement, pour les grandes tailles, le grain de 800 devient plus intéressant. Ce phénomène provient de deux effets antagonistes. Le premier est le surcoût lié à la vérification du vol. Plus le grain est petit, plus on vérifie souvent si un vol doit avoir lieu. Si l'on note S_b le surcôut lié à la boucle et s_b le surcoût d'une vérification, N le nombre d'éléments à traiter et G le grain, alors :

$$S_b = \frac{s_b N}{G} \tag{6.1}$$

Le deuxième effet est le temps d'attente du voleur lors d'un vol. En moyenne, le voleur attend pendant la moitié du temps d'exécution d'une boucle. Si l'on note T_b le temps d'exécution d'une boucle, et T_{seq} le temps d'exécution séquentiel, alors $T_b = \frac{T_{seq}G}{N}$. Si on note T_{vol} le temps d'attente lors du vol alors $T_{vol} \approx \frac{T_{seq}G}{2N}$. Le surcôut lié au vol, noté S_v, est donc de :

$$S_v = n_{vol}T_{vol} \approx \frac{n_{vol}T_{seq}G}{2N} \tag{6.2}$$



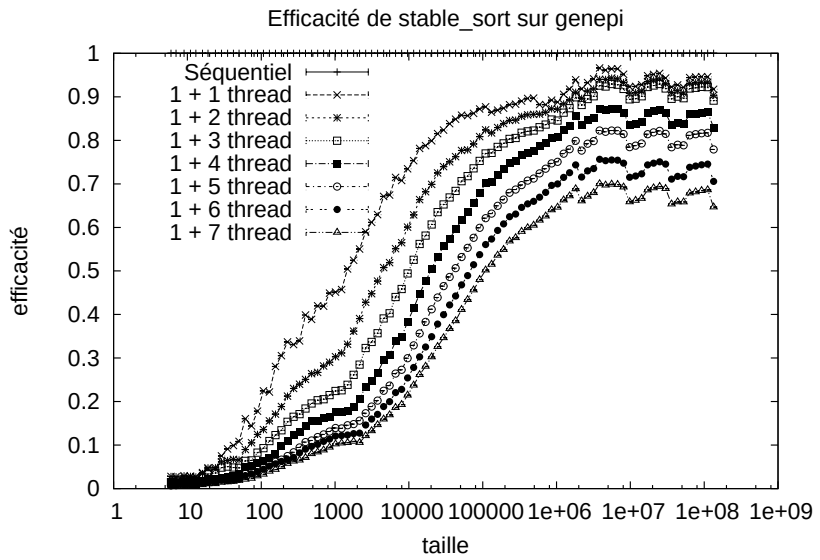
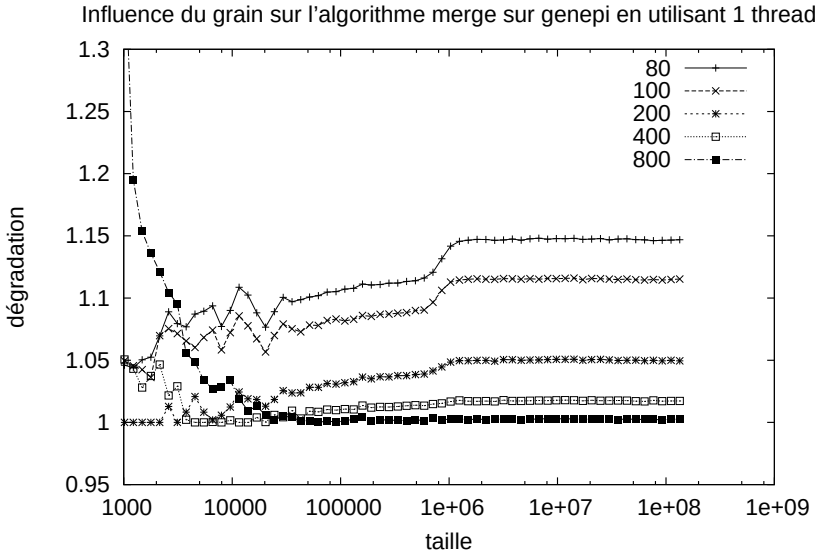


FIGURE 6.11 – Efficacité de l'algorithme *stable_ sort* sur Bordereau et Genepi en fonction du nombre de threads utilisés



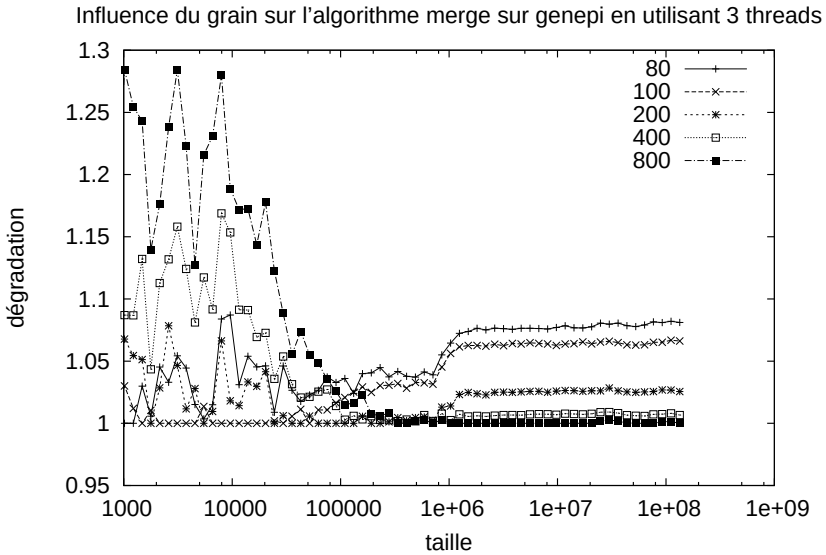


FIGURE 6.12 – Influence du grain sur Genepi. Comparaison entre 1 et 3 threads sur l'algorithme *merge*

Dans [FLR98] il est montré que le nombre de vols est proportionnel au nombre de threads. Le surcoût lié au nombre de vols s'exprime par $S_v \approx c\frac{PT_{seq}G}{2N}$ où P est le nombre de threads qui participent au travail et c une constante. Le surcoût dû au grain peut donc s'exprimer comme :

$$S = S_v + S_b \approx c\frac{PT_{seq}G}{2N} + \frac{s_b N}{G} \tag{6.3}$$

Le deuxième graphique de la Figure 6.12 permet de vérifier ces propriétés. En effet, sur de petites tailles, les meilleures performances sont obtenues avec un grain de 100 éléments, puis avec un grain de 200 éléments, et finalement, pour les grandes tailles, avec un grain de 800 éléments.

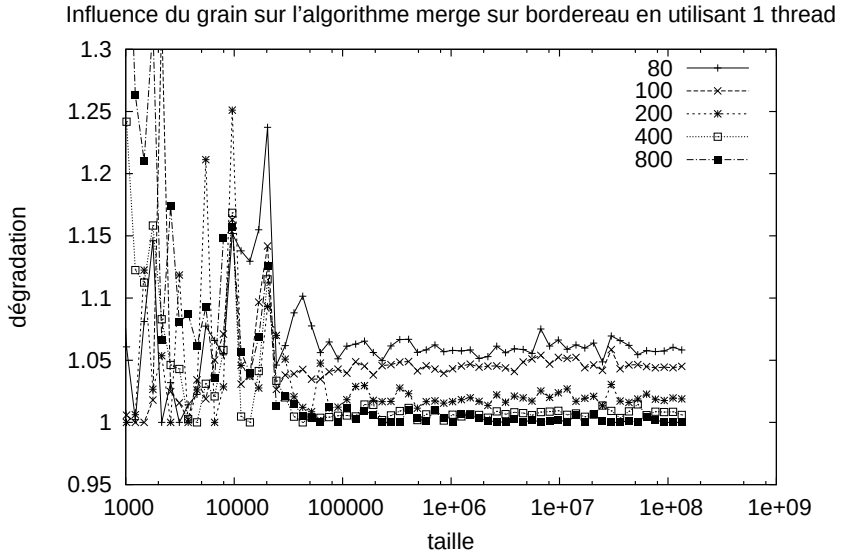
Le second graphique fait également apparaître que le surcoût quand le nombre d'éléments devient grand diminue avec avec le nombre de threads. Le surcoût lié à l'emploi d'un grain de 80 est de $1,15$ en utilisant 1 thread et de $1,08$ en utilisant 3 threads. Il semble donc qu'un autre surcoût, dépendant du nombre de threads amortit ce dernier. Le *mauvais* passage à l'échelle observé dans la section précédente plaide également en ce sens.

Les mêmes phénomènes peuvent-être observés sur Bordereau, Figure 6.13. Cependant, les résultats sont beaucoup moins clairs que sur Genepi, particulièrement sur les données de petite taille. Sur les données de grande taille, le surcoût dû au grain est moindre que sur Genepi. Cependant, il faut se souvenir que PaSTeL passe moins bien à l'échelle sur Bordereau que sur Genepi. Il existe peut-être également d'autres facteurs limitant que le grain employé par PaSTeL.

6.4 Discussions et conclusions

Les expériences conduites dans ce chapitre ont permis de montrer la viabilité d'Expo dans la conduite d'expériences de grandes tailles. Elles ont également mis en avant un certain nombre de problèmes qui ne sont que partiellement solutionnés, notamment la taille des données en mémoire et les performances de Ruby. Néanmoins, Expo a montré sa souplesse dans la conduite des expériences et sa résistance aux pannes. De futures évolutions de Taktuk permettront peut-être d'améliorer les performances du lancement des tâches et donc de réduire la durée d'une campagne similaire.

PaSTeL, pour sa part, permet d'observer avec beaucoup de précision l'influence de nombreux paramètres sur son moteur de vol de travail. Cependant, cette campagne soulève plus de questions qu'elle n'en résout. Par exemple, elle n'explique pas pourquoi Genepi et Bordereau ont parfois des comportements aussi éloignés. De même, le mauvais passage à l'échelle sur ce type



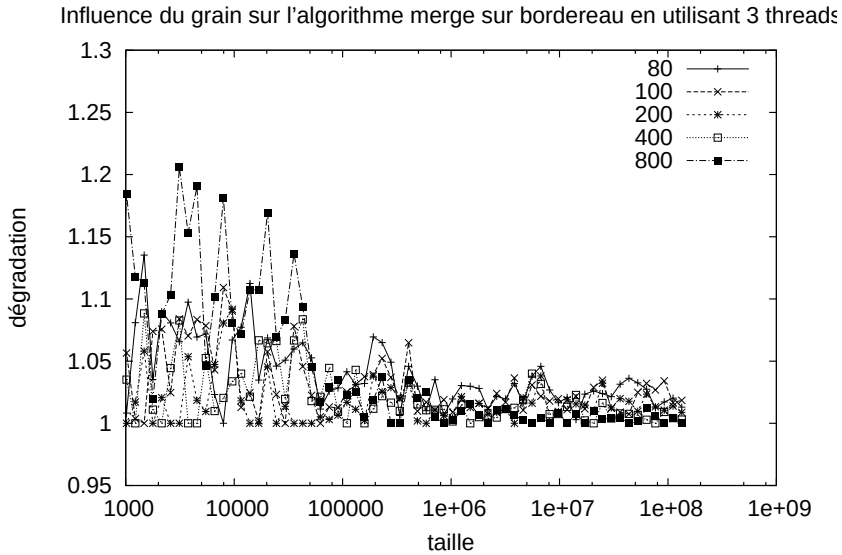


FIGURE 6.13 – Influence du grain sur Bordereau. Comparaison entre 1 et 3 threads sur l'algorithme *merge*

de données, au temps de traitement très court, semble indiquer un problème de contention mémoire. Mais il faudrait pousser les expériences plus avant pour pourvoir en obtenir confirmation. Il faudrait également vérifier le bon passage à l'échelle de PaSTeL lorsque le temps de traitement des données devient plus long, c'est à dire lorsque l'on ménage la bande passante mémoire. Des études préliminaires ont montré un bon passage à l'échelle, mais une vraie campagne d'expériences permettrait de s'en assurer.

Chapitre 7

Conclusion et perspectives

Dans cette thèse, deux approches différentes ont été proposées pour résoudre une partie des problèmes expérimentaux posés par les nouvelles architectures. Les méthodologies et les outils proposés, bien que très éloignés, sont complémentaires.

Dans le contexte des grilles légères, nous avons montré la nécessité d'utiliser une méthodologie rigoureuse dans la conduite des expériences. En effet, du fait de la taille de ces plates-formes, leur état est difficile à déterminer. Cet état peut cependant avoir un impact non négligeable sur les performances et sur le comportement des applications testées. Ces travaux ont notamment permis de détecter un certain nombre de défauts matériels et logiciels de Grid'5000. Ces défauts ont depuis été corrigés.

A la suite des ces travaux, nous avons proposé Expo, un environnement modulaire de conduite d'expériences sur grille de calcul. Expo met l'emphase sur le contrôle des expériences et de la plate-forme cible. Une proposition forte d'Expo est l'usage d'un langage dédié intégré à un langage objet existant, Ruby. Ce langage offre différents niveaux d'abstraction pour décrire les expériences. De plus, étant intégré à Ruby, il permet une grande souplesse dans la description des expériences. Une autre contribution forte est la possibilité de découpler la partie interprétation de l'expérience de la partie exécution de cette expérience. Cette séparation assure un meilleur contrôle des commandes et plus de sécurité pour l'archivage des expériences. Expo a été validé par la conduite d'une expérience à grande échelle.

Dans le contexte des architectures multi-cœurs, nous avons montré l'importance d'une utilisation efficace de ces technologies. En effet, pour présenter un réel intérêt pour les utilisateurs en termes d'économies ou de puissance de calcul, il est nécessaire de savoir programmer en parallèle ces architectures, mais les différentes plates-formes multi-cœurs présentent de multiples visages.

Nous avons donc proposé PaSTeL, un environnement dédié à l'étude de l'adéquation entre le support exécutif et la plate-forme cible. PaSTeL permet d'étudier avec précision l'influence de divers paramètres sur le comportement d'une application ou d'un algorithme parallèle. PaSTeL reste cependant réaliste dans son comportement, il présente des performances comparables à d'autres bibliothèques parallèles destinées aux architectures multi-cœurs.

7.1 Perspectives à court terme

La campagne d'expériences a soulevé de nombreuses questions et il semble important de poursuivre les expériences pour expliquer les comportements observés. Plusieurs pistes expérimentales peuvent être explorées. Il faut tout d'abord rechercher pourquoi PaSTeL présente un mauvais passage à l'échelle sur Bordereau et ensuite étudier plus finement l'influence du placement des threads et des données sur la plate-forme Genepi pour tenter de comprendre les effets de cache observés. Il est également possible d'effectuer des mesures avec des temps de traitement plus longs, pour valider le bon passage à l'échelle de PaSTeL dans ce contexte.

Il est aussi important de mettre Expo à disposition de la communauté pour éventuellement obtenir de plus larges retours sur son utilisation et son comportement. En effet, pour l'instant Expo n'a été utilisé qu'au sein du laboratoire. Certaines des expériences réalisées n'utilisent que le module de contrôle tandis que d'autres utilisent Expo et son langage. Les premiers retours sont positifs, mais un plus large spectre d'utilisateurs serait le bienvenu.

7.2 Perspectives à moyen terme

PaSTeL propose pour l'instant un modèle de vol de travail synchrone et un lancement en cascade des threads. Ce modèle présente de nombreux avantages, mais il serait instructif de le comparer à d'autres modèles. L'implantation d'un vol asynchrone pour le cas où le traitement d'un élément est long est un premier objectif.

L'instrumentation de PaSTeL est également à poursuivre. L'intégration de bibliothèques de prise de trace est prévu pour permettre une meilleure compréhension des résultats observés. Notamment, l'accès au défaut de cache en fonction de la répartition des threads et des données donnerait de nombreuses pistes de compréhension.

Dans le cadre d'Expo, le langage esquissé doit être encore affiné, notamment pour la partie objet. En effet, la gestion de tâches à la fois hiérarchiques

et parallèles pose problème et nécessite encore du travail. La partie séquentielle du langage pourrait également bénéficier de certaines améliorations.

Il faudrait en outre envisager d'écrire une nouvelle implantation d'Expo dans un langage plus performant que Ruby, tant au niveau de la consommation mémoire que de l'utilisation du processeur. Ceci est particulièrement capital pour le serveur d'exécution, qui peut devenir gourmand en ressources lorsque de nombreuses commandes sont exécutées en parallèle.

L'intégration d'un module de base de donnée est également nécessaire. Cependant, le design adéquat ne nous est pas apparu et il va donc être nécessaire d'étudier plus avant la question. L'association entre le modèle relationnel d'une base de donnée et le modèle objet hiérarchique d'Expo semble difficile à faire efficacement.

Bibliographie

[ACH+08] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu nad Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification. Technical report, MAR 2008.

[AJR+01] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL : An Adaptive, Generic Parallel C++ Library. In *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pages 193–208, August 2001.

[Alb07] Jeannie Raye Albrecht. *Distributed Application Management.* PhD thesis, University of California, San Diego, June 2007.

[AMDa] AMD. Opteron seconde génération. `http://products.amd.com/en-us/OpteronCPUResult.aspx?f1=Second-Generation+AMD+Opteron%E2%84%A2`.

[AMDb] AMD. Opteron troisième génération. `http://products.amd.com/en-us/opteroncpuresult.aspx?f1=Third-Generation+AMD+Opteron%E2%84%A2`.

[Art] Cilk Arts. Cilk++. `http://www.cilk.com`.

[ATSV06] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Planetlab application management using plush. *SIGOPS Oper. Syst. Rev.*, 40(1) :33–40, 2006.

[BBC+04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *First Symposium on Networked Systems Design and Implementation (MSDI)*, pages 253–266, March 2004.

[BBE+99] Sandeep Bajaj, Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, Padma Haldar, Mark Handley, Ahmed Helmy, John Heidemann, Polly Huang, Satish Kumar, Steven McCanne, Reza

Rejaie, Puneet Sharma, Kannan Varadhan, Ya Xu, Haobo Yu, and Daniel Zappala. Improving simulation for network research. Technical Report 99-702b, University of Southern California, March 1999.

[BCS] E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal component model. `http://fractal.ow2.org/specification/index.html`.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03 : Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk : an efficient multithreaded runtime system. In *PPOPP '95 : Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[BKT00] P. Buneman, S. Khanna, and W.C. Tan. Data provenance : Some basic issues. *Lecture notes in computer science*, pages 87–93, 2000.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5) :720–748, 1999.

[Boa] OpenMP Architecture Review Board. Openmp 3.0 specifications. `http://www.openmp.org/mp-documents/spec30.pdf`.

[Bor07] Shekhar Borkar. Thousand core chips : a technology perspective. In *DAC '07 : Proceedings of the 44th annual conference on Design automation*, pages 746–749, New York, NY, USA, 2007. ACM.

[Bra] Tim Bray. Bonnie64. `http://code.google.com/p/bonnie-64/`.

[CDD+05] Franck Cappello, Frederic Desprez, Michel Dayde, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Nouredine Melab, Raymond Namyst, Pascale Primet, Olivier Richard, Eddy Caron, Julien Leduc, and Guillaume Mornet. Grid'5000 : A large scale, reconfigurable, controlable and monitorable grid platform.

In *Grid2005 6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[Chu04] B.N. Chun. DART : Distributed automated regression testing for large-scale network applications. In *International Conference on Principles of Distributed Systems (OPODIS), LNCS*, volume 8, 2004.

[Cok] Russell Coker. Bonnie++. `http://sourceforge.net/projects/bonnie/`.

[Cor] Standard Performance Evaluation Corporation. Spec. `http://www.spec.org`.

[Dan04] Vincent Danjean. *Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs*. PhD thesis, École normale supèrieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07, France, December 2004. 156 pages.

[Dik00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta*, page 63. Usenix, 2000.

[Dil09] Bruno Dillenseger. Clif, a framework based on fractal for flexible, distributed load testing. *Annals of Telecommunications*, 64(1) :101–120, 2009.

[DM98] L. Dagum and R. Menon. Openmp : an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1) :46–55, Jan-Mar 1998.

[DM03] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. White paper, redhat, February 2003.

[DMS94] Jack J. Dongarra, Hans W. Meuer, and Erich Strohmaier. Top500 supercomputer sites. Technical report, Supercomputer, 1994.

[DRT04] Yves Denneulin, Emmanuel Romagnoli, and Denis Trystram. A synthetic workload generator for cluster computing. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, USA, 2004. IEEE Computer Society.

[ESL07] Eric Eide, Leigh Stoller, and Jay Lepreau. An experimentation workbench for replayable networking research. In *4th USENIX Symposium on Networked Systems Design & Implementation*, pages 215–228, 2007.

[FJP+05] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto, Jr., and Hong-Linh Truong. Askalon : a tool set for cluster and grid computing : Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4) :143–169, 2005.

[FK97] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, 1997. `ftp://ftp.globus.org/pub/globus/papers/globus.pdf`.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5) :212–223, 1998.

[FQH05] Thomas Fahringer, Jun Qin, and Stefan Hainzer. Specification of Grid Workflow Applications with AGWL : An Abstract Grid Workflow Language. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005)*, Cardiff, UK, May 9-12, 2005. IEEE Computer Society Press.

[GBP07] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI : A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors. In *Parallel Symbolic Computation'07 (PASCO'07)*, number 15–23, London, Ontario, Canada, 2007. ACM.

[GCRD98] F. Galilee, G.G.H. Cavalheiro, J.-L. Roch, and M. Doreille. Athapascan-1 : On-line building data flow graph in a parallel language. *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 88–95, Oct 1998.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. ADDISON-WESLEY, third edition, January 2005. `http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf`.

[GP08] Romaric Guillier and Pascale Vicat-Blanc Primet. Methodologies and tools for exploring transport protocols in the context of high-speed networks. In *IEEE TCSC Doctoral Symposium*, May 2008.

[HBMK05] John J. Hoffman, Andrew Byrd, Kathryn M. Mohror, and Karen L. Karavanic. Pperfgrid : A grid services-based tool for the exchange of heterogeneous parallel performance data. *Parallel*

and Distributed Processing Symposium, International, 5 :175b, 2005.

[HIM+05] Lisa Hsu, Ravi Iyer, Srihari Makineni, Steve Reinhardt, and Donald Newell. Exploring the cache design space for large scale cmps. *SIGARCH Comput. Archit. News*, 33(4) :24–33, 2005.

[IEE] IEEE. Posix. `http://www.unix.org/version3/ieee_std.html`.

[Inta] Intel. Atom. `http://www.intel.com/products/processor/atom/`.

[Intb] Intel. Core 2 duo. `http://www.intel.com/products/processor/core2duo/`.

[Intc] Intel. Core i7. `http://www.intel.com/products/processor/corei7/`.

[Intd] Intel. Pentium d. `http://www.intel.com/cd/channel/reseller/asmo-na/eng/products/desktop/processor/processors/pentium-d/feature/index.htm`.

[Inte] Intel. Pentium iii. `http://www.intel.com/design/intarch/pentiumiii/pentiumiii.htm`.

[Intf] Intel. Pentium iii xeon. `http://www.intel.com/support/processors/pentiumiii/xeon/sb/CS-022664.htm`.

[Jac96] Christine Jacqmot. *Load Management in Distributed Computing Systems : Toward Adaptative Strategies*. PhD thesis, Université catholique de Louvain, January 1996.

[Jai91a] Raj Jain. *The Art of Computer Systems Performance Analysis*, page 403. John Wiley & Sons, Inc, 1991.

[Jai91b] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc, 1991.

[Jai91c] Raj Jain. *The Art of Computer Systems Performance Analysis*, chapter 16-23. John Wiley & Sons, Inc, 1991.

[JG97] E. Johnson and D. Gannon. HPC++ : Experiments with the parallel standard template library. In *Proceedings of the 11th international conference on Supercomputing*, pages 124–131. ACM New York, NY, USA, 1997.

[JM96] Christine Jacqmot and Elie Milgrom. Évaluation empirique des performances d'un système informatique : application à l'équilibrage de charge. In *Placement dynamique et répartition de charge : application aux systèmes répartis et parallèles*, pages 231–250, December 1996.

[Jona] Rick Jones. netperf. http://www.netperf.org.

[Jonb] Roel Jonkman. Netspec. http://www.ittc.ku.edu/netspec/.

[Kle] Andi Kleen. libnuma. http://oss.sgi.com/projects/libnuma/.

[KV07] A. Kukanov and M.J. Voss. The Foundations for Scalable Multi-Core Software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4), November 2007.

[KVE+92] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner. Symmetric multiprocessing in solaris 2.0. *Compcon Spring '92. Thirty-Seventh IEEE Computer Society International Conference, Digest of Papers.*, pages 181–186, Feb 1992.

[Lin] Mark A. Lindner. libconfig. http://www.hyperrealm.com/libconfig/.

[LMC03] Arnaud Legrand, Loris Marchal, and Henri Casanova. Scheduling distributed applications : the simgrid simulation framework. In *CCGRID '03 : Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 138, Washington, DC, USA, 2003. IEEE Computer Society.

[Lov93] D.B. Loveman. High performance fortran. *Parallel & Distributed Technology : Systems & Applications, IEEE*, 1(1) :25–42, Feb 1993.

[MR01] C. Martin and O. Richard. Parallel launcher for clusters of PC, parallel compting. In *Parco'01 (Parallel Computing), Naples*, 2001.

[MRH05] Cyrille Martin, Olivier Richard, and Guillaume Huard. Déploiement adaptatif d'applications parallèles. *Technique et Science Informatiques (TSI)*, 2005.

[Mue93] Frank Mueller. A library implementation of posix threads under unix. In *Proceedings of the USENIX Conference*, 1993.

[NC] WD Norcott and D Capps. http://www.iozone.org.

[Nov06] Diego Novillo. Openmp and automatic parallelization in gcc. In *GCC Developers Summit*, 2006.

[NR08] Lucas Nussbaum and Olivier Richard. Lightweight emulation to study peer-to-peer systems. *Concurrency and Computation : Practice and Experience*, 20(6) :735–749, 2008.

[NVI] NVIDIA. Geforce gtx280 specifications. http://www.nvidia.com/docs/IO/55506/GPU_Datasheet.pdf.

[OAF+04] T Oinn, M.J. Addis, J. Ferris, D.J. Marvin, M. Senger, T. Carver, M. Greenwood, K Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna : a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17) :3045–3054, June 2004.

[PF02] Radu Prodan and Thomas Fahringer. Zenturio : An experiment management system for cluster and grid computing. In *Proceedings of the 4th International Conference on Cluster Computing (CLUSTER 2002)*, September 2002.

[Pro] The Linux VServer Project. `http://www.linux-vserver.org/Linux-VServer-Paper`.

[PWDC] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. High performance linpack. `http://www.netlib.org/benchmark/hpl/`.

[Rat] Justin Rattner. Teraflops research chip. `http://techresearch.intel.com/articles/Tera-Scale/1449.htm`.

[Riz97] Luigi Rizzo. Dummynet : a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1) :31–41, 1997.

[Sch97] R.R. Schaller. Moore's law : past, present and future. *Spectrum, IEEE*, 34(6) :52–59, Jun 1997.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee : a many-core x86 architecture for visual computing. In *SIGGRAPH '08 : ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[SGG04] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, December 2004.

[SL95] Alexander Stepanov and Meng Lee. The standard template library. Technical report, Silicon Graphics Inc and Hewlett-Packard Laboratories, October 1995.

[SPBP06] Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using planetlab for network research : myths, realities, and best practices. *SIGOPS Oper. Syst. Rev.*, 40(1) :17–24, 2006.

[SRG03] R. D. Stevens, A. J. Robinson, and C. A. Goble. mygrid : personalised bioinformatics on the information grid. *Bioinformatics*, 19 Suppl 1, 2003.

[SSP07] J. Singler, P. Sanders, and F. Putze. MCSTL : The Multi-Core Standard Template Library. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694. Springer, 2007. `http://algo2.iti.uni-karlsruhe.de/singler/mcstl/`.

[Sup] NASA Advanced Supercomputing. Nas parallel benchmark. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[SV08] Erik Saule and Brice Videau. PaSTeL. Une implantation parallèle de la STL pour les architectures multi-coeurs : une analyse des performances. In *Proceedings électronique de RenPar 18*, February 2008.

[TBG+08] Samuel Thibault, François Broquedis, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. An Efficient OpenMP Runtime System for Hierarchical Architectures. In Barbara M. Chapman, Weimin Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, editors, *A Practical Programming Model for the Multi-Core Era, 3rd International Workshop on OpenMP, IWOMP 2007, Beijing, China, June 3-7, 2007, Proceedings*, volume 4935 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2008.

[Thi07] Samuel Thibault. *Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications.* PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, December 2007. 128 pages.

[Tic98] Walter F. Tichy. Should computer scientists experiment more ? *COMPUTER : IEEE Computer*, 31 :32–40, 1998.

[Top] Top500.Org. Top500. `http://www.top500.org`.

[VR08] Brice Videau and Olivier Richard. Expo : un moteur de conduite d'expériences pour plates-forme dédiées. In *Conférence Française en Systèmes d'Exploitation (CFSE)*, 2008.

[VSM08] Brice Videau, Erik Saule, and Jean-François Méhaut. PaSTeL : Parallel Runtime and Algorithms for Small Datasets. Technical Report 6650, INRIA, 2008.

[VSM09] Brice Videau, Erik Saule, and Jean-François Méhaut. PaSTeL : Parallel Runtime and Algorithms for Small Datasets. In *2009 International Workshop on Multi-Core Computing Systems (MuCoCoS'09)*. IEEE Computer Society Press, March 2009.

[VTR07] Brice Videau, Corinne Touati, and Olivier Richard. Toward an Experiment Engine for Lightweight Grids. In *MetroGrid workshop : Metrology for Grid Networks*. ACM publishing, October 2007.

[VYW+02] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator, 2002.

[WGN+02] Brian White, Shashi Guruprasad, Mac Newbold, Jay Lepreau, Leigh Stoller, Robert Ricci, Chad Barb, Mike Hibler, and Abhijeet Joglekar. Netbed : an integrated experimental environment. *Computer Communication Review*, 32(3) :27, 2002.

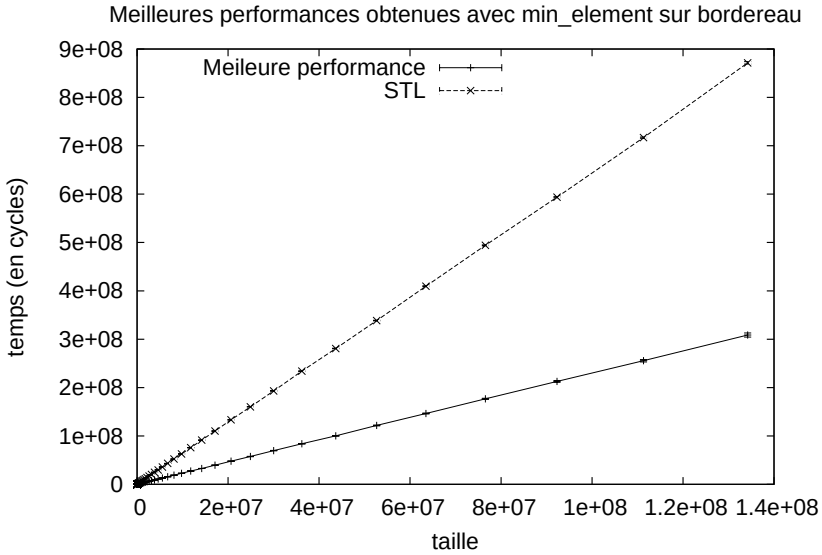
[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI) :255–270, 2002.

[Wor05] J. Worringen. Experiment management and analysis with perfbase. *Cluster Computing, IEEE International Conference on*, 0 :1–11, 2005.

[WRCW05] Yanyan Wang, Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf. Automating experimentation on distributed testbeds. In *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 164–173, New York, NY, USA, 2005. ACM.

[YB05] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4) :171–200, September 2005.

Annexes



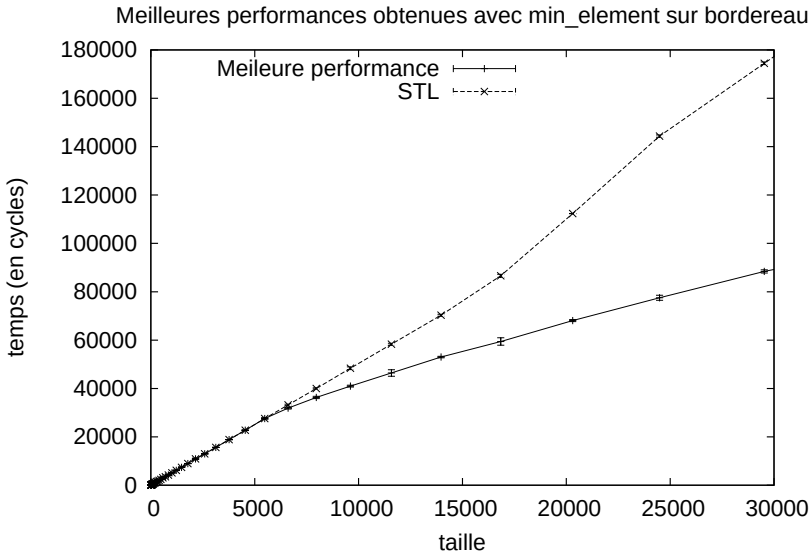
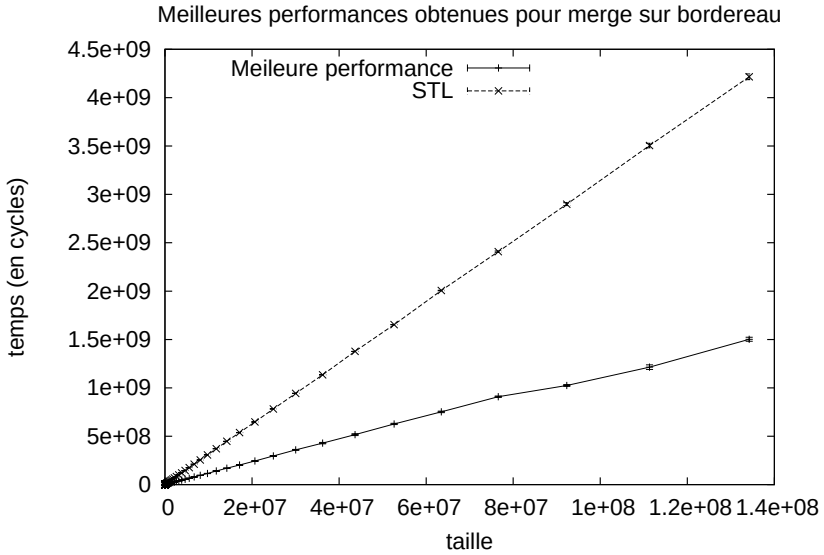


FIGURE 7.1 – Meilleures performances obtenues sur Bordereau avec l'algorithme *min_ element*



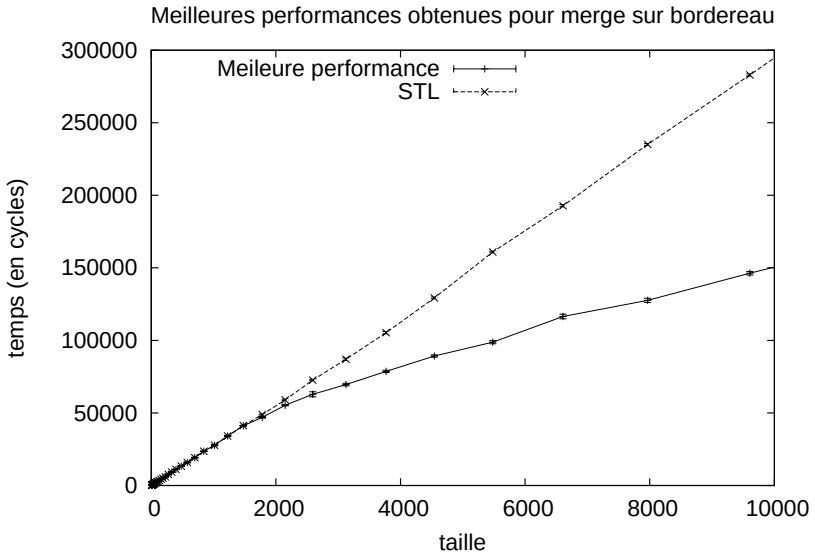
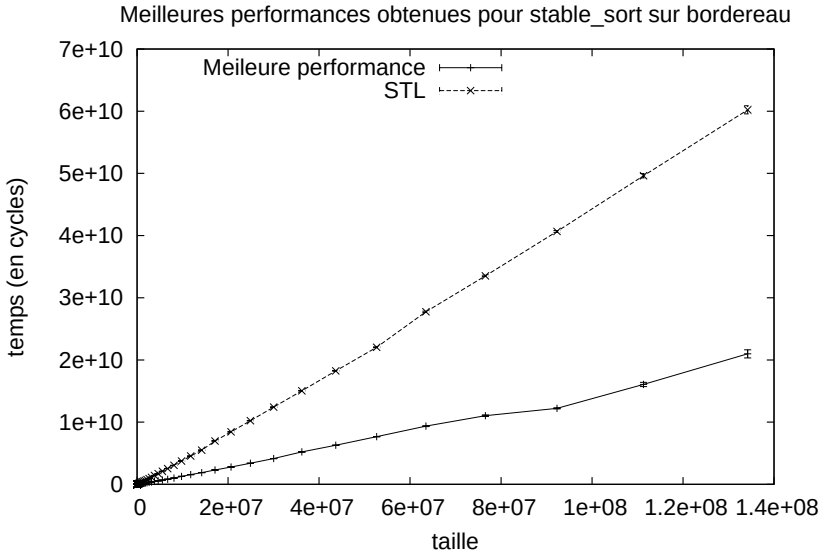


FIGURE 7.2 – Meilleures performances obtenues sur Bordereau avec l'algorithme *merge*



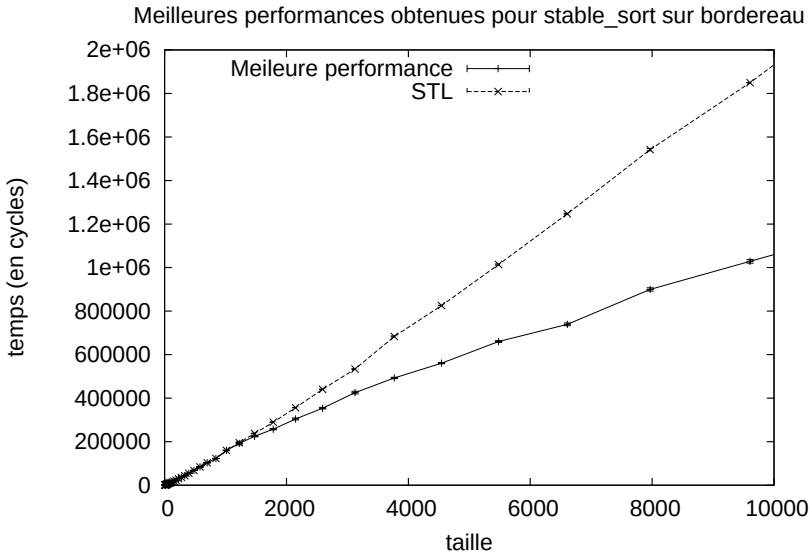
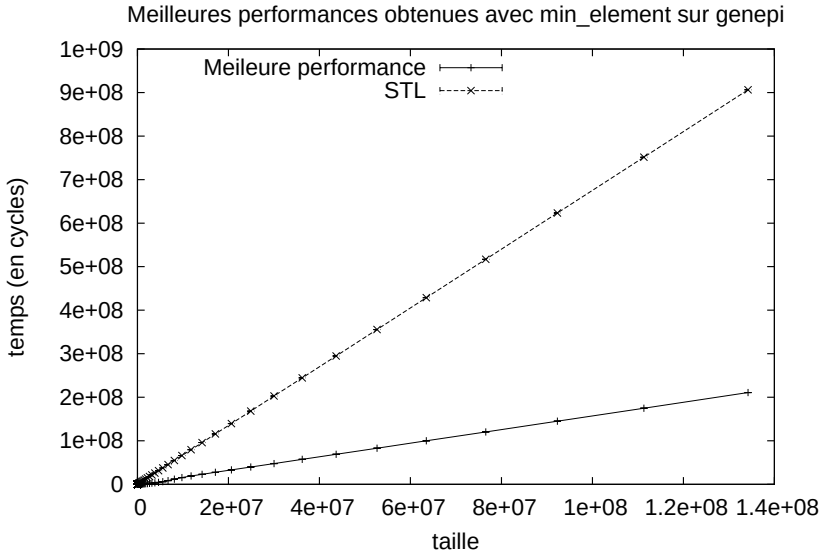


FIGURE 7.3 – Meilleures performances obtenues sur Bordereau avec l'algorithme *stable_sort*



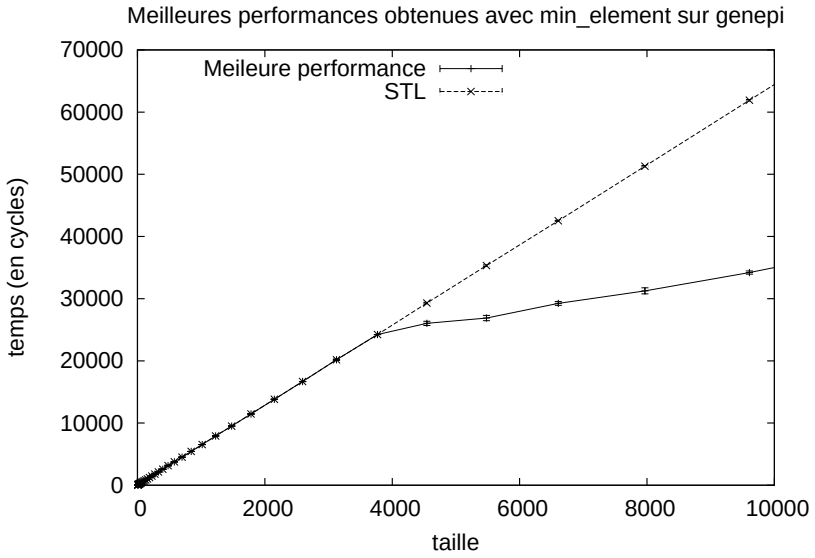
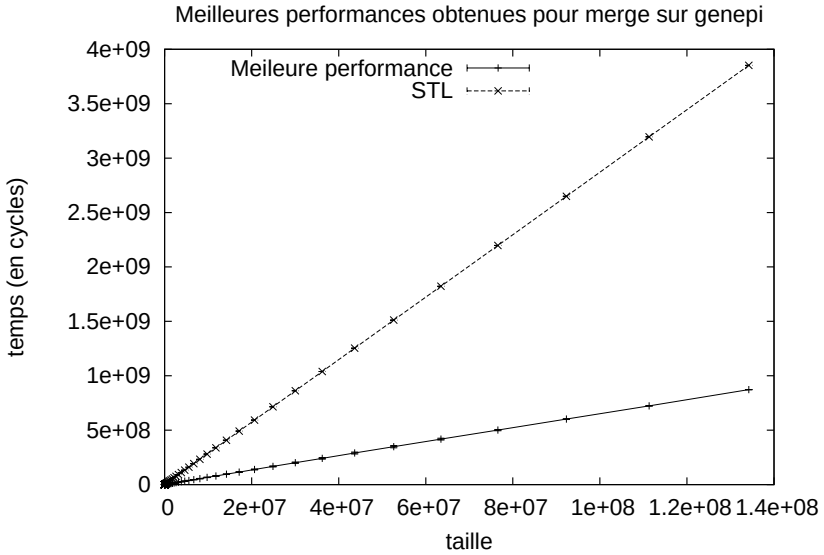


FIGURE 7.4 – Meilleures performances obtenues sur Genepi avec l'algorithme *min_ element*



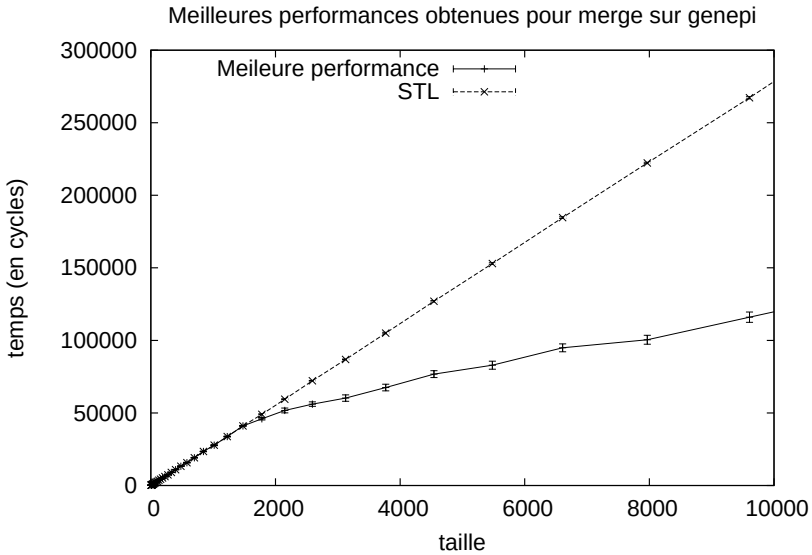
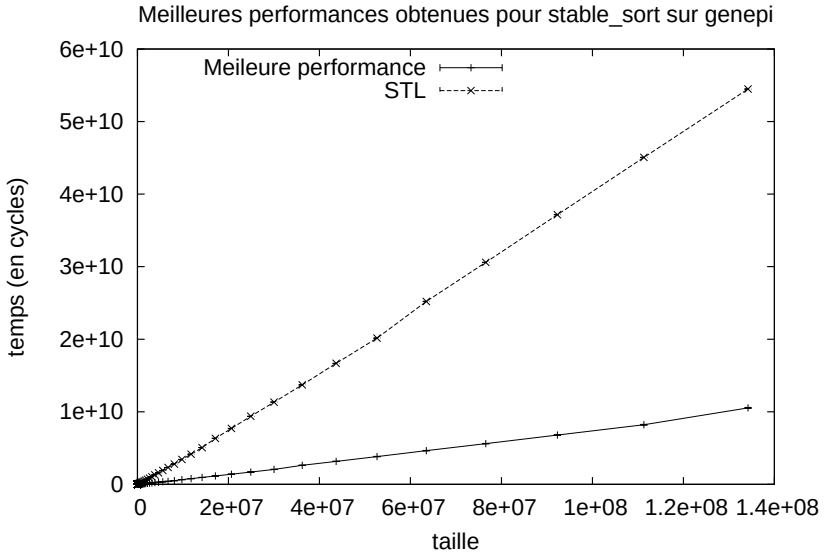


FIGURE 7.5 – Meilleures performances obtenues sur Genepi avec l'algorithme *merge*



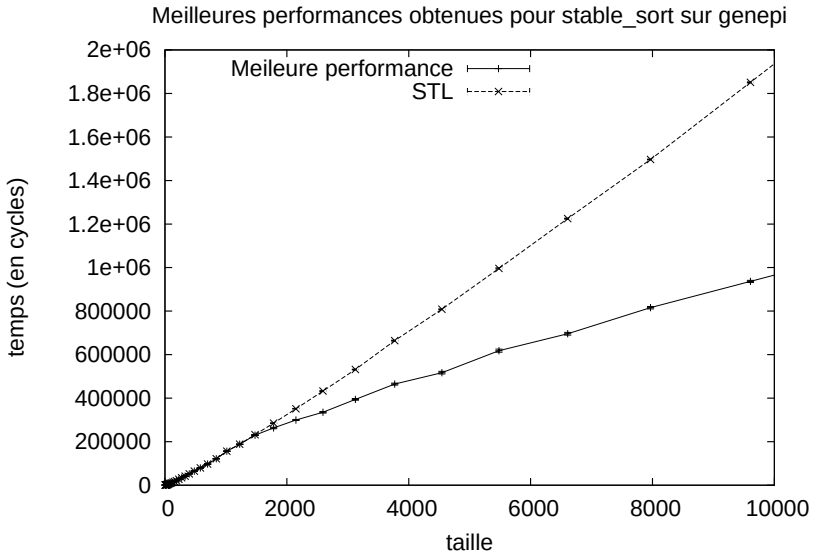
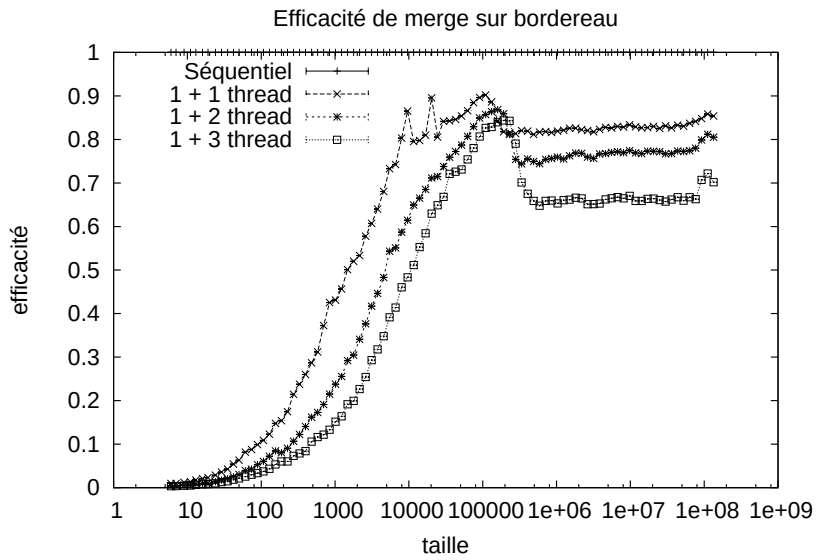


FIGURE 7.6 – Meilleures performances obtenues sur Genepi avec l'algorithme *stable_sort*



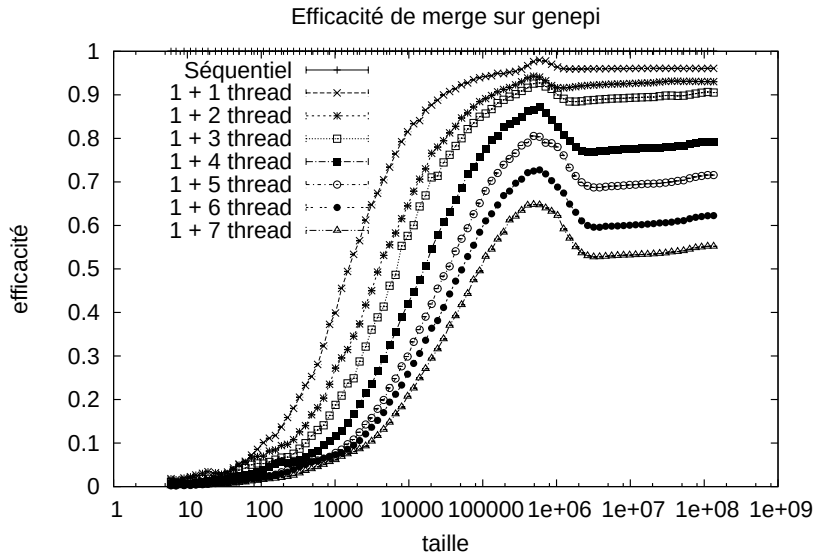
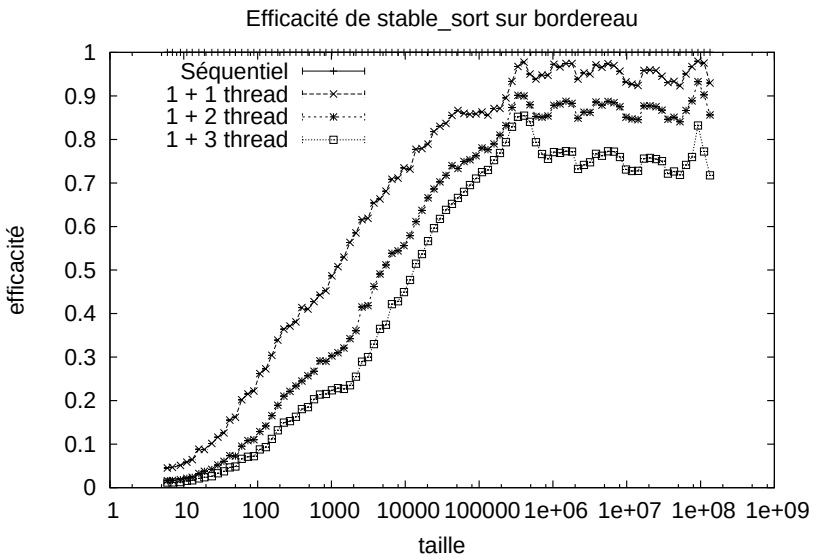


FIGURE 7.7 – Efficacité de l'algorithme *merge* sur Bordereau et Genepi en fonction du nombre de threads utilisés



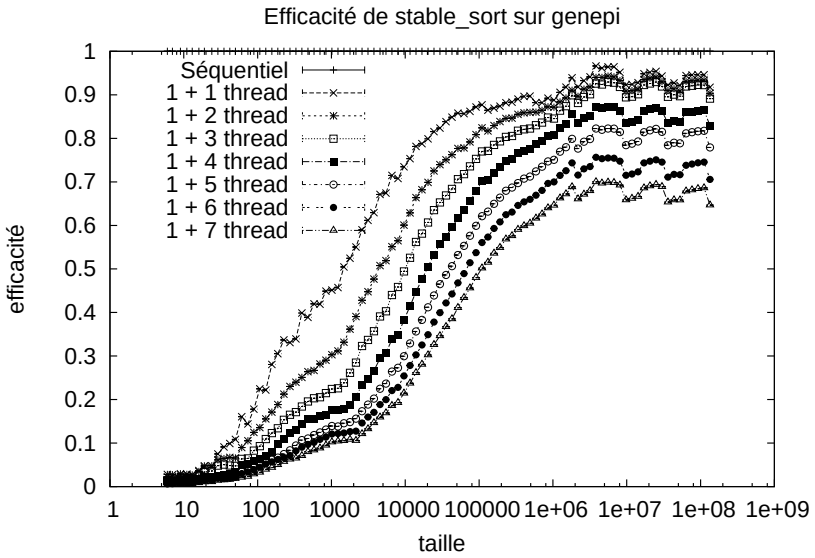
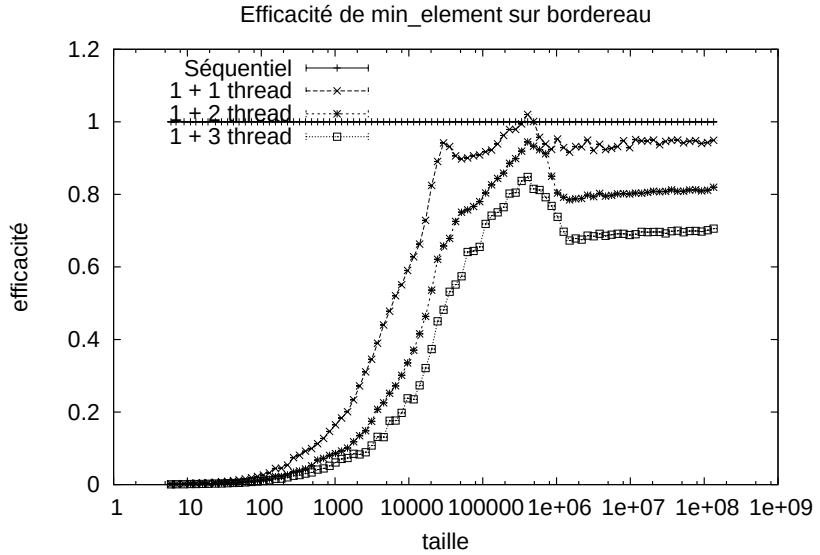


FIGURE 7.8 – Efficacité de l'algorithme *stable_ sort* sur Bordereau et Genepi en fonction du nombre de threads utilisés



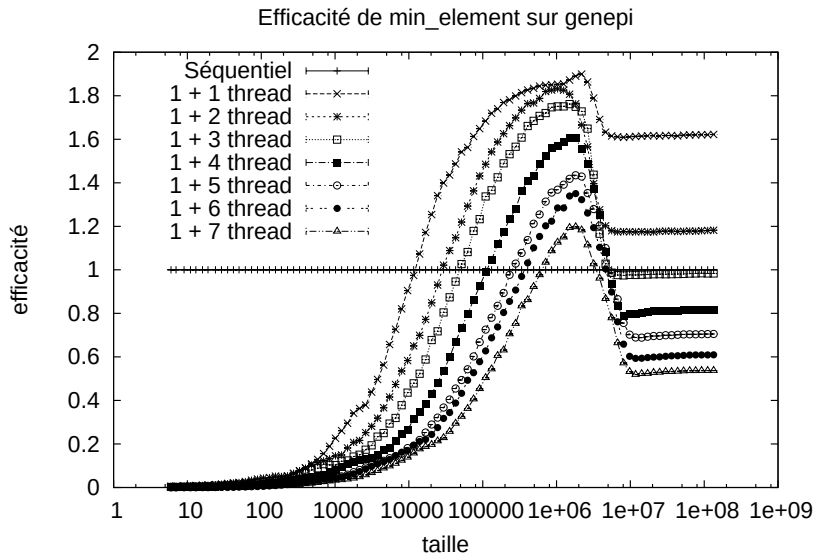


FIGURE 7.9 – Efficacité de l'algorithme *min_ element* sur Bordereau et Genepi en fonction du nombre de threads utilisés

résumé : Les besoins en puissance de calcul ne cessent d'augmenter. Pour répondre à ces besoins, de nouvelles architectures sont apparues. Les grilles et grappes de calcul, qui permettent d'agréger la puissance de plusieurs machines et les processeurs multi-cœurs qui permettent d'offrir plus de puissance sans nécessité d'augmenter la fréquence et la consommation énergétique du processeur.
Cependant, l'étude de ces nouvelles architectures n'est pas aisée. Pour l'étude des grilles de calcul, nous disposons de plates-formes dédiées. Néanmoins, la conduite d'expérience sur ces plates-formes complexe est une tâche difficile à mettre en œuvre. Dans le cas des processeurs multi-cœurs, leur comportement est mal connu.
Dans cette thèse nous proposons deux outils dédiés à l'étude des nouvelles architectures. Le premier, Expo, est un logiciel de conduite d'expérience sur grille dédiée à l'expérimentation. Expo permet d'automatiser en partie la conduite d'une expérience, tout en essayant de garantir son bon déroulement. Expo propose également un langage concis de description d'expérience adapté à la problématique des grilles.
Le second outil, PaSTeL, est dédié à l'étude de l'adéquation entre un support exécutif et une architecture. Hautement paramétrable, il permet d'étudier les multiples facettes d'une solution de parallélisation s'exécutant sur une architecture donnée.
Les deux outils ont été validés au cours de leur développement, mais également lors d'une campagne expérimentale visant à étudier le comportement d'un moteur de vol de travail sur différentes architectures multi-cœurs.
mots clés : Expérimentation, Reproductibilité, Grilles, Multi-Cœurs, PaSTeL, Expo

abstract : Needs in computing power are increasing steadily. In order to meet those needs, new architectures have appeared. Computing grids and clusters are gathering the computing power of several machines, and multi-core processors are offering more computing power without the need to increase the frequency and electrical power usage.
Nonetheless, The study of those new architectures is not an easy task. In order to study computing grids researchers have dedicated platforms. But the conduct of experiments on such platforms is a difficult process. In the case of multi-core processors, their behavior is not well known.
In this thesis we put forward two tools dedicated to the study of new architectures. The first one, Expo, is an experiment management software for grid architectures. Expo allows for automated experiment conduct while still trying to guarantee its sound unrolling. Expo also provides a language to describe experiments on grid infrastructures. The second one, PaSTeL, is dedicated to the study of interactions between architectures and execution strategies. Highly tunable, it grants user with means to study in depth the behavior of a parallel solution running on a given architecture.
Both tools have been validated during the development phase, but also during an experimental study of a work stealing engine on several multi-core architectures.
keywords : Experimentation, Reproducibility, Grid, Multi-Core, PaSTeL, Expo