HAL
open science

# Prédiction de performances d'applications de calcul haute performance sur réseau Infiniband

Jérôme Vienne

## To cite this version:

Jérôme Vienne. Prédiction de performances d'applications de calcul haute performance sur réseau Infiniband. Calcul parallèle, distribué et partagé [cs.DC]. Université de Grenoble, 2010. Français. NNT : . tel-00728156

**HAL Id: tel-00728156**

**https://theses.hal.science/tel-00728156v1**

Submitted on 5 Sep 2012

**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 DE GRENOBLE

# THÈSE

Pour obtenir le grade de

## DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par


## Jérôme VIENNE


Thèse dirigée par **M. Jean-François Méhaut**
et codirigée par **M. Jean-Marc Vincent**

préparée au sein **du Laboratoire d'Informatique de Grenoble**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

# Prédiction de performances d'applications de calcul haute performance sur réseau Infiniband

Thèse soutenue publiquement le **le 1 Juillet 2010**,
devant le jury composé de :

**M. Frédéric Desprez**
Directeur de recherche, INRIA Lyon, Président
**M. Eddy Caron**
Maître de conférences, École Normale Supérieure de Lyon, Rapporteur
**M. François Spies**
Professeur des universités, Université de Franche-Comté, Rapporteur
**M. Xavier Vigouroux**
BULL SAS, Examinateur
**M. Jean-François Méhaut**
Professeur des universités, Université Joseph Fourier, Directeur de thèse
**M. Jean-Marc Vincent**
Maître de conférences, Université Joseph Fourier, Co-Directeur de thèse

*En mémoire de mon père, Vienne Francis (1953-2009)*

# Remerciements

Une thèse est un long chemin que l'on ne peut parcourir seul. C'est grâce à des rencontres, des échanges mais aussi le savoir de plusieurs personnes que l'on y arrive.

Je voudrais remercier très fortement les membres de mon jury : tout d'abord, Frédéric Desprez pour m'avoir fait l'honneur de présider ce jury de thèse ; Eddy Caron et François Spies pour avoir accepté de rapporter sur ma thèse, leur lecture attentive et la pertinence de leurs remarques m'ont permis d'améliorer la qualité du document ; Xavier Vigouroux pour avoir accepté de faire partie de mon jury malgré son emploi du temps chargé.

Cette thèse s'est inscrite dans le cadre d'une convention CIFRE entre la société BULL SAS et l'équipe MESCAL du LIG. Je tiens donc à remercier la société BULL SAS ainsi que tous ces employés pour avoir soutenu les travaux de recherche de cette thèse. Plus particulièrement, je remercie Jean-François Lemerre, initiateur de ce sujet de thèse, qui m'a beaucoup appris sur l'univers du *HPC*, Christophe Berthelot toujours disponible pour répondre à mes questions et qui m'a fait découvrir l'environnement du benchmarking et enfin Xavier Vigouroux qui m'a intégré dans son équipe et m'a permis de comprendre l'ensemble des problématiques industrielles liées à son activité.

Une thèse s'est aussi des rencontres. Et je tenais à remercier Julien Bourgeois. Sans sa publication à Renpar (comme quoi ça sert ☺), je n'aurais sûrement pas pu découvrir sa thèse et l'outil *Chronosmix*, et je ne serais pas arrivé si facilement à *Rose* et donc à faire ma partie sur l'analyse statique. Son expérience dans le domaine de la prédiction de performance et la compilation de codes parallèles m'a permis d'éviter de nombreux pièges. Je voudrais aussi remercier Bogdan Cornéa avec qui j'ai beaucoup travaillé pour la mise en place d'un protocole expérimental, sur l'utilisation de *Rose* et de *PAPI*. Nous avons eu de très nombreux échanges durant plus d'une année (J'espère que mes coups de fils et mes mails ne vont pas trop lui manquer ☺). Je voudrais aussi remercier Michihiro Koibuchi pour avoir accepté de me prendre en Internship à Tokyo pendant 3 mois et avec qui j'ai beaucoup appris dans le domaine du réseau.

Finalement, je voudrais remercier Jean-François Méhaut pour avoir accepté de diriger officiellement cette thèse dont l'encadrement scientifique était assuré par Jean-Marc Vincent. Il y a 9 ans, quand j'étais étudiant en *RICM*, je n'aurais jamais pensé travailler avec mon professeur de probabilité. Suite à ma reprise d'étude, il m'a donné une chance en 2003 en m'intégrant en *Magistère*. Mon premier stage avec Jean-Louis Roch et Samir Jafar m'a permis de découvrir la super ambiance de travail de laboratoire ainsi que le monde de la recherche. Puis, il m'as pris sous son aile en me confiant son bébé *PSi2*. J'ai alors découvert le monde des chaînes de Markov et de la simulation parfaite. Ensuite, nous avons continué notre collaboration sur cette thèse. Je le remercie donc de m'avoir

fait confiance et je voudrais lui dire combien j'ai apprécié travailler avec lui. J'ai apprécié sa rigueur scientifique, sa pédagogie, sa patience ainsi que l'ensemble des choses que j'ai pu apprendre à ses cotés (même si certaines ne sont toujours pas rentrées). J'espère que dans le futur, nous continuerons à collaborer ensemble.

Au cours de ces 9 années au sein du laboratoire, j'ai rencontré énormément de monde. Il serait difficile de tous les citer et je ne voudrais pas citer de personnes particulières de peur d'en oublier. Merci à tous les membres de l'équipe MESCAL et MOAIS. Juste une dédicace spéciale à l'ensemble de mes anciens collègues de bureau (Dr. Maxime, Dr. Luiz, Dr. Thais, Daniel (Dr. dans 1 an) , Rémi et le petit dernier le soon Dr. Yanik ☺).

Enfin, un grand merci à ma famille pour son soutien et ses encouragements ! Et plus spécialement à ma petite femme, Madame Vienne Dar'ya, qui m'a toujours supporté et encouragé même pendant ces longs mois de vie d'ermite quand je rédigeais ce manuscrit.

# Table des matières

**1 Introduction** **1**
1.1 Contexte scientifique et industriel . . . . . 2
1.2 La prédiction de performances . . . . . 2
1.3 Contributions de la thèse . . . . . 3
1.4 Organisation du manuscrit . . . . . 6

**I L'ENVIRONNEMENT DU CALCUL HAUTE PERFORMANCE** **7**

**2 Le calcul haute performance** **11**
2.1 Introduction . . . . . 11
2.2 Nœuds de calcul multi-coeurs . . . . . 12
2.2.1 L'environnement du processeur . . . . . 12
2.2.2 L'architecture mémoire . . . . . 15
2.3 Le réseau haute performance *InfiniBand* . . . . . 17
2.3.1 La bibliothèque bas niveau *libibverbs* . . . . . 17
2.3.2 Performance de la libibverbs . . . . . 18
2.3.3 L'interface de communication *MPI* . . . . . 21
2.3.4 Performance de la bibliothèque *MPI* . . . . . 22
2.3.5 La topologie réseau et le routage . . . . . 24
2.4 Les outils de benchmarking *MPI* . . . . . 26
2.4.1 IMB . . . . . 26
2.4.2 MPBench . . . . . 27
2.4.3 Mpptest . . . . . 28
2.4.4 SkaMPI . . . . . 28
2.4.5 NetPIPE . . . . . 29
2.4.6 MPIBench . . . . . 29
2.4.7 Bilan . . . . . 29
2.5 La programmation parallèle . . . . . 31
2.5.1 La taxinomie de Flynn . . . . . 31
2.5.2 La programmation par mémoire partagée . . . . . 32
2.5.3 La programmation par passage de messages . . . . . 32

2.5.4 Conclusion . . . . . 33
2.6 Les benchmarks utilisés dans le calcul haute performance . . . . . 34
2.6.1 STREAM . . . . . 34
2.6.2 Linpack . . . . . 34
2.6.3 Les NAS Parallel Benchmarks . . . . . 35
2.6.4 SpecMPI . . . . . 35
2.6.5 Bilan . . . . . 35
2.7 Conclusion . . . . . 36

**3 Les techniques de modélisation de performance des systèmes parallèles ou distribués 39**
3.1 Introduction . . . . . 39
3.2 Approche Analytique et modèles abstraits . . . . . 40
3.2.1 Amdahl . . . . . 41
3.2.2 Le modèle PRAM . . . . . 41
3.2.3 Adve . . . . . 42
3.2.4 Les méthodes d'analyse formelle . . . . . 44
3.2.5 Le Bulk Synchronous Parallelism . . . . . 44
3.3 La simulation . . . . . 46
3.3.1 Simics . . . . . 46
3.4 Les environnements de prédictions de performances . . . . . 47
3.4.1 POEMS . . . . . 47
3.4.2 WARPP . . . . . 47
3.4.3 Saavedra et Smith ; Chronosmix . . . . . 48
3.5 Vue d'ensemble des approches . . . . . 49
3.6 Discussion . . . . . 50

**II MODÉLISATION DES APPLICATIONS PARALLÈLES 53**

**4 Modélisation des phases de calcul 57**
4.1 Introduction . . . . . 57
4.2 Découpage de code et l'outil *ROSE* . . . . . 59
4.2.1 ROSE . . . . . 59
4.2.2 Les différentes représentations d'un programme . . . . . 59
4.2.2.1 L'Abstract Syntax Tree (AST) . . . . . 60
4.2.2.2 Le Data Dependance Graph (DDG) . . . . . 62
4.2.2.3 Le Control Dependence Graph (CDG) . . . . . 62
4.2.2.4 Le System Dependence Graph (SDG) . . . . . 62
4.3 Observations préliminaires . . . . . 64
4.3.1 Protocole expérimental . . . . . 64
4.3.2 Les temps de chargement . . . . . 66

4.3.3 L'impact du niveau d'optimisation du compilateur . . . . . 68
4.3.4 Discussion . . . . . 68
4.4 Formalisme de la modélisation . . . . . 69
4.4.1 Instructions simples et blocs de base . . . . . 71
4.4.2 Les structures de boucle . . . . . 72
4.4.3 Les structures conditionnelles . . . . . 74
4.4.4 Les sous-routines . . . . . 75
4.5 Evaluation sur 10 blocs . . . . . 76
4.5.1 Impact du compilateur . . . . . 77
4.5.2 Introduction du contexte . . . . . 79
4.5.3 Impact du niveau de compilation sur l'estimation . . . . . 80
4.5.4 Discussion . . . . . 81
4.6 Conclusion . . . . . 82

**5 Modélisation des phases de communication 83**
5.1 Introduction . . . . . 83
5.2 Travaux préliminaires . . . . . 83
5.2.1 Approche . . . . . 84
5.2.2 Modèle prédictif des temps de communications . . . . . 86
5.2.3 Modèles de répartition de bande passante . . . . . 87
5.2.4 Validation . . . . . 87
5.2.5 Discussion . . . . . 87
5.3 La gestion de la concurrence dans le réseau *InfiniBand* pour les communications inter-nœud . . . . . 89
5.4 Observations préliminaires . . . . . 90
5.4.1 Protocole expérimental . . . . . 90
5.4.2 Expériences introductives . . . . . 93
5.4.2.1 Latence . . . . . 93
5.4.2.2 Débit . . . . . 93
5.4.3 Schémas de communications complexes . . . . . 97
5.4.4 Modification des paramètres . . . . . 99
5.4.4.1 Temps de départ et tailles de messages différents . . . . 100
5.4.4.2 Comparaison des pénalités entre génération de cartes *InfiniBand* différentes . . . . . 100
5.4.4.3 Comparaison des pénalités avec d'autres réseaux ou implémentation *MPI* . . . . . 102
5.4.4.4 Impact du routage . . . . . 104
5.4.5 Bilan des expériences . . . . . 105
5.5 Modèle de répartition de bande passante du réseau *InfiniBand* . . . . . 106
5.5.1 Objectif . . . . . 106
5.5.2 Approche . . . . . 106

5.5.3 Exemple de calcul de pénalité . . . . . 108
5.6 Conclusion . . . . . 108

**III EVALUATION DU MODÈLE 111**

**6 Evaluation de la prédiction 115**
6.1 Introduction . . . . . 115
6.2 Outil de simulation . . . . . 115
6.3 Méthode d'évaluation . . . . . 117
6.4 Évaluation sur des graphes synthétiques . . . . . 117
6.4.1 Arbres . . . . . 117
6.4.2 Graphes complets . . . . . 119
6.4.3 Discussion . . . . . 121
6.5 Évaluation du modèle de communication sur une application . . . . . 121
6.5.1 Le programme Socorro . . . . . 121
6.5.2 Obtention des traces . . . . . 122
6.5.3 Évaluation . . . . . 123
6.6 Bilan . . . . . 125

**7 Conclusion et perspectives 127**
7.1 Objectifs de la thèse . . . . . 127
7.2 Démarche proposée . . . . . 127
7.3 Travaux réalisés . . . . . 128
7.4 Perspectives . . . . . 129
7.4.1 Approfondir la modélisation . . . . . 129
7.4.2 Développement d'outils de prédiction de pertes de performance pour le réseau *InfiniBand* . . . . . 130
7.4.3 Utilisation de l'outil dPerf pour la génération de trace . . . . . 130

**IV ANNEXES 133**

**A PAPI 135**

**B Code exemple 137**

**Bibliographie 142**

**Résumé 152**

# Table des figures

1.1 Framework général . . . . . 4

2.1 Exemple du problème de cohérence de cache . . . . . 14
2.2 Architecture UMA, NUMA et cohérence de cache . . . . . 16
2.3 Comparaison des latences en microsecondes des opérations élémentaires de la bibliothèque libibverbs . . . . . 19
2.4 Comparaison des débits des opérations élémentaires de la bibliothèque libibverbs . . . . . 20
2.5 Mesure du débit sur Infiniband . . . . . 20
2.6 Étude de la latence au niveau *MPI* . . . . . 23
2.7 Étude du débit niveau *MPI* . . . . . 23
2.8 Exemple de routage statique InfiniBand sur un *Fat-Tree* . . . . . 25

4.1 Exemple d'éléments impactant sur le temps d'exécution . . . . . 58
4.2 Découpage de la phase de calcul . . . . . 58
4.3 Code d'exemple . . . . . 60
4.4 AST obtenu avec *Rose* . . . . . 61
4.5 DDG de la fonction *main* obtenu avec *Rose* . . . . . 63
4.6 SDG : Les relations entre les procédures et leurs dépendances apparaissent 65
4.7 Evolution du ratio entre le temps d'execution d'une boucle et le nombre d'itérations sur Core 2 quad . . . . . 66
4.8 Evolution du ratio entre le temps d'une boucle et son nombre d'itération sur Nehalem . . . . . 67
4.9 Evolution du ratio suivant les options de compilation sur Core 2 quad avec icpc . . . . . 69
4.10 Evolution du ratio suivant les options de compilation sur Core 2 quad avec g++ . . . . . 70
4.11 Décomposition des blocs en sous-blocs . . . . . 72
4.12 Exemple de résolution par analyse statique . . . . . 74
4.13 Évolution du ratio sur Core 2 quad avec icpc et une optimisation -O2 . . . 81

5.1 Pseudo-code du programme de test . . . . . 85
5.2 Simulation à événements discrets . . . . . 88

5.3 Pseudo-code du programme de test modifié . . . . . 92
5.4 Impact de la charge du réseau sur la latence suivant les versions des cartes *InfiniBand* . . . . . 94
5.5 Conflit Entrant/Entrant . . . . . 95
5.6 Conflit Sortant/Sortant . . . . . 95
5.7 Conflit Entrant/Sortant . . . . . 96
5.8 Conflits complexes sur le réseau *InfiniBand* avec *MPI BULL* 2. . . . . 98
5.9 Comparaison des pénalités suivant la génération de carte *InfiniBand* . . . 101
5.10 Comparaison des pénalités suivant le réseau ou l'implémentation *MPI* . . 103
5.11 Schéma détaillé d'une communication. . . . . 107

6.1 Framework du simulateur . . . . . 116
6.2 Précision du modèle *InfiniBand* : cas des arbres . . . . . 118
6.3 Précision du modèle *InfiniBand* : cas des graphes complets . . . . . 120
6.4 Matrice de communication point-à-point pour l'application Socorro avec 64 processus *MPI* en taille mref. . . . . 122
6.5 Matrice de communication point-à-point pour l'application Socorro avec 32 processus *MPI* en taille mref . . . . . 123
6.6 Évaluation du modèle *InfiniBand* avec Socorro avec 32 processus *MPI* . . 124
6.7 Comparaison entre le temps prédit et mesuré avec MPIBULL2 . . . . . 124

7.1 Framework de l'outil DPerf . . . . . 131

# Introduction 1

Au fil des années, la puissance des grappes de calcul n'a cessé de croître. Au début du siècle, le nombre d'opérations à virgule flottante allait jusqu'au tera flop, les avancées ont été telles que la barrière du péta flop a été atteinte en 2008. Cette montée en puissance est liée à l'ensemble des progrès réalisés aussi bien au niveau matériel (processeurs plus puissants, réseaux plus rapides, etc.) que logiciel (compilation plus optimisée, amélioration des bibliothèques de communications, etc.). Ces améliorations ont permis d'accélérer le traitement des données et d'aborder des problèmes de taille de plus en plus importante. Néanmoins, lorsque l'on souhaite améliorer les performances d'une application parallèle, deux voies sont envisageables :

- optimiser le programme parallèle, afin qu'il exploite au mieux les ressources.
- adapter la configuration de la grille de calcul, afin de fournir un meilleur environnement pour l'exécution du programme parallèle.

En règle générale, ces deux techniques sont utilisées pour optimiser les performances d'un programme, mais pas toujours dans le même ordre. En effet, beaucoup de paramètres rentrent en ligne de compte. Lorsqu'un code est complexe et comporte plusieurs millions de lignes de code, il peut être difficile à optimiser rapidement. Le choix de l'achat d'une nouvelle grappe de calcul peut s'avérer plus judicieux. Par contre, si pour des raisons économiques ou des choix stratégiques, l'achat n'est pas envisageable, l'optimisation du code reste la seule solution. Pour cela, il faut généralement faire appel à des outils permettant d'identifier les pertes de performance.
Si un changement de configuration est choisi. Les entreprises proposent généralement un appel d'offre aux différents constructeurs. Cet appel regroupe un cahier des charges complet demandant une estimation des performances au niveau réseau ou de l'accès mémoire, mais aussi les résultats attendus aux différents tests, voire des estimations de temps d'exécution d'applications développées par l'entreprise. Le constructeur remportant l'appel d'offre étant celui offrant le meilleur rapport entre les résultats estimés et le coût global de l'architecture.
Afin de pouvoir offrir une réponse la plus correct possible, il est nécessaire de développer des méthodes permettant de prédire le plus justement les réponses à ces offres. Ces méthodes devant avoir une mise en œuvre simple et rapide, afin de pouvoir aider aux dé-

cisions concernant les choix architecturaux tels que la quantité de mémoire, la puissance du processeur, le type de réseau, etc.
Les grappes de calcul utilisées dans le calcul haute performance étant généralement homogène, notre approche vise à décomposer le programme et d'estimer ces temps de calcul en se basant sur un ensemble de micro-benchmarks effectués sur un seul nœud. L'émergence des architectures multi-cœurs a créé un partage des ressources et notamment de la ressource réseau. C'est pourquoi, il est nécessaire de développer une modélisation du comportement du réseau, lorsqu'il est soumis à la concurrence. Cela permet ainsi d'avoir un modèle prédictif de performance incluant l'aspect calcul et communication.
L'ensemble des travaux, présentés ici, permet d'avoir une approximation des temps de calcul des programmes parallèles ainsi que des temps de communications point-à-point sur le réseau *InfiniBand*, fournissant ainsi une estimation du temps d'exécution du programme, et permettant de guider les constructeurs sur leurs choix. Ces choix sont importants, car ils permettent de fournir la réponse la plus adaptée aux requêtes des clients.

## 1.1 Contexte scientifique et industriel

Cette thèse s'inscrit dans le cadre du projet *LIPS* (LInux Parallel Solution) qui regroupe la société *BULL*, l'*INRIA* et le laboratoire *LIG (équipe Mescal)*. Le but du projet *LIPS* est de fournir à BULL des compétences et des solutions afin de mieux répondre aux demandes des clients.
*BULL* est le leader européen du calcul haute performance (HPC). Ses secteurs d'activité sont la conception de serveurs ouverts et de solutions de stockage sécurisées, le développement d'intergiciels pour les applications, le support ainsi que le service.
L'*Institut National de Recherche en Informatique et en Automatique* (*INRIA*) est un organisme de recherche publique, placé sous la double tutelle des ministères de la recherche et de l'industrie, qui a pour vocation d'entreprendre des recherches fondamentales et appliquées dans les domaines des sciences et technologies de l'information et de la communication (STIC).
Les axes de recherche de l'équipe MESCAL du laboratoire LIG comprennent la modélisation, la simulation, l'évaluation et l'optimisation des grilles de calcul et plus généralement des grands systèmes à évènements discrets, par des techniques déterministes et stochastiques.
Cette thèse s'inscrit dans la continuité des travaux de Martinasso Maxime sur les communications concurrentes [67].

## 1.2 La prédiction de performances

Depuis l'apparition des machines parallèles, de nombreuses méthodes sont apparues afin d'estimer les performances de ces machines. Néanmoins, il existe très peu d'outils

d'extrapolation de performance. Pourtant, un réel besoin existe dans le milieu industriel. Lors des procédures d'appel d'offre, il est important de proposer une machine maximisant les performances des codes client tout en tenant compte des contraintes. Si un constructeur est trop pessimiste sur ces prédictions, il risque de perdre le marché au profit de la concurrence. S'il est trop optimiste, il risque de devoir payer des pénalités, ou de se voir refuser l'achat de sa grappe de calcul, lors des tests de vérifications de performances.
Pour cela, il est indispensable de développer une méthode permettant d'estimer au mieux les temps de calcul et de communication des applications parallèles afin de dimensionner au mieux la grappe de calcul.
Les grappes de calcul dans le domaine du calcul haute performance sont souvent homogènes. En utilisant cette particularité, l'objectif est de développer une méthode d'estimation de temps de calcul en utilisant un seul nœud. Pour les communications, l'augmentation du nombre de cœurs au sein d'un nœud a amené à un partage de la ressource réseau créant des problèmes de contention. La prise en compte de cette concurrence est impérative afin d'estimer au mieux les temps des communications et de comprendre certaines pertes de performance.

## 1.3 Contributions de la thèse

La contribution apportée par cette thèse s'articule autour de trois domaines liés à l'estimation de performance. La figure 1.1 permet d'avoir une vision globale de la démarche proposée.

### Modélisation des temps de calcul

Afin de pouvoir extrapoler les temps de calcul des applications parallèles en fonction des paramètres, il a été nécessaire de trouver une méthode permettant de caractériser les temps de calcul tout en s'appuyant sur le code source. Saavedra [88] avait eu l'idée d'évaluer une application en termes d'opérations élémentaires et de s'appuyer sur un micro-benchmarking de ces opérations sur la plate-forme cible, afin d'estimer le temps d'exécution de cette application. Bourgeois [12, 55] reprit l'idée et l'appliqua à des programmes parallèles. Néanmoins, au vu des progrès effectués dans les techniques de compilation et de l'évolution de l'architecture des processeurs, il était nécessaire de trouver une méthode pouvant englober les effets liés aux optimisations lors de la compilation.
Pour cela, nous avons utilisé un découpage statique du programme (program slicing). Ce concept a été introduit en 1979 par Weiser [121], et n'a cessé d'évoluer par l'ajout de nombreuses représentations [86, 104]. Il y a notamment le *Program Dependence Graph* (PDG), par Ottenstein et al. [53], qui permet de connaître les relations de dépendance

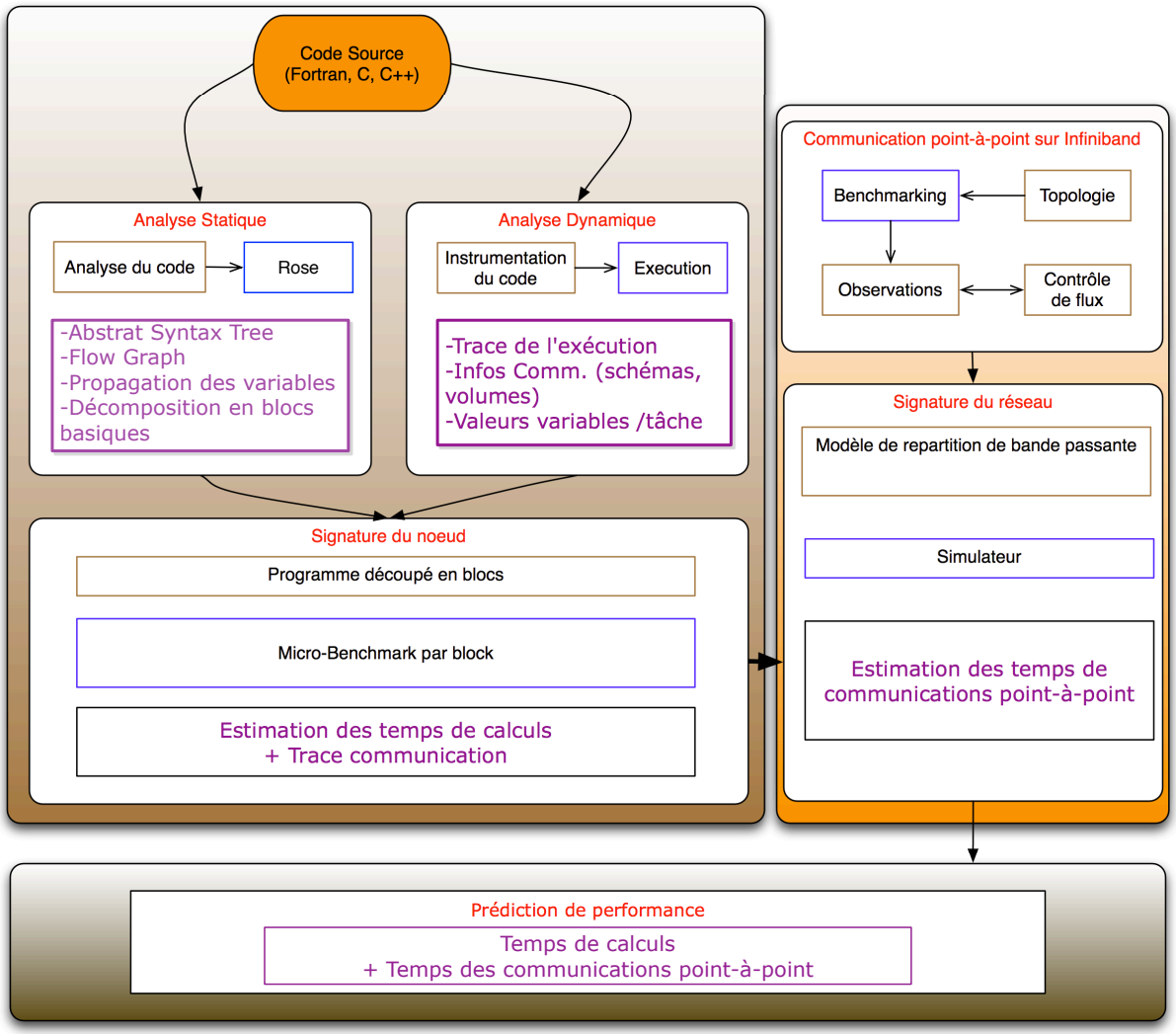



FIGURE 1.1 – Framework général

entre les valeurs suivant les structures de contrôle, ainsi que le *System Dependence Graph* (SDG), introduit par Horwitz [50], qui permet, en plus, de connaître le contexte avant l'appel de la procédure.
L'utilisation du découpage permet d'avoir le programme parallèle sous forme de blocs, et de connaître l'ordre de ces blocs ainsi que leurs dépendances via les différentes représentations existantes. Une fois ces blocs obtenus, différentes mesures de temps sont effectuées sur chacun de ces blocs afin d'avoir l'estimation du temps de leurs exécutions. L'utilisation de cette technique à plusieurs avantages :

– Les représentations étant peu différentes entre les langages de programmation utilisés dans le calcul haute performance, il est possible d'étendre la mise en œuvre

effectuée pour les programmes en langage C aux langages tels que le C++ et Fortran.
- Les grappes de calcul étant généralement homogène, il est possible de caractériser le temps d'exécution de chaque tâches sur la plate-forme cible en utilisant uniquement un seul nœud.
- Le temps requis pour obtenir la prédiction est relativement faible par rapport à l'exécution réelle de l'application.

Lors de l'analyse statique, si toutes les variables essentielles pour l'estimation du temps de calcul ne sont pas obtenues, il est nécessaire de s'appuyer sur une exécution du programme parallèle afin de les obtenir. Néanmoins, ces variables n'étant pas dépendantes de l'architecture, l'exécution pourra être effectuée sur n'importe quelle grappe de calcul. Les résultats obtenus, via cette méthode, varient en fonction des compilateurs et des options de compilation, mais les prédictions obtenues ont une erreur faible (entre 5 et 15%) par rapport aux valeurs mesurées lors de l'exécution réelle.

**Modélisation des communications concurrentes sur *InfiniBand***

L'augmentation des cœurs au sein des nœuds a accru la nécessité de la prise en compte des communications concurrentes et de la congestion.
Adve [117] a été l'un des premiers à soumettre un modèle considérant les effets de la congestion. Dans son modèle, le temps total d'une exécution était décomposé en quatre parties :

$$t_{total} = t_{computation} + t_{communication} + t_{resource\ contention} + t_{synchronisation}$$

Mais la nature non-déterministe de la congestion ainsi que la difficulté à déterminer les retards moyens de synchronisation rendait ce modèle difficile à mettre en place. Clement et Steed [18, 101] avait proposé l'utilisation d'un facteur de congestion $\gamma$ qui améliore le modèle d'Hockney [81], permettant d'exprimer l'impact de la congestion sur des réseaux partagés. Le temps d'une communication étant alors défini par :

$$T = l + \frac{b\gamma}{W}$$

où $l$ est la latence, $b$ la taille du message, $W$ la bande passante et $\gamma$ représente le nombre de processus. L'obtention du temps de communications soumises à la concurrence était alors obtenu de façon simple, tout en donnant des résultats assez précis, mais il était nécessaire que tout les processus communiquent au même moment pour cela.
Martinasso [67] repris l'idée du modèle de Clement, mais en tenant compte de l'aspect dynamique lié à l'évolution des schémas de communication au cours de l'exécution d'un programme parallèle. Ces travaux ont permis de définir un modèle de communications concurrentes pour le réseau *Gigabit Ethernet* et *Myrinet*. En nous basant sur le protocole

de Martinasso, ainsi que sur l'expérimentation et l'analyse des communications concurrentes sur réseau *InfiniBand*, nous avons développé un modèle permettant d'estimer les temps des communications point-à-point, en tenant compte de cette concurrence et cela de façon dynamique.

**Amélioration d'un environnement d'estimation de performance d'application parallèle**

Pour l'estimation des temps de communication, le simulateur de Martinasso [67] a été modifié, permettant maintenant d'estimer les temps des communications point-à-point, en tenant compte des effets de la concurrence, sur le réseau *InfiniBand* en plus des réseaux *Gigabit Ethernet* et *Myrinet*.

Les travaux présentés ont fait l'objet d'une conférence internationale [114], d'une conférence nationale [113] et d'un internship de 3 mois au *National Institute of Informatics (NII)* de Tokyo.

## 1.4 Organisation du manuscrit

La première partie introduit d'abord l'environnement du calcul haute performance avant de présenter les différentes approches existantes pour l'estimation de performance. Dans le premier chapitre, une introduction sur les différents composants de la grappe de calcul est effectuée en décrivant d'abord les éléments du nœud, suivi d'une présentation du réseau *InfiniBand*, puis d'une description des programmes parallèles ainsi que des outils de benchmarking utilisés. Le deuxième chapitre s'intéresse aux différents travaux de recherche réalisés dans le domaine de la prédiction de performance, afin de positionner nos travaux par rapport aux modèles existants.

La deuxième partie présente notre approche basée sur l'expérimentation et l'observation des applications parallèles. Le chapitre 4 présente la modélisation des phases de calcul. Cette modélisation s'appuyant à la fois sur une analyse statique du code source et un processus de micro-benchmarking. Le chapitre 5 s'intéresse à la modélisation de la concurrence sur le réseau *InfiniBand* et l'effet des composants réseaux sur ce modèle.

Enfin, la troisième partie traite de la validation de la méthode mise en place et de l'étude de la précision du modèle de prédiction des temps de communication. Cette précision sera d'abord montrée pour notre modèle réseau via, d'une part, un ensemble de graphes de communications synthétiques et, d'autre part, une application parallèle présente dans SpecMPI [75, 100] appelée *Socorro*.

# I

# L’environnement du calcul haute performance

*Cette première partie présente le calcul haute performance d'un point de vue des performances et de l'évaluation de ces performances.*
*Le premier chapitre décrit les composants de la grappe de calcul, les applications parallèles ainsi que la description des différents benchmarks utilisés pour définir les performances du réseau ou d'une grappe de calcul.*
*Le second chapitre est consacré aux différentes techniques utilisées pour prédire les performances des applications parallèles. L'étude des différentes méthodes existantes permettra de nous positionner et de présenter notre approche.*

# Le calcul haute performance 2

## 2.1 Introduction

Les progrés technologiques permettent aux utilisateurs de s'attaquer à des problèmes de complexité croissante. Par exemple, la simulation numérique d'applications scientifiques crée une demande insatiable en ressource de calculs. En effet, les scientifiques s'efforcent de résoudre des problèmes de plus en plus vastes tout en requérant une plus grande précision, le tout dans un délai plus court. Un exemple de ce type de problèmes est la prévision météorologique. Il s'agit de diviser l'atmosphère en un maillage en trois dimensions, où chaque cellule représente l'état d'une partie de l'atmosphère qui varie avec le temps. Les caractéristiques temporelles et spatiales de la température peuvent être simulées par itération en réalisant un certain nombre de calculs (dérivées de modèles théoriques) sur la base des états de chaque cellule et de ses voisins. La qualité du calcul des prévisions est déterminée par la taille des cellules divisant l'atmosphère ainsi que par la durée de chaque itération. En conséquence, la quantité de calcul nécessaire pour avoir une prévision précise est énorme, de plus ces opérations doivent être achevées dans un court délai, afin d'être utile.
Un moyen d'accroître la puissance de calcul disponible pour résoudre ce genre de problème est l'utilisation d'une grappe de calcul. Une grappe de calcul est composée de nœuds reliés par un réseau de communication. Si un problème peut être subdivisé en $n$ sous problèmes, un programme parallèle peut être écrit pour résoudre simultanément ces sous problèmes sur $n$ unités de calcul. Idéalement, cela réduirait le temps de $\frac{1}{n}$ par rapport à une exécution sur une seule unité. Mais ceci est rarement le cas dans la pratique, pour deux raisons. Premièrement, beaucoup de problèmes contiennent un nombre important de calculs qui ne peuvent être parallélisés facilement ou complètement. Cela se traduit par une non utilisation d'une partie de la puissance de calcul disponible. Deuxièmement, beaucoup de problèmes nécessitent un nombre important de communications et de synchronisations entre les sous problèmes, introduisant des délais d'attentes. Pour illustrer ces deux points, reprenons notre exemple de prévision météorologique. Tout d'abord, il y a la lecture des états initiaux de chaque cellule ainsi que la fusion des résultats à la fin de la simulation. Les communications entre les cellules (par conséquent les unités de

calcul) sont longues et fréquentes, car le calcul de l'état de la cellule dépend de sa valeur mais aussi de celles de ces voisins, et cela, pour chacune des nombreuses itérations. La synchronisation, quant à elle, est également nécessaire à la fin de chaque itération pour s'assurer de la cohérence des états.
Le découpage du problème n'est pas le seul moyen d'optimiser les performances d'un programmes. Le choix du processeur, du langage de programmation, du compilateur mais aussi du réseau influencent grandement le temps d'exécution d'une application sur une grappe de calcul. Au travers de ce chapitre, il sera présenté l'ensemble des éléments qui composent le calcul haute performance et qui influent sur ces performances. Dans un premier temps, nous verrons le nœuds de calcul multiprocesseurs (section 2.2) avec les principaux composants logiciels et matériels. Cela sera suivi ensuite (section 2.3) par une présentation du réseau haute performance utilisé dans le cadre de cette thèse, à savoir le réseau *InfiniBand*. Puis, les différents outils de mesure de performance des communications *MPI* (section 2.4) seront décrit, avant de finir par une présentation des programmes parallèles (section 2.5), ainsi que de certains benchmarks usuels (section 2.6).

## 2.2 Nœuds de calcul multi-coeurs

Une grappe est composée de nœuds reliés par un réseau. Les performances du nœud sont donc primordiaux pour déterminer la puissance d'une grappe. A l'intérieur du nœud, plusieurs éléments matériel et logiciel influent sur ces performances. Cette section décrira d'abord le processeur ainsi que les éléments impactant ou permettant de voir ces performances avant de s'intéresser à l'architecture mémoire.

### 2.2.1 L'environnement du processeur

Le processeur est composé d'une Unité Arithmétique et Logique (UAL) et d'une unité de contrôle. L'UAL peut avoir différentes configurations, elle peut être très simple mais aussi très complexe afin d'effectuer des opérations difficiles. Le rôle principal de l'UAL consiste à effectuer les opérations données en suivant les instructions qui peuvent être, par exemple, d'ordre arithmétique (addition, soustraction, ...) ou logique (test de supériorité, égalité, ...). En plus de l'UAL, le processeur dispose de registres afin de stocker les opérandes et les résultats intermédiaires du calcul, et de maintenir les informations permettant de déterminer l'état du calcul. Le processeur dispose aussi de plusieurs niveaux de cache mémoire pour stocker les informations.
Les efforts des constructeurs se portent maintenant vers des architectures multi-cœurs qui consistent à contenir plusieurs unités de traitement, appelées cœurs, sur une même puce (ou *die*).
L'avantage des architectures multi-coeurs est de pouvoir proposer, pour une même fréquence, une puissance de calcul plus importante. Mais ce gain n'est pas toujours visible

et de nombreux nouveaux problèmes se posent.
Tout d'abord, l'augmentation du nombre de cœurs crée de nombreux goulots d'étranglement. En effet, des accès concurrents peuvent avoir lieu au niveau de la mémoire, réduisant à la fois la bande passante, et augmentant aussi la latence. D'autre part, cela crée une certaine instabilité des performances, de par le fait qu'il y a de nombreuses interactions entre les différents composants et que des phénomènes de faux partage peuvent apparaître.

Cette partie s'intéressera d'abord aux éléments conditionnant les performances du processeur. Elle débutera par la présentation de la mémoire cache et de la technique principalement utilisée pour le problème de la cohérence de cache. Elle continuera sur l'étude des registres spéciaux appelés les compteurs de performances avant de présenter le rôle du compilateur. Pour conclure, l'architecture mémoire présente dans les grappes de calcul sera introduite.

**La mémoire cache**

En plus des registres, le processeur possède donc plusieurs niveaux de mémoire cache. Le rôle de la mémoire cache est de permettre au processeur de disposer plus rapidement des instructions et données redondantes. Bien que permettant un gain de rapidité pour l'accès aux données, cette mémoire est de capacité limitée, principalement du fait qu'elle est relativement coûteuse. Il y a, généralement, plusieurs niveaux de cache au sein d'un processeur. Le cache L1, qui est le plus petit, est partitionné en deux parties : une pour les instructions et une pour les données. Le cache L2, et parfois L3, sont souvent uniquement utilisés pour contenir les données. Le temps d'accès aux caches est mesuré en cycles processeur et est plus ou moins grand en fonction de la proximité et de la taille des caches. A titre d'exemple, sur le dernier processeur d'Intel, le nehalem, l'accès au cache L1 prend 4 cycles, alors qu'il faut 10 cycles pour l'accès au L2 et 40 cycles pour le L3.
Si le processeur souhaite écrire ou accéder à une donnée en mémoire, il vérifie d'abord si cette donnée est dans le cache via le contrôleur du cache. S'il la trouve, il effectuera son opération dans le cache, sinon, il chargera la donnée dans le cache en la recopiant depuis la mémoire principale. Cette opération peut amener des incohérences s'il y a plusieurs processeurs ou cœurs au sein d'un même nœud, car la donnée chargée dans le cache d'un processeur peut très bien être modifiée en mémoire centrale par un autre processeur ou cœur. C'est le problème de cohérence de cache.

**Le problème de la cohérence de cache** La reproduction des données au travers des caches permet de réduire le temps d'accès à ces données mais aussi la contention liée à l'accès simultané, par plusieurs cœurs, à des données partagées. Néanmoins, du fait que chaque processeur ne voit le contenu de la mémoire qu'au travers de son cache, il est nécessaire de s'assurer que les différents cœurs n'ont pas de valeurs différentes pour le même espace mémoire.

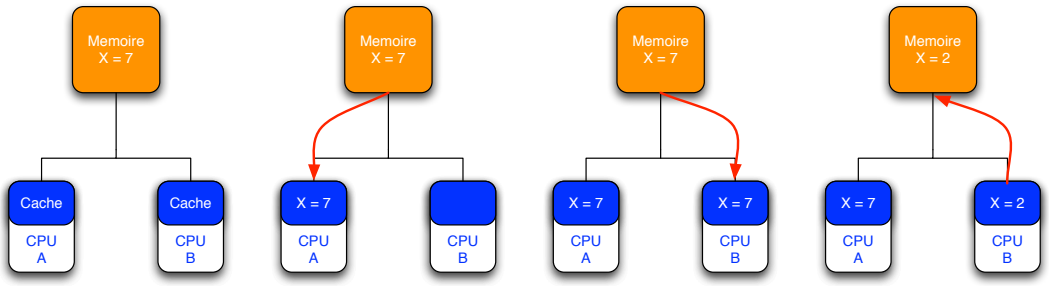



FIGURE 2.1 – Exemple du problème de cohérence de cache

L'exemple, donné par la figure 2.1, permet de mieux comprendre le problème de la cohérence de cache. Deux cœurs différents, A et B, accèdent au même espace mémoire, puis le cœur B modifie celui-ci. A partir de ce moment, le cœur A a une valeur obsolète de la variable contenue dans l'espace mémoire. C'est ce qui est appelé le problème de cohérence de caches.
Il existe plusieurs mécanismes pour résoudre ce problème [41]. Nous nous intéresserons ici uniquement au protocole couramment utilisé dans les systèmes à mémoire partagée (voir section 2.2.2), appelé le "Snooping".
Avec ce mécanisme, chaque contrôleur de cache surveille, via le bus, les données demandées par les processeurs. La meilleure solution pour résoudre le problème de cohérence de cache est de s'assurer qu'un processeur a un accès exclusif à une donnée avant de la modifier. Avant l'écriture, toutes les copies de la donnée contenues dans les caches des autres processeurs sont invalidées. Après cette étape, le processeur effectue l'écriture, modifiant la valeur de la donnée contenue dans la mémoire locale par celle contenue dans son cache. Quand un autre processeur essaye d'accéder à la donnée via son cache, il aura un défaut de cache, car la donnée contenue sera invalidée, le forçant donc à mettre à jour la valeur de la variable.
Si deux processeurs essaient d'écrire simultanément sur le même espace mémoire, un seul pourra le faire. Le cache du processeur "perdant" sera alors invalidé. Il lui faudra alors recevoir une nouvelle copie de la donnée (avec sa valeur mise à jour) avant de faire son écriture.
Du fait que toutes les transactions sont vues par tous les contrôleurs, il est nécessaire de disposer d'une bonne bande passante, car ce mécanisme n'est pas extensible, et plus y aura de processeurs plus les ressources nécessaires en terme de bande passante seront importantes pour assurer des échanges rapides.

### Les compteurs de performances

Les compteurs de performances sont des registres spéciaux à l'intérieur du processeur qui permettent de mesurer l'activité de celui-ci. Le nombre de ces compteurs ne cessent

d'augmenter avec le temps, ainsi le *Pentium III* en possédait 2 alors que le *Pentium 4* en possédait 18. L'avantage des compteurs hardwares est que leur utilisation directe est très peu intrusive, du fait qu'il faut moins de 80 cycles processeur pour y accéder. Ces compteurs nous donnent des informations très précises, mais de bas niveau, sur les performances des applications. Ils peuvent permettre de connaître le nombre de tops d'horloge nécessaire durant une mesure, mais nous renseignent aussi sur l'accès aux données. Ainsi, les compteurs tel que le L2_MISS ou L3_MISS permettent de savoir si les données auxquelles accède le programme sont dans le cache du processeur ou non. Une grande valeur de ces compteurs indiquerait que les données ne sont pas présentes dans le cache et donc que des optimisations du programme pourrait être effectuées, afin d'améliorer les performances de celui-ci.
Il existe de nombreux outils et bibliothèques pour accéder aux compteurs de performances, dans le cadre des travaux réalisé ici, l'outil *PAPI* sera utilisé. Ce dernier est présenté en Annexe A.

#### Compilateurs

Le compilateur permet de traduire en langage machine le programme écrit en langage de programmation. Cette traduction peut être générique pour un type de processeur ($i386$, $x86_64$) ou bien être spécifique à une architecture donnée. Il en résulte que les performances d'une application sur un nœud reposent aussi sur les optimisations apportées par le compilateur. Afin d'exploiter au mieux le potentiel de ces processeurs, Intel a développé ses propres compilateurs pour les langages C, C ++ et Fortran. Chaque nouvelle version du compilateur *Intel* apporte des optimisations pour les nouvelles architectures existantes, mais aussi le support de nouvelles instructions comme le *SSE4* pour la version 10.0 ou le *C++0x* pour la version 11.0.

### 2.2.2 L'architecture mémoire

Les ordinateurs parallèles [24, 33] sont souvent classés en fonction de leurs architectures mémoires, c'est-à-dire que l'on considère d'un côté, les machines à mémoire partagée, et de l'autre, les machines à mémoire distribuée. La différence entre ces deux types de machine parallèle se situe dans la façon dont le processeur et la mémoire communiquent entre eux.
Dans les machines à mémoire partagée, tous les cœurs peuvent accéder à la mémoire au travers d'un espace d'adressage global. Les systèmes à mémoire partagée peuvent être subdivisés en deux parties, avec d'un côté les architectures à accès uniforme en mémoire *"Uniform Memory Architectures"* (UMAs) connues aussi sous le nom de *Symmetric Multi-Processors* (SMP) et de l'autre les architectures à accès non uniforme *"Non-Uniform Memory Architectures"* (NUMAs) (voir figure 2.2). La distinction entre ces deux classes est que les processeurs en UMAs peuvent accéder à la mémoire en temps constant,

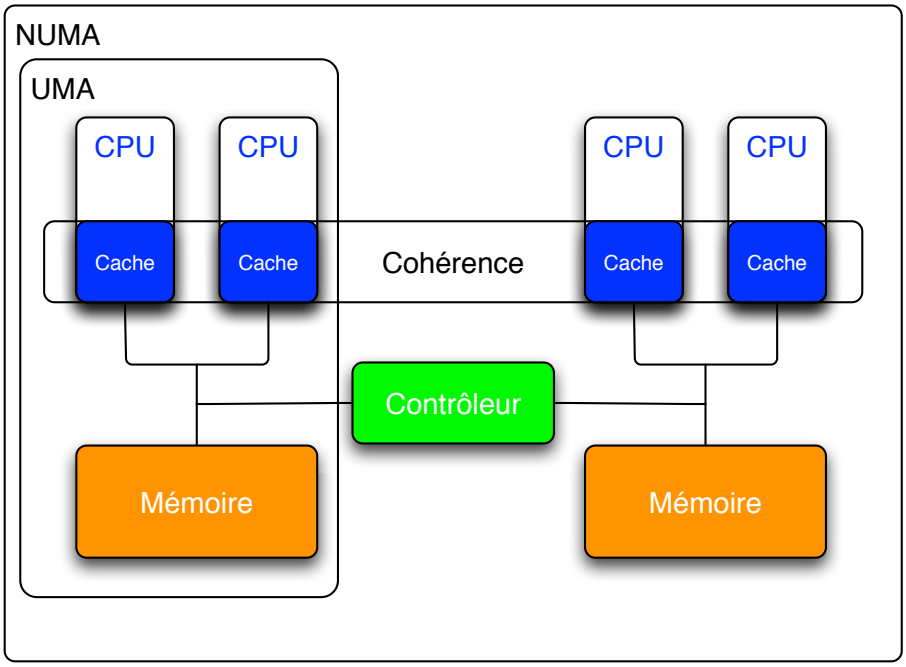



FIGURE 2.2 – Architecture UMA, NUMA et cohérence de cache

contrairement aux processeurs en NUMAs où le temps d'accès diffère en fonction de la distance avec la mémoire. Il est plus facile de connecter de façon efficace un grand nombre de processeurs en utilisant la technique NUMA. En effet, les machines UMA actuelles ont entre 2 et 64 coeurs alors que les machines NUMA actuelles dépassent les 1024 coeurs. Presque toutes les implémentations des machines UMA et NUMA ont un système de cohérence de cache, afin de s'assurer que les copies des données stockées dans le cache du processeurs sont synchronisées avec la mémoire globale.
A l'inverse, dans les systèmes à mémoire distribuée, chaque processeur a son propre espace mémoire et communique en envoyant une copie des données de sa mémoire vers celle d'un autre processeur en utilisant un système d'échange de messages. Les systèmes à mémoire distribuée communiquent plus lentement que les systèmes à mémoire partagée, mais permettent de construire des grappes de calcul plus grandes pour un coût financier inférieur.
Actuellement, les constructeurs de grappes de calcul mixent les deux architectures mémoires en utilisant des nœuds SMP (*Symmetric MultiProcessing*) communiquant au travers d'un réseau, afin de profiter du meilleur des deux.

## 2.3 Le réseau haute performance *InfiniBand*

Issu d'une fusion de deux technologies en 1999 (avec d'un côté *Future I/O*, développée par *Compaq*, *IBM* et *Hewlett-Packard*, et de l'autre *Next Generation I/O*, développée par *Intel*, *Microsoft* et *Sun*), la technologie Infiniband a su se faire une place au sein des grappes de calcul. Son succès peut se voir au travers du top 500. En juin 2003, une seule grappe de calcul était sous Infiniband, mais au fur et à mesure des avancées technologiques, son nombre s'est accru au point qu'à l'heure actuelle 181 grappes de calcul (36,2% du classement) utilisent le réseau haute performance *InfiniBand*.
Dans le contexte d'une grappe de calcul, la rapidité du réseau *InfiniBand* permet de réduire les phases de communication et par conséquent, de limiter le temps d'attente des cœurs. Le réseau *InfiniBand* est souvent caractérisé, comme beaucoup d'autres, par deux paramètres : la latence et la bande passante (ou débit). La latence correspond au temps nécessaire à l'envoi et à la réception du premier octet. La bande passante, quant à elle, correspond au débit maximal obtenu via l'envoi de messages de grandes tailles.
Différents niveaux de communication apparaissent avec les architectures multi-cœurs [16]. On parle de communication intra-socket lorsque les communications sont entre deux cœurs appartenant au même processeur, de communication inter-socket lorsque la communication à lieu à l'intérieur du nœud mais entre deux processeurs différents et de communication inter-nœud lorsque la communication a lieu entre deux nœuds différents.
Au travers de cette section, les performances du réseaux *InfiniBand* seront d'abord présentés du point de vue de la couche logicielle accédant directement au matériel, à savoir la libibverbs. L'interface de programmation par passage de messages, appelé *Message Passing Interface* (*MPI*) sera introduit ensuite. La dernière partie concernera la topologie réseau et les algorithmes de routages appliqués au réseau *InfiniBand* dans le cadre du calcul haute performance.

### 2.3.1 La bibliothèque bas niveau *libibverbs*

La bibliothèque *libibverbs* est la bibliothèque de communication bas niveau proposée pour le réseau *InfiniBand*. Elle permet deux modes d'échange, le *send/rev* et le *RDMA*.

#### Send/Recv

La sémantique *send/recv* est proche de la sémantique *MPI* de passage de messages. Elle demande aux deux parties réalisant l'échange de faire une requête sur la carte Infiniband : Un *Send* (envoi) et un *Receive* (réception). Elle implique un travail de la carte Infiniband assez important qui doit faire la correspondance entre le *Send* et le *Receive*. Ainsi, la carte Infiniband prend en charge le matching, travail consistant à faire correspondre les requêtes d'émission (sends) et de réception (receives). Cela a pour avantage de délester le processeur d'une partie de la réception du message dans *MPI*, mais induit une

latence supplémentaire, la carte fonctionnant à une fréquence inférieure au processeur.

#### RDMA

La sémantique *RDMA* est une sémantique d'écriture distante. Une fois la zone enregistrée d'un côté, l'autre côté peut lire et écrire directement dans la zone. Cette sémantique implique des mécanismes plus simples de la part de la carte Infiniband, mais demande une gestion de la mémoire, des protections et des verrous de la part du CPU local.

#### Utilisation dans les couches supérieures

*MPI* utilise en général les deux modes de communication. Le mode *RDMA* peut-être utilisé afin de réduire la latence des petits messages ainsi que pour augmenter le débit des grands messages. En effet, si le mode *send/recv* a de bonnes performances, la sémantique *MPI* a des contraintes qui l'obligent à recopier les données dans un buffer intermédiaire, puis à les recopier vers le buffer destination. La bande passante et la corruption du cache s'en trouvent très nettement impactés. Par conséquent, pour les grands messages, le mode *RDMA* est utilisé.

### 2.3.2 Performance de la libibverbs

Il est relativement simple d'étudier les performances de la *libibverbs* dans le cadre des communications point-à-point. En effet, il existe différents outils mis à disposition, avec cette bibliothèque, pour mesurer la latence et le débit au niveau de *libibverbs*. Les expérimentations présentées, dans cette section, consistent à l'utilisation de ces outils pour étudier les performances des différents modes d'échange entre deux nœuds *Novascale* (Cluster BULL) équipés d'une carte *InfiniBand ConnectX QDR*.

#### Latence

La latence des trois opérations élémentaires, présentée par la figure 2.3, a été mesurée avec les outils de mesure *ib_send_lat*, *ib_write_lat* et *ib_read_lat*. L'abscisse de la courbe indiquant la taille des messages et l'ordonnée le temps en microseconde. L'intervalle de confiance n'est pas indiqué car les mesures s'avèrent relativement stable.

Les latences suivent les performances habituelles de la famille des cartes *ConnectX*, bien inférieures aux générations précédentes. Le RDMA Write reste le chemin le plus rapide avec 0.9 microsecondes contre 1.19 microsecondes pour le mode *send/recv*. Avant 256 octets, la latence est particulièrement basse. Ce phénomène est dû à une technique d'inlining des données (données mises avec le *header*). Cette technique est possible jusqu'à 400 octets et permet d'envoyer des données en un seul message. Elle est impossible

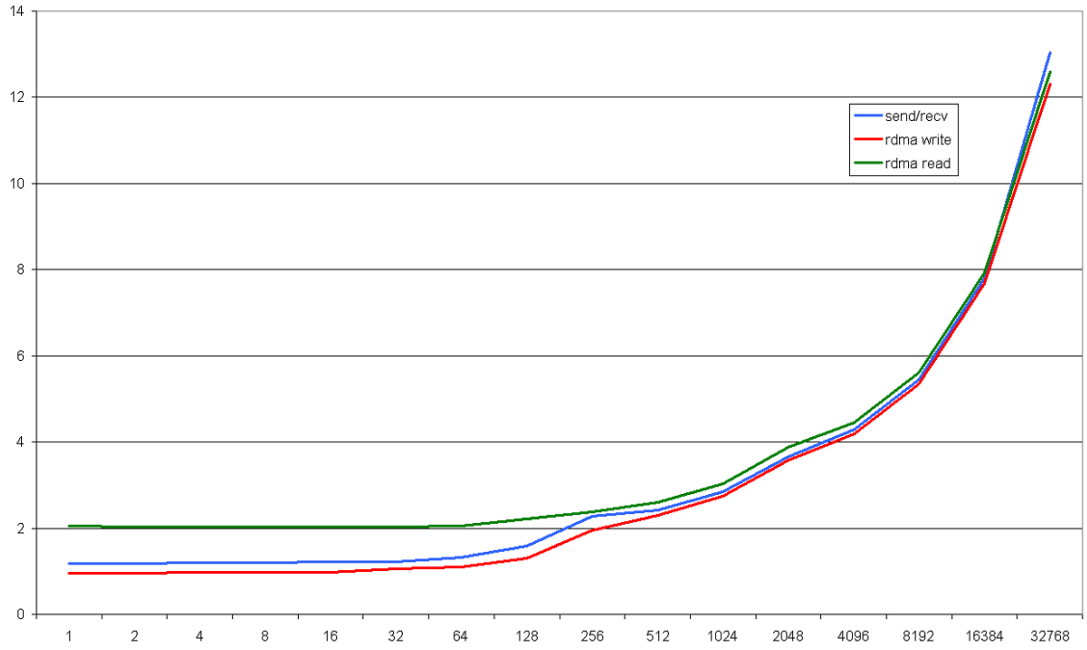



FIGURE 2.3 – Comparaison des latences en microsecondes des opérations élémentaires de la bibliothèque libibverbs

avec l'opération *RDMA Read* qui nécessite un aller-retour, faisant stagner sa latence à 2 microsecondes.

### Débit

L'obtention des débits, présentés dans la figure 2.4, a été obtenu suite à l'exécution des programmes ib_send_bw, ib_write_bw et ib_read_bw. Ces programmes fournissent la bande passante unidirectionnelle et bidirectionnelle des trois opérations élémentaires *send/recv*, *rdma write* et *rdma read*. L'abscisse de la courbe indiquant la taille des messages et l'ordonnée le débit en Mo/s.

On trouve une bande passante unidirectionnelle de 3200 Mo/s et bidirectionnelle de 6400 Mo/s. Il est probable que le bus *PCI Express Gen2* soit le facteur limitant ici. En effet, si l'on rassemble les mesures faites dans le passé avec la technologie Infiniband DDR, on obtient le tableau 2.5.

Il est donc probable qu'une fois encore, le débit de la carte Infiniband QDR soit limité par le bus *PCI Express Gen 2* qui comme son prédécesseur *Gen 1* montre une efficacité de 80% soit 3200 Mo/s. Lorsque la carte *ConnectX* fonctionnera sur un bus *PCI Express*

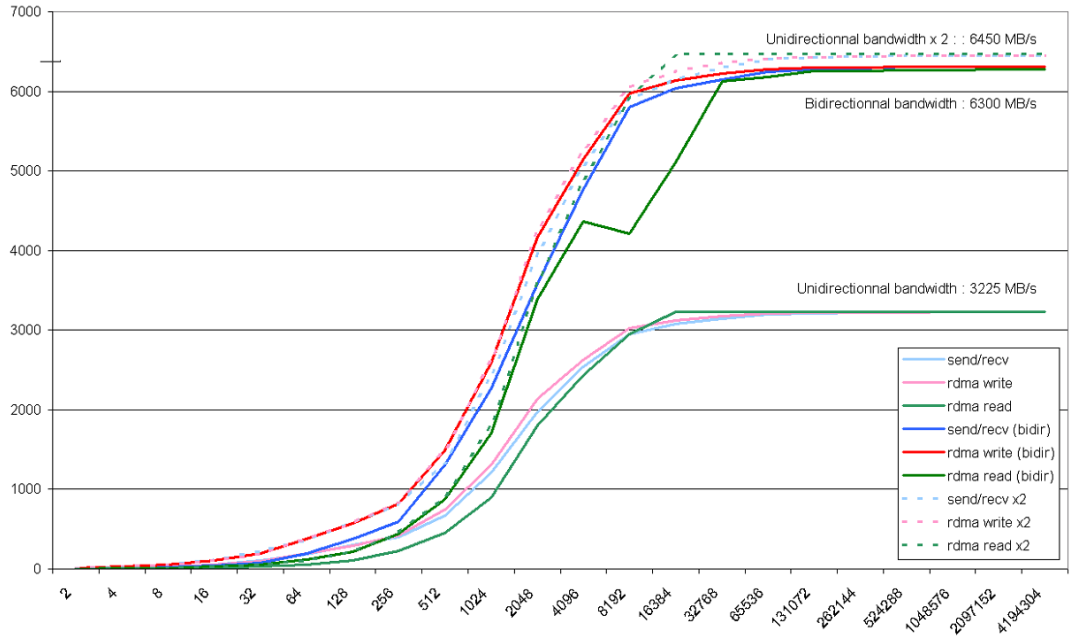



FIGURE 2.4 – Comparaison des débits des opérations élémentaires de la bibliothèque libibverbs

| Carte | PCI Express 8x Gen 1 | PCI Express 8x Gen 2 | PCI Express 8x Gen 3 |
|---|---|---|---|
| Infiniband 4x DDR (2 Go/s théorique) | 1600 Mo/s | 1800 Mo/s | |
| Infiniband 4x QDR (4 Go/s théorique) | | 3200 Mo/s | 3600 Mo/s (extrapolé) |

FIGURE 2.5 – Mesure du débit sur Infiniband

*Gen3* , où lorsqu'une nouvelle génération le permettant arrivera, il sera donc possible que la technologie QDR dépasse 3200 Mo/s par sens.
Comme pour la version *DDR*, on retrouve une infériorité et une instabilité de *RDMA Read*. Les implémentations actuelles de MPI privilégient pour cette raison les appels à *RDMA Write* pour les échanges de grands messages. Le léger surcoût du protocole *send/-recv* se fait sentir sur les petites tailles puis disparaît. La bande passante bidirectionnelle est quasiment égale à la bande passante unidirectionnelle x 2. On observe une perte de seulement 2,3%, ce qui reste très limitée.

### 2.3.3 L'interface de communication *MPI*

Dans les années 80, plusieurs compagnies ont commencé à construire et à vendre des machines parallèles. Typiquement, l'environnement de programmation de ces machines était constitué d'un langage de programmation séquentiel (C ou FORTRAN) et d'une bibliothèque de passage de messages afin que les processus (définition en section 2.5) puissent communiquer entre eux. Chaque fabriquant avait sa propre bibliothèque, ce qui signifiait qu'un programme parallèle développé pour un nCUBE/10, par exemple, ne pouvait se compiler et donc s'exécuter sur un Intel iPSC. Les programmeurs devaient donc récrire leurs programmes du fait du manque de portabilité, ce qui a amené, après quelques années, à la nécessité d'un standard de bibliothèque de passage de messages pour les ordinateurs parallèles.
En 1989, la première version de ce standard, écrite par l'*Oak Ridge National Laboratory*, est apparu sous le nom de *PVM* (*Parallel Virtual Machine*). *PVM* permettait l'exécution de programme parallèle dans des environnements hétérogènes. Alors que cette version était utilisée par l'*Oak Ridge National Laboratory* et la communauté universitaire, elle n'était pas encore disponible au public. L'équipe modifiera deux fois la bibliothèque avant de la rendre disponible au public en mars 1993 [32], sous le nom de PVM version 3. PVM est alors devenu très populaire auprès des programmeurs d'applications parallèles.
Durant cette même période en avril 1992, le *Center of Research on Parallel Computing* sponsorisait le *Workshop on Standards for Message Passing in Distributed Memory Environnement*. Ce workshop a attiré plus de 60 personnes représentant plus de 40 organisations, principalement des États Unis et d'Europe. La plupart des vendeurs de machines parallèles ainsi que les chercheurs des universités, laboratoires gouvernementaux et du milieu industriel étaient présents. Ce groupe discuta des caractéristiques de base d'une bibliothèque de passage de messages et créa un groupe de travail pour continuer ce processus de standardisation, appelé le *Message Passing Interface Forum*. En novembre 1992, une première esquisse vu le jour. Ce forum s'est réuni, à de multiple reprises, entre novembre 1992 et avril 1994, pour débattre et raffiner ce premier jet. Au lieu de simplement utiliser une bibliothèque par passage de message telle que *PVM* ou une bibliothèque propriétaire, le *MPI Forum* [70] essaya de prendre et de choisir le meilleur de chaque bibliothèque. La version 1.0 de ce standard, appelé communément *Message Passing Interface* (*MPI*), apparu en mai 1994. Depuis, des travaux ont continué pour améliorer ce standard, en particulier sur les entrées/sorties parallèles et sur l'implémentation en FORTRAN 90 et C++. Une deuxième version du standard, appelée *MPI*-2, a été adoptée en 1997.
*MPI* est donc une interface de programmation (API) de communication, portable, permettant l'échange de données via une série d'appel de fonctions. Ces échanges pouvant se faire soit via des communications point-à-point soit via des communications collectives. *MPI* est très rapidement devenu populaire et est maintenant largement utilisé dans le calcul haute performance. Néanmoins, la spécification *MPI* ne fait que définir une

norme avec des recommandations. Les choix de l'implantation, comme celui des algorithmes pour les communications collectives, par exemple, restent libres, amenant ainsi différentes mise en œuvre de *MPI* libres ou propriétaires.
Dans la suite de cette partie, les performances de l'implantation *MPIBULL 2* seront évaluées sur carte *ConnectX QDR*.

#### Présentation de *MPIBULL2*

La bibliothèque de passage de messages *MPIBULL 2* est une bibliothèque répondant au standard *MPI*-2 et développé au sein de la société BULL afin d'optimiser les performances de ces grappes de calcul *Novascale et Bullx clusters*. Elle supporte différents types de réseaux haute performance tel que *Gigabit Ethernet*, *Quadrics* ou *InfiniBand*. Pour la partie *InfiniBand*, cette bibliothèque s'est appuyée d'abord sur la bibliothèque *MVAPICH* développée par le Network Lab de l'Ohio State University, puis courant 2009, sur la bibliothèque *OpenMPI* développée par un consortium de 16 membres qui sont soit des organismes de recherche (INRIA, Los Alamos National Laboratory, ...) soit des industriels (CISCO, IBM, ...).

### 2.3.4 Performance de la bibliothèque *MPI*

Pour évaluer les performances de la bibliothèque *MPI MPIBULL 2* sur le réseau *InfiniBand*, cette partie se concentrera sur l'étude de la latence et de la bande passante.

#### Latence

La figure 2.6 montre la latence *MPI* qui a été mesurée avec le benchmark *IMB* (présenté en section 2.4.1) en mode *pingpong*. Les courbes de la *libibverbs* sont rappelées pour comparaison. L'abcisse de la courbe représente la taille des messages et l'ordonnée le temps en microseconde.
*MPI* ajoute 0.3 microsecondes de latence et arrive donc, en utilisant le RDMA, à 1.2 microsecondes. On retrouve le décrochement dû à l'inlining à 128 octets au lieu de 256.

#### Débit

La figure 2.7 permet de voir les débits qui ont été mesurés au niveau *MPI* avec l'application *IMB* en mode *ping-pong* (bande passante unidirectionnelle) et *sendrecv* (bande passante bidirectionnelle). Pour comparaison, les chiffres équivalents au niveau inférieur (libibverbs) ont été rapportés sur le graphique. Les chiffres titrés *RDMA* sont ceux du RDMA Write, le seul utilisé par *MPI*.

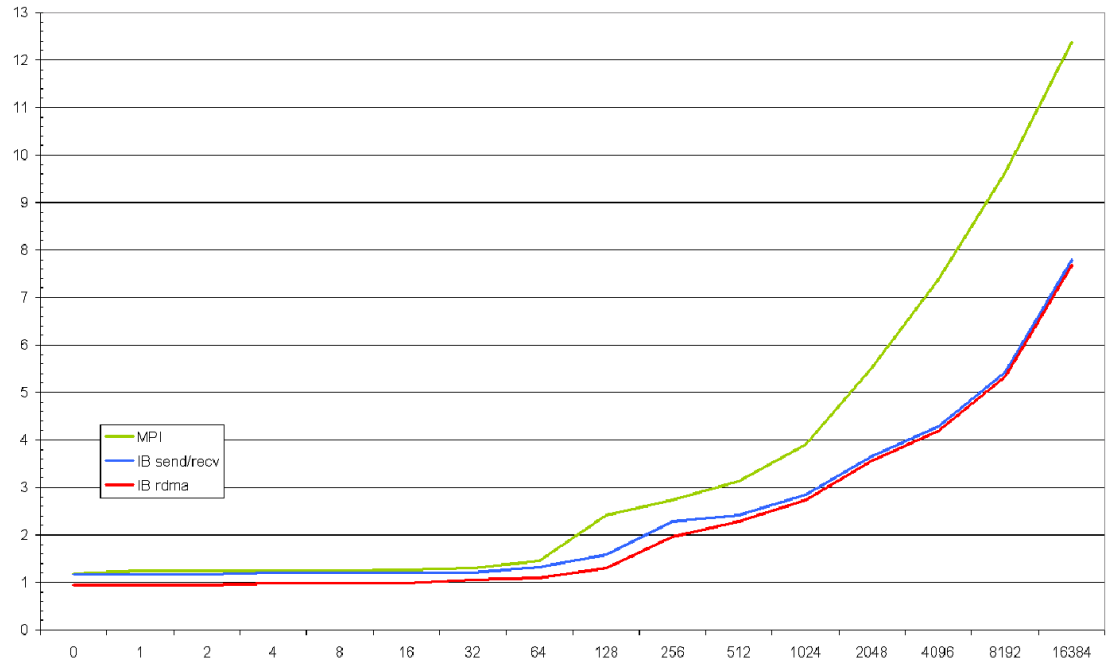



FIGURE 2.6 – Étude de la latence au niveau *MPI*

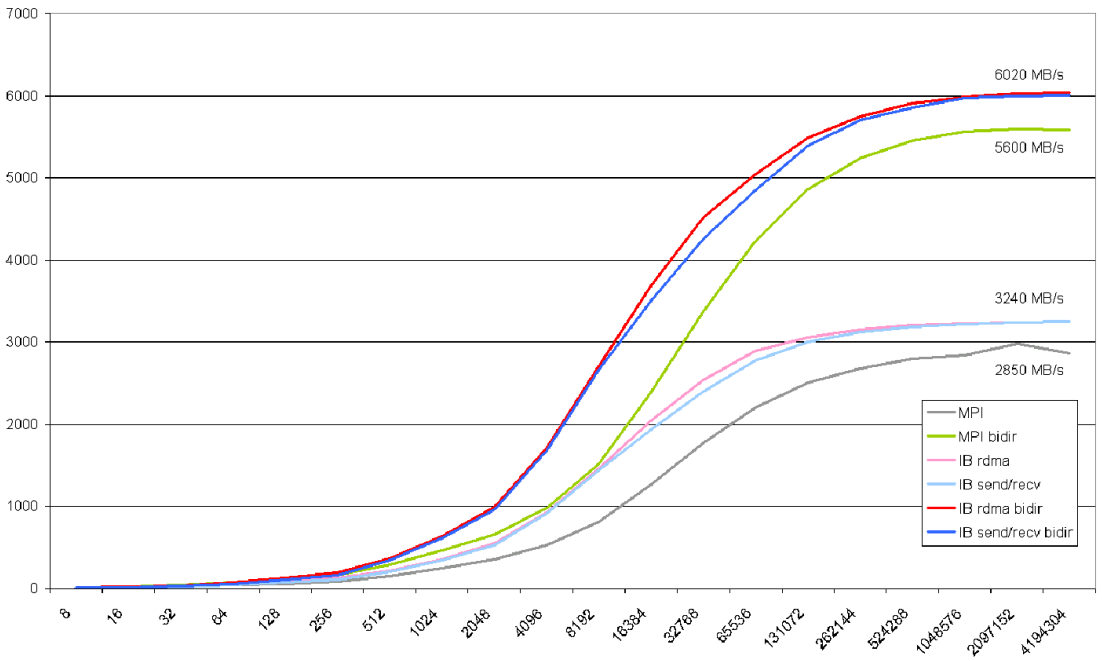



FIGURE 2.7 – Étude du débit niveau *MPI*

Le débit *MPI* souffre d'une pénalité allant de 7% pour les grands messages à 25% pour les messages de taille moyenne. Cette différence est importante et peut s'expliquer ainsi :

- la vitesse de transfert en *QDR* est tellement importante que le temps de transmission d'un paquet de 16k est de seulement 7 microsecondes. Un surcoût de 4 microsecondes suffit donc à impacter fortement les performances. Ce surcoût aurait fait peu de différence avec les techniques précédentes mais l'arrivée du *QDR* met en évidence la lenteur d'autres composants
- les benchmarks étant différents, la méthode utilisée pour effectuer la mesure peut influencer sur le résultat.

### 2.3.5 La topologie réseau et le routage

Différentes topologies avec des propriétés particulières ont été proposées pour être utilisée pour les grappes de calcul : les arbres (trees) [14, 61], les réseaux de Benes (Benes Network) [7], les réseaux clos (Clos Network) [19] et beaucoup d'autres. Un aperçu complet est disponible dans le livre de Leighton [60].
La topologie utilisée dans le cadre de notre recherche est la topologie *Fat-Tree*) [61]. Elle sera donc décrite avant d'aborder les algorithmes de routage sur *InfiniBand* et le routage statique.

**Topologie Fat-Tree** La topologie *Fat-Tree*, appelé aussi CBB pour *Constant Bisectional Bandwidth*, a été proposé en 1985 par Charles Leiserson [62]. Le but étant de proposer un réseau non-bloquant de commutateurs, tout en ayant un nombre minimal de commutateurs. L'avantage du *Fat-Tree* est de proposer une latence constante et une bande passante bidirectionnelle linéaire avec le nombre de nœuds.

**Les algorithmes de routage** Le réseau *InfiniBand* utilise une table de routage statique distribuée. Chaque commutateur possède une table de correspondance (appelé *Linear Forwading Table*) qui définit par quel port doit passer un message pour arriver à sa destination. Au démarrage ou lors d'une modification de la topologie, le *Subnet Manager* (SM) découvre la topologie, calcule la table de correspondance pour chacun des commutateurs, et la fournit à ces derniers. Il existe actuellement cinq algorithmes de routage différents : MINHOP, UPDN, FTREE, DOR et LASH. Il est aussi possible de charger une table de routage provenant d'un fichier.

**MINHOP** essaie de trouver le plus court chemin entre tous les nœuds et d'équilibrer le nombre de chemins pour chaque lien. Néanmoins cette méthode peut amener à des cycles causant des dépendances circulaires [23] pouvant engendrer des interblocages sur le réseau.

**UPDN** utilise l'algorithme de routage Up/Down [92] qui permet d'éviter les dépendances circulaires en réduisant le nombre de chemins possibles sur le réseau [89].

**FTREE** implémente un algorithme de routage optimisé pour les réseaux *fat trees*. Il est sans interblocage mais nécessite une topologie *Fat-Tree* [115].

**DOR (Dimension Order Routing)** essaie de déterminer le plus court chemin pour les réseaux de type *k-ary n-cube* (Torus, Hypercube) mais peut créer des dépendances circulaires.

**LASH (Layered Shortest Path)** [96] utilise des liens virtuels pour casser les dépendances cycliques créées par l'algorithme DOR.

Dans le cadre de ce document, l'algorithme de routage utilisé sera FTREE.

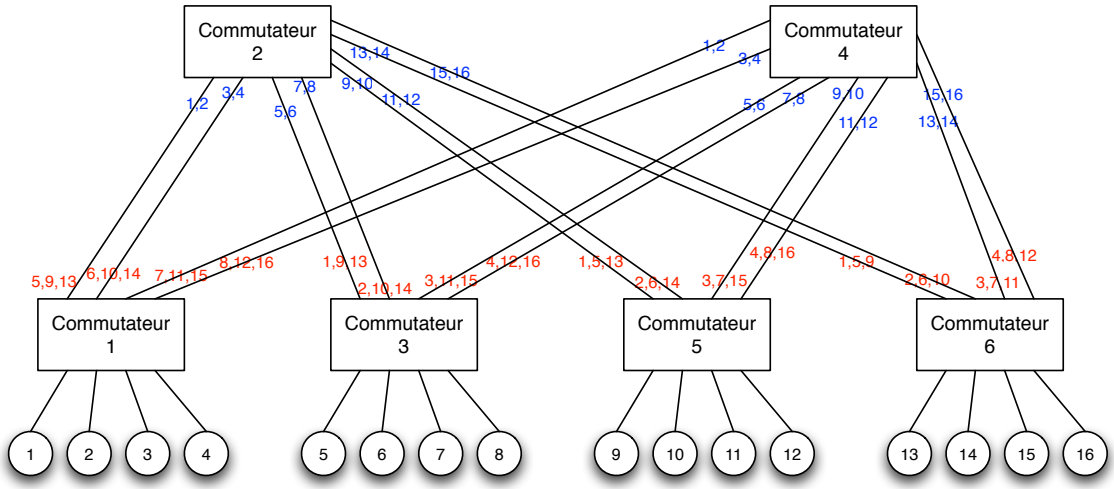



FIGURE 2.8 – Exemple de routage statique InfiniBand sur un *Fat-Tree*

**Le routage sous *InfiniBand*** Contrairement à un réseau haute performance tel que *Myrinet* ou *Quadrics* qui propose un routage dynamique (adaptatif) permettant de réduire le temps de parcours des messages, le routage sur *InfiniBand* est déterministe (statique). Cela signifie que pour aller d'un nœud A à un nœud B, les messages passeront toujours par les mêmes liens et commutateurs quelle que soient leur charge. Pour *InfiniBand*, le routage est basé sur la destination, ce qui signifie que lorsqu'un paquet est dans un nœud ou un commutateur, le reste du chemin à parcourir sera déterminé par sa destination. L'avantage du routage statique est qu'il est facilement implémentable et distribuable. Malheureusement, comme nous le verrons dans la deuxième partie de ce document (chapitre 5), cela peut amener à des pertes de performance.
Pour illustrer cela, prenons l'exemple d'une grappe de calcul composée de 16 nœuds avec 5 commutateurs en *Fat-Tree* [49]. Si l'on applique l'algorithme de routage adéquat, nous

obtenons un routage tel qu'il est décrit dans la figure 2.8. Pour le premier commutateur, tous les messages ascendants s'adressant aux nœuds 5, 9 ou 13 (couleur rouge) devront passer par le même port de ce commutateur, il en est de même pour les messages s'adressant aux nœuds 6, 10, 14 ou 7, 11, 15 ou encore 8,12,16. Par contre, pour les messages descendants (couleur bleue), nous pouvons voir que uniquement deux destinataires devront se partager les liens. Si l'on considère un message par son nœud source $s$ et son nœud destination $d$, nous pouvons faire des couples $(s, d)$. Du fait du routage statique, si nous avons au même moment (1,5), (2,9) et (3,13), ils devront se partager le même lien physique amenant une augmentation du temps de transmission du message. (1,5), (2,10), (3,15) sera un schéma de communication similaire qui donnera de meilleures performances. Comme nous le verrons dans le chapitre 5, les contentions au niveau du commutateur influencent la bande passante du réseau.

## 2.4 Les outils de benchmarking *MPI*

Il existe un grand nombre d'outils pour estimer le temps de communications des appels *MPI*, certains des outils présentés ici ne sont plus maintenus, mais restent fonctionnels et originaux de par la méthode utilisée pour mesurer les appels, ou la présentation des résultats.

### 2.4.1 IMB

A l'origine appelé *PMB* [76] pour *Pallas MPI Benchmark*, et développé par la compagnie allemande Pallas, le logiciel s'est ensuite appelé *IMB* (*Intel MPI Benchmark*), suite au rachat de la compagnie par Intel en 2003. *IMB* couvre l'ensemble des appels de transfert de données *MPI*. Il est à la fois simple d'utilisation et bien documenté, ce qui en fait un outil de mesure particulièrement apprécié et utilisé. Ce benchmark dispose de trois modes de test, les classiques communications point-à-point entre deux processus, les collectives ainsi que les *MULTI*. En mode *MULTI*, la mesure s'effectue alors que des communications de type identique et concurrentes s'exécutent sur la machine. Toutefois, la précision des observations d'*IMB* est entravée par une méthode de mesure inadéquate. En effet, *IMB* utilise l'appel à la fonction *MPI_WTIME* pour déterminer le temps d'exécution des opérations *MPI*. L'appel à *MPI_WTIME* comporte principalement deux désavantages.
Tout d'abord, la précision d'un *MPI_WTIME* n'est pas souvent assez fine comparé à la durée d'une communication *MPI*. *MPI_WTIME* renvoie souvent vers une fonction du type *gettimeofday*. Il faut savoir qu'une telle fonction est très perturbatrice au niveau du système, car celle-ci requiert à peu près 1700 cycles processeur pour être effectuée alors qu'un appel PAPI (voir Annexe A) requiert autour de 150 cycles et qu'un appel direct au compteur hardware du processeur réduit cela à moins de 80 cycles. Pour compenser cette imprécision, dans les communications point-à-point, *IMB* mesure le temps d'exécution

d'une opération *MPI* en la répétant un grand nombre de fois au sein d'une boucle et en calculant une valeur moyenne. Le message, une fois reçu, est donc renvoyé à la source par la destination (principe de l'aller-retour). Le problème de cette approche est qu'une utilisation, même partielle, de la grappe de calcul lors de la mesure peut fausser le résultat obtenu, du fait de possible contention au niveau du commutateur.
Ensuite, même si l'appel à *MPI_WTIME* était suffisamment précis, le problème de synchronisation d'horloge entre les processus communiquant existerait toujours. En effet, *IMB* ne dispose d'aucun processus de synchronisation entre les horloges des nœuds dans la grappe de calcul. Il est donc impossible, avec *IMB*, d'examiner précisément le temps d'un appel *MPI* ou de savoir comment les messages interagissent entre eux. De ce fait, *IMB* n'est pas capable de mesurer la variabilité des performances qui peuvent survenir lors des mesures.
Malgré ces défauts, *IMB* reste l'outil le plus utilisé pour mesurer le temps des communications *MPI* dans le milieu industriel. Nous utiliserons plus tard, en section 5.4.2.1, le mode *MULTI* d'*IMB* pour mesurer l'impact de la charge sur la latence des cartes *InfiniBand*.

### 2.4.2 MPBench

*MPBench* [73] est un outil permettant de mesurer les performances d'un réseau via huit différents appels *MPI*. Par défaut, la taille des messages varie de 4 à 64 ko, par puissance de 2. La mesure de performance s'effectue, pour chaque taille de message, au sein d'une boucle qui s'exécute 100 fois (valeur par défaut). Chaque nouvelle mesure, lors d'une nouvelle taille de message, est précédée d'une mesure préliminaire pour éviter la perte de performance liée au chargement des données dans le cache. Pour déterminer la durée des mesures, l'appel à la fonction *MPI_WTIME* est effectué. Il y a deux types de tâches dans *MPBench*, la tâche maître qui est unique et les tâches esclaves. Pour les mesures point-à-point, il y a uniquement 2 tâches, une maître et une esclave. Alors que pour les autres, le nombre d'esclaves peut varier et est de 16 par défaut.
*MPBench* mesure donc les différentes métriques du réseau en calculant un temps moyen. Parmi ces différentes métriques, il y a :

**La Bande Passante** La mesure s'effectue entre deux processus *MPI*. Le processus maître envoie les données (envoi bloquant) alors que le processus esclave (réception bloquante) les reçoit. Lors de la fin d'une boucle, qui correspond à l'envoi d'un certain nombre de messages de taille identique, le processus esclave envoie un acquittement de 4 octets au processus maître. Cela permet au processus maître de savoir que l'ensemble des messages a été reçu et d'arrêter le *timer*. Le fait d'envoyer ce message a peu d'impact sur le calcul du temps moyen vu que le délai d'envoi d'un tel message est très court et qu'il est divisé par le nombre d'itérations.

**L'aller-retour** Cette métrique correspond au traditionnel *ping-pong*. Le processus maître envoie un message (envoi bloquant) au processus esclave, une fois les données reçues (réception bloquante) le processus esclave les renvoi aux processus maître.

**La Bande Passante Bidirectionnelle** La mesure s'effectue entre deux processus *MPI*. Chaque tâche envoie et reçoit des données via des appels non-bloquants. Pour s'assurer de la réception des messages des deux côtés, un appel à *MPI_Waitall()* est effectué.

Comme pour d'autres benchmarks, nous pouvons voir que *MPBench* ne nous informe pas de la variabilité des résultats et ne prend pas en compte la mesure de la contention réseau qui pourrait être causée par d'autres communications sur le réseau.

### 2.4.3 Mpptest

*Mpptest* est un benchmark qui contourne les principaux pièges lors de la mesure des performances des communications *MPI*, comme présenté dans [36]. *Mpptest* peut mesurer les communications point-à-point synchrones et asynchrones ainsi que certaines communications collectives. La taille des messages est choisie automatiquement par un schéma adaptatif qui utilise une interpolation linéaire afin d'éliminer les artefacts dans le graphe résultant de la mesure.
Le principe fondamental des techniques utilisée dans *Mpptest* est que le résultat des tests de performances doit être reproductible. Pour cela, *Mpptest* mesure le temps moyen de plusieurs itérations d'un message *MPI* comme *MPBench* ou *IMB*, mais répète ce processus plusieurs fois et ne retient que la valeur moyenne la plus petite. Le but de cette technique est de faire disparaître (et donc d'ignorer) les messages les plus lents lors des mesures. Bien que les résultats soient alors reproductibles, la répartition des mesures est faussée. De plus, comme *Mpptest* utilise le temps d'un aller-retour afin de faire une mesure uniquement sur un seul processus, il ne peut donc pas être utilisé pour obtenir une perspective globale des effets de la contention réseau qui peuvent survenir lors d'échanges de message entre d'autres processus.

### 2.4.4 SkaMPI

*SKaMPI* [82] est l'un des outils les plus complets dans le cadre du benchmarking des communications *MPI*. Il est conçu de telle sorte qu'il permet de tester les différents modes de communication, tout en ayant la possibilité de mesurer les performances de nouveaux schémas de communications. La taille des messages est sélectionnée automatiquement par une méthode d'interpolation linéaire semblable à celle de *Mpptest*. Le temps des appels est mesuré soit avec l'horloge du système, soit avec *MPI_WTIME*. Il n'y a aucune tentative de synchronisation entre les processus. De ce fait, les mesures des communications point-à-point sont effectuées en faisant un *aller-retour*. L'originalité de *SKaMPI* réside dans sa façon de calculer le temps des appels *MPI*. Dans le fichier de script, il est demandé de définir le nombre d'itérations minimum, maximum ainsi que l'erreur type (*standard deviation error*). Durant le benchmark, il exécute donc le nombre d'itérations minimum défini, et s'arrête dès que l'erreur type ou le nombre d'itérations maximum est atteint. Le

fichier résultat nous donne donc pour chaque taille de message, le temps en microseconde, l'erreur type et le nombre de mesure nécessaire pour obtenir le résultat. Tout comme d'autres benchmarks, *SKaMPI* n'est pas en mesure de définir précisément les effets de la contention liés aux échanges de messages survenant d'autres processus.

### 2.4.5 NetPIPE

*NetPIPE* [98] est un outil de mesure de performance du réseau indépendant du protocole fonctionnant sous différentes bibliothèques de communication (*MPI*, *MPI*-2, *PVM*, *shmem*) et sur différents réseaux (*Myrinet*, *Quadrics*, *InfiniBand*, *Gigabit Ethernet*, . . . ). *NetPIPE* se contente de mesurer la performance d'un simple *ping-pong* entre deux processus. Il fait cela en mesurant le temps des opérations individuellement via l'horloge du système et en reportant le temps de mesure le plus bas. Ce type de mesure ne permet pas de connaître la variabilité des résultats. De plus, *NetPIPE* ne permet pas de mesurer les temps de communications de schémas complexes où il serait possible de voir les effets de la contention.

### 2.4.6 MPIBench

*MPIBench* [37, 38] est un outil, comme Mpptest, qui s'appuie sur les travaux de Gropp et Lusk [36], concernant les différents pièges à éviter lors de prise de mesure de communication *MPI*. Contrairement aux autres benchmarks, la mesure du temps s'effectue par des appels directs aux compteurs hardwares. Une procédure de synchronisation des horloges est effectuée au début des mesures, afin d'obtenir la meilleure précision possible. L'originalité de *MPIBench* tient surtout de la possibilité d'une part, de pouvoir mesurer l'ensemble des communications *MPI* et de les combiner en différents groupes et, d'autre part, de pouvoir donner la répartition des temps obtenus pour chaque communication. *MPIBench* permet de connaître ainsi la variation des résultats de chaque groupe et de l'ensemble. En fichier résultat, *MPIBench* peut générer la distribution en temps et en tailles des différents appels. Ce benchmark permet bien de voir l'impact des contentions sur l'ensemble des communications, néanmoins, l'utilisation directe des compteurs hardwares limite la diversité des architectures cibles.

### 2.4.7 Bilan

Cette section a montré que les benchmarks *MPI* souffrent d'insuffisances. Il en existe principalement deux :

**La Precision** Pour contrer la perturbation de la mesure liée à l'utilisation de timer à "gros grains". Les benchmarks utilisent un grand nombre de répétions pour obtenir une valeur moyenne. Le problème de cette approche est que l'on ne peut connaître la variabilité du résultat, qui peut être un facteur important dans les performances d'un

programme.
Beaucoup de benchmarks s'appuient sur *MPI_WTIME* pour mesurer le temps écoulé. Sachant que la fonction sur laquelle pointe *MPI_WTIME* est désignée à l'installation de la bibliothèque *MPI*, il est important de bien la configurer lors de ce processus. Une même bibliothèque *MPI* pourrait donner des performances totalement différentes suivant son installation. Une solution à cela serait de faire appel aux compteurs hardware comme le fait *MPIBench*, mais le problème de cette approche est qu'au vu du grand nombre d'architectures, il serait complexe de pouvoir toutes les couvrir. Une solution pourrait être l'utilisation de bibliothèque permettant l'accès aux compteurs hardwares pour un grand nombre de processeurs, telle que *PAPI* (voir Annexe A).

**Le Manque d'information** Avec l'augmentation des cœurs au sein des processeurs et l'apparition de trois niveaux de communications (voir section 2.3), il est important de pouvoir connaître à quel niveau nous sommes lors de mesure. Or, aucun benchmark n'a été vraiment conçu pour informer de cela. Si l'on ne fait pas attention au placement des tâches, il est possible de se retrouver à mesurer les performances des communications intra-nœud alors que notre intention était de mesurer les performances des communications inter-nœud.
D'autre part, comme nous l'avons vu en section 2.3.5, le routage *InfiniBand* est déterministe. Or, le même problème se pose encore, il est impossible de connaître via les benchmarks actuels si certains messages passent par le même lien. Cela peut entraîner des réponses différentes lors de l'exécution du même benchmark sur une même plate-forme, mais sur des nœuds différents, sans pour autant que l'utilisateur comprenne les raisons de ces différences.

| Critère \ Benchmark | IMB | MPBench | Mpptest | SkaMPI | NetPIPE | MPIBench |
|---|---|---|---|---|---|---|
| Mesure | MPI_WTIME | MPI_WTIME | MPI_WTIME | MPI_WTIME | Horloge système | Compteur hardware |
| Concurrence | oui | non | non | oui | non | oui |
| Variabilité | non | non | non | non | non | oui |
| Schéma comm. | non | non | non | oui | non | oui |
| Routage | non | non | non | non | non | non |

TABLE 2.1 – Comparaison des benchmarks existants

Le tableau 2.1 permet de représenter l'ensemble des différents éléments traités dans cette partie. Cinq critères ont été retenus. Il y a :
- La méthode utilisée pour déterminer le temps d'exécution des opérations *MPI*.
- La possibilité d'étudier l'impact des communications concurrentes
- La possibilité d'observer la variabilité des résultats
- La possibilité d'utiliser des schémas de communications définis par l'utilisateur

– La possibilité de savoir si le routage crée des accès concurrents au niveau des liens lors des mesures.

## 2.5 La programmation parallèle

Il existe différentes méthodes de programmation, qui peuvent être utilisées, pour créer des programmes parallèles s'exécutant sur des noeuds de calcul [34, 48]. Une bonne méthode d'écriture de programme parallèle se doit de :

– permettre au programmeur de décomposer le problème en éléments (processus) qui peuvent être traités de façon parallèle.
– fournir un moyen de placer ces processus sur les unités de calcul.
– permettre une communication et une synchronisation entre ces processus.

Les processus sont des parties d'un programme parallèle qui s'exécutent sur les coeurs des processeurs. Habituellement, dans le cadre du calcul haute performance (et ainsi que dans les travaux présentés dans cette thèse), un seul processus s'exécute sur chaque coeur. Les deux principales méthodes de programmation sont étroitement associés aux principales architectures mémoires existantes (vu en section 2.2.2) : la programmation à mémoire partagée *"shared memory programming"* pour les architectures à mémoire partagée et la programmation par passage de messages *"message-passing programming"* pour les architectures à mémoire distribuée. Dans le cadre de nos travaux, il sera utilisé la bibliothèque par passage de message *MPI*, présentée en section 2.3.3.

### 2.5.1 La taxinomie de Flynn

Selon la taxinomie de Flynn [31], elles sont toutes les deux des méthodes de programmation de type *"Multiple Instruction, Multiple Data streams"* (MIMD), ce qui signifie que chaque coeur peut traiter indépendamment des données différentes. Flynn classe une autre méthode de programmation parallèle comme étant *"Single Instruction, Multiple Data streams"* (SIMD). Dans le cadre du calcul SIMD, un cœur joue le rôle d'unité de contrôle quand les autres cœurs travaillent de façon synchrone pour traiter les données en parallèle. Le modèle SIMD convient particulièrement bien aux traitements dont la structure est très régulière, comme c'est le cas pour le calcul matriciel. Bien que très répandu au début de l'ère du calcul parallèle, les architectures SIMD tendent à disparaître, même si les cartes graphiques utilisent encore ce type d'architecture. Néanmoins, l'approche du traitement SIMD des données en parallèle reste bien vivant. Cela est dû au fait que ce type de traitement peut être exécuté sur des architectures MIMD, qui sont suffisamment puissantes pour exécuter n'importe quel type de programme parallèle. Dans ce contexte, les programmes qui traitent les données en parallèle sont souvent appelés *"Single-Program, Multiple-Data"* (SPMD). Les méthodes de programmation, telle que la programmation par mémoire partagée ou par passage de messages, peuvent être utilisées

pour écrire des programmes SPMD. En outre, elles peuvent aussi être utilisées pour écrire des programmes *Multiple-Program, Multiple-Data* (MPMD).

### 2.5.2 La programmation par mémoire partagée

La programmation par mémoire partagée est l'approche la plus ancienne de celles existantes pour la programmation parallèle. Cela est dû, principalement, au fait que les architectures *SMP* (pour *Symmetric Multiprocessing*) apparurent avec les premiers ordinateurs parallèles et sont encore présentes aujourd'hui. Comme nous l'avions dit en section 2.2.2, les processus des programmes parallèles, appelés aussi threads, communiquent en opérant sur des données partagées dans un espace d'adressage global. Une synchronisation, en utilisant des sémaphores, est requise pour prévenir les autres threads de tout tentative d'écriture/lecture d'une structure de donnée partagée, lorsqu'un thread est en train d'écrire sur celle-ci. Cela a été décrit par Dijkstra dans ces articles concernant les sections critiques, les deadlock, les exclusions mutuelles et le dîner des philosophes [25, 26], ainsi que par Courtois et al's sur l'article abordant le problème des lecteurs et écrivains multiples [22]. Les programmes à mémoire partagée sont considérés, par beaucoup, comme étant les programmes parallèles les plus faciles à aborder, car les programmeurs doivent uniquement se soucier des synchronisations ; les communications n'étant pas nécessaires car il y a toujours uniquement une seule copie de toutes les structures de données. De plus, les compilateurs sont maintenant capables d'extraire le parallélisme d'un code séquentiel, ce qui fait que l'écriture d'un programme parallèle à mémoire partagée est beaucoup plus facile à réaliser. Malheureusement, comme la hiérarchie mémoire est cachée au programmeur, de part le paradigme de la programmation sur mémoire partagée, les performances impliquant la hiérarchie mémoire ne peuvent pas être exploitées pour obtenir un rendement maximal.

### 2.5.3 La programmation par passage de messages

La programmation par passage de messages d'un autre côté requiert de déplacer les copies des données au travers de la hiérarchie mémoire en utilisant une communication explicite. Dans ce type de programmation, chaque processus a sa propre mémoire locale. Les processus doivent communiquer et collaborer par paires, avec un processus qui initie l'opération d'envoi des données, et à qui doit correspondre une opération de réception lancée par un autre processus. Ces opérations d'envoi et de réception peuvent être soit synchrones, l'envoi et/ou la réception occupe le processeur jusqu'à la fin de l'opération, soit asynchrones. Une opération d'envoi asynchrone initie la transmission du message avant de rendre immédiatement la main au processus appelant, sans attendre que le message soit reçu. Une opération de réception asynchrone vérifie s'il y a des messages dans la file d'attente d'arrivée et réceptionne le message si cela est possible, sinon, il rend im-

médiatement le contrôle au processus appelant. Afin de permettre la synchronisation, les opérations de passage de message asynchrones doivent être utilisées en conjonction avec des opérations qui testent l'arrivée des messages. L'utilisation des opérations asynchrones est plus complexe que l'utilisation synchrone, mais cela fourni aux programmeurs la possibilité de recouvrir les communications par du calcul. Cela permet aussi au programmeur d'atténuer les effets de la latence en retardant la synchronisation jusqu'au moment ou cela est absolument nécessaire, afin de permettre d'augmenter la quantité de calcul durant cet intervalle.
Bien que ces facteurs nécessitent plus d'efforts de programmation, ils offrent un plus grand contrôle sur la localité et la synchronisation des données, qui peuvent être exploitées pour optimiser les performances. Du fait que la méthode du passage de message est une approche MIMD, elle est idéale pour les ordinateurs parallèles à mémoire distribuée, et peut également faire face efficacement aux problèmes irréguliers. Toutefois, une distribution prudente des données et répartition de charge sont nécessaires afin de veiller à ce que les cœurs soient gardés occupés, ce qui augmente encore l'effort de programmation.

### 2.5.4 Conclusion

Bien que les techniques de programmation par mémoire partagée et par passage de message soient très différentes dans la pratique, elles sont fondamentalement en mesure de décrire le parallélisme de façon générale. De ce fait, tout programme à mémoire partagée peut être traduit dans une sémantique équivalente en programme par passage de message et vice versa [17, 122].
Les bibliothèques *"Distributed Shared Memory"* (*DSM*) sont disponibles afin de simuler l'apparence d'un système à mémoire partagée sur une grappe de calcul à mémoire distribuée et les bibliothèques de passage de message sont utilisables sur des noeuds à mémoire partagés. Les bibliothèques *DSM* sont généralement fournies pour apporter la commodité de la programmation à mémoire partagée sur des machines à mémoire distribuée, mais elles ne permettent pas le même niveau de performance qu'un programme écrit en passage de message sur la même machine. La même chose est vraie lorsque l'on utilise des bibliothèques par passage de message dans un environnement à mémoire partagée, mais elles sont encore en mesure d'offrir de hautes performances, car elles continuent d'exploiter la hiérarchie mémoire de l'application. Comme nous l'avons vu en 2.3, les opérations à mémoire distantes (*RDMA*) sont possibles sur les architectures à mémoire distribuée et ces routines de communication ont été introduites dans de nombreuses bibliothèques de communication de bas niveau. Ces routines permettent aux processus de lire ou d'écrire directement des données appartenant à un processus distant sans sa coopération ou copie. Cela offre l'option de la simplicité de la programmation sur mémoire distribuée dans le paradigme du passage de message.
Du point de la modélisation de performance, les programmes parallèles sont simplement des programmes par passage de message déguisés, les programmes à mémoire partagée

pouvant être exprimés en programme par passage de message. En outre, ni la mémoire partagée ni le paradigme de la programmation parallèle sont aussi puissants que l'approche par passage de message. Par exemple, ils ne peuvent pas exprimer des programmes qui permettent d'atteindre un rendement optimal sur des machines à hiérarchies mémoire complexes.

## 2.6 Les benchmarks utilisés dans le calcul haute performance

Afin de pouvoir comparer les performances des grappes de calcul, certains programmes de benchmarking sont souvent utilisés. Ces programmes peuvent s'intéresser soit à une caractéristique précise de la machine soit à l'ensemble de la machine. Dans cette partie, quatre types de programmes seront présentés.

### 2.6.1 STREAM

*STREAM* [1] fait partie de la catégorie des micro-benchmarks. Cette application est devenue un standard pour mesurer la bande passante mémoire réelle. Néanmoins, plusieurs facteurs influent sur ces résultats. Tout d'abord, au niveau logiciel, il y a le compilateur. Pour un processeur *Intel*, les sources compilées avec le compilateur *Intel* donneront souvent des résultats meilleurs qu'avec *gcc*. Ensuite, au niveau software, il est important de bien définir une taille de problème suffisamment grande. En effet, si le tableau n'est pas assez grand, il sera contenu uniquement dans le cache, donnant des performances relativement bonnes mais les mesures correspondront aux temps d'accès au cache et non à la bande passante mémoire.

### 2.6.2 Linpack

*Linpack* est un benchmark datant du début des années 80 et permettant de tester les performances des machines au niveau du calcul flottant via la résolution d'un système d'équation linéaire. Une implémentation appelée *HPL* [2], pour *High-Performance LINPACK*, a ensuite été écrite afin de pouvoir comparer les performances des machines à mémoire distribuée dans le domaine du calcul haute performance. Depuis 1993, il existe un classement permettant de connaître les machines obtenant les meilleurs résultats. Ce classement est connu sous le nom de *TOP500* [3].
Là encore, le compilateur et les bibliothèques associées à *HPL* influent beaucoup sur les résultats. En effet, il faut savoir que *HPL* fait appel une API appelée *BLAS* (Basic Linear

1. http ://www.cs.virginia.edu/stream/
2. http ://www.netlib.org/benchmark/hpl/
3. www.top500.org

Algebra Subprograms) pour effectuer des opérations sur les vecteurs et les matrices. Or, il faut savoir que l'opération la plus gourmande en temps est SAXPY (ou DAXPY en double précision). Elle est contenue dans l'API *BLAS* et consiste à ajouter le résultat de la multiplication d'un scalaire par un vecteur à un autre vecteur. Cette opération est très souvent optimisée dans les implémentations. Il y a même des compilateurs intégrant un "daxpy recognizer" [29] pour remplacer le code optimisé à la main. Les performances de *HPL* sont donc souvent dépendants du compilateur et de l'implémentation *BLAS*. Néanmoins, ce benchmark, grâce au TOP500, permet de suivre et d'analyser l'évolution hardware et software des grappes de calcul disponibles sur le marché.

### 2.6.3 Les NAS Parallel Benchmarks

Développé par la *NASA Advanced Supercomputing (NAS) Division*, les NAS Parallel Benchmarks [4] [5] sont une série de onze petites applications parallèles. Elles ont été beaucoup étudiées et sont souvent utilisées pour comparer les performances des grappes de calcul. L'avantage de cette suite est qu'elle propose des algorithmes avec des schémas de communications variés, permettant de tester différents comportements. Néanmoins, ces applications n'ont pas été conçues pour des grappes de calcul de très grande taille, limitant alors leur utilisation.

### 2.6.4 SpecMPI

*SpecMPI* [5] est composé d'une série de 13 applications parallèles issues du monde réel. L'avantage de ces benchmarks, par rapport aux NAS, est qu'ils sont conçus pour supporter les grappes de calcul de grandes tailles. Actuellement, il existe deux versions de *SpecMPI*, la première supporte entre 4 et 512 cœurs, alors que la seconde supporte entre 64 et 2048 cœurs. L'avantage de *SpecMPI* est que ces applications utilisent un très large éventail de communication *MPI* permettant de fournir un panel de test pour évaluer les performances des implémentations *MPI* sur des grappes de calcul de grande taille.

### 2.6.5 Bilan

Il existe un très grand nombre de benchmarks pour caractériser les grappes de calcul dans le calcul haute performance. Ces programmes sont utilisés à plusieurs niveaux :

**Au niveau du vendeur** pour montrer les performances de leurs grappes de calcul et attirer les acheteurs.

**Au niveau de l'acheteur** pour avoir des critères et un point de comparaison entre les différentes propositions des vendeurs.

4. http ://www.nas.nasa.gov/Software/NPB/
5. http ://www.spec.org/mpi2007/

**Au niveau du designer** pour adapter au mieux la configuration de la grappe de calcul afin d'avoir de meilleures performances.

Néanmoins, les différents résultats obtenus ne doivent être considérés que comme des indicateurs, car choisir une grappe de calcul ayant les meilleurs résultats aux différents benchmarks ne voudra pas dire qu'elle sera la mieux adaptée aux besoins d'utilisation, entraînant alors un surcoût à l'achat de la machine et une sous exploitation de ces possibilités.

## 2.7 Conclusion

Ce chapitre a permis de voir qu'une grappe de calcul est construite autour d'un ensemble de nœuds de calcul relié par un réseau haute performance connecté suivant une topologie donnée. L'ensemble des applications de calcul haute performance, s'exécutant sur ces nœuds, utilise la bibliothèque par passage de message *MPI*. Il est possible de connaître les performances *MPI* d'un réseau haute performance en utilisant des benchmarks spécifiques.

La puissance de calcul d'une grappe dépend grandement des caractéristiques de ces nœuds, et principalement de son processeur. Faute de pouvoir monter en fréquence, les processeurs ont eu leur nombre de cœurs augmentés. Néanmoins, l'augmentation des cœurs crée des problèmes de partage amenant à une instabilité de ces performances.

Le réseau haute performance *InfiniBand* se distingue des autres réseaux par sa faible latence et sa grande bande passante. La bibliothèque de communication bas niveau *libibverbs* ainsi que l'interface de programmation par passage de message, *MPI*, permettent d'exploiter au mieux les performances du réseau *InfiniBand*. Néanmoins, comme nous l'avons signalé, le routage déterministe du réseau *InfiniBand* peut être une cause de baisse de performance, un mauvais routage pouvant amener des communications à se partager des liens et fausser les observations réalisées. Cet élément doit être pris en compte pour avoir une compréhension des dégradations.

Il existe différents benchmarks proposant de nombreuses techniques pour mesurer les performances d'un réseau. Toutefois, il n'existe pas actuellement de benchmark précis pouvant permettre une étude de la contention de schémas de communication complexe, tout en informant l'utilisateur des risques de dégradation liés au routage. Il est pourtant nécessaire d'avoir un tel outil afin de pouvoir étudier précisément l'impact du partage de la bande passante au niveau du nœud.

En utilisant le benchmark développé par Martinasso [67] et amélioré au cours de cette thèse, il sera possible d'observer les pénalités liées au partage de la bande passante au niveau du nœud permettant alors une modélisation des comportements des communications concurrentes. Mais dans le cadre de l'extrapolation de performance, il est aussi nécessaire

de pouvoir estimer les temps de calcul de l'application afin de savoir exactement le moment du départ des communications.

Cette thèse se concentrera donc à la fois sur l'estimation des temps de calcul et de communication afin de prédire les performances d'une application sur une grappe de calcul. Cette analyse permettra alors de pouvoir mieux dimensionner les grappes de calcul, afin de répondre aux critères de performances (temps d'exécution ou de communication, bande passante, latence, ...) d'une application donnée.

Il existe déjà de nombreuses méthodes différentes pour pouvoir caractériser les performances d'une application parallèle. Dans le chapitre suivant, nous proposerons un aperçu des différentes approches permettant de comprendre et de prédire le comportement logiciel et matériel des grappes de calcul.

# Les techniques de modélisation de performance des systèmes parallèles ou distribués 3

## 3.1 Introduction

La genèse du calcul parallèle date des années 1960 et 1970. A cette époque, la recherche sur les algorithmes parallèles était principalement fondée sur la théorie. Dans son livre [79], Peterson résume bien les principaux formalismes alors introduits pour les opérations parallèles. Bien que ces travaux aient fourni des formalismes robustes qui continuent à servir de base pour la modélisation des processus parallèles, le temps de calcul nécessaire pour résoudre ces problèmes croit de façon exponentielle avec la taille du système. Cela fait que ces modèles sont applicables à de petits systèmes, ou dans des systèmes où le coût en terme temps de calcul n'est pas le critère principal pour trouver une solution (section 3.2.4).
Dans les années 80, d'importants travaux de recherche ont porté leurs attentions sur les moyens pratiques de concevoir des algorithmes parallèles efficaces en utilisant le modèle PRAM (Parallel Random Access Machine) et ces variantes (section 3.2.2). Ces modèles requièrent qu'un algorithme soit complètement décrit en termes d'accès à la mémoire. Cela a permis aux chercheurs de comprendre à un niveau de grain très fin les caractéristiques des algorithmes parallèles étudiés. Malheureusement, les modèles de machines, sur lesquelles ces études ont été basées, ont été incapables de refléter la complexité des processus physiques qui se produisent dans un système informatique parallèle. En effet, le modèle PRAM ignore les surcoûts liés aux synchronisations, aux communications de données et à l'ordonnancement. Bien que de nombreuses extensions au modèle PRAM ont été proposées pour tenter de surmonter ces limitations diverses, ils ont en grande partie échoué, car ces nombreuses modifications ne pouvaient pas être efficacement unifiées.
Au cours de la fin des années 80 et au début des années 90, les chercheurs ont commencé à envisager de nouvelles façons d'aborder le problème de la modélisation des performances des programmes parallèles. De nouvelles méthodes ont été mises au point et cela

a permis une évaluation directe de la performance en se basant sur la structure du code source (cf. section 3.2.5). Dans de nombreux cas, ces techniques ont permis de modéliser les performances à l'aide de plusieurs paramètres et de mise en équations. Bien que ces méthodes aient permis la création de programmes efficaces avec des performances prévisibles, ces techniques n'étaient pas abordables pour un nombre croissant de problèmes irréguliers que le calcul haute performance visait à résoudre et qui nécessitait des structures de contrôle plus complexe.
Pour faire face à ces programmes plus complexes, les chercheurs se sont tournés vers des techniques statistiques et de modélisation Markovienne pour essayer d'estimer les caractéristiques des performances d'un programme. Bien que ces approches aient réussi à modéliser des situations particulières, ils ont souffert de deux problèmes. Le premier a été la forte exigence de calcul de ces approches, car les techniques de résolution étaient généralement d'ordre exponentielle. La deuxième difficulté a été plus importante : ces techniques n'ont pas vraiment aidé les programmeurs à mieux comprendre leurs programmes parallèles ou à leur donner une idée sur la façon qui leur permettrait de l'améliorer. Cependant, ces techniques sont encore en usage aujourd'hui pour certains problèmes bien définis.
Plus récemment, la modélisation a favorisé la recherche de techniques qui impliquent une compréhension détaillée de la performance de petits morceaux de code (Sections 3.4.3 et 3.4.2) plutôt qu'une notion générale de performance moyenne de blocs macroscopiques de code. En un sens, cela est très similaire aux précédentes techniques de modélisations PRAM, bien que des progrès significatifs ont été réalisés dans la compréhension des causes de la dégradation des performances des programmes parallèles (Section 3.2.3). Cette philosophie a encouragé le développement d'outils de modélisation de performance capables de s'attaquer à un grand nombre de programmes parallèles.
Nous pouvons donc distinguer trois différents types d'approches pour modéliser les performances des programmes parallèles : l'approche analytique ou par modèle abstrait, celle par simulation et enfin par des outils de prédiction alliant plusieurs techniques. Dans ce chapitre, nous allons donc voir pour chacune de ces approches un aperçu des différentes méthodes mise en place pour prédire les performances

## 3.2 Approche Analytique et modèles abstraits

Ces approches sont assez traditionnelles pour construire des modèles de performance. Elles exigent une compréhension profonde des algorithmes qui sont utilisés, des détails sur leur mise en œuvre et une bonne connaissance de la structure dynamique de l'exécution de l'application. Ces méthodes sont plus utilisées dans le domaine de la recherche que dans le milieu industriel. Toutefois, elles continueront à être utilisée jusqu'à ce que d'autres méthodes automatisées deviennent suffisamment matures pour atteindre le niveau désiré de précision. Sur une plus petite échelle, ces modèles peuvent également être

utilisées pour valider des modèles construits avec d'autres techniques.

### 3.2.1 Amdahl

Le premier modèle couramment utilisé pour déterminer le gain de performance des programmes parallèles a été conçu en 1967 par Amdahl [3]. Ce modèle est utilisé pour calculer le *speedup* possible S d'une version parallèle d'un programme comparé à sa version séquentielle, en utilisant la formule :

$$S = \left( \frac{1}{r_s + \frac{r_p}{n}} \right)$$

où $r_s$ est la partie séquentielle du programme, $r_p$ correspond à la partie parallèle du programme et $n$ est le nombre de processeur utilisé dans la version parallèle. La partie séquentielle d'un code représentant le temps d'exécution associé à toutes les parties du code qui ne peuvent être exécutées uniquement que sur un seul processeur. Le modèle d'Amdhal ne tenant pas compte du coût des transfers des messages, il est surtout utilisé pour évaluer les performances d'un programme dans le cadre de la programmation à mémoire partagée.

### 3.2.2 Le modèle PRAM

En 1978, Fortune et Wylie ont décrit un modèle abstrait de machine parallèle appelé *Parallel Random Access Machine* (PRAM). Les premiers modèles relatifs à la PRAM ont été décrits par Schwartz [93] et Goldschlager [35]. La PRAM visait à fournir un modèle général de calcul parallèle. Car, bien que les modèles spécifiques exploitaient pleinement le matériel disponible, ils étaient rarement portables. Le modèle fourni par la PRAM visait à faire abstraction des détails des machines et des styles de programmation. Cela permettait alors de se concentrer plutôt sur les possibilités de parallélisme pour un problème donné, mais au prix de l'optimisation basée sur la connaissance de ces détails.
La PRAM est une machine parallèle idéalisée et constituée de $P$ processus de communication synchrone via une mémoire partagée. Chaque processus est capable d'exécuter une instruction ou d'exécuter une opération de communication par cycle d'horloge. Il existe plusieurs familles de PRAM qui sont classées par la sémantique utilisée pour accéder à la mémoire partagée. Il y a :

- l'Exclusive Read, Exclusive Write (EREW) PRAM : chaque processus ne peut lire ou écrire à un endroit de la mémoire que si aucun autre processus n'y accède à ce moment-là.
- la Concurrent Read, Exclusive Write (CREW) PRAM : chaque processus peut lire n'importe quel endroit de la mémoire à tout instant, mais aucune écriture simultanée de deux processus à un même endroit n'est possible.

– la Concurrent Read, Concurrent Write (CRCW) PRAM : chaque processus peut lire et écrire n'importe où dans la mémoire à tout moment.

Dans le cas de la PRAM CRCW, une nouvelle sous-classification s'applique, basée sur la résolution de conflits à arbitrer lors d'écriture concurrente.
La simplicité et la généralité du modèle PRAM a conduit à sa large reconnaissance comme outil de recherche, entre autre pour la recherche sur :

– le problème des accès concurrents.
– le problème du placement des données en mémoire afin de minimiser les phénomènes d'accès concurrents [10, 11]

Malheureusement, le coût du modèle liés à la PRAM ne se révèle pas très utile dans la pratique. Ce modèle est incapable d'exprimer la disparité des coûts réels associés à des machines, comme par exemple la différence entre l'accès local et à distance de la mémoire. Au cours des années, plusieurs ajouts au modèle basique PRAM ont été réalisés pour tenter de fixer le décalage entre le modèle et la réalité. Ces extensions essayaient de tenir compte des processus asynchrones [20], de la latence réseau et d'une bande passante limitée [2] et de la topologie local [45, 51, 105]. Une bonne étude du modèle de base PRAM et de ses variantes peut être trouvé dans [42]. Malgré ces améliorations, le modèle PRAM n'est pas encore en mesure de produire des estimations précises des coûts de code fonctionnant sur des plates-formes matérielles. Son utilité, pour la prévision des performances sur des machines parallèles, est donc de portée très limitée.

### 3.2.3 Adve

En 1993, Adve [117] a soumis un mémoire de thèse qui a analysé le comportement et les performances des programmes parallèles. Le modèle, qui a été présenté, est une étape importante dans la modélisation des performances des programmes parallèles, car elle a fourni beaucoup plus d'informations qualitatives et quantitatives sur les performances d'un programme parallèle que les précédentes méthodes, le tout pour un coût de calcul comparable. Alors que la plupart des modèles présentés, avant le modèle d'Adve, ne pouvait être appliqué qu'à des programmes avec de simples structures de synchronisation ou requéraient des solutions techniques complexes. Son modèle pouvait permettre à un programmeur de prévoir assez précisément l'impact des changements sous-jacent du système, permettant ainsi de guider les décisions de conception du programme, pour trouver une solution parallèle efficace à un problème.
Durant sa thèse Adve a développé et validé un modèle de prédiction de performance de programmes parallèles déterministes. Il a testé sa précision, son efficacité ainsi que sa fonctionnalité sur des programmes réels avec un ensemble de données d'entrée. Le modèle utilise des valeurs déterministes pour le temps moyen des tâches et des communications, alors que la contention des ressources partagées est calculé par un modèle stochastique. Combiné avec des représentations abstraites du comportement des programmes et des systèmes, le modèle a permis d'analyser des programmes et des systèmes hypothé-

tiques ainsi que les combinaisons de ces éléments.
Bien qu'une limite fondamental du modèle d'Adve est qu'il ne prend pas en compte la variance dû aux délais de communications, ses recherches ont montré que, en réalité, pour beaucoup de codes sur de nombreuses machines, le principal effet de ces délais est d'accroître le temps d'exécution entre les points de synchronisation tout en laissant la variance inchangée. Ce résultat a alors contredit une hypothèse de l'époque qui indiquait qu'il y avait une grande variance dans les temps d'exécution des programmes parallèles. L'élément clé de la thèse d'Adve [117] est qu'il peut être raisonnable d'ignorer la variance des tâches et de leurs temps d'exécution quand le coût des synchronisations est important dans un programme parallèle.
Dans le modèle d'Adve, le temps d'exécution total est la somme de quatre composants :

$$t_{total} = t_{computation} + t_{communication} + t_{resource\ contention} + t_{synchronisation}$$

où $t_{computation}$ exclu tout calcul effectué lorsqu'il se chevauche avec une communication. Bien que ce modèle soit conceptuellement simple, dans la pratique, il est difficile à mettre en place. De part la nature non-déterministe de la congestion ainsi que la difficulté à déterminer les retards moyens de synchronisation.
L'approche d'Adve requière principalement deux entrées. La première est le graphe de tâches du programme. La deuxième est l'ensemble des paramètres d'utilisation des ressources pour les tâches qui sont soit déduits ou mesurés par expérimentation. Construire le graphe de tâches d'un programme est équivalent à reproduire la structure des contrôles parallèles du programme. Ceci peut être réalisé à partir d'une compréhension de base du programme et de l'exécution partielle ou non du programme. Adve a utilisé un graphe de tâches pour représenter le comportement du programme, car il pensait qu'il s'agissait d'un niveau approprié d'abstraction pour un modèle analytique. Il a donc défini :

- Une tâche comme une unité basique de travail
- Un graphe de tâches comme un graphe acyclique orienté qui décrit le parallélisme inhérent à un programme, où les noeuds représentent les tâches et les arêtes représentent les relations entre les tâches.
- Un processus comme une entité qui peut être placée sur un processeur pour exécuter des tâches.
- Un graphe de tâches condensées, comme un graphe de tâches réduit, de sorte à se que chaque noeud représente un ensemble de tâches qui pourraient être exécutées par un seul et même processus.
- Un graphe de tâches fork-join, composé de phases de calculs parallèles séparées par des barrières de synchronisation.

La construction d'un graphe de tâches condensées réduit le graphe de sorte que chaque sommet représente le travail effectué entre les points de synchronisation. Les graphes de tâches condensées peuvent être d'ordres de grandeur plus petits que le graphe de tâches et sont des constructions utiles pour réduire la complexité. Une sous-classe fréquente des graphes de tâches condensées sont les graphes de tâches *fork-join* qui sont en mesure

de décrire les programmes écrits dans les langues procéduraux. Ces graphe de tâches fork-join ont des frontières entre les points de synchronisation, ce qui permet d'éviter une explosion de l'espace d'état. Un certain nombre d'autres modèles ont été développés pour réduire le graphe de tâches, mais ils sont tout aussi limités, que le relativement simple, mais extrêmement commun, graphe de tâches fork-join [4, 28, 107, 108, 116].
Un problème gênant de ces approches est que certains programmes sont non-déterministes, et peuvent varier considérablement entre leurs différentes exécutions. Cela se produit, en partie, à cause d'effets dépendant des données. Bien que d'autres techniques ait été utilisées avec succès pour modéliser cela en utilisant des temps d'exécution de tâche stochastique [4, 28, 57, 66, 90], ils ne sont pas incorporables dans le modèle de tâches déterministes d'Adve.

### 3.2.4 Les méthodes d'analyse formelle

Il existe plusieurs méthodes bien connues pour l'analyse formelle de la concurrence. Les plus importantes sont le Communicating Sequential Process (CSP) [47, 84] de Hoare proposé en 1978, et le Calculus of Communicating Systems (CCS) [71] de Milner introduit en 1980. Ces deux méthodes sont étroitement liées [13] et sont inspirées du travail pionnier de Dijkstra sur les Cooperating Sequential Processes [27] en 1968. CSP et CCS sont souvent décrits comme des algèbres de processus, parce qu'ils fournissent un cadre mathématique pour spécifier le comportement des processus parallèles.
Les méthodes formelles définissent une syntaxe rigoureuse et des sémantiques pour un petit nombre d'opérations qui décrivent un traitement séquentiel, une synchronisation, une communication, une interruption ainsi que des choix déterministes ou non-déterministe. Ces opérateurs peuvent être utilisés pour créer des modèles d'applications spécifiques. Ces modèles sont ensuite évalués pour toutes les séquences d'évènements possibles qui pourraient survenir, généralement en respectant certaines conditions d'acceptation aux fins de la validation du modèle. Ces conditions sont l'absence de deadlock ou de livelock ou des contraintes pour le contrôle des paramètres du modèle. Essentiellement, l'objectif principal de la plupart des méthodes formelles est de prouver ou de réfuter de manière rigoureuse le bon fonctionnement des systèmes concurrents. Afin d'atteindre cet objectif, tous les détails de fonctionnement d'un système parallèle doivent être décrits précisément. En raison de la complexité inhérente à cela, en termes de construction et d'évaluation du modèle, les méthodes formelles sont principalement réservées pour la modélisation des systèmes concurrents, où l'échec n'est pas une option. Par exemple, les protocoles de communication sont souvent vérifiés en utilisant des techniques de modélisation formelle.

### 3.2.5 Le Bulk Synchronous Parallelism

En 1990, Valiant [111] décrit une technique pour l'écriture efficace des programmes parallèles et pouvant prédire les performances, appelée Bulk Synchronous Parallelism

(BSP). Valiant a vu que l'un des plus gros problèmes, avec le passage de message, est que l'approche analytique des coûts découlant des modèles de prédiction de performance était très difficile, en raison du nombre et de la complexité des transferts de données qui lui sont associés. Au lieu de cela, la méthodologie BSP demande aux programmes parallèles d'être structuré de telle manière que leurs calculs et communications soient séparés, de sorte que chacun puisse être considéré comme une grande quantité. BSP estime les actions de communication *en masse*, car il est plus simple d'estimer les limites du temps de communication ainsi par rapport à une application par passage de message non structuré. Un bon aperçu et comparaison de BSP avec d'autres techniques peut être trouvé dans [97].
Le calcul dans les programmes BSP s'écoule au travers une série de *supersteps* parallèles, qui sont divisés en trois phases. Dans la première phase, chaque couple processeur/mémoire $P$ est impliquée dans le calcul en utilisant uniquement des données locales. Cela peut être modélisé par le paramètre $s$ de McColls [68] qui représente le nombre d'opérations basiques (telles que l'addition ou la multiplication) qui peuvent être traitées par un processeur par unité de temps. Dans la seconde phase, les processus s'échangent les données via une phase de communication. Durant cette phase de communication, n'importe quel nombre de messages peuvent être envoyés et reçus. Le schéma de communication est défini parce qu'on appelle une $h - relation$, qui implique que chaque processus envoie ou reçoit au plus $h$ messages. Habituellement, $h$ est un paramètre qui prend aussi en compte la taille totale $m$ en octets des messages rencontrés par un processus, c'est-à-dire $hm$, mais est généralement abrégé en $h$. Le temps de communication est représenté par le paramètre $g$ qui représente le temps nécessaire à une $h - relation$ pour envoyer son message de façon continu entre des processus aléatoires. Le paramètre $g$ est normalement défini empiriquement pour une machine particulière, et est lié à la *bisection bandwidth* de cette dernière, les performances de la stack réseau, la gestion du buffer utilisé, la stratégie de routage et du *BSP run-time system*. Dans la phase finale, une barrière de synchronisation est effectuée. La durée de cette synchronisation est représentée par le paramètre $l$ qui est également déterminé de manière empirique.
Le temps d'exécution d'un *superstep* BSP peut être calculé à partir du code du programme et des paramètres de l'architecture cible qui ont été décrits ci-dessus. Le modèle de coût standard utilisé pour ce faire est :

$$cost\ of\ a\ superstep = max_{i=1}^{p}(w_i) + max_{i=1}^{p}(h_i g) + l$$

où $w_i$ est le coût pour le calcul local du processus $i$, et $h_i$ est le nombre de message envoyé ou reçu par le processus $i$. En regardant le modèle de coût standard, il est clair qu'un programme BSP efficace doit : équilibrer le temps de calcul des processus pour minimiser $w_i$ ; équilibrer les communications entre les processus pour minimiser $h_i g$ ; et minimiser le nombre de *supersteps* pour réduire le nombre de barrière de synchronisation de durée $l$ qui sont requises.
La réponse à ces exigences, pour un programme spécifique, peut être facilitée par les ou-

tils de la BSPlib Toolset [46], qui peuvent créer une description des performances d'une trace obtenue par une unique exécution du code sur une machine parallèle. Le modèle de performance qui est généré peut être utilisé dans le processus de conception pour l'écriture du programme BSP, dans le portage du programme BSP vers un nouvel ordinateur parallèle, ou lors des décisions d'achat d'un ordinateur parallèle (basé sur les paramètres $s$, $g$, $l$ de cette machine). Le BSPlib Toolset fournit également un aperçu de l'ampleur de l'amélioration de la performance qui pourrait être faite en utilisant une implémentation asynchrone de passage de message tel que le *Message-Passing Interface* (MPI) [99] ou E-BSP [56], qui a étendu la définition de la *h-relation* par l'ajout des notions de localité et de communication asymétrique dans le modèle BSP.

## 3.3 La simulation

Les méthodes de prédiction de performance basées sur la simulation consistent à exécuter une application avec un programme qui simule l'architecture et/ou le réseau cible. Comme chacune des instructions doivent être simulées, ces méthodes peuvent s'avérer parfois lentes.

### 3.3.1 Simics

Ils existent différents outils capable de simuler l'exécution d'un programme au niveau de l'instruction sur une architecture et, par conséquent, de déterminer sa performance. Simics [64], par exemple, est capable de simuler un système informatique complet incluant l'OS et les drivers pour une grande variété de plate-forme. Il est capable de modéliser fidèlement le transfert des données dans le cache et la mémoire sur sa plate-forme virtuelle et cela, sans modifier le fichier binaire. Ce type d'approche fait que les sessions de debuggage et d'analyse de performances sont totalement non intrusive, permettant l'accès à des informations détaillés, tout en étant arrêtable et redémarable à volonté. De plus, comme Simics simule exactement l'ensemble du système, il est capable de simuler complètement les couches de communications ainsi que les processeurs multi-cœurs. Malheureusement, les temps de communication ne tiennent pas compte des contentions. Et du fait que les programmes parallèles sont très dépendant des performances du réseau, un modèle de réseau plus précis serait nécessaire pour obtenir des résultats de simulation corrects.
Un outil comme Simics est extrêmement précis, mais il a un très gros désavantage : il est extrêmement lent. Par exemple, Simics est seulement capable de simuler un programme avec un ratio de 1/50 jusqu'à 1/400. C'est-à-dire qu'au mieux, pour 1 seconde de temps du programme écoulée , il lui faut 50 secondes de simulation. Bien que des méthodes, tel que le support du multithreading et la possibilité de distribuer la simulation (via *Simics Accelerator 2.0*), aient été implémentées, ce type d'outil n'est pas fait pour la prédiction d'application parallèle. La prédiction de performance avec un tel outil prendrait beaucoup

trop de temps. La principale raison d'exister de ce type d'outil est d'aider les designers au niveau hardware, de l'OS, ou du compilateur à optimiser les performances d'opérations de très bas niveau.

## 3.4 Les environnements de prédictions de performances

Ces environnements sont des résultats de recherches beaucoup plus récents que les modèles abstraits vus précédemment. Ils sont souvent basés sur l'utilisation d'un ensemble d'outils afin d'avoir une vision des performances des programmes.

### 3.4.1 POEMS

POEMS [1] est un environnement permettant la modélisation de systèmes distribués ou parallèles complexes développé par Adve. La modélisation incluse dans POEMS couvre un large domaine incluant l'application, les bibliothèques, le système d'exploitation et l'architecture hardware. La tentative de POEMS a été d'assembler des modèles de ces différents domaines pour former une modélisation globale. Chaque domaine étant modélisé par un outil spécifique. Le framework de POEMS est un système complexe qui a pour but de former un modèle global en utilisant l'analyse, la simulation et des mesures directs. POEMS utilise une représentation graphique du programme parallèle sous forme de graphe de tâches.

L'inconvénient de cette méthode est sa grande complexité à être mise en place avec une possible durée de la simulation relativement longue. D'autre part, les graphes de tâches nécessaire ne peuvent être générés que par le compilateur Rice dHPF [69].

### 3.4.2 WARPP

WARPP [40] (WARwick Performance Prediction) est un toolkit composé de différents outils de benchmarking et d'un simulateur, codé en Java, développé par l'Université de Warwick permettant d'évaluer les performances d'application *MPI* utilisant un réseau *InfiniBand*. Ce simulateur est issu de la combinaison de différents travaux effectués à l'université de Warwick (projet PACE des années 1990) sur la modélisation de performance, l'analyse, l'instrumentation automatique, le profiling de code mais aussi la simulation à évènement discret.

Le mécanisme mis en place est le suivant :

- Le code source de l'application parallèle est analysé afin d'être instrumenté. Le code instrumenté est alors exécuté sur l'architecture cible permettant ainsi de connaître les temps d'exécutions entre chaque appel MPI.
- Les différents types de communications MPI sont benchées en utilisant IMB (voir section 2.4.1) sur l'architecture cible en prenant en compte trois niveaux de com-

munications : intra-socket (même processeur, mais 2 coeurs différents), inter-socket (même noeud, mais 2 processeurs différents) et via l'interconnect.
– Les entrée/sorties sont elles aussi benchées sur l'architecture cible via l'outil IOR [94].

L'ensemble des informations récoltées permet d'estimer le temps d'exécution d'une application parallèle sur le simulateur et de pouvoir comparer les performances si l'on changeait le placement des tâches. Un élément intéressant dans ce simulateur est que l'on peut rajouter des traces de bruits permettant l'étude des perturbations dans l'exécution de l'application. L'inconvénient de ce simulateur est qu'il est nécessaire d'exécuter au moins une fois le programme sur la grappe de calcul cible, rendant impossible toute extrapolation pour une grappe de plus grande taille. De plus, changer le placement d'une tâche peut entraîner une modification de son temps d'exécution. D'autre part, le benchmark *IMB* présente certains inconvénients, comme nous l'avons déjà signalé. Ce simulateur même si il obtient une faible erreur dans sa prévision, ne tient pas compte des contentions possibles liés au partage de la bande passante au niveau du noeud ou du switch, ainsi que de la table de routage. La précision des temps de communications est donc bonne pour des programmes parallèles communicant peu mais s'avérerait mauvaise si des schémas de communications complexes apparaissaient ou si la table de routage était mal configurée.

### 3.4.3 Saavedra et Smith ; Chronosmix

Une approche à la fois simple et attrayante pour modéliser les performances a été proposée par Saavedra et Smith en 1992 [88]. Bien que cette méthode n'ait pas été conçue pour prendre en compte le parallélisme, il est possible d'appliquer les principes inhérents de la méthode aux programmes parallèles. Ils avaient noté que les techniques de benchmarking traditionnelles ne parvenaient pas à elles seules à caractériser les programmes et les machines, et donc que les résultats obtenus de cette manière étaient liées à un programme spécifique sur une machine particulière. Ils ont alors montré qu'une modélisation des performances du système pouvait être réalisée avec :

– un outil de (micro) benchmarking appelé *machine characteriser* qui permet de déterminer la performance des opérations de base sur une machine parallèle.
– un programme d'analyse pour statistiquement compter le nombre de ces opérations dans chaque bloc de base et compter le nombre d'itérations de ces blocs de base, durant l'exécution d'une version instrumentée du programme ;
– un *"execution predictor"*, qui combine les résultats des deux précédentes étapes et prévoit la performance globale du programme.

La seule limitation notable de cette technique est de demander à chaque programme d'être en fait exécuté au moins une fois avant la modélisation, ce qui limite son utilité comme outil de prototypage. Pour vérifier leur approche, Saavedra et Smith ont utilisé leur méthode pour prédire la performance des benchmarks SPEC'89 (the Standard Performance Evaluation Corporation) [110] et Perfect Club [8] sur plusieurs machines RISC (Reduced

Instruction Set Computer). Pour résumer les résultats, leur méthode peut prévoir rapidement le temps d'exécution global avec une précision de 30% sur 95% des cas. La difficulté de mener des micro-benchmarks précis a été citée comme le principal facteur limitant la précision globale de modélisation.
En 2000, Bourgeois [12, 55] a repris l'idée de Saavedra et l'a appliqué à des programmes parallèles MPI. Son outil appelé *ChronosMix* s'appuyait sur différents modules et permettait d'avoir rapidement une évaluation des performances d'un programme.
Un micro benchmark était exécuté sur la machine cible pour déterminer le temps d'exécution des instructions de bases (module MTC pour *Machine Time Characteriser*). Les temps de communications des primitives MPI étaient obtenus via l'exécution de script en utilisant l'outil SKaMPI (module CTC pour *Communication Time Characteriser*). Ces informations permettaient de caractériser l'architecture cible. Le programme cible était analysé par l'outil Sage++ [9], qui décomposait le code en instructions élémentaires via une analyse statique. A l'intérieur du module *PIC* (*Program Instruction Characteriser*), l'ensemble des informations étaient réunies et insérées dans un simulateur qui fournissait une évaluation pouvant être visualisée avec l'outil ParaGraph [43]. Si des données étaient manquantes pour la simulation, une exécution du code était effectuée afin de les récupérer. Cette approche permettait d'avoir d'un coté une caractérisation de l'application et de l'autre de l'architecture. Pour avoir une évaluation sur une nouvelle architecture, il suffisait de relancer uniquement la partie *MTC* et *CTC* alors que pour avoir l'évaluation d'un nouveau programme sur une architecture benchée, il suffisait de relancer la partie PIC. De plus, l'approche permettait l'intégration de l'hétérogénéité des noeuds au sein du processus de prédiction.

## 3.5 Vue d'ensemble des approches

Le nombre de techniques de modélisation de performance de programme parallèle est conséquent, et il est impossible de toutes les nommer. Certains modèles sont spécifiques à une architecture [58, 119], ou à un problème particulier qui touche les différentes classes de Cluster [83, 120]. Néanmoins, les techniques présentées dans ce chapitre sont suffisantes pour introduire les difficultés qui sont liées à la modélisation des performances des programmes parallèles sur un large éventail d'architectures. A l'issue de ce chapitre, nous pouvons retenir que :

- Il existe un réel besoin de développer des méthodes pour évaluer les performances des programmes parallèles. Mais la complexité croissante des architectures fait qu'il devient de plus en plus difficile de produire un modèle pouvant être utilisé pour tous les types de programmes sur l'ensemble des architectures.
- Les caractéristiques qu'une technique de modélisation de performance doit posséder sont la généralisation, la compréhension, la précision, la robustesse et le coût.
- Il existe différentes sources impactant sur les performances d'un programme. Parmi

celle-ci, il est possible de citer les problèmes d'accès concurrents aux ressources ainsi que de synchronisation. Ces problèmes peuvent être difficile à modéliser.

## 3.6 Discussion

| Critère \ Outil | POEMS | WARPP | Chronosmix |
|---|---|---|---|
| Temps de calcul | Graphe de Tâches | Trace d'exécution | Micro Benchmarking |
| Temps de communication | Modèle | Resultat IMB | Résultat SkaMPI |
| Contention | non | non | partiel |

TABLE 3.1 – Comparaison des benchmarks existants

Les différentes techniques présentées ici montre la diversité des approches utilisées pour estimer les performances d'un programme parallèle. Néanmoins, si l'on se concentre sur les environnements de prédiction, il est possible de voir via le tableau 3.1, qu'aucun des modèles présentés ne permet d'extrapoler le temps d'un programme sur une architecture multi-cœur, tout en prenant en compte le problème du partage de la bande passante.

Au niveau des temps de calcul, l'idée de Saavedra permet d'avoir une estimation tout en étant proche du code. Néanmoins, les optimisations des compilateurs font, avec un tel niveau d'abstraction, que l'estimation serait très éloignée de la réalité. Avoir une estimation à un niveau englobant plus d'informations permettrait de s'en rapprocher. L'idée de cette thèse est donc de caractériser le temps d'exécution des applications en se basant sur un benchmarking, non pas des instructions, mais de blocs d'instructions.

Au niveau des communications, l'approche par benchmarking de WARPP et de Chronosmix présente des lacunes car elle ne permet pas de caractériser proprement l'impact de la contention et de son surcoût engendrées. La seule solution avec ces outils serait de tester tous les schémas de communications possibles, ce qui ferait un nombre de test très important. Néanmoins, l'utilisation du benchmarking permettrait d'observer les effets de la concurrence sur le temps des communications. Malheureusement, les outils utilisés, à savoir IMB et SkaMPI, ne permettent pas une estimation complète des effets de la concurrence sur *InfiniBand*. L'un et l'autre ne donnant aucune information sur une éventuelle concurrence au niveau du routage. L'utilisation d'un nouvel outil de benchmarking sera donc nécessaire pour permettre l'établissement d'un modèle de répartition de bande passante. Un tel modèle couplé à un modèle linéaire permettrait de pouvoir calculer les temps de communication quel que soit le schéma de communication.

Dans le chapitre 4, nous utiliserons une méthode basée sur l'analyse et le découpage du code source en bloc afin d'estimer les temps de calcul. Le chapitre 5 détaillera un mo-

dèle de répartition de bande passante basé sur une analyse détaillée des communications concurrentes sur *InfiniBand*.

# II

# Modélisation des applications parallèles

*Un programme parallèle durant son exécution effectue des phases de calcul et de communication. Ces phases sont bien distinctes dans le cas de communications synchrones, avec une transition d'une phase de calcul vers une phase de communication synchrone et inversement. C'est en partant de ce principe que nous avons décidé d'étudier d'un coté l'estimation des temps de calcul et de l'autre l'estimation des temps de communications.*

*Cette seconde partie est dédiée à la modélisation des temps de calcul et de communication.*
*Dans le premier chapitre, nous estimerons les temps de calcul en nous basant sur une analyse statique du programme parallèle et un processus de micro-benchmarks. L'analyse statique a l'avantage d'être particulièrement rapide et nous donnent une représentation paramétrique du modèle. Ceci signifie qu'un modèle statique se définit en fonction de plusieurs paramètres qui régissent l'exécution du programme. Ces paramètres peuvent ensuite être modifiés sans avoir recours à nouveau à un processus de modélisation.*
*Dans le second chapitre, nous étudierons le comportement des communications sur InfiniBand lorsqu'elles sont mises en concurrence. Pour cela, nous utiliserons un ensemble d'expériences, qui nous permettrons de mettre en valeur le retard des communications lié au coût de ces accès concurrents. En se basant sur ces observations, un modèle sera construit, permettant de prédire ces pénalités en fonction des schémas de communication. La méthodologie mise en place permettra de pouvoir extrapoler les performances des applications parallèles, et ainsi de dimensionner au mieux les ressources.*

# Modélisation des phases de calcul 4

## 4.1 Introduction

Une méthode d'évaluation de performance d'une application se doit d'être rapide et précise pour être efficace. Trouver une méthode englobant un maximum de paramètres, et donnant une bonne estimation des performances d'une application, à l'intérieur d'une grappe de calcul, est très complexe du fait de la multiplicité des éléments entrant en ligne de compte.
La figure 4.1 montre quelques éléments impactant sur le temps d'exécution d'un programme. Tout d'abord, il y a la localité des données. En effet, une donnée placée en mémoire local demandera un temps d'accès plus long par rapport à une donnée déjà présente dans le cache. Autre élément important, la bande passante mémoire qui permet de connaître la quantité de données pouvant être transférée par unité de temps. De plus, dans un environnement multi-cœurs, il peut arriver qu'une ressource comme le stockage ou l'accès à une donnée soit sollicitée par plusieurs cœurs créant des phénomènes de congestion qui augmentent leurs temps d'accès mais aussi le temps d'exécution. Pour réduire les accès à la mémoire centrale, des mécanismes matériels existent dans le processeur tel le préchargement, qui consiste à précharger les données dans les caches. Un autre mécanisme existant est la prédiction de branchement, qui consiste à réduire la perte de cycle induite par les branchements conditionnelles. Tous ces éléments, ainsi que d'autres, influent sur l'exécution d'un programme parallèle. L'objectif de nos travaux a donc été de trouver une méthode pouvant englober un maximum de ces éléments tout en ayant un niveau d'abstraction relativement élevé.

L'idée principale de la méthode mis en place, pour la modélisation des phases de calcul, se base sur une approche par analyse statique via une méthode de découpage de code (*program slicing)* [86, 104]. En effet, comme le montre la figure 4.2, durant la phase de calcul, entre deux communications, il est possible de découper le code en un ensemble de blocs d'instructions. Pour cela, l'outil ROSE, décrit en section 4.2, est utilisé permettant ainsi d'obtenir ces décompositions. Cette méthode permet d'avoir une analyse du code indépendante de l'architecture. Suite à cela, une série de micro-benchmarks de ces blocs est effectuée sur un nœud de la grappe de calcul afin d'estimer le temps de calcul. L'idée

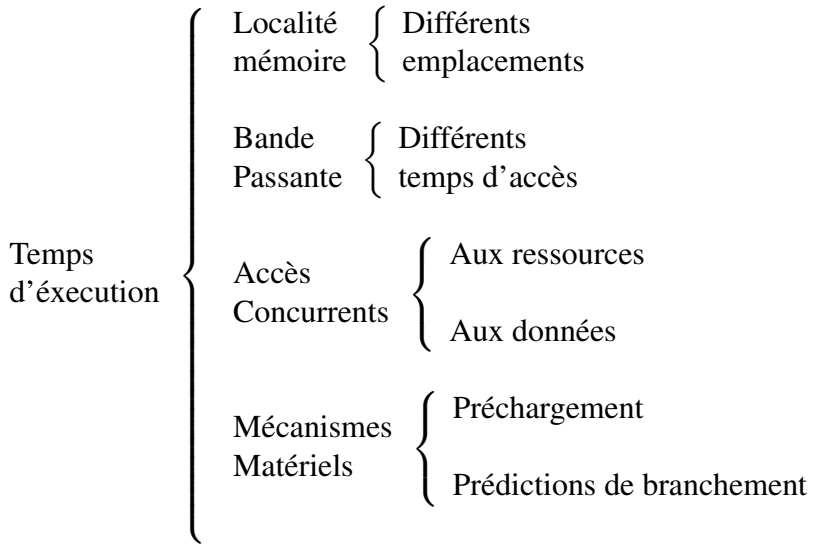



FIGURE 4.1 – Exemple d'éléments impactant sur le temps d'exécution

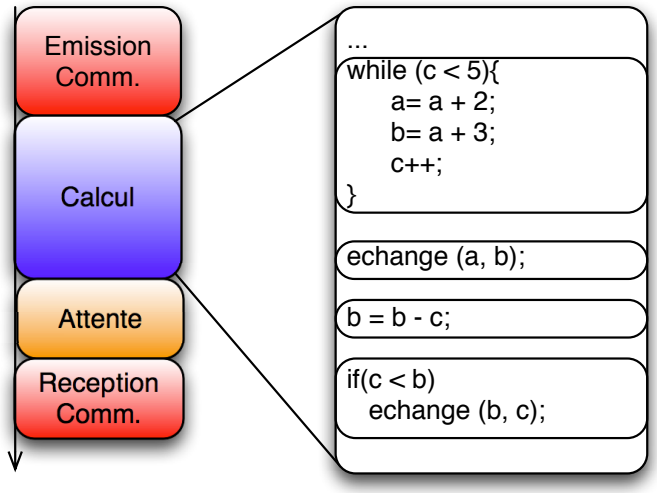



FIGURE 4.2 – Découpage de la phase de calcul

étant de profiter de l'homogénéité des grappes de calcul tout en ayant un niveau d'abstraction suffisant pour avoir un aperçu des performances du programme sur l'architecture cible.
Ce chapitre sera décomposé en quatre parties. Tout d'abord, l'outil Rose [85, 91] sera présenté, ainsi que les différentes représentations utilisées (section 4.2). Ensuite, des observations préliminaires (section 4.3) seront présentés afin d'introduire le benchmarking

par blocs. Puis il sera décrit, en section 4.4, le formalisme de la modélisation utilisé. Pour finir, en section 4.5, l'impact du compilateur, du niveau de compilation sur la modélisation proposée ainsi que l'effet de l'introduction d'un contexte pour améliorer la précision seront étudiés.

## 4.2 Découpage de code et l'outil *ROSE*

L'analyse statique consiste à étudier un programme sans l'exécuter au travers de l'étude de son code source. Pour cela il existe des techniques telles que le découpage de programme (*program slicing*) qui permettent de décomposer le code sous forme de représentations. Pour obtenir ces représentations et les exploiter, l'outil *ROSE* [85, 91] a été utilisé. Les quatre types de représentations qui seront utilisées à savoir l'*AST*, le *DDG*, le *CDG* et le *SDG*, seront présentées dans cette partie.

### 4.2.1 ROSE

Développé au *Lawrence Livermore National Laboratory* (LLNL), ROSE est un outil open source permettant l'analyse et la transformation de programme dans différents langages de programmation. Les langages supportés sont le Fortran 77/95/2003, le C et le C++. La transformation de code consiste à retranscrire un programme écrit dans un langage (par exemple, en Fortran 77) dans un autre langage (par exemple, en C). L'analyse de code via *Rose* peut être soit pour une analyse statique, soit pour des optimisations ou encore pour de la sécurisation de code. *Rose* est un projet très dynamique et complet qui a notamment reçu le prix *2009 R&D 100*.
Le choix de *Rose* est dû à plusieurs facteurs. Tout d'abord, le projet Rose est un projet actif et ouvert à la communauté, il est donc très facile d'avoir rapidement des réponses sur les détails techniques. Ensuite, contrairement à un grand nombre de logiciels d'analyse statique qui sont spécifique à un langage, l'outil *Rose* supporte tous les langages rencontrés dans l'univers du calcul haute performance. Bien que les résultats du découpage de code soient légèrement différents en fonction du langages, 95% de la structure reste commune. Enfin, les découpages proposés par *Rose* sont simples et facilement parcourables grâce à un ensemble de procédures mis à disposition.

### 4.2.2 Les différentes représentations d'un programme

Pour expliquer le rôle des différentes représentations générées dans *ROSE*, il sera utilisé un simple programme dont le code source peut être vu sur la figure 4.3.

```
#include<stdio.h>

int add( int x, int y){
   return x+y;
}

int main( ) {
  int a=12;
  int b, c, d, f, e, g;

  b = a / 4;
  c = a  * 3;
  if ( a < b )
     d  = a - 2;
  else
     d = a / 4;
  f = 4;
  while( f != 0){
     e = b * c;
     f --;
  }
  f = a + b + c + d + e;
  g = add(a, f);
}
```

FIGURE 4.3 – Code d'exemple

#### 4.2.2.1 L'Abstract Syntax Tree (AST)

L'*AST* est une représentation abstraite fondamentale du programme. Il existe un *AST* par fichier inclus dans le programme source. Via *Rose*, il est possible de concaténer les différents *AST* obtenus, pour les programmes comportant plusieurs fichiers, afin d'obtenir un *AST* global [80]. Cela permet ainsi l'étude complète de gros projets comportant plusieurs fichiers et des milliers de lignes de code. Cette représentation est facilement analysable, et durant son parcours, des transformations peuvent être envisageables. C'est grâce à cette représentation que *Rose* peut effectuer ces transformations de code source en un nouveau code source (sous un autre langage de programmation). Un aperçu de la forme de l'*AST* de notre programme peut être vu sur la figure 4.4.

Dans notre approche, l'*AST* est utilisé pour identifier les éléments clés comme les blocs et les instructions *MPI*. Si notre programme ne comprend qu'un seul fichier avec une seule fonction, et qu'à l'intérieur de cette fonction, il n'existe aucune dépendance, de

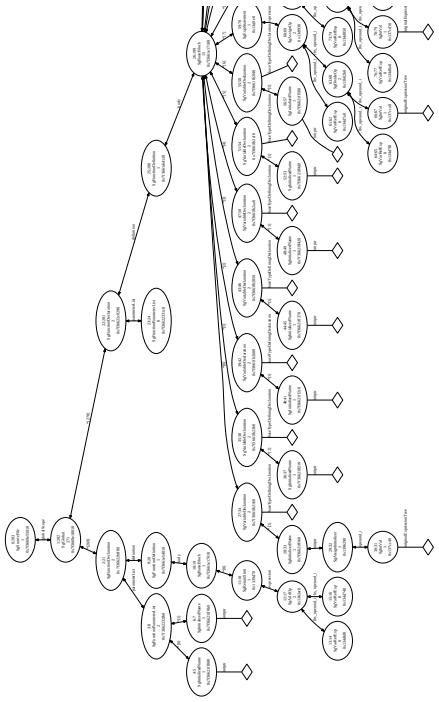


FIGURE 4.4 – AST obtenu avec *Rose*

boucle ou de condition, cette représentation est suffisante pour une analyse statique. Si ce n'est pas le cas, les autres représentations décrites ci-dessous sont nécessaires.

#### 4.2.2.2 Le Data Dependance Graph (DDG)

Il est important de connaître le parcours des données dans chaque fonction. Par parcours, nous parlons du moment où elles sont définis jusqu'au moment où elles sont utilisées. Le *DDG* [106] permet de voir très précisément le parcours des données à l'intérieur de la procédure où elles sont définies, et donne la possibilité au compilateur de faire l'analyse hiérarchique de la dépendance des données. Cette analyse aide, par exemple, le compilateur à faire des optimisations comme la substitution des données par leurs valeurs. Dans notre cas, elle permet d'identifier le parcours des variables et de trouver la valeur de certaines d'entre elles. Il y a un *DDG* par procédure. La figure 4.5 montre le graphe de dépendances des données et les relations entre les données $a$, $b$, $c$, $d$, $e$, $f$ et $g$ pour la fonction *main*.

#### 4.2.2.3 Le Control Dependence Graph (CDG)

Le *CDG* s'intéresse au contrôle mais aussi aux relations entre les différents éléments de la fonction et nous informe sur leur condition d'exécution dans le programme. Dans notre cas, cela permet de connaître les différents blocs accessibles suite à une condition, par exemple. Il y a un *CDG* par procédure.

#### 4.2.2.4 Le System Dependence Graph (SDG)

Une fois le *CDG* et *DDG* obtenus, ceux-ci sont fusionnés afin d'obtenir le *Program Dependence Graph*(*PDG*) [53]. L'avantage du *PDG* est qu'il donne à la fois les relations de dépendances entre les contrôles mais aussi des données au sein des procédures. Ces différentes relations sont traduites par des arrêtes dans le graphe représentant le *PDG*. Le problème de cette représentation est qu'elle ne donne aucune information sur les paramètres des appels de ces procédures.
Introduit par Horwitz [50] pour les programmes comportant plusieurs procédures, le *SDG* est construit à partir de la fusion des *PDG*. Le *SDG* regroupe les données, les structures de contrôles ainsi que les relations de dépendance entre les différentes procédures au sein du programme. Ce graphe est très similaire au *PDG*. En effet, le *PDG* de la fonction *main* est un sous-graphe du *SDG*. En d'autre mot, un programme sans appel de procédure aura son *PDG* et son *SDG* identique.

La représentation d'un *SDG* comporte plusieurs types de sommet pour représenter les appels de procédure et leurs paramètres, parmi ceux-ci il y a :

**Les sommets *Entry*** représentent les procédures dans le programme.

**Les sommets *Actualin* et *Actualout*** représentent les paramètres d'entrée et de sortie de la procédure appelante. Ils sont dépendants des contrôles (control dependent) des sommets qui appels la fonction.

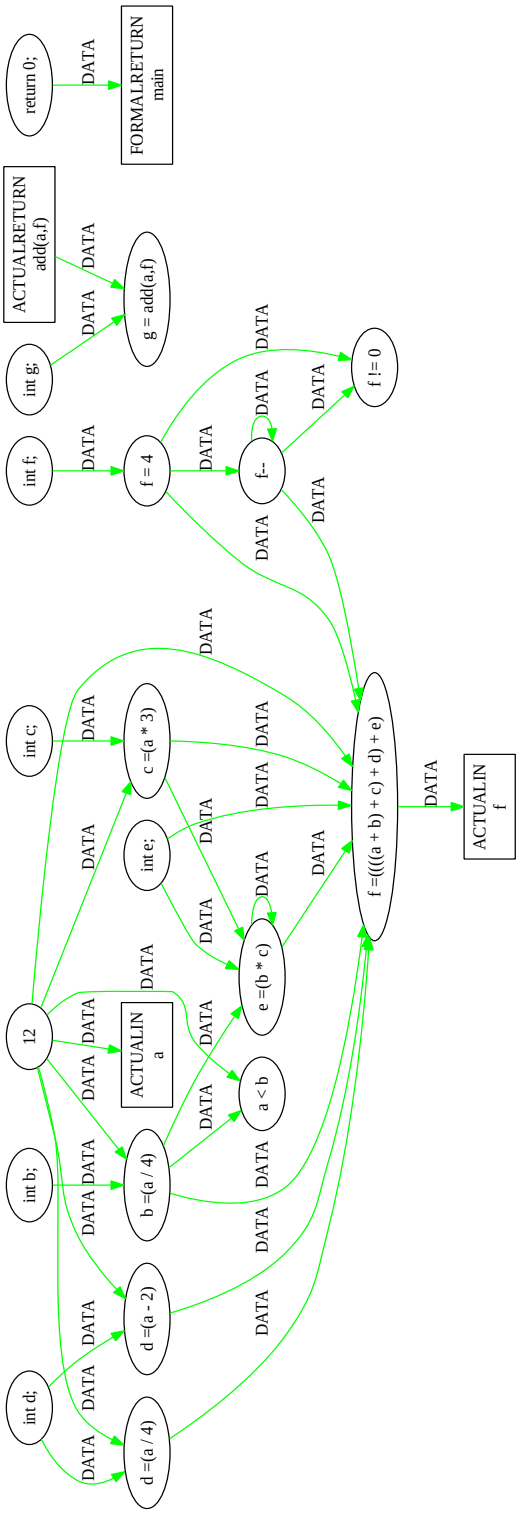



FIGURE 4.5 – DDG de la fonction *main* obtenu avec *Rose*

**Les sommets *Formalin* et *Formalout*** représentent les entrées et les sorties des données de la procédure appelée.

Les arcs de dépendances de données et de contrôles sont utilisés pour lier les *PDGs* dans le SDG. Ces arcs sont :

**Les arcs Call** relient l'appel de la fonction à son sommet *Entry*.

**Les arcs *Paramter_in*** lient les sommets *Actualin* avec les sommets *Formalin*.

**Les arcs *Paramter_out*** lient les sommets *Actualout* avec les sommets *Formalout*.

Toutes ces informations permettent de pouvoir propager les constantes, de trouver les dépendances et les relations entre les différentes procédures présentes au sein du programme. La figure 4.6 illustre le SDG de notre exemple.

## 4.3 Observations préliminaires

L'idée principale des travaux présentés est de décomposer le code source en blocs d'instructions grâce aux informations données par le découpage, puis d'effectuer des mesures sur ces blocs afin d'estimer leurs temps d'exécution. Le formalisme de la procédure de découpage sera décrit en section 4.4. Mais avant cela, il est intéressant de voir un ensemble de petites observations basées sur le micro-benchmarking d'une boucle. L'idée étant d'observer l'évolution du temps d'une boucle et de regarder les différents éléments pouvant influencer son temps d'exécution.

### 4.3.1 Protocole expérimental

#### Objectifs

L'objectif de ces expériences est d'observer l'évolution du temps de l'exécution d'une boucle en faisant varier le nombre d'itérations puis d'étudier l'impact du compilateur et des options de compilation sur cette évolution.

#### Grappes de calcul

Les grappes de calcul utilisées pour ces expériences, sont à base de nœuds *Novascale* composés de processeurs multi-cœurs Core 2 quad ou Nehalem.

#### Programme utilisé

Le protocole expérimental est implanté par un script développé en collaboration avec Cornea B. de l'équipe *OMNI* du *LIFC* et mesurant le temps d'exécution d'une boucle en fonction du nombre d'itérations via l'utilisation de la bibliothèque *PAPI* 3.7.2. Les

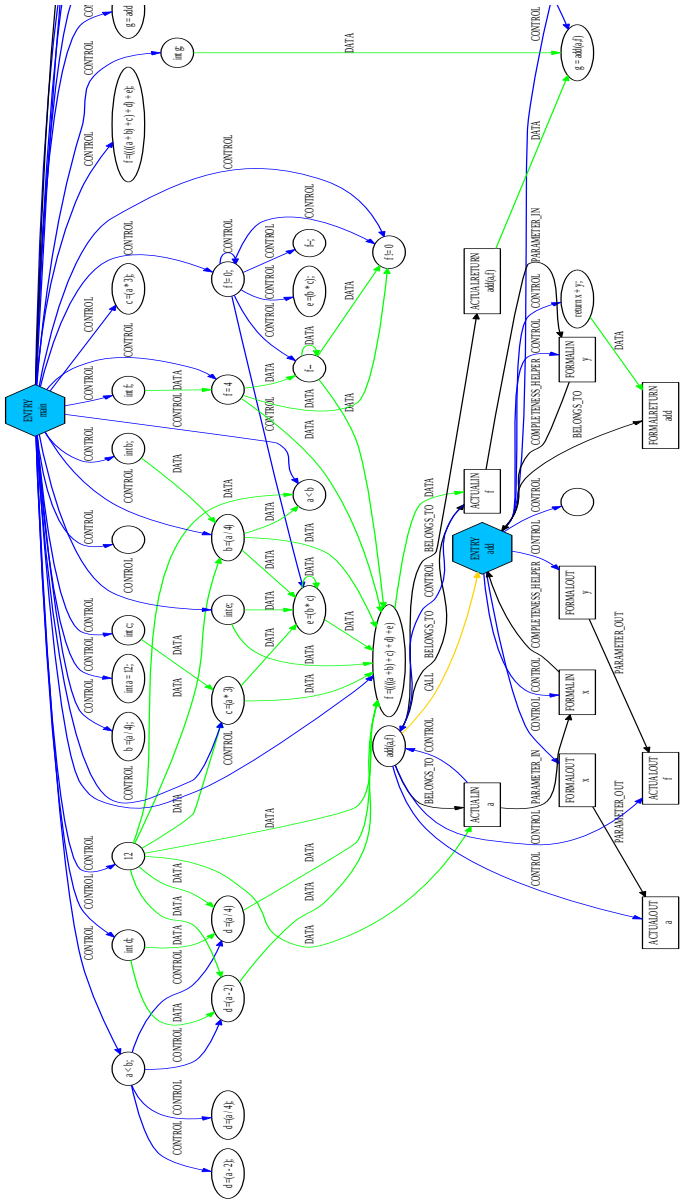



FIGURE 4.6 – SDG : Les relations entre les procédures et leurs dépendances apparaissent

compilateurs utilisés sont *icpc* 11.1 et *g++* 4.4.1. Les expériences sur Nehalem ont été réalisées en collaboration avec Cornea B. de l'équipe *OMNI* du *LIFC*.

### 4.3.2 Les temps de chargement

Dans la première expérimentation, le compilateur *Intel* icpc a été utilisé sans optimisation (option -O0). On observe l'évolution du ratio entre le temps d'exécution d'une boucle, en nanosecondes, et le nombre d'itérations sur un processeur *Core 2 quad*. La prise de mesure est effectuée avant le début de la boucle et se finit lorsque à la sortie de celle-ci. La figure 4.7 montre le résultat.

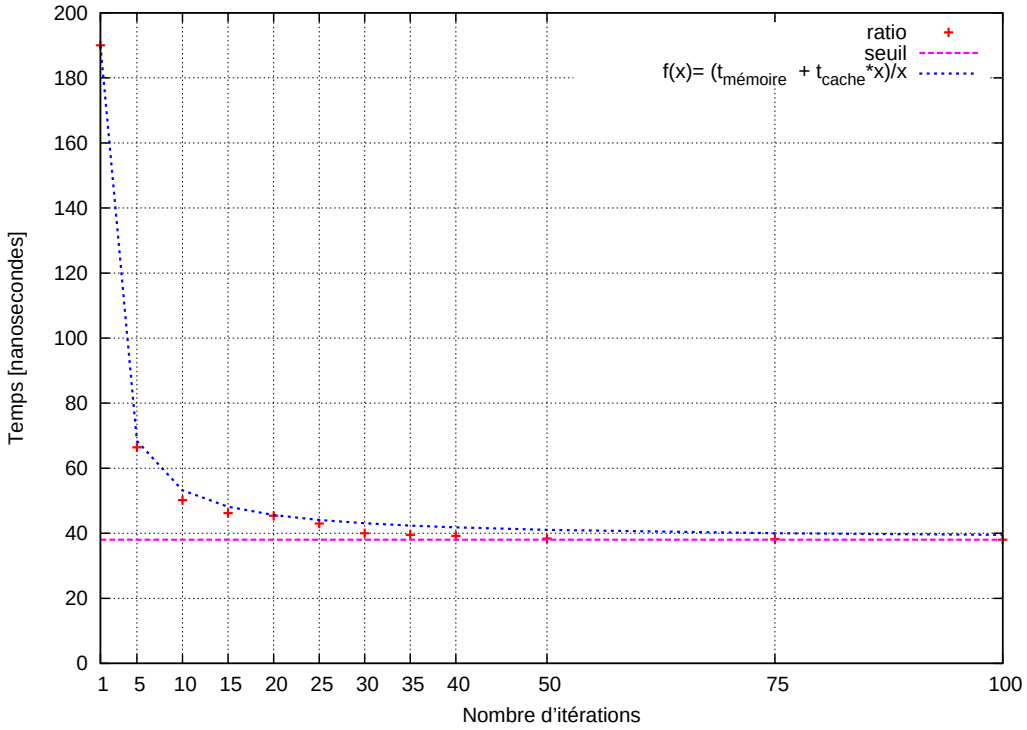



FIGURE 4.7 – Evolution du ratio entre le temps d'execution d'une boucle et le nombre d'itérations sur Core 2 quad

Tout d'abord, nous remarquons que ce rapport évolue très rapidement jusqu'à un seuil. Sur la figure 4.7, ce seuil est de 38 nanosecondes.
En effet, pour une seule itération, le processeur doit charger les données présentes en mémoire dans ces caches. Or le coût de ce chargement, à l'échelle de la nanoseconde, est assez important. Mais une fois que ces données sont chargées dans le cache, elles ne sont pas déchargées jusqu'à la fin de la boucle et donne donc un accès aux données beaucoup plus rapide au processeur. Ce phénomène est appelé "l'effet de cache". Il faut donc un certains nombre d'itérations, pour que le coût du temps d'accès à la mémoire, soit minoré

par le temps d'exécution total de la boucle. La figure 4.8 [1] illustre le même principe sur une architecture Nehalem.

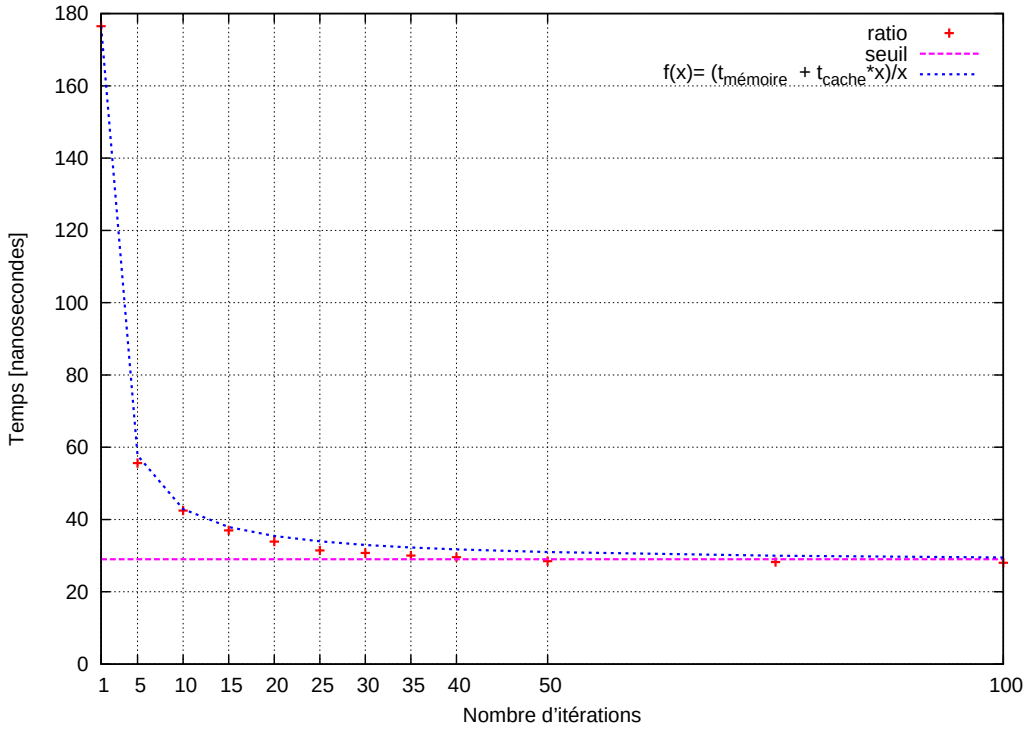



FIGURE 4.8 – Evolution du ratio entre le temps d'une boucle et son nombre d'itération sur Nehalem

Bien que le temps de chargement et le seuil soient différents, le comportement observé est identique pour ces deux architectures.

Ce processus peut se mettre sous forme d'équation, ce ratio pourrait donc être calculé par l'équation $f$ :

$$f(nombre_d'itération) = \frac{Temps_mémoire + Temps_cache \times nombre_d'itérations}{nombre_d'itérations}$$

Cette équation est représentée sur les figures 4.7 et 4.8 par la courbe $f(x)$. Il est facile de remarquer que le ratio et la courbe tracée sont relativement proche. Ainsi, si l'on souhaite connaître le temps d'exécution d'un bloc contenu dans une boucle, il suffit de pouvoir connaître le temps de chargement des données de la mémoire et de son temps d'accès une fois dans le cache.
Maintenant que nous avons observé ce phénomène, il nous faut regarder l'impact des

1. Expérience réalisée par Cornea B. de l'équipe *OMNI* du *LIFC*

options de compilations, et si le même phénomène est observé avec un autre compilateur *gcc*.

### 4.3.3 L'impact du niveau d'optimisation du compilateur

Il est possible d'optimiser un code lors du processus de compilation. Néanmoins vu que le nombre d'options d'optimisation est important, il ne sera considéré ici, que les options d'optimisation de base $-Ox$. Il existe quatre niveaux d'optimisations :

**-O0** : Ce niveau n'effectue aucune optimisation

**-O1** : Ce niveau optimise la taille du code et la localité des données. Il peut augmenter les performances d'application ayant un grand nombre de ligne de code, plusieurs branchement mais où le temps d'exécution n'est pas dominé par le temps des boucles.

**-O2** : C'est le niveau par défaut. Optimise la vitesse d'exécution du programme.

**-O3** : Il reprend les optimisations de -O2 tout en étant plus agressif sur l'optimisation des boucles, des accès mémoire et le prefetching. Cette optimisation ne donne pas forcément de meilleures performances que l'option -02 et est principalement recommandée pour les applications ayant un grand nombres de boucles, une utilisation massive des nombres à virgule flottante ou utilisant des données de grande taille.

La figure 4.9 montre le résultat de l'expérimentation. Globalement, le comportement est identique. Plus le nombre d'itération est grand, plus le rapport étudié tend vers un seuil. Bien sûr, ce seuil n'est pas forcément identique en fonction des optimisations. Concernant le temps de chargements, ce dernier est différent suivant les optimisations. Il est à noter, pour les options d'optimisations -02 et -03, que le ratio est très bas entre 4 et 14 itérations. Ce phénomène sera expliqué en 4.5.3.
La figure 4.10 montre les résultats obtenus avec le compilateur *g++* sur la même plateforme.

Le premier constat est que l'on peut voir qu'il y a aussi une grande irrégularité dans les mesures. Ces irrégularités ne sont pas typique de *g++* car elles apparaissent aussi sur les expériences avec le compilateurs *Intel*. Mais comme précédemment, plus le nombre d'itérations est important plus le rapport tend vers une valeur limite. Il est a noter qu'en règle général, les performances du compilateur *gcc/g++* sont moins bonnes que *icc/icpc*. Cela est dû au fait que les optimisations sont moins poussées, en effet *gcc* ne contrôle pas aussi bien les optimisations hardware que le compilateur d'*Intel*.

### 4.3.4 Discussion

Ces expérimentations permettent de constater que le temps d'exécution d'une boucle a un comportement identique en fonction du compilateur et de ses options. De plus, il a été possible de caractériser ce temps par deux paramètres : le temps de chargement des données de la mémoire et le temps de chargement des données situées dans le cache. Les

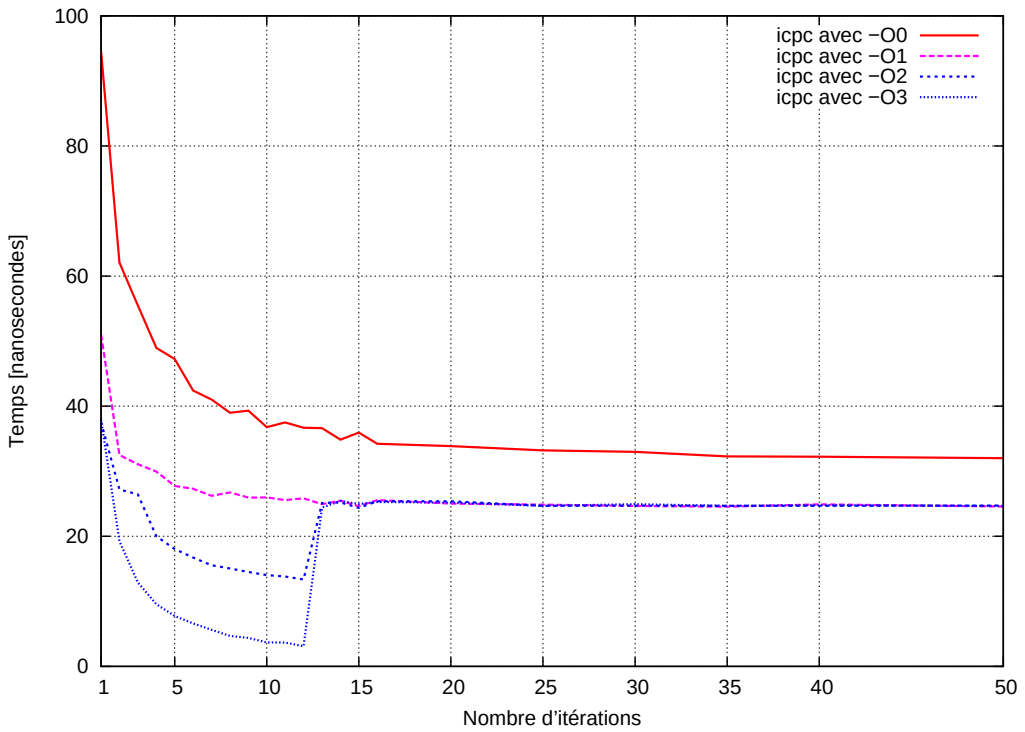



FIGURE 4.9 – Evolution du ratio suivant les options de compilation sur Core 2 quad avec icpc

options de compilation impactant sur ces deux paramètres.
L'idée étant de modéliser la partie calcul d'un programme en fonction de ces deux paramètres. Pour cela, un ensemble de scripts permettant d'obtenir ces deux valeurs ont été développés. Ces scripts utilisent le découpage du programme en blocs fourni par *Rose* pour obtenir ces différents temps de chargement. Néanmoins en fonction de la structure du bloc, leurs utilisations sont différents. La section suivante abordera la modélisation des différentes structures existantes en fonction de ces deux paramètres.

## 4.4 Formalisme de la modélisation

Pour commencer, il faut considérer que le temps de calcul du code entre chaque communication est statiquement déterminer par une séquence de blocs entre chaque appel *MPI*. Entre chacune de ces séquences de blocs, les appels *MPI* peuvent être sujet à contention.
Bien que cette contention soit dynamique, ces effets sur la performance peuvent être simuler en considérant d'un coté le modèle qui sera mis en place dans le chapitre 5 et de l'autre le fait que chaque appel *MPI* peut être une source de non déterminisme, qui peut

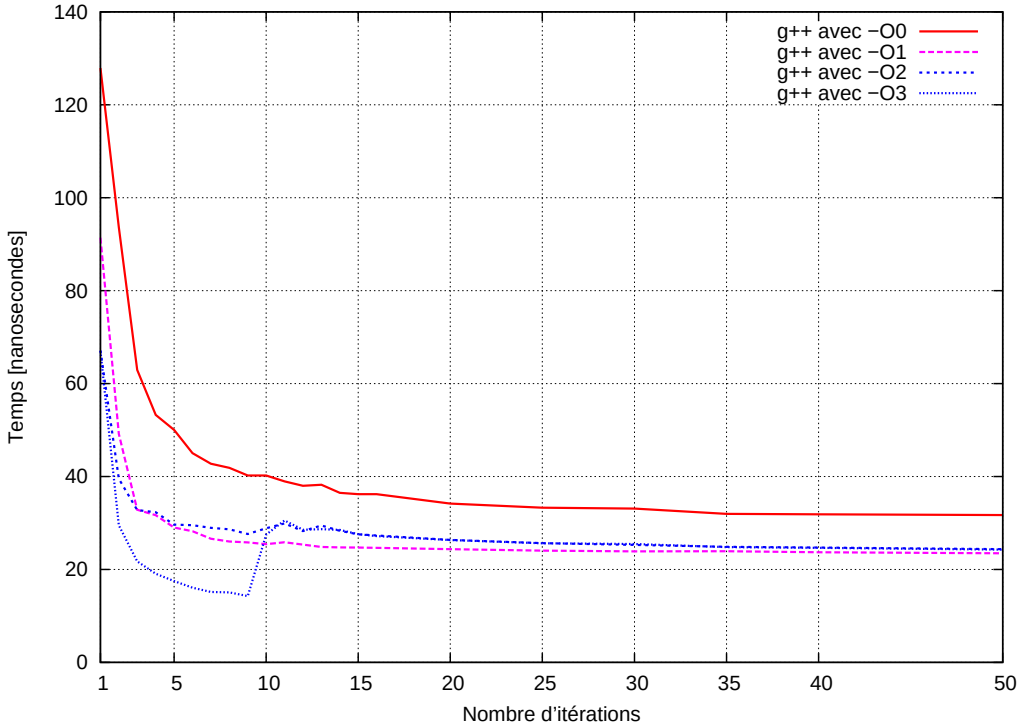



FIGURE 4.10 – Evolution du ratio suivant les options de compilation sur Core 2 quad avec g++

affecter le cours de l'exécution du programme. Pour appliquer cette abstraction sur des programmes parallèles, il nous a fallu décomposer le code source en blocs d'instructions et en appel *MPI* via utilisation de l'outil Rose.
Bien que les modèles décrits pour estimer le temps de calcul des blocs peuvent servir pour l'ensemble des langages de programmation procédurale, le langage de programmation C sera utilisé pour faciliter la compréhension.
Il y a peu de structures de code qui composent la base de ces langages. Il y a les instructions simples, les boucles, les structures conditionnelles et les sous-routines. Dans le modèle de performance qui a été développé au travers de ce document, les appels aux fonctions *MPI* sont considérés séparément. Cette décision affecte la manière d'étudier les autres structures (hormis les instructions simples), car il peut y avoir des appels *MPI* à l'intérieur de ceux-ci. Ceci nous a amené à subdiviser ces structures en plusieurs entités qui seront séparées par des appels *MPI*. Cette section présentera le modèle mis en place pour chacune des structures de code et les appels *MPI* associés.

### 4.4.1 Instructions simples et blocs de base

En C, une instruction est une expression suivie d'un point virgule. Les expressions sont une séquence d'opérateurs et d'opérandes qui sont utilisés pour faire un calcul. Par exemple, le code :

$$a = b + c;$$

Ce fragment de code est une instruction simple, où l'expression $a = b + c$ a comme opérande $a$ et $b$ et pour opérateur $+$. Nous pourrions dire, en nous appuyant sur les travaux de Saavedra (Section 3.4.3), que le temps d'exécution $T$ d'une succession d'instructions, qui formerait un bloc d'instructions, pourrait être réduit à :

$$T_{bloc} = T_{instruction_1} + \ldots + T_{instruction_n}$$

Mais les techniques de compilations ont progressé, rendant les compilateurs plus "intelligents". Lors du processus de compilation, l'ensemble des instructions jugées inutiles sont supprimées et le code optimisé. C'est pourquoi, il est important de rassembler les instructions simples en blocs.
Une fois rassemblées en blocs, il faut pouvoir estimer le temps d'exécution de ces instructions. Il existe des techniques hardwares et softwares permettant le pré-chargement des données afin de réduire ce temps de chargement. Néanmoins différentes expériences ont montré que, pour des blocs d'instructions de grande taille, le temps de chargement des données de la mémoire devait être pris en compte avec le temps de chargement des données du cache. Ainsi, le temps d'un bloc d'instructions peut être modélisé par :

$$T_{bloc} = Temps_mémoire + Temps_cache$$

Néanmoins, il nous faut prendre en compte les appels *MPI*. Bien que les appels *MPI* soient considérés comme des instructions, ils ne peuvent pas faire partie d'un bloc. Donc, si un appel *MPI* apparaît au sein d'un bloc d'instructions, ce dernier doit être séparé en trois sous bloc. La figure 4.11 illustre cette séparation.

Le temps d'exécution de l'ensemble de ces blocs pourrait être modélisé par l'équation :

$$T_{ensemble} = T_{segment_1} + T_{MPI_Send} + T_{segment_2}$$

Considérer le temps d'exécution ainsi est nécessaire car le temps d'achèvement de l'appel *MPI_Send* ($T_{MPI_Send}$) dépend du niveau de contention du réseau au moment de l'appel. En utilisant cette technique, l'instant de départ du *MPI_Send* est conservé. Cette information est nécessaire pour notre simulateur afin de connaître le nombre de messages en transit simultanément. Nous pouvons généraliser le temps de n'importe quel appel *MPI* par :

$$T_{(appel_MPI)} = T_{(appel_MPI)(arguments)}$$

Ainsi, le temps des appels *MPI* sera déterminé dynamiquement et dépendra du type d'appel, du moment de l'initialisation, de l'origine, de la destination ainsi que du niveau

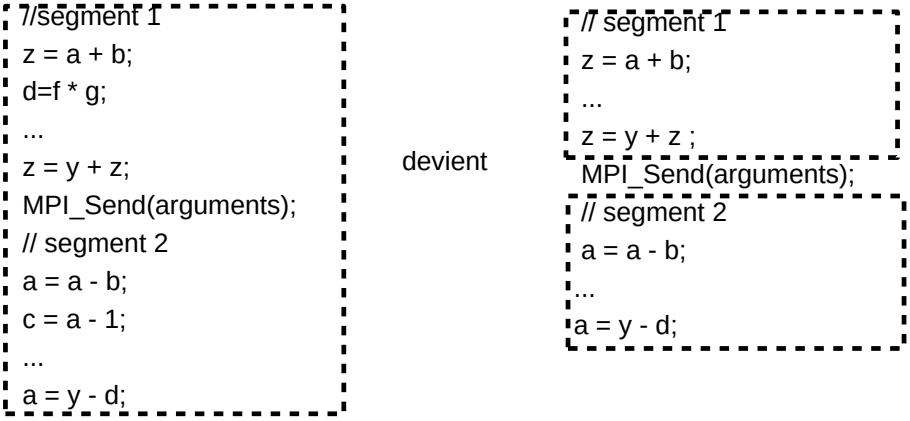



FIGURE 4.11 – Décomposition des blocs en sous-blocs

de contention.

### 4.4.2 Les structures de boucle

Les boucles permettent aux programmeurs de répéter un ensemble d'instructions au travers d'un intervalle défini dans le cas d'une boucle de type *loop*, ou tant qu'une certaine condition n'est pas valide dans le cas d'une boucle de type *while* ou *do... while*.
En règle générale, la structure de boucle *loop* est utilisée dans des situations qui demandent un nombre explicite de répétition, par exemple, lors du parcours de tous les éléments d'un tableau. À l'inverse, la structure de boucle *while* est utilisée quand le nombre d'itérations n'est pas statiquement déterminé, comme, par exemple, lorsqu'on utilise une méthode itérative de calcul qui doit se répéter jusqu'à ce qu'une certaine précision soit atteinte. Ceci étant aussi possible avec une structure de boucle *loop* en manipulant la condition de la boucle ou en utilisant la commande *exit*. Néanmoins, ces deux types de boucle peuvent être considérés comme équivalente à condition que le nombre d'itérations soit identique. Du point de vue de la modélisation, une description commune correspondrait au nombre d'itérations et, dans certains cas, à la valeur de la condition de la boucle au début de chaque itération. Une fois ces valeurs connues, la boucle peut très facilement être modélisée et l'intérieur de la boucle peut être considéré comme un bloc d'instructions. Bien sûr, si à l'intérieur de ce bloc, des appels *MPI* sont identifiés, il faudra décomposer celui-ci, comme nous l'avons vu précédemment.
Dans la partie 4.3, il a été montré que le temps moyen d'une itération d'une boucle variait en fonction du nombre de répétitions. Dans le cas d'une boucle sans appel *MPI*, le temps

de la boucle sera estimé en fonction du nombre d'itérations et des deux différents temps de chargement. Dans le cas d'une boucle comportant des appels *MPI*, la boucle sera déroulée et le temps de chacun des sous-blocs sera estimé comme des blocs de base. Bien sûr, il nous faut avant cela déterminer le nombre de répétitions de la boucle.
D'une façon générale, il est difficile de définir de façon statique le nombre d'itération d'une boucle, un problème bien connu est celui du critère d'arrêt (*halting problem*) [109]. Néanmoins, dans certains cas, et spécialement dans la façon dont les codes des programmes parallèles haute performance sont écrits, il est possible de définir le nombre d'itération. En effet, les programmes sont souvent construits afin de résoudre des problèmes spécifiques. Plus précisément, ils sont souvent écrits pour résoudre des problèmes spécifiques de taille définie. Ainsi, pour beaucoup de programmes parallèles haute performance, augmenter le nombre de processus *MPI* n'augmentera pas la taille du problème. Il en resulte que dans certains codes, le nombre d'itération des boucles *loop* est défini explicitement et ne dépend que de variables dynamiques. Dans ce cas, il est relativement simple de trouver, via l'étude de l'*AST* et du *SDG*, le nombre d'itération d'une boucle. Si ce n'est pas le cas, deux possibilités se présentent :

- La première est d'étudier le code source du programme et de déterminer le nombre d'itération de chaque boucle. Cela peut être une simple valeur ou bien un ensemble de conditions. Dans ce dernier cas, ces conditions pourraient être des paramètres du modèle d'exécution du programme.
- La seconde possibilité est d'utiliser une analyse semi-statique. Du fait que les programmes parallèles dans le calcul haute performance sont souvent déterministes. Il suffirait de faire une analyse post-mortem d'une trace de l'exécution, nous permettant ainsi de découvrir le nombre d'itérations de chaque boucle. Et cela, sur n'importe quelle grappe de calcul, vu que nous ne récupérerions pas des mesures mais uniquement des valeurs pour un nombre de tâches *MPI* définies.

Cette seconde approche est particulièrement appropriée pour les boucles de type *while*, car le nombre d'itérations requis ne dépend des données et n'est pas déterminable statiquement. Dans le cas de nos travaux, la deuxième possibilité a été le choix retenu.
Bien que cela nécessite l'exécution du code, cela est avantageux pour plusieurs raisons. Tout d'abord, cette approche couvre un plus grand nombre de programmes en tenant compte de la dépendance des données. Par exemple, cela nous permet de connaître le nombre d'itérations des codes s'arrêtant après avoir obtenu une certaine précision. De plus, si nous utilisons la première méthode, il peut être parfois difficile et long de trouver une valeur ou un ensemble de conditions pour paramétrer le nombre d'itérations de chaque boucle. Cette technique est beaucoup plus simple et rapide.
Pour conclure sur les structures de boucle, le temps de calcul d'une boucle $T_{loop}$, qui ne contient pas de message *MPI* et qui réalise le même nombre d'opérations par itération, peut se formaliser ainsi :

$$T_{boucle} = Temps_mémoire + Temps_cache \times n$$

avec $n$ le nombre d'itération.

### 4.4.3 Les structures conditionnelles

On appelle structure conditionnelle, les instructions qui permettent de tester si une condition est vraie ou fausse. Sémantiquement, une structure conditionnelle exécute le code si la condition est vraie. Comme pour les structures de boucles, la modélisation de ces structures nécessite de connaître la valeur des conditions. Encore une fois, le problème du critère d'arrêt fait, de façon général, qu'il est impossible statiquement d'établir la valeur de ces conditions. Heureusement, comme c'est le cas des boucles, la manière dont les structures conditionnelles sont souvent utilisée dans les programmes de calcul haute performance simplifie le problème dans beaucoup de circonstance. Les paragraphes suivants décrirons les différentes approches utilisées pour estimer ces conditions.

```
MPI_Comm_size(MPI_COMM_WORLD, &MPI_size);
MPI_Comm_rank(MPI_COMM_WORLD, &MPI_rank);

if(MPI_size == 1){
        // Bloc exécuté si il n'y a qu'un seul processus
        // dans le groupe
        ...
} else {
        // Bloc exécuté si il y a plus d'un seul processus
        // dans le groupe
}
...
if(MPI_rank == 0){
        // Bloc exécuté uniquement par le processus 0
        ...
} else {
        //Bloc exécuté par tous les autres processus MPI
        ...
}
```

FIGURE 4.12 – Exemple de résolution par analyse statique

Le premier cas étudié sera celui présenté par la figure 4.12. Ce cas, relativement fréquent dans les codes parallèles distribués, correspond aux conditions de branchement liées au code *MPI*. *MPI_size* correspondant au nombre de tâches *MPI* existantes, et MPI_rank correspondant au numéro de la tâche *MPI*. Ces deux variables ne sont pas, strictement parlant, des données statiques car leurs valeurs ne sont pas discernable du code source

seul. Néanmoins, elles ne sont pas dynamiques car leurs valeurs ne changent pas durant l'exécution. En réalité, ces valeurs sont automatiquement définies lors du lancement d'un programme *MPI*. Ainsi, si l'on connaît le nombre de tâches *MPI* qui vont être utilisée pour lancer le programme (*MPI_size*), il nous sera facile d'étudier chaque processus *MPI* (MPI_rank) l'un après l'autre, et de définir le parcours de l'exécution du programme pour chaque tâche. L'avantage de cette approche est d'être indépendante du nombre de tâche *MPI*, et de permettre l'étude des performances dans différentes circonstances.
Le deuxième cas apparaît lorsqu'aucune fonction *MPI* n'est contenue dans la condition. Dans ce cas, plusieurs choix s'offre à nous. Il y a d'abord, la détermination des conditions via une analyse statique, malheureusement, une approche simple est difficile car les conditions peuvent être dépendantes de variables calculées avant la condition. La deuxième solution consisterait à se servir de la trace effectuée pour retrouver la valeurs des inconnues, comme il a été vu pour les boucles. Actuellement, cette solution est mise en place. Mais d'autres pistes sont en cours d'étude, notamment les différentes recherches effectuées au niveau de la prédiction de branchement statique [30, 59, 124, 125]. L'idée étant d'utiliser différentes traces pour obtenir un modèle probabiliste sur les conditions de branchements, mais pour être valable, la sélection des traces est crucial dans la prédiction.

Pour conclure cette partie, le formalisme des branches conditionnelles pourrait se définir ainsi :

$$T_{condition} = c_1 : T_{bloc_1} | \ldots | c_n : T_{bloc_n}$$

Avec le temps de la condition égale au temps du bloc $i$ ($bloc_i$) qui aurait sa condition, $c_i$, valant vraie.

### 4.4.4 Les sous-routines

La dernière structure a étudier pour compléter ce formalisme et qui est nécessaire pour écrire des programmes procédurales est la sous-routine, appelée aussi sous programme, fonction ou procédure. La différence de nom n'est pas algorithmique mais dépend simplement du langage de programmation. De manière général une fonction renvoie une valeur alors qu'une procédure ne le fait pas.
Les sous-routines sont une abstraction utilisée par le programmeur pour clarifier le code afin de le rendre moins complexe ou pour éviter toute récurrence. Une sous-routine est constituée d'une partie déclarative où un nom d'appel est défini, d'une liste de paramètre qui sont passés à la sous-routine lors de son appel et d'un corps composé d'instructions avec optionnellement une valeur de retour. Plusieurs fonctions ont déjà été décrite précédemment. Par exemple, tous les appels *MPI* mentionnés sont des fonctions, plus exactement des appels de bibliothèque. Néanmoins, ces appels sont à part, car leurs temps d'exécution seront calculés par un autre modèle. D'une manière général, les appels de

bibliothèque sont principalement utilisés pour faire des opérations complexes mais définies, et sont par conséquent plus intéressants à être modélisés et considérés comme une entité indivisible. De plus, ces appels sont supposés être invisibles pour le programmeur, et celui-ci n'a pas à s'interroger sur les détails de leurs implémentations.
Tout comme la modélisation des boucles ou des structures conditionnelles, la modélisation des sous-routines peut s'avérer très lourde et difficile. Mais encore une fois, la manière dont les sous-routines sont utilisées dans le contexte du calcul haute performance simplifie le problème. La principale difficulté dans la modélisation des sous-routines apparaît quand des récursions sont présentes. La récursion se produit lorsque l'invocation d'une sous-routine amène à l'invocation de cette même sous-routine avant que la sous-routine originale s'achève. La compréhension de ces situations peut être déterminée par induction mais cela se révèle vite être une entreprise complexe. Heureusement, bien que la récursion puisse produire des solutions élégantes, elle est très rarement utilisée dans les programmes de calcul haute performance. Ceci est dû principalement au fait que la récursion tend à ajouter un surcoût dans le temps d'exécution du programme et que des solutions alternatives peuvent toujours être trouvée. Par conséquent, plutôt que de fournir une procédure complexe pour faire face à ce rare problème, la possibilité de récurrence sera exclus de notre technique de modélisation.
Sans récursion, la modélisation des sous-routines devient un problème trivial. Sémantiquement, une sous-routine sert simplement de transfert du point d'appel de la sous-routine au début du corps de celle-ci, où l'exécution se poursuit. Par conséquent, de part notre technique, la seule chose nécessaire est d'insérer l'ensemble des instructions de la sous-routines dans le bloc précédent l'appel (en faisant attention qu'il n'y ai pas d'appel *MPI* à l'intérieur). Cette technique est la même que celle existante dans tous les compilateurs C avec les fonctions $inline$. De ce fait, la modélisation des sous-routines et leurs estimations de temps d'exécution peuvent être effectuées via les techniques vu précédemment. Ceci peut être résumé par le formalisme :

$$T(sous - routine) = T_{corps\ des\ sous-routines}$$

Où $T_{corps\ des\ sous-routines}$ est évalué en appliquant les règles vu pour les blocs d'instructions, boucles et structures conditionnelles sur l'ensemble de la sous-routine.

## 4.5 Evaluation sur 10 blocs

Chaque optimisation du compilateur apporte son lot d'améliorations, complexifiant le code et parfois la prédiction. Un programme composé de 10 blocs est proposé en exemple, afin de voir l'impact des optimisations sur notre décomposition en bloc. L'exemple pris est donc un code composé de simples instructions ainsi que de boucles (code source en Annexe B). Ce choix n'est pas anodin. En effet, pour étudier l'impact des optimisations et des compilateurs sur un code, il nous faut un code qui puisse être optimisé au mieux

afin de connaître les limites de notre méthode. Or un code comportant un grand nombre de boucles sera bien plus optimisé en $-O2$ ou $-O3$ qu'en $-O1$. Dans cette partie, deux champs seront étudiés : l'impact du compilateur et du niveau de compilation sur la prédiction. L'évaluation du modèle de calcul se base sur la comparaison entre le temps prédit $T_p$ et le temps mesuré $T_m$.

$$E_{relative} = |\frac{T_p - T_m}{T_m}| \times 100$$

Toute l'étude a été effectuée avec Cornea B. de l'équipe *OMNI* du *LIFC* en utilisant un ensemble de scripts pour l'estimation des temps de chaque bloc. Les versions des compilateurs sont identiques aux expériences précédentes.

### 4.5.1 Impact du compilateur

Le but ici est de voir si l'utilisation de différents temps de chargement, pour l'ensemble des blocs, donne une bonne estimation, lors d'une compilation en *-O1* et si les observations réalisées avec *icpc* peuvent se confirmer avec le compilateur *g++*.

Pour cela, il a été comparé la somme des estimations du temps de chaque bloc, avec le temps réellement mesuré. Le tableau 4.1 montre les résultats acquis avec *icpc* basés sur un temps moyen obtenu avec 100 répétitions. Nous avons donc pour chaque bloc différentes mesures. L'unité de temps est la nanosecondes.

**la première colonne** identifie le numéro du bloc.

**La deuxième colonne** correspond au temps moyen mesuré, pour chaque bloc, lors de l'exécution du programme dans son intégralité. Durant son exécution une mesure est faite entre chaque bloc, le temps affiché correspond au temps moyen d'exécution du bloc, en nanoseconde.

**La troisième colonne** correspond au temps de chargement des données losqu'elles sont en mémoire.

**La quatrième colonne** correspond au temps de chargement des données lorsqu'elles sont dans le cache.

**La cinquième colonne** correspond au temps estimé en utilisant le modèle défini précédemment.

**La dernière colonne** nous donne l'erreur obtenue.

Enfin, la dernière ligne correspond au temps d'exécution du code lorsqu'il n'y a pas de mesure entre les différents blocs. Un appel a *PAPI* est donc effectué avant le début du premier bloc et un autre après le dernier bloc, la différence de temps entre les deux appels étant le temps écoulé.

La première remarque portera sur l'impact de l'utilisation de *PAPI* pour nos mesures. L'utilisation de mesures intermédiaires n'introduit que très peu de perturbation sur le

| | Temps de référence [ns] | Temps mémoire [ns] | Temps cache [ns] | Estimation [ns] | Erreur Relative |
|---|---|---|---|---|---|
| $t_{block1}$ | 42 | 27 | 8 | 35 | 17% |
| $t_{block2}$ | 2472 | 4 | 40 | 2398 | 3% |
| $t_{block3}$ | 72 | 9 | 24 | 33 | 54% |
| $t_{block4}$ | 1035 | 12 | 23 | 1038 | <1% |
| $t_{block5}$ | 115 | 11 | 54 | 65 | 43 |
| $t_{block6}$ | 1512 | 9 | 43 | 1510 | <1% |
| $t_{block7}$ | 79 | 23 | 17 | 40 | 49% |
| $t_{block8}$ | 560 | 22 | 14 | 522 | 7% |
| $t_{block9}$ | 103 | 22 | 42 | 64 | 35% |
| $t_{block10}$ | 1364 | 9 | 36 | 1377 | <1% |
| total | 7 354 | - | - | 7083 | - |
| total sans interm. PAPI | 7 218 | - | - | - | - |

TABLE 4.1 – Estimation du temps d'exécution par bloc avec icpc

temps d'exécution du programme, vu que celui-ci n'est majoré que de 2%.
Au niveau de la prédiction, il apparaît que les estimations des temps des boucles sont relativement bonne avec une erreur maximale de 7%. Par contre, pour les blocs hors de la boucle, l'erreur est beaucoup plus grande. Plusieurs raisons peuvent expliquer cela.
Tout d'abord, la lecture d'un bloc implique un changement d'état (chargement des registres, effacements de données...) qui peut être anticipé par le processeur via des techniques comme le prefetching. Or, lors du benchmarking de chaque bloc, pour obtenir le temps de chargement, l'état précédent n'est pas présent. De plus, lorsque l'on benchmark un bloc, on compile le bloc séparément. Or, en faisant cela, le compilateur ne tient pas compte de l'ensemble des données précédentes ou suivantes. Une expérience, introduisant la notion de "contexte", sera présentée en section 4.5.2, afin de voir si ce problème peut être résolu.
Néanmoins, en appliquant ce modèle, l'erreur globale de prédiction du temps d'exécution, sur cet exemple, est de 3,7% avec un temps prédit égal à 7083 nanosecondes.

Le tableau 4.2 présente les résultats des expérimentations obtenues avec le compilateur g++ en utilisant le même protocole que le compilateur *Intel*. Par rapport aux expérimentations menées précédemment, les résultats présentés avec g++ sont beaucoup moins instables. Alors qu'auparavant, l'estimation des temps des blocs impairs, c'est-à-dire des blocs hors boucle, était relativement éloignée par rapport à la réalité, ici, pour certains blocs, les résultats sont relativement précis. En effet, excepté le bloc 3, les blocs impairs ont une erreur inférieure à 19%. Concernant les boucles, les résultats sont toujours précis

| | Temps de référence [ns] | Temps mémoire [ns] | Temps cache [ns] | Estimation [ns] | Erreur Relative |
|---|---|---|---|---|---|
| $t_{block1}$ | 40 | 26 | 11 | 37 | 7% |
| $t_{block2}$ | 2496 | 23 | 45 | 2746.9 | 10% |
| $t_{block3}$ | 67 | 13 | 23 | 36 | 47% |
| $t_{block4}$ | 859 | 15 | 20 | 933 | 8% |
| $t_{block5}$ | 38 | 35 | 22 | 35 | 7% |
| $t_{block6}$ | 1498 | 23 | 44 | 1570 | 5% |
| $t_{block7}$ | 36 | 16 | 20 | 36 | <2% |
| $t_{block8}$ | 455 | 19 | 15 | 541 | 19% |
| $t_{block9}$ | 113 | 28 | 52 | 80 | 29% |
| $t_{block10}$ | 1490 | 16 | 39 | 1513 | <2% |
| total | 7092 | - | - | 7527 | - |
| total w/o interm PAPI | 6 921 | - | - | - | - |

TABLE 4.2 – Estimation du temps d'exécution par bloc avec g++

même si, pour la boucle 8, une erreur de 19% apparaît.
Globalement, l'estimation du temps de calcul de l'application présente une erreur de 6% avec un temps de 7527.

Comme nous avons pu le voir sur les tableaux 4.1 et 4.2, il y a une différence importante dans l'estimation des temps des blocs impaires (hors boucle). Pour mieux prédire ces blocs, il serait intéressant de voir si il serait possible d'introduire l'environnement dans lequel se trouve le code pendant son execution. C'est-à-dire d'introduire, la boucle précédent le bloc afin d'avoir un état proche de celui existant lors de son execution dans le programme.

### 4.5.2 Introduction du contexte

Pouvoir estimer un temps global d'exécution précisement, via la décomposition par bloc, est une bonne chose. Mais des communications pouvant avoir lieux entre ces blocs, il faut aussi avoir une bonne précision sur l'estimation de chaque bloc. L'idée présentée ici est d'introduire un contexte. Ce principe consiste à introduire la boucle précédent les blocs impairs. Par exemple, avant le calcul du temps du bloc 3, la boucle du bloc 2 a été introduite, afin de voir si il était possible de restaurer un état proche de celui existant durant l'exécution réelle. L'idée étant de savoir si les mesures observées n'étaient pas en relation avec un effet de cache, lié aux temps de chargement des instructions après une boucle. Les résultats pour chaque bloc sont donnés sur le tableau 4.3.

| Bloc | Temps de référence[ns] | Estimation sans contexte[ns] | Estimation avec contexte[ns] |
|---|---|---|---|
| Bloc 3 | 72 | 33 | 48 |
| Bloc 5 | 115 | 65 | 68 |
| Bloc 7 | 79 | 40 | 64 |
| Bloc 9 | 103 | 64 | 50 |

TABLE 4.3 – Impact du contexte sur l'estimation des temps des blocs avec icpc

Malheureusement, l'introduction du contexte n'amène pas un meilleur résultat. En effet, suivant les blocs, les résultats peuvent être meilleurs, comme pour le bloc 7, ou plus mauvais, comme pour le bloc 9. Ceci peut s'expliquer par le fait que les optimisations du compilateur sont faites sur la globalité du code et non sur des parties de blocs. De plus, le compilateur, en fonction des instructions présentes, peut réordonner et réécrire le programme pour optimiser son temps d'exécution. Ce résultat montre qu'il est parfois difficile d'avoir une très bonne estimation en ayant ce niveau d'abstraction. Il nous reste maintenant à regarder l'impact du niveau de compilation en étudiant les résultats, de notre code exemple, avec un niveau de compilation plus poussé.

### 4.5.3 Impact du niveau de compilation sur l'estimation

Comme il a été vu en section 4.3.3, l'optimisation des boucles est surtout effectuée au niveau des options *-O2* et *-O3*. Jusqu'à présent, le compilateur *Intel* offrait de bon résultat avec notre modèle, mais qu'en est il lorsque le code est beaucoup plus optimisé ? Pour cela, le rapport du bloc 2 sera étudié.
La figure 4.13 donne le résultat de l'expérimentation sur Core 2 Quad. Avec l'option *-O2*, le rapport est plus irrégulier. Le temps total de l'exécution de 10 itérations est presque équivalent à celui d'une seule itération ! Une des raisons de ce comportement peut s'expliquer par un déroulement de la boucle (*unroll*) par le compilateur lorsqu'il y a peu d'itérations.

Une autre observation que nous pouvons effectuer est que la décomposition en temps de chargement mémoire et en temps d'accès aux caches est difficilement utilisable, ici, du fait des irrégularités du temps d'exécution. Mais ces irrégularités ne sont pas toujours présentes, en effet passé 13 itérations, le temps moyen redevient plus "stable" et tend de nouveau vers un seuil. Au vu du comportement, il faut alors définir deux seuils pour pouvoir estimer le temps d'exécution d'une boucle. Encore une fois, de notre niveau d'abstraction, il est difficile de prendre en compte l'ensemble des optimisations effectuées par le compilateur pour ce type de structure.

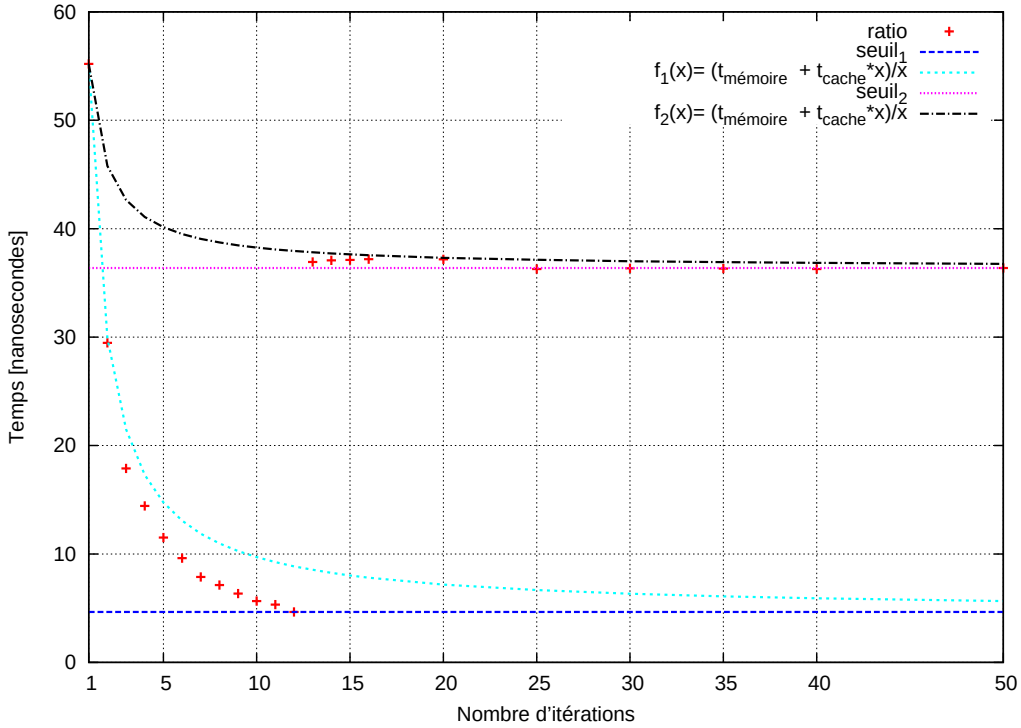



FIGURE 4.13 – Évolution du ratio sur Core 2 quad avec icpc et une optimisation -O2

### 4.5.4 Discussion

La décomposition par bloc donne une estimation du temps global relativement précise. Mais inviduellement, cette précision est moins bonne. Cela s'explique par le fait que les blocs sont testés séparément et non en séquence. L'introduction du contexte n'a pas été concluant, et n'a pas permis d'affiner la prédiction. De plus, des optimisations poussées peuvent amener des perturbations dans la logique des temps de chargements en cache.
Ceci est sûrement le contre-coup lié à la simplicité et la rapidité d'estimation de la méthode. En effet, il serait tout a fait possible d'avoir une meilleur approximation des résultats en travaillant directement avec le code assembleur optimisé, et en estimant le temps des parties de code correspondant à nos blocs. Mais descendre à un tel niveau engendrerait une plus grande complexité vu qu'il faudrait prendre en compte d'autres éléments comme la localité des données et des instructions, l'état des registres, de la mémoire, etc... Tout ceci nous éloignant de notre objectif premier qui est d'avoir une méthode rapide, simple, portable et relativement précise pour extrapoler le temps d'exécution de programmes parallèles.

## 4.6 Conclusion

Ce chapitre a présenté les différents éléments utilisés pour l'estimation des phases de temps de calcul des applications parallèles. A savoir : l'analyse statique, le découpage du code en blocs et le micro-benchmarking de ces blocs.
La partie 4.3 a permis d'introduire les différents temps de chargements (mémoire et cache), qui sont important pour l'estimation des temps d'un bloc via notre méthode.
Ensuite, le formalisme de la modélisation a été présenté en section 4.4, permettant de connaître les choix effectués pour les différentes structures présentes dans les langages de programmation procédurale.
En section 4.5, la précision de l'estimation des temps de calcul a été étudiée. Ces expérimentations ont permis de faire plusieurs observervations. Tout d'abord, l'estimation du temps d'exécution par blocs permet d'avoir une estimation du temps global d'exécution du programme relativement précise. Néanmoins, la précision entre les différents blocs n'est pas toujours au rendez-vous et dépend du compilateur. En effet, le compilateur *Intel* semble permettre une meilleure approximation que *gcc* pour un même niveau de compilation. De plus, lors de l'utilisation d'options de compilation poussées, les temps prédits s'avèrent souvent mauvais. Cette imprécision est dûe à plusieurs facteurs, mais principalement au découpage par bloc ne permet pas d'optimisations sur l'ensemble du code. Néanmoins, au vu du niveau d'abstraction utilisé, les résultats obtenues sont relativement précis. L'objectif principal étant d'avoir une méthode simple permettant d'estimer les temps de calcul rapidement tout en ayant ces composants réutilisables et paramétrable. Réunir un ensemble de blocs, par exemple les blocs entre deux communications, permettrait d'avoir une plus grande précision mais ferait perdre l'aspect réutilisable de la méthode, car un changement de paramètre pourrait modifier l'ensemble et demanderait une nouvelle étude pour le recomposer.
Les expériences menées se sont principalement focalisées sur l'utilisation d'un seul cœur. Néanmoins, une approche pour des modèles multi-cœurs est à l'étude. L'idée étant de voir la variation des temps d'exécutions des blocs du code parallèle lorsque les autres cœurs sont occupés, afin de voir si un modèle probabiliste pourrait être envisagé pour estimer ce temps.

# Modélisation des phases de communication 5

## 5.1 Introduction

L'arrivée des architectures multi-cœurs a accru le nombre de tâches communicantes entre les nœuds. Cette augmentation a entraîné un accroissement de l'utilisation et du partage de la ressource réseau. Les indices de performances utilisés couramment tels que la latence, la bande passante ou encore la bande passante de bissection (*bisection bandwidth* [44]) ne permettent pas de caractériser ce partage. Le temps supplémentaire nécessaire à une communication soumise à ce partage peut être un nouvel indice de performance significatif et facile à obtenir [67, 114]. Cet indice a été défini par Martinasso [67] comme étant la pénalité. Au travers de ce chapitre, et en s'appuyant sur les travaux précédents de Martinasso, il sera mis en place un protocole expérimental afin d'étudier et de comprendre l'impact de la concurrence sur les temps de communications pour le réseau *InfiniBand*. Cette étude mènera à la création d'un modèle de répartition de bande passante.
Cette thématique entrant dans la continuité des travaux de Martinasso, la première partie de ce chapitre se consacrera à résumer sa démarche afin de pouvoir positionner nos travaux. La deuxième partie du chapitre s'intéressera à décrire la gestion de la concurrence sur le réseau *InfiniBand*. Puis une série d'observations et d'expérimentations seront présentées dans la partie 5.4 afin d'étudier le comportement des communications point-à-point sur le réseau *InfiniBand* lorsqu'elles sont soumises à la concurrence. Cette série d'expérimentations permettra de définir un modèle de partage de bande passante pour le réseau *InfiniBand* décrit en section 5.5.

## 5.2 Travaux préliminaires

Ce chapitre portera sur la description des travaux de Martinasso sur les communications concurrentes dans les réseaux haute performance. La première partie décrira l'approche expérimentale permettant d'étudier l'évolution des pénalités. Cette étude a permis de définir deux modèles. Tout d'abord, un modèle prédictif des temps de communication,

commun pour chaque architecture réseau, permettant d'intégrer l'aspect dynamique des communications d'une application tout en tenant compte des pénalités. Puis un modèle de partage de bande passante, spécifique à chacun des réseaux, déterminant le coefficient de ralentissement des communications défini comme étant la pénalité.

### 5.2.1 Approche

L'approche utilisée, pour comprendre le partage de la bande passante, s'est effectuée via l'observation de l'évolution des temps des communications en étudiant l'impact de différents composants réseaux. Cette approche s'est organisée de façon incrémentale. Tout d'abord, par l'étude du comportement de deux communications soumises à la concurrence pour les réseaux *Gigabit Ethernet*, *Myrinet* et *Quadrics*, permettant de définir différents types de conflits simples. Puis par l'étude de schémas de communication plus complexes afin de déterminer la dépendance entre les pénalités suivant ces différents schémas.

La première partie de ces expérimentations s'est portée sur l'étude de deux communications en faisant varier différents paramètres sur différents réseaux, à savoir :
- L'implémentation *MPI* et la grappe de calcul
- Le placement des tâches sur les nœuds de la grappe
- Les paramètres des communications concurrentes : nombres, taille et date de départ

Pour cela, il s'est appuyé sur un programme de test, décrit dans la figure 5.1, permettant ainsi d'étudier le temps de chaque communication soumise à la concurrence, via les différents indices statistiques obtenus. Cette étude a permis de décrire quatre conflits élémentaires. Un conflit étant un ensemble de communications en concurrence. Ces conflits sont :

**Le conflit simple entrant (Conflit E/E).** Ce conflit fait intervenir deux communications entrantes simultanément dans un nœud, mais partant de deux nœuds différents. Cela correspond donc à un partage d'accès à la sortie du commutateur ou à l'entrée du nœud destination.

**Le conflit simple sortant (Conflit S/S).** Ce conflit est l'opposé du conflit précédent. En effet, les deux communications sortent simultanément d'un même nœud. Le partage de la ressource a donc lieu au niveau du nœud émetteur.

**Le conflit entrant/sortant (Conflit E/S).** Les deux communications de ce conflit se rencontrent au niveau du commutateur et d'un nœud. Les deux communications de ce conflit se rencontrent au niveau du commutateur et d'un nœud. Cependant, du fait que les liens sont bidirectionnelles, les messages ne se rencontrent pas. Il en est de même au niveau de la carte, car les cartes réseaux disposent souvent d'un espace mémoire différent pour la réception et l'émission des messages.

**Le conflit intra-nœud/extra-nœud (Conflit I/S ou I/E).** Dans ce cas, une des deux communications de ce conflit ne transfère pas de données sur le réseau. La concurrence

Début du code

```
; Parameters in:
;        nb_test : integer        // nombre de tests
;        nb_pro  : integer        // nombre de tâches
;        size : array of nb_pro*integer        // taille des communications
;        delay : array of nb_pro*integer        // date de départ des communications
;
; Parameters out:
;        // résultats statistiques
;        median, average : double
;        standard deviation, absolute deviation : double
;        minimum, maximum : double
;        quartile : array of 4*double

if (process_rank % 2 ==  0) {        // tâche paire correspondante à une émission
        repeat 5 times {        // éviter l'influence du cache
                MPI_Send (process_rank+1, size[process_rank])
        }
        MPI_Barrier
        repeat nb_test times {        // Mesures
                wait delay[process_rank]
                start_timer
                MPI_Send (process_rank+1,size[process_rank])
                stop_timer
                MPI_Barrier
        }
} else {         // tâche impair correspondante à une réception
        repeat 5 times {        // éviter l'influence du cache
                MPI_Recv (process_rank-1, size[process_rank])
        }
        MPI_Barrier
        repeat nb_test times {        // Mesures
                MPI_Recv (process_rank-1,size[process_rank])
                MPI_Barrier
        }
}
```

Fin du code

FIGURE 5.1 – Pseudo-code du programme de test

se situe uniquement au niveau de l'accès mémoire.

L'étude a permis de voir une différence de comportement dans la gestion de ces conflits entre les réseaux *Gigabit Ethernet*, *Myrinet* et *Quadrics*. Néanmoins, globalement, il a pu être montré que la pénalité n'intervenait qu'à partir d'un seuil correspondant au changement de protocole *MPI*, lorsque ce dernier passe en protocole de *rendez-vous*.

La deuxième partie s'est alors portée sur l'étude de schémas de communication plus complexes. La méthode de calcul des pénalités, utilisée auparavant, consistait au rapport entre le temps d'une communication soumise à la concurrence et le temps d'une communication non soumise à la concurrence. Or, pour des schémas de communications complexes, différents conflits avec différentes pénalités peuvent apparaître, proposant alors différents temps de communications. Il en résulte que des communications peuvent finir avant d'autres, modifiant alors le schéma de communication et donc les pénalités. Les

pénalités résultantes ne correspondant pas à un seul schéma de communication, mais à un ensemble de schémas avec un nombre de communication de plus en plus faible.
Afin que les pénalités représentent toujours le partage de la bande passante pour un schéma de communication donné, Martinasso avait modifié son programme de test. Au lieu de s'intéresser aux temps des communications pour l'envoi de messages de taille fixe, il s'est intéressé à la quantité de données pouvant être envoyée durant le temps d'une communication de 20 Mo sans conflit. Le but étant, pour un schéma de communication donné, d'obtenir des temps égaux pour chacune des communications. La pénalité étant alors calculée pour un seul type de schéma de communication. Les pénalités obtenues sont donc le rapport entre les tailles de messages envoyées pendant le conflit et celles envoyées sans conflit (20 Mo).
Ces expériences ont permis de montrer plusieurs points intéressants. Tout d'abord qu'il existait d'une part une certaine cohérence entre le nombre de communications en concurrence et la valeur des pénalités et, d'autre part, une certaine hiérarchie entre ces pénalités. Néanmoins, de part leurs protocoles de contrôle de flux, chaque réseau répartissait la bande passante différemment en fonction des conflits. Cela a permis de montrer le caractère dynamique de la pénalité. En effet, suivant le schéma de communication et les conflits engendrés, les pénalités induites par la concurrence variaient.

Suite à ces observations, Martinasso a défini deux modèles. Tout d'abord un modèle de prédiction des temps de communication, commun aux réseaux, puis un modèle de répartition de bande passante (pour le calcul de la pénalité) spécifique à chaque réseau.

### 5.2.2 Modèle prédictif des temps de communications

Martinasso a considéré une application parallèle comme un ensemble d'événements de calcul et de communication. De plus, ces observations ont permis de montrer que la pénalité évoluait lors du début ou de la fin d'une ou plusieurs communications. En se basant sur cela, il a utilisé le principe de la simulation à événements discrets afin de déterminer le temps des communications.

Pour déterminer la quantité de données envoyée, un modèle proche de celui de Clement et Steed [18, 101] a été utilisé. Ce modèle avait été conçu pour les programmes utilisant *PVM*. Le principe consistait à prendre en compte la contention dans les réseaux partagés (comme un réseau ethernet non-commuté) en augmentant le temps de communication du modèle linéaire d'Hockney [81] par un facteur de contention $\gamma$ :

$$T = l + \frac{b \times \gamma}{W}$$

où $l$ correspond à la latence, $b$ la taille du message, $W$ la bande passante. Comme nous l'avons vu précédemment, la pénalité évolue en fonction du schéma de communication. Or, sur ce modèle, $\gamma$ était supposé constant et égal au nombre de processus de communication.

Pour adapter cette équation à ces besoins, il a suffit de remplacer la valeur du facteur de contention $\gamma$ par la pénalité $p$ engendrée par le modèle de répartition de bande passante, pour un schéma de communication donné, à un instant $t$. Le temps d'une communication $c$ est donc :

$$T_c = l + \frac{b \times p}{W}$$

C'est cette équation que Martinasso a utilisé pour estimer la quantité de données envoyées entre deux événements pour une certaine pénalité. La figure 5.2 illustre l'algorithme mis en œuvre.

### 5.2.3 Modèles de répartition de bande passante

Pour estimer la valeur des pénalités associées à chaque communication pour un graphe de communication donné, Martinasso a choisi deux axes d'approche différents. Pour le réseau *Myrinet*, il a choisi l'approche descriptive basée sur une description du comportement du contrôle de flux, alors que pour le réseau *Gigabit Ethernet*, l'approche choisie était quantitative, privilégiant des équations basées sur des paramètres mesurés sur le réseau. Ces deux approches ont permis de calculer les valeurs des pénalités induites par les conflits, pour chacun des réseaux.

### 5.2.4 Validation

Pour évaluer l'efficacité de ces modèles et les valider, Martinasso a construit un simulateur se basant sur la simulation à événements discrets. Ce simulateur sera décrit dans le chapitre 6. En utilisant son simulateur, il a évalué ces modèles sur un ensemble de graphes de communications synthétiques (arbres et graphes complets) ainsi que sur les graphes de communication de l'application Linpack (HPL). L'ensemble de ces expériences a permis de montrer la précision des prédictions obtenues par ces modèles.

### 5.2.5 Discussion

Le choix de l'approche expérimentale proposé par Martinasso a permis de déterminer un protocole permettant d'étudier les comportements des réseaux haute performance lorsqu'ils sont soumis à la concurrence. Néanmoins, ces observations se sont appuyées sur l'étude des temps d'envois de messages à un niveau local (carte réseau), sans se soucier des problèmes pouvant intervenir au niveau du commutateur. Ce choix est parfaitement compréhensible car les réseaux *Gigabit Ethernet* et *Myrinet* n'utilisent pas un routage statique comme le réseau *InfiniBand*. Dans le cadre de l'étude du réseau *InfiniBand*, le protocole expérimental a été modifié afin de tenir compte du routage, permettant ainsi d'étudier l'évolution des pénalités sans les perturbations éventuelles causées par un problème de routage au niveau du commutateur.

```
Début de l'algorithme
; Parameters in:
;        Cluster : Formalism   // Formalisme de la grappe
;        Application  : Formalism   // Formalisme de l'application
;        Mapping : Formalism   // Formalisme du placement des tâches sur les nœuds
;        Model : Parameter   // Paramètres du modèle (bande passante,etc)
;
; Parameters out:
;        communications : array of Communication // Temps des communications


G : Graph // graphe de communication
c : Communication
p : array of double // tableau des pénalités, p[c] est la pénalité de la communicaction c
time_interval_end : time
time_interval_start : time
time_interval : time
time_step : time
bw : double
l : double


bw = GetModelBandwidth(Model)
l = GetModelLatency(Model)
time_step = 0
// construction du graphe au temps 0
G := buildGraph(Cluster,Application,Mapping,time_step)

while( G has communication) {
        // déterminer les pénalités à partir du modèle
        p := model_of_bandwidth_sharing(G,Model)

        // déterminer le premier événement à se produire: début ou fin d'une communication

        // intervalle de temps avant le début d'une nouvelle communication
        time_interval_start = TimeOfNextCommunicationToStart(G,time_step) - time_step

        // intervalle de temps jusqu'à la terminaison d'une communication
        c = FindFirstCommunicationToEnd(G,p,time_step)
        time_interval_end = GetCommunicationSize(c)*p[c]/bw + l

        // intervalle de temps avant le premier événement
        time_interval = min(time_interval_start,time_interval_end)


        // déterminer les quantités de données envoyées pour chaque communication
        foreach c in G {
                SetCommunicationSize(GetCommunicationSize(c) - (bw*(time_interval - l))/p[c])
                SetCommunicationTime(GetCommunicationTime(c) + time_interval)
        }

        // construction du graphe au temps time_step + time_interval
        time_step = time_step + time_interval
        G := buildGraph(Cluster,Application,Mapping,time_step)

}
Fin de l'algorithme
```

FIGURE 5.2 – Simulation à événements discrets

## 5.3 La gestion de la concurrence dans le réseau *InfiniBand* pour les communications inter-nœud

Une communication inter-nœud correspond à l'envoi d'un message du nœud source vers un nœud destination au travers du réseau les reliant. Comme énoncé précédemment, le problème du routage statique peut amener à des phénomènes de partage non désirés [49]. Néanmoins, lorsque plusieurs communications inter-nœud utilisent conjointement une ressource, un mécanisme de partage a lieu, créant alors des communications concurrentes. Ce partage peut avoir lieu soit au niveau du commutateur soit au niveau de la carte. Il convient donc de s'intéresser aux mécanismes de gestion de la concurrence sur *InfiniBand*.
Au niveau du commutateur, la concurrence peut éventuellement créer un Head of Line (HoL) [39], provenant d'une saturation de la mémoire du commutateur et amenant un ralentissement de l'ensemble des communications. Néanmoins, ce type de phénomène reste rare, car il ne se produit que dans des situations bien spécifiques. Pour éviter cela, les cartes ConnectX disposent d'un mécanisme de contrôle de flux supplémentaire appelé *Explicit Congestion Notification*(ECN) [39]. Un autre moyen d'éviter les *HoL* est d'utiliser les liens virtuels (ou *Virtual Lines* (VL)). Le principe consiste à créer des canaux logiques au travers du canal physique. Actuellement, les implantations *MPI* n'utilisent qu'un seul VL. Pour le réseau *InfiniBand*, le rôle du commutateur reste donc globalement limité à transférer les messages d'un point d'entrée vers un point de sortie en suivant le routage défini par le *subnet manager*. Si plusieurs communications se chevauchent sur un même lien, au niveau du commutateur, ce seront les cartes émettrices qui réduiront la vitesse d'arrivée des paquets.
L'élément qui gère donc la concurrence pour le réseau *InfiniBand* est la carte réseau via ces mécanismes de contrôle de flux(*credit based flow control*, *static rate control*, *Explicit Congestion Notification*).
Au travers des différentes expériences présentées ici, le rôle et l'impact du *credit based flow control* et du *static rate control* pourront être mis en valeur.
Le *credit based flow control* est un mécanisme de contrôle basé sur le principe du crédit. Chaque carte réceptrice dispose de crédits correspondant à la quantité de données pouvant être reçue sans perte, la connaissance du nombre de crédit disponible se faisant via l'échange de paquets spécifiques. Lorsqu'une carte réceptrice n'a plus de crédit, donc d'espace mémoire disponible pour la réception, la carte émettrice arrête l'envoi des données.
Le *static rate control* est un autre mécanisme de contrôle de flux, qui se charge d'équilibrer les débits lorsqu'un lien est saturé.

## 5.4 Observations préliminaires

L'objectif de ces observations est d'étudier expérimentalement la concurrence entre les communications point-à-point sur le réseau *InfiniBand*. En plus du nombre de communications en concurrence, les effets de cette concurrence peuvent varier suivant le contexte expérimental, à savoir :

- Les caractéristiques des communications (taille des messages, date de départ).
- Le modèle de la carte *InfiniBand*.
- L'implémentation *MPI*.

Afin de voir l'impact de chacun de ces contextes expérimentaux, ceux-ci seront étudiés séparément en augmentant progressivement la complexité (nombre de communications). De plus, une comparaison avec d'autres réseaux haute performance sera effectuée, avant de voir l'impact du routage statique sur le réseau *InfiniBand*.
Dans un premier temps, le protocole expérimental sera établi pour définir les objectifs ainsi que le matériel et les programmes utilisés. Puis, une série d'expériences introductives sera commentée pour étudier le comportement des différentes formes d'expressions de la concurrence (les conflits) identifiées par Matinasso. Ensuite, des expérimentations plus élaborées seront abordées afin de déterminer l'impact du contexte expérimental sur la concurrence. Enfin, une synthèse des observations présentera les différents résultats.

### 5.4.1 Protocole expérimental

Le protocole expérimental permet de décrire les objectifs ainsi que le dispositif expérimental.

#### Objectif

Pour étudier l'impact de la concurrence, Martinasso avait étudié la variabilité des temps de communication induite par la concurrence.
Sachant que les principales caractéristiques d'un réseau haute performance sont la latence et la bande passante, l'impact de la concurrence sur ces deux éléments sera étudiée. Puis, il sera introduit la notion de pénalité, correspondant au rapport entre le temps d'une communication soumise à la concurrence et une communication non soumise à la concurrence. Les expérimentations seront décomposées en deux parties.
Tout d'abord, des expériences introductives seront faites, afin de bien comprendre l'impact de la concurrence sur les performances des communications. La première expérience portera sur l'évolution de la latence en fonction du nombre de communication en concurrence et suivant la génération de la carte *InfiniBand*. Puis, il sera étudié le comportement des communications sur un large panel de taille de message, lorsqu'elles sont soumises à la concurrence. Les différents conflits simples, décrit par Martinasso, seront étudiés durant cette phase. Une étude statistique sera aussi menée afin de connaître l'impact de la

concurrence sur la stabilité des communications.
Dans un second temps, des expériences présentant des schémas de communication plus complexes seront étudiées. L'indice de performance utilisé, pour ces expériences, sera la pénalité. Différentes configurations seront étudiées afin d'avoir une vision relativement complète des éléments impactant sur cette pénalité.

### Grappes de calcul

Les grappes de calcul utilisées seront à base de nœud *Novascale*, composées de processeurs multi-cœurs Woodcrest et de différentes générations de carte *InfiniBand*. Les générations de cartes sont nommées, chez Mellanox, en fonction de la puce présente au sein de la carte. Il y a eu successivement la Tavor (Infinihost), la Sinaï (ou Infinihost III LX), l'Arbel (ou Infinihost III EX), la ConnectX et la dernière génération la ConnectX-2. Une autre carte *InfiniBand* de fabricant différent sera utilisée à savoir la carte *QLogic*.

### Implantation *MPI*

Deux implémentations *MPI* seront utilisées. Il y aura l'implémentation *MPI BULL* 2 ainsi que l'implémentation *ScaMPI* 5.5 . *ScaMPI* étant une version de *MPI* développée par la société *Scali*. (La branche *MPI* de *Scali* a d'ailleurs été rachetée en 2008 par *Platform Computing*, qui a renommé ScaMPI en *Platform MPI* 5.6.)
Sauf si cela est signalé, la version de *MPI* utilisée sera *MPI BULL* 2.

### Programmes utilisés

Différents logiciels seront utilisés pour l'étude de la concurrence. Pour la latence, le logiciel de benchmark de communication *IMB* (voir section 2.4.1) sera utilisé avec le mode *MULTI*. Les tests seront effectués via l'envoi d'un message de 1 octet soit en unidirectionnel (un nœud envoi et un nœud reçoit) soit en bidirectionnel (chacun des nœuds reçoit et envoie le message).

*IMB* ne pouvant pas prendre en compte différents schémas de communication, le programme de test de Martinasso sera utilisé. Néanmoins, ce programme a été légèrement modifié afin qu'il puisse prendre en compte la table de routage et la topologie. Ces deux informations pouvant être récupérées via l'utilisation respective des commandes *ibdiagnet* et *ibnetdiscover*. Le programme est détaillé dans la figure 5.3.
Le but du programme est de pouvoir étudier la variation des temps de chaque communication lorsqu'elles sont soumises à la concurrence. Pour cela, le programme envoie des données en utilisant les communications bloquantes *MPI_Send* et *MPI_Recv* suivant un schéma de communication donné. Différents paramètres peuvent varier tel que le nombre de répétitions du test, le nombre de communications, la taille des messages, la date de

Début du code

```
; Parameters in:
;        nb_test : integer // nombre de tests
;        nb_pro  : integer // nombre de tâches
;        size : array of nb_pro*integer // taille des communications
;        delay : array of nb_pro*integer // date de départ des communications
;        topo : File // topologie réseau
;        rout : File // table de routage
; Parameters out:
;        // résultats statistiques
;        median, average : double
;        standard deviation, absolute deviation : double
;        minimum, maximum : double
;        quartile : array of 4*double

Check_Topo(topo, rout) // vérifie que le chemin réseau n'est pas partagé au niveau du commutateur

if (process_rank % 2 ==  0) // tâche paire correspondante à une émission
        repeat 5 times // éviter l'influence du cache
                MPI_Send (process_rank+1, size[process_rank])
        }
        MPI_Barrier
        repeat nb_test times // Mesures
                wait delay[process_rank]
                start_timer
                MPI_Send (process_rank+1,size[process_rank])
                stop_timer
                MPI_Barrier
        }
} else { // tâche impair correspondante à une réception
        repeat 5 times { // éviter l'influence du cache
                MPI_Recv (process_rank-1, size[process_rank])
        }
        MPI_Barrier
        repeat nb_test times {        // Mesures
                MPI_Recv (process_rank-1,size[process_rank])
                MPI_Barrier
        }
}
```

Fin du code

FIGURE 5.3 – Pseudo-code du programme de test modifié

départ. A l'issue de l'exécution, différents indices statistiques sont fournis.
Il est à noter que pour les messages de petites tailles, le temps mesuré ne correspond pas à la latence, mais au temps de copie du message dans le tampon mémoire de la carte. De plus, l'étude se portera principalement sur des communications passant uniquement par un seul commutateur.

### Placement des tâches *MPI*

Le placement des tâches ne pouvant se faire sur l'implémentation *MPI BULL* 2. Celui-ci se fera suivant la configuration par défaut des cartes mères *Intel*, à savoir, un placement Round Robin par socket. Les tâches sont assignées en remplissant d'abord le premier so-

cket du nœud. La tâche sera fixée sur un cœur, suivant un placement défini en paramètre du programme de test.

**Résumé des paramètres expérimentaux variables**

Les paramètres variables pour ces expériences seront donc :
- La génération de la carte *InfiniBand*.
- Le placement des tâches
- Les paramètres de communications : nombre, taille, date de départ des messages
- l'implémentation *MPI*

### 5.4.2 Expériences introductives

Les premières expériences porteront sur l'étude de l'impact de la concurrence au niveau de la latence et du débit. Dans les deux cas, les communications transmettront la même quantité de donnée au même instant.

#### 5.4.2.1 Latence

L'étude de la latence a été effectuée sur plusieurs générations de cartes *InfiniBand*. Cette expérience permet de voir l'amélioration progressive des cartes *InfiniBand* concernant la gestion de la concurrence pour les petits messages.
La figure 5.4 montre l'évolution de la latence en fonction de la charge du réseau et de la version de la carte. Les communications se faisant uniquement entre deux nœuds.
Pour la génération Sinai, les cartes supportaient très mal la réception et l'envoi simultané de données de petites tailles. La latence d'une carte Sinai 1.0 passant de 3.15 microsecondes, pour une seule communication unidirectionnelle, à 134.8 microsecondes lorsqu'il y a huit communications bidirectionnelles.
Les générations successives ont vu une amélioration du support de la charge grâce à l'évolution de la puce contenue dans la carte et de la norme *InfiniBand*. A l'heure actuelle, l'une des dernières générations de carte, la ConnectX, supporte très bien la charge indépendamment de la version du bus *PCI* (génération 1 ou 2). Au vu de l'augmentation du nombre de cœurs au sein du nœud, ces progrès étaient nécessaires afin de rendre le réseau *InfiniBand* toujours aussi performant. Néanmoins, la carte de Qlogic supporte moins bien la charge réseau par rapport aux versions ConnectX de Mellanox disponible depuis la même période.

#### 5.4.2.2 Débit

Pour étudier l'impact de la concurrence sur la bande passante, le programme de test défini auparavant sera utilisé sur des nœuds disposant de cartes de génération *Arbel* (*Infini-*

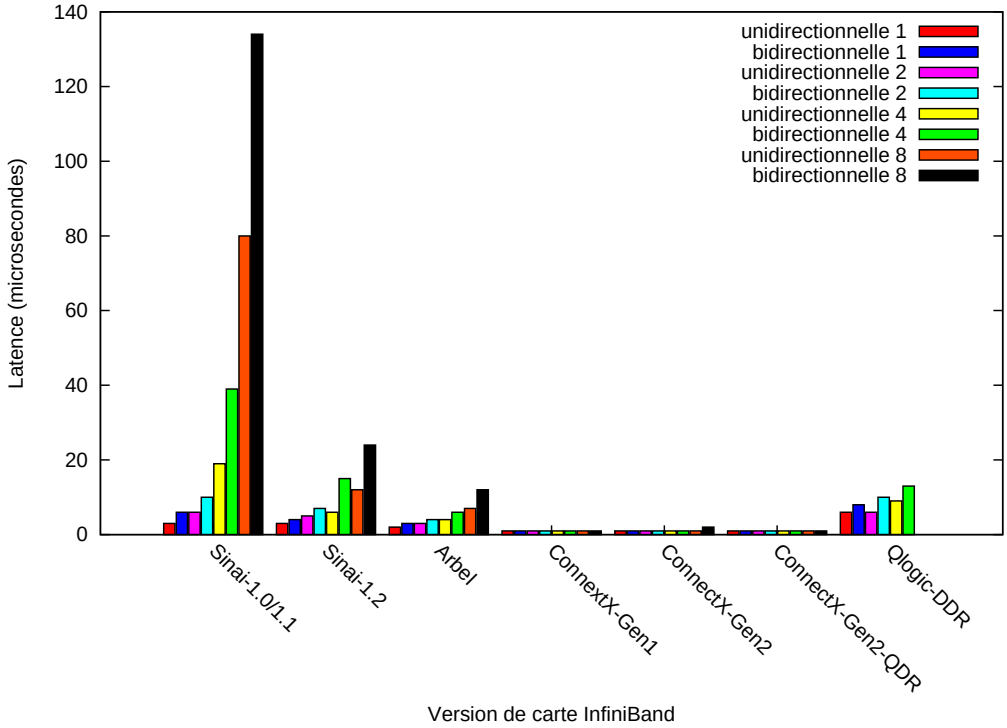



FIGURE 5.4 – Impact de la charge du réseau sur la latence suivant les versions des cartes *InfiniBand*

*host III*). Ces expériences seront effectuées avec deux communications mises en concurrence et comparées avec le temps d'une communication non soumise à la concurrence. Chacune des communications envoyant la même quantité de donnée au même moment.
Lorsque des communications sont soumises à la concurrence, des conflits apparaissent. Quatre conflits simples avaient été identifiés par Martinasso. L'impact de chacun de ces conflits sera étudié d'abord sous la forme d'un graphe à l'échelle logarithmique puis par une analyse statistique.

**Conflit simple entrant (conflit E/E) :** La figure 5.5 montre le résultat de l'expérimentation. Le conflit E/E est décomposé en deux phases distinctives séparées par un saut à 8Ko.
Tout d'abord, pour les messages de petites tailles, il y a peu de variations entre les temps des communications soumises à la concurrence et le temps de la communication seule. Cela est dû au protocole utilisé. En effet, pour ces messages, l'*eager protocol* est utilisé et ce dernier dépend principalement de la latence, qui est faiblement impacté par la concurrence, comme nous l'avons vu précédemment. Passé les 8Ko, c'est le *rendezvous protocol* qui intervient, créant alors un surcoût lié à la préparation de la réception du message. Ici, le temps des communications concurrentes est presque doublé ($\times 1.7$) par rapport au temps

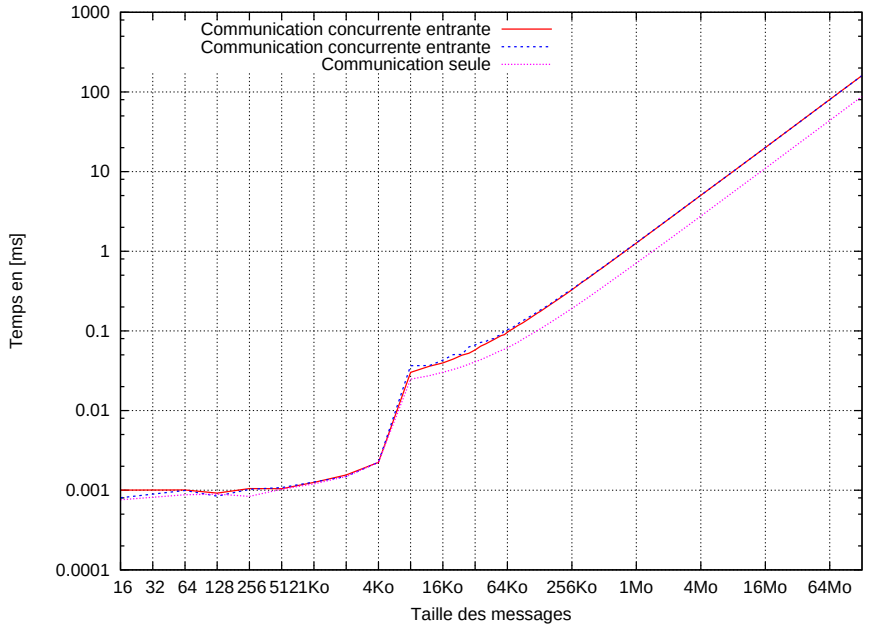



FIGURE 5.5 – Conflit Entrant/Entrant

de la communication seule.

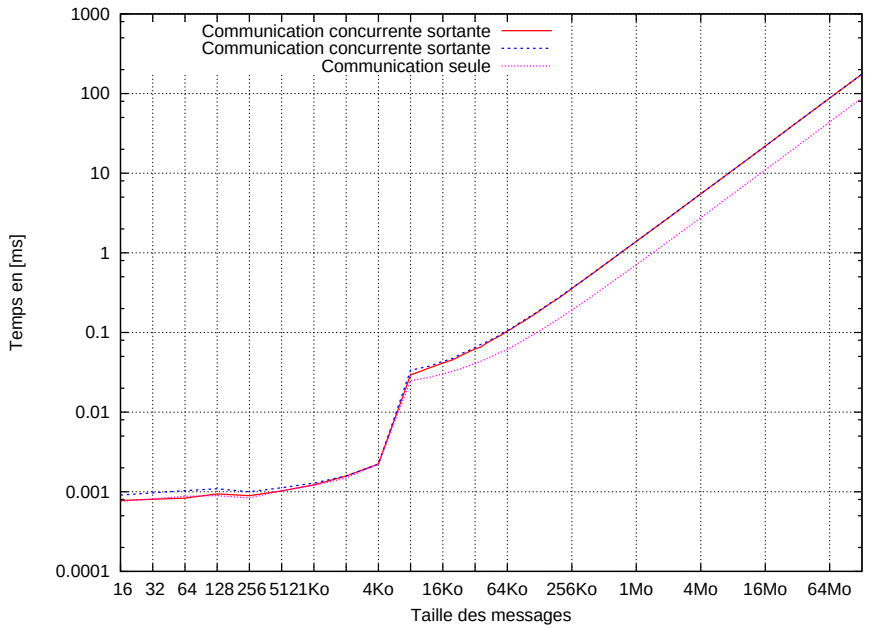



FIGURE 5.6 – Conflit Sortant/Sortant

**Conflit simple sortant (conflit S/S) :** Le comportement observé est exactement le même que le conflit E/E, comme le montre la figure 5.6. Néanmoins, dans ce type de conflit, la pénalité est de 1.6.

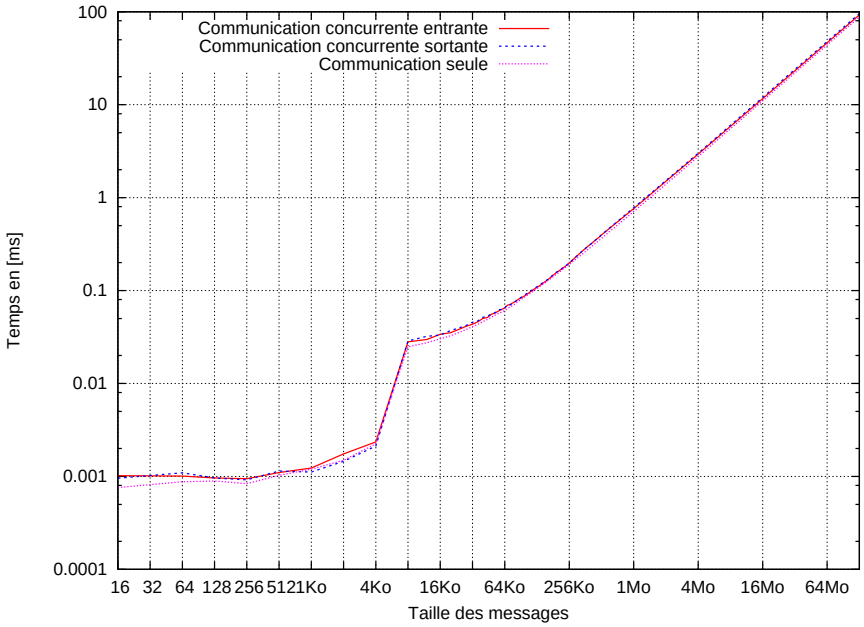



FIGURE 5.7 – Conflit Entrant/Sortant

**Conflit entrant/sortant (Conflit E/S) :** Les deux communications de ce conflit se rencontrent au niveau du commutateur et d'un nœud. Cependant, du fait que les liens *InfiniBand* sont bidirectionnels, les messages ne se rencontrent pas. Il en est de même au niveau de la carte, car les cartes *InfiniBand* disposent d'un espace mémoire différent pour la réception et l'émission des messages. La figure 5.7 confirme que le conflit E/S est peu coûteux quelle que soit la taille du message.

**Conflit intra-nœud/extra-nœud (Conflit I/S ou I/E) :** Des expériences ont montré que l'impact de ces types de conflit était très faible et limité à une taille de message relativement grande (supérieur à 7Mo).

**Indices statistiques en fonction des conflits simples :**

Le tableau 5.1 montre les résultats obtenus par notre benchmark et permet d'étudier la stabilité des différents conflits simples. Pour les messages de 16 octets, le temps calculé est en microseconde alors que pour les messages de 16 Mo, le temps est en seconde. Les expériences, pour les messages de 16 octets, montrent une certaine instabilité liée

| | | 16o [$\mu$s] | | | | | 16Mo[s] | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Moy. | Med | ET | Min | Max | Moy | Med | ET | Min | Max |
| Sans conflit | | 0.68 | 0.95 | 1 | 0 | 2 | 0.011 | 0.011 | 0 | 0.011 | 0.011 |
| Conflit E/E | Com1 | 0.67 | 0.95 | 0.4 | 0 | 3 | 0.018 | 0.018 | 0 | 0.018 | 0.018 |
| | Com2 | 0.67 | 0.95 | 0.4 | 0 | 1.9 | 0.018 | 0.018 | 0 | 0.018 | 0.018 |
| Conflit S/S | Com1 | 0.67 | 0.95 | 0.5 | 0 | 2.14 | 0.017 | 0.017 | 0 | 0.017 | 0.017 |
| | Com2 | 0.65 | 0.95 | 0.5 | 0 | 2.14 | 0.017 | 0.017 | 0 | 0.017 | 0.017 |
| Conflit E/S | Com1 | 0.67 | 0.95 | 0.5 | 0 | 3 | 0.011 | 0.011 | 0 | 0.011 | 0.011 |
| | Com2 | 0.65 | 0.95 | 0.5 | 0 | 1.9 | 0.011 | 0.011 | 0 | 0.011 | 0.011 |

TABLE 5.1 – Indice statistique des temps de communications des conflits simples pour le réseau Infiniband et MPIBull 2.

probablement à un problème de mesure dû à l'échelle de temps utilisée relativement petite. Pour les messages de 16 Mo, les expériences montrent que les temps des communications sont particulièrement stables et cela quel que soit le conflit. Comme il sera vu plus tard, le choix de l'implémentation *MPI* n'impacte pas sur le comportement des communications.

**Bilan des expérimentations**

Les améliorations des cartes *InfiniBand* ont permis de grandement réduire les perturbations de la latence liées à une charge trop importante. Actuellement, l'impact de cette charge ne se ressent plus sur les dernières générations de cartes *InfiniBand*.
Néanmoins, des conflits simples tels que le conflit E/E et le conflit S/S augmentent la durée des communications, pour les messages supérieurs à 8 ko. Ce ralentissement, défini comme étant la pénalité, est causé par un changement de protocole *MPI* (*rendezvous*). La pénalité pour ces conflits correspond à une multiplication par 1.7 du temps d'une communication sans concurrence pour le conflit E/E et par 1.6 du temps d'une communication sans concurrence pour le conflit S/S.
Pour le conflit E/S, il n'y a presque aucune pénalité, car les liens réseaux sont full-duplex. Enfin, un conflit I/E ou I/S, de par la nature du partage, impacte très peu sur la durée de la communication passant par la carte réseau.
De plus, une communication soumise ou non à la concurrence est stable.

### 5.4.3 Schémas de communications complexes

**Objectif**

L'élaboration de schémas de communication complexes se fera soit par l'augmentation du nombre de communications, soit par l'ajout de nouveaux nœuds. Cela permettra de voir l'évolution des pénalités en fonction du nombre de communications. L'objectif principal étant d'observer et de déterminer la relation entre l'évolution de la pénalité suivant

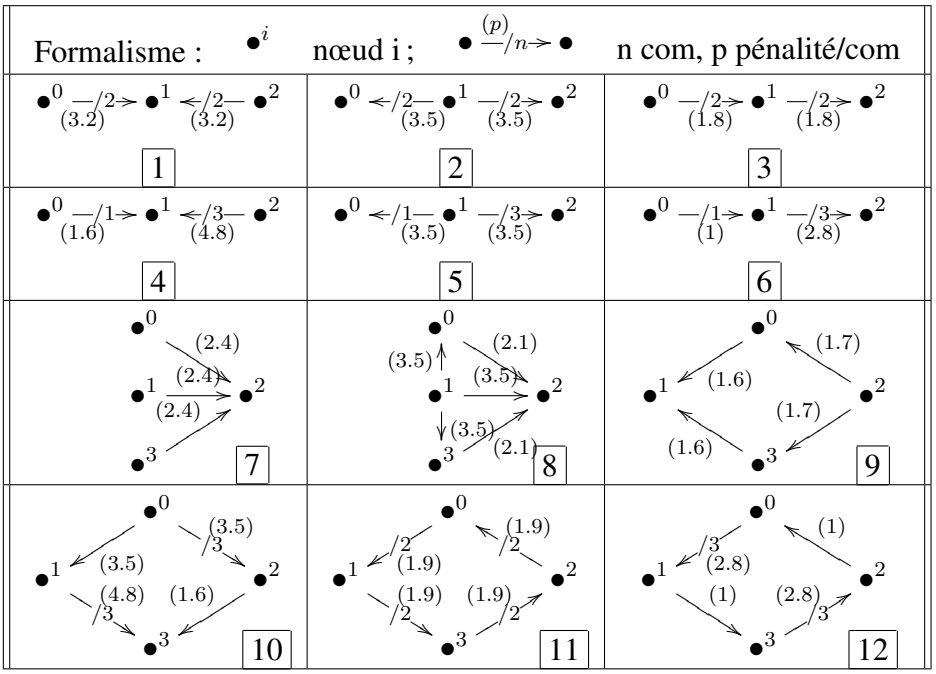



FIGURE 5.8 – Conflits complexes sur le réseau *InfiniBand* avec *MPI BULL* 2.

les schémas de communications. Les expériences suivantes montrent donc le partage de la bande passante pour un schéma de communication donné et vont permettre d'observer les dépendances entre les conflits.

### Résultats

La figure 5.8 présente les résultats obtenus suivant différents schémas, les points représentant des nœuds, la valeur entre parenthèse représentant la valeur de la pénalité pour chacune des communications, la flèche représentant la communication et la valeur sur la flèche le nombre de communications.
Ainsi le schéma 1 peut être interprété comme représentant trois nœuds (0, 1 et 3), le nœud 0 envoyant deux communications vers le nœud 1 et le nœud 2 envoyant deux communications vers le nœud 1. Les pénalités résultantes de ce schéma étant de 3.2 pour chacune des communications.
Les schémas 1 et 7, représentant des communications se dirigeant vers le même nœud, peuvent montrer que l'évolution est assez prévisible. En effet, lorsque quatre communications se dirigent vers un nœud, la pénalité vaut 3.2, alors que pour trois communications cette pénalité vaut 2.4. Néanmoins, le schéma 4 montre un tout autre rapport. La pénalité des communications allant du nœud 2 vers le nœud 1 étant le triple de la pénalité de la communication allant de 0 vers 1.

Pour les schémas 2 et 5, représentant quatre communications sortantes d'un nœud, la pénalité est identique. Les schéma 3 et 6 montrent qu'il semble exister une certaine hiérarchie dans les pénalités, mais pour s'en rendre bien compte, l'ajout d'un nœud et de communications supplémentaires est nécessaire.
Le schéma 8 indique bien une certaine hiérarchie et une propagation dans les pénalités. En effet, toutes les communications sortantes du nœud 1 ont la même pénalité alors que les nœuds 0 et 3 ne reçoivent qu'un seul message. Il semblerait donc que la communication allant du nœud 1 vers le nœud 2 ralentisse l'ensemble des communications quittant le nœud 1. Au niveau du nœud 2, les communications reçues n'ont pas les mêmes pénalités. En comparant avec le schéma 7, les pénalités des communications sortantes du nœud 0 et 3 sont assez proches alors que la communication venant du nœud 1 a une pénalité beaucoup plus importante.
Le schéma 9 permet d'avoir des informations supplémentaires sur les pénalités. Au niveau du nœud 0 et du nœud 3, nous avons un conflit E/S qui ne doit pas créer de pénalités. Or, ce n'est pas le cas. Au niveau du nœud 1, il y a un conflit E/E avec une pénalité de 1.6 qui correspond à ce que nous avions vu précédemment. Il en est de même pour le nœud 2, le conflit S/S existant ici à la même pénalité qu'auparavant. Les conflits E/E et S/S semblent donc prioritaires par rapport aux conflits E/S. Une même observation est possible sur le schéma 10.
Le schéma 11 représente un conflit E/S avec deux communications entrantes et sortantes pour chaque nœud. Les pénalités obtenues pour chacune des communications sont proches du schéma 3. Le schéma 12 obtient lui aussi des résultats identiques au schéma 6.
Les schémas complexes représentés ici ont permis de montrer une hiérarchie dans les pénalités. De plus, il semblerait que lorsqu'une communication sortante d'un nœud est ralentie, l'ensemble des communications sortantes le sont aussi. Les raisons de ce phénomène peuvent s'expliquer par la connaissance du contrôle de flux du réseau *InfiniBand* et notamment du *credit based flow control* comme nous le verrons.

### 5.4.4 Modification des paramètres

Durant l'exécution d'un programme parallèle, différents schémas de communications apparaissent. Les expériences précédentes ont permis de faire ressortir les pénalités des schémas de communication spécifiques. Or dans la réalité, toutes les communications n'ont pas la même taille et ne finissent pas au même moment. Il faut donc étudier l'évolution des pénalités lorsque le schéma de communications est modifié (par la terminaison d'une communication par exemple) et déterminer l'impact d'un changement de génération de carte ou d'implémentation *MPI* sur ces pénalités. Cette partie permettra aussi de montrer les mécanismes matériels et logiciels qui peuvent expliquer l'évolution des pénalités.

| | Sans Conflit | Conflit S/S | Conflit S/S + départ différent |
|---|---|---|---|
| Temps (16Mo) [ms] | 11 | com1 : 17.6 ; com2 : 17.6 | com1 : 14.8 ; com retardée : 14.7 |

TABLE 5.2 – Impact du temps de départ et de la taille du message sur la concurrence

#### 5.4.4.1 Temps de départ et tailles de messages différents

Pour illustrer l'impact d'un temps de départ différent, une expérience sera réalisée avec deux communications créant un conflit S/S. Comme le montre le tableau 5.2 :

- le temps d'une communication sans conflit d'une taille de 16 Mo est de 11 ms
- le temps de deux communications d'une taille de 16 Mo, démarrant au même instant, est de 17.6 ms.

L'objectif ici est de démarrer le conflit lorsque la première communication a déjà envoyé la moitié de ces données, c'est-à-dire 8 Mo. Pour cela, le départ de la seconde communication est retardé de 5.5 ms. De cette façon, le conflit apparaîtra à la moitié de la diffusion de la communication non retardée, jusqu'à ce qu'une des deux communications soient terminées. Les communications étant chacune soumises à la concurrence pendant l'envoi des 8 derniers Mo de la première communication et des 8 premiers Mo de la seconde.
Comme il a été montré en 5.4.2.2, la pénalité d'un conflit S/S est de 1.6. Il est possible de retrouver algébriquement un résultat proche de celui obtenu expérimentalement. En effet, ce schéma peut se traduire par $t_{16Mo} = t_{8Mo} + 1.6 \times t_{8Mo}$. Sachant que le temps d'envoi de 8Mo correspond à 5.5 ms, le calcul nous donne $t_{16Mo} = 5.5 + 1.6 \times 5.5 = 14.3ms$. Cette expérience ainsi que d'autres ont permis de conclure que les pénalités évoluaient uniquement au moment de la création ou de la terminaison des communications.
De part ces observations, il est facile de se rendre compte que le changement de taille n'apporte rien de nouveau. En effet, comme l'avait montré Martinasso pour le réseau *Gigabit Ethernet* et *Myrinet*, la modification de la taille d'un message entraîne uniquement une modification de son temps d'émission. Par exemple, un conflit S/S avec deux messages de tailles différentes ne changera pas la valeur de la pénalité lorsque ces communications seront en concurrence, mais uniquement le temps de cette concurrence.

#### 5.4.4.2 Comparaison des pénalités entre génération de cartes *InfiniBand* différentes

La figure 5.9 illustre les résultats obtenus lorsque l'on compare une carte Arbel avec une carte ConnectX. Malgré le fait que les cartes ConnectX soient plus performantes en terme de latence et de bande passante, les pénalités obtenues sont presque identiques. La raison de ce résultat est liée aux faits que l'implémentation *MPI BULL* 2 utilisée (qui s'appuie sur MVAPICH2 1.0) fait uniquement appel à la bibliothèque de bas niveau *libibverbs* pour l'envoi des messages. Malgré le fait que les communications sur ConnectX soient plus rapide que les communications sur carte Arbel, cela ne change rien au niveau des pénalités. Elles démarreront et finiront au même moment. L'utilisation d'une biblio-

| Schema de Communication | Interconnect | |
|---|---|---|
| | Infiniband InfinihostIII | Infiniband ConnectX |
| $\bullet^0 \xrightarrow{(a)} \bullet^1$ | a =1 | a = 1 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$ | a = 1.7<br>b = 1.7 | a = 1.6<br>b = 1.6 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$, $(c)$: $\bullet^0 \to \bullet^3$ | a = 2.6<br>b = 2.6<br>c = 2.6 | a = 2.5<br>b = 2.5<br>c = 2.5 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$, $(c)$: $\bullet^0 \to \bullet^3$, $(d)$: $\bullet^4 \to \bullet^1$ | a = 2.6<br>b = 2.6<br>c = 2.6<br>d = 1.4 | a = 2.5<br>b = 2.6<br>c = 2.6<br>d = 1.4 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$, $(c)$: $\bullet^0 \to \bullet^3$, $(d)$: $\bullet^4 \to \bullet^1$, $(e)$: $\bullet^5 \to \bullet^1$ | a = 4.2<br>b = 4.2<br>c = 4.2<br>d = 2.5<br>e = 2.6 | a = 4.1<br>b = 4.1<br>c = 4.1<br>d = 2.5<br>e = 2.5 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$, $(c)$: $\bullet^0 \to \bullet^3$, $(d)$: $\bullet^4 \to \bullet^1$, $(e)$: $\bullet^5 \to \bullet^1$, $(f)$: $\bullet^6 \to \bullet^3$ | a = 4.2<br>b = 4.2<br>c = 4.2<br>d = 2.6<br>e = 2.7<br>f = 1.5 | a = 4.2<br>b = 4.1<br>c = 4.3<br>d = 2.5<br>e = 2.7<br>f = 1.6 |

FIGURE 5.9 – Comparaison des pénalités suivant la génération de carte *InfiniBand*

thèque *MPI* plus récente, comme MVAPICH2 1.4 ou OpenMPI 1.4, donnerait sûrement des résultats différents car elles utilisent mieux les possibilités de la carte ConnectX et bénéficie d'optimisations pour la réception des messages [6, 95, 102].

#### 5.4.4.3 Comparaison des pénalités avec d'autres réseaux ou implémentation *MPI*

La figure 5.10 montre une comparaison, pour un ensemble de schémas de communications identiques, entre trois réseaux haute performance : *Gigabit Ethernet*, *Myrinet* 2000 et *InfiniBand*. Les résultats présentés pour les réseaux *Gigabit Ethernet* et *Myrinet* sont issus de la thèse de Matinasso [67]. Pour le réseau *InfiniBand*, deux implémentations *MPI* seront comparées.

**Comparaison des pénalités avec d'autres réseaux**
Avant de comparer les résultats pour les différents réseaux, il convient de rappeler le mécanisme de contrôle de flux utilisé par le réseau *Gigabit Ethernet* et *Myrinet*.

***Gigabit Ethernet* :** Le contrôle de flux utilisé par *Gigabit Ethernet* est le protocole TCP [54]. Lorsqu'une carte en réception est congestionnée, elle envoie une trame *pause frame*. A la réception de cette trame, la carte réceptrice arrête l'envoi pendant une certaine durée. Le niveau de congestion ainsi que la durée d'attente ne sont pas définis dans le standard.

***Myrinet* :** Le contrôle de flux utilisé est le protocole *Stop & Go*. Cela signifie qu'une carte ne peut pas envoyer (*Go*) et recevoir (*Stop*) un message au même moment.

Globalement, la répartition de la bande passante diffère suivant le réseau haute performance. Pour certains schémas, des pénalités supérieures au nombre de communications apparaissent. Ce temps perdu est lié au temps additionnel nécessaire pour la gestion de la concurrence via le contrôle de flux.
*Gigabit Ethernet* et *InfiniBand* ont des comportements très différents sur des schémas complexes. Lorsque la communication $a$ est ralentie par la réception des messages $d$ et $e$ sur le nœud 1, l'ensemble des communications ayant le même nœud source que $a$ est ralenti alors que ce n'est pas le cas pour *Gigabit Ethernet*.
*Myrinet* et *InfiniBand* semblent avoir un comportement relativement proche pour ces schémas. L'explication pour cela est assez simple. Sur *Myrinet*, comme énoncé auparavant, une carte ne peut envoyer qu'un seul message à la fois. Sur *InfiniBand*, une carte ne peut pas envoyer de données si la carte réceptrice ne dispose pas assez d'espace pour accueillir le message. En regardant les deux derniers schémas de communications, il est possible de voir que la communication $a$, $b$ et $c$ ralentissent. La raison est liée aux fait que les paquets des messages sont envoyés selon le principe *FIFO*, et vu que les paquets de la communication $a$ sont en conflit avec les paquets des communications $d$ et $e$, ceux-ci sont ralentis, bloquant les paquets des communications $b$ et $c$.

| Schémas de communication | Réseau | | | |
|---|---|---|---|---|
| | Gigabit Eth. | Myrinet 2000 | Infiniband Scali MPI | Infiniband MPIBULL2 |
| $\bullet^0 \xrightarrow{(a)} \bullet^1$ | a = 1 | a = 1 | a =1 | a=1 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$ | a = 1.5<br>b = 1.5 | a =1.9<br>b = 1.9 | a = 1.6<br>b = 1.6 | a = 1.6<br>a = 1.6 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$, $(c)$: $\bullet^0 \to \bullet^3$ | a = 2.2<br>b = 2.2<br>c = 2.2 | a = 2.8<br>b = 2.8<br>c = 2.8 | a = 2.5<br>b = 2.5<br>c = 2.5 | a = 2.5<br>b = 2.5<br>c = 2.5 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$, $(c)$: $\bullet^0 \to \bullet^3$, $(d)$: $\bullet^4 \to \bullet^1$ | a = 2.1<br>b = 2.1<br>c = 2.1<br>d = 1.1 | a = 2.8<br>b = 2.8<br>c = 2.8<br>d = 1.5 | a = 2.6<br>b = 2.7<br>c = 2.6<br>d = 1.6 | a = 2.6<br>b = 2.6<br>c = 2.6<br>d = 1.6 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$, $(c)$: $\bullet^0 \to \bullet^3$, $(d)$: $\bullet^4 \to \bullet^1$, $(e)$: $\bullet^5 \to \bullet^1$ | a = 4.4<br>b = 2.6<br>c = 2.6<br>d = 2.6<br>e = 2.6 | a = 4.4<br>b = 4.2<br>c = 4.2<br>d = 2.5<br>e= 2.5 | a = 4.2<br>b = 4.2<br>c = 4.2<br>d = 2.5<br>e = 2.5 | a = 4.1<br>b = 4.1<br>c = 4.1<br>d = 2.5<br>e = 2.5 |
| $\bullet^0$, $(a)$: $\bullet^0 \to \bullet^1$, $(b)$: $\bullet^0 \to \bullet^2$, $(c)$: $\bullet^0 \to \bullet^3$, $(d)$: $\bullet^4 \to \bullet^1$, $(e)$: $\bullet^5 \to \bullet^1$, $(f)$: $\bullet^6 \to \bullet^3$ | a = 4.4<br>b = 2.0<br>c = 3.3<br>d = 2.6<br>e = 2.6<br>f = 1.4 | a = 4.5<br>b = 4.5<br>c = 4.5<br>d = 2.5<br>e= 2.5<br>f = 1.3 | a = 4.2<br>b = 4.2<br>c = 4.2<br>d = 2.7<br>e = 2.6<br>f = 1.5 | a = 4.2<br>b = 4.1<br>c = 4.3<br>d = 2.5<br>e = 2.7<br>f = 1.6 |

FIGURE 5.10 – Comparaison des pénalités suivant le réseau ou l'implémentation *MPI*

**Comparaison d'implémentation *MPI***
Les deux implémentations testées proposent des pénalités identiques pour l'ensemble des schémas étudiés. La raison du résultat est la même que pour les générations de cartes différentes. Les deux implémentations s'appuient uniquement sur la bibliothèque de bas niveau *libibverbs* pour l'envoie et la réception des messages sans proposer de mécanisme de contrôle de flux software ou exploitant mieux les possibilités de la carte *ConnectX*.

#### 5.4.4.4 Impact du routage

Comme cela a déjà été annoncé auparavant, *InfiniBand* utilise un routage statique. Or dans une grappe de calcul, il peut arriver que des nœuds soient déconnectés, suite à une panne par exemple, voire que des liens réseaux soient ajoutés ou retirés. Dans son protocole, *InfiniBand* détecte la disparition d'un lien et, si cela arrive, déterminera alors un nouveau chemin à l'ensemble des messages qui passaient par ce lien. Malheureusement, si ce lien réapparaît, il n'y a pas de restauration des chemins modifiés. Cela peut amener à une perte de performance considérable surtout si un lien est saturé.
Un autre élément pouvant amener à un mauvais routage est la mauvaise définition de l'algorithme de routage. Par défaut *opensm*, qui est l'agent qui s'occupe du routage, utilise l'algorithme *Min Hop*. Un tel algorithme pourrait mal diriger les messages d'une grappe de calcul utilisant une topologie *Fat-Tree*.
Pour illustrer ces propos, des expériences sur la grappe *griffon* de Nancy, disposant d'un réseau *InfiniBand* (10Gb) composé de cartes ConnectX DDR, ont été effectuées. Ces expériences consistant en la mesure de la bande-passante lorsque plusieurs communications passent au travers d'un commutateur au même instant.
La première expérience consistera à mesurer la bande passante qui servira de référence.

- Griffon-2 vers Griffon-3 : bande passante = 1334 Mo/s

La deuxième expérience utilisera deux communications.

- Griffon-1 vers Griffon-13 : bande passante = 1322 Mo/s
- Griffon-2 vers Griffon-14 : bande passante = 1335 Mo/s

La troisième expérience utilisera deux communications utilisant un même lien à l'intérieur du commutateur :

- Griffon-10 vers Griffon-13 : bande passante = 920 Mo/s
- Griffon-4 vers Griffon-14 : bande passante = 965 Mo/s

La quatrième expérience utilise cinq communications ayant un lien commun à l'intérieur du commutateur :

- Griffon-3 vers Griffon-34 : bande passante = 385 Mo/s
- Griffon-4 vers Griffon-59 : bande passante = 384 Mo/s
- Griffon-5 vers Griffon-20 : bande passante = 382 Mo/s
- Griffon-6 vers Griffon-56 : bande passante = 385 Mo/s
- Griffon-9 vers Griffon-17 : bande passante = 382 Mo/s

La dernière expériences rajoute, à ces cinq communications, deux communications n'ayant pas de lien communs avec elles.

- Griffon-3 vers Griffon-34 : bande passante = 387 Mo/s
- Griffon-4 vers Griffon-59 : bande passante = 382 Mo/s
- Griffon-5 vers Griffon-20 : bande passante = 386 Mo/s
- Griffon-6 vers Griffon-56 : bande passante = 382 Mo/s
- Griffon-9 vers Griffon-17 : bande passante = 384 Mo/s
- Griffon-1 vers Griffon-13 : bande passante = 1333 Mo/s
- Griffon-2 vers Griffon-14 : bande passante = 1332 Mo/s

Il est à noter que le débit des cartes est de 10Gbits/s pour les communications unidirectionnelles et que les liens à l'intérieur du commutateur sont à 16 Gbit/s.
Ces simples expériences montrent l'importance du routage pour l'obtention de bonnes performances dans un réseau *InfiniBand*. L'intérieur des commutateurs *InfiniBand* a une configuration de type *crossbar*, composée de ligne et de colonne. Le problème, illustré ici, est lié aux faits que les communications des nœuds 3, 4, 5, 6, 9 et 10 utilisent la même ligne, engendrant donc un partage de la bande passante. Le partage de la bande passante est alors effectué via un autre mécanisme de contrôle de flux, appelé *static rate control*, qui équilibre les débits. La seule solution pour résoudre ce problème est une recréation de la table de routage.
En terme de pénalité, un mauvais routage peut fausser l'ensemble des résultats d'une expérience. Par exemple, si l'on souhaiterait connaître la pénalité liée à un schéma correspondant à 3 communications sortantes chacune d'un nœud différent et se dirigeant vers un autre nœud, les résultats seraient différents si les communications partaient des nœuds 3, 4 et 5 et des nœud 2, 3 et 4. Voilà pourquoi notre programme de test vérifie le routage afin d'être sûr qu'il n'existe pas de conflit de ce type.

### 5.4.5 Bilan des expériences

L'ensemble des expériences a permis de montrer l'impact de la concurrence sur le partage de la bande passante pour le réseau *InfiniBand*. Ce retard lié à ce partage a été appelé pénalité. Il a été possible de voir que cette pénalité évolue en fonction du schéma de communication, mais reste indépendante de la génération de la carte réseau et de l'implémentation utilisée. Pour des schémas de communication complexes, il a été montré une certaine hiérarchie dans les pénalités, liée au contrôle de flux utilisé par le réseau *InfiniBand*.
Les différentes expériences ont permis de montrer que l'évolution des pénalités était liée à la création ou à la terminaison d'une communication et qu'elle pouvait s'expliquer par le contrôle de flux. Il faut donc mettre en place un processus de modélisation des pénalités permettant de calculer leurs valeurs pour chacune des communications en s'appuyant sur le comportement du contrôle de flux.

## 5.5 Modèle de répartition de bande passante du réseau *InfiniBand*

Pour utiliser le modèle prédictif de temps de communication défini par Martinasso et utilisé dans son simulateur, il nous faut définir un modèle de répartition de la bande passante pour le réseau *InfiniBand* basé sur les observations présentées auparavant.

### 5.5.1 Objectif

L'objectif du modèle est prédire les pénalités de chaque communication pour un graphe de communication donné. Les graphes considérés étant statiques et correspondant aux schémas de communication entre deux événements, c'est-à-dire le commencement ou la terminaison d'une communication.

### 5.5.2 Approche

Si l'on regarde le bilan des différentes expériences décrites dans ce chapitre, plusieurs éléments importants restent à retenir :

- Lorsqu'un nœud émet plus de deux communications sortantes et que si l'un des nœuds destination reçoit plus de trois messages, le contrôle de flux *InfiniBand* ralentie l'ensemble des communications sortantes
- Il existe une priorité entre les conflits.
- Le partage de la bande passante entre un conflit basé sur des communications entrantes est différent de celui basé sur des communications sortantes.

Avant d'exposer le modèle, il convient de définir certaines notations. Considérons la figure 5.11 qui représente une communication $c_i$ allant d'un nœud $s$ vers un nœud $e$. Cette communication rentre en conflit en réception au niveau du nœud $e$ mais est aussi en conflit en émission au niveau du nœud $s$. Cela amène à définir deux notations :

- $\Delta_e(e)$ qui correspond au degré entrant du nœud $e$, c'est-à-dire le nombre de communications ayant comme destination le nœud $e$.
- $\Delta_s(s)$ qui correspond au degré sortant du nœud $s$, c'est-à-dire le nombre de communications émises par le nœud $s$.

Il faut rajouter à cela deux autres notations :

- $\Phi(e)$ qui correspond au nombre de communications ayant pour destination le nœud $e$ mais venant de nœuds différents.
- $\Omega(s, e)$ qui correspond au nombre de message venant du nœud $s$ vers le nœud $e$.

Si le nœud $e$ ne reçoit que des communications venant de nœuds différents alors $\Phi(e)$=$\Delta_e(e)$. Par contre si, par exemple, il y a trois messages venant du nœud $s$ et allant du nœud $e$ alors $\Phi(e) \neq \Delta_e(e)$ et $\Omega(s, e) = 3$

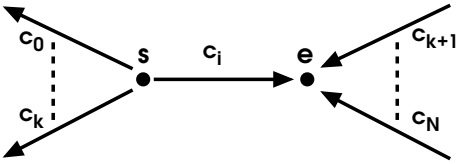



FIGURE 5.11 – Schéma détaillé d'une communication.

Pour déterminer la pénalité d'une communication $c_i$, deux valeurs sont calculées. La pénalité provoquée par le conflit en émission, notée $p_s$ et la pénalité provoqué par le conflit en réception, notée $p_e$.

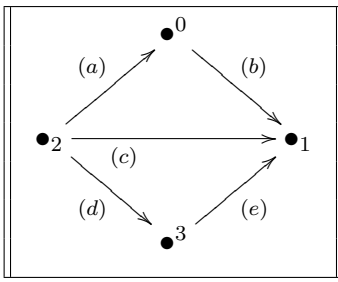
$$p_s = \begin{cases} 1 \text{ si } \Delta_s(i) = 1 \\ \begin{cases} \text{si } \Delta_s(i) \geq 2 \; et \; \Delta_e(i) \geq 3 : \; \Delta_s(i) \times \beta_s \times \gamma_r \\ \text{sinon : } \; \Delta_s(i) \times \beta_s \end{cases} \end{cases}$$

$$p_e = \begin{cases} 1 \text{ si } \Delta_e(i) = 1 \\ \text{sinon } \Phi(e) \times \beta_e \times \Omega(s,e) \end{cases}$$

Finalement la pénalité associée à la communication $c_i$ est :

$$p = max(p_s \in s, p_e)$$

Sachant que $\beta_s$ et $\beta_e$ représente la pénalité causée par le partage de la ressource réseau respectivement pour la communication sortante et entrante. $\gamma_r$ est le coefficient de ralentissement qui intervient lorsque le nombre de communications entrantes sature la mémoire tampon.




| Coms | $T$ mesuré [s] | $T$ prédit [s] |
|---|---|---|
| a | 0.046 | 0.049 |
| b | 0.027 | 0.031 |
| c | 0.046 | 0.049 |
| d | 0.046 | 0.049 |
| e | 0.027 | 0.031 |
| | | |

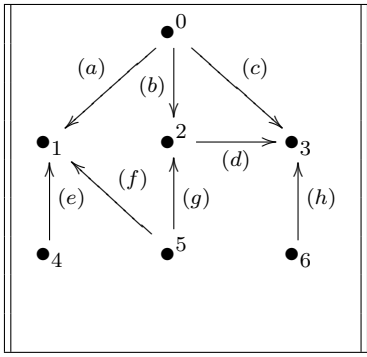
TABLE 5.3 – Vérification des paramètres, taille des communications de 20Mo, cluster Novascale.

Le calcul de $\beta_s$ s'effectue en s'appuyant sur le résultat de la pénalité obtenue pour un conflit simple S/S. Pour cela, il suffit de diviser par deux la valeur de la pénalité obtenue. Ainsi, pour les expériences précédentes, $\beta_s = \frac{1.7}{2} = 0.85$. Par analogie, le calcul de

$\beta_e$ s'effectue en s'appuyant sur le résultat de la pénalité obtenue pour un conflit simple E/E. Pour cela, il suffit de diviser par deux la valeur de la pénalité du conflit. Ainsi, pour les expériences précédentes, $\beta_e = \frac{1.6}{2} = 0.8$. Pour calculer $\gamma_r$, il faut s'appuyer sur une schéma de communication comme celui présenté sur le tableau 5.3. A cause du contrôle de flux au niveau de la communication $c$, l'ensemble des communications $a$, $c$ et $d$ sont retardées. Pour calculer ce retard, il faut étudier le rapport, appelé $\gamma_r$, entre la pénalité obtenue et la pénalité supposé (sans retard). Pour cela il suffit d'appliquer la formule : $\gamma_r = \frac{t_c}{3 \times \beta_e \times t_{ref}} = 1.58$ avec $t_{ref}$ le temps nécessaire pour envoyer 20 Mo sans concurrence.

### 5.5.3 Exemple de calcul de pénalité

Le tableau 5.4 illustre le calcul des pénalités pour un graphe donné sur une grappe de calcul *Novascale* de *BULL*.La valeur des paramètres sont $\beta_s$=0.85, $\beta_e$=0.8 et $\gamma_r$=1.58. Le tableau n'indique par la valeur de $\Phi(e)$ pour chaque communication car elle est identique à $\Delta_e(e)$. Concernant la valeur de $\Omega$, cette dernière est constante et vaut 1.




| Coms | $\Delta_s$ | $\Delta_e$ | $p_s$ | $p_e$ | $p$ |
|---|---|---|---|---|---|
| a | 3 | 3 | 4 | 2.4 | 4 |
| b | 3 | 2 | 2.4 | 1.6 | 4 |
| c | 3 | 3 | 4 | 2.4 | 4 |
| d | 1 | 3 | 1 | 2.4 | 2.4 |
| e | 1 | 3 | 1 | 2.4 | 2.4 |
| f | 2 | 3 | 2.7 | 2.4 | 2.7 |
| g | 2 | 2 | 1.9 | 1.6 | 2.7 |
| h | 1 | 3 | 1 | 2.4 | 2.4 |

TABLE 5.4 – Exemple d'application du modèle *InfiniBand* sur un graphe.

## 5.6 Conclusion

Ce chapitre a permis de comprendre et de modéliser l'impact de la concurrence au niveau des temps de communication des messages point-à-point pour le réseau haute performance *InfiniBand*.
Pour cela, une première partie s'est intéressée à décrire et expliquer le protocole mis en place par Martinasso ainsi que les outils qu'il a développés pour estimer le temps des communications soumises à la concurrence. Ce protocole a été légèrement modifié pour pouvoir prendre en compte le routage statique propre au réseau *InfiniBand*. Suite à cela, une série d'expérimentations a été menée afin de pouvoir étudier le comportement des

conflits et la valeur des pénalités en fonction du schéma de communication. Ces expériences ont permis de montrer l'impact du contrôle de flux sur la valeur des pénalités.
La dernière partie a porté sur la modélisation du partage de la bande passante du réseau *InfiniBand* en se basant sur les observations effectuées précédemment et sur une approche quantitative.
L'efficacité du modèle va être évaluée dans le prochain chapitre au travers d'un ensemble de graphes statiques et d'une application parallèle.

# III

# Evaluation du modèle

*Dans les deux précédents chapitres, deux différents thèmes ont été abordés. D'un coté un modèle d'estimation des temps de calcul et de l'autre un modèle de prédiction de temps de communications en concurrences.*

*Ce chapitre se concentrera uniquement à l'évaluation du second thème en comparant les résultats prédits pour différents schémas avec leurs résultats mesurés.*

# Evaluation de la prédiction 6

## 6.1 Introduction

Le but de ce chapitre est principalement d'évaluer le modèle de prédiction des temps de communications définis dans le chapitre 5. Cette évaluation se fera en comparant les résultats prédits avec les résultats mesurés lors d'expérimentations. Ces expérimentations seront d'abord basées sur un ensemble de graphes synthétiques, avant de finir par l'évaluation d'une application tirée de *SpecMPI*.
La première partie de ce chapitre (section 6.2) décrit l'outil de simulation pour l'estimation des temps de communication. Suite à cela, les différentes métriques utilisées pour estimer l'erreur seront présentées (section 6.3).
La seconde partie s'intéresse à l'évaluation du modèle de communication sur différents types de graphes. Pour cela, nous allons suivre le protocole d'évaluation mise en place par Martinasso. L'analyse se portera donc, tout d'abord, sur les graphes synthétiques (section 6.4) composés d'arbres et de graphes complets qui permettront d'exploiter la pertinence du modèle. Ensuite pour finir la validation, il sera utilisé des graphes d'applications (section 6.5) issus de traces du benchmark *Soccoro*, afin de montrer la précision du modèle sur une application réelle.

## 6.2 Outil de simulation

Afin de pouvoir estimer les temps de communication, un simulateur à événements discrets [114] avait été développé par Martinasso. Ce simulateur a été modifié afin de tenir compte du routage et de notre modèle pour les communications *InfiniBand*. La figure 6.1 illustre les paramètres d'entrées et les résultats du simulateur. Les paramètres d'entrées de ce simulateur sont :

– La séquence d'événements représentant l'application. Les événements étant soit des événements de calcul, défini par un temps, soit des événements de communication, défini par le numéro de la tâche source et destination ainsi que la quantité de données à transmettre.

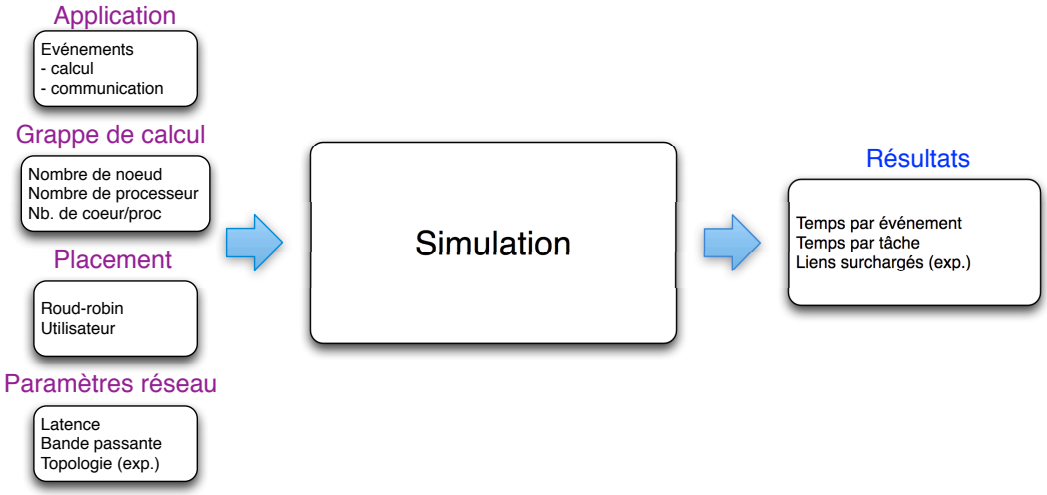



FIGURE 6.1 – Framework du simulateur

- Les caractéristiques de la grappe de calcul. A savoir, le nombre de nœud, le noms des nœuds, le nombre de processeurs par nœud ainsi que le nombre de cœur par processeur.
- Le placement des tâches. Cela peut se faire automatiquement via un placement de type *Round-Robin* ou manuellement. Le placement permet de savoir exactement où sont placées les différentes tâches au sein du nœud.
- Les paramètres réseaux. A savoir la latence, la bande passante ainsi que la taille du tampon pour connaître le moment du changement de protocole. Il est possible de donner aussi la table de routage ainsi que la topologie du réseau *InfiniBand* afin de tenir compte du partage de bande passante au niveau du commutateur mais cette implémentation n'a pas été complètement validée et reste expérimentale.

Le simulateur ne considère les événements de communication que comme des communications point-à-point synchrones. La durée d'une communication comprend le temps de synchronisation entre l'émetteur et le récepteur ainsi que le transfert des données. Les résultats obtenus par le simulateur sont :

- Le temps de communication de chaque événement de communication
- Le temps totale par tâche *MPI* ainsi que le volume de communication pour chacune de ces tâches
- Les liens du commutateurs qui sont surchargés, indiquant alors un problème de routage ou de câblage. Ce résultat est encore expérimentale car il nécessite un plus grand nombre d'expériences pour être validé.

Ce simulateur permet maintenant d'estimer le temps des communications point-à-point soumises à la concurrence pour les réseaux *Gigabit Ethernet*, *Myrinet* et *InfiniBand*. De plus, la version actuelle prend maintenant en charge la mise en correspondance des messages lorsque le numéro du récepteur ou du destinataire vaut *MPI_ANY_SRC* en utilisant

le principe d'une file d'attente FIFO.

## 6.3 Méthode d'évaluation

L'évaluation du modèle s'effectuera en comparant le temps de communication prédit $T_p$ avec le temps de communication mesuré $T_m$. Cette comparaison se fera en calculant l'erreur relative et l'erreur absolue. L'erreur relative permet de voir la précision au niveau de la prédiction de la communication, alors que l'erreur absolue donne une vision plus globale de l'erreur. Pour un graphe $G$, comportant $N$ communications notées $c_k$ (avec $0 \leq k < N$), le calcul des erreurs est le suivant :

$$E_{rel}(c_k) = |\frac{T_p - T_m}{T_m}| \times 100$$
$$E_{abs}(G) = \frac{1}{N}\sum_{k=1}^{N} E_{rel}(c_k)$$

Le temps prédit $T_p$ étant le temps obtenu par notre simulateur.

## 6.4 Évaluation sur des graphes synthétiques

Deux types de graphes synthétiques seront étudiés. Il y aura d'abord les arbres puis les graphes complets. Ce choix est lié aux différents types de conflits que proposent ces graphes avec d'un coté, des graphes comportant peu de communications concurrentes (arbres) et de l'autre des graphes où chaque communication sortante est en conflit avec au moins une communication sur son nœud destination.
Sur chacun des graphes, l'ensemble des communications débute simultanément et transmet 20 Mo de données. Chaque communication part d'un cœur différent au niveau du nœud. Le temps sans conflit d'une communication point-à-point entre deux nœuds de 20 Mo est de 14 millisecondes. Pour ces expériences, une grappe de calcul *Novascale* équipée de cartes ConnectX sera utilisée. Les paramètres calculés pour la prédiction sont $\beta_s = 0.85$, $\beta_e = 0.8$, $\gamma_r = 1.58$

### 6.4.1 Arbres

Les arbres ne proposent pas de conflits complexes, mais restent intéressant pour étudier l'efficacité du modèle sur des conflits simples. La figure 6.2 indique les résultats obtenus pour quatre types d'arbres orientés. Les arbres A1 et A2 ont une orientation de communications opposés alors que l'arbre A3 et A4 proposent un grand nombre de conflits entrants et sortants. Le temps des communications est en milliseconde.

| Schémas | Temps des communications (ms) | | | |
|---|---|---|---|---|
| A1 | com. | $T_m$ | $T_p$ | Erreur ($E_{rel}$) |
| | a | 35 | 33 | 5.7 |
| | b | 35 | 33 | 5.7 |
| | c | 34 | 33 | 2.9 |
| | d | 23 | 22 | 4.3 |
| | e | 23 | 22 | 4.3 |
| | f | 22 | 22 | 0 |
| | g | 23 | 22 | 4.3 |
| | Moyenne des erreurs absolues $E_{abs}$ = 3.9 | | | |
| A2 | com. | $T_m$ | $T_p$ | Erreur ($E_{rel}$) |
| | a | 31 | 33 | 6.4 |
| | b | 30 | 33 | 10 |
| | c | 31 | 33 | 6.4 |
| | d | 19 | 22 | 15.7 |
| | e | 20 | 22 | 10 |
| | f | 21 | 22 | 4.7 |
| | g | 20 | 22 | 10 |
| | Moyenne des erreurs absolues $E_{abs}$ = 9 | | | |
| A3 | com. | $T_m$ | $T_p$ | Erreur ($E_{rel}$) |
| | a | 22 | 23 | 4.5 |
| | b | 22 | 23 | 4.5 |
| | c | 21 | 22 | 4.7 |
| | d | 22 | 22 | 0 |
| | e | 13 | 14 | 7.7 |
| | f | 21 | 22 | 4.7 |
| | g | 22 | 22 | 0 |
| | Moyenne des erreurs absolues $E_{abs}$ = 3.7 | | | |
| A4 | com. | $T_m$ | $T_p$ | Erreur ($E_{rel}$) |
| | a | 31 | 33 | 6.4 |
| | b | 22 | 23 | 4.5 |
| | c | 35 | 36 | 2.8 |
| | d | 31 | 33 | 6.4 |
| | e | 31 | 33 | 6.4 |
| | f | 34 | 36 | 5.8 |
| | g | 34 | 36 | 5.8 |
| | Moyenne des erreurs absolues $E_{abs}$ = 5.44 | | | |

FIGURE 6.2 – Précision du modèle *InfiniBand* : cas des arbres

Pour l'arbre A1, les prédictions sont bonnes avec une erreur de prédiction de 2 millisecondes au maximum. Cela montre que les pénalités des conflits sortants sont assez bien prédites. Par contre, pour l'arbre A2, composé de conflits entrants, l'erreur absolue est plus grande avec une erreur relative de 15.7% pour la communication $d$. Malgré le fait que les conflits du nœud 1 sont identiques au conflit du nœud 3, les mesures effectuées sont différentes alors que les prédictions sont identiques. Les conflits entrants peuvent certainement avoir une légère variabilité qui n'est pas prise en compte dans le modèle.
L'arbre A3 et A4, combinant des conflits homogènes, montrent de bons résultats. De manière générale, le modèle proposé permet d'avoir de très bonnes estimations. Néanmoins, vu que le réseau *InfiniBand* possède une bande passante très importante, la moindre erreur d'estimation de pénalité peut entraîner une grosse erreur de prédiction.

### 6.4.2 Graphes complets

Dans ce type de graphe, chaque communication est en conflit aussi bien au niveau de la source que de la destination. De plus, chaque nœud est en conflit avec quatre communications simultanément, permettant ainsi de bien étudier les corrélations entre les conflits. L'évaluation se fera avec trois graphes complets ayant chacun cinq nœuds, soit dix communications par graphe. La figure 6.3 permet de voir les résultats obtenus.
Le graphe K1 comporte un ensemble de combinaisons de conflits entrants et sortants. L'erreur absolue résultante de la prédiction est satisfaisante, mais il est possible de constater une erreur de 17.5% pour la communication $j$. Il semblerait que cette communication ne soit pas ralentie autant que les autres au niveau du nœud 2. Cela pourrait s'expliquer par le fait que les autres nœuds émettent d'autres communications sortantes qui sont ralenties au niveau de la réception. Néanmoins, cette accélération n'est pas prise en compte par le modèle.
Sur le graphe K2, chaque nœud reçoit autant de communications qu'il n'en envoie. Le même effet que précédemment est observé, certaines communications semblent finir plus rapidement que d'autres. Néanmoins, l'écart entre le temps des communications accélérées et le temps moyen des communications n'est que de 1 milliseconde. Les erreurs obtenues semblent donc plus liées à une éventuelle variabilité du réseau qu'à un réel effet d'accélération.
Le graphe K3 représente des conflits plus hétérogènes. Même si l'erreur absolue est faible, le même phénomène que le graphe K1 est observé. Certaines communications semblent profiter du ralentissement d'autres communications et se retrouvent avec un temps de communication réel beaucoup plus faible que celui prédit. Il semblerait, encore une fois, que le ralentissement de certaines communications crée une accélération de la réception d'autres messages.

| Schémas | Temps des communications (ms) | | | |
|---|---|---|---|---|
| K1 | com. | $T_m$ | $T_p$ | Erreur ($E_{rel}$) |
| | a | 69 | 72 | 4.3 |
| | b | 69 | 72 | 4.3 |
| | c | 69 | 72 | 4.3 |
| | d | 70 | 72 | 6 |
| | e | 50 | 53 | 6 |
| | f | 54 | 53 | 1.8 |
| | g | 52 | 53 | 1.9 |
| | h | 30 | 33 | 10 |
| | i | 45 | 47 | 4.4 |
| | j | 40 | 47 | 17.5 |
| | Moyenne des erreurs absolues $E_{abs}$ = 8.6 | | | |
| K2 | com. | $T_m$ | $T_p$ | Erreur ($E_{rel}$) |
| | a | 22 | 24 | 9 |
| | b | 23 | 24 | 4.3 |
| | c | 21 | 24 | 14.2 |
| | d | 22 | 24 | 9 |
| | e | 22 | 24 | 9 |
| | f | 22 | 24 | 9 |
| | g | 21 | 24 | 14.2 |
| | h | 22 | 24 | 9 |
| | i | 22 | 24 | 9 |
| | j | 22 | 24 | 9 |
| | Moyenne des erreurs absolues $E_{abs}$ = 10.4 | | | |
| K3 | com. | $T_m$ | $T_p$ | Erreur ($E_{rel}$) |
| | a | 66 | 71 | 7.5 |
| | b | 66 | 71 | 7.5 |
| | c | 66 | 71 | 7.5 |
| | d | 66 | 71 | 7.5 |
| | e | 18 | 22 | 22.2 |
| | f | 20 | 22 | 10 |
| | g | 31 | 34 | 9.6 |
| | h | 33 | 34 | 3 |
| | i | 31 | 34 | 9.6 |
| | j | 29 | 34 | 17.2 |
| | Moyenne des erreurs absolues $E_{abs}$ = 10.1 | | | |

FIGURE 6.3 – Précision du modèle *InfiniBand* : cas des graphes complets

### 6.4.3 Discussion

Les graphes synthétiques présentés ici ont permis de montrer à la fois la bonne approximation du modèle mise en place, mais aussi des effets d'accélération non pris en charge par ce modèle qui demanderont un ajustement de celui-ci. De manière générale, notre modèle s'avère souvent pessimiste en prédisant des temps de communication souvent plus long que les mesures réelles. Mais les temps des communications étant très faible (à cause de la très grande bande passante du réseau *InfiniBand*), une plus grande précision sera difficile à obtenir.

## 6.5 Évaluation du modèle de communication sur une application

Maintenant que le comportement du modèle a été étudié sur des graphes de communications synthétiques et qu'il a montré d'assez bons résultats. Il convient maintenant de tester les prédictions des temps de communications d'une application réelle. Trouver une application comportant un très grand nombre de communications en concurrence est assez difficile, car celles-ci sont souvent optimisées afin de communiquer uniquement avec leurs voisins proches (cœurs du même nœud), réduisant les communications utilisant le réseau et donc l'effet de concurrence. Néanmoins, au sein de *SpecMPI*, il existe un benchmark, appelé *Socorro*, qui utilise un très grand nombre de communications bloquantes point-à-point soumises à la concurrence.
Cette section sera en trois parties. Tout d'abord, le benchmark Socorro sera brièvement présenté. Puis, la méthode utilisée pour obtenir le graphe de communication logique sera expliquée. Et enfin une évaluation détaillée sur 32 cœurs de Socorro sera présentée ainsi que les résultats obtenus pour 48 et 64 cœurs.

### 6.5.1 Le programme Socorro

Socorro est à la base un programme permettant d'utiliser la théorie de la fonctionnelle de la densité (*Density Functional Theory* ou *DFT*) sur des grappes de calcul haute performance. La plupart du code est écrit en langage Fortran 90/95 et est complété par des routines de support en C++. Socorro a été sélectionné pour faire parti des applications présente dans la suite de benchmark SpecMPI [75, 100]. Il est néanmoins disponible sous la licence GNU GPL [1].

Socorro est intéressant, car les tâches *MPI* communiquent avec toutes les autres tâches et cela durant les mêmes intervalles de temps. La figure 6.4, représentant la matrice de communication pour 64 processus *MPI*, permet d'avoir un aperçu du nombre de messages

1. http ://dft.sandia.gov/Socorro/mainpage.html

échangés. L'ordonnée représente les tâches émettrices, l'abscisse les tâches réceptrices et la couleur des cases identifies le nombre de messages point-à-point échangés. Cette particularité amène donc un partage de la ressource réseau sur une période commune. Cette application est donc parfaite pour évaluer le modèle de partage de bande passante. Mais pour pouvoir avoir une estimation des temps des communications, il faut d'abord générer une trace de l'application.

### 6.5.2 Obtention des traces

L'obtention d'une trace fidèle pour obtenir le graphe de communication logique d'une application est difficile car il est nécessaire d'être le moins intrusif possible afin de ne pas perturber les mesures [65].
Différentes applications existent pour réaliser cette tâche. Pour les grappes de calcul utilisant des processeurs *Intel*, il existe *Intel Trace Analyzer and Collector*(*ITAC*) [52], mais le format d'écriture des traces est propriétaire. Il existe d'autres logiciels libres pour l'analyse et le profilage d'application tel que TAU (Tuning and Analysis Utilities) [72, 87] ou Scalasca [103, 123], mais le format de trace n'est pas modifiable afin d'avoir uniquement des événements de calcul et de communication. Afin d'avoir un format de trace adapté au simulateur, l'outil MPE [118] avait été utilisé par Martinasso [67], et le sera encore

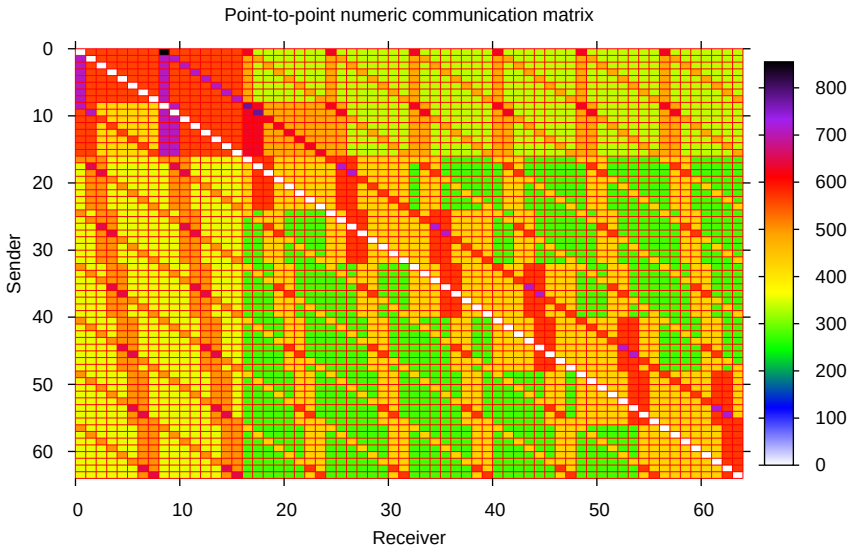



FIGURE 6.4 – Matrice de communication point-à-point pour l'application Socorro avec 64 processus *MPI* en taille mref.

pour Socorro. Cet outil permet d'avoir le graphe logique des communications *MPI*. Une mesure du temps avant et après l'appel de la fonction *MPI* permet d'avoir les temps entre les appels *MPI* et donc le temps des événements de calcul. Le résultat obtenu fournit donc des événements de calcul et les paramètres des appels des fonctions *MPI* au sein d'une trace. L'impact du coût de traçage, via cette méthode, est relativement faible et ne modifie le temps d'exécution que très faiblement (<1%).

### 6.5.3 Évaluation

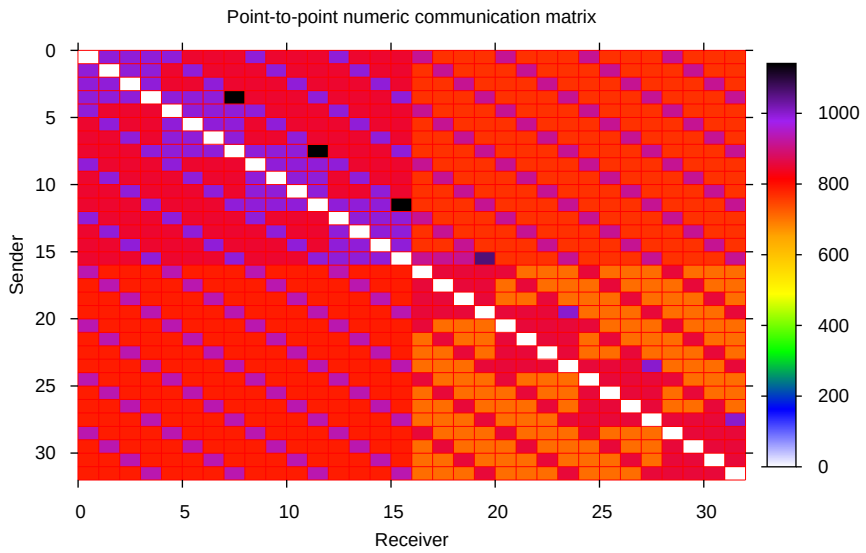



FIGURE 6.5 – Matrice de communication point-à-point pour l'application Socorro avec 32 processus *MPI* en taille mref

L'évaluation du modèle se fera donc à partir des traces obtenues avec MPE et sera focalisée sur l'étude de Socorro avec 32 cœurs pour une taille de type mref (problème de taille "moyenne" sous *SpecMPI*). La figure 6.5, représentant la matrice de communication, permet de constater que l'ensemble des tâches de Socorro communiquent avec toutes les autres tâches et cela de façon importante étant donné qu'il y a au minimum autour de 750 messages échangés. La taille de chaque message étant entre 14 et 16ko.
Le placement des tâches se fera selon l'ordre round robin par processeur, c'est-à-dire que les tâches seront assignées en remplissant d'abord le premier nœud.
La figure 6.6 permet de comparer les temps mesurés des communications avec les temps prédits des communications pour 32 processus *MPI*. Globalement, l'erreur de prédiction

est de 8.91%, ce qui est relativement faible. L'erreur maximal de prédiction est de 11.23% pour le processus *MPI* 19 et l'erreur minimal est de 5% pour le processus *MPI* 25.

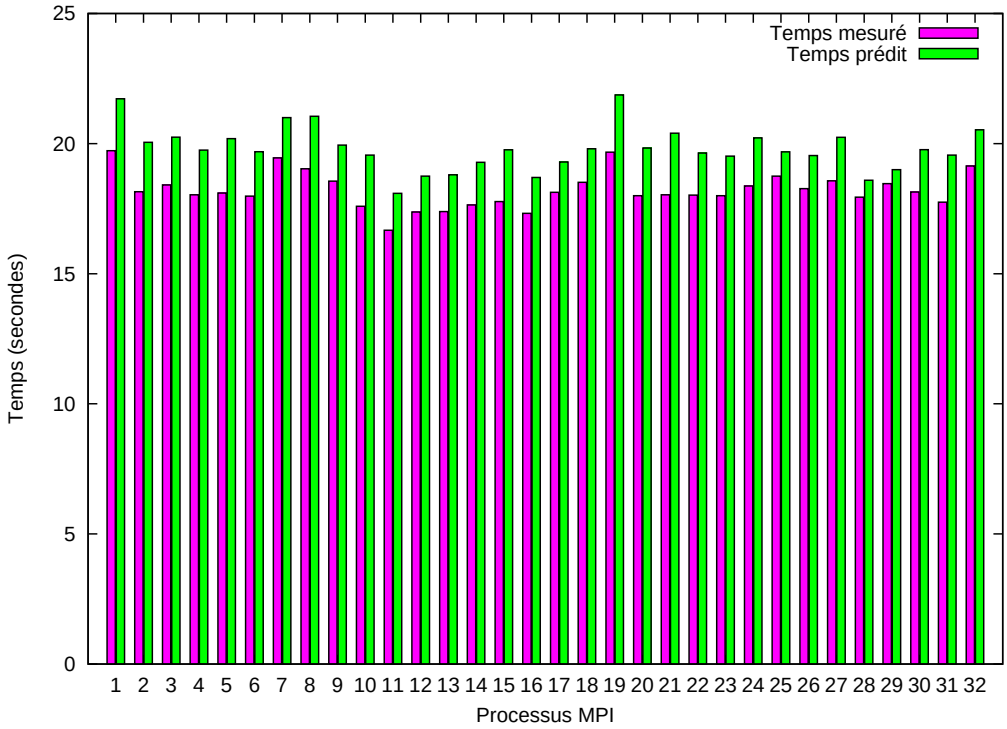



FIGURE 6.6 – Évaluation du modèle *InfiniBand* avec Socorro avec 32 processus *MPI*

D'autres résultats ont été obtenus pour un nombre de processus *MPI* différents et sont visibles sur la figure 6.7.

| Nb Tâche | Temps mesuré [s] | Temps prédit [s] | Erreur (%) |
|---|---|---|---|
| 32 | 583 | 634 | 8.91 |
| 48 | 829 | 921 | 11.11 |
| 64 | 1830 | 2018 | 10.31 |

FIGURE 6.7 – Comparaison entre le temps prédit et mesuré avec MPIBULL2

De façon générale, les résultats obtenus montrent la bonne précision de notre modèle. Néanmoins, sur une application comme Socorro, il est dommage de ne pas pouvoir actuellement générer une carte des conflits. En effet, même si nos prédictions semblent corrects, il serait intéressant de pouvoir observer l'ensemble des conflits générés afin d'obtenir un ensemble de graphes de communication. Une fois les graphes obtenues, il serait possible

de comparer les mesures prédites avec les mesures obtenues afin de pouvoir encore améliorer le modèle.
De plus, l'approche a des limites liées à la congestion mémoire. En effet, plus le nombre de cœurs est important au sein du nœud, plus l'impact du problème de congestion mémoire est important. Or, cette approche ne permet de prendre en compte ce type de phénomène.

## 6.6 Bilan

L'objectif de ce chapitre a été de démontrer la précision du modèle développé au cours de cette thèse pour les communications point-à-point soumises à la concurrence. Tout d'abord, une validation auprès de graphes synthétiques a été effectuée. Les résultats obtenus se sont montrés concluant. Ensuite, une estimation des temps de communication a été effectuée sur l'application *Socorro*. Pour cela, il a fallu d'abord obtenir des traces de l'application pour avoir une décomposition en événements. Le choix de l'outil de traçage s'est porté sur *MPE*. Une fois les traces obtenues, une comparaison entre les temps des différentes tâches *MPI* de *Socorro* prédits et mesurés a été réalisée. Même si les résultats obtenus sont correctes, il est encore difficile de pouvoir extraire de la simulation les graphes de communications soumis à la concurrence, afin d'avoir une vision plus microscopique de nos prédictions et de pouvoir améliorer le modèle.

# Conclusion et perspectives 7

## 7.1 Objectifs de la thèse

Les travaux présentés se sont intéressés à l'extrapolation des performances d'applications parallèles utilisant le réseau *InfiniBand*. Afin de pouvoir répondre au mieux aux différents appels d'offres, les constructeurs de grappes de calcul ont besoin d'outils permettant d'aider au mieux la prise de décisions en terme de design architectural. Nos travaux se sont donc intéressés à l'estimation des temps de calcul et à l'étude de la congestion sur le réseau *InfiniBand*. Ces deux problèmes sont souvent abordés de manière globale. Néanmoins, une approche globale ne permet pas de comprendre les raisons des pertes de performance liées aux choix architecturaux.
Notre approche s'est orientée vers une étude plus fine. Pour l'estimation des temps de calcul, notre choix s'est basé sur le découpage de code (*program slicing*), permettant de pouvoir estimer le temps de chaque blocs. Pour les temps de communication, nos travaux se sont inscrit dans la continuité de ceux de Martinasso. Cela a permis une étude locale de la concurrence entre les communications, tout en tenant compte de l'aspect routage nécessaire pour le réseau *InfiniBand*. Cette démarche a pu montrer le dynamisme de la concurrence sur le réseau *InfiniBand* et a permis de le comprendre, amenant à la définition d'un modèle de partage de bande passante.
D'un point de vue industriel, il est intéressant de pouvoir prédire et comprendre les comportements de l'application afin d'adapter au mieux la solution proposée.

## 7.2 Démarche proposée

La démarche proposée s'articule autour de plusieurs axes.

Pour la partie calcul, la démarche s'appuie sur la décomposition du code source en bloc d'instructions via une méthode de découpage de code (*program slicing*). Ces blocs étant ensuite benchés afin de pouvoir obtenir les deux éléments clés pour notre modèle, à savoir, le temps de chargement de donnée dans la mémoire ainsi que le temps de traitement

des données lorsqu'elles sont contenues à l'intérieur de la mémoire cache. L'avantage de cette technique permet d'avoir rapidement une estimation du temps d'exécution des différentes phases de calcul présentes dans un programme parallèle. Une étude des différents facteurs influençant les performances de ces différents blocs a été menée. Néanmoins, une technique purement statique n'est pas toujours envisageable. En effet, il peut être parfois nécessaire d'associer le découpage en blocs à une exécution du programme. Les éléments recherchés étant non dépendant de l'architecture, cette étape peut être effectuée sur n'importe quel support (grappe de calcul homogène ou non).

Pour la partie communication, il a été d'abord proposé une série d'observations afin de voir les différentes caractéristiques des communications concurrentes. Pour cela, le protocole et les outils développés par Martinasso ont été modifiés afin de pouvoir les étudier. Ensuite, l'observation de différents schémas complexes a permis de comprendre les mécanismes du partage de la bande passante. Suite à cela, l'évaluation des différents facteurs pouvant influencer ce partage a permis de mieux comprendre le rôle du contrôle de flux du réseau *InfiniBand*. Enfin, un modèle de partage de répartition de bande passante a été proposé, afin de pouvoir prédire au mieux les effets de la concurrence entre les communications. Le modèle a ensuite été évalué. Tout d'abord, par un ensemble de comparaisons entre les résultats obtenus et prédits sur des graphes de communication synthétiques, puis sur une application parallèle ayant un très grand nombre de communications concurrentes.

## 7.3 Travaux réalisés

Les travaux de cette thèse se sont portés sur l'évaluation et la modélisation des temps de calcul et de la concurrence sur le réseau *InfiniBand*.
Le chapitre 4 décrit le protocole utilisé pour l'estimation des temps de calcul. Pour estimer au mieux les temps de calcul, il faut rester le plus proche possible du code et choisir donc un faible niveau d'abstraction. Le choix d'une approche par découpage du code s'est donc montré comme étant la meilleure solution. Des observations préliminaires ont permis de mettre en évidence qu'il était possible de représenter le temps d'un bloc avec deux paramètres : le temps de transfert de la mémoire vers le processeur et le temps de traitement des données lorsqu'elles sont dans le cache. La comparaison entre différents compilateurs a permis de montrer que ces deux paramètres pouvaient être suffisant au vu des comportements observés. Suite à cela, un formalisme a été proposé afin de pouvoir utiliser ces deux paramètres pour l'estimation des temps des blocs pour l'ensemble des structures existantes. Ensuite, une évaluation sur 10 blocs a permis de montrer que globalement l'erreur de prédiction est correct. Mais que, pour des blocs non inclus dans de boucle, ce découpage ne permettait pas de prendre en compte les optimisations globales effectuées par le compilateurs. Néanmoins, la technique reste à la fois simple et efficace, une fois que le processus d'analyse est mis en place.

Le chapitre 5 s'est concentré sur l'étude du partage de la bande passante sur le réseau *InfiniBand* en se basant sur le protocole de Martinasso. Une série d'expériences introductives a permis de faire apparaître le dynamisme de la congestion sur le réseau *InfiniBand*. Différents facteurs (carte réseau, version *MPI*) ont été étudiés afin de bien comprendre leurs influences. L'ensemble des expériences a permis de mettre en place un modèle de partage de bande passante pour le réseau *InfiniBand*.
Ce modèle a ensuite été implémenté dans un simulateur. Le chapitre 6 s'intéresse à la validation de la modélisation. En utilisant le simulateur, il a été possible de comparer les résultats mesurés avec ceux prédits sur un ensemble de graphes. L'évaluation sur des graphes synthétiques s'est montrée particulièrement bonne. Pour terminer une décomposition en événements de calcul et de communication du programme *Socorro* a été effectuée permettant ainsi de valider notre modèle sur une application parallèle réelle. L'erreur de prédiction pour l'ensemble s'est avérée faible.

## 7.4 Perspectives

A partir des travaux exposées, plusieurs champs d'investigation sont à explorer.

### 7.4.1 Approfondir la modélisation

Dans le cadre des communications, plusieurs pistes d'études existent pour améliorer notre modèle.
Nous avons considéré un réseau ayant une table de routage correct. Or, il peut arriver, du fait de redémarrage de machine par exemple, que le routage change, introduisant des conflits au niveau des liens réseau ou à l'intérieur du commutateur, modifiant alors les temps de communication. La prise en compte de ce phénomène est tout à fait possible, mais demande une étude approfondie du partage de la bande passante au niveau du commutateur.
Les dernières versions des cartes *InfiniBand* inclut un protocole appelé *Explicit Congestion Notification* (*ECN*) [39], réduisant la bande passante des cartes émettrices lorsque la quantité de message, passant par un même lien, est importante. Bien que ce protocole ne concerne pas l'ensemble des schémas de communications, une étude pourrait être effectuée afin de voir le réel impact de ce protocole sur l'estimation des pénalités.
Enfin, les nouveaux commutateurs Infiniband InfiniScale IV proposent la gestion du routage adaptatif [63] afin d'éviter les problèmes de congestion liés à une mauvaise table de routage. Il serait intéressant de voir si le modèle proposé permet une bonne estimation des pénalités avec ce type de commutateur.

Dans le cadre de la prédiction des temps de calcul, le modèle proposé ne considère

pour le moment qu'une seule tâche par nœud alors que les processeurs actuels comportent de plus en plus de cœurs. Une étude sur l'impact liée aux variations du chargement des données, du fait des accès concurrents à la mémoire cache, pourrait permettre d'adapter le modèle proposé aux architectures multi-cœurs. Les différents exemples proposés ont montrés une erreur plus importante dans l'estimation des blocs d'instruction hors boucle, une étude du code source généré pourrait permettre de prendre en compte les différences de code et d'améliorer les estimations.

Pour conclure, il faudrait étudier les interactions présentent entre les phases de calcul et de communications et certainement rajouter une prise en compte liée aux accès concurrents à la mémoire tel que le propose Adve(section 3.2.3).

### 7.4.2 Développement d'outils de prédiction de pertes de performance pour le réseau *InfiniBand*

Malgré l'arrivée de commutateur ayant une table de routage s'adaptant en fonction du trafic, il faudra un certain temps avant que l'ensemble des grappes de calcul ne dispose de cette technologie. La création d'outils testant la table de routage permettrait d'informer les administrateurs en expliquant, d'une part les raisons des pertes de performance de la grappe de calcul (changement dans la topologie ou mauvaise configuration de la table de routage), et en donnant, d'autre part, des solutions pour les corriger.

### 7.4.3 Utilisation de l'outil dPerf pour la génération de trace

*DPerf* [21] est un outil de prédiction de temps d'exécution d'application parallèle ou répartie en cours de développement par l'équipe *OMNI* du *LIFC*. Le framework de l'outil peut être vu sur la figure 7.1. S'appuyant sur les différentes représentations intermédiaires de *Rose*, *DPerf* décompose le code en différents blocs. Une fois cette décomposition faite, il effectue un benchmarking de tout ces blocs afin d'estimer leurs temps de calcul. Pour trouver la valeur des variables non identifiables statiquement, *DPerf* s'appuie sur un profiling de l'application effectué dynamiquement.
Une fois ce processus effectué, DPerf peut fournir deux types de trace. Soit une trace pour l'application *SimGrid* [15, 112], qui est un outil permettant la simulation d'application distribuée dans les environnements hétérogènes, soit une trace pour le simulateur [114] utilisé dans cette thèse pour notre estimation des temps de communication.

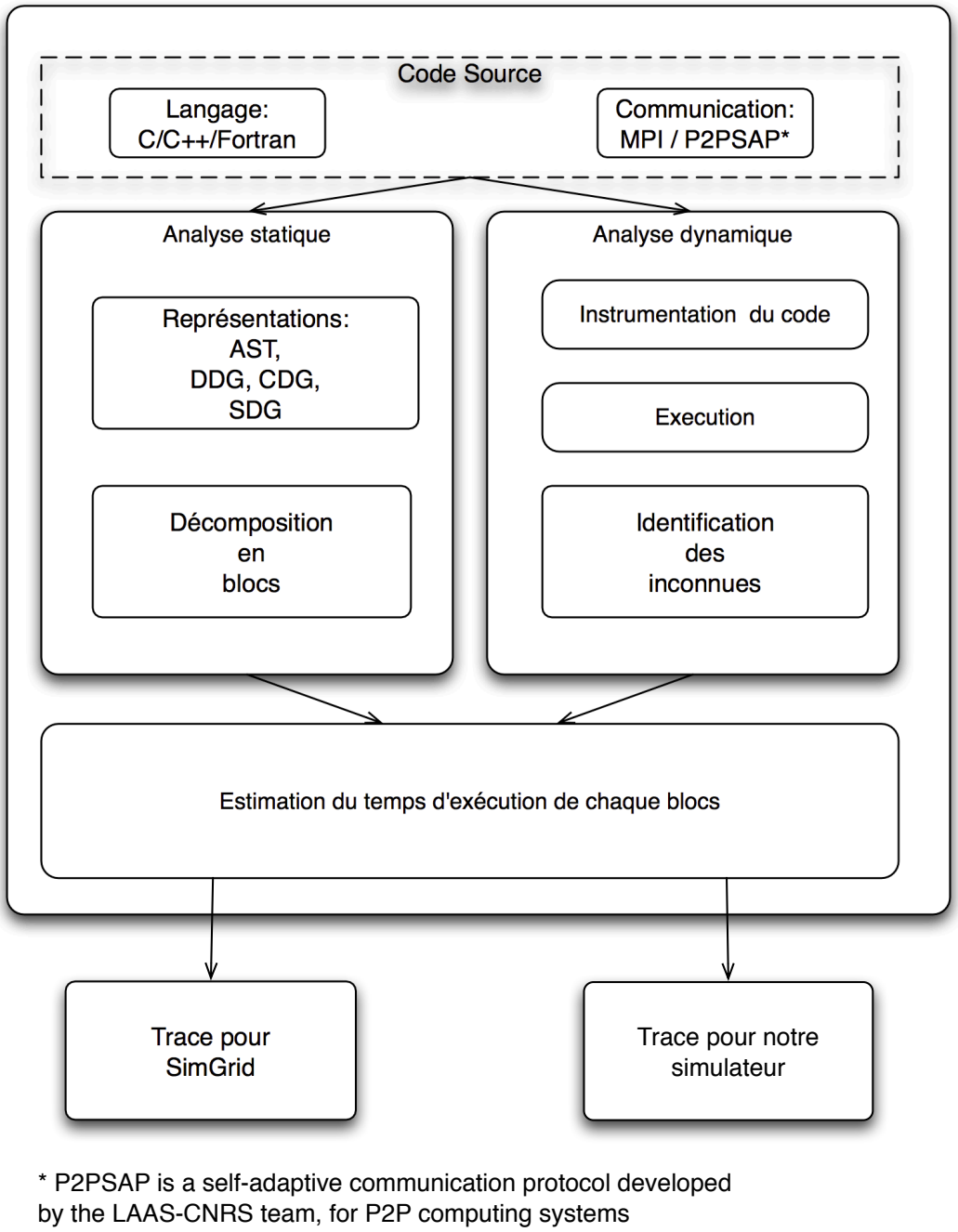



FIGURE 7.1 – Framework de l'outil DPerf

# IV

# Annexes

# PAPI A

La plupart des multiprocesseurs disposent de compteurs de performance. Déjà présenté en 2.2.1, ces compteurs sont des registres spéciaux permettant d'avoir accès à des informations sur les activités du processeur. L'avantage de ces registres est que leurs accès sont peu intrusifs et ne perturbent donc pas trop l'activité du processeur.

Il existe principalement deux différents modules qui sont perfct [77] et perfmon [78]. Une étude comparative de ces deux modules est disponible dans [126]. Notre choix s'est porté sur perfctr permettant l'accès à la bibliothèque *PAPI*. *PAPI* [72, 74], pour *Performance Application Programming Interface*, est une bibliothèque, développé à l'Université du Tennesse, qui a pour objectif de produire une spécification permettant l'accès aux compteurs de performance pour un grand nombre d'architecture processeur.

Il y a deux types d'événements possible sur *PAPI* :
- Les événements prédéfinis, indépendant de la plate-forme comme par exemple le nombre d'instruction.
- Les évènements natifs, dépendant de la plate-forme comme par exemple le nombre de cache miss de niveau 3 (L3_CACHE_MISS).

Les événements prédéfinis peuvent être dérivés d'évènements natifs. Par exemple, *PAPI_L1_TCM*, correspondant au nombre de cache miss du cache L1, est égal à la somme du nombre de cache miss niveau donnée du cache L1 et du nombre de cache miss niveau instruction du cache L1. Sur Nehalem, le nombre d'événement pouvant être reporté est égal à 117.

Deux interfaces sont disponibles sous *PAPI*, un interface de bas niveau et un second de haut niveau.
**L'interface de haut niveau** est utilisé pour effectuer des mesures simples et rapides. Ces mesures ne nécessitent aucune initialisation. Les informations disponibles, via cet interface sont, par exemple :
- *PAPI_num_counters()* permet de connaître le nombre de compteur de performance présent sur le processeur.
- *PAPI_flops()* renvoie le nombre de flops depuis le dernier appel de PAPI_flops().

**L'interface de bas niveau** permet d'effectuer des mesures très précises. Contrairement à l'interface de haut niveau, il nécessite une initialisation via *PAPI_library_init* avant l'appel des fonctions de cette interface. Les informations disponibles, via cet interface sont, par exemple :

- *PAPI_get_real_nsec* et *PAPI_get_real_usec* permettent d'accéder aux compteurs afin de connaître le temps réel écoulé en nano et en micro secondes respectivement.
- *PAPI_get_virt_nsec* et *PAPI_get_virt_usec* permettent d'accéder aux compteurs afin de connaître le temps virtuel(temps CPU) écoulé en nano et en microsecondes respectivement.

Au cours d'un cycle d'horloge, le processeur peut effectuer différentes instructions. Il passe donc un certain nombre de temps pour chacune de ces instructions. Le temps réel correspond donc au nombre de cycle d'horloge écoulé pour effectuer une instruction alors que le temps virtuel correspond au temps consacré uniquement à cette instruction.
Pour l'ensemble de nos expériences, nous utiliserons *PAPI_get_real_nsec* pour faire nos mesures.

# Code exemple B

```
#include<cstdio>

int main( ) {

//1
a = b-45+a;
b=b+a*123;
sum = b+a+322;
diff = b-a+3;
d = diff+diff;
c = diff-2 - d +b;
di = d+c;
aux=b*450;
prod=a*b+c*d;
di=prod/3;

//2
for (j=0;j<60;j++){
        a=b/a;
        sum = a*322;
        diff = (sum+b)*a;
        c = (a+b)*diff;
        d = diff-2 - c;
        b = diff * a/b;
        aux=a*45;
        a=b/b;
        b=aux*6+b;
        sum += a*a;
        diff =a+sum;
        c=d+a;
```

```
        a=d/sum;
        a+2;
        b+5;
}
//3
b = 11/a;
a=a*b;
c = a+322;
diff = (c-b)+a;
c = (a*b)/diff;
d = diff*2 - c;
d = diff + a-b;
aux=a;
a=b*b;
b=b+6;
c += a+a;
diff =a*c;
d=c/aux;
b=a*c;
a++;
b++;

//4
for (j=0;j<45;j++){
        a=a/b;
        b=a*2;
        sum = a+b;
        diff = a-b;
        c = a*b;
        d = a/b;
        mo = a/b;
        aux=a;
        a=b*b;
        b=aux;
        sum += a+a;
        diff =a-sum;
        c=d*mo;
        mo=d/sum;
        sum = sum+a+b*b+b+a;
}
//5
```

```
a=b/a*2+12;
b=b*b-a;
c=c+a+b;
d=d/(c+12/c*a+b);
sum=sum+a+b+c+d;
diff=sum-diff-a*23;
prod=a*b-c*d*diff;
di=prod/234+diff/21+sum/43;
mo=21*c+2*d+3*b+2*a+diff/di;
aux=a;
a=b*c+d;
b=aux;

//6
for (j=0;j<35;j++){
        a=123+b;
        b=b*24+a/2;
        c=c*a/c*(b+1)/b;
        d=234*a;
        sum=sum+a+b-c+d;
        diff=sum-diff-a-b-c-d;
        prod=prod+(diff+d*c+c*98);
        di=(diff+sum)/di;
        mo=di+sum+diff/c;
        aux=a-b*c+d;
        b=c/a;
}
//7
a=123+b;
b=24+a*2;
c=c+b*(b+a);
d=2*a*b;
sum=sum+a+b+c+d*21;
diff=sum-diff*26-a*2-b/3-c-d;
prod=diff/2+ a*d+c*(c+c*8);
di=diff/sum*12;
mo=di*sum-diff*c;
aux=a*b*c*d/2;

//8
for (j=0;j<35;j++){
```

```
            b=a/2;
            c=c+a/b;
            d=12*b-a;
            sum=sum+a+d*21;
            diff=sum-diff;
            prod=diff/2+d;
            di=diff/d*12;
            mo=di/diff;
            aux=a*b;
}
//9
a=6*b+43;
b=a/b+2;
c=3*c+d-a+b;
d=d/c+a/b;
sum=sum+a+b/c+d;
diff=diff -(d-a)*23-a*c-b;
prod=diff*sum;
di=a/b+b/c+c/d;
mo=mo/di;
aux=mo*2;
a=d;
d=aux;
j=0;

//10
do{
            a=a+3;
            b=b/a+9;
            c=c/a;
            d=d+c;
            sum=(sum+a+b+c+d)/2;
            diff=(c-d)*diff;
            prod=diff*2- sum/3;
            di=a/b*432;
            mo=mo/2+di;
            aux=mo*24;
            a=d/b;
            d=aux;
            j++;
} while (j<38);
```

```
return 0;
}
```

# Bibliographie


[1] V. S. ADVE, R. BAGRODIA, J. C. BROWNE, E. DEELMAN, A. DUBE, E. N. HOUSTIS, J. R. RICE, R. SAKELLARIOU, D. J. SUNDARAM-STUKEL, P. J. TELLER ET M. K. VERNON, *Poems : End-to-end performance design of large parallel adaptive computational systems*, IEEE Transactions on Software Engineering, 26 (2000), p. 1027–1048.

[2] A. AGGARWAL, A. K. CHANDRA ET M. SNIR, *On communication latency in pram computations*, *in* SPAA '89 : Proceedings of the first annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, 1989, ACM, p. 11–21.

[3] G. AMDAHL, *Validity of the single processor approach to achieving large-scale computing capabilities*, *in* Proceedings of the American Federation of Information Processing Societies, 1967, p. 483–485.

[4] H. H. AMMAR, S. M. R. ISLAM, M. H. AMMAR ET S. DENG, *Performance modeling of parallel algorithms*, *in* ICPP (3), 1990, p. 68–71.

[5] D. H. BAILEY, E. BARSZCZ, J. T. BARTON, D. S. BROWNING, R. L. CARTER, R. A. FATOOHI, P. O. FREDERICKSON, T. A. LASINSKI, H. D. SIMON, V. VENKATAKRISHNAN ET S. K. WEERATUNGA, *The nas parallel benchmarks*, rap. tech., The International Journal of Supercomputer Applications, 1991.

[6] P. BALAJI, S. BHAGVAT, D. K. PANDA, R. THAKUR ET W. GROPP, *Advanced flow-control mechanisms for the sockets direct protocol over infiniband*, *in* ICPP '07 : Proceedings of the 2007 International Conference on Parallel Processing, Washington, DC, USA, 2007, IEEE Computer Society, p 73.

[7] V. E. BENES, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York and London, 1965.

[8] M. BERRY, D. C. ET P. KOSS ET D. KUCK ET S. LO ET Y. PANG ET L. POINTER, R. ROLOFF, A. SAMEH, E. CLEMENTI, S. CHIN, D. SCHNEIDER, G. FOX, P. MESSINA, D. WALKER, C. HSIUNG, J. SCHWARZMEIER, K. LUE, S. ORSZAG, F. SEIDL, O. JOHNSON, R. GOODRUM ET J. MARTIN, *The PERFECT Club Benchmarks : Effective performance evaluation of supercomputers*, IJSA, 3 (1989), p. 9–40.

[9] F. BODIN, P. BECKMAN, D. GANNON, J. GOTWALS, S. NARAYANA, S. SRINIVAS ET B. WINNICKA, *Sage++ : An object-oriented toolkit and class library for building fortran and c++ restructuring tools*, *in* In The second annual object-oriented numerics conference, 1994, p. 122–136.

[10] S. BOKHARI, *A shortest tree algorithm for optimal assignments across space and time in a distributed processor system*, IEEE Transactions on Software Engineering, 7 (1981), p. 583–589.

[11] S. H. BOKHARI, *On the mapping problem*, IEEE Trans. Comput., 30 (1981), p. 207–214.

[12] J. BOURGEOIS ET F. SPIES, *Performance Prediction of an NAS Benchmark Program with ChronosMix Environment*, *in* Euro-Par '00 : Proceedings from the 6th International Euro-Par Conference on Parallel Processing, London, UK, 2000, Springer-Verlag, p. 208–216.

[13] S. D. BROOKES, *On the relationship of ccs and csp*, *in* Proceedings of the 10th Colloquium on Automata, Languages and Programming, London, UK, 1983, Springer-Verlag, p. 83–96.

[14] S. A. BROWNING, *The tree machine : a highly concurrent computing environment*, Thèse doctorat, California Institute of Technology, Pasadena, CA, USA, 1980.

[15] H. CASANOVA, A. LEGRAND ET M. QUINSON, *Simgrid : a generic framework for large-scale distributed experiments*, *in* Proceedings of the 10th Conference on Computer Modeling and Simulation (EuroSim'08), 2008.

[16] L. CHAI, Q. GAO ET D. K. PANDA, *Understanding the impact of multi-core architecture in cluster computing : A case study with intel dual-core system*, *in* CCGRID, IEEE Computer Society, 2007, p. 471–478.

[17] CHARLES HOWARD KOELBEL, *Compiling Programs for Distributed Memory Machines*, Thèse doctorat, Purdue University, Department of Computer Science, 1990.

[18] M. J. CLEMENT, M. R. STEED ET P. E. CRANDALL, *Network performance modeling for PVM clusters*, *in* Proceedings of Supercomputing, 1996.

[19] C. CLOS, *A study of non-blocking switching networks*, Bell System Technology Journal, 32 (1953), p. 406–424.

[20] R. COLE ET O. ZAJICEK, *The apram : incorporating asynchrony into the pram model*, *in* SPAA '89 : Proceedings of the first annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, 1989, ACM, p. 169–178.

[21] B. CORNEA ET J. BOURGEOIS, *Simulation of a P2P Parallel Computing Environment - Introducing dPerf, A Tool for Predicting the Performance of Parallel MPI or P2P-SAP Applications*, Technical report RT2010-04, LIFC, 2010.

[22] P. J. COURTOIS, F. HEYMANS ET D. L. PARNAS, *Concurrent control with "readers" and "writers"*, Commun. ACM, 14 (1971), p. 667–668.

[23] W. J. DALLY ET C. L. SEITZ, *Deadlock-free message routing in multiprocessor interconnection networks*, IEEE Trans. Comput., 36 (1987), p. 547–553.

[24] S. DASGUPTA, *A hierarchical taxonomic system for computer architectures*, Computer, 23 (1990), p. 64–74.

[25] E. W. DIJKSTRA, *Solution of a problem in concurrent programming control*, Commun. ACM, 8 (1965), p 569.

[26] E. W. DIJKSTRA, *Hierarchical ordering of sequential processes*, Acta Informatica, 1 (1971), p. 115–138.

[27] E. W. DIJKSTRA, *Cooperating sequential processes*, The origin of concurrent programming : from semaphores to remote procedure calls, (2002), p. 65–138.

[28] M. DUBOIS ET F. A. BRIGGS, *Performance of synchronized iterative processes in multiprocessor systems*, IEEE Trans. Softw. Eng., 8 (1982), p. 419–431.

[29] B. EINARSSON, R. BOISVERT, F. CHAITIN-CHATELIN, R. COOLS, C. DOUGLAS, K. DRITZ, W. ENRIGHT, W. GROPP, S. HAMMARLING, H. P. LANGTANGEN, R. POZO, S. RUMP, V. SNYDER, E. TRAVIESAS-CASSAN, M. VOUK, W. WALSTER ET B. WICHMANN, *Accuracy and Reliability in Scientific Computing*, Software-Environments-Tools, SIAM, Philadelphia, PA, 2005.

[30] J. A. FISHER ET S. M. FREUDENBERGER, *Predicting conditional branch directions from previous runs of a program*, *in* ASPLOS-V : Proceedings of the fifth international conference on Architectural support for programming languages and operating systems, New York, NY, USA, 1992, ACM, p. 85–95.

[31] M. J. FLYNN, *Some computer organizations and their effectiveness*, IEEE Transactions on Computers, 21 (1972), p. 948–960.

[32] A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK ET V. SUNDERAM, *PVM : Parallel virtual machine : a users' guide and tutorial for networked parallel computing*, MIT Press, Cambridge, MA, USA, 1994.

[33] W. K. GILOI, *Towards a taxonomy of computer architecture based on the machine data type view*, *in* ISCA '83 : Proceedings of the 10th annual international symposium on Computer architecture, New York, NY, USA, 1983, ACM, p. 6–15.

[34] W. K. GILOI, *Parallel supercomputer architectures and their programming models*, Parallel Comput., 20 (1994), p. 1443–1470.

[35] L. M. GOLDSCHLAGER, *A unified approach to models of synchronous parallel machines*, *in* STOC '78 : Proceedings of the tenth annual ACM symposium on Theory of computing, New York, NY, USA, 1978, ACM, p. 89–94.

[36] W. GROPP ET E. L. LUSK, *Reproducible measurements of mpi performance characteristics*, *in* Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, London, UK, 1999, Springer-Verlag, p. 11–18.

[37] D. GROVE ET P. CODDINGTON, *Precise mpi performance measurement using mpibench*, *in* In Proceedings of HPC Asia, 2001.

[38] D. A. GROVE ET P. D. CODDINGTON, *Communication benchmarking and performance modelling of mpi programs on cluster computers*, Parallel and Distributed Processing Symposium, International, 15 (2004), p 249b.

[39] M. GUSAT, D. CRADDOCK, W. DENZEL, T. ENGBERSEN, N. NI, G. PFISTER, W. ROONEY ET J. DUATO, *Congestion control in infiniband networks*, *in* HOTI '05 : Proceedings of the 13th Symposium on High Performance Interconnects, 2005, p. 158–159.

[40] S. D. HAMMOND, G. R. MUDALIGE, J. A. SMITH, S. A. JARVIS, J. A. HERDMAN ET A. VADGAMA, *Warpp : a toolkit for simulating high-performance parallel scientific codes.*, *in* SimuTools, O. Dalle, G. A. Wainer, L. F. Perrone et G. Stea, éds., ICST, 2009, p 19.

[41] J. HANDY, *The cache memory book*, Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[42] T. J. HARRIS, *A survey of pram simulation techniques*, ACM Comput. Surv., 26 (1994), p. 187–206.

[43] M. T. HEATH ET J. E. FINGER, *ParaGraph : A Tool for Visualizing Performance of Parallel Programs*, NCSA, October 9, 1992.

[44] J. HENNESSY ET D. PATTERSON, *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann, 2003.

[45] T. HEYWOOD ET S. RANKA, *A practical hierarchical model of parallel computation*, *in* Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on, 1991, p. 18–25.

[46] J. M. D. HILL, B. MCCOLL, D. C. STEFANESCU, M. W. GOUDREAU, K. LANG, S. B. RAO, T. SUEL, T. TSANTILAS ET R. H. BISSELING, *Bsplib : The bsp programming library*, 1998.

[47] C. A. R. HOARE, *Communicating sequential processes*, Commun. ACM, 26 (1983), p. 100–106.

[48] R. W. HOCKNEY ET C. R. JESSHOPE, *Parallel computers : architecture, programming and algorithms / R.W. Hockney, C.R. Jesshope*, Hilger, Bristol :, 1981.

[49] T. HOEFLER, T. SCHNEIDER ET A. LUMSDAINE, *Multistage Switches are not Crossbars : Effects of Static Routing in High-Performance Networks*, *in* Proceedings of the 2008 IEEE International Conference on Cluster Computing, IEEE Computer Society, Oct. 2008.

[50] S. HORWITZ, T. REPS ET D. BINKLEY, *Interprocedural slicing using dependence graphs*, *in* PLDI '88 : Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, New York, NY, USA, 1988, ACM, p. 35–46.

[51] R. IBBETT, G. CHOCHIA, P. COE, M. COLE, P. HEYWOOD, T. HEYWOOD, R. POOLEY, P. THANISCH ET N. TOPHAM, *Algorithms, architectures and models of computation*, technical report ECS-CSG-22-96, Dept. of Computer Science, Edinburgh University, mars 1996.

[52] INTEL CORPORATION, *Intel Trace Analyzer website*. `http://software.intel.com/en-us/intel-trace-analyzer/`.

[53] K. J. O. J. FERRANTE ET J. WARREN, *The program dependence graph and its use in optimization*, ACM Trans. Program. Lang. Syst., 9 (1987), p. 319–349.

[54] J. POSTEL, *Transmission Control Protocol*. USC/Information Sciences Institute, sept. 1981. RFC 793, `http://www.ietf.org/rfc/rfc793.txt`.

[55] JULIEN BOURGEOIS, *Prédiction de performance statique et semi-statique dans les systèmes répartis hétérogènes.*, Thèse doctorat, Université de Franche-Comté, Janvier 2000.

[56] B. H. H. JUURLINK ET H. A. G. WIJSHOFF, *The e-bsp model : Incorporating general locality and unbalanced communication into the bsp model*, *in* In Proc. Euro-Par'96, 1996, p. 339–347.

[57] A. KAPELINKOV, R. R. MUNTZ ET M. D. ERCEGOVAC, *A modeling methodology for the analysis of concurrent systems and computations*, J. Parallel Distrib. Comput., 6 (1989), p. 568–597.

[58] D. J. KERBYSON, H. J. ALME, A. HOISIE, F. PETRINI ET M. WASSERMAN, H. J.AND GITTINGS, *Predictive performance and scalability modeling of a large-scale application*, *in* Supercomputing '01 : Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), New York, NY, USA, 2001, ACM, p. 37–37.

[59] A. KRALL, *Improving semi-static branch prediction by code replication*, *in* PLDI '94 : Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, New York, NY, USA, 1994, ACM, p. 97–106.

[60] F. LEIGHTON, *Introduction to parallel algorithms and architectures : Arrays, trees, and hypercubes*, Morgan Kaufman Publishers Inc., (1992).

[61] C. E. LEISERSON, *Fat-trees : universal networks for hardware-efficient supercomputing*, IEEE Trans. Comput., 34 (1985), p. 892–901.

[62] ——, *Fat-trees : universal networks for hardware-efficient supercomputing*, IEEE Trans. Comput., 34 (1985), p. 892–901.

[63] D. LUGONES, D. FRANCO ET E. LUQUE, *Dynamic routing balancing on infiniband networks*, The Journal of Computer Science and Technology (JCS&T), 8 (2008), p. 104–110.

[64] P. S. MAGNUSSON, M. CHRISTENSSON, J. ESKILSON, D. FORSGREN, G. HÅLLBERG, J. HÖGBERG, F. LARSSON, A. MOESTEDT ET B. WERNER, *Simics : A full system simulation platform*, Computer, 35 (2002), p. 50–58.

[65] E. MAILLET, *Le traçage logiciel d'applications parallèles : conception et ajustement de qualité*, Thèse doctorat, Institut National Polytechnique de Grenoble, Grenoble, Mars 1992.

[66] V. W. MAK ET S. F. LUNDSTROM, *Predicting performance of parallel computations*, IEEE Trans. Parallel Distrib. Syst., 1 (1990), p. 257–270.

[67] M. MARTINASSO, *Analyse et Modélisation des Communications Concurrentes dans les Réseaux Haute Performance*, Thèse doctorat, Université Joseph Fourier, école doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique, Grenoble, Mai 2007.

[68] W. MCCOLL, *Foundations of time-critical scalable computing*, *in* In Proceedings of the 15th IFIP World Computer Congress. Osterreichische Computer Gesellschaft, 1998, p. 93–107.

[69] J. M. MELLOR-CRUMMEY, V. S. ADVE, B. BROOM, D. G. CHAVARRÍA-MIRANDA, R. J. FOWLER, G. JIN, K. KENNEDY ET Q. YI, *Advanced optimization strategies in the rice dhpf compiler*, Concurrency and Computation : Practice and Experience, 14 (2002), p. 741–767.

[70] MESSAGE PASSING INTERFACE FORUM, *MPI : A message-passing interface standard*, International Journal of Supercomputer Applications, (1994), p. 165–414.

[71] R. MILNER, *A Calculus of Communicating Systems*, vol. 92 de Lecture Notes in Computer Science, Springer-Verlag, 1980.

[72] S. MOORE, D. CRONK, F. WOLF, A. PURKAYASTHA, P. TELLER, R. ARAIZA, M. G. AGUILERA ET J. NAVA, *Performance profiling and analysis of dod applications using papi and tau*, *in* DOD_UGC '05 : Proceedings of the 2005 Users Group Conference on 2005 Users Group Conference, Washington, DC, USA, 2005, IEEE Computer Society, p 394.

[73] P. MUCCI ET K. LONDON, *The mpbench report*, rap. tech., University of Tenesse, 1998.

[74] P. J. MUCCI ET S. V. MOORE, *Papi users group*, *in* SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing, New York, NY, USA, 2006, ACM, p 43.

[75] M. S. MÜLLER, M. VAN WAVEREN, R. LIEBERMANN, B. WHITNEY, H. SAITO, K. KALYAN, J. BARON, B. BRANTLEY, C. PARROTT, T. ELKEN, H. FENG ET C. PONDER, *SPEC MPI2007 - An application benchmark for clusters and hpc systems*, *in* ISC, 2007.

[76] G. PALLAS, *Pallas MPI Benchmarks, Part MPI-1*, Pallas Technical Report, (2000).

[77] PERFCTR, *Perfctr project webpage*. http ://user.it.uu.se/ mikpe/linux/perfctr/.

[78] PERFMON, *Perfmon project webpage*. http ://perfmon2.sourceforge.net/.

[79] J. L. PETERSON, *Petri Net Theory and the Modeling of Systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[80] D. QUINLAN, R. VUDUC, T. PANAS, J. HÄRDTLEIN ET A. SÆBØRNSEN, *Support for whole-program analysis and verification of the one-definition rule in c++*, *in* Static Analysis Summit Gaithersburg, MD, June 2006.

[81] R. W. HOCKNEY, *The Communication Challenge for MPP : Intel Paragon and Meiko CS-2*, *in* Parallel Computing, North-Holland, vol. 20, 1994, p. 389–398.

[82] R. REUSSNER, P. SANDERS ET J. L. TRÄFF, *Skampi : a comprehensive benchmark for public benchmarking of mpi*, Sci. Program., 10 (2002), p. 55–65.

[83] D. RIDGE, D. BECKER, P. MERKEY ET T. STERLING, *Beowulf : Harnessing the power of parallelism in a pile-of-pcs*, *in* Proceedings, IEEE Aerospace, 1997, p. 79–91.

[84] A. W. ROSCOE, C. A. R. HOARE ET R. BIRD, *The Theory and Practice of Concurrency*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[85] ROSE, *Rose project webpage*. http ://www.rosecompiler.org/.

[86] J. RUSSELL, *Program slicing literature survey*. 2001.

[87] S. S. SHENDE AND A. D. MALONY, *The Tau Parallel Performance System*, International Journal of High Performance Computing Applications, 20 (2006), p. 287–311.

[88] R. H. SAAVEDRA ET A. J. SMITH, *Analysis of benchmark characteristics and benchmark performance prediction*, rap. tech., EECS Department, University of California, Berkeley, Dec 1992.

[89] J. C. SANCHO, A. ROBLES ET J. DUATO, *Effective strategy to compute forwarding tables for infiniband networks*, Parallel Processing, International Conference on, (2001), p. 48–57.

[90] V. SARKAR, *Determining average program execution times and their variance*, SIGPLAN Not., 24 (1989), p. 298–312.

[91] M. SCHORDAN ET D. J. QUINLAN, *A source-to-source architecture for user-defined optimizations*, *in* JMLC, 2003, p. 214–223.

[92] M. D. SCHROEDER, A. D. BIRRELL, M. BURROWS, H. MURRAY, R. M. NEEDHAM, T. L. RODEHEFFER, E. H. SATTERTHWAITE ET C. P. THACKER, *Autonet : A high-speed, self-configuring local area network using point-to-point links*, IEEE Journal on Selected Areas in Communications, (1991).

[93] J. T. SCHWARTZ, *Ultracomputers*, ACM Trans. Program. Lang. Syst., 2 (1980), p. 484–521.

[94] H. SHAN, K. ANTYPAS ET J. SHALF, *Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark*, *in* SC

'08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Piscataway, NJ, USA, 2008, IEEE Press, p. 1–12.

[95] G. M. SHIPMAN, T. S. WOODALL, R. L. GRAHAM, A. B. MACCABE ET P. G. BRIDGES, *Infiniband scalability in open mpi*, *in* Proceedings of IEEE Parallel and Distributed Processing Symposium, April 2006.

[96] T. SKEIE, O. LYSNE ET I. THEISS, *Layered shortest path (lash) routing in irregular system area networks*, Parallel and Distributed Processing Symposium, International, 2 (2002), p 0162.

[97] D. B. SKILLICORN, J. M. D. HILL ET W. F. MCCOLL, *Questions and answers about bsp.*, Scientific Programming, 6 (1997), p. 249–274.

[98] Q. O. SNELL, A. R. MIKLER ET J. L. GUSTAFSON, *Netpipe : A network protocol independent performance evaluator*, *in* IASTED International Conference on Intelligent Information Management and Systems, juin 1996.

[99] STAFF, *Using mpi-portable parallel programming with the message-passing interface, by william gropp*, Sci. Program., 5 (1996), p. 275–276.

[100] STANDARD PERFORMANCE EVALUATION CORPORATION, *SPEC MPI2007 Benchmark suite*. http ://www.spec.org/mpi2007/.

[101] M. R. STEED ET M. J. CLEMENT, *Performance prediction of pvm programs*, *in* IPPS '96 : Proceedings of the 10th International Parallel Processing Symposium, Washington, DC, USA, 1996, IEEE Computer Society, p. 803–807.

[102] S. SUR, M. J. KOOP ET D. K. PANDA, *High-performance and scalable mpi over infiniband with reduced memory usage : an in-depth performance analysis*, *in* SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing, New York, NY, USA, 2006, ACM, p 105.

[103] Z. SZEBENYI, B. J. WYLIE ET F. WOLF, *Scalasca parallel performance analyses of PEPC*, *in* Proc. of the 1st Workshop on Productivity and Performance (PROPER) in conjunction with Euro-Par 2008, vol. 5415 de Lecture Notes in Computer Science, Springer, 2009, p. 305–314.

[104] F. TIP, *A survey of program slicing techniques*, Journal of Programming Languages, 3 (1995), p. 121–189.

[105] P. D. L. TORRE ET C. P. KRUSKAL, *Submachine locality in the bulk synchronous setting (extended abstract)*, *in* Euro-Par '96 : Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II, London, UK, 1996, Springer-Verlag, p. 352–358.

[106] R. A. TOWLE, *Control and data dependence for program transformations.*, Thèse doctorat, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1976.

[107] D. TOWSLEY, C. ROMMEL ET J. STANKOVIC, *Analysis of fork-join program response times on multiprocessors*, IEEE Transactions on Parallel and Distributed Systems, 1 (1990), p. 286–303.

[108] T.-F. TSUEI ET M. K. VERNON, *Diagnosing parallel program speedup limitations using resource contention models.*, *in* ICPP (1), B. W. Wah, éd., Pennsylvania State University Press, 1990, p. 185–189.

[109] A. M. TURING, *On computable numbers, with an application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, 2 (1936), p. 230–265.

[110] J. UNIEJEWSKI, *Spec benchmark suite : designed for today's advanced systems*, Rap. tech. 1, SPEC Newsletter, 1989.

[111] L. G. VALIANT, *A bridging model for parallel computation*, Commun. ACM, 33 (1990), p. 103–111.

[112] P. VELHO ET A. LEGRAND, *Accuracy study and improvement of network simulation in the simgrid framework*, *in* SIMUTools'09, 2nd International Conference on Simulation Tools and Techniques, 2009.

[113] J. VIENNE ET MARTINASSO, *Evaluation et modélisation des communications concurrentes*, *in* AEP09, Aussois, France, jun 2008.

[114] J. VIENNE, M. MARTINASSO, J.-M. VINCENT ET J.-F. MÉHAUT, *Predictive models for bandwidth sharing in high performance clusters*, *in* Cluster 2008, Tsukuba, Japan, sep 2008.

[115] A. VISHNU, A. R. MAMIDALA, H.-W. JIN ET D. K. PANDA, *Performance modeling of subnet management on fat tree infiniband networks using opensm*, Parallel and Distributed Processing Symposium, International, 19 (2005), p 296b.

[116] D. F. VRSALOVIC, D. P. SIEWIOREK, Z. Z. SEGALL ET E. F. GEHRINGER, *Performance prediction and calibration for a class of multiprocessors*, IEEE Trans. Comput., 37 (1988), p. 1353–1365.

[117] V.S. ADVE, *Analyzing the Behavior and Performance of Parallel Programs*, Thèse doctorat, University of Wisconsin, Computer Sciences Department, 1993.

[118] W. GROPP AND E. LUSK, *User's Guide for MPE : Extensions for MPI Programs*. `http://www.mcs.anl.gov/mpi/mpich1/docs/mpeman.pdf`.

[119] H. WABNIG ET G. HARING, *Paps—a testbed for performance prediction of parallel applications*, Parallel Comput., 22 (1997), p. 1837–1851.

[120] M. WARREN, D. J. BECKER, M. P. GODA, J. K. SALMON ET T. STERLING, *Parallel supercomputing with commodity components*, *in* In International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA, 1997, p. 1372–1381.

[121] M. WEISER, *Program slicing*, IEEE Trans. Software Eng., 10 (1984), p. 352–357.

[122] S. A. WILLIAMS, *Programming models for parallel systems*, John Wiley & Sons, Inc., New York, NY, USA, 1990.

[123] F. WOLF, B. J. WYLIE, E. ÁBRAHÁM, D. BECKER, W. FRINGS, K. FÜRLINGER, M. GEIMER, M.-A. HERMANNS, B. MOHR, S. MOORE, M. PFEIFER ET Z. SZEBENYI, *Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications*, *in* Proc. of the 2nd HLRS Parallel Tools Workshop, Springer, July 2008, p. 157–167.

[124] C. YOUNG ET M. D. SMITH, *Improving the accuracy of static branch prediction using branch correlation*, *in* ASPLOS-VI : Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, New York, NY, USA, 1994, ACM, p. 232–241.

[125] ——, *Static correlated branch prediction*, ACM Trans. Program. Lang. Syst., 21 (1999), p. 1028–1075.

[126] D. ZAPARANUKS, M. JOVIC ET M. HAUSWIRTH, *Accuracy of performance counter measurements*, *in* ISPASS, 2009, p. 23–32.

# Résumé

Prédiction de performances d'application de calcul haute performance
sur réseau Infiniband.

Afin de pouvoir répondre au mieux aux différents appels d'offres, les constructeurs de grappe de calcul ont besoin d'outils et de méthodes permettant d'aider au mieux la prise de décisions en terme de design architectural. Nos travaux se sont donc intéressés à l'estimation des temps de calcul et à l'étude de la congestion sur le réseau *InfiniBand*. Ces deux problèmes sont souvent abordés de manière globale. Néanmoins, une approche globale ne permet pas de comprendre les raisons des pertes de performance liées aux choix architecturaux. Notre approche s'est donc orientée vers une étude plus fine.
Pour évaluer les temps de calcul, la démarche proposée s'appuie sur une analyse statique ou semi-statique du code source afin de le découper en blocs, avant d'effectuer un micro-benchmarking de ces blocs sur l'architecture cible. Pour l'estimation des temps de communication, un modèle de répartition de bande passante pour le réseau *InfiniBand* a été développé, permettant ainsi de prédire l'impact lié aux communications concurrentes. Ce modèle a ensuite été intégré dans un simulateur pour être validé sur un ensemble de graphes de communication synthétiques et sur l'application Socorro.


**Mots-clés :** Évaluation de performance, réseaux haute performance, analyse statique, communications concurrentes, congestion réseau, *MPI*, *InfiniBand*, modélisation et simulation.


---


# Abstract

High performance computing application prediction using Infiniband network

Manufacturers of computer clusters require tools to assist them in making better decisions in terms of architectural design. To address this need, in this thesis work, we focus on the specific issues of estimating computation times and InfiniBand network congestion. These two problems are often dealt with globally. However, an overall approach does not explain the reasons of performance loss related to architectural choices. So our approach was to conduct a more detailed study.

In this thesis work, we focus on the following : 1) the estimation of computation time in a Grid, and 2) the estimation of communication times over Infiniband networks. To evaluate the computation time, the proposed approach is based on a static or semi-static analysis of the source code, by cutting it into blocks, before making a micro-benchmarking of these blocks on the targeted architecture. To estimate the communication time, a model of bandwidth sharing for Infiniband networks has been developed, allowing one to predict the impact related to concurrent communications. This model was then incorporated into a simulator to be validated on a set of synthetic communication graphs and on the application Socorro.


**Keywords :** Concurrent communications, network contention, *MPI*, *InfiniBand*, high performance networks, static analysis, modelling and simulation, performance evaluation.