N° d'ordre : 3238

Université Bordeaux 1
Laboratoire Bordelais de Recherche en Informatique
CEA/DAM Île de France

THÈSE

pour obtenir le grade de

Docteur
Spécialité : Informatique

au titre de l'école doctorale de Mathématiques et Informatique

présentée et soutenue publiquement le 31 octobre 2006

par Monsieur Marc PÉRACHE

Contribution à l'élaboration d'environnements de programmation dédiés au calcul scientifique hautes performances

Directeur de thèse :	Monsieur	Raymond	NAMYST	
Encadrement CEA :	Monsieur	Hervé	JOURDREN	
Après avis de :	Monsieur	William	JALBY,	Rapporteur
	Monsieur	Jean-François	MÉHAUT,	Rapporteur

Devant la commission d'examen formée de :

Monsieur	William	JALBY,	Rapporteur
Monsieur	Hervé	JOURDREN	
Monsieur	Jean-François	MÉHAUT,	Rapporteur
Monsieur	Raymond	NAMYST	
Monsieur	Jean	ROMAN,	Président/Rapporteur de soutenance

Remerciements

Je remercie tout d'abord mon encadrant au CEA Hervé Jourdren et mon directeur de thèse Raymond Namyst pour avoir accepté de superviser et guider mon travail pendant ces trois années de thèse.

Je tiens à remercier tous les membres du CEA et en particulier Pierre Leca pour m'avoir accueilli durant ces trois ans. Je remercie plus particulièrement David Dureau pour son encadrement lors de mes stages de maîtrise et DEA ainsi que sa disponibilité durant la thèse en particulier pour la phase d'évaluation de MPC. Je remercie aussi Dominique Rodrigues d'avoir accepté de modifier son application pour compléter les évaluations de MPC. Enfin, je remercie Pascal Havé qui a été un bêta-testeur patient et compréhensif.

Je tiens également à remercier les membres de l'équipe Runtime pour leur accueil chaleureux lors de mes différentes visites à Bordeaux ainsi que leurs conseils avisés lors, entre autres, de la rédaction de la thèse.

Je remercie toutes les personnes avec qui j'ai eu l'occasion de collaborer lors des différents projets auxquels j'ai participé. Je remercie plus particulièrement, les équipes de Bull et du laboratoire Prism pour leur collaboration, leur accueil et leur disponibilité.

Je suis tout particulièrement reconnaissant à William Jalby et Jean-François Méhaut d'avoir acceptés d'être rapporteurs de ma thèse. Je remercie également tous les membres de mon jury de thèse, William Jalby, Hervé Jourdren, Jean-François Méhaut, Raymond Namyst et Jean Roman.

Je dis aussi un grand merci à ma famille qui s'est fortement impliquée dans les aspects, pas toujours agréables, de correction des fautes d'orthographe et de logistique du pot de thèse.

Merci enfin à ma femme, Séverine, pour son soutien inconditionnel, son aide et sa compréhension lors des différentes étapes de ces trois années et en particulier lors de la rédaction.

Table des matières

1 Introduction **1**
- 1.1 Motivations du travail . . . 1
- 1.2 Objectif de la thèse et contribution . . . 2
- 1.3 Organisation du document . . . 2

I Environnement de programmation pour le calcul scientifique **5**

2 Présentation et évolution de l'architecture des supercalculateurs **7**
- 2.1 Terminologie des supercalculateurs . . . 7
- 2.2 Historique : de l'Analytical Engine à BlueGene/L . . . 8
- 2.3 Tendance : toujours plus de "processeurs" . . . 9
  - 2.3.1 La fin de la course à la puissance séquentielle . . . 10
  - 2.3.2 Architectures des processeurs . . . 12
  - 2.3.3 Les architectures ccNUMA . . . 16
  - 2.3.4 Discussion . . . 17
- 2.4 Les supercalculateurs d'aujourd'hui . . . 18
  - 2.4.1 Une architecture massivement parallèle : l'IBM BlueGene/L . . . 18
  - 2.4.2 Une architecture mémoire partagée : SGI Altix 3700 Bx2 . . . 19
  - 2.4.3 Une architecture hybride de type grappe de NUMA : TERA10 . . . 20
- 2.5 Mise en évidence d'une tendance pour les supercalculateurs de demain . . . 21

3 Applications visées en calcul scientifique parallèle **23**
- 3.1 Exemple de maillage et de structures de données en mécanique des fluides . . . 23
  - 3.1.1 Maillages eulériens structurés . . . 23
  - 3.1.2 Maillages lagrangiens non-structurés . . . 24
  - 3.1.3 Maillages AMR (Adaptive Mesh Refinement) . . . 25
- 3.2 L'approche Single Process Multiple Data (SPMD) par décomposition de domaines . . . 25

4 Panorama des modèles de programmation parallèle et outils existants **27**
- 4.1 Définitions . . . 27
- 4.2 Modèle de programmation à mémoire distribuée . . . 28
  - 4.2.1 Des communications explicites . . . 28
  - 4.2.2 Outils . . . 29
  - 4.2.3 Discussion . . . 30
- 4.3 Modèle de programmation à mémoire partagée . . . 31

4.3.1 Des communications transparentes . . . 31
4.3.2 Outils . . . 32
4.3.3 Discussion . . . 35
4.4 Modèle de programmation mixte . . . 36
4.4.1 Tirer le meilleur parti de l'architecture sous-jacente . . . 37
4.4.2 Outils . . . 37
4.4.3 Discussion . . . 38
4.5 Discussion . . . 38

5 Mise en évidence de la complexité de programmation des architectures actuelles 41
5.1 Description de la dépendance des optimisations logicielles vis-à-vis de l'architecture 41
5.1.1 Architectures NUMA . . . 41
5.1.2 Puces multicœurs et/ou multithreads . . . 44
5.1.3 Parallélisme massif . . . 45
5.1.4 Discussion . . . 45
5.2 Mise en évidence de la dépendance des outils logiciels existants à l'architecture . . . 46
5.2.1 Bibliothèque de communication . . . 46
5.2.2 Bibliothèque de threads . . . 47
5.3 Difficulté d'intégration des composants dans les approches hybrides . . . 47
5.4 Une contrainte de robustesse . . . 47

II La bibliothèque MPC : MultiProcessor Communications 49

6 Choix d'un modèle de programmation pour MPC 51
6.1 Mise en évidence des caractéristiques d'une bibliothèque de communication . . . 51
6.1.1 Haute portabilité : du monoprocesseur à la grappe de multiprocesseur . . . 51
6.1.2 Équilibrage de charge . . . 52
6.1.3 Hautes performances . . . 52
6.1.4 Tolérance aux pannes . . . 53
6.2 Notre choix : la programmation par passage de messages . . . 53
6.2.1 Motivation . . . 53
6.2.2 Et les autres modèles ? . . . 54
6.3 Interface de programmation choisie . . . 55

7 Modèles d'exécution de MPC 57
7.1 Choix du modèle de gestion des tâches : le multithreading . . . 57
7.1.1 Programmation par passage de messages et multithreading . . . 57
7.1.2 Le multithreading mixte : un contrôle total . . . 59
7.1.3 Stratégies de programmation et d'exécution . . . 62
7.2 Choix du modèle de gestion mémoire . . . 64
7.2.1 Approche choisie . . . 65
7.2.2 Migration interprocessus . . . 66
7.3 Choix du modèle de gestion des communications . . . 67
7.3.1 Communications collectives . . . 67
7.3.2 Communication point à point . . . 72

7.4 Choix du modèle de gestion de la tolérance aux pannes . . . 74
7.4.1 Une approche coordonnée . . . 74
7.4.2 Mécanisme . . . 74
7.5 Mise en place du modèle d'équilibrage de charge . . . 75

III Implantation et évaluation **77**

8 Éléments d'implémentation de MPC sur architectures hiérarchiques **79**
8.1 Gestion des tâches . . . 79
8.1.1 Bibliothèque de threads . . . 79
8.1.2 Placement . . . 80
8.1.3 Scrutation . . . 81
8.2 Gestion mémoire . . . 81
8.2.1 Méthode d'allocation . . . 81
8.2.2 Localité des données . . . 82
8.2.3 Limitations . . . 83
8.3 Les communications . . . 83
8.3.1 Communications intranœuds . . . 83
8.3.2 Communications internœuds . . . 84
8.4 Migration de threads interprocessus . . . 84
8.5 Tolérance aux pannes . . . 86
8.5.1 Réalisation de l'image d'une tâche . . . 86
8.5.2 Rétablissement d'une image . . . 86
8.5.3 Redimensionnement . . . 86

9 Évaluation des performances de MPC sur des cas représentatifs **89**
9.1 Langages et architectures supportées . . . 89
9.2 Micro évaluation . . . 90
9.2.1 Les machines de test . . . 90
9.2.2 Gestion mémoire . . . 90
9.2.3 Communications collectives . . . 92
9.2.4 Communications point à point . . . 93
9.3 Applications représentatives . . . 93
9.3.1 Jacobi . . . 95
9.3.2 Advection . . . 97
9.3.3 Conduction . . . 97
9.4 Applications de production . . . 99
9.4.1 Code Fortran de sismique : PRODIF . . . 100
9.4.2 Code C# d'advection/striation . . . 101
9.4.3 Code C++ AMR d'hydrodynamique : HERA . . . 101
9.4.4 Évaluation à grande échelle en calcul teraflopique : le code infrason Dragst-R 102

10 Conclusions et perspectives **105**
10.1 Conclusions . . . 105
10.2 Perspectives . . . 106

Bibliographie **109**

A MPC **115**
A.1 Interface MPC 115
A.2 Exemples de migration de MPI vers MPC 122
A.2.1 Exemple de programme C 122
A.2.2 Exemple de programme Fortran 123

B Présentation détaillée des schémas **125**
B.1 Advection 125
B.2 Conduction 126

Table des figures

2.1 Évolution de la fréquence maximale des processeurs . . . 11
2.2 Exemples d'architectures multicœur . . . 13
2.3 Architecture ccNUMA . . . 17
2.4 BlueGene/L . . . 19
2.5 Schéma d'une C-Brick . . . 20
2.6 Schéma d'interconnexion des R-Bricks . . . 21
2.7 Architecture Bull NovaScale . . . 22

3.1 Exemple de maillage eulérien structuré . . . 24
3.2 Exemple de maillage lagrangien non-structuré . . . 24
3.3 Exemple d'évolution de maillage AMR . . . 25
3.4 Exemple de parallélisation SPMD . . . 26

4.1 Bibliothèque de niveau utilisateur . . . 33
4.2 Bibliothèque de niveau noyau . . . 34
4.3 Bibliothèque à deux niveaux . . . 34

7.1 Illustration du modèle d'implémentation . . . 58
7.2 Illustration de l'ordonnanceur de MPC . . . 60
7.3 Illustration du modèle de gestion mémoire . . . 65
7.4 Sémantique des opérations collectives . . . 68
7.5 Exemple d'opération collective . . . 71
7.6 Illustration des communications point à point sur un anneau (envoi/reception) . . . 73

8.1 Illustration du modèle de gestion mémoire . . . 82
8.2 Migration de threads . . . 85

9.1 Comparaison des allocations mémoires . . . 91
9.2 Comparaison des barrières en threads . . . 92
9.3 Comparaison des latences en mémoire partagée . . . 94
9.4 Comparaison des débits en mémoire partagée . . . 95
9.5 Comparaison des temps d'exécution du cas test Jacobi . . . 96
9.6 Comparaison des temps d'exécution du cas test Advection . . . 98
9.7 Comparaison des temps d'exécution du cas test Conduction . . . 99
9.8 Tests d'extensibilité du code Fortran de sismique : PRODIF (4.7 millions de mailles) 100
9.9 Cas AMR figé mono matériau . . . 102

Chapitre 1

Introduction

Ce document présente les travaux que j'ai réalisés durant ma thèse au CEA/DAM Île de France sous la direction conjointe de Raymond NAMYST et Hervé JOURDREN.

1.1 Motivations du travail

Dans le cadre du calcul scientifique parallèle et plus particulièrement du calcul hautes performances, il est important d'utiliser au mieux les architectures matérielles à notre disposition. En effet, une grande puissance de calcul est nécessaire pour simuler des phénomènes physiques complexes. D'ailleurs, on constate souvent une sous-exploitation de ces supercalculateurs dédiés à la simulation. En effet, les optimisations de code, spécifiques à chaque supercalculateur, sont souvent trop complexes à mettre en œuvre sur des gros codes de simulation. C'est pourquoi, les performances, en nombre d'opérations par seconde lors des simulations, sont généralement bien en deçà des performances crêtes théoriques. Cette sous-exploitation conduit généralement à une incapacité de simuler certains phénomènes alors que la puissance théorique le permet. Il est donc crucial de concevoir des environnements de programmation parallèle efficaces pour accroître la complexité des phénomènes simulés sans pour autant recourir à un renouvellement de matériel.

Les performances d'un code de simulation parallèle sont fortement liées à l'environnement de programmation en charge des communications. Les environnements de programmation par passage de messages implémentant le standard MPI sont, à l'heure actuelle, les plus utilisés en calcul scientifique parallèle. Les implémentations de ce standard qui apportent les meilleures performances aux codes de simulation sont généralement les implémentations dites "constructeur". Ces implémentations sont conçues et optimisées par le constructeur du supercalculateur en tirant parti des spécificités de l'architecture sous-jacente. Malheureusement, ces approches sont majoritairement optimisées pour les applications ayant une répartition équilibrée de la charge sur les processeurs. Les quelques implémentations du standard MPI adaptées aux codes déséquilibrés (TOMPI, AMPI,...) ne permettent malheureusement pas d'égaler les performances des meilleures implémentations pour codes équilibrés. Les environnements de programmation n'offrent donc pas suffisamment de flexibilité pour offrir de bonnes performances à tous les codes.

Un autre facteur important dans la conception de codes de calcul performants réside dans la bonne utilisation des ressources de la machine et en particulier du processeur. Les environnements de programmation comme les implémentations MPI n'apportent généralement aucune aide à l'utilisateur sur ce point. Or, on peut montrer aisément que la gestion des mémoires cache a

un fort impact sur les performances globales du code de simulation. Cette gestion est entre autre très liée à l'ordonnancement des tâches parallèles qui dépend généralement du support exécutif et donc en parti de la bibliothèque de communication. Néanmoins à l'heure actuelle, l'amélioration des performances des codes de simulation doit être faite par le concepteur sans l'aide du support exécutif utilisé pour la parallélisation.

Force est de constater que les implémentations actuelles des bibliothèques de communication n'offrent pas toutes les fonctionnalités nécessaires à la construction aisée de codes de calcul parallèle performants.

1.2 Objectif de la thèse et contribution

Il s'agit dans un premier temps, de faire le point sur les raisons qui font qu'un grand nombre de codes de calcul ne sont pas suffisamment performants. Il est donc nécessaire de comprendre les interactions entre l'architecture des supercalculateurs, les grandes classes de solveurs utilisés en calcul hautes performances et les outils communément utilisés pour la parallélisation. Les outils utilisés actuellement ne permettent pas aux codes de calcul d'utiliser au mieux les architectures matérielles ; il faut donc identifier précisément ces manques.

Cela nous conduira à proposer un certain nombre d'outils et de méthodes dans le but d'enrichir les environnements de programmation parallèle. Plusieurs axes seront explorés en détail. On peut citer :

- les méthodes et outils facilitant une conception de codes de calcul déséquilibrés performants ;
- les outils favorisant une bonne utilisation des ressources matérielles disponibles ;
- les méthodes améliorant les techniques de communication.

Tous ces aspects ont été intégrés au sein d'une bibliothèque de communication complètement opérationnelle que nous avons utilisé pour les évaluations présentées dans ce document.

La contribution majeure de ma thèse consiste donc en la conception d'une approche originale de parallélisation de codes de calcul scientifique. Ces travaux ont abouti à la création d'un environnement de parallélisation utilisant des communications par passage de messages. Cet environnement est nommé MPC : MultiProcessor Communications. MPC intègre de nombreuses méthodes et approches élaborées durant la thèse mais aussi les fonctionnalités classiques des environnements de communication. Ces travaux ont donné lieu à de nombreux séminaires et à une publication.

1.3 Organisation du document

Ce document s'organise en trois parties. La première partie présente le contexte de notre étude sous la forme de trois chapitres. Le premier chapitre apporte une présentation, une analyse et la détermination d'une tendance concernant l'architecture des supercalculateurs. Le second chapitre présente le monde du calcul scientifique parallèle à travers une présentation d'applications visées. Le troisième dresse un panorama des modèles de programmation et outils existants.

La deuxième partie du document décrit les choix architecturaux relatifs à MPC, notre bibliothèque de communication. Cette partie se subdivise en deux chapitres. Le premier présente et justifie le choix du modèle de programmation. Le second présente la mise en place des modèles d'exécution.

La troisième partie étudie l'implémentation et l'évaluation de MPC. Cette partie est composée de deux chapitres. Le premier chapitre présente des éléments d'implémentation des modèles d'exécution. Le second évalue les performances de MPC.

Dans le dernier chapitre du document, nous tirons les conclusions de notre étude et proposons quelques pistes de recherches qui s'inscrivent dans la perspective des nôtres.

Première partie

Environnement de programmation pour le calcul scientifique

Chapitre 2

Présentation et évolution de l'architecture des supercalculateurs

L'objectif de ce chapitre est de présenter les particularités techniques et les spécificités architecturales de différents types de supercalculateurs. Cette présentation permettra de déterminer les principales caractéristiques des supercalculateurs. Elle mettra aussi en évidence la tendance d'évolution des supercalculateurs. La caractérisation des supercalculateurs est cruciale pour la conception d'environnement de programmation dédiés au calcul scientifique. En effet, pour obtenir de bonnes performances, les applications doivent tirer parti de l'architecture sous-jacente.

Ce chapitre se décompose en cinq parties. La première partie va mettre en place l'ensemble des définitions relatives aux supercalculateurs. Il s'en suivra un historique de l'évolution des supercalculateurs. Forts de ces informations, nous détaillerons les différentes limitations physiques qui régissent les architectures actuelles. Ensuite, nous présenterons trois exemples représentatifs des architectures de supercalculateurs actuels. Enfin, nous dégagerons une tendance d'évolution des supercalculateurs pour les années à venir.

2.1 Terminologie des supercalculateurs

Dans cette section, nous allons définir l'ensemble des notions en relation avec les architectures des supercalculateurs. Ces notions seront utilisées dans la suite du document. Nous allons commencer par la brique de base de la majorité des calculateurs : l'architecture SMP.

Définition 2.1: ***Symetrical MultiProcessing (SMP)*** **–** Un système SMP est constitué de plusieurs processeurs identiques connectés à une *unique* mémoire physique.

Les architectures de type SMP permettent donc à un nombre relativement restreint de processeurs (en général 2 à 4 processeurs) de partager une mémoire commune. L'architecture NUMA est une extension de l'architecture SMP. Elle permet à un nombre plus élevé de processeurs de communiquer par mémoire partagée.

Définition 2.2: ***Non Uniform Memory Access (NUMA)*** **–** Un système NUMA est constitué de plusieurs processeurs connectés à *plusieurs* mémoires distinctes reliées entre elles par des mécanismes matériels.

Les architectures NUMA et en particulier l'architecture ccNUMA seront détaillées en 2.3.3. La différence entre une architecture NUMA et une architecture SMP concerne la structure de la mémoire. Dans un cas elle est unique, dans l'autre il y a plusieurs mémoires physiques. Les architectures NUMA introduisent donc la notion de localité du banc mémoire par rapport à un processeur donné. Suivant le placement des données sur les bancs mémoire, le temps d'accès, pour un processeur donné, va varier d'un facteur appelé le facteur NUMA. Dans le domaine du calcul hautes performances les architectures NUMA comme SMP sont très souvent appelées nœuds de calcul car ce sont les briques de base des supercalculateurs.

Définition 2.3: ***Nœud*** **–** Un nœud est constitué d'un ou plusieurs processeurs et d'une mémoire associée. C'est le *plus grand* ensemble de processeurs partageant matériellement de la mémoire.

La puissance et le nombre de processeurs d'un nœud n'étant pas toujours suffisants, de nombreux supercalculateurs sont de type grappe.

Définition 2.4: ***Grappe*** **–** Une grappe (ou *cluster*) est un ensemble de nœuds interconnectés par un réseau.

Il existe une distinction entre les grappes suivant la manière dont la mémoire est gérée. Les grappes peuvent être de type mémoire distribuée ou mémoire partagée.

Définition 2.5: ***Système à mémoire distribuée*** **–** Un système à mémoire distribuée met en jeu plusieurs ressources de calcul qui n'ont pas de mémoire partagée, que ce soit de manière physique ou logicielle.

Définition 2.6: ***Système à mémoire partagée*** **–** Un système à mémoire partagée met en jeu plusieurs ressources de calcul qui ont de la mémoire partagée physiquement ou de manière logicielle.

2.2 Historique : de l'Analytical Engine à BlueGene/L

La préhistoire des supercalculateurs se confond plus ou moins avec celle de l'ordinateur. Le concept de calculatrice programmable a tout d'abord germé dans l'esprit du mathématicien londonien Charles Babbage (1791-1871). Son *Analytical Engine*, qu'il conçoit à partir de 1834, est bien entendu mécanique et comporte un "mill" qui est l'unité centrale, un "store" qui est la mémoire, un lecteur de cartes perforées et une imprimante pour les sorties. Néanmoins, ce tout premier ordinateur ne sera jamais réellement terminé.

Il faudra encore attendre cent ans pour voir apparaître le premier ordinateur binaire, le *Z1*, qui fut réalisé par Konrad Zuse en 1938. Il est strictement mécanique. Le programme est stocké sur bande perforée et il dispose d'une mémoire de 64 nombres binaires sur 22 bits représentés en virgule flottante. Ce n'est qu'en 1941 que Konrad Zuse réalisera une version électromécanique de son ordinateur (le Z3). Pendant l'été 1942, John Atanasoff et Clifford Berry présentent le premier calculateur électronique, l'*ABC*. Il comporte 311 tubes à vide et fonctionne à une fréquence de 60Hz. Il réalise une instruction à la seconde mais n'est pas programmable.

Si de nombreux calculateurs spécialisés dans le cryptage/décryptage furent construits durant la seconde guerre mondiale ; ce n'est qu'en 1955 qu'apparaît l'*IBM 704*, le premier ordinateur commercial scientifique à virgule flottante. Il dispose d'une puissance de 5 kFLOPS. On considère

souvent que cette machine marque le début de l'ère des superordinateurs dédiés au calcul scientifique. L'*IBM 704* utilisait une mémoire à tores de ferrite de 32768 mots de 36 bits. C'est sur cette machine que sera développé le langage Fortran.

La première révolution dans le calcul scientifique survient en 1958 avec le lancement du premier ordinateur commercial entièrement transistorisé, le *CDC 1604*, développé par Seymour Cray. La machine la plus puissante manipule le nombre flottant sur 48 bits. Le *CDC 1604* sera suivi en 1965 par le *CDC 6600*. La machine est rythmée par une horloge à 10MHz, sa mémoire est de 256 000 mots de 60 bits. Le *CDC 6600* restera durant cinq ans la machine la plus puissante au monde. Seymour Cray va alors imposer la suprématie de son architecture vectorielle[1] avec le *Cray 1* qui restera longtemps LE supercalculateur.

La seconde révolution dans le calcul scientifique survient vers la fin des années 80 de la réflexion suivante : "on ne pourra bientôt plus poursuivre la course à la puissance à la manière de Seymour Cray". Il faut donc passer au parallélisme. De nombreuses machines parallèles apparaissent comme la *Connection Machine* et ses 65 536 processeurs. Entre ce parallélisme massif et une architecture séquentielle, il y a une ligne médiane qui se dessine avec une utilisation des microprocesseurs qui offrent maintenant un rapport performance/prix imbattable. Dans les années 90, les processeurs vectoriels reculent et le parallélisme se banalise ainsi que l'utilisation des microprocesseurs. On observe ainsi deux architectures de supercalculateurs parallèles : la première permet à un nombre raisonnable de processeurs de communiquer par mémoire partagée ; la deuxième utilise un grand nombre de processeurs à mémoire locale mais sans mémoire partagée.

Ces dernières années ont vu se multiplier des solutions mixtes réunissant, via un réseau d'interconnexion rapide, des nœuds de calcul constitués de plusieurs processeurs communiquant par mémoire partagée. Ces architectures sont connues sous le nom de "grappe de SMP".

2.3 Tendance : toujours plus de "processeurs"

La tendance d'évolution des supercalculateurs se situe au niveau des briques de base qui les composent : les microprocesseurs et les nœuds de calcul. Aujourd'hui, nous sommes à un tournant en matière de calcul hautes performances. Cette évolution, ou plutôt révolution vient, en partie des microprocesseurs eux-mêmes. En effet, comme ce fut le cas dans les années 80 pour les supercalculateurs, l'évolution des microprocesseurs constituant les supercalculateurs ne peut plus se poursuivre sur un modèle séquentiel. On voit donc apparaître de plus en plus de parallélisme au sein même du microprocesseur. On peut comparer cette évolution des microprocesseurs à la seconde révolution des supercalculateurs. Nous allons détailler cette évolution en mettant en évidence les raisons qui poussent les fondeurs de microprocesseurs à changer d'approche. Ensuite, nous décrirons les nouvelles architectures de microprocesseurs issues de l'insertion du parallélisme dans les microprocesseurs. Enfin, nous détaillerons l'architecture NUMA, utilisée comme nœud de calcul dans un grand nombre de supercalculateurs construits autour du microprocesseur. Cette architecture permet d'étendre le nombre de microprocesseurs communiquant par mémoire partagée. L'évolution des processeurs et l'architecture NUMA illustrent bien la tendance actuelle à l'augmentation du nombre de "processeurs" communiquant par mémoire partagée.

[1] Sur les architectures vectorielles, une instruction va s'appliquer à un ensemble de données (un vecteur).

2.3.1 La fin de la course à la puissance séquentielle

Jusqu'à aujourd'hui, les performances des microprocesseurs ont continué d'augmenter de manière exponentielle au cours des années pour deux raisons principales. La première est que les transistors, qui sont le cœur des microprocesseurs, sont devenus de plus en plus petits au cours du temps en suivant la loi de Moore[48]. Cette diminution affecte directement, en terme de fréquence, les performances des processeurs construits autour de ces transistors. De plus, les concepteurs de processeurs ont été capables, grâce au nombre croissant de transistors intégrables dans un processeur de même surface, d'extraire du parallélisme des applications séquentielles grâce, notamment, à des méthodes comme l'ILP[2]. L'augmentation des performances de ces processeurs construits sur le modèle de von Neumann[3] amorce un fort ralentissement ces dernières années. En effet, si dans les années 90, la puissance augmentait de 60% par an, elle n'augmentait plus que de 40% en 2000 pour atteindre 20% en 2004[27]. On constate donc que les méthodes utilisées jusqu'alors ont atteint leurs limites.

Toute l'évolution des processeurs a été basée sur la loi de Moore. La loi de Moore définit le nombre de transistors que l'on peut intégrer sur une puce de taille constante. Cette loi permet donc de déterminer l'évolution de la finesse de gravure. En effet, la diminution de la finesse de gravure permet d'augmenter le nombre de transistors mais aussi d'augmenter la fréquence de ces transistors car la taille d'un transistor induit la fréquence maximale à laquelle il peut changer d'état. L'augmentation de la fréquence, pour permettre celle des performances, a atteint ses limites. La figure FIG. 2.1 décrit l'évolution des fréquences maximales de processeurs au cours du temps. On constate que les fondeurs de processeurs ont changé d'approche et ne font plus évoluer la fréquence comme par le passé. En effet, ils se heurtent à des phénomènes physiques limitants comme le temps d'interconnexion[4] qui ne suit pas la finesse de gravure[25]. Par exemple, si les trois dimensions d'un fil d'interconnexion sont diminuées du même facteur, le temps d'interconnexion reste, lui, quasi constant. C'est l'une des raisons du ralentissement de la montée en fréquence.

La deuxième raison, de la limitation de la fréquence des processeurs, vient de deux phénomènes physiques très liés : la consommation électrique et la dissipation thermique. Ces dernières sont proportionnelles au produit du nombre de transistors par la fréquence d'horloge. En effet, un transistor consomme et dissipe à chaque changement d'état. Donc, plus il y a de transistors, plus il y a de consommation électrique et de dissipation thermique. Ce phénomène est amplifié par la fréquence d'horloge qui va augmenter la fréquence des changements d'état et donc la consommation et la dissipation. La consommation électrique et la dissipation thermique ont aujourd'hui atteint des seuils critiques avec certains processeurs ayant une consommation de plus de 100W et nécessitant d'imposants systèmes de refroidissement. On ne peut donc plus augmenter la fréquence des processeurs avec la conception actuelle des transistors.

Le troisième phénomène freinant la montée en fréquence est d'ordre plus général. Comme on peut le voir dans [27, 50], la montée en fréquence n'est pas synonyme d'augmentation des performances pour toutes les applications. En effet, la différence entre la fréquence du processeur et celle de la mémoire est telle que tout accès mémoire devient extrêmement coûteux c'est-à-dire que beaucoup de cycles sont perdus à attendre que des données en mémoire soient mises dans les registres. Si les fondeurs de processeurs sont conscients du problème et ajoutent de plus en plus

[2]Instruction Level Parallélism : extraction du parallélisme au niveau des instructions à partir d'un flot séquentiel.
[3]Un processeur exécute un flot séquentiel d'instruction et est connecté à une mémoire.
[4]Le temps d'interconnexion est le temps nécessaire pour communiquer une information d'un transistor à l'autre.

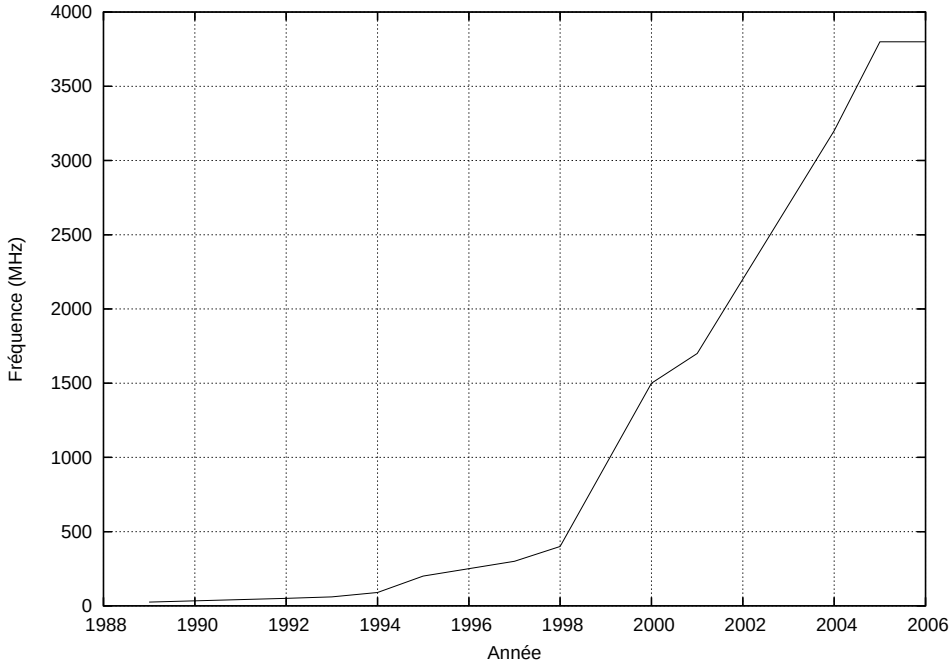


FIG. 2.1 – Évolution de la fréquence maximale des processeurs

de mémoire cache dans les processeurs, il n'en reste pas moins que les applications nécessitant de grandes quantités de mémoire ne profitent pas complètement de la montée en fréquence. L'augmentation des performances des applications n'est plus proportionnelle à la montée en fréquence. On a atteint un seuil où ce n'est pas la fréquence processeur mais bien la bande passante mémoire qui limite les applications. Il n'est donc plus intéressant d'avoir des fréquences de processeur trop élevées.

La seconde voie d'augmentation des performances des processeurs vient de l'ILP. La première piste à avoir été explorée dans ce domaine est d'extraire du parallélisme d'un flot d'exécution grâce aux pipelines. Aujourd'hui, on a atteint des profondeurs de pipeline limites pour les processeurs scalaires. En effet, une profondeur de pipeline de 10 ou 20 étages est particulièrement difficile à mettre en place car la charge de travail entre deux étages du pipeline finit par se réduire à une unique opération minimale comme l'addition de deux entiers. De plus, la logique nécessaire à l'ajout d'un étage de pipeline est très importante et apporte trop de complexité par rapport au gain de performances. La seconde piste utilisée est d'extraire du parallélisme en réordonnançant les instructions. La complexité de la logique nécessaire pour exploiter dynamiquement plus de quelques instructions par cycle est fonction du carré du nombre d'instructions que l'on peut exécuter simultanément. Il devient donc trop compliqué de poursuivre l'augmentation des performances en utilisant des méthodes d'extraction du parallélisme au niveau des instructions.

On constate que les méthodes utilisées jusqu'à présent pour augmenter les performances des processeurs ont atteint leurs limites. Il est donc nécessaire de trouver une autre approche pour utiliser les transistors supplémentaires qu'apporte la loi de Moore. La solution viendra encore une fois du parallélisme avec de nouveaux types de processeurs comme les processeurs multicœurs ou les processeurs multithreads.

2.3.2 Architectures des processeurs

L'architecture des processeurs est en mutation vers plus de parallélisme. Deux nouveaux types de processeurs émergent : les processeurs multicœurs et les processeurs multithreads. Ces deux caractéristiques ne sont, par ailleurs, pas exclusives puisqu'il existe des processeurs à la fois multicœurs et multithreads.

2.3.2.1 Architectures multicœurs

Les architectures multicœurs [27, 40, 38, 41, 33] sont une approche pour permettre l'augmentation globale des performances d'un processeur. Nous suivrons le plan suivant pour présenter les processeurs multicœurs. Nous allons tout d'abord présenter le concept de processeur multicœur, ensuite nous détaillerons trois approches de processeurs multicœurs. Enfin, nous mettrons en évidence les avantages et inconvénients de l'approche multicœur.

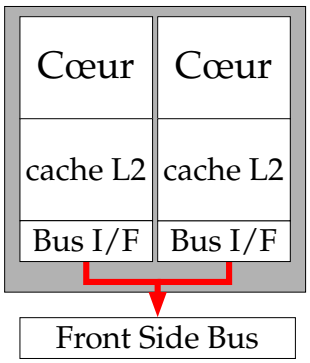
Conception

La diminution de la finesse de gravure des transistors permet, pour un processeur donné, de diminuer la surface nécessaire à ce processeur. L'idée du processeur multicœur est simple. Si l'on arrive à suffisamment diminuer la taille d'un processeur, pourquoi ne pas essayer de mettre plusieurs processeurs ainsi réduits (autrement appelés cœurs) au sein d'un seul microprocesseur qui sera dit multicœur. L'approche la plus retenue pour la conception des processeurs multicœurs généralistes est de répliquer des cœurs de processeurs identiques au sein du processeur multicœur. Chaque cœur est en première approximation un processeur monocœur d'une génération précédente. Bien évidemment,il est nécessaire d'ajouter, en plus des cœurs, la logique indispensable pour gérer des accès multiples à la mémoire, mais aussi celle pour la gestion de plusieurs mémoires cache, ... Les architectures multicœurs permettent d'agréger la puissance de tous les cœurs présents dans le processeur. Ainsi, un processeur n-cœur composé de n cœurs identiques, est en théorie n fois plus puissant que chaque cœur pris séparément à fréquence équivalente. Cette approche permet donc une augmentation de la puissance théorique des processeurs sans changer la fréquence. Comme nous allons le voir, suivant les approches choisies, l'augmentation de puissance réellement utilisable peut être fortement dégradée par rapport à l'augmentation de la puissance théorique.

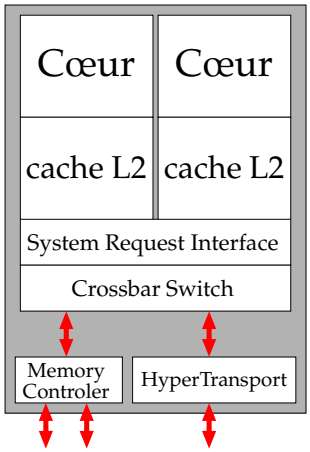
Les approches

Il existe plusieurs approches pour réaliser un processeur multicœur. Hormis la distinction concernant le cœur lui-même, la différence majeure réside dans la gestion de la mémoire cache entre les différents cœurs. Cette mémoire peut être partagée entre les différents cœurs comme c'est le cas dans le processeur UltraSparc T1 de Sun (FIG. 2.2(c)). La mémoire cache peut aussi être spécifique à chaque cœur comme c'est le cas des processeurs Pentium D d'Intel (FIG. 2.2(a)) et Athlon64 X2 d'AMD (FIG. 2.2(b)). Dans cette section, nous allons détailler trois processeurs multicœurs : le Pentium D d'Intel, l'Athlon64 X2 d'AMD et l'UltraSparc T1 de Sun.

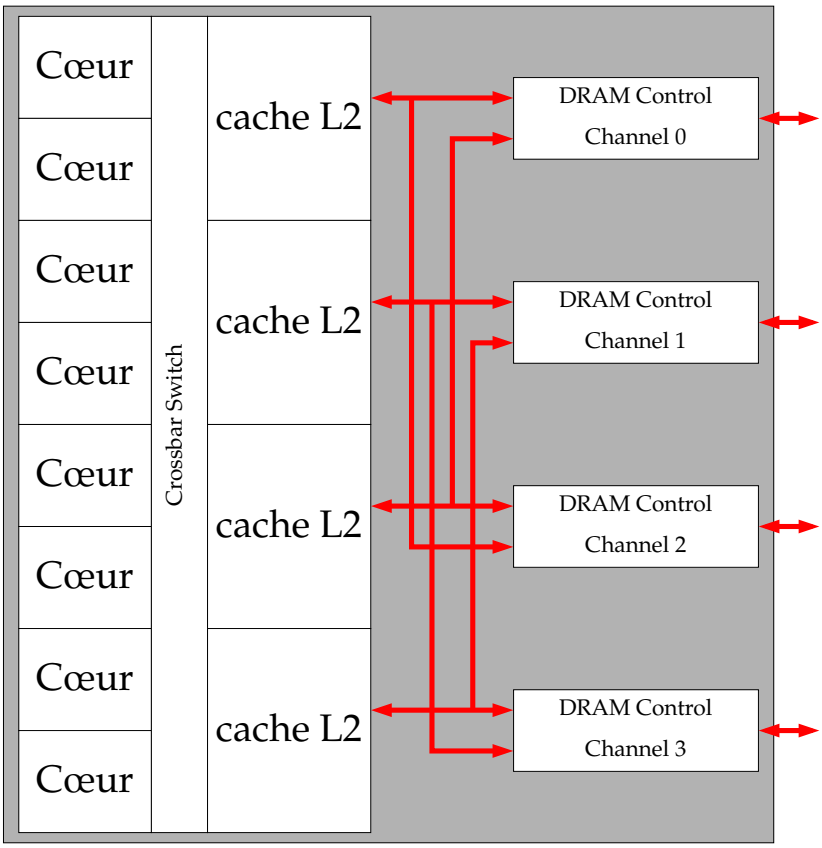
Intel avec son Pentium D propose un processeur constitué de deux cœurs Pentium 4 dépourvus de la technologie HyperThreading[44]. Ces deux cœurs sont implantés de la manière la plus simple qui soit. Ils sont juste accolés et partagent le bus d'accès à la mémoire centrale avec une politique d'arbitrage de bus classique. On peut à juste titre comparer le Pentium D à une machine



(a) Biccœur selon Intel (Pentium D)



(b) Biccœur selon AMD (Athlon64 X2)



(c) Octocœur selon Sun (UltraSparc T1)

FIG. 2.2 – Exemples d'architectures multicœur

biprocesseur SMP. En effet, le seul moyen de faire communiquer les deux cœurs est d'utiliser la mémoire centrale. Ce processeur ne tire pas vraiment tous les avantages d'une architecture bicœur mais vise seulement à fournir le double de puissance théorique par rapport à son homologue monocœur. Dans la pratique, une telle approche est fortement limitée par la latence mémoire qui est doublée dans le pire cas. Au lieu de résoudre le problème de la limitation de la bande passante mémoire, cette approche va amplifier le problème et ce bien que cette limitation soit déjà très contraignante pour les architectures monocœurs comme nous l'avons vu en 2.3.1.

L'approche utilisée par AMD pour son Athlon64 X2 est un peu plus avancée que celle d'Intel. En effet, l'Athlon64 X2 tire parti de la technologie HyperTransport développée initialement pour les architectures multiprocesseurs. Cette technologie a été étendue pour permettre des communications directes (sans passer par la mémoire centrale) entre les deux cœurs. On peut donc dire que l'Athlon64 X2 dispose d'une meilleure intégration des deux cœurs au sein du processeur. Néanmoins, cette approche subit de la même manière que la précédente le goulot d'étranglement de l'accès mémoire, même si elle facilite les communications entre les différents cœurs.

L'approche de Sun est sans conteste l'approche la plus aboutie. En effet, l'intégration des différents cœurs est très poussée. Elle permet aux différents cœurs de communiquer directement, mais permet aussi de limiter l'impact de l'accès mémoire par une multiplication des contrôleurs mémoire et des mémoires cache partagées. On peut comparer l'UltraSparc T1 à une machine SMP car elle dispose des composants principaux, à savoir plusieurs unités de calcul et une mémoire commune. De plus, chaque cœur est un cœur multithread (le concept de processeur/cœur multithread sera développé plus tard), il peut donc y avoir recouvrement des latences mémoire par commutation d'un thread à l'autre.

Comme nous venons de le voir, la multiplication des cœurs dans un processeur permet d'augmenter les performances théoriques. Mais suivant l'approche, les performances réellement exploitables dépendent surtout du niveau d'intégration et de la complexité de la gestion de la mémoire cache ainsi que des capacités d'accès à la mémoire centrale.

Avantages/inconvénients

Le principal avantage des architectures multicœurs est de pouvoir palier aux limites en terme de montée en fréquence et de consommation électrique/dissipation thermique mais aussi d'augmenter l'intégration. Néanmoins, quelle que soit l'approche, si la puissance théorique d'un processeur multicœur est le produit du nombre de cœurs par la puissance individuelle de chaque cœur, il faut nuancer cette puissance par la dégradation de la bande passante mémoire due à l'adjonction de cœurs. Comme nous l'avons mentionné en 2.3.1, le goulot d'étranglement mémoire est important pour de nombreuses d'applications. L'architecture multicœur n'est pas la solution miracle pour l'augmentation des performances de toutes les applications. De plus, il faut nécessairement une application parallèle pour pouvoir tirer avantage d'une telle architecture. Le gain en performance des applications qui était "gratuit" jusqu'à présent (il suffisait de changer de processeur) est bel et bien fini. Les codes séquentiels désirant profiter des performances des architectures multicœurs doivent nécessairement être parallélisés. En ce qui concerne les codes déjà parallélisés, ils doivent être adaptés pour tenir compte du problème de latence mémoire.

2.3.2.2 Architectures multithreads

Les architectures de processeurs multithreads[57, 62, 58, 44] ont pour but d'optimiser l'utilisation des processeurs en évitant le gaspillage de cycles. Ce gaspillage provient essentiellement des latences mémoire et de la profondeur des pipelines. Nous allons tout d'abord voir quel est le concept d'une architecture de processeur multithread. Ensuite, nous détaillerons trois approches pour la conception de processeurs multithreads. Enfin, nous discuterons des intérêts d'une telle architecture.

Concept

L'approche multithread a pour objectif de limiter le nombre de cycles d'horloge perdus à attendre des données venant de la mémoire ou à attendre la fin d'une opération complexe comme la multiplication ou la division qui peuvent prendre plusieurs cycles. Pour atteindre cet objectif, les architectures multithreads vont "recouvrir" les cycles perdus par un thread en exécutant des instructions provenant d'un autre thread. Illustrons le principe dans le cas des accès mémoire. Il est impossible pour une grande majorité d'applications d'avoir tous les opérandes situés dans les registres. En effet, il y a trop peu de registres. Il faut donc souvent faire des accès mémoire. Comme il n'est pas toujours possible, au sein d'un flot d'exécution, de recouvrir ces latences mémoire grâce aux opérations de pré-chargement, on choisit alors d'utiliser les opérations d'un autre flot d'exécution pour recouvrir ces latences. Un processeur multithread est donc un processeur capable de gérer plusieurs flots d'exécution ou threads dans le but d'optimiser l'utilisation de ses unités de calcul.

Approches

Il existe trois politiques de gestion des différents threads d'un processeur multithread[58]. La première est nommée *Interleaves multithreading*. Cette approche consiste à entremêler les instructions des différents threads de manière cyclique en exécutant successivement une instruction de chaque thread. La seconde approche qui est retenue dans le processeur Ultrasparc T1 de Sun[40] est nommée *Blocked multithreading*. Elle consiste à exécuter les instructions d'un thread tant qu'il ne fait pas d'instruction ayant une latence élevée comme pour un accès mémoire. Dans le cas d'une instruction à forte latence, le thread est dit bloqué et l'on commence à exécuter un autre thread jusqu'à ce qu'il soit à son tour bloqué. La troisième approche qui est celle connue sous le terme *HyperThreading* dans les processeurs d'Intel[44] est nommée *Simultaneous multithreading*. Cette approche consiste à entremêler les instructions de plusieurs threads pour utiliser au mieux toutes les unités de calcul disponibles.

Avantages/inconvénients

Comme nous venons de le voir, l'approche multithread permet d'éviter le gaspillage de cycles d'horloge sur l'ensemble des applications exécutées sur un processeur. Cette méthode permet donc, dans le meilleur des cas, d'approcher au maximum les performances théoriques d'un processeur en recouvrant au mieux les "cycles perdus" lors de l'exécution d'un programme. Néanmoins, les performances théoriques doivent être revues à la baisse à cause de la complexité de l'ordonnancement des flots d'exécution. De plus, la plupart des processeurs multithreads ne sont

pas dotés de notion de qualité de service. Il se peut donc qu'un flot d'exécution perturbe l'exécution de tous les autres flots d'exécution avec qui il partage des ressources. En effet, un des flots exécuté peut, par exemple, monopoliser la totalité de la mémoire cache et provoquer la suppression des lignes de cache utilisées par les autres flots. Dans un cas extrême, ce comportement peut provoquer un ralentissement global de l'exécution de tous les flots d'exécution par rapport à une exécution sur un processeur non multithread équivalent[47]. Ce phénomène est si bien reconnu que de nombreuses personnes désactivent l'*HyperThreading* des processeurs Intel pour les transformer en processeurs monothreads.

2.3.2.3 Discussion

La transition vers les processeurs multicœurs (multithreads ou non) est inévitable car les efforts du passé pour augmenter les performances des processeurs ont atteint leurs limites physiques. Des techniques comme la mise en place des pipelines et autres méthodes visant à extraire du parallélisme d'un code séquentiel ou bien encore l'augmentation de la fréquence ne sont plus envisageables. Bien que les nouvelles technologies multicœur/multithread permettent d'augmenter de manière significative les performances brutes, elles ne pourront être pleinement bénéfiques que lorsque les applications auront définitivement abandonné les approches de programmation séquentielle au profit d'une programmation parallèle. L'approche privilégiée pour cette évolution est la parallélisation des applications via l'utilisation de processus légers. Bien souvent, cette parallélisation devra être explicite même si certaines évolutions des compilateurs et bibliothèques permettent ou vont permettre l'extraction automatique de processus légers[14, 23].

2.3.3 Les architectures ccNUMA

Concept

L'architecture ccNUMA[46, 36, 59] est une agrégation de plusieurs machines de type SMP. Cette agrégation est nommée nœud NUMA comme on peut le voir sur la figure 2.3. Ces nœuds ont leurs mémoires locales interconnectées par un réseau d'interconnexion matériel qui donne l'illusion d'une mémoire unique. La mémoire étant physiquement distribuée, il n'y a aucune garantie que l'accès aux données soit satisfait en un temps uniforme. Les données mémoire étant distribuées sur les mémoires physiques des différents nœuds, le temps d'accès à une donnée pour un processeur donné n'est donc pas uniforme. En effet, celui-ci varie suivant la position de la donnée dans la mémoire locale au nœud contenant ce processeur ou dans la mémoire d'un autre nœud. La partie "cc" de "ccNUMA" veut dire *cache coherent*. Ce terme montre la capacité donnée à chaque processeur de disposer, pour chaque variable, d'une donnée cohérente ; i.e. une donnée ayant la même valeur pour tous les processeurs. Ceci implique que les mémoires cache des processeurs soient mises à jour pour maintenir cette cohérence. Il existe deux méthodes pour maintenir la cohérence. La première est le *snoopy bus protocol* dans lequel les mémoires cache diffusent les valeurs des variables sur un bus et mettent à jour leurs copies locales des variables. La seconde méthode est *directory memory* qui désigne une partie de la mémoire qui garde une trace de quelle mémoire cache détient quelles copies ainsi que la validité de ces copies.

Il a existé des architectures NUMA sans mécanisme de cohérence de cache. Actuellement, seules les ccNUMA subsistent, les autres étant trop difficiles à programmer. Par abus de langage,

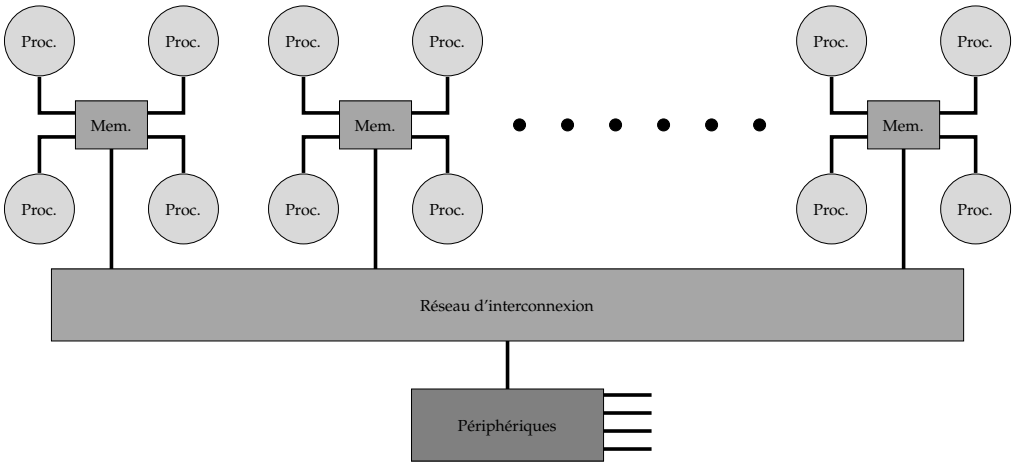


FIG. 2.3 – Architecture ccNUMA

et dans le reste du document, nous considérerons les architectures NUMA comme des architectures ccNUMA.

Avantages/inconvénients

Le principal intérêt des architectures NUMA est de permettre à un grand nombre de processeurs (jusqu'à 2048 pour l'architecture SGI Altix 3700) de communiquer par mécanisme de mémoire partagée. Très souvent, les temps de communication par mémoire partagée entre deux nœuds d'une architecture NUMA sont nettement inférieurs à ceux des communications réseaux dans le cas d'une grappe par exemple. Il est donc intéressant de concevoir ces architectures pour les applications parallèle bien que ces architectures soient particulièrement difficiles à programmer, par rapport à une architecture SMP, pour en tirer la quintessence. En effet, il faut tenir compte de la hiérarchie mémoire pour favoriser les accès mémoire ayant le temps d'accès le plus cours et ainsi optimiser les performances des applications. Bien qu'étant à mémoire partagée, ces architectures n'en restent pas moins particulièrement adaptées à la programmation mémoire distribuée avec des bibliothèques de communication optimisées mais aussi à la programmation multithread. En effet, disposer d'un grand nombre de processeurs au sein d'une architecture à mémoire partagée permet aux applications déséquilibrées de facilement migrer un processus ou un thread d'un processeur à l'autre et ainsi d'équilibrer la charge de calcul

2.3.4 Discussion

La montée en puissance des supercalculateurs se fait par l'intermédiaire d'une augmentation des performances des briques de base (processeurs, nœuds de calcul,...) qui les composent. Cette augmentation des performances passe par l'accroissement au sens large du nombre de processeurs. En effet, on constate une hausse du nombre de cœurs dans les microprocesseurs ainsi qu'une hausse du nombre de microprocesseurs au sein des nœuds de calcul comme les architectures NUMA. Cette transition architecturale permet de palier aux limitations physiques qui brident la course à la puissance des supercalculateurs telle qu'on l'a connue jusqu'alors. Néanmoins cette transition n'est pas sans conséquence sur les applications. En effet, la complexification

des architectures matérielles implique la mise en œuvre de techniques logicielles comme le placement mémoire ou encore l'utilisation du multithreading au sein des applications. Une application qui ignorerait la complexité des architectures et n'incorporerait pas ces techniques logicielles aurait des performances bien en deçà des capacités du supercalculateur. La transition logicielle induite par la transition architecturale est plus que jamais d'actualité car les supercalculateurs les plus puissants d'aujourd'hui utilisent déjà les innovations technologiques présentées.

2.4 Les supercalculateurs d'aujourd'hui

Dans cette section, nous présentons trois supercalculateurs actuels présents au top500 et représentatifs des trois types d'architecture de supercalculateurs. Ces calculateurs ont été conçus autour des composants de base que nous venons de décrire et permettent donc à un grand nombre de flots d'exécution de s'exécuter simultanément. Nous présentons tout d'abord une architecture massivement parallèle : BlueGene/L puis une architecture mémoire partagée le SGI Altix 3700 et enfin une architecture hybride TERA10.

2.4.1 Une architecture massivement parallèle : l'IBM BlueGene/L

La machine BlueGene/L[7, 59] située au Lawrence Livermore National Laboratory est la championne du monde au top500[5] de Juin 2006. Elle développe, au banc d'essai, une puissance de 281 téraFlops.

BlueGene/L est une architecture massivement parallèle composée de 131 072 cœurs. Chaque cœur est un PowerPC 440 cadencé à 700MHz. L'architecture de BlueGene/L est très hiérarchique comme le montre la figure 2.4. En effet, les cœurs sont tout d'abord regroupés deux par deux au sein d'un processeur multicœur. Chaque processeur va disposer d'une mémoire de 512Mo qui lui est propre. Seuls les cœurs peuvent donc communiquer par mémoire partagée. Ces processeurs sont à leur tour regroupés deux par deux pour former une *compute card*, soit 4 cœurs par *compute card*. Ces *compute cards* sont alors regroupées par 16 pour former une *node card* soit 64 cœurs par *node card*. Ces *node cards* sont regroupées par 32 pour former des armoires de calcul disposant donc de 2 048 cœurs. Enfin, les 64 armoires sont interconnectées pour former un ensemble de 131 072 cœurs.

Le fort niveau d'intégration de cette architecture vient de la faible consommation des cœurs utilisés. Cette intégration a pour avantage de permettre des communications très efficaces. En effet, la probabilité de communication d'un cœur avec ses proches voisins est très élevée car ses voisins sont au nombre de 2 048. BlueGene/L dispose en plus de cinq réseaux d'interconnexion. Deux d'entre eux sont destinés aux communications interprocesseurs. Le premier se présente sous la forme d'un tore 3D et est principalement dédié pour les communications point à point et est doté d'une bande passante de 175Mo/s. Le second a une structure arborescente et est destiné aux opérations collectives comme la diffusion ou la réduction. Ce dernier dispose d'une bande passante de 350Mo/s. Les trois autres réseaux sont utilisés pour la gestion interne de la machine.

BlueGene/L est une architecture destinée à exécuter des codes de calcul mémoire distribuée. En effet, cette approche est la plus appropriée à la topologie de la machine. IBM fournit par ailleurs une version de MPI optimisée pour tirer parti des différents réseaux d'interconnexion. L'utilisation recommandée pour un code de calcul classique est d'utiliser sur chaque processeur un cœur pour

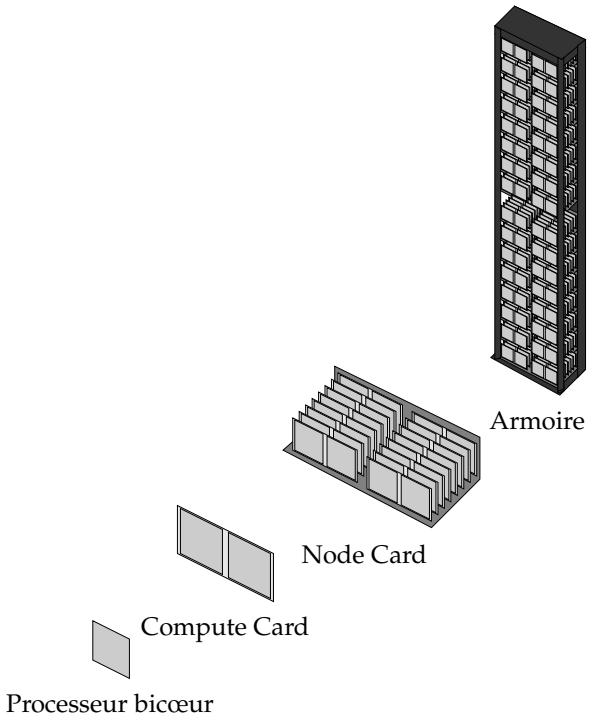


FIG. 2.4 – BlueGene/L

le calcul et l'autre pour les communications. Ceci a pour effet de réduire la puissance crête à 183 teraFlops.

Cette architecture est destinée aux études de physique des matériaux, et notamment aux études de vieillissement des têtes nucléaires au Lawrence Livermoore National Laboratory.

2.4.2 Une architecture mémoire partagée : SGI Altix 3700 Bx2

L'architecture SGI Altix 3700 Bx2[59] peut être considérée comme un supercalculateur à part entière avec jusqu'à 2 048 processeurs Itanium2 dans un système à mémoire partagée.

L'architecture SGI Altix 3700 Bx2 est conçue autour du système de cohérence de cache NUMA-flex de SGI et du processeur Itanium 2 d'Intel. L'architecture SGI Altix est composée de *Bricks*. Ces derniers peuvent être de plusieurs types. Les C-Bricks (voir FIG. 2.5) ou modules de calcul sont composés de deux modules contenant chacun deux processeurs Itanium 2 et une mémoire associée. Les FSBs[5] de ces processeurs sont interconnectés à un ASIC[6] appelé *SHUB*. L'interface SHUB permet de connecter les deux processeurs à la mémoire, au système d'entrée sortie et à d'autres SHUBs via le réseau NUMAlink. Le SHUB permet aussi d'interconnecter les deux groupes de deux processeurs du C-Bricks avec une bande passante égale à celle du FSB de l'Itanium 2 soit 6,4Go/s. Les R-Bricks sont les modules chargés d'implémenter le système global de mémoire partagée. Ceux-ci sont des routeurs du système NUMAlink qui, reliés au C-Bricks via les SHUBs (voir

[5]Front Side Buffer : bus système qui permet au processeur de communiquer avec la mémoire centrale du système.

[6]Application Specific Integrated Circuit : circuit intégré spécialisé. En général, il regroupe un grand nombre de fonctionnalités uniques et/ou sur mesure.

FIG. 2.6), permettent de donner l'illusion d'une mémoire unique.

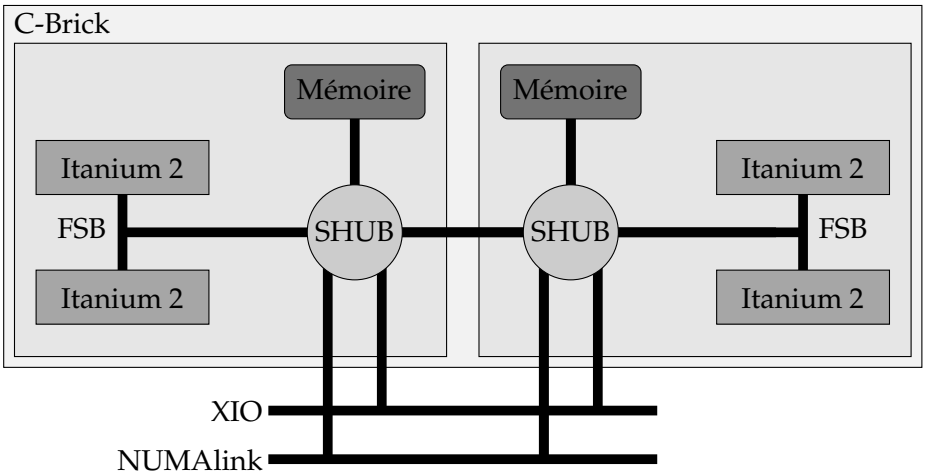


FIG. 2.5 – Schéma d'une C-Brick

L'architecture SGI Altix 3700 Bx2 est conçue pour les applications parallèles mémoire partagée. C'est une architecture qui pousse à l'extrême le concept d'architecture NUMA car elle dispose de nombreux niveaux de hiérarchie mémoire. Il paraît clair que les applications performantes destinées à ce type d'architecture doivent être particulièrement optimisées pour limiter la contention et gérer correctement les accès mémoire. Cet exemple d'intégration au sein d'un espace mémoire unique de 2 048 processeurs est extrême à l'heure actuelle. Néanmoins, avec la banalisation de processeurs multicœurs et multithreads, il y a fort à parier que des machines avec un grand nombre de flots d'exécution qui s'exécutent en parallèle au sein d'un espace mémoire unique vont se démocratiser dans le monde des supercalculateurs. Cette architecture montre avec les techniques d'aujourd'hui ce que pourra être la complexité de nombreuses machines dans un futur proche.

Cette architecture est la brique de base de systèmes encore plus grands comme la machine Columbia de la NASA figurant en quatrième position au top500. Celle-ci compte plus de 10 240 processeurs dont une sous-partie de 2 048 processeurs qui partagent une mémoire unique.

2.4.3 Une architecture hybride de type grappe de NUMA : TERA10

La machine TERA10 est un supercalculateur de type grappe de nœuds NUMA et est située au CEA/DAM Île de France. Ce supercalculateur d'une puissance crête de plus de 55 teraFlops est construit autour du nœud de calcul Bull NovaScale [3, 59] et de l'interconnexion Quadrics[4, 59].

Le nœud de calcul Bull NovaScale est une architecture de type NUMA (voir FIG. 2.7). Ce nœud est composé de 8 processeurs bicœurs Itanium 2 Montecito répartis sur les 4 QBB[7] soit 2 processeurs par QBB sur les 4 possibles par QBB. Ce nœud dispose de plus d'une mémoire totale de 48 Go. La mémoire et les processeurs de chaque QBB sont reliés entre eux par le FAME Scalability Switch qui est en charge de la cohérence mémoire, des accès distants et des entrées/sorties. Les 544 nœuds de calcul sont reliés entre eux par une interconnexion Quadrics Elan4 pour former une machine dotée de près de 9 000 processeurs.

[7]Quad Brick Block : carte mère quadri-processeur.

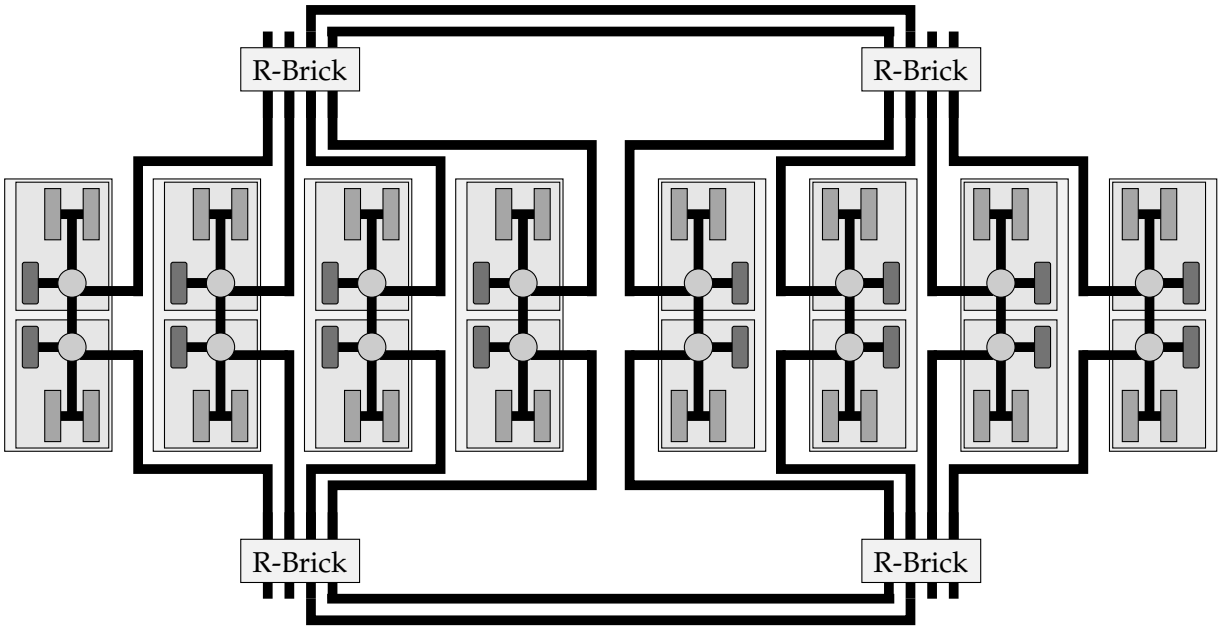


FIG. 2.6 – Schéma d'interconnexion des R-Bricks

TERA10 met donc en œuvre deux approches : l'approche mémoire partagée au sein des nœuds de calcul et l'approche mémoire distribuée entre les nœuds de calcul. Cette architecture est donc un des nombreux exemples de supercalculateurs construits sur l'approche hybride mémoire partagée/mémoire distribuée.

2.5 Mise en évidence d'une tendance pour les supercalculateurs de demain

Le nombre de processeurs/cœurs des supercalculateurs est en constante augmentation. Cette tendance va sans doute se poursuivre dans le futur. En effet, l'augmentation du nombre de processeurs/cœurs est, à l'heure actuelle, la seule solution pour permettre un accroissement de la puissance. On constate de plus que les architectures tendent toutes à devenir de type hybride. En effet, même l'architecture BlueGene qui est dite une architecture mémoire distribuée massivement parallèle est en fait hybride car les processeurs sont des architectures bicœurs et donc les cœurs partagent la mémoire physique. L'augmentation du nombre de cœurs dans les processeurs va donc imposer une sous-structure mémoire partagée dans tous les supercalculateurs. On constate que les architectures mémoire partagée uniquement ne permettent pas à l'heure actuelle de fournir la puissance suffisante pour figurer dans le top 10 des machines les plus puissantes. Ces architectures mémoire partagée ont été agrégées pour former des grappes composées de nœuds de grande taille. Les architectures totalement mémoire distribuée ou totalement mémoire partagée sont donc en perte de vitesse. On peut donc dire que la tendance actuelle et future des supercalculateurs est l'architecture hybride mémoire partagée/mémoire distribuée.

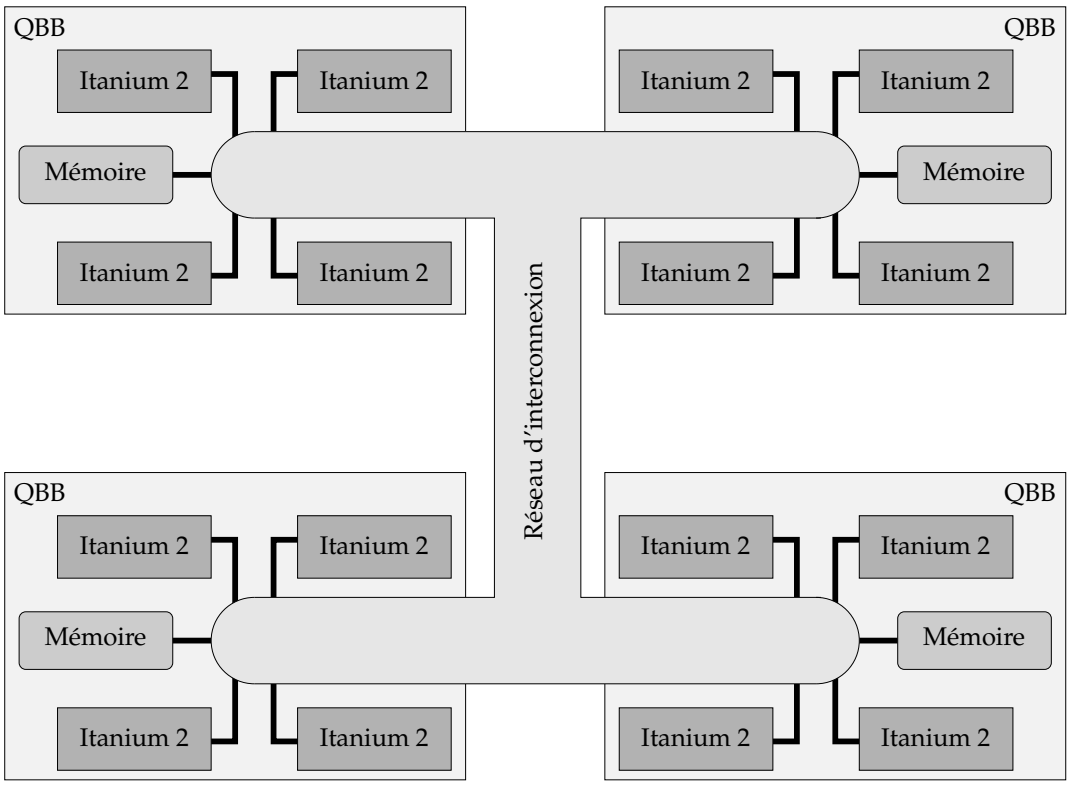


FIG. 2.7 – Architecture Bull NovaScale

Ce chapitre a permis de présenter les différentes caractéristiques des architectures de supercalculateurs d'aujourd'hui et de déterminer une ligne directrice concernant l'évolution des supercalculateurs. Il a aussi permis de mettre en évidence les nouveaux problèmes rencontrés et donc la nécessité d'une évolution des environnements logiciels destinés au calcul hautes performances. Cette analyse est très importante vis-à-vis des travaux présentés dans ce document car l'environnement de programmation que nous avons développé se doit d'être performant sur tous types d'architectures de supercalculateur. Il convient maintenant de présenter les différents modèles de programmation et les outils disponibles pour construire des applications destinées aux supercalculateurs.

Chapitre 3

Applications visées en calcul scientifique parallèle

Ce chapitre a pour objectif de présenter les caractéristiques courantes des codes scientifiques parallèles et en particulier dans le cadre de la mécanique des fluides. Les codes construit sur ce modèle sont très courants dans le monde du calcul scientifique et plus particulièrement au CEA. Cette présentation va permettre de déterminer les principales difficultés rencontrées par les codes de calcul du fait des structures de données utilisées ou de la méthode de parallélisation. Nous détaillerons entre autre l'impact des choix de conception du code sur les performances. La connaissance des applications est très importante pour nos travaux car elle va guider la conception de notre environnement de programmation.

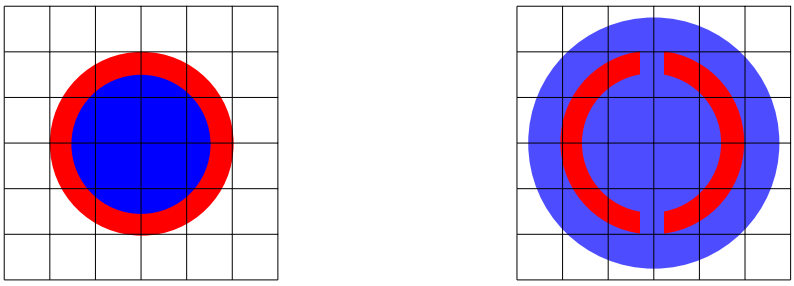
Ce chapitre se décompose en deux grandes parties. La première va décrire différentes structures de données couramment utilisées par les applications. La deuxième partie va décrire l'approche de programmation SPMD qui est une approche standard de parallélisation de code.

3.1 Exemple de maillage et de structures de données en mécanique des fluides

3.1.1 Maillages eulériens structurés

Les codes utilisant un maillage eulérien structuré sont des codes où le maillage reste fixe au cours du temps. Pour les cas les plus simples, ce maillage se réduit à un tableau. La figure 3.1 est un exemple de maillage eulérien structuré.

Les codes à maillage eulérien structuré sont des codes robustes du point de vue numérique[20]. Les cas simples monomatériaux utilisant cette approche sont, de plus, aisément optimisables en utilisant par exemple la méthode de directions alternées[45]. Cette méthode couplée à une optimisation des tableaux, pour utiliser au mieux la mémoire cache des processeurs, permet d'obtenir de très bonnes performances. Néanmoins cette approche est vite consommatrice en mémoire. En effet, cette approche va nécessiter un grand nombre de mailles pour obtenir une précision élevée. De plus, dès que le nombre de matériaux de la simulation augmente, le nombre de dimensions des tableau suit cette augmentation ce qui complexifie considérablement les optimisations du fait de l'ajout d'indirections pour accéder aux données de tel ou tel matériau.



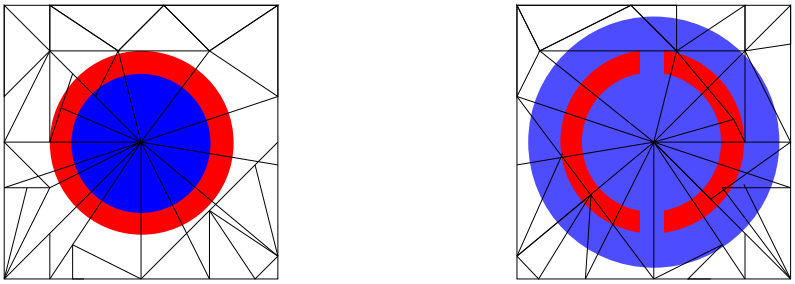
(a) Cycle 0 (b) Cycle 1

FIG. 3.1 – Exemple de maillage eulérien structuré

Les codes utilisant des maillages structurés sont courants et peuvent être très performants pour réaliser des simulations de systèmes physiques utilisant peu de matériaux. Pour les simulations plus complexes, il faut s'orienter vers des maillages plus complexes.

3.1.2 Maillages lagrangiens non-structurés

Les codes lagrangiens sont des codes où les nœuds du maillage se déplacent à la vitesse de la matière. La figure 3.2 est un exemple de maillage lagrangien non-structuré.



(a) Cycle 0 (b) Cycle 1

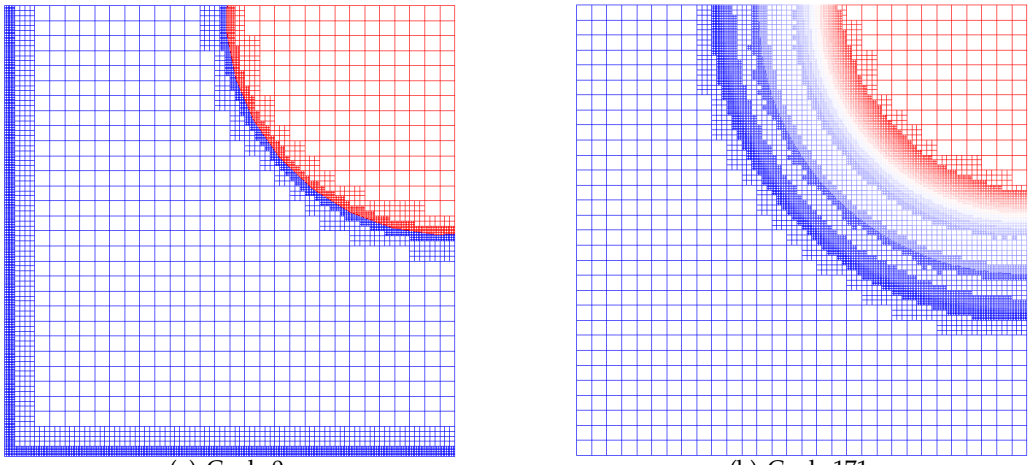
FIG. 3.2 – Exemple de maillage lagrangien non-structuré

Ces codes offrent donc un excellent rapport précision coût[20]. En effet, le maillage suivant la matière, la précision est automatiquement augmentée dans les zones ou les phénomènes physiques sont les plus marqués. Néanmoins, ils peuvent se révéler fragiles du point de vue numérique, par exemple en présence de déformations trop importantes ou d'apparition de surfaces de glissement. Ces codes, de par l'absence de structure, sont très difficilement optimisables car le voisinage d'une maille est inconnu a priori. Il y a donc de nombreuses indirections et ces codes génèrent de nombreux défauts de cache.

Les maillages lagrangiens non-structurés ont de bonnes propriétés de précision mais les difficultés mathématiques qu'ils engendrent les rendent relativement fragiles. Ce sont donc des codes que l'on rencontre plus rarement.

3.1.3 Maillages AMR (Adaptive Mesh Refinement)

Un maillage AMR est un maillage qui évolue en cours de calcul pour offrir une précision accrue dans les zones où les phénomènes physiques sont les plus marqués. Dans ces zones, les mailles vont être subdivisées pour accroître la précision (la précision est d'autant plus élevée que les mailles sont de taille plus réduite). La figure 3.3 est une illustration d'évolution de maillage AMR.



(a) Cycle 0 (b) Cycle 171

FIG. 3.3 – Exemple d'évolution de maillage AMR

Un maillage AMR permet d'éviter le choix d'un maillage dès l'instant initial. En effet, le maillage va être raffiné au cours du calcul pour mieux contrôler le coût et la précision de la simulation. Lorsque le domaine de calcul est découpé suivant la méthode classique SPMD, cela induit des déséquilibrages de charge entre les tâches parallèles : le maillage s'adaptant au cours de la simulation, les charges de travail entre les tâches vont évoluer au cours du calcul. Ainsi, la parallélisation efficace des codes de calcul est d'un enjeu très important. Comme les maillages non structurés, les codes utilisant des maillages AMR sont difficilement optimisables dus au nombre élevé d'indirections induites par le fait que la structure du maillage change.

3.2 L'approche Single Process Multiple Data (SPMD) par décomposition de domaines

L'approche de parallélisation SPMD par décomposition de domaines est très courante en calcul scientifique[13]. Cette approche consiste à découper le domaine de calcul initial en sous-domaines, chacun de ces derniers étant attribué à une tâche parallèle. La figure 3.4 illustre ce mécanisme.

Une notion importante pour l'approche de parallélisation SPMD par décomposition de domaines est la notion de mailles fantômes[45]. Ces mailles vont permettre de donner l'illusion que le domaine de calcul est continu. Comme on peut le voir sur la figure 3.4, ces mailles permettent de répliquer les mailles du bord du sous-domaine dans le sous-domaine voisin. Ainsi, dans un grand nombre de cas, le schéma numérique va pouvoir être appliqué sur toutes les mailles du sous-domaine et va utiliser les valeurs des mailles fantômes sur les frontières du sous-domaine. Les

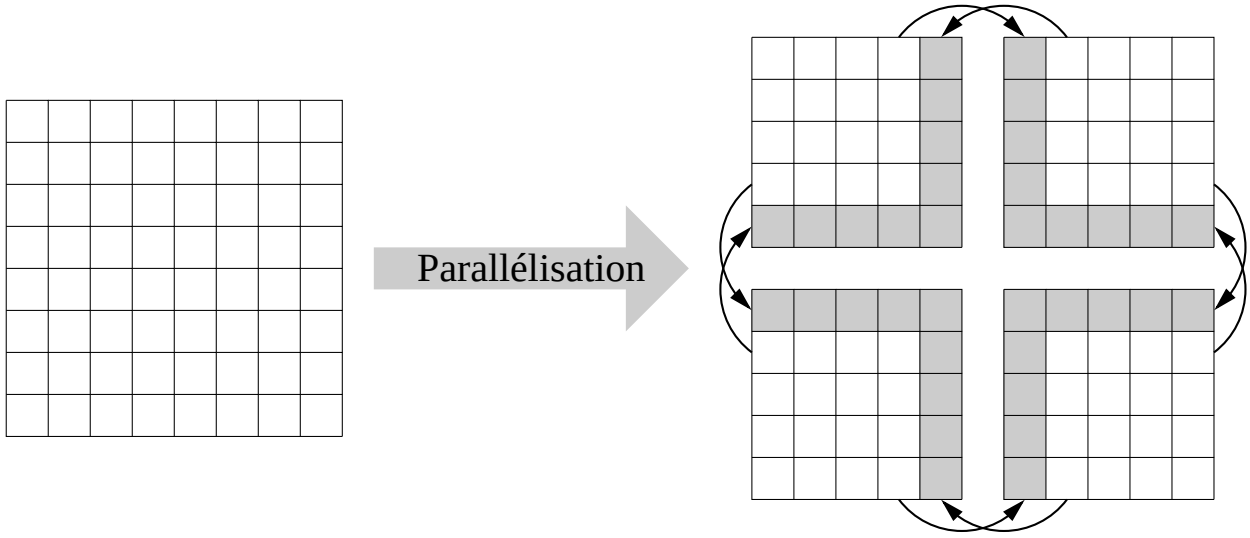


FIG. 3.4 – Exemple de parallélisation SPMD

mailles fantômes vont être mises à jour régulièrement au cours du calcul pour refléter les calculs faits dans les sous-domaines voisins. Il faut noter que le nombre de mailles fantômes nécessaire peut varier suivant le schéma utilisé.

Cette méthode est très pratique car, souvent, elle n'implique pas ou peu de modifications du schéma numérique. Les communications induites par la mise à jour des mailles fantômes sont fréquemment un frein à l'extensibilité des applications. Une des difficultés majeures de cette approche est de garder un ratio temps de calcul sur temps de communication raisonnable en choisissant judicieusement le nombre de sous-domaines et de processeurs que l'on utilise.

Ce chapitre a permis de détailler et analyser les approches couramment utilisée dans la conception de codes de calcul hautes performances, pour la mécanique des fluides, indépendamment des outils disponibles. Nous avons constaté que la complexité des structures de maillage influe directement sur la facilité d'optimiser les codes de calcul, en particulier au niveau des mémoires caches. Les codes de calcul scientifique complexe ont donc besoin d'outils et de méthodes pour aider à la parallélisation et l'optimisation.

Chapitre 4

Panorama des modèles de programmation parallèle et outils existants

Ce chapitre a pour objectif de détailler les modèles de programmation parallèle qui vont permettre de concevoir des applications parallèles hautes performances destinées aux supercalculateurs. Nous présenterons de plus différents outils représentatifs de chaque modèle de programmation. Cette étude a pour objectif de déterminer le modèle et les fonctionnalités nécessaire à la parallélisation et l'optimisation de code de calculs complexes.

Nous commencerons ce chapitre par les définitions des notions fondamentales que nous manipulerons dans la suite du document. Ensuite, nous détaillerons et analyserons le modèle de programmation mémoire partagée ainsi que les outils implémentant ce modèle. La section suivante présentera le modèle de programmation mémoire partagée ainsi que les outils représentatifs. Le troisième modèle présenté sera l'approche mixte mémoire partagée et mémoire distribuée. Ces trois modèles constituent les trois approches utilisées pour la conception d'applications parallèles. La dernière partie de ce chapitre fera une analyse comparative des différents modèles et des outils que nous avons présentés.

4.1 Définitions

Dans cette section, nous définissons les notions essentielles des modèles de programmation parallèle. La principale notion utilisée en programmation parallèle est celle de tâche. En effet, un programme parallèle est une interaction de plusieurs tâches. Un programme non parallèle est, quant à lui, composé d'une seule tâche.

Définition 4.1: ***Tâche*** **–** Une tâche (ou flot d'exécution ou thread) est une suite logique séquentielle d'actions résultat de l'exécution d'un programme.

Une notion connexe à la notion de tâche est celle de processus. Le processus est une instance d'un programme qui est une notion liée au système d'exploitation et à la gestion de la mémoire.

Définition 4.2: ***Processus*** **–** Un processus est constitué d'une ou plusieurs tâches qui partagent un espace d'adressage commun. Si un processus comporte plusieurs tâches, il est dit multi-thread.

Le nerf de la programmation parallèle est la conception de mécanismes qui permettent à ces tâches de communiquer entre elles pour former une application de calcul parallèle.

Définition 4.3: ***Calcul parallèle*** **–** Le calcul parallèle consiste en le découpage d'un programme en plusieurs tâches qui peuvent être exécutées en même temps dans le but d'améliorer le temps global d'exécution du programme.

Attention, la notion de processus est souvent confondue avec celle de tâche dans les implémentations de bibliothèques de communication couramment utilisées en calcul parallèle. Dans nos travaux, et donc dans le reste du document, ces notions sont bien distinctes.

Enfin les notions de portabilité et de portabilité des performances sont très importantes pour pouvoir caractériser correctement les aptitudes d'une application de calcul parallèle.

Définition 4.4: ***Portabilité*** **–** capacité qu'a un code à s'adapter à tout type d'architecture.

Définition 4.5: ***Portabilité des performances*** **–** capacité qu'a un code à s'adapter à tout type d'architecture en conservant de bonnes performances.

Les concepts fondamentaux du calcul scientifique étant posés, nous allons maintenant détailler les différentes approches qui caractérisent le calcul parallèle.

4.2 Modèle de programmation à mémoire distribuée

Le parallélisme à mémoire distribuée est sans doute l'approche la plus utilisée en calcul parallèle hautes performances. Nous allons en présenter les principales caractéristiques. Il s'en suivra une présentation de différents outils logiciels. Enfin, nous détaillerons les principaux intérêts et les limites de cette approche.

4.2.1 Des communications explicites

En parallélisme à mémoire distribuée, la machine parallèle est vue comme un ensemble de processeurs ayant chacun leur mémoire associée. La machine parallèle est assimilée à une grappe de nœuds monoprocesseurs monocœurs non multithreadés. À chaque couple processeur/mémoire, nous allons associer une tâche(voir définition 4.1). Les mémoires des différentes tâches étant séparées, au sens logique, il est nécessaire d'avoir recours à des communications explicites pour accéder aux données des autres tâches. Les deux paradigmes de communication principaux sont le passage de messages et l'appel de procédure à distance (RPC).

4.2.1.1 Passage de messages

La communication par passage de messages est la plus utilisée en calcul hautes performances. Généralement, elle permet d'effectuer des communications *point à point*, i.e. un émetteur vers un récepteur et des opérations *collectives*, i.e. qui impliquent un ensemble de tâches. Lors d'une communication par passage de messages point à point, un émetteur envoie les données et le récepteur les reçoit explicitement. Il existe un autre type de communication par passage de messages : les communications collectives. Les communications collectives peuvent être divisées en trois catégories : synchronisation, communication et calcul. La catégorie synchronisation est essentiellement une barrière où toutes les tâches attendent toutes les autres tâches. Les opérations collectives de catégorie communication sont de deux types :

1. diffusion (*broadcast*) : permet de diffuser une même donnée à toutes les tâches.
2. dispersion/regroupement (*scatter/gather*) : permet de diffuser des blocs (distincts ou non) à toutes les tâches, ou de regrouper des blocs venant de toutes les tâches.

La troisième catégorie de communication collective est la réduction. La réduction (*reduce*) permet, quant à elle, d'effectuer un calcul sur un vecteur distribué, par exemple la recherche du minimum d'un tableau distribué sur toutes les tâches. Le résultat du calcul est ensuite communiqué à une ou plusieurs tâches.

4.2.1.2 Appel de procédure à distance

L'appel de procédure à distance permet d'exécuter des procédures dans le contexte d'une autre tâche. Ce type de communication permet donc de transférer à la fois une donnée et un traitant pour cette donnée. L'appel de procédure à distance fournit aussi des valeurs de retour résultant de l'exécution dans le contexte de la tâche distante. L'appel de procédure à distance se comporte comme un appel à une procédure classique sauf que son exécution a lieu dans le contexte d'une autre tâche, souvent un autre processus. L'intérêt du modèle de programmation par RPC est de transférer un flot de contrôle associé à des données.

4.2.2 Outils

Nous présentons ici deux outils (ou plutôt standards) logiciels pour la programmation mémoire distribuée. Ces outils sont caractéristiques du calcul hautes performances. Nous ne présentons volontairement pas les RPC car elles sont fastidieuses à utiliser et nécessitent souvent des mécanismes de synchronisation supplémentaire. Actuellement, les RPC ne sont plus utilisées dans le domaine du calcul scientifique hautes performances.

Parallel Virtual Machine (PVM)[55, 28] est un outil de développement d'applications mémoire distribuée en environnement hétérogène. Les performances ne sont pas le but de PVM, mais les services offerts contrebalancent ceci. PVM est apparue à une époque où il n'existait pas encore de standard de programmation parallèle si bien qu'elle a été adoptée très rapidement (car c'était l'unique outil dans son genre) pour devenir un standard de facto. PVM est très flexible et a été pensé en tant que système portable et interopérable. Les surcoûts induits par des caractéristiques telles que le support pour la dynamicité, ou la gestion des appels non-bloquants grèvent les performances suffisamment pour que l'usage de MPI se généralise.

Message Passing Interface (MPI) [18] est à l'heure actuelle l'unique standard pour le développement d'applications parallèles. Elle est le produit, à la différence de PVM, de réflexions entre des utilisateurs, des développeurs et des constructeurs. Ces derniers avaient en effet tendance à offrir avec leurs machines des outils de développement non portables. Ces réflexions ont commencé au début des années 90 pour aboutir à une première version vers 1993. MPI est un ensemble de spécifications et de fonctionnalités qui ne fait aucune supposition sur le matériel sous-jacent. Cette indépendance lui a assuré son succès : de multiples implémentations sont disponibles, aussi bien libres que commerciales. Les premières visent la portabilité et le support des configurations hétérogènes tandis que les secondes ont pour objectif une exploitation optimale du matériel. Cependant, la majorité des implémentations de cette interface exhibent des défauts en ce qui concerne la réactivité face aux événements réseaux. MPI est complémentaire de PVM au niveau des fonctionnalités [32] même si cette

dernière a cédé du terrain. MPI est une réussite car des programmes peuvent effectivement fonctionner sur des architectures très différentes tout en ayant de bonnes performances. MPI a cependant échoué sur un point, puisque deux implémentations différentes se révèlent incapables de communiquer (le *MPI Paradox*).

1. *MPICH*[31] est une implémentation libre de MPI. Elle présente l'avantage de disposer de nombreuses versions suivant le type de matériel réseau utilisé mais aussi d'un module mémoire partagée qui lui permet d'être utilisé pour des communications intranœuds. C'est donc une implémentation polyvalente adaptée aux architectures mémoire partagée, mémoire distribuée et hybride.
2. *MPI/LAM*[54] est l'autre implémentation libre de MPI la plus répandue à l'heure actuelle. Historiquement, il s'agit même d'une des toutes premières implémentations libres du standard. Tout comme MPICH, MPI/LAM dispose de différents modules destinés à gérer de nombreuses architectures réseaux et dispose aussi d'un module mémoire partagée. Cette implémentation dispose en outre d'un mécanisme de points de reprise utilisant la bibliothèque *Berkeley Lab Checkpoint/Restart* (BLCR)[63]. Ce mécanisme n'est pas pour autant utilisable sur toutes les architectures car il nécessite une modification du système pour être utilisable.
3. *Thread-Only MPI (TOMPI)*[17] est une implémentation partielle de MPI utilisant une bibliothèque de threads pour supporter des tâches MPI alors que les outils précédents utilisent un processus par tâche. Les tâches/threads ainsi créés peuvent migrer entre les processeurs d'un nœud si la bibliothèque de threads le permet. Ceci est souvent le cas comme nous allons le voir dans le modèle mémoire partagée. *Thread-Only MPI*, comme son nom l'indique, utilise uniquement des communications interthreads et ne dispose donc pas de mécanismes pour effectuer des communications internœuds.
4. *MPI-Lite*[53] est une implémentation de MPI utilisant les threads comme support pour les tâches MPI. Similairement à TOMPI, elle n'est pas destinée à fournir un support internœud mais peut proposer un équilibrage dynamique de charge.
5. *Adaptive MPI (AMPI)*[34, 35] est une implémentation de MPI qui utilise des processus légers comme processeurs virtuels. Les tâches MPI ainsi déployées peuvent migrer entre les processeurs d'un nœud multiprocesseur, mais aussi entre les nœuds pour réaliser un équilibrage dynamique de charge et ainsi redéployer un calcul. La bibliothèque de threads utilisée est CHARM++[1]. Cette bibliothèque est donc utilisable sur les trois types d'architectures de supercalculateurs et dispose en outre de méthode de points de reprise pour la tolérance aux pannes[64].

4.2.3 Discussion

Les différents outils représentatifs ayant déjà été présentés, nous allons maintenant voir quelles sont les caractéristiques de l'approche mémoire distribuée en terme de portabilité, de conception d'application performante et de facilité de programmation.

L'approche mémoire distribuée a pour intérêt d'être particulièrement portable. En effet, elle est adaptée aux architectures massivement parallèles car le modèle de programmation est directement calqué sur l'architecture de la machine. L'approche mémoire distribuée est également adaptable aux architectures mémoire partagée comme on a pu le voir aux travers des différents outils.

Néanmoins, cette approche ne tire pas complètement avantage de l'architecture mémoire partagée car elle impose une copie des données lors de la communication entre tâches. En effet, chaque tâche ayant son propre espace mémoire, les données présentes dans l'espace mémoire de l'émetteur doivent être recopiées dans l'espace mémoire du récepteur pour que ce dernier puisse y accéder. L'approche mémoire distribuée est évidemment adaptable sur les architectures hybrides qui sont, rappelons-le, une combinaison des deux architectures précédentes. Comme pour les architectures mémoire partagée, l'approche mémoire distribuée n'est généralement pas optimale pour les communications intranœuds qui sont induites par la composante mémoire partagée de l'architecture hybride. On peut donc dire que l'approche mémoire distribuée est portable car elle peut être adaptée à tout type d'architecture. Néanmoins, cette adaptation est souvent faite au détriment de certaines optimisations.

En plus de la portabilité, un autre intérêt de l'approche mémoire distribuée est la vision de la répartition des données qu'elle procure. En effet, à chaque instant on a la connaissance exacte de la localisation des données et de leur état vis-à-vis des autres tâches. Le programmeur a donc un contrôle total de la cohérence et du placement des données. Il est, par conséquent, plus aisé de mettre en œuvre des politiques d'optimisation des accès mémoire et de la gestion de la mémoire cache avec une telle connaissance. Cette vision de la répartition des données a néanmoins un coût en terme de conception des applications. En effet, la totalité de la complexité de la gestion de la cohérence des données reste à la charge de l'utilisateur. C'est donc sur lui que va reposer à la fois la justesse et l'efficacité de la cohérence de données.

Le désavantage majeur que l'on attribue au paradigme mémoire distribuée est la difficulté de mettre en œuvre un équilibrage dynamique de charge entre les processeurs. Cette limitation vient surtout des implémentations des bibliothèques de communication qui font la supposition d'un code parfaitement équilibré et n'autorisent pas d'avoir plus de tâches que de processeurs sans un énorme surcoût en terme de performances. Cette approche, dite de surcharge car on surcharge de tâches les processeurs, permet de facilement rééquilibrer la charge au sein d'un nœud de calcul par exemple.

L'approche mémoire distribuée est l'approche la plus utilisée en calcul scientifique hautes performances. Premièrement, elle est conceptuellement portable sur toutes les architectures. Deuxièmement, son succès fait qu'elle dispose d'un excellent support logiciel et matériel avec des implémentations de MPI[43] particulièrement optimisées pour certaines architectures. Cette approche est tellement utilisée que des solutions matérielles sont conçues avec pour objectif d'avoir de très bonnes performances avec une approche mémoire distribuée et communication par passage de messages. Nous pouvons citer les cartes réseau faites par Quadrics[51].

4.3 Modèle de programmation à mémoire partagée

La seconde approche utilisée en calcul hautes performances, est l'approche mémoire partagée. Comme pour l'approche mémoire distribuée, nous allons en présenter le concept, suivi d'une présentation de différents outils représentatifs, puis une discussion sur les apports de cette méthode.

4.3.1 Des communications transparentes

En parallélisme à mémoire partagée, la machine parallèle est vue comme un ensemble de processeurs qui accèdent à la même mémoire centrale. La mémoire peut être physiquement partagée

par plusieurs processeurs (SMP), ou une mémoire physiquement distribuée avec l'illusion d'une mémoire partagée assurée par des mécanismes matériels (NUMA) ou logiciels (DSM).

Définition 4.6: ***Distributed Shared Memory (DSM)*** **–** mécanisme logiciel permettant de donner l'illusion d'une mémoire unique à une mémoire physiquement distribuée.

Dans l'approche mémoire partagée, les échanges de données entre les différentes tâches se font simplement par des lectures/écritures vers la même zone de la mémoire. Cette simplicité apparente de communication masque les difficultés de consistance et de cohérence qui sont les difficultés majeures de la programmation mémoire partagée. En effet, il faut définir dans quelle mesure, les différentes tâches voient les modications apportées par les autres sur une zone mémoire partagée. En effet, il est rare que toutes les tâches voient réellement directement la même mémoire. Dans le cas des approches DSM et NUMA, la mémoire est physiquement distribuée et ce sont des mécanismes ajoutés qui assurent la vision unifiée de la mémoire souvent via l'utilisation de copies locales. Dans le cas des architectures SMP, les mémoires cache des processeurs vont constituer des copies locales des données. Même dans le cas d'une architecture monoprocesseur, les registres du processeur peuvent faire apparaître des copies locales de données. Ces copies locales conduisent à l'existence possible à un moment donné de plusieurs versions d'une donnée avec potentiellement des valeurs différentes. Les garanties sur la propagation des modifications de ces données en mémoire sont spécifiées par un modèle de cohérence. En ce qui concerne les structures de données complexes, il faut par exemple s'assurer qu'aucune tâche ne peut lire une structure donnée qui n'est que partiellement mise à jour du fait que les opérations complexes peuvent utiliser plusieurs cycles d'horloge. Si une partie de la cohérence des données peut être assurée par la matériel ou le compilateur, la totalité de la consistance des données reste à la charge du programmeur. La conception d'une application mémoire distribuée requiert l'utilisation de mots clés spécifiques comme `volatile` et de primitives de synchronisation comme les `mutex` chargés d'assurer la cohérence et la consistance des données.

La parallélisation mémoire partagée des applications repose très souvent sur l'utilisation de bibliothèques de multithreading qui permettent à plusieurs tâches de partager le même espace d'adressage. L'autre méthode est l'utilisation de segments de mémoire partagée qui permettent à plusieurs processus de partager de la mémoire. La parallélisation des applications n'étant pas toujours chose aisée, des langages et compilateurs spécialisés ont vu le jour pour faciliter la programmation parallèle à mémoire partagée. Dans ce cas, le parallélisme est géré directement dans le langage. Le compilateur génère alors un code parallèle, utilisant les méthodes citées, en suivant les indications données par le programmeur.

4.3.2 Outils

L'approche programmation parallèle par mémoire partagée est très liée à la notion de processus léger ou thread. Nous allons donc commencer la présentation des outils par les bibliothèques de threads. Nous verrons ensuite une autre approche pour réaliser des communications par mémoire partagée en utilisant les segments de mémoire partagée. Nous finirons par une présentation d'un outil d'aide à la parallélisation par mémoire partagée.

Bibliothèque de threads L'utilisation des bibliothèques de threads est l'approche classique pour concevoir une application mémoire partagée. Le niveau auquel est implémenté l'ordonnanceur constitue la caractéristique intrinsèque la plus importante des bibliothèques de threads.

Les threads peuvent être gérés dans une bibliothèque entièrement en espace utilisateur, ou bien au sein même du noyau avec l'aide éventuelle d'une bibliothèque en espace utilisateur.

1. *Bibliothèque de niveau utilisateur (voir* FIG. *4.1)* Les bibliothèques de threads de niveau utilisateur telle que GnuPth[21, 22] sont les premières apparues sur les systèmes d'exploitation, sans doute parce qu'il est plus facile de travailler en espace utilisateur que de modifier les systèmes d'exploitation. Malgré leurs limitations, comme l'impossibilité d'exploiter les machines multiprocesseurs et le blocage de tous les threads du processus lors d'un appel système bloquant, certaines bibliothèques de niveau utilisateur sont encore utilisées et développées. En effet, il est facile de les adapter à un besoin particulier et, n'ayant pas à dialoguer avec le système d'exploitation, elles sont très efficaces.

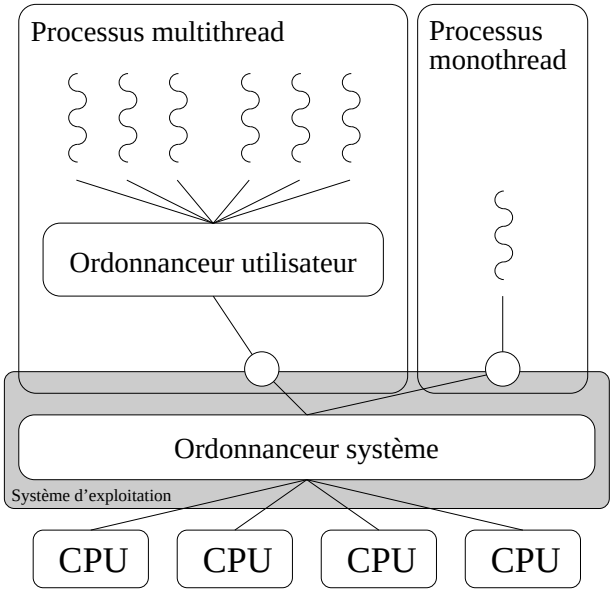


FIG. 4.1 – Bibliothèque de niveau utilisateur

2. *Bibliothèque de niveau noyau (voir* FIG. *4.2)* À l'opposé des bibliothèques de niveau utilisateur, les bibliothèques de niveau noyau (exemple NPTL[19],LinuxThreads[42]) sont généralement dédiées à un système particulier. Ces bibliothèques utilisent l'ordonnanceur du système pour gérer leurs processus légers ; une étroite intégration entre le système et la bibliothèque s'avère donc nécessaire. Les bibliothèques fournies avec les systèmes d'exploitation récents sont généralement des bibliothèques de niveau noyau (plus rarement à deux niveaux), ceci afin de permettre aux processus légers d'exploiter véritablement les machines multiprocesseurs.

3. *Bibliothèque mixtes ou à deux niveaux (voir* FIG. *4.3)* Comme les bibliothèques de niveau noyau, les bibliothèques mixtes(exemple NGPT[6], Solaris Thread) sont également capables d'exploiter les machines multiprocesseurs. De plus, elles ont l'avantage de garder un ordonnanceur en espace utilisateur et donc d'être efficaces et flexibles. Toutefois, la nécessité de faire coopérer deux ordonnanceurs rend l'écriture de telles bibliothèques plus difficile.

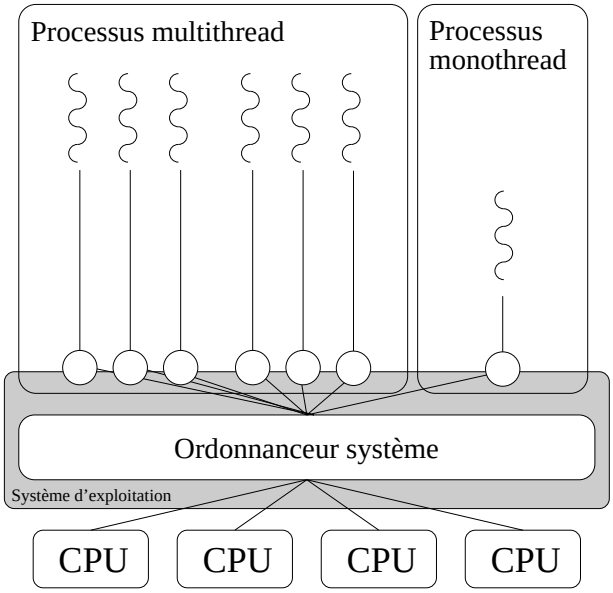


FIG. 4.2 – Bibliothèque de niveau noyau

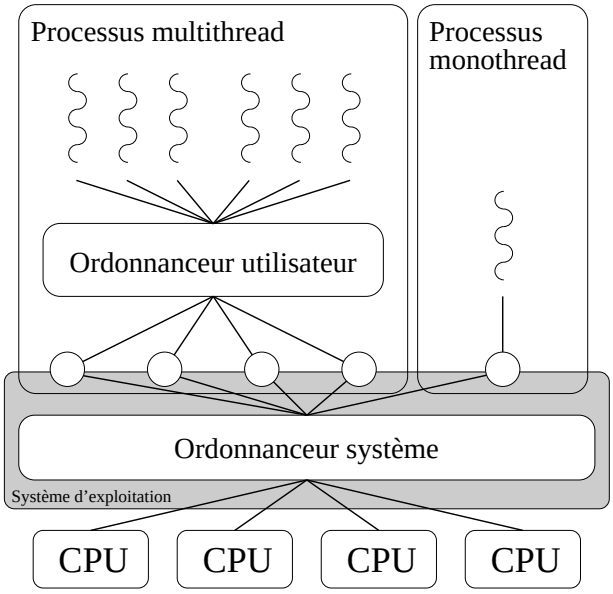


FIG. 4.3 – Bibliothèque à deux niveaux

Segment de mémoire partagée Les segments de mémoire partagée sont des zones mémoire qui sont accessibles en lecture/écriture par plusieurs processus. Ces segments permettent un schéma de communication similaire à celui des threads tout en mettant en œuvre des processus. Cette approche est très utilisée pour l'implémentation pour architecture mémoire partagée des approches mémoire distribuée que nous avons présentées précédemment (exemple MPI).

OpenMP[15] OpenMP est une extension du langage de programmation. Grâce à des directives insérées par le programmeur dans le code et grâce à une analyse des dépendances entre les données, un compilateur OpenMP est capable d'extraire du parallélisme des boucles de calcul. La parallélisation des portions de code indépendantes va être construite par le compilateur en utilisant des threads. OpenMP est donc une approche simplifiée pour utiliser une parallélisation de codes via des threads. Ce parallélisme utilise donc en interne une bibliothèque de threads. Cette bibliothèque est très souvent la bibliothèque de threads POSIX fournie avec le système d'exploitation. En terme de performance, les codes utilisant OpenMP pour la parallélisation sont donc très dépendants de la bibliothèque de threads et du compilateur OpenMP utilisé.

4.3.3 Discussion

Comme pour l'approche mémoire distribuée, nous allons maintenant détailler les caractéristiques de l'approche mémoire partagée en terme de portabilité, de conception d'application hautes performances, et enfin de facilité de programmation.

L'approche programmation mémoire partagée n'est pas particulièrement portable. En effet, elle nécessite d'être dans un environnement mémoire partagée que ce soit de manière physique (architectures SMP, NUMA) ou logicielle (DSM). Dans les approches DSM, les mécanismes de cohérence de données sont pris en charge de manière logicielle alors que dans le cas des SMP ce mécanisme est matériel. La principale difficulté rencontrée par les DSM est d'arriver à fournir un mécanisme complexe et performant de cohérence de données. Généralement, les approches DSM ne permettent pas de refléter parfaitement une architecture dotée de mémoire physiquement partagée. C'est une des raisons pour laquelle la portabilité d'une telle approche est limitée aux communications intranœuds. Le débat entre modèle de programmation mémoire distribuée via MPI et modèle de programmation mémoire partagée via DSM est très ancien. Pendant longtemps, on a dit "MPI, c'est plus efficace mais c'est plus difficile à programmer" alors que les "DSM, c'est moins efficace mais plus facile à programmer". Puis les DSM ont commencé à devenir de plus en plus performantes. Cette amélioration a été faite au prix de la facilité de programmation en truffant le code de primitives servant à contrôler le protocole de cohérence des données. Au final, les DSM efficaces sont devenues aussi difficiles à utiliser que les approches MPI. Ceci est sans doute une autre des raisons expliquant le déclin des DSM en calcul scientifique et donc de la portabilité des approches mémoire partagée.

La conception d'application hautes performances parallèle est intimement liée à la performance des communications. L'approche mémoire partagée dispose d'un atout majeur : la communication *zéro copie*. En effet, les données qui sont partagées en mémoire peuvent être lues par n'importe quelle tâche à tout moment sans le coût d'une recopie. Il faut nuancer ce phénomène par la nécessité d'utiliser des primitives de synchronisation (*mutex*, *spinlock*,...) pour maintenir la consistance des données en mémoire partagée. En effet, la manipulation des données en mémoire n'est pas atomique. Par exemple, l'initialisation d'un tableau nécessite plusieurs instructions machine. Le programmeur est donc tenu de mettre en œuvre des politiques de synchronisation pour permettre à chaque tâche du programme parallèle d'accéder à des données consistantes et non pas en cours de modification. La difficulté de création d'un code de calcul parallèle mémoire partagée efficace réside principalement dans la gestion des primitives de synchronisation.

Les primitives de synchronisation usuelles sont des opérations relativement coûteuses lors de l'exécution. En effet, ces opérations ont souvent recours à l'attente active ou requièrent un

changement de contexte. L'attente active est la synchronisation la plus simple à mettre en œuvre. Néanmoins, cette approche utilise des accès répétitifs à la mémoire tant que la condition n'est pas satisfaite. Cette approche monopolise donc un processeur durant la phase d'attente. Cette méthode est coûteuse en bande passante mémoire et en temps processeur. Il faut donc que les phases d'attente active (et donc les sections de code en exclusion mutuelle) soient très courtes pour ne pas trop pénaliser l'exécution. Les méthodes plus évoluées comme les *mutex* vont utiliser l'attente active pour des synchronisations internes et demander un changement de contexte en cas de non disponibilité des données. Ainsi, les mutex permettent de recouvrir les phases de synchronisation par du calcul. Comme nous venons de le voir, l'attente active est une méthode coûteuse mais un changement de contexte l'est encore plus. En effet, la tâche qui est stoppée en attente d'une donnée va, dans la majorité des cas, perdre toutes les données qu'elle a chargées dans la mémoire cache du processeur, ce qui constituera un coût supplémentaire lors du redémarrage de la tâche. A ce coût, il faut ajouter celui de l'ordonnanceur qui va être sollicité pour trouver une nouvelle tâche à exécuter ainsi que le coût du changement de contexte lui-même. En effet, il faut sauvegarder tous les registres processeurs du thread courant et restaurer ceux du thread qui va s'exécuter.

Comme nous venons de le voir, les méthodes de synchronisation vont faire des appels à l'ordonnanceur de tâches. Ce dernier doit donc être finement adapté à l'application qu'il doit ordonnancer pour limiter le coût d'ordonnancement et maximiser le recouvrement des phases de synchronisation. C'est pourquoi les bibliothèques standards de processus légers qui sont utilisées par les programmes OpenMP ne donnent pas entière satisfaction car elles sont trop généralistes. Ceci confère au programme OpenMP des performances souvent médiocres par rapport à celles escomptées.

Le dernier point qui pose problème aux applications utilisant la programmation mémoire partagée est directement lié aux architectures NUMA. Comme nous l'avons vu, la localité des données est primordiale sur ce type d'architecture. Or, dans le cas de la programmation mémoire partagée, il est difficile de savoir où placer les données qui vont être accédées par plusieurs processeurs. Souvent, il n'existe pas de placement qui permet un accès local systématique. Il faut donc veiller à bien placer toutes les tâches, grâce à un ordonnancement adapté, pour pouvoir placer les données en minimisant le nombre d'accès à des données non locales.

La programmation mémoire partagée est certes difficile à mettre en œuvre de façon efficace mais a l'énorme avantage de permettre l'équilibrage de charge qui est un problème majeur dans le cadre du calcul scientifique. En effet, comme toutes les données sont visibles de tous les processeurs, on peut dynamiquement déplacer les tâches d'un processeur sans aucune difficulté. Dans le cas des architectures NUMA, il faut néanmoins veiller à conserver une bonne répartition des données pour maintenir des performances optimales.

4.4 Modèle de programmation mixte

La dernière approche que nous allons présenter est un savant mélange des deux approches précédentes. Nous allons, dans un premier temps, détailler les fondements de cette approche. Ensuite, nous verrons quelques outils utilisés pour mettre en œuvre une approche mixte. Enfin, nous discuterons les apports de cette approche.

4.4.1 Tirer le meilleur parti de l'architecture sous-jacente

Le parallélisme mixte a pour objectif de cumuler les avantages des approches mémoire partagée et mémoire distribuée. L'approche vise souvent à se rapprocher au plus de l'architecture sous-jacente. Nous allons donc utiliser une approche mémoire partagée pour les communications intranœuds et l'approche mémoire distribuée pour les communications internœuds. Si cette approche est la plus souvent retenue, on peut aussi envisager d'utiliser une approche mixte au sein d'un nœud NUMA de grande taille pour faciliter le placement des données en mémoire. Sur une architecture Bull NovaScale, on peut par exemple utiliser l'approche mémoire partagée au sein de chaque QBB et une approche mémoire distribuée entre les QBBs.

4.4.2 Outils

L'outil le plus connu est en fait constitué de deux outils présentés précédemment : MPI et OpenMP. Les outils suivants intègrent les deux approches de parallélisme. L'utilisateur peut donc choisir pour chaque communication, l'approche adéquate.

MPI + OpenMP est une solution utilisant MPI pour les communications internœuds et OpenMP pour procéder à une parallélisation du code MPI s'exécutant sur un nœud multiprocesseur. Cette approche s'adresse prioritairement aux grappes homogènes de machines multiprocesseurs. Le problème de la collaboration d'outils différents se pose ici et le gain en performance n'est pas avéré ; seules certaines classes d'applications peuvent en tirer un avantage[12]. En effet, il est souvent impossible d'effectuer des communications MPI au sein des portions de code parallélisées avec OpenMP. Un code OpenMP + MPI est souvent une succession de portions de code parallélisées avec OpenMP suivies de communications MPI.

Nexus[26] Nexus est un environnement qui intègre la gestion du multithreading et des communications. Dès le départ, l'accent a été mis sur la portabilité et la haute performance. Cet environnement intègre des mécanismes de gestion de protocoles de communication multiples. Il est basé sur le paradigme de communication d'invocation à distance, appelé *Remote Service Request*. L'abstraction proposée est appelée pointeur global qui est incarné sous la forme d'un lien présentant un point de départ (ou *startpoint*) et un point d'arrivée (ou *endpoint*). Un lien est une connexion point-à-point unidirectionnelle sur laquelle circulent des requêtes allant du *startpoint* vers le *endpoint*. Un *endpoint* est lié à son contexte (processus), un *startpoint* peut être transféré d'un contexte à l'autre. L'établissement de la connexion est entièrement dynamique. Nexus est une bibliothèque particulièrement intéressante car elle offre une approche unifiée de communications quel que soit le mode réel de communication entre les tâches.

PM2[49] PM2 (*Parallel Multithreaded Machine*) est un environnement multithreadé de programmation parallèle en contexte mixte. PM2 est principalement dédié aux machines multiprocesseurs et aux grappes de multiprocesseurs. PM2 est composé de deux parties. Marcel est la sous-partie chargée de la gestion des threads dotée d'un ordonnanceur à deux niveaux. Madeleine est la sous-partie chargée des communications. Ces deux sous-parties interagissent finement pour fournir un ensemble particulièrement cohérent et efficace. Comme pour le duo MPI/OpenMP l'environnement PM2 propose des communications mémoire partagée entre les threads et mémoire distribuée entre les différents noeuds. PM2 dispose en outre d'un mécanisme de mémoire virtuellement partagée et de migration de threads entre les dif-

férents nœuds utilisés. PM2 est une bibliothèque particulièrement intéressante car elle permet de disposer d'un environnement optimisé pour effectuer des communications mémoire partagée et mémoire distribuée. La migration de threads proposée par PM2 vient compléter un mécanisme efficace de communication en permettant l'équilibrage dynamique de charge.

4.4.3 Discussion

L'approche mixte mémoire distribuée et mémoire partagée est très intéressante car elle permet de tirer parti de tous les avantages de la configuration physique de la machine. Néanmoins, elle souffre d'une forte complexité d'utilisation qui reste souvent à la charge du programmeur.

Dans cette approche, le programmeur doit lui-même décider du type de communication à utiliser pour chaque communication. Il doit faire ce choix en tenant compte de la topologie de l'architecture utilisée et du placement des tâches sur cette architecture.

L'antagonisme des approches mémoire distribuée/mémoire partagée fait qu'aucun outil ne peut masquer la complexité des communications entre tâches. Si l'on prend des outils de haut niveau comme la solution MPI+OpenMP, on obtient un parallélisme par étape où l'on ne peut pas réellement mélanger les parties OpenMP et MPI. Ceci est en grande partie dû au fait que ces bibliothèques ne sont pas totalement compatibles car les implantations de MPI ne sont pas toutes compatibles avec une exécution multithread (elles sont non réentrantes).

Si l'on utilise des bibliothèques de plus bas niveau comme Nexus ou PM2, la synchronisation et la cohérence des données doivent être entièrement gérées "à la main" au niveau même du code utilisateur.

Les approches mixtes permettent donc d'épouser au plus près l'architecture matérielle mais au prix de gros efforts dans la conception des applications.

4.5 Discussion

Dans le domaine du calcul scientifique hautes performances, il existe de nombreux outils souvent appelés intergiciels qui ont pour but de dissimuler la complexité de l'architecture au programmeur de codes de calcul. Ces outils apportent un certain nombre de fonctionnalités récapitulées dans le tableau 4.1.

Dans le domaine du calcul scientifique, le standard incontesté est MPI avec ses différentes implémentations. Néanmoins, on remarque, que les différentes implémentations sont loin de se valoir et nous ne parlerons pas ici des implémentations constructeur spécialisées pour telle ou telle architecture. Il est donc évident que l'utilisateur va devoir choisir l'implémentation qui est la plus adaptée à son architecture et à ses besoins. Le problème est que ce choix est remis en cause à chaque nouvelle architecture traitée. Le choix de l'implémentation va influer sur l'implémentation du code de calcul car, bien que MPI soit un standard, le comportement des nombreuses fonctions de communication peut fortement varier en terme de performance d'une implémentation à l'autre. Il peut donc s'avérer nécessaire de changer de politique de communication suivant l'implémentation de MPI utilisée. L'approche mémoire distribuée avec communication via MPI dispose donc d'implémentations extrêmement efficaces sur tout type d'architectures mais il n'existe pas d'implémentation performante sur un large spectre d'architectures.

Les outils de programmation mémoire partagée sont eux aussi victimes des mêmes problèmes. Prenons l'exemple des bibliothèques de threads qui sont très adaptées pour cette approche. La

Bibliothèque	Portabilité			Équilibrage			Optimisation			Robustesse
	Mémoire partagée	Mémoire distribuée	Mixte	Migration intranœud	Migration internœud	Notification déséquilibre	SMP	NUMA	Ordonnancement	Tolérance aux pannes
MPICH	√	√	√				√			
MPI/LAM	√	√	√				√			non portable
TOMPI	√			√			√			
MPI Lite	√						√			
AMPI	√	√	√	√	√		√		√	√
Thread	√			√			√			
Segment partagé	√			√			√			
OpenMP	√			√			√			
MPI+OpenMP	√	√	√	√			√			
Nexus	√	√	√	√			√			
PM2	√	√	√	√	√		√	√	√	

TAB. 4.1 – Comparaison des solutions proposées

multiplicité des implémentations du standard POSIX fait qu'un code doit être adapté pour chaque bibliothèque. Ce constat est encore plus marqué que pour MPI. En effet, le passage d'une bibliothèque préemptive à une bibliothèque non-préemptive peut considérablement changer le comportement de l'application. Cette modification de comportement peut même conduire à des blocages de l'application dans l'une des implémentations mais pas dans l'autre, en cas d'appels système bloquants par exemple. On note toutefois que l'approche mémoire partagée par thread peut permettre un contrôle très fin du comportement de l'application par l'utilisation d'ordonnanceur à deux niveaux. Cette approche peut donc permettre une optimisation accrue du comportement.

Dans le cadre des outils mixtes mémoire partagée et mémoire distribuée, la difficulté d'utilisation réside dans la complexité de programmation des applications sur ce modèle. Bien que cette approche permette de calquer le comportement pour obtenir un maximum de performance de l'architecture sous-jacente, la difficulté de programmation en fait une méthode peu utilisée dans le cadre du calcul scientifique.

Avec l'analyse des différents modèles de programmation, on constate que la difficulté de programmation est inversement proportionnelle à la facilité d'obtenir de bonnes performances. La solution idéale serait une approche offrant la simplicité d'utilisation d'une approche mémoire partagée avec la flexibilité et le contrôle offerts par les approches mixtes.

Ce chapitre a permis de détailler et comparer les trois modèles de programmation parallèle. Il a aussi permis de présenter des outils caractéristiques de chaque modèle de programmation.

Dans le cadre des travaux présentés ici, cette étude permet de déterminer les difficultés inhérentes à chaque modèle ce qui va nous aider dans le choix judicieux d'un modèle pour notre environnement. Il convient maintenant de déterminer quelles sont les contraintes imposées par le matériel présenté au chapitre 2 sur les modèles et plus généralement sur les méthodes de programmation.

Chapitre 5

Mise en évidence de la complexité de programmation des architectures actuelles

L'objectif de ce chapitre est de mettre en évidence l'influence de la complexité des architectures de supercalculateurs sur la conception d'applications hautes performances. Cette mise en évidence nous permet de déterminer quel sont les fonctionnalités qu'il faudra mettre en place pour la conception d'un environnement de programmation parallèle portable et efficace.

Pour cela, nous allons dans un premier temps décrire les dépendances des optimisations logicielles vis-à-vis de l'architecture sous-jacente. Ensuite, nous soulignerons la dépendance des outils à l'architecture. Puis, nous détaillerons la difficulté d'intégration des composants dans les approches mettant en œuvre plusieurs outils. Enfin, nous exposerons les contraintes de robustesse engendrées par les architectures actuelles.

5.1 Description de la dépendance des optimisations logicielles vis-à-vis de l'architecture

Pour souligner l'étroit lien entre les optimisations et les architectures pour lesquelles on veut optimiser un code, nous allons mettre en évidence les problèmes de performance que l'on rencontre sur les différentes architectures. Nous allons commencer par exposer les difficultés engendrées par la distribution de la mémoire sur les architectures à mémoire hiérarchique. Puis, nous détaillerons les effets de l'adjonction de cœur pour les architectures multicœurs. Ensuite, nous montrerons les difficultés introduites par les architectures massivement parallèles. Enfin, nous discuterons de la façon de concevoir une application parallèle à la fois portable et efficace.

5.1.1 Architectures NUMA

Les architectures NUMA ont une distribution physique de la mémoire qui forme une hiérarchie. Par exemple, l'architecture Bull NovaScale (voir 2.4.3) est une architecture NUMA où les QBBs constituent le dernier niveau de hiérarchie mémoire. Toutes les difficultés rencontrées sur les architectures NUMA proviennent de cette hiérarchie mémoire. Les difficultés majeures sont au

nombre de trois. La première est celle de la localité des données par rapport au processeur exécutant le programme. La deuxième est celle du faux partage de données entre threads. La dernière est celle de la contention au niveau de l'allocation/désallocation de la mémoire dans les processus multithreadés.

5.1.1.1 Problème de localité des données

Le principal problème des architectures NUMA est la localisation des données[24, 9]. En effet, une mauvaise distribution des données sur les différentes mémoires physiques peut avoir deux effets. Le premier est un temps d'accès aux données excessif. Le second est la contention au niveau des mécanismes de communication entre les mémoires physiques.

Sur architecture NUMA, un accès en mémoire distante est généralement plus coûteux qu'un accès en mémoire locale. Par exemple, ce surcoût est un facteur trois pour les architectures Bull NovaScale. La mauvaise localisation peut avoir plusieurs causes. La première cause de mauvais placement survient lorsque la mémoire en relation avec le processeur exécutant une tâche est pleine au moment de l'allocation. S'il n'existe aucun mécanisme permettant de libérer de la mémoire par migration ou libération de données, le système d'exploitation est contraint de faire une allocation distante. Une autre source de mauvaise localisation des données peut survenir suite à une migration de processus orchestrée par l'ordonnanceur du système d'exploitation. En effet, la majorité des ordonnanceurs ne tiennent pas compte des allocations mémoire effectuées. Il se peut donc qu'ils déplacent un processus sur un processeur où tous les accès mémoire de ce processus seront distants. Ce mauvais placement aura pour effet une perte de performance mais aussi un effet boule de neige. En effet, la mémoire utilisée par le processus occupe alors inutilement une mémoire distante. Cette occupation inutile peut engendrer une saturation d'un bloc mémoire physique, d'une QBB par exemple, alors que tous les processus font des accès distants à ce bloc de mémoire. Tous les nouveaux processus créés sur le groupe de processeurs dépendant de cette mémoire pleine vont alors faire des allocations distantes. Cette mauvaise gestion mémoire aura pour effet de pénaliser encore plus le système dans son ensemble.

Le deuxième effet de la mauvaise localisation des données est une saturation des mécanismes de communication entre les différentes mémoires physiques. Ces mécanismes ne sont généralement pas capables d'offrir une bande passante suffisante pour que tous les processeurs puissent faire des accès distants avec un débit comparable à un accès local. Une configuration où tous les accès sont distants a donc pour effet de saturer les mécanismes d'interconnexion. Cette saturation vient s'ajouter au surplus de latence résultant de l'accès distant et provoque un ralentissement global du système.

Une mauvaise politique d'allocation ou un ordonnancement non maîtrisé a des effets majeurs sur les performances d'une application s'exécutant sur architecture NUMA. Il convient donc de palier les lacunes du système d'exploitation pour obtenir une application performante.

5.1.1.2 Problème du faux partage

Le problème du faux partage apparaît avec l'exécution de programmes multithreadés sur tout type d'architecture[39, 56]. Néanmoins, il est particulièrement visible et pénalisant sur les architectures NUMA.

Le faux partage est lié au concept de pagination mémoire présent dans les systèmes d'exploitation actuels. La mémoire d'un processus est alors divisée en plusieurs pages. Chacune de ces pages

représente un ensemble contigu d'adresses virtuelles ou physiques. Le problème du faux partage survient lorsque deux threads se sont vus allouer des espaces mémoire sur une même page mémoire alors qu'ils ne désirent pas communiquer entre eux via ces espaces mémoire.

Sur une architecture NUMA, une allocation de ce type va créer des accès en mémoire distante si les deux threads ne s'exécutent pas sur des processeurs dépendant de la même mémoire locale. En effet, les données sont généralement réparties par page sur les différentes mémoires physiques. Donc, si les deux threads sont sur des processeurs ne partageant pas la même mémoire locale alors l'un des deux fera tous ses accès au segment alloué en mémoire distante. Les effets de cette allocation peuvent être amplifiés si le système d'exploitation dispose d'un mécanisme automatique de migration de pages pour favoriser la localité des données. En effet, ce mécanisme va essayer de diminuer les accès distants d'un thread en rapprochant les données accédées par ce thread via une migration qui aura pour effet d'éloigner les données d'un autre thread. Le mécanisme risque donc de multiplier les migrations de pages sans jamais pouvoir faire diminuer le nombre total d'accès distants.

La non-adaptation des bibliothèques d'allocation au multithreading a pour effet de disperser les données des threads. En effet, les segments alloués sont entremêlés donc il peut arriver que des allocations successives d'un thread ne soient pas successives en mémoire. Dans le cas, par exemple, d'une allocation de tableau par colonne successive, les différentes colonnes ne seront pas contiguës en mémoire et donc un parcours du tableau ne sera pas optimal dans l'utilisation des mémoires cache des processeurs. En effet, il sera possible que les colonnes des différents threads soient allouées en alternance impliquant qu'une ligne de cache soit utilisée par deux threads à la fois. Ce comportement peut induire un nombre élevé de défauts de cache sur les architectures multiprocesseurs.

Le phénomène de faux partage est directement lié à la bibliothèque d'allocation mémoire. Cette dernière ne tient pas compte de l'aspect multithreadé des applications et procède donc à une allocation pénalisante en terme de performance. Une solution à ce problème est l'utilisation de bibliothèques d'allocation spécialisée[10, 24, 11]. Ces bibliothèques d'allocation spécialisée comme HOARD vont utiliser une politique d'allocation qui assure qu'aucune page mémoire ne sera utilisée pour les allocations de plusieurs threads. Néanmoins, ces bibliothèques ne sont pas vraiment efficaces sur architecture NUMA car elles ne sont pas tout à fait adaptées.

5.1.1.3 Problème de la contention de l'allocation

Le problème de la contention à l'allocation est directement lié au nombre de processeurs de l'architecture. Ce phénomène est donc particulièrement visible sur les architectures NUMA. Cette contention touche toutes les applications multithreadées car, comme pour le faux partage précédent, la bibliothèque d'allocation mémoire ne tient pas compte de la présence de threads[10].

Les bibliothèques d'allocation mémoire sont généralement conçues à la base pour les applications monothreads et ont été étendues pour les applications multithreads. Cette extension ne tient pas compte du nombre, potentiellement élevé, de threads pouvant faire une demande d'allocation mémoire à un instant donné. Dès lors, l'allocation mémoire est un goulot d'étranglement supplémentaire pour les applications multithreads. En effet, les différents threads sont en concurrence à chaque allocation. Ce problème, qui pourrait être présent entre les processus au niveau du système d'exploitation, a été partiellement levé par l'utilisation du système de pagination qui va faire un effet tampon. Dès lors, la contention n'apparaît que lors de l'allocation d'une page mémoire supplémentaire.

On constate que le problème de la contention et celui du faux partage ont les mêmes causes. En effet, ces deux problèmes viennent du fait que les allocations dans les programmes multithreads sont totalement centralisées car les bibliothèques d'allocation mémoire n'ont pas tenu compte de la présence de threads dans l'application. Une solution commune aux deux problèmes est donc d'utiliser une bibliothèque d'allocation mémoire avec une politique par thread et non plus par processus.

La difficulté de programmation des architectures NUMA réside principalement dans l'allocation mémoire. En effet, le placement des données est crucial dans la conception d'application performante. C'est pourquoi il est nécessaire d'avoir une politique de localité très optimisée couplée à une interaction fine avec l'ordonnanceur pour conserver cette localité. La conservation de la localité peut donc entrer en conflit avec l'équilibrage dynamique de charge. Les problèmes rencontrés sur les architectures NUMA sont amplifiés lors de la conception d'une application multithread.

5.1.2 Puces multicœurs et/ou multithreads

Comme nous l'avons détaillé en 2.3.2, les nouvelles architectures de processeurs sont complexes et requièrent quelques précautions d'utilisation pour profiter pleinement de leur puissance. Toutes les difficultés que l'on rencontre sur ces nouvelles architectures proviennent de l'absence de politique de qualité de service sur les processeurs. Cette lacune se traduit dans les faits par la possibilité qu'un des threads, s'exécutant sur l'un des cœurs d'un processeur multicœur, monopolise toutes les ressources du processeur. Ainsi les autres cœurs n'auraient plus de ressources disponibles.

On peut distinguer deux ressources primordiales qui peuvent se tarir. La première est la mémoire cache dans le cas des processeurs multicœurs avec mémoire cache partagée ou dans le cas des processeurs multithreads. En ce qui concerne les processeurs multithreads, il se peut qu'un des threads utilise toute la mémoire cache du processeur. Il force ainsi le thread suivant à recharger systématiquement les lignes de cache qui lui sont nécessaires. Ce phénomène qui survient souvent lors des changements de contexte orchestrés par l'ordonnanceur du système d'exploitation peut devenir très pénalisant sur les processeurs multithreads car la fréquence de changement de thread est très élevée. Dans le cas des processeurs multicœurs, plusieurs threads peuvent se livrer à une concurrence pour remplir les lignes de cache. Ce comportement aura pour effet de produire une activité d'accès mémoire élevée qui peut ralentir le système entier. Il n'existe aucun moyen logiciel permettant de résoudre ce problème qui est purement matériel. Néanmoins, une répartition des threads sur la totalité des cœurs de la machine peut permettre de limiter les effets. En effet, si on associe, sur un processeur bicœur avec mémoire cache partagée, un thread faisant du calcul intensif et un second chargé d'effectuer des communications sur ce processeur alors les conflits de mémoire cache seront faibles. En revanche, si on place deux threads de calcul intensif sur ce processeur, les conflits seront nombreux.

La deuxième ressource pouvant se tarir est la bande passante mémoire. En effet, la bande passante mémoire est partagée entre tous les threads s'exécutant sur un processeur. Il convient donc de limiter au maximum les accès à la mémoire. Par exemple, un thread faisant de l'attente active peut complètement saturer le bus mémoire et ainsi ralentir l'évolution des threads qui cohabitent avec lui sur le processeur. Il convient donc de veiller à limiter et contrôler ce type d'approche pourtant courante dans les bibliothèques de communication comme les implémentations de MPI. Cette difficulté peut typiquement être résolue en programmant avec soin les procédures fortement

consommatrices en accès mémoire.

L'absence de politique fine de qualité de service sur les architectures multicœurs et/ou multithreads peut conduire à de forts ralentissements de l'exécution des applications à cause d'un mauvais partage des ressources. Dans le cadre du calcul scientifique, la ressource la plus critique est la mémoire, qu'elle soit cache ou centrale. Comme nous venons de le mettre en évidence, c'est aussi la ressource la plus vulnérable sur les processeurs multicœurs et/ou multithreads. C'est pourquoi il convient de tenir compte de cette particularité architecturale dans la conception des applications de calcul hautes performances.

5.1.3 Parallélisme massif

Les performances des codes de calcul parallèle sur des architectures massivement parallèles reposent quasi exclusivement sur les performances et surtout sur l'utilisation du réseau d'interconnexion.

Les réseaux d'interconnexion utilisés sur les supercalculateurs sont très différents les uns des autres. Ils peuvent utiliser le passage de messages comme dans les réseaux Quadrics ou bien encore des segments de mémoire partagée par les réseaux comme dans les réseaux SCI. Si l'on veut approcher les latences et débits optimaux de ces réseaux, il convient d'utiliser l'approche de communication pour laquelle ils ont été conçus. Une fois le protocole et la bibliothèque choisis, la taille même des messages peut être un critère de performance. Il se peut que l'on soit amené à subdiviser ou agréger des messages en fonction du débit et de la latence du réseau pour obtenir un temps de communication minimal. Ces contraintes forcent une spécialisation des applications pour chaque réseau que l'on compte utiliser.

Les difficultés rencontrées sur architectures massivement parallèles se situent au niveau de la portabilité des applications. Plus une application est optimisée pour une architecture, moins elle est portable. Plus précisément, il est nécessaire de concevoir des modules spécifiques à chaque type de réseau pour avoir une application à la fois portable et efficace.

5.1.4 Discussion

La conception d'une application de calcul hautes performances portable se doit de tenir compte de toutes les caractéristiques liées à l'architecture sous-jacente. Comme nous venons de le voir il peut s'agir d'utiliser des politiques d'allocation mémoire adaptées pour les architectures à mémoire hiérarchique. Ce peut être aussi l'utilisation de politiques d'ordonnancement ou une adaptation des codes de calcul aux architectures multicœurs et multithreads. Il faut de plus impérativement utiliser un protocole de communication bien adapté aux réseaux d'interconnexion. Enfin, il faut identifier les faiblesses des bibliothèques que l'on est amené à utiliser et les combler si besoin est. Il n'y a donc pas de méthode universelle pour la conception d'une application hautes performances. Les applications de calcul hautes performances, doivent nécessairement tenir compte de l'architecture et de l'environnement de travail pour être efficaces. Une application de calcul hautes performances portable est donc une application caméléon qui s'adapte à la hiérarchie mémoire, au type de processeur, au type de réseau d'interconnexion qu'elle va utiliser,...

5.2 Mise en évidence de la dépendance des outils logiciels existants à l'architecture

Les optimisations nécessaires à la mise en place d'outils performants sont intimement liées à l'architecture sous-jacente. Il découle de ceci une spécialisation des outils suivant les architectures, ce qui diminue la portabilité des applications construites à l'aide de ces outils. Dans cette section, nous allons tout d'abord souligner le fort lien qui unit les bibliothèques de communication à l'architecture. Ensuite, nous présenterons les caractéristiques qui lient les bibliothèques de threads à telle ou telle architecture.

5.2.1 Bibliothèque de communication

La dépendance des bibliothèques de communication à l'architecture sous-jacente est très forte. En effet, la conception d'une bibliothèque de communication performante nécessite de prendre en compte les contraintes matérielles concernant les communications internœuds mais aussi intranœuds.

La première dépendance forte se situe au niveau du matériel d'interconnexion des nœuds de calcul pour les architectures de type grappe. En effet, les protocoles de communication peuvent fortement varier d'une carte réseau à une autre pour une utilisation en calcul hautes performances. Une bibliothèque de communication efficace doit donc disposer d'un module de communication pour chaque réseau d'interconnexion. Ceci a pour effet de considérablement complexifier ces bibliothèques. Si l'on choisit un protocole portable comme TCP/IP, on constate très vite son manque de performance sur les réseaux d'interconnexion hautes performances utilisés dans les supercalculateurs de type grappe.

La seconde dépendance se situe au niveau des communications intranœuds. Il est nécessaire que la bibliothèque de communication tienne compte des spécificités de l'architecture au niveau de la mémoire. Il est en outre nécessaire que la bibliothèque de communication s'adapte aux limitations des processeurs, en terme de bande passante mémoire par exemple, pour ne pas conduire à une mauvaise utilisation des capacités des processeurs. Cette adaptation peut être mise en œuvre par une optimisation des mécanismes d'attente en cas de communication bloquante par exemple.

Ces dépendances conduisent à une spécialisation des bibliothèques de communication. Si l'on prend l'exemple de MPI, la majorité des constructeurs de supercalculateurs fournissent une implémentation du standard qui est ultra optimisée pour leur architecture. Néanmoins, cela ne garantit pas la portabilité des performances des applications construites sur MPI. En effet, suivant l'implémentation il peut être avantageux d'utiliser les envois de messages bloquants (`MPI_Send`) et sur d'autre les envois de messages bloquants synchronisés (`MPI_Ssend`). Ce type d'envoi peut, suivant les implémentations, nécessiter ou non des recopies mémoires ce qui va directement influer sur les performances. Il est donc courant d'avoir à modifier son code de calcul pour s'adapter à l'implémentation de MPI qui est optimisée pour le supercalculateur. De plus, chaque implémentation prend certaines libertés ou interprète différemment la norme, ce qui a pour effet de réduire la portabilité des applications construites sur les bibliothèques de communication pourtant standardisées.

5.2.2 Bibliothèque de threads

Les bibliothèques de threads doivent, elles aussi, s'adapter à l'architecture sous-jacente pour être performantes. Le choix même du type de bibliothèque est lié à l'architecture. Par exemple, il n'est pas envisageable d'utiliser une bibliothèque de niveau utilisateur sur une architecture multi-processeur car cette bibliothèque ne permet pas d'utiliser la totalité des processeurs de la machine.

Les bibliothèques de threads ne sont pas conçues pour le calcul hautes performances. Il est donc nécessaire de les modifier pour obtenir de bonnes performances. Ces modifications se situent principalement au niveau du placement des threads sur les processeurs. Il faut donc adapter les bibliothèques à chaque architecture.

5.3 Difficulté d'intégration des composants dans les approches hybrides

Les approches hybrides consistent à combiner différents outils en tirant parti des points forts de chacun de ces outils. Cette approche a donc pour avantage de construire une solution sur des outils éprouvés et connus pour leurs bonnes performances.

Cette approche, bien que très intéressante, se heurte à une complexité d'intégration augmentant avec le nombre d'outils que l'on veut faire cohabiter. Concrètement, associer plus de deux outils pour la création d'une bibliothèque de communication relève souvent de l'impossible. En effet, il est nécessaire de connaître dans le détail le fonctionnement interne de chaque composant pour pouvoir les intégrer efficacement. Si l'on prend l'exemple de l'intégration OpenMP + MPI, il faut tenir compte des différentes approches de OpenMP au niveau de la bibliothèque de threads mais aussi de la compatibilité plus ou moins bien supportée de MPI vis-à-vis des threads. Au final, il faut traiter quasiment au cas par cas chaque implémentation de MPI que l'on veut coupler avec OpenMP.

MPI + OpenMP est un exemple d'intégration mais toutes les bibliothèques standardisées posent un problème similaire. On peut citer les bibliothèques de threads, d'allocation et/ou de placement mémoire (comme les *memsets*, la bibliothèque de gestion des architectures NUMA *libnuma* ou les *cpusets*), ... Toutes ces bibliothèques cohabitent difficilement. Prenons un exemple concret : l'implémentation MPIBull utilise les *cpusets* pour contrôler le placement des processus MPI. Or, la bibliothèque *cpuset* est incompatible avec la bibliothèque *libnuma* qui est présente dans certaines bibliothèques de threads comme Marcel. Ceci implique une incompatibilité entre les bibliothèques MPIBull et Marcel.

5.4 Une contrainte de robustesse

Les supercalculateurs actuels ont un nombre de composants très élevé. En effet, le nombre de nœuds de calcul est élevé. Il en va de même pour le nombre de processeurs, le nombre de bancs de mémoire, ... Si chacun de ces composants pris séparément est très fiable, aucun n'est à l'abri d'une panne. Le nombre de ces dysfonctionnements est de l'ordre d'un incident tous les deux jours pour une grappe de grande taille comme la machine TERA1 du CEA/DAM[29]. Il existe plusieurs solutions à ce problème.

La première solution pour palier à ces incidents est la duplication des ressources matérielles. Cette duplication a un coût et ne peut être entreprise que pour certaines ressources. Par exemple, il est intéressant de dupliquer les connexions réseau. En cas de fonctionnement normal, la bande

passante est accrue et apporte donc un surcroît de performance. En cas de défaillance d'un composant réseau, l'interconnexion continue de fonctionner. En revanche, il est peu utile et très coûteux de dupliquer un nœud de calcul pour la seule éventualité d'une défaillance mémoire ou processeur. Il n'est donc pas possible de palier à la totalité des dysfonctionnements matériels ou logiciels.

Malheureusement, certains calculs complexes peuvent nécessiter des temps de calcul très longs de l'ordre de la semaine ou du mois et nécessiter un grand nombre de nœuds de calcul. Ces applications sont donc très vulnérables aux pannes étant donné la quantité de matériel utilisée et la durée d'exploitation. Pour ces applications, il est nécessaire de mettre en place un mécanisme de tolérance aux pannes qui permettrait d'éviter la perte du calcul effectué en cas de défaillance.

La conception des supercalculateurs les a rendus très vulnérables aux pannes matérielles ou logicielles. Pour limiter les conséquences de ce problème sur la production de résultats, les applications doivent se doter d'un mécanisme de tolérance aux pannes.

Ce chapitre a permis de mettre en évidence l'influence de la complexité des architectures sur la conception des applications. En effet, certains types d'architectures fréquents dans la conception des supercalculateurs nécessitent des optimisations spécifiques pour en tirer la quintessence. Nous avons aussi mis en évidence la forte dépendance des outils existants à l'architecture pour laquelle ils ont été conçus. Puis, nous avons montré la difficulté d'intégration de ces composants ultra optimisés pour la conception d'une application portable et performante. Enfin, nous avons montré la nécessité d'un mécanisme de tolérance aux pannes. Forts de ces remarques, nous allons pouvoir mettre en place un modèle de programmation efficace et robuste pour notre bibliothèque de communication : MPC.

Deuxième partie

La bibliothèque MPC : MultiProcessor Communications

Chapitre 6

Choix d'un modèle de programmation pour MPC

L'objectif de ce chapitre est de déterminer un modèle de programmation et une API pour notre bibliothèque MPC. Ce modèle de programmation devra, bien sûr, répondre aux exigences du calcul scientifique hautes performances.

Le chapitre est subdivisé en trois parties. La première partie va définir les caractéristiques d'une bibliothèque de communication en calcul hautes performances. Dans une deuxième partie, ces caractéristiques vont permettre de choisir le modèle de programmation parallèle qui va être utilisé par MPC. Dans la troisième partie, nous allons présenter l'interface qui sera présente dans MPC.

6.1 Mise en évidence des caractéristiques d'une bibliothèque de communication

La bibliothèque MPC doit fournir tous les services habituellement proposés par les bibliothèques de communication existantes, mais doit également combler les défauts de ces dernières. Pour cela, nous nous sommes fixés quatre objectifs majeurs qui caractérisent MPC. Le premier objectif est la portabilité, le second est l'équilibrage de charge, le troisième est la haute performance et le dernier est la tolérance aux pannes.

6.1.1 Haute portabilité : du monoprocesseur à la grappe de multiprocesseur

Comme nous l'avons vu en 2.4, les architectures des supercalculateurs sont très variées. Néanmoins, l'évolution récente des supercalculateurs permet d'affirmer que généralement ces machines sont parallèles. Plus précisément, ce sont des architectures de type grappe de nœuds de calcul. Ces nœuds ont des architectures très variées. Actuellement, ces architectures varient de la simple architecture monoprocesseur jusqu'à des gros nœuds de type NUMA à 2048 processeurs. Cet écart va sans doute s'amplifier avec les nouvelles architectures de processeurs qui vont permettre d'envisager de gros nœuds NUMA de processeurs multicœurs multithreads. Notre objectif étant de fournir une bibliothèque capable de s'adapter efficacement à la majorité des supercalculateurs d'aujourd'hui et de demain. Il est donc nécessaire d'identifier un modèle de programmation parallèle *unique* dont l'implémentation efficace permettra de fournir de bonnes performances

quelle que soit l'architecture matérielle. En effet, il n'est pas raisonnable d'appréhender la diversité matérielle au cas par cas en définissant un modèle de programmation par architecture. La complexité et la gestion de ces architectures doivent nécessairement être transparentes pour l'utilisateur final.

6.1.2 Équilibrage de charge

Comme nous l'avons plusieurs fois mentionné, l'équilibrage dynamique de charge est primordial pour un grand nombre d'applications. Par exemple, les applications de type raffinement adaptatifs de maillages (AMR) sont particulièrement déséquilibrées par nature. Ces applications nécessitent donc une politique d'équilibrage dynamique de charge pour obtenir une utilisation maximale des processeurs. La majorité des solutions logicielles mémoire distribuée existantes n'offrent pas de solution à ce problème d'équilibrage dynamique de charge. En ce qui concerne les solutions mémoire partagée, un équilibrage de charge par migration peut être orchestré par l'ordonnanceur système ou l'ordonnanceur de la bibliothèque de threads dans le cas des bibliothèques mixtes[1]. Les solutions offertes au concepteur de codes de calcul déséquilibrés se résument souvent à ne pas avoir de politique d'équilibrage de charge ou bien d'avoir une politique automatique non maîtrisée. La troisième solution consistant à modifier le code de calcul pour mettre en place un équilibrage de charge applicatif, par exemple par migration de mailles entre sous-domaines. Cette dernière méthode est souvent très complexe à mettre en place.

Un des objectifs que nous nous fixons est donc de fournir des outils permettant, aux programmeurs des applications, de mettre en œuvre le plus facilement possible, une politique d'équilibrage de charge maîtrisée. Ces outils vont nécessairement influer sur le modèle de programmation choisi ainsi que la méthode d'équilibrage de charge. En effet, le modèle de programmation devra offrir des propriétés permettant l'équilibrage indépendamment de l'application utilisant la bibliothèque.

Bien que l'équilibrage de charge soit primordial pour certaines applications, il ne doit pas perturber l'exécution des codes équilibrés. Cette condition n'est pas réellement vérifiée par les ordonnanceurs en contexte mémoire distribuée. Les outils ne devront donc pas influer sur les codes équilibrés. Cette distinction étant connue uniquement par l'application, il sera nécessaire d'avoir une politique conjointe entre l'application et la bibliothèque de communication pour mettre en œuvre une politique d'équilibrage de charge efficace et transparente.

6.1.3 Hautes performances

La bibliothèque de communication MPC se destinant au calcul hautes performances, il est donc nécessaire qu'elle offre de très bonnes performances en tirant au mieux parti de l'architecture sous-jacente. Pour évaluer ces performances, nous allons nous comparer à des bibliothèques constructeur connues pour leur extrême optimisation envers l'architecture matérielle sous-jacente. Ces hautes performances devront passer par une adaptation dynamique de la bibliothèque MPC à l'architecture sous-jacente. Un autre objectif majeur est donc, en plus de bonnes performances, une portabilité de ces performances car nous désirons concevoir une bibliothèque portable comme nous l'avons mentionné en 6.1.1. Il faudra donc tirer les leçons des difficultés mises en évidence au chapitre 5 pour la conception de notre bibliothèque.

[1] Les bibliothèques de niveau utilisateur sont hors contexte car elles n'utilisent qu'un seul processeur.

6.1.4 Tolérance aux pannes

Le dernier objectif, et non des moindres, est la tolérance aux pannes. Comme nous l'avons mis en évidence au chapitre 5 un mécanisme de tolérance aux pannes est indispensable à toute application de calcul hautes performances. C'est pourquoi MPC devra fournir un mécanisme de tolérance aux pannes transparent qui lui permettra de reprendre un calcul qui a été interrompu suite à une panne matérielle. Ce mécanisme devra donc permettre à l'utilisateur de sauvegarder des points de reprise en cours de calcul. Les dits points de reprise pourront être utilisés ultérieurement pour reprendre le calcul en cas de panne matérielle.

Comme nous avons pu le voir au chapitre 4 et dans le tableau 4.1, il n'existe pas d'outils de programmation parallèle qui remplissent tous les objectifs que nous nous sommes fixés. Cela justifie donc la conception d'une nouvelle bibliothèque de communication qui sera adaptée à un large spectre de codes de calcul tout en étant performante et portable sur un grand nombre d'architectures.

6.2 Notre choix : la programmation par passage de messages

Parmi les trois paradigmes de programmation parallèle, nous avons choisi le paradigme mémoire distribuée car c'est celui qui permettra d'offrir tous les services requis tout en offrant de bonnes performances.

6.2.1 Motivation

Nous allons détailler ici les différents éléments qui ont motivé le choix d'une approche mémoire distribuée par passage de messages. Pour cela, nous verrons les caractéristiques de l'approche qui sont particulièrement intéressantes pour élaborer une bibliothèque portable. Ensuite, nous détaillerons les avantages de l'approche pour la mise en œuvre d'une politique d'équilibrage de charge. Enfin, nous mettrons en évidence les bonnes propriétés de l'approche qui faciliteront la mise en place des optimisations nécessaires à l'élaboration d'une bibliothèque hautes performances.

L'approche mémoire distribuée est sans aucun doute l'approche la plus polyvalente en terme d'adaptation aux architectures matérielles. Tout d'abord, cette approche est en parfaite adéquation avec les architectures mémoire distribuée. En effet, l'approche est calquée sur l'architecture et permet donc une mise en œuvre efficace. Ensuite, l'approche mémoire distribuée est également bien adaptée aux architectures à mémoire partagée car il est relativement simple de mettre en œuvre une politique de communication intertâches efficace. L'approche mémoire distribuée étant largement répandue, nous disposons d'un grand nombre de bibliothèques de communication mémoire distribuée efficaces sur architectures à mémoire distribuée comme à mémoire partagée. Enfin, l'approche mémoire distribuée est bien adaptée aux architectures mixtes qui, rappelons-le, sont des architectures combinant mémoire partagée et mémoire distribuée. L'approche étant en adéquation avec chacune des architectures composant les architectures mixtes, l'approche choisie l'est donc aussi avec cette dernière architecture. On peut donc dire que l'approche mémoire distribuée nous permet de tenir notre premier objectif de haute portabilité.

Si tous les modèles de programmation parallèle permettent la mise en place d'un équilibrage de charge, l'approche mémoire distribuée par passage de messages facilite grandement cette mise

en place. En effet, cette approche permet d'avoir des tâches relativement indépendantes car chacune dispose de son propre ensemble de données. Ces données peuvent être modifiées en cours d'exécution, mais cette modification requiert un accord préalable de la tâche recevant la modification (voir 4.2 pour le détail des communications par passage de messages). L'indépendance des données qui caractérise l'approche choisie permet de réaliser "une image" des tâches en cours d'exécution du calcul. Cette image qui va regrouper l'ensemble des données d'une tâche ainsi que son état dans l'exécution du programme va permettre de réaliser aisément l'équilibrage de charge par migration de tâches mais aussi la mise en place d'une politique de tolérance aux pannes. L'objectif d'équilibrage de charge ainsi que celui de tolérance aux pannes peuvent donc être remplis en utilisant une approche mémoire distribuée par passage de messages.

L'objectif de hautes performances peut, a priori, être rempli par n'importe quelle approche mais est grandement facilité par l'indépendance des tâches qu'apporte le modèle de programmation mémoire distribuée. En effet, ce modèle permet d'avoir une vision très précise de l'ensemble des données mises en œuvre par l'application. Cette connaissance précise va, par exemple, permettre de faire de nombreuses optimisations sur les architectures NUMA où le placement des données est primordial. Ensuite, toutes les opérations coûteuses de synchronisation sont présentes uniquement dans la bibliothèque de communication. Nous disposons donc de tous les éléments et du contrôle nécessaire à la mise en place des optimisations qui réduisent les pénalités inhérentes aux synchronisations.

L'approche mémoire distribuée avec communications par passage de messages est très appropriée aux objectifs que nous nous sommes fixés. Cette approche ne pose donc aucun obstacle à la mise en place d'une bibliothèque portable, dotée d'une politique d'équilibrage de charge et offrant les performances nécessaires au calcul hautes performances.

6.2.2 Et les autres modèles ?

Comme nous venons de le voir, la programmation mémoire distribuée avec communications par passage de messages est particulièrement bien adaptée pour satisfaire nos objectifs. Nous allons maintenant détailler les raisons qui font que les approches mémoire partagée et mixtes ne sont pas en adéquation parfaite avec nos objectifs.

L'approche mémoire partagée est une approche fonctionnant exclusivement sur architectures mémoire partagée ou sur les architectures qui disposent d'un mécanisme matériel ou logiciel rendant l'illusion d'une mémoire partagée. Cette restriction sur le matériel entre directement en conflit avec notre objectif de portabilité. S'il est vrai que l'on peut contourner ce problème en utilisant une mémoire virtuellement partagée logicielle, nous avons vu en 4.3 que cette solution n'est pas particulièrement efficace en terme de performance. L'approche mémoire partagée n'est donc pas en adéquation avec l'objectif de portabilité dans le cas où l'on utilise pas de DSM et avec l'objectif de performance en cas d'utilisation de DSM. Néanmoins, l'approche mémoire partagée permet une mise en œuvre aisée d'une politique d'équilibrage de charge. En effet, il suffit dans ce cas de migrer les threads/processus entre les processeurs pour équilibrer la charge. Cette facilité est à nuancer par le souci de performance qui impose, dans le cas des architectures NUMA par exemple, une politique de migration des données entre les mémoires physiques pour maximiser le taux d'accès local à la mémoire. L'objectif de tolérance aux pannes peut toujours être mis en place grâce à un point de synchronisation global qui va forcer l'arrêt de toutes les tâches et procéder à la création d'une image globale qui va contenir les données de toutes les tâches. En ce qui concerne la reprise de l'application, il suffit de relire les données et redémarrer les tâches. Nous pouvons

donc dire que la conjonction de la portabilité et de la performance est problématique dans le cas de la programmation mémoire partagée.

L'approche mixte mémoire partagée/mémoire distribuée met en œuvre les qualités et défauts des deux approches qui la compose. Le principal avantage de l'approche mixte est sa portabilité. En effet, de par sa conception, cette approche est adaptée aux architectures mémoire partagée comme aux architectures mémoire distribuée. Cette approche valide donc l'objectif de portabilité. La majorité des difficultés concerne les deux autres objectifs. Tout d'abord, l'équilibrage de charge est particulièrement difficile à mettre en œuvre du fait de la non-homogénéité de l'approche. En effet, il serait particulièrement difficile et coûteux de déplacer une tâche a d'un ensemble de tâches A communicant par mémoire partagée à un autre ensemble B sachant que les communications allant de A à B utilisent l'approche mémoire distribuée. Cette difficulté provient du fait qu'il est impossible de déterminer a priori quelle sont les données spécifiques à a qu'il faudrait migrer et celles que a partage avec les autres tâches de A. Le modèle de programmation mixte seul ne permet donc pas de garantir une politique d'équilibrage de charge entre les groupes de tâches communiquant par mémoire distribuée. L'approche mixte entre donc en conflit avec l'objectif d'équilibrage de charge bien qu'elle offre des possibilités d'équilibrage aisé au sein des ensembles utilisant l'approche mémoire partagée. De plus, l'optimisation d'une application de cette complexité sera particulièrement difficile du fait du peu d'information qu'apporte le modèle de programmation. Le modèle peut donc entrer en conflit avec notre objectif de hautes performances. En ce qui concerne l'objectif de tolérance aux pannes, l'approche mixte pose exactement les mêmes problèmes que l'approche mémoire partagée pour la création des points de sauvegarde des ensembles de tâches communiquant par mémoire partagée.

Comme nous venons de le voir, aucune approche n'est écartée pour une incompatibilité avec les objectifs de haute portabilité et d'équilibrage de charges. En effet, il est toujours possible d'implémenter une bibliothèque qui fournirait les objectifs fixés. Néanmoins, la complexité de mise en œuvre risque de rejaillir sur les performances et sur la facilité d'utilisation de cette bibliothèque. Cette complexité fait que les approches mémoire partagée et mixtes ne peuvent être retenues comme modèle de programmation pour MPC.

6.3 Interface de programmation choisie

Comme nous avons choisi l'approche mémoire distribuée avec communications par passage de messages, notre interface de programmation va être proche du standard le plus répandu : MPI. En effet, une interface proche de celle de MPI permettra aux utilisateurs de pouvoir migrer facilement de MPI à MPC. Ce transfert peut même être envisagé à l'aide d'un pré-processeur à l'image de ceux utilisés dans AMPI ou TOMPI qui permettent l'utilisation transparente pour l'utilisateur d'une approche multithreadée. Le choix de ne pas fournir une interface identique à celle de MPI est délibéré. En effet, le modèle de gestion des tâches que nous allons mettre en œuvre diffère de celui de MPI. Dès lors, une interface identique aurait pour effet de faire oublier que MPC n'est pas une implémentation de MPI mais bien une autre approche. C'est pourquoi nous fournissons uniquement une interface comparable dans le but de faciliter la migration d'un système à l'autre.

L'interface représentative de MPC se décompose en quatre grandes parties :

– les communications point à point qui sont principalement composées du couple de fonctions `MPC_Send`/`MPC_Recv` pour les communications bloquantes ainsi que le couple de fonctions

`MPC_Isend`/`MPC_Irecv` pour les communications asynchrones ;
- les communications collectives principalement composées de `MPC_Barrier` pour la barrière, `MPC_Reduce` pour les réductions et `MPC_Broadcast` pour les diffusions ;
- la primitive de migration de tâche `MPC_Migrate` ;
- la primitive de création de points de reprise `MPC_Checkpoint`.

L'annexe A décrit l'interface complète de MPC et illustre la transformation nécessaire pour convertir un code utilisant MPI en un code utilisant MPC.

Ce chapitre a permis de déterminer un modèle de programmation qui est en adéquation avec l'ensemble des contraintes relatives au calcul scientifique hautes performances. Ce chapitre a aussi permis de mettre en évidence la nécessité d'une nouvelle bibliothèque de communication pour le calcul scientifique hautes performances. La définition de l'interface de programmation, qui a découlé du choix du modèle, est la dernière étape avant la mise en place des modèles d'exécution de notre bibliothèque.

Chapitre 7

Modèles d'exécution de MPC

L'objectif de ce chapitre est de spécifier les modèles d'exécution de toutes les composantes de notre bibliothèque. Maintenant que le choix du modèle de programmation utilisateur a été fait, il convient de déterminer le fonctionnement interne et la sémantique de MPC.

Pour cela, nous allons définir le modèle d'exécution des tâches qui va fortement influencer tous les autres modèles. Ensuite, nous détaillerons notre politique de gestion mémoire, puis la gestion des communications point à point et collectives. Par la suite, nous présenterons le modèle de tolérance aux pannes de notre bibliothèque. Enfin, nous détaillerons les outils du modèle d'équilibrage dynamique de charge.

7.1 Choix du modèle de gestion des tâches : le multithreading

Le modèle de gestion des tâches mis en place dans MPC est un modèle multithread. Nous assimilons chaque tâche issue de l'approche mémoire distribuée à un thread. Ces threads communiqueront donc entre eux par passage de messages comme l'auraient fait des tâches MPI. Pour démontrer la validité de ce choix, nous allons tout d'abord montrer que l'approche multithread est en accord avec le modèle de programmation que nous avons choisi précédemment. Ensuite, nous détaillerons le type de multithreading que nous allons utiliser. Enfin, nous détaillerons les différentes optimisations que nous permet l'approche multithread. Ces optimisations ont par ailleurs motivé et guidé notre choix.

7.1.1 Programmation par passage de messages et multithreading

Le modèle de programmation interne de la bibliothèque est antagoniste au modèle de programmation utilisateur. L'un est un modèle mémoire distribuée tandis que l'autre est un modèle mémoire partagée. Néanmoins, la programmation multithread mémoire partagée permet de construire une bibliothèque de communication mémoire distribuée avec passage de messages efficace.

L'approche multithread permet de mettre en place des communications entre threads par les mécanismes de communication par mémoire partagée. Il sera donc possible de mettre en place une solution portable de communication par mémoire partagée. En effet, les solutions non multithreads de mémoire partagée comme les segments de mémoire partagée souffrent de grosses limitations en terme de portabilité car ces fonctionnalités dépendent du système d'exploitation.

Tous les systèmes d'exploitation n'ont pas la même politique, en particulier, en ce qui concerne la taille maximale des segments. Dès lors, il est nécessaire de tenir compte de ces contraintes pour concevoir une bibliothèque portable. De plus, la taille du segment étant fixée par avance, ceci rajoute une contrainte supplémentaire dans la conception des communications. Enfin, une telle approche nécessite des recopies mémoire supplémentaires pour placer les messages dans le segment de mémoire partagée puis pour les recopier du segment de mémoire partagée vers la destination. Il est donc plus facile de gérer les communications intranœuds en mémoire partagée qu'en mémoire distribuée. En ce qui concerne les communications internœuds, nous sommes dépendants des technologies réseaux et le modèle d'exécution n'influe que peu sur cette partie des communications.

Pour implémenter une approche mémoire distribuée, il est nécessaire de définir le fonctionnement des tâches au sein de la bibliothèque de threads. L'idée de base est relativement simple et consiste à utiliser un thread différent pour chaque tâche. Il y aura donc autant de threads que de tâches. La communication entre les tâches sera en fait une communication entre threads, pour la partie intranœud. Cette approche n'est pas remise en cause pour les communications internœuds. Il suffit dans ce cas de multiplexer les messages des threads sur le réseau d'interconnexion pour mettre en place la communication internœud. La figure 7.1 illustre cette approche de fonctionnement.

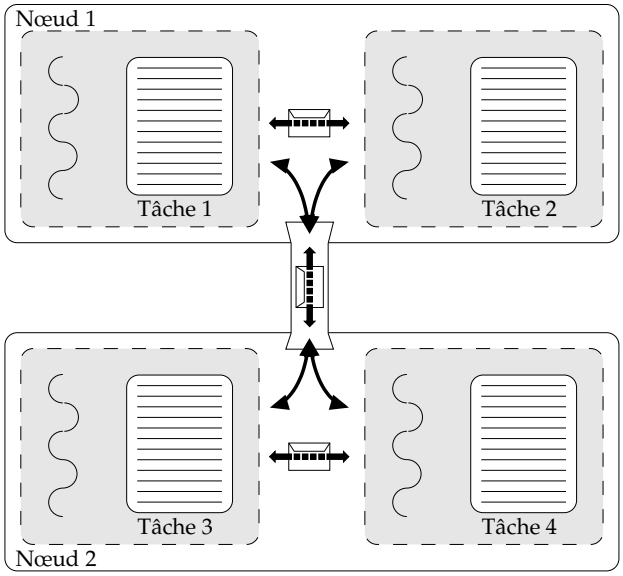


FIG. 7.1 – Illustration du modèle d'implémentation

L'approche mémoire distribuée avec communications par passage de messages peut tout à fait être implémentée à l'aide de threads, les communications étant faites par mémoire partagée. Nous pouvons ainsi profiter des avantages de la mémoire partagée et faire des communications entre tâches qui ne nécessitent pas de recopies superflues. Il reste à définir quel type de multithreading nous allons utiliser.

7.1.2 Le multithreading mixte : un contrôle total

Le choix du type de multithreading s'est porté sur un multithreading mixte. En effet, nous avons vu lors des chapitres précédents qu'il était nécessaire de bien maîtriser l'ordonnancement et le placement des threads pour pouvoir tirer parti des architectures actuelles des supercalculateurs. Nous allons présenter en détail le principe d'un ordonnanceur à deux niveaux comparable à celui que nous utilisons dans MPC. Puis nous décrirons les avantages et inconvénients de cette approche.

7.1.2.1 Ordonnancement à deux niveaux

Un ordonnanceur à deux niveaux est un ordonnanceur double. Il est constitué de l'ordonnanceur du système d'exploitation et d'un ordonnanceur de niveau utilisateur qui fait partie de la bibliothèque de threads.

L'ordonnanceur du système d'exploitation ne permet pas, en général, d'obtenir un comportement optimum des threads et ce en particulier en calcul scientifique. Néanmoins il est capable de gérer tous les appels système bloquants ainsi que les signaux mais aussi une certaine réactivité des processus. Dans un contexte de calcul scientifique, ces événements sont marginaux par rapport aux calculs en eux-mêmes et la réactivité n'est pas une priorité. La priorité est le temps d'exécution du calcul. C'est pourquoi nous préférons utiliser un ordonnanceur de niveau utilisateur.

Comme nous l'avons vu, les ordonnanceurs de niveau utilisateur uniquement (bibliothèques de niveau utilisateur) ne sont pas appropriés pour gérer des machines multiprocesseurs. Nous ne pouvons donc pas utiliser un ordonnanceur utilisateur seul pour palier aux inadaptations de l'ordonnanceur du système d'exploitation. Nous allons donc concevoir une application dotée d'un ordonnanceur à deux niveaux. Ceci va nous permettre de "court-circuiter" l'ordonnanceur du système en le déplaçant en espace utilisateur.

Le premier niveau de l'ordonnanceur est celui du système d'exploitation qui va être en charge de la gestion complexe des appels système et autres gestions bas niveau pour lesquelles il est optimisé. Ensuite, nous utilisons un ordonnanceur utilisateur qui va utiliser un ensemble de threads de niveau noyau encore appelés processeurs virtuels et va ordonnancer les threads utilisateur sur ces threads noyau. Ainsi, avec autant de threads noyau que de processeurs, la partie utilisateur de l'ordonnanceur à deux niveaux va totalement contrôler le placement des threads ainsi que leur ordonnancement. De plus, l'ordonnanceur utilisateur va gérer la fréquence des changements de contexte et donc limiter les défauts de cache engendrés par l'interruption d'un thread en phase de calcul intensif. Cet ordonnanceur pourra en outre être modifié pour implémenter des fonctions spécifiques aux bibliothèques de communication. Ces modifications à implémenter au niveau du système d'exploitation seraient beaucoup trop lourdes en terme de portabilité et de maintenance tout en risquant de ralentir le système d'exploitation pour les autres applications. La figure 7.2 illustre le fonctionnement d'un ordonnanceur à deux niveaux couplé à une bibliothèque de communication.

Un ordonnanceur à deux niveaux est donc bien approprié à une utilisation en calcul scientifique mais a quelques inconvénients. Le premier est la complexité de mise en œuvre d'un tel ordonnanceur. La seconde contrainte est commune à tous les mécanismes d'ordonnancement de niveau utilisateur préemptifs. En effet, il est nécessaire de protéger les appels système et appels à des bibliothèques qui ne sont réentrants que vis-à-vis des threads POSIX. Effectivement, il ne faut

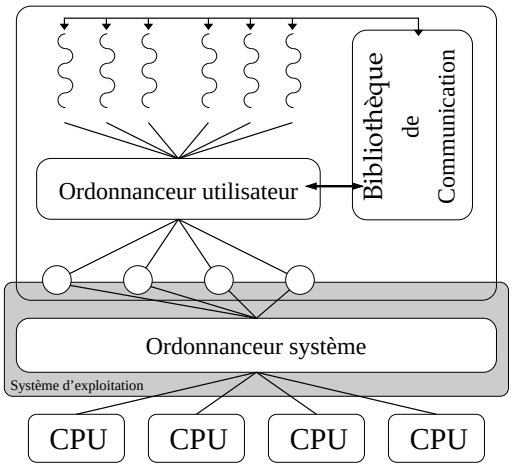


FIG. 7.2 – Illustration de l'ordonnanceur de MPC

pas effectuer de changements de contexte entre des threads utilisateur lorsqu'un thread est dans un de ces appels car ce changement est invisible pour ces bibliothèques ou appels système. Ce changement de contexte produirait donc un comportement incohérent. Une solution appropriée au calcul scientifique est de concevoir des ordonnanceurs non préemptifs. La préemption est une notion très utile pour les applications interactives mais pas pour le calcul scientifique. En effet, interrompre un thread en cours de calcul pour le remplacer par un autre thread va généralement avoir pour effet de faire perdre toutes les données préchargées dans la mémoire cache. De plus, comme les codes de calcul ne sont en général pas interactifs, la préemption n'a que peu d'intérêt. Une approche non préemptive est donc totalement appropriée. Enfin, les ordonnanceurs utilisateur ont une limitation en cas d'appel système bloquant. En effet, dans ce cas le thread noyau qui est utilisé au moment de l'appel bloquant est endormi et il n'y a donc plus que $n-1$ threads noyau qui peuvent être utilisés sur les n processeurs de la machine. Il y a donc une perte de parallélisme. Dans le contexte du calcul scientifique, ces appels sont relativement rares donc leur impact est négligeable. En effet, les appels bloquants les plus utilisés sont les entrées/sorties, appels qui sont faiblement parallélisés et qui auraient donc été pénalisants quelles que soient l'approche et la bibliothèque choisies.

L'ordonnanceur à deux niveaux est sans doute l'approche de bibliothèque de threads la plus appropriée au calcul scientifique car elle permet de bien tirer parti de l'architecture multiprocesseur ainsi que de contrôler les points critiques que sont le placement et l'ordonnancement des threads.

7.1.2.2 Type d'ordonnanceur utilisateur choisi

La distinction majeure entre deux implémentations de bibliothèques de threads à deux niveaux réside dans l'ordonnanceur utilisateur. Les bibliothèques de threads actuelles ont des ordonnanceurs généralistes qui ne font aucune supposition sur le comportement des threads. Dans notre cas, le modèle de programmation mémoire distribuée avec communications par passage de messages nous fournit un schéma d'exécution assez précis. De plus, nous avons pris le parti de coupler l'ordonnanceur à la bibliothèque de communication pour obtenir un meilleur ordonnancement. On peut donc dire que tous les ordonnanceurs existants auraient nécessité de lourdes modifica-

tions pour être parfaitement adaptés à nos besoins. C'est pourquoi nous avons choisi de concevoir un nouvel ordonnanceur spécifiquement adapté à nos besoins. Les paragraphes suivants vont décrire cet ordonnanceur.

Pour limiter les communications et la contention entre les différents processeurs lors de phases de réordonnancement, nous avons choisi de mettre en place un ordonnanceur à files multiples. Pour limiter encore plus les communications entre les processeurs, nous avons choisi de ne pas implémenter de mécanisme de priorités. En effet, la présence de priorités implique que chaque processeur doit chercher à exécuter le thread de plus haute priorité. Ceci nécessite de parcourir l'ensemble des threads prêts de chaque processeur pour déterminer quel est le thread à exécuter. Cette approche est très pénalisante sur une architecture de type NUMA. De plus, les priorités ne sont pas souvent utilisées dans le calcul scientifique hautes performances. En effet, tous les threads de notre modèle sont principalement des threads de calcul ; ils ont donc tous la même priorité. Avec un modèle d'ordonnanceur files multiples sans priorité, il est possible, sauf cas de famine de threads prêts sur un processeur, de ne jamais avoir besoin de partager des données d'ordonnancement entre les différents processeurs virtuels.

L'ordonnanceur utilisateur doit permettre de maintenir le placement des threads sur les processeurs en accord avec le placement des données en mémoire. Pour cela, nous avons choisi de mettre en place un ordonnanceur à files multiples sans vol de tâches. En effet, le vol de tâches a pour effet d'accroître la migration des threads et ce, particulièrement au niveau du réveil des opérations collectives. En effet, à cet instant, la quasi totalité des processeurs sont sans travail et l'on va réveiller les threads au fur et à mesure. On assiste donc à une redistribution des threads sur les processeurs car chaque processeur essaye de voler les threads des voisins tant qu'il ne peut réveiller les siens.

Néanmoins, comme tous les codes de calcul ne sont pas équilibrés, nous avons tout de même choisi de mettre en place un mécanisme d'équilibrage permettant de migrer volontairement des threads d'un processeur à un autre. Cette migration ne pourra être faite que si l'application le décide. Reporter les mécanismes d'équilibrage de charge au niveau utilisateur a pour avantage principal de permettre à l'utilisateur, qui connaît bien son application, de faire un équilibrage optimal. En effet, avec cette approche, il devient possible de contrôler finement le placement des threads sur les processeurs. Pour compléter ce mécanisme de migration de threads, nous avons mis en place un mécanisme de détection du déséquilibrage de charge entre tous les processeurs. Ceci permet donc de mettre en place au niveau utilisateur une politique d'équilibrage de charge globale basée soit sur la connaissance de l'évolution du comportement du code de calcul, soit plus traditionnellement en utilisant une détection automatique des problèmes de déséquilibre couplée à un mécanisme de rééquilibrage automatique.

De plus, l'ordonnanceur va être non-préemptif. En effet, la préemption d'une tâche en cours de calcul a pour effet de lui faire perdre les données préchargées dans la mémoire cache. Or, les applications utilisant des primitives de passage de messages font régulièrement des appels bloquants pour les communications. C'est lors de ces appels que les tâches vont passer la main. Le choix d'un ordonnanceur non-préemptif a pour effet d'interdire les techniques d'attente active au sein des tâches. Néanmoins, ce type d'approche, particulièrement inefficace sur architectures hiérarchiques, pourra aisément être remplacé par une approche basée sur la scrutation que nous détaillerons plus tard.

Ce type d'ordonnanceur associé avec un placement "au plus près" des données utilisées par chaque processeur est généralement particulièrement efficace sur tous types d'architectures mul-

tiprocesseurs. De plus, cet ordonnanceur contribue à réutiliser au mieux la mémoire cache des processeurs en exécutant toujours le même ensemble de threads. Cette caractéristique va fortement influer sur les temps d'exécution des programmes.

7.1.3 Stratégies de programmation et d'exécution

Cette section a pour objectif de décrire les différentes optimisations que nous permet une approche multithread avec un ordonnanceur à deux niveaux que nous venons de décrire.

7.1.3.1 Surcharge

Le mécanisme de surcharge ou "Domaine Overloading" est un principe général d'optimisation des codes de calcul SPMD qui consiste à utiliser plus de tâches que de processeurs.

La méthode de surcharge permet en premier lieu le recouvrement aisé des communications. En effet, le surnombre de tâches, par rapport au nombre de processeurs, permet de recouvrir les temps d'attente d'une tâche par l'exécution d'une autre tâche. Cette méthode de recouvrement des communications peut être mise en place sans aucune modification du schéma de communication de l'application. Ce type d'optimisation n'est que très rarement supporté par les implémentations de MPI par exemple qui ne sont efficaces qu'avec une tâche par processeur. Il existe d'autres méthodes de recouvrement des communications par l'utilisation de communications asynchrones. Néanmoins, ces approches nécessitent une modification de l'application et sont fortement liées à l'implémentation de MPI. Par exemple, certaines implémentations de *MPICH* ne disposent pas de mécanismes permettant de faire progresser automatiquement les communications asynchrones. Ainsi, c'est à l'utilisateur de faire des appels spécifiques pour réellement recouvrir la latence des communications par du calcul.

Un second avantage de la surcharge est de permettre une meilleure utilisation des mémoires cache. En effet, dans le cas d'un code de calcul SPMD, la taille des données utilisées par une tâche est proportionnelle au nombre de tâches. Un nombre suffisant de tâches peut donc permettre de réduire la taille des données de chaque tâche jusqu'à ce que ces données puissent en majorité tenir en mémoire cache. Ceci a pour effet de considérablement augmenter les performances. Les optimisations usuelles pour l'utilisation de la mémoire cache, sur les grosses applications de calcul scientifique, sont très souvent coûteuses et spécifiques au processeur. Avec cette approche, il suffit de choisir le bon nombre de tâches pour un problème donné pour bénéficier des gains par effet de cache. Cette approche est particulièrement intéressante dans un contexte multithread. En effet, le faible coût d'un changement de contexte associé à une politique d'ordonnancement efficace permet de limiter les surcoûts d'ordonnancement de la méthode. En revanche, l'approche multithread ne permet pas de réduire le surcoût en communication et en occupation mémoire.

Le troisième avantage de la surcharge est de permettre la mise en place d'un mécanisme d'équilibrage de charge à gros grains qui sera transparent pour l'application. En effet, un nombre de threads supérieur au nombre de processeurs permet de répartir la charge de calcul en distribuant les threads sur les processeurs. Ainsi, on peut mettre en place un équilibrage de charge qui ne nécessite aucune modification au niveau de l'application ; il suffit d'avoir plus de threads que de processeurs.

Néanmoins cette approche n'est pas parfaite car elle va souvent augmenter le volume des communications dans le cas des applications SPMD. En effet, en augmentant le nombre de sous-domaines, on va accroître le nombre de mailles fantômes à synchroniser à chaque pas de temps.

Bien qu'il y ait un surcoût en terme de communications, nous verrons dans le chapitre 9.4.4 que l'on parvient tout de même à obtenir des gains significatifs.

7.1.3.2 Équilibrage de charge

Dans le cadre d'une bibliothèque de communication destinée aux supercalculateurs, l'équilibrage de charge doit permettre de réguler la charge sur la totalité des processeurs du supercalculateur. Plutôt que de proposer une politique d'équilibrage automatique, qui ne serait pas optimisée pour chaque application, nous avons décidé de fournir à l'utilisateur tous les moyens lui permettant de mettre en œuvre un équilibrage efficace pour son application. Ces moyens se décomposent en deux parties. La première consiste à fournir des informations concernant l'utilisation des processeurs. La seconde consiste à fournir des moyens pour migrer les tâches entre les processeurs.

La détection du déséquilibre de charge entre les processeurs est une condition très importante pour permettre une bonne répartition des charges. La charge d'un processeur est difficile à évaluer. En effet, il faut déterminer quelles sont les opérations à prendre en compte pour déterminer la charge. Une approche simpliste consiste à évaluer la charge de calcul de chaque tâche (au sens de l'approche programmation mémoire distribuée) et de concevoir la charge d'un processeur comme la somme des charges de toutes les tâches qu'il exécute. Cette évaluation est intéressante pour peu que la part des communications ne soit pas trop importante. En effet, dans ce dernier cas, nous risquons de sous-évaluer la charge d'un processeur et ainsi risquer de ralentir l'exécution globale de l'application. Le cas des communications est un exemple mais on peut tenir le même raisonnement pour les entrées/sorties par exemple. Il est donc difficile de déterminer tous les paramètres qui peuvent influer sur la charge d'un processeur. C'est pourquoi, nous avons choisi de prendre le problème à l'envers. Nous allons chercher à évaluer le taux d'inactivité d'un processeur plutôt que son taux d'activité. Pour cela, nous allons utiliser les propriétés des bibliothèques de threads utilisateur qui disposent d'un thread *idle*. Le thread *idle* est le thread qui est exécuté lorsque plus aucun autre thread n'est prêt à être exécuté. En calculant la fréquence d'ordonnancement du thread *idle* nous avons donc un aperçu du taux d'inactivité du processeur et par conséquent une évaluation de son taux d'activité. L'information ainsi calculée tient compte de toutes les opérations faites directement par les tâches mais aussi de toutes les opérations implicites comme les opérations de communication utilisant une politique de scrutation que nous détaillerons plus tard. Cette information sur la répartition des charges processeur est présentée au niveau de l'utilisateur sous la forme d'un tableau indiquant le taux d'occupation de chaque processeur.

Une fois un défaut de répartition de charge mis en évidence, il est nécessaire de fournir à l'utilisateur un moyen de résoudre ce problème. En contexte multithread et au sein d'un même processus, il est particulièrement aisé de procéder à une migration de threads et donc de tâches entre les processeurs utilisés par ce processus. L'énorme avantage de cette approche est qu'il est possible de mettre en œuvre une politique globale d'équilibrage de tous les processeurs utilisés via une allocation mémoire adaptée, qui sera présentée en 7.2, couplée à un suivi des communications présenté en 7.5. En effet, l'approche multithread mixte permet de connaître l'état précis d'un thread via son contexte qui est représenté par l'ensemble des registres processeurs et sa pile d'exécution. Une gestion mémoire appropriée permet, quant à elle, d'identifier les données utilisées. Comme nous utilisons une approche mémoire distribuée pour les tâches, les threads ne font donc pas de communication mémoire partagée si l'on exclut l'implémentation interne à MPC des communications intertâches. Il suffit alors de migrer tout le contexte d'une tâche d'un processeur à l'autre, pour mettre en place une politique d'équilibrage de charge au niveau utilisateur.

L'approche multithread mixte permet la mise en œuvre d'une politique d'équilibrage de charge entièrement utilisateur et ne nécessitant donc pas de modification du système d'exploitation. Le contrôle de l'ordonnancement des threads, permet en outre d'étendre l'équilibrage de charge à la totalité des processeurs utilisés via des mécanismes internes à MPC de suivi des communications et de gestion mémoire.

7.1.3.3 Politique de scrutation

La mise en place d'une politique de scrutation efficace est très importante dans la conception d'une bibliothèque de communication. En effet, les bibliothèques de communication sont souvent amenées à mettre en place des méthodes d'attente de satisfaction d'une condition. On peut citer l'attente engendrée par l'utilisation de communications bloquantes. Sans support de l'ordonnanceur, ces phases d'attente ne peuvent être traitées que par de l'attente active. Or, cette dernière méthode est à prohiber dans un contexte général car elle provoque une monopolisation d'un processeur pour la seule vérification de cette condition. En contexte de surcharge, l'approche d'attente active va tout simplement utiliser des ressources qui pourraient l'être pour faire du calcul .

La présence en espace utilisateur de l'ordonnanceur de threads permet d'intégrer une politique de scrutation à cet ordonnanceur. Cette intégration a pour objectif de supprimer l'aspect consommation de ressources de l'attente active tout en conservant la bonne réactivité de cette méthode. En effet, si l'on confie à l'ordonnanceur la charge de la scrutation, alors le thread en attente d'événement peut être mis dans l'état bloqué et il ne consommera donc plus aucune ressource. L'ordonnanceur, qui a une vision globale de tous les threads, peut alors effectuer l'opération de scrutation périodiquement en fonction de l'activité des autres threads. Ainsi, l'opération de scrutation peut être plus fréquente lorsque le nombre de threads prêts diminue ce qui va favoriser, par exemple, l'avancement des communications pour augmenter le nombre des threads prêts en satisfaisant les requêtes en attente. Dans le cas où l'ordonnanceur dispose de nombreux threads prêts à faire du calcul, cette fréquence peut être diminuée pour favoriser l'avancement des calculs.

L'intégration d'une politique de scrutation dans l'ordonnanceur de threads nécessite la conception d'un serveur de scrutation. Les threads en attente d'événement vont alors enregistrer leur requête de scrutation sur le serveur et passer dans l'état bloqué. Régulièrement, l'ordonnanceur de threads va vérifier les requêtes qui sont satisfaites et réveiller les threads en attente sur ces requêtes. Il va aussi exécuter les fonctions de scrutation des requêtes insatisfaites.

La présence d'une politique de scrutation au sein de l'ordonnanceur de threads permet de faciliter la création d'une bibliothèque de communication efficace.

L'utilisation d'une bibliothèque de threads permet de mettre en place une politique efficace de surcharge de tâches par rapport au nombre de processeurs. Le fait que cette bibliothèque de threads soit de type mixte permet la mise en place d'outils pour l'équilibrage de charge mais aussi d'une politique de scrutation qui va faciliter l'élaboration de la partie communication de la bibliothèque MPC.

7.2 Choix du modèle de gestion mémoire

La gestion de la localité des données en mémoire est un point crucial dans l'obtention de bonnes performances sur les architectures de type NUMA en particulier, mais aussi sur toute

architecture multiprocesseur/multicœur/multithread. Ces phénomènes de localité mémoire sont par ailleurs exacerbés par l'utilisation de threads. Comme l'approche que nous avons choisie est multithread, nous allons détailler la gestion mémoire mise en œuvre pour obtenir de bonnes performances sur les architectures de supercalculateurs.

7.2.1 Approche choisie

Pour obtenir de bonnes performances sur architecture à mémoire hiérarchique, il convient de favoriser la localité des données. Comme le modèle de programmation utilisateur est une approche mémoire distribuée avec passage de messages, les différentes tâches ne partagent pas de données en mémoire. Une allocation mémoire optimale est donc une allocation qui va placer les données de chaque tâche dans la mémoire locale au processeur qui l'exécute.

L'approche multithread utilisée en interne pour propulser les différentes tâches nous permet à tout instant de connaître exactement la répartition des tâches sur les processeurs. Nous allons donc coupler l'allocation aux informations que nous fournit l'ordonnanceur de threads pour définir dans quelle partie de la mémoire il faut allouer les données.

Si l'approche multithread nous permet de connaître précisément la localisation des tâches, elle apporte de nombreux problèmes concernant l'allocation mémoire. En premier lieu, le problème du faux partage que nous avons mentionné au 5.1.1.2. Pour résoudre ce problème il faut que les threads ne partagent pas des pages mémoire inutilement. Pour cela, il faut utiliser une allocation *thread-aware* comme la bibliothèque HOARD[10] qui va distinguer les allocations par thread. Néanmoins, ce type d'allocateur ne tient pas compte des aspects NUMA des architectures et va redistribuer les pages libérées indifféremment à n'importe quel thread. Ceci aura pour effet de détériorer la localité des données. Il convient donc, en plus d'une allocation évitant le faux partage, de mettre en œuvre une réutilisation des pages libérées qui tienne compte des aspects NUMA. C'est pour cela que nous avons choisi une approche d'allocation par thread. Chaque thread dispose d'un tas local sur lequel il va allouer ses données. Ainsi, avec une granularité au niveau de la page mémoire, le faux partage est éliminé. De plus, les pages mémoire utilisées étant spécifiques à chaque thread, la réutilisation des pages libérées conserve la localité mémoire. La figure 7.3 illustre le modèle de gestion mémoire.

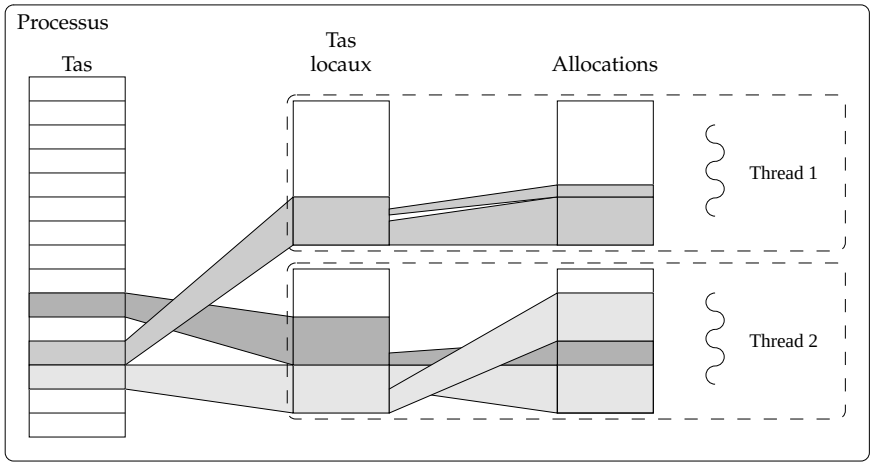


FIG. 7.3 – Illustration du modèle de gestion mémoire

L'autre problème mis en évidence lors de l'utilisation des allocateurs mémoire standards est une contention au niveau de l'allocation. Cette contention est levée avec l'approche choisie. En effet, chaque thread étant propriétaire de son propre tas, il n'y a pas de contention car un thread ne peut allouer sur le tas d'un autre. Néanmoins, il est possible qu'un thread ait à libérer des données allouées par un autre thread. Dans ce cas, il y aura prise d'un verrou pour la libération. Ce cas de figure, bien que possible en général, est impossible dans notre approche car les tâches propulsées par les threads, adoptent un paradigme mémoire distribuée et donc ne partagent pas de mémoire. Les threads vont donc opérer des allocations/désallocations sans jamais créer de contention autre que lors de l'appel système pour l'allocation réelle des pages mémoire au sein du processus.

Bien que cette approche permette de résoudre les problèmes de faux partage et de contention, elle n'est pas parfaite. En effet, la réutilisation des pages libérées est un problème majeur de cette approche. Comme le montre la figure 7.3 dans le cas du thread 2, les tas locaux ne sont pas forcément contigus. Cette caractéristique fait que même si le thread 2 libérait toute sa mémoire, il serait néanmoins impossible qu'il réutilise les deux pages mémoire libérées pour satisfaire une allocation de plus d'une page mémoire. En effet, les deux pages mémoire n'ont pas des adresses contiguës en mémoire virtuelle. Le thread 2 sera donc obligé d'allouer un bloc supplémentaire de deux pages mémoire. Certains comportements comme une suite d'appels à la fonction `realloc` avec des tailles demandées croissantes illustrent cet exemple et s'avèrent fortement consommateur en mémoire avec cette gestion mémoire. Un moyen simple de limiter ce phénomène consiste par exemple à remettre en commun les pages libérées par les différents threads au sein d'un même niveau de hiérarchie mémoire (une QBB par exemple). Ainsi, l'on aura plus de pages et donc plus de pages consécutives susceptibles d'être fusionnées tout en conservant la localité des données. Cette amélioration sera encore meilleure si on alloue des blocs bien plus gros qu'une page au niveau du système et que l'on anticipe la taille du plus gros bloc à allouer. Il est à noter que sur de nombreux systèmes, l'allocation des pages mémoire a réellement lieu au moment de la première tentative de lecture ou d'écriture. Allouer un bloc de taille supérieure n'est donc pas synonyme de surconsommation de mémoire physique mais seulement de surconsommation d'espace d'adressage virtuel. Cette dernière n'est pas forcément un handicap si l'espace d'adressage est sur 64 bits par exemple. Cette optimisation sur la taille des blocs peut être faite à un niveau intermédiaire entre le tas du processus et les tas locaux aux threads. Ainsi, on limite la surconsommation mémoire et on évite le fractionnement.

L'approche que nous avons choisie permet donc de lever la contention, mais aussi de lever le problème du faux partage et enfin permet la mise en place d'une allocation locale des données. De plus, cette approche permet d'envisager une migration des pages mémoire en cas de migration de threads. Cette migration de pages est néanmoins conditionnée à l'existence d'une telle primitive dans le système d'exploitation.

7.2.2 Migration interprocessus

Les outils de migration de tâches pour l'équilibrage de charge nécessitent de pouvoir déplacer à tout instant les données d'un thread d'un processus à l'autre. L'approche de gestion mémoire nous permet de facilement identifier quelles sont les données allouées dynamiquement utilisées par un thread. L'approche mémoire distribuée et l'utilisation de threads interdisent l'utilisation de variables globales sans passer par des clés comme celles proposées dans l'API POSIX avec les `pthread_*specific`. La totalité des données utilisées par un thread est donc allouée dynami-

quement ou dans la pile du thread. Un simple déplacement de ces données et des valeurs des registres permettent de migrer un thread ; il reste néanmoins à gérer les communications de ce thread.

Pour mettre en place la migration, il faut que les adresses virtuelles des données à migrer ne soient pas utilisées dans le processus cible. Pour assurer cette propriété, il suffit que les allocations utilisent des adresses différentes entre tous les processus. Une méthode pour obtenir cette garantie est de vérifier, à chaque allocation, que la plage de nouvelles adresses n'est pas déjà utilisée dans un autre processus. La primitive *isomalloc* de PM2 est une implémentation de cette méthode[8]. Une approche plus simple consiste à diviser l'espace d'adressage total en portions égales pour chaque processus. Cette méthode est particulièrement adaptée aux architectures avec un espace d'adressage important comme sur les architectures 64 bits. Cette dernière méthode permet une allocation sans communication entre les différents processus mais au prix d'une vision statique de la répartition des adresses utilisables par thread et donc par là même une limitation du nombre d'adresses utilisables au sein de chaque processus. Cette approche est aussi particulièrement bien adaptée au modèle de gestion mémoire que nous utilisons.

Cette méthode permet la migration transparente de threads entre les processus utilisés. Pour cela, il faut que les tâches respectent scrupuleusement le paradigme de programmation et ne fassent pas d'appel à des bibliothèques utilisant des mémoires tampon comme les entrées/sorties au moment de la migration. Par exemple, il ne faut pas avoir de descripteur de fichier ouvert lors de la migration.

L'approche de gestion mémoire choisie est particulièrement conçue pour tenir compte des spécificités des applications multithreads sur architecture hiérarchique. De plus, cette approche permet l'extension de la méthode d'équilibrage de charge intraprocessus à un équilibrage de charge global sur la totalité des processeurs de la machine. Cette gestion mémoire optimisée est donc parfaitement adaptée à notre modèle de programmation.

7.3 Choix du modèle de gestion des communications

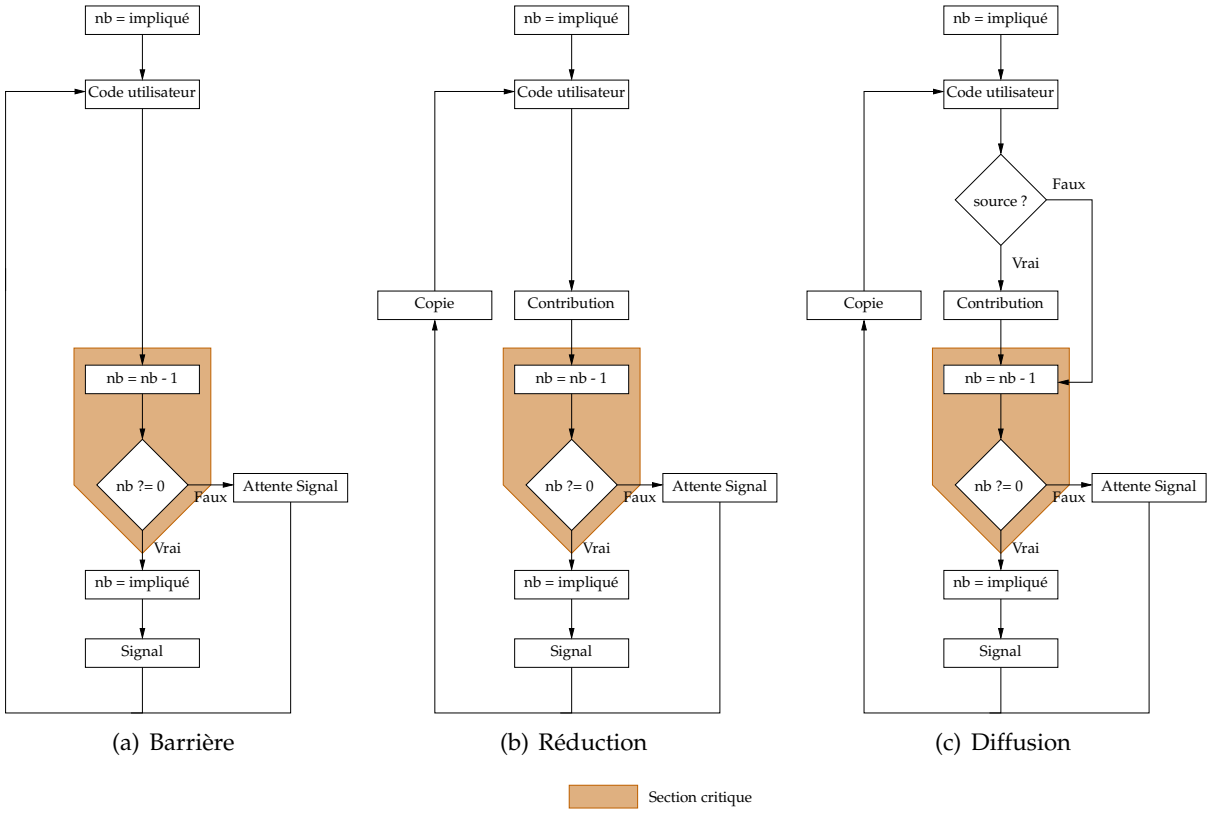
La gestion des communications est primordiale dans la conception d'une bibliothèque de communication. L'environnement MPC disposant en outre de sa propre bibliothèque de threads et de son allocateur mémoire, il convient donc de concevoir des méthodes de communication qui vont dialoguer avec ces composants pour offrir un ensemble cohérent et performant.

7.3.1 Communications collectives

Les communications collectives sont un des types de communication proposé par les approches de communication par passage de messages. Ces communications sont particulièrement critiques en terme de performances. Nous allons décrire ici les caractéristiques des communications collectives. Ensuite, nous détaillerons les principales difficultés de ces communications dans notre contexte de programmation. Enfin, nous présenterons une approche pour réaliser des communications collectives avec un support de l'ordonnanceur de threads.

7.3.1.1 Concept de communications collectives centralisées

Comme nous l'avons vu en 4.2.1.1, on peut classer les communications collectives en trois catégories suivant le service fourni. Il y a la catégorie des synchronisations comme la barrière ; la catégorie des communications comme la diffusion ou les dispersions/regroupements de données. Enfin, la catégorie calcul dont la réduction est un exemple. Néanmoins, cette caractérisation des communications collectives, n'est pas représentative de leur schéma de fonctionnement. En effet, les opérations de barrière, diffusion et réduction ont un schéma de fonctionnement commun centralisé alors que les opérations de dispersions/regroupements ont une approche distribuée. Nous allons donc particulièrement nous intéresser aux communications collectives de type centralisé qui sont antagonistes avec notre modèle de programmation distribuée. Ce sont donc celles-ci qui posent réellement problème. Les communications dispersions/regroupements seront, quant à elles, traitées comme un ensemble de communications point à point. Les communications collectives centralisées, de par leur aspect centralisé, sont particulièrement critiques. En effet, elles vont cadencer le rythme de l'exécution. Les opérations collectives réduction, barrière et diffusion ont la même approche comme on peut le voir sur la figure 7.4. C'est pourquoi nous allons les traiter et les optimiser toutes à la fois en construisant un algorithme commun aux trois communications.



(a) Barrière (b) Réduction (c) Diffusion

FIG. 7.4 – Sémantique des opérations collectives

7.3.1.2 Problèmes induits par la surcharge

La surcharge de tâches induit des problèmes nouveaux par rapport aux implémentations actuelles de nombreuses bibliothèques de communication. Ces problèmes sont au nombre de deux.

Le premier problème est dû au couple utilisation de la surcharge et utilisation des communicateurs de la sémantique MPI. En effet, avec l'utilisation des communicateurs, il est possible d'avoir sur un même processeur un groupe de tâches participant à une communication collective et un autre groupe qui effectue des phases de calcul. La phase de synchronisation des tâches de la communication collective doit donc être faite judicieusement pour permettre d'avoir à la fois une attente et un réveil efficace mais aussi pour permettre aux tâches en cours de continuer leur exécution de manière optimale. Pour parvenir à résoudre cette difficulté, il convient de disposer d'une information globale sur l'état de toutes les tâches. Comme une tâche est un thread, il faut connaître précisément l'état des threads. Le module de MPC disposant de toutes ces informations est l'ordonnanceur de threads. La conception de méthodes performantes implémentant les communications collectives nécessite donc une interaction fine avec l'ordonnanceur de threads.

Le deuxième problème, induit par la surcharge, est le nombre de tâches à "réveiller" lors de la fin de la communication collective. En effet, chaque processeur va devoir réveiller un nombre potentiellement élevé de tâches. Il n'est pas envisageable de mettre en place un réveil en $O(n)$ où n est le nombre de tâches. Si on analyse finement le problème du réveil, on se rend compte que cela consiste à faire passer un ensemble de threads de l'état bloqué à l'état prêt dans l'ordonnanceur de threads. La communication collective ayant la connaissance des tâches participant et l'ordonnanceur de threads maîtrisant l'état des threads, une interaction de ces deux parties doit permettre de réduire la complexité du réveil à une complexité en $O(1)$.

On constate donc que pour réaliser des communications collectives efficaces dans notre contexte, une interaction fine avec l'ordonnanceur de threads est nécessaire.

7.3.1.3 Optimisation des communications collectives et interaction avec l'ordonnanceur de threads

Difficultés dans la conception des communications collectives

L'approche de modèle d'exécution que nous avons choisie nous permet de modifier aisément et de façon portable l'ordonnancement des tâches. Plusieurs problèmes sont induits par le mécanisme de surcharge de tâches. Pour résoudre ces problèmes, nous avons mis en place un mécanisme de communications collectives interagissant avec l'ordonnanceur. Outre les deux problèmes liés à la surcharge, il y a deux autres problèmes qui tiennent plus de la sémantique des communications. Le premier problème vient de la section critique qui brise le parallélisme des opérations. Le second problème pose la question du réveil efficace des threads.

Le premier obstacle à la réalisation de communications collectives efficaces réside dans la portion en exclusion mutuelle de ces communications. En effet, cette partie est intrinsèquement séquentielle. Pour limiter ce phénomène, il vient tout de suite à l'idée de procéder de manière arborescente entre les processeurs. En effet, bien que les processeurs doivent exécuter plusieurs tâches, ils n'exécutent qu'une seule tâche à chaque instant. Dans les approches avec un ordonnancement mixte non-préemptif, cette assertion est renforcée par le fait qu'un thread ne peut être interrompu. Il est donc parfaitement inutile de mettre en place une structure arborescente pour effectuer les communications collectives au sein de chaque processeur. L'approche séquentielle convient parfaitement pour l'intraprocesseur car les threads s'exécutent naturellement en exclusion mutuelle.

Le deuxième obstacle réside dans les phases de blocage/réveil des tâches. En effet, il est nécessaire que les tâches bloquées n'influent pas sur le comportement des tâches encore actives. Les méthodes d'attente active sont donc à bannir. Nous allons donc mettre en place une procédure de blocage/réveil des tâches par processeur virtuel. Les tâches bloquées vont être au fur et à mesure retirées de l'ordonnanceur local à chaque processeur. Ainsi les autres threads s'exécutant sur le processeur virtuel ne sont pas influencés par les threads en attente de communication collective. En ce qui concerne le réveil, toutes les tâches précédemment bloquées sur chaque processeur virtuel vont être réinsérées dans la liste des tâches prêtes en une unique opération qui sera indépendante du nombre de tâches. En effet, tous les threads bloqués dans la communication collective sont prêts à être débloqués en même temps.

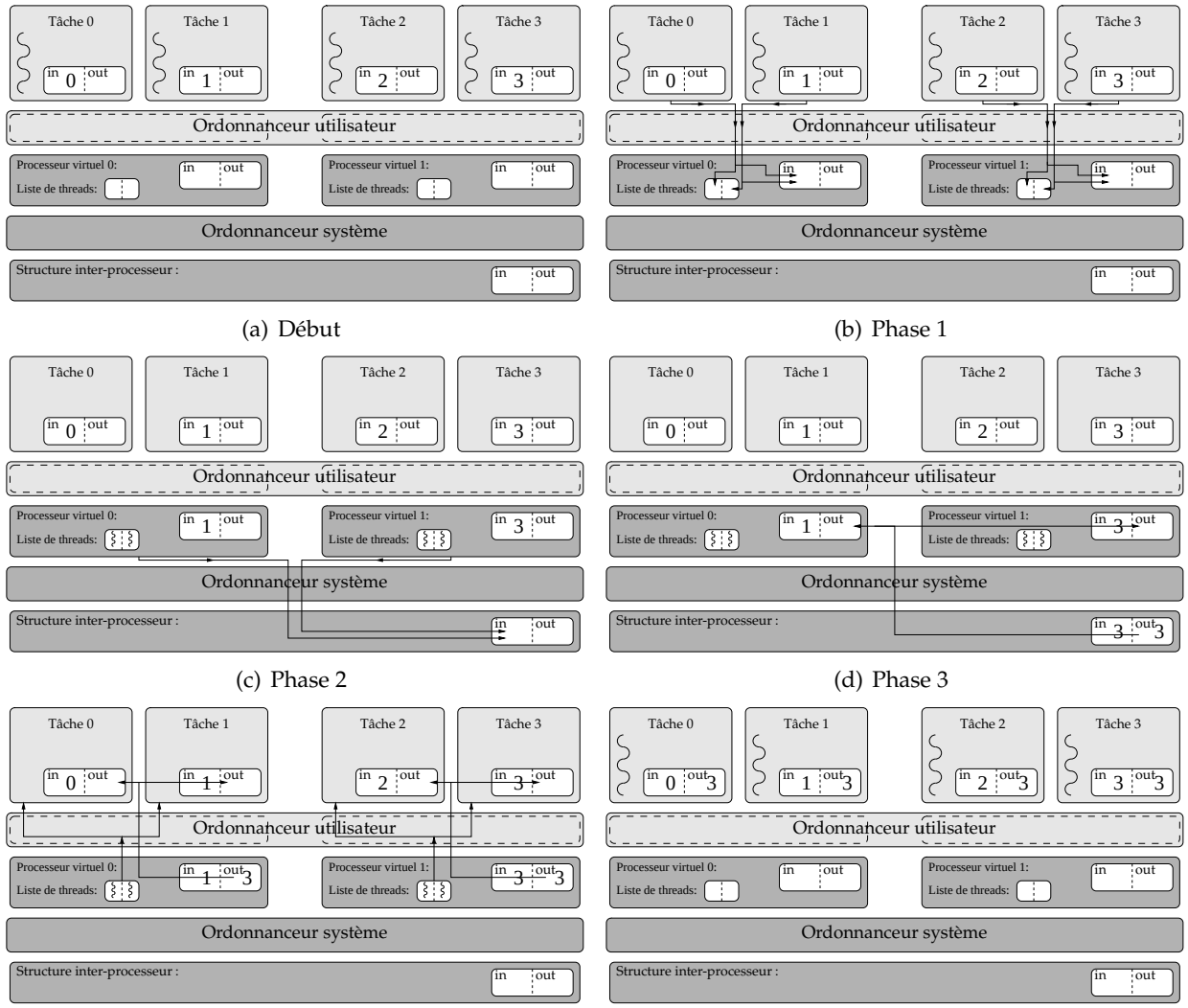
Algorithme

L'algorithme que nous avons mis en place utilise le concept de "processeur impliqué". Chaque communication collective possède un ensemble de processeurs impliqués. Cet ensemble est constitué de processeurs virtuels qui possèdent dans leur file de tâches à exécuter des tâches participant à la communication collective. Dans le but de maximiser la localité des données, l'algorithme est distribué sur les différents processeurs, ce qui permet à chaque processeur virtuel de décider de l'état de chaque tâche indépendamment des autres processeurs virtuels. Chaque processeur impliqué possède une structure de données qui contient déjà, d'une part, le nombre de ces tâches impliquées dans la communication collective et, d'autre part, une file de tâches qui ont contribué à la communication collective et qui sont donc bloquées. Enfin cette structure contient les éléments nécessaires à la conception d'une structure arborescente pour la communication collective. Dans un premier temps, nous considérons qu'il n'y a pas de migration des threads sur les processeurs virtuels. Chaque thread a donc un processeur dit favori qui est conservé tout au long de l'exécution du programme. La figure 7.5 illustre l'algorithme de communications collectives sur un exemple de réduction avec calcul du maximum.

L'algorithme se décompose en quatre phases. La première phase (voir figure 7.5(b)) a pour but de réaliser les sous-parties de communication collective sur chaque processeur impliqué. C'est-à-dire que tous les threads concernés par la communication collective vont apporter leur contribution dans la structure de données de leur processeur favori. Cette phase est intrinsèquement séquentielle car un processeur virtuel ne peut exécuter plusieurs threads à un instant donné. Cette phase s'opère donc sans aucune synchronisation. On ne passera à la seconde phase que lorsque tous les threads auront contribué sur leur processeur favori et se seront donc bloqués après s'être enregistrés sur leur processeur favori.

La seconde phase (voir figure 7.5(c)) consiste en la réalisation de l'opération collective entre les processeurs impliqués. Comme chaque processeur impliqué dispose de la contribution de tous les threads impliqués dont il a la charge, on peut procéder de manière arborescente au calcul du résultat de la communication collective. Cette phase nécessite des synchronisations entre les processeurs, c'est pourquoi nous avons choisi une approche arborescente qui limite la contention et maximise le parallélisme. Chaque processeur impliqué va donc contribuer à l'opération collective et se bloquer en attente de libération. Néanmoins, le blocage n'est que partiel car un processeur impliqué peut exécuter d'autres threads non impliqués dans la communication durant cette phase d'attente. À la fin de cette phase, le dernier processeur impliqué qui va contribuer à la communication collective va initier la troisième phase.

La troisième phase (voir figure 7.5(d)) va opérer le réveil des processeurs impliqués et la copie



(a) Début (b) Phase 1

(c) Phase 2 (d) Phase 3

(e) Phase 4 (f) Fin

FIG. 7.5 – Exemple d'opération collective

des données, ces dernières étant le résultat de la communication collective. Il y a donc diffusion arborescente de résultats avec réveil des processeurs virtuels impliqués. Cette phase se termine lorsque tous les processeurs impliqués ont été réveillés.

La quatrième phase (voir figure 7.5(e)) va mettre en œuvre le réveil des threads bloqués sur chaque processeur impliqué. Comme les threads se sont enregistrés au moment du blocage, le processeur impliqué va transmettre la liste des threads à réveiller à l'ordonnanceur de ce processeur virtuel. L'ordonnanceur va alors réinjecter la liste des threads dans la liste des threads prêts. Ainsi, tous les threads seront débloqués en une opération.

L'approche mise en œuvre permet d'optimiser chaque phase en fonction des informations de placement des threads sur les processeurs virtuels. Néanmoins, nous avons fait la supposition que les threads ne pouvaient migrer d'un processeur virtuel à l'autre, ce qui interdit, pour l'instant, toute politique d'équilibrage de charge.

Prise en compte de la migration de threads

Pour pouvoir intégrer une politique d'équilibrage de charge, il faut faire évoluer l'algorithme présenté ici. L'information critique pour l'algorithme est le nombre de threads impliqués sur chaque processeur virtuel. En effet, cette information permet de définir si un processeur virtuel est impliqué ou non mais permet aussi de décider du moment auquel on peut initier la seconde phase. Il faut donc maintenir cette information à jour au fil des migrations.

La première méthode envisageable est la mise à jour systématique à chaque migration de threads. Ainsi, les structures seront prêtes lorsqu'une communication collective sera initiée. Néanmoins, cette approche s'avère coûteuse dans la mesure où l'on peut faire trop de mises à jour et que ces dernières vont nécessiter des synchronisations à chaque migration. Nous avons donc rejeté cette approche.

L'approche choisie est une mise à jour paresseuse. Cette approche a pour objectif de limiter le nombre de mises à jour. Il n'y aura pas de mise à jour systématique à la migration. Lorsqu'un thread va faire un appel à une communication collective, il va en premier lieu vérifier si son processeur virtuel actuel est le même que celui du dernier appel à cette communication collective. Si le processeur est le même il n'y a pas de mise à jour à faire. Il faut noter que le thread peut avoir migré plusieurs fois entre les deux appels mais que ces migrations n'ont aucune importance du point de vue de la communication collective. Le deuxième cas de figure survient si le thread a changé de processeur virtuel. Dans ce cas, il va contribuer soit sur le processeur virtuel courant soit sur son ancien processeur virtuel. Si le processeur courant est un processeur impliqué et n'a pas entamé la seconde phase, le thread peut contribuer sur le processeur courant. Dans le cas contraire, le thread va contribuer sur l'ancien processeur virtuel. Dans tous les cas, il va se bloquer sur le processeur courant. En ce qui concerne la mise à jour des informations de chaque processeur, nous allons profiter des synchronisations lors de la seconde phase pour centraliser les modifications et utiliser la troisième phase pour diffuser et mettre à jour les informations.

L'approche de mise à jour que nous avons choisie permet de limiter au strict minimum les mises à jour tout en conservant les bonnes propriétés de notre algorithme.

7.3.2 Communication point à point

Les communications point à point sont le deuxième type de communication de l'approche mémoire distribuée avec communications par passage de messages. Ces communications sont très fréquentes et requièrent donc d'être particulièrement optimisées.

7.3.2.1 Maximisation du parallélisme

Le principe même des communications point à point est qu'elles ne font intervenir que deux tâches. Pour maximiser le parallélisme, il faut que ces communications entre deux tâches n'interfèrent pas sur l'exécution des autres tâches. Dans un contexte de programmation mémoire partagée, il convient de mettre en place des structures de données pour les communications qui doivent à la fois être accessibles simultanément par un grand nombre de tâches distinctes mais aussi permettre une implémentation efficace des communications. C'est pour cela que nous avons choisi de mettre en place une structure de communication de type graphe complet orienté. Ainsi, il n'y a pratiquement jamais de concurrence pour déposer des messages. En effet, c'est seulement lors de la réalisation effective de la copie sur la tâche a du message de b vers a que b ne pourra pas envoyer

de messages à a. De plus, cette approche a pour effet de permettre une localisation rapide des messages suivant leur couple $(source, destination)$. Néanmoins, elle est légèrement plus coûteuse en quantité mémoire utilisée qu'une approche utilisant une seule file. Nous avons donc choisi de privilégier l'axe performance par rapport à l'axe consommation mémoire. La figure 7.6 illustre le fonctionnement des communications point à point dans le cas d'un schéma de connexion de type anneau.

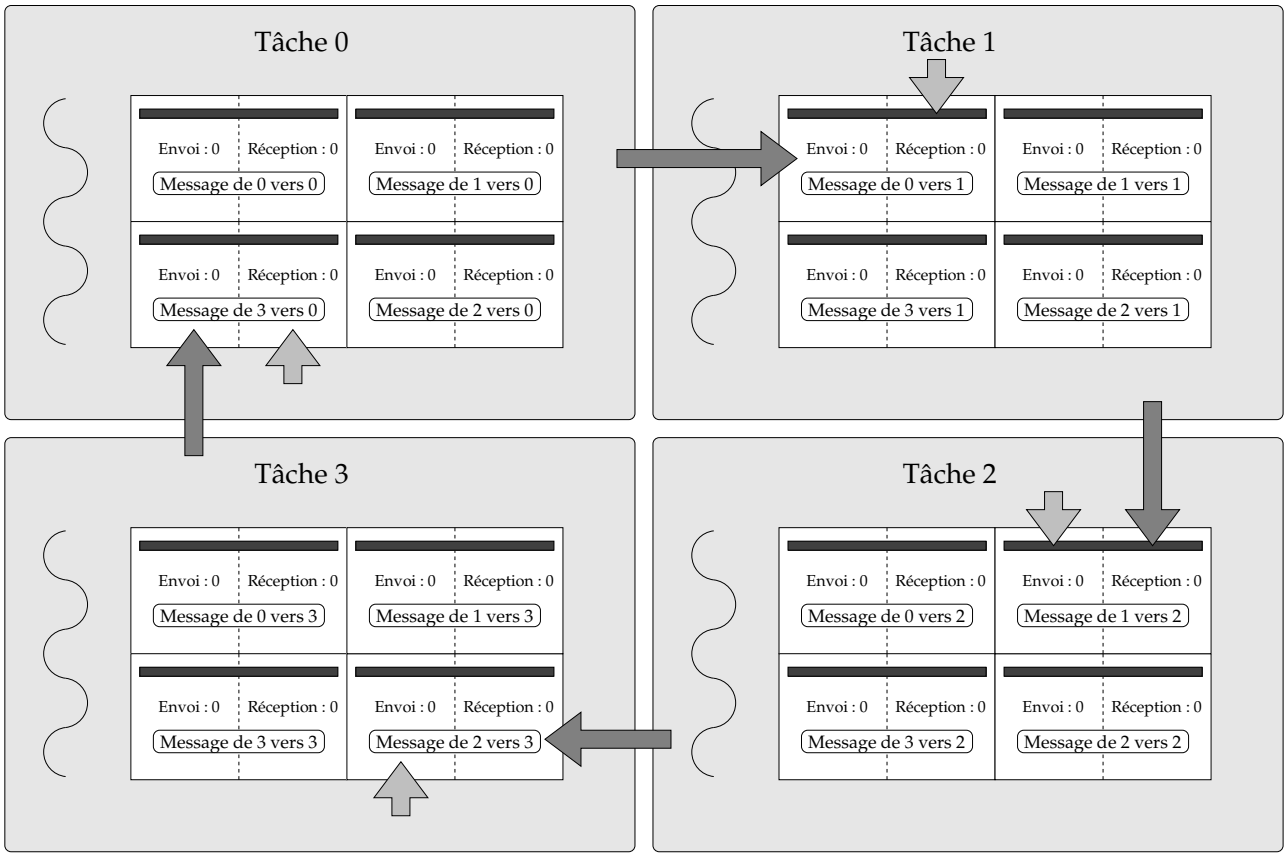


FIG. 7.6 – Illustration des communications point à point sur un anneau (envoi/reception)

7.3.2.2 Politique des copies des messages

Pour profiter au maximum de l'aspect mémoire partagée du modèle de programmation interne de MPC, nous avons choisi de mettre en place une approche zéro copie. Nous n'utilisons donc pas de mémoire tampon intermédiaire dans laquelle les messages seraient recopiés. En effet, cette approche aurait engendré un surcoût au niveau de la conception pour un gain faible pour les petits messages. De plus, nous disposons de la méthode de surcharge qui va permettre de recouvrir efficacement ces latences dans les communications. Le schéma de communication est donc très simple. Une copie de message de la source vers le destinataire ne peut être effectuée que lorsque les deux parties sont prêtes respectivement à émettre et recevoir le message. Dès lors, une approche simple de copie, par la tâche réceptrice au plus tard, peut aisément et efficacement être mise en place. Cette approche a pour avantage de précharger les nouvelles données dans la mémoire cache

du processeur. En effet, on va effectuer la copie au moment où cette copie devient la condition indispensable à la poursuite de l'exécution de la tâche. Réaliser la copie à ce moment va permettre à la fois d'éviter le blocage quand cela est possible mais aussi rentabiliser le chargement des lignes de cache induit par la copie du message en permettant la réutilisation de ces lignes par la tâche réceptrice qui va poursuivre son exécution.

7.3.2.3 Gestion de la scrutation

Pour les cas où il n'est pas possible de procéder tout de suite à la copie du message car celui-ci n'est pas encore prêt, nous utilisons la politique de scrutation interne à l'ordonnanceur que nous avons présentée. Cette méthode de scrutation permet, rappelons-le, d'avoir un comportement efficace en cas de surcharge tout en maintenant une bonne réactivité pour le réveil des tâches bloquées. En effet, les tâches en cours d'exécution ne sont pas interrompues en cours de calcul ce qui permet de maintenir de bonnes performances pour l'exécution des boucles de calcul dans le contexte de surcharge. Néanmoins, cette approche non préemptive peut retarder le réveil d'une tâche émettrice en attente de la réception d'un message. Ici encore, la surcharge apporte la solution car avec un nombre suffisant de tâches par processeur, le phénomène d'attente est atténué car les phases d'attente de chaque tâche sont recouvertes par les phases de calcul des autres tâches. La surcharge n'apporte donc pas de réel problème concernant la gestion de la scrutation dans le cadre des communications point à point.

7.4 Choix du modèle de gestion de la tolérance aux pannes

Le protocole de tolérance aux pannes est une fonctionnalité très importante pour les simulations nécessitant plusieurs jours, semaines, voire mois de calcul. Le protocole est défini par une approche de fonctionnement et un mécanisme qui définissent quelles données sont sauvegardées en cas de panne. C'est ce que nous allons décrire ici.

7.4.1 Une approche coordonnée

Comme la plupart des implémentations du standard MPI qui implémentent une politique de tolérance aux pannes, nous avons choisi une approche par mécanisme de points de reprise coordonnés. Cette technique assure que l'ensemble des points de reprise forment un état global cohérent. De plus, la reprise est simplifiée. Un autre avantage de cette technique est que l'espace de stockage nécessaire est minimisé puisqu'il n'est nécessaire de conserver qu'un point de reprise par tâche. Néanmoins, cette méthode a un inconvénient majeur qui est sa latence. En effet, cette méthode nécessite une coordination globale de toutes les tâches avant d'initier un point de reprise.

7.4.2 Mécanisme

Le type de point de reprise que nous allons mettre en œuvre est simple. Il va permettre d'écrire sur disque l'état d'une tâche sans tenir compte des entrées/sorties, ni des communications en attente. On suppose donc que l'utilisateur gère lui-même les entrées/sorties, et qu'il ne met pas en place un schéma de communication de part et d'autre du point de reprise. C'est-à-dire qu'il n'y a pas d'opérations d'envoi qui dépendent d'une réception située après le point de reprise et vice

versa. Dans ce contexte, les seules données à sauvegarder sont les données allouées dynamiquement par la tâche (ses données locales). Il n'y a pas de données globales aux tâches car le modèle est orienté programmation distribuée. Il faut de plus sauvegarder le contexte processeur et la pile de la tâche. Grâce à la méthode mise en œuvre pour la gestion mémoire, il est aisé d'identifier les données de chaque tâche et donc d'en effectuer une copie sur disque.

Cette approche permet la mise en place d'une méthode de points de reprise quasi transparente dans la mesure où l'utilisateur a juste à respecter les contraintes citées ci-dessus pour effectuer un point de reprise.

Les capacités de reprise vont fortement différencier notre méthode de points de reprise des autres implémentations. En effet, grâce à la gestion mémoire qui est sous notre contrôle ainsi que le mécanisme de migration de tâches internœuds que nous avons mis en place, il est possible de reprendre une application interrompue sur un nombre de nœuds et/ou de processeurs différents du moment de la protection. Effectivement, il est possible de faire une redistribution des tâches au moment de la reprise pour s'adapter à la nouvelle configuration. Comme tous nos mécanismes de communication sont conçus pour bien se comporter en cas de surcharge, il est possible de faire varier le nombre de processeurs utilisés de un au nombre de tâches créées initialement. En effet, utiliser plus de processeurs que de tâches n'a que peu d'intérêt.

7.5 Mise en place du modèle d'équilibrage de charge

Comme nous l'avons décrit tout au long de la présentation des modèles d'exécution, la priorité a été donnée à la flexibilité. Ainsi, le nombre de tâches par processeur peut être variable. De plus, nous disposons d'un mécanisme permettant de sauvegarder une tâche sur disque. Tous les ingrédients sont donc présents pour proposer des outils afin de mettre en œuvre une politique d'équilibrage de charge par migration de tâches. Pour faciliter la mise en œuvre d'une telle politique, nous avons assoupli les contraintes en terme de communication en mettant en place un système de suivi des messages en cas de migration. Les messages en attente sont donc déplacés en même temps que la tâche qui doit migrer. Néanmoins les contraintes sur les entrées/sorties sont conservées.

Ce chapitre a permis de définir les modèles d'exécution en fonction des besoins des applications qui vont utiliser MPC. Il a aussi permis de mettre en avant les interactions fortes entre les différents modules de MPC. Ces interactions rarement présentes dans les bibliothèques de communication sont une des principales caractéristiques de MPC. Il convient maintenant de procéder à l'implémentation de ces modèles.

Troisième partie

Implantation et évaluation

Chapitre 8

Éléments d'implémentation de MPC sur architectures hiérarchiques

L'objectif de ce chapitre est de décrire les solutions logicielles que nous avons mises en œuvre pour adapter l'environnement logiciel du supercalculateur à nos modèles d'exécution.

Pour cela, nous allons présenter la solution logicielle mise en place pour la gestion des tâches. Ensuite, nous détaillerons la mise en œuvre de la gestion de la mémoire, puis, les techniques de gestion des communications. Nous poursuivrons avec les solutions techniques qui nous ont permis de concevoir la politique de tolérance aux pannes. Nous finirons avec les solutions mises en place pour l'équilibrage de charge.

8.1 Gestion des tâches

Comme nous l'avons vu au chapitre 7, le modèle de gestion des tâches est multithread. L'implémentation de ce modèle réside donc dans l'implémentation d'une bibliothèque de threads constituée de toutes les primitives de synchronisation et d'un ordonnanceur optimisé pour le calcul scientifique. Cette section se décompose en trois parties. La première décrit les principales caractéristiques techniques de la bibliothèque de threads. La seconde partie détaille les méthodes mises en œuvre pour assurer le placement des threads. La troisième partie détaille l'implémentation de la méthode de scrutation.

8.1.1 Bibliothèque de threads

La bibliothèque de threads intégrée à MPC est une bibliothèque utilisateur disposant d'un ordonnanceur à deux niveaux. Elle a été conçue sur le modèle des bibliothèques NGPT, GnuPTH et Marcel.

Le système de gestion des threads est inspiré des bibliothèques NGPT et GnuPTH et plus particulièrement des travaux de Ralf S. Engelschall[22]. Ces derniers proposent une approche portable de gestion des threads utilisateur basée sur `setjmp/longjmp` ou `makecontext/swapcontext` suivant les architectures. Cette approche présente le gros avantage de ne pas nécessiter de code assembleur pour effectuer les changements de contexte. Néanmoins, et par souci de performance, nous avons ajouté des versions optimisées de primitives de changements de contexte pour l'architecture Itanium. En effet, cette architecture ne disposait pas de versions suffisamment perfor-

mantes. Ces primitives assembleur ont été inspirées par la bibliothèque Marcel. Cette approche nous permet donc de disposer d'une version portable et d'une version optimisée lorsque cela s'avère nécessaire.

Un autre point technique dans la conception de bibliothèques de threads est la conception des opérations de synchronisation. Ces opérations reposent toutes sur la possibilité d'exécuter de manière ininterruptible un test et une affectation (*test-and-set*). De même que pour le changement de contexte, nous disposons de deux méthodes pour effectuer le *test-and-set*. La première et la plus portable utilise les threads POSIX. Dans cette méthode, nous utilisons les *mutex* pour garantir l'exécution en exclusion mutuelle. Cette méthode est portable mais pas très performante. La deuxième méthode utilise l'instruction assembleur *test-and-set*. Cette version est, elle, efficace mais nécessite d'être implémentée pour chaque architecture. Actuellement, nous l'avons implémentée pour toutes les architectures supportées.

Le dernier point technique est la conception de l'ordonnanceur. Comme nous l'avons dit, ce dernier est un ordonnanceur multifile, non préemptif, sans méthode d'équilibrage interne. Il est donc particulièrement simple à concevoir. Néanmoins, par souci d'efficacité, nous avons porté une attention toute particulière au placement des données. En effet, les files et structures de l'ordonnanceur doivent être placées judicieusement pour éviter les accès distants sur les architectures NUMA. Nous avons donc opté pour une distribution totale des données. Chaque processeur virtuel dispose d'une structure propre qui contient les listes de threads ainsi que toutes les informations concernant son activité (utilisées pour l'équilibrage de charge). Cette structure est créée à l'initialisation et placée par le processeur virtuel en mémoire locale à chaque processeur virtuel. L'ordonnanceur de threads a par la suite été enrichi de fonctionnalités spécifiques que nous détaillerons par la suite.

Les fils directeurs qui ont guidé la création de la composante thread de MPC sont portabilité et performance. Comme ces deux notions sont parfois incompatibles, nous avons pris le parti de proposer une version totalement portable qui peut être remplacée par une approche optimisée non-portable quand cela est possible.

8.1.2 Placement

Sur architecture NUMA, le placement des threads en fonction de leurs données est primordial dans la conception d'applications performantes. C'est pourquoi toute la conception de MPC a été faite avec ce souci.

Pour garantir le placement des données et des threads, nous avons besoin d'outils système qui nous permettent de garantir le placement. Nous avons utilisé les bibliothèques de placement *libnuma* et *cpuset/memset* pour assurer un placement optimal. Ces bibliothèques présentes sous Linux offrent globalement les mêmes fonctionnalités mais sont incompatibles. Nous avons donc choisi de gérer les deux approches en fonction des bibliothèques annexes utilisées par les applications. On peut citer la bibliothèque *MPIBull* qui est incompatible avec la *libnuma* et qui requiert donc l'utilisation de la bibliothèque *cpuset/memset*. Ces bibliothèques nous permettent de "boulonner" les processeurs virtuels sur les processeurs physiques et de forcer toutes les allocations faites par les threads utilisant ces processeurs virtuels comme propulseurs à devenir locales. Ainsi, nous obtenons une allocation des données et un placement qui minimisent les accès distants.

Les primitives de placement n'ont pu être mises en place que pour le système Linux. En effet, il n'existe pas d'interface portable pour effectuer ce type d'optimisation. N'ayant à disposition aucune autre architecture NUMA, nous n'avons pas implémenter de méthodes de placement pour

d'autres systèmes.

8.1.3 Scrutation

Comme nous l'avons vu au chapitre précédent, de nombreuses optimisations reposent sur la méthode de scrutation interne à l'ordonnanceur. Contrairement à de nombreuses implémentations de méthodes de scrutation au niveau de l'ordonnanceur, nous avons pris le parti d'autoriser tous les appels de fonctions de synchronisation dans les fonctions de scrutation.

La principale difficulté résidant dans notre approche de scrutation se situe dans les primitives de synchronisation. En effet, les fonctions de scrutation vont être appelées par l'ordonnanceur lui-même. Lors d'un appel à une primitive de synchronisation comme un *mutex*, il se peut que le *mutex* soit déjà pris. Le comportement normal d'un thread dans ce cas est de passer la main à un autre thread et d'attendre que le *mutex* soit libéré. L'ordonnanceur ne peut pas avoir ce comportement car il ne peut ni se bloquer ni passer la main. Néanmoins, il peut chercher à exécuter un thread qui est prêt, il peut aussi suspendre les appels aux fonctions de scrutation tant que le verrou n'est pas libre. Pour cela, il a été nécessaire de modifier les fonctions de synchronisation pour mettre en place un traitant spécial dans le cas où le thread appelant la fonction de synchronisation est l'ordonnanceur. Dans ce cas et si l'appel est bloquant, l'ordonnanceur va stopper la progression de l'exécution de la scrutation tant que le verrou n'est pas libre. Pour que cette attente ne soit pas de l'attente active, il va tenter d'exécuter les threads prêts pour recouvrir ce temps d'attente.

La méthode de scrutation que nous avons mise en place est très générale car elle permet de fournir toutes fonctions au serveur de scrutation. Elle est de plus efficace car elle va chercher à recouvrir au mieux les opérations de synchronisation.

La gestion des tâches est la partie de MPC la plus sensible en terme de performance. En effet, c'est la bibliothèque de threads qui est en charge de l'ordonnancement des threads mais aussi du placement des threads qui est crucial pour la gestion mémoire. Elle est en outre en charge de la scrutation qui va nous permettre de concevoir des communications performantes. On peut donc dire que toutes les composantes de MPC vont faire appel à la gestion des taches.

8.2 Gestion mémoire

La gestion mémoire est un point critique pour les architectures à mémoire hiérarchique. En effet, l'obtention de bons résultats de performance est couplée au fait d'avoir une bonne localité des données. Nous allons présenter les méthodes d'allocation mémoire et de localité des données que nous avons mises en place. Ensuite, nous discuterons des limites de la méthode choisie.

8.2.1 Méthode d'allocation

Pour tenir compte de l'aspect multithread de notre application, nous avons reconstruit une bibliothèque d'allocation mémoire par thread. Cette bibliothèque va mettre en place un tas local virtuel pour chaque thread.

Chaque thread va disposer d'un tas local discontinu à la granularité d'une page mémoire système. C'est-à-dire que chaque thread gère un ensemble de superblocs mémoire constitués chacun d'un multiple de pages mémoire système. Le fait de choisir un multiple de pages mémoire système permet d'éviter le faux partage entre threads. Pour chaque allocation, le thread va chercher

à allouer le bloc de la taille demandée dans un superbloc de son tas local. Si aucun superbloc ne convient alors le thread va allouer un nouveau superbloc de taille suffisante en utilisant l'appel `mmap`. Ce nouveau super bloc va donc agrandir le tas local. L'utilisation de *mmap* a un double intérêt. Tout d'abord, elle permet d'allouer un espace mémoire mais elle permet aussi de choisir à quelle adresse virtuelle l'allouer. Ainsi, nous pouvons aisément mettre en œuvre la séparation des espaces mémoire virtuelle(voir 7.2) requis pour la migration de threads ou pour le mécanisme de tolérance aux pannes. Comme chaque thread est seul maître de l'allocation dans ses superblocs, il n'y a pas de problème de contention hormis lors de l'appel à `mmap` qui est de toute façon un point de contention système incontournable puisqu'il nécessite une synchronisation entre tous les processus ou threads noyau du système. La figure 8.1 illustre le mécanisme d'allocation mémoire.

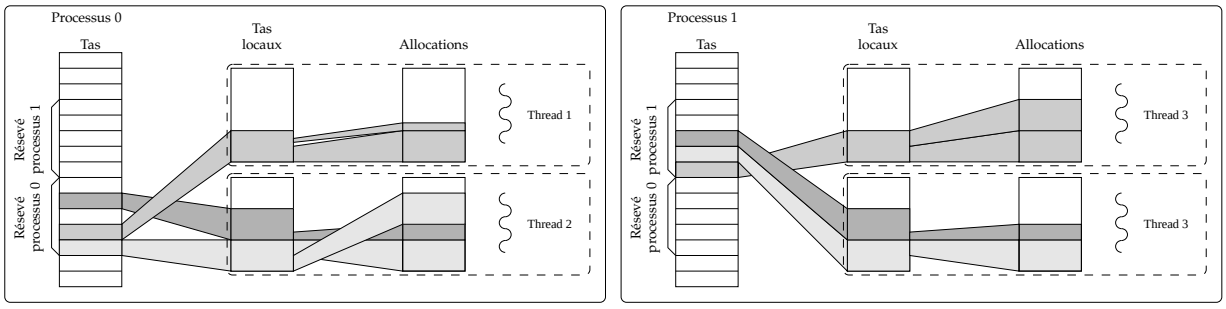


FIG. 8.1 – Illustration du modèle de gestion mémoire

Une évolution de la méthode d'allocation mémoire présentée ici peut être d'étendre la méthode avec un tas local par QBB pour avoir une meilleure utilisation de la mémoire et diminuer le fractionnement tout en conservant la localité des données.

8.2.2 Localité des données

La mise en place et la conservation d'une bonne localité des données est fortement couplée à la bibliothèque de threads et à la méthode d'allocation.

Pour mettre en place une localité des données, il est nécessaire que l'allocation des superblocs constituant le tas virtuel des threads soit locale aux threads. Cette localité est obtenue par l'utilisation de bibliothèques spécifiques NUMA au sein de la bibliothèque de threads. Ces bibliothèques, couplées à l'ordonnanceur de threads, garantissent que les threads s'exécutant au-dessus d'un processeur virtuel ont toutes leurs données proches du processeur virtuel. On peut donc dire que tant que le thread ne change pas de processeur virtuel, il fera tous ses accès en mémoire locale. En cas de migration sur un autre processeur, la bibliothèque d'allocation sera en mesure de spécifier la liste des pages à migrer. Cette liste va être constituée des pages composant le tas local. Cette dernière optimisation n'est pas encore implémentée car elle requiert des modifications du système d'exploitation qui sont en cours de réalisation pour une de nos architectures de test.

A l'heure actuelle, MPC est en mesure d'assurer la localité des données tant qu'il n'est pas mis en œuvre de migration de threads entre les processeurs virtuels. En cas de migration, seul un support du système peut permettre de rétablir la localité.

8.2.3 Limitations

Comme nous venons de le remarquer, la méthode d'allocation mémoire souffre de quelques limitations qui seront résolues via un support du système. Il existe aussi d'autres limitations plus liées à la méthode elle-même.

L'approche d'allocation mémoire que nous avons choisie utilise un tas virtuel par thread. La particularité de ce tas est qu'il est discontinu. Ceci a pour effet qu'il n'est pas possible, dans la majorité des cas, de fusionner des superblocs. Cette particularité peut poser problème dans le cas d'appels successifs croissants à `realloc`. En effet, l'allocateur va créer des superblocs de taille croissante sans jamais pouvoir réutiliser les superblocs trop petits. Ces superblocs sont alors perdus si l'on ne peut les fusionner pour reformer un superbloc plus grand. Deux solutions s'offrent à nous pour corriger ce problème. La première et la plus simple consiste à construire un superbloc contigu suffisamment grand pour que toutes les allocations du thread puissent être faites au sein de ce superbloc. Ceci requiert une connaissance préalable des tailles allouées ou une surestimation de la taille maximale possible. Cette dernière approche est très vite limitée sur architecture 32 bits car l'espace d'adressage virtuel est beaucoup trop petit dès lors que l'on utilise un grand nombre de threads. La deuxième solution serait de réutiliser les superblocs alloués par d'autres threads pour pouvoir fusionner les superblocs. Néanmoins, cette approche nécessite un support du système pour maintenir la localité des données allouées.

La méthode d'allocation mémoire souffre de limitations liées à l'incapacité de recycler les pages libérées par les autres threads sur architectures NUMA. Seul un support approprié du système peut permettre de lever ces limitations.

L'approche de gestion mémoire par thread est tout à fait en accord avec le modèle de programmation que nous avons choisi. Avec cette approche, tout est fait pour éviter le faux partage et maintenir une bonne localité mémoire. Cette approche souffre tout de même de quelques limitations qui pourront être levées dans un avenir proche avec un support du système approprié.

8.3 Les communications

Nous distinguons deux types de communication interthread ; les communications internœuds et intranœuds. Nous avons choisi cette nomenclature car le fonctionnement optimal de MPC suggère de placer un processus par nœud.

8.3.1 Communications intranœuds

Le schéma de communication interthread au sein d'un même espace d'adressage tient particulièrement compte de la spécificité de l'architecture. L'implémentation des méthodes présentées en 7.3 ne pose pas de réelles difficultés. Nous ne traiterons donc ici que les optimisations complémentaires apportées.

Comme nous l'avons déjà vu, la contention des bus mémoire est critique. Il convient donc de limiter au maximum les accès concurrents à des données. Pour atteindre cet objectif, nous avons totalement distribué les files d'attente des messages de sorte qu'une file de messages ne soit partagée que par deux threads au plus. Ceci permet à la fois de lever la contention sur les verrous et limiter la contention des bus mémoire lorsque l'architecture le permet (architecture NUMA). En

effet, les structures utilisées ont été judicieusement placées en fonction du placement des threads pour maximiser les accès locaux d'un thread à ses structures de messages.

Une autre caractéristique importante des communications interthreads intranœuds est l'instantanéité des communications. En effet, dès que les requêtes de réception et envoi ont été postées dans les structures, il ne reste plus qu'à effectuer la copie mémoire. Il n'y a pas à "faire avancer" comme dans certaines implémentations MPI. Dans le cas où la copie ne peut être réalisée immédiatement et que la tâche est bloquée, la requête est transmise au serveur de scrutation qui va se charger d'effectuer la copie quand cela sera possible.

La gestion des communications intranœuds tire les bénéfices de la gestion des tâches et de la gestion mémoire pour proposer une approche efficace et optimisée des communications.

8.3.2 Communications internœuds

Les communications internœuds vont être sollicitées dès lors que deux threads situés dans des processus différents désirent communiquer ou lors d'opérations collectives.

Devant la multitude de réseaux disponibles sur les supercalculateurs, nous avons choisi de fonder notre implémentation sur une "valeur sûre" : MPI. En effet, il existe de nombreuses versions de MPI optimisées pour tout type de réseaux. En fait, nous n'utilisons que peu de fonctionnalités de MPI. En effet, MPC disposant d'une carte de la répartition de tâches sur les différents nœuds, une communication point à point internœud va simplement consister à envoyer le message avec comme entête la tâche pour laquelle elle est destinée au bon processus MPI. Les seuls fonctionnalités utilisées ici sont donc l'envoi/réception asynchrones de messages.

La première phase dans les communications collectives est la communication collective intranœud. En effet, ces communications étant arborescentes, il est judicieux de les réaliser au sein de chaque noeud d'abord puis entre les nœuds. En ce qui concerne la partie internœud des opérations collectives, il a fallu tenir compte du comportement de MPI. Comme les appels aux communications collectives sont bloquants, il est impossible de les utiliser car cela risquerait de bloquer un processeur virtuel. Nous avons donc réimplémenté un mécanisme de communications collectives basé sur des envois de messages asynchrones de manière arborescente.

L'implémentation des communications internœuds n'a pas nécessité un gros travail car la majorité de la complexité est laissée à la bibliothèque de communication. Les fonctions utilisées étant très simples, on peut aisément envisager d'utiliser d'autres bibliothèques de communication. Actuellement, une version utilisant TCP/IP est en cours de développement pour les machines où MPI ne serait pas présent.

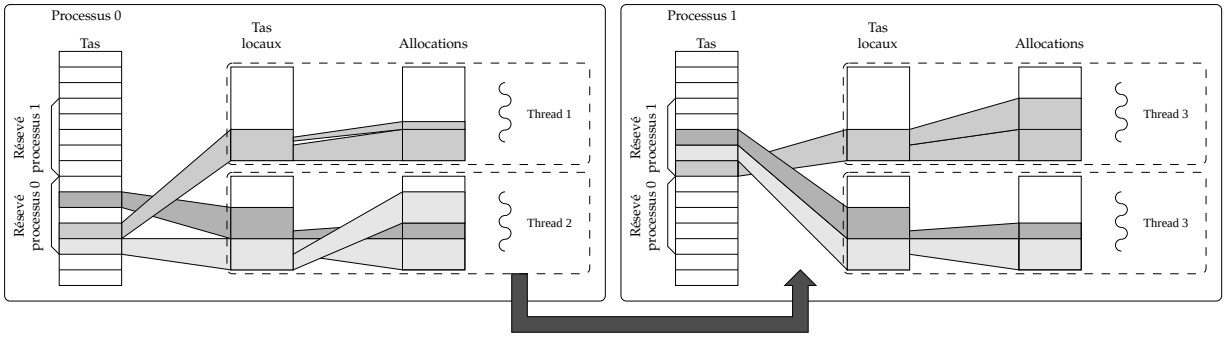
Le schéma de communication repose quasi essentiellement sur les fonctionnalités de l'ordonnanceur et de la gestion mémoire. L'implémentation des mécanismes de communication a consisté à coder les algorithmes élaborés en cherchant à optimiser le placement des données pour obtenir des mécanismes de communication performants.

8.4 Migration de threads interprocessus

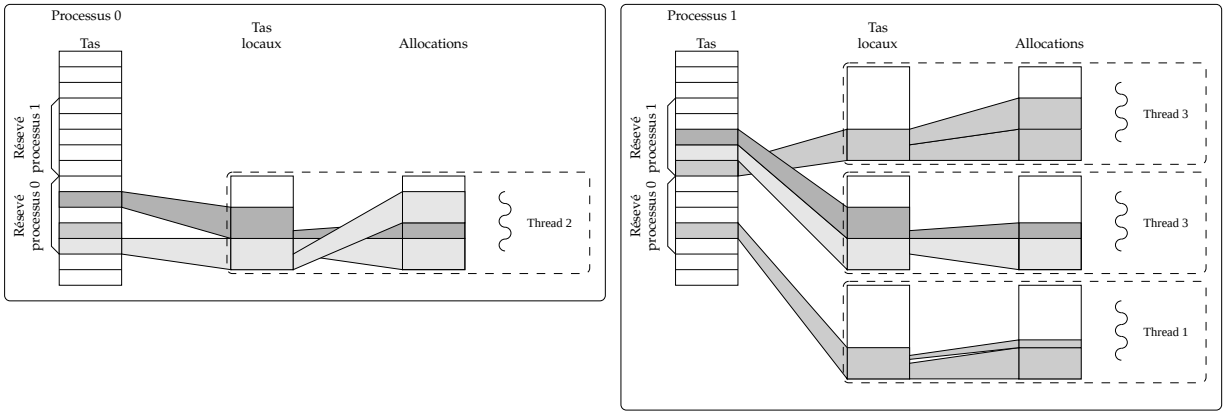
Le mécanisme de migration de threads est une fonctionnalité importante de MPC. Comme nous l'avons déjà mentionné, cette migration a pour but de déplacer des threads entre les différents processus utilisés. Pour réaliser cette migration nous allons déplacer les données utilisées

par le thread et le redémarrer dans un autre processus en prenant soin de maintenir la cohérence des communications.

La première partie de la migration consiste à pouvoir effectivement déplacer un thread et toutes ses données. Pour parvenir à ceci, nous avons déjà prévu le mécanisme d'allocation approprié qui garantit l'unicité de l'utilisation des adresses mémoire sur l'ensemble des processus impliqués. Avec cette garantie, migrer un thread revient simplement à recopier les données allouées par ce thread (son tas local) et la pile du thread dans le processus destination. Ensuite, il faut placer le thread en question dans la liste des threads prêts de l'ordonnanceur de thread qui va reprendre son exécution comme si aucune migration n'avait eu lieu. La figure 8.2 illustre la gestion mémoire de la méthode de migration.



(a) Avant



(b) Après

FIG. 8.2 – Migration de threads

La gestion des communications est très importante lors de la migration car il faut s'assurer qu'aucun message ne se perde. La méthode utilisée va consister à réémettre tous les messages non traités par le thread qui doit migrer. Ces messages sont tous contenus dans les structures décrites en 7.3.2. Il suffit donc, après la migration, de traiter cette liste de messages en les envoyant vers le processus destination qui va les replacer dans les structures de communication comme si c'était de nouveaux messages. En ce qui concerne les opérations collectives, nous avons simplement implémenté l'algorithme décrit en 7.3.1.3 pour le traitement des migrations.

La mise en place de la migration de threads n'a pas constitué de réelle difficulté car tout avait été prévu lors de la conception des algorithmes et de la gestion mémoire.

8.5 Tolérance aux pannes

Le mécanisme de tolérance aux pannes que nous avons mis en place est basé sur le concept de points de reprise. Son implémentation se fait donc en deux parties : le mécanisme permettant la création d'une image et celui permettant de rétablir une image.

8.5.1 Réalisation de l'image d'une tâche

La méthode de réalisation d'une image d'une tâche est très similaire à celle utilisée pour la migration de tâches et utilise massivement le système d'allocation mémoire.

La première phase dans la réalisation de l'image d'une tâche consiste à la retirer de la file des tâches prêtes de l'ordonnanceur. Ensuite, la réalisation de l'image d'une tâche est principalement gérée par l'allocateur mémoire. En effet, ce dernier dispose de la liste des pages allouées par une tâche. Il est donc capable de réaliser une copie de ces pages sur disque. Pour que la copie de la tâche soit complète, il faut aussi copier la pile de la tâche ainsi que l'ensemble de ces registres. Pour ces données, l'ordonnanceur intervient de nouveau. En effet, ce dernier dispose de structures contenant ces données. Il va donc les copier sur le disque. Ainsi, nous avons pu réaliser l'image d'une tâche. Étant donné que les tâches ne partagent pas directement de données, il n'y a donc pas de souci de données en cours de modification par une autre tâche. En ce qui concerne les messages en cours, nous imposons à l'utilisateur d'avoir fini toutes ses communications asynchrones avant de réaliser l'image.

La gestion des communications étant laissée à la charge de l'utilisateur et la réalisation étant coordonnée entre toutes les tâches, il suffit d'ajouter à ces données quelques informations comme les communicateurs déjà créés pour avoir toutes les informations nécessaires à une reprise ultérieure.

8.5.2 Rétablissement d'une image

Le rétablissement d'une image est guère plus compliqué que la réalisation car c'est encore le système de gestion mémoire qui fait le travail.

Le rétablissement d'une image est l'inverse de la création de l'image. Nous allons utiliser `mmap` pour replacer les segments mémoire précédemment alloués à la même adresse virtuelle que lors de la création de l'image. Puis, nous allons restaurer la pile et les registres au sein d'une structure qui sera fournie à l'ordonnanceur. Enfin, l'ordonnanceur va pouvoir exécuter le thread comme si ce dernier n'avait jamais été arrêté.

En plus du rétablissement de l'image des threads, il faut récréer les communicateurs en utilisant les données sauvegardées.

8.5.3 Redimensionnement

Le redimensionnement est un mécanisme qui permet de reprendre une exécution avec un nombre de processus utilisés différent de l'initial.

Le redimensionnement est en fait un mélange des méthodes présentées pour réaliser le point de reprise et celles de la migration. Ce mécanisme va en fait consister à reprendre les tâches dans un processus différent (ne contenant pas les mêmes tâches) de celui dans lequel elles étaient au moment de la création du point de reprise. Ceci revient à dire que les threads vont devoir migrer

avant de pouvoir reprendre le calcul. Le redimensionnement consiste donc simplement à fournir, au moment du rétablissement de l'image, une nouvelle répartition des threads sur les processus.

Le redimensionnement du nombre de processus est en fait une fonctionnalité qui découle directement de la migration et la création des points de reprise. Il fallait seulement ajouter la possibilité de spécifier une répartition des threads au démarrage pour l'obtenir.

Le travail nécessaire pour la mise en place des mécanismes de tolérance aux pannes avait été déjà grandement fait lors de la mise en place de la migration de tâches et de la gestion mémoire. Cette fonctionnalité a donc nécessité plus une modification de l'existant que la création de nouveaux modules.

Ce chapitre a permis de décrire les solutions logicielles que nous avons mises en œuvre pour adapter l'environnement logiciel du supercalculateur à nos modèles d'exécution. Il convient maintenant de vérifier expérimentalement si la bibliothèque que nous avons construite remplit les objectifs fixés.

Chapitre 9

Évaluation des performances de MPC sur des cas représentatifs

L'objectif de ce chapitre est de montrer que MPC remplit nos objectifs de performance et de robustesse. Pour cela, nous allons soumettre la bibliothèque MPC à un ensemble de tests représentatifs du calcul hautes performances et comparer les temps d'exécution avec d'autres bibliothèques ayant le même modèle de programmation.

L'évaluation de MPC va porter sur quatre grandes familles de tests. Tout d'abord,les tests de portabilité. Ensuite, une batterie de tests élémentaires évaluant certaines caractéristiques de MPC. Puis, des tests construits autour de codes de calcul représentatifs du calcul scientifique. Enfin, nous détaillerons des évaluations de MPC sur des applications réelles.

9.1 Langages et architectures supportées

MPC est une bibliothèque écrite en C en suivant la norme POSIX et utilise quelques fonctionnalités assembleur optionnelles. Elle a donc aisément été portée sur les architectures suivantes : i686/Linux, ia64/Linux, Alpha/OSF1, PowerPC/AIX, Sparc/Solaris. Sur chacune de ces architectures, MPC peut être compilé avec GCC mais aussi avec le compilateur natif qui permet souvent d'améliorer les performances. Néanmoins, certaines optimisations de MPC utilisent des fonctionnalités qui dépendent du système d'exploitation. Par exemple, l'utilisation de la bibliothèque `memset` de placement des données en mémoire n'est possible que sur les architectures Linux disposant de cette bibliothèque. Pour faciliter la compilation de MPC, toutes les particularités du système sont automatiquement détectées lors du script `configure`. Certaines optimisations assembleur n'ont pas été réalisées pour toutes les architectures. En effet, certaines primitives comme les *spinlocks* ou encore les primitives de changement de contexte peuvent être améliorées par l'utilisation de fonctions assembleur qui sont par définition non-portables. Le choix d'écrire MPC en C est en partie motivé par le fait que la majorité des langages propose une interface vers C. MPC a donc été aisément enrichie de plusieurs interfaces. A l'heure actuelle, il existe une interface C/C++, une interface Fortran et une interface C# pour la machine virtuelle Mono[30, 16]. De plus, nous avons récemment réalisé une API compatible POSIX threads qui permet, via une recompilation, d'utiliser toutes les caractéristiques des threads MPC dans une application multithread conçue pour PThread.

9.2 Micro évaluation

La première partie de la présentation des résultats porte sur des tests élémentaires. Ces tests ont pour but d'évaluer les différentes parties de MPC mises en jeu lors de l'exécution d'un code de calcul. Avant de présenter les tests, nous allons détailler les architectures matérielles sur lesquelles nous allons effectuer nos tests. Ensuite, nous présenterons des évaluations comparatives du module de gestion mémoire de MPC. Puis, nous détaillerons une comparaison des performances des fonctions de communication collectives avec d'autres bibliothèques de communication par passage de messages. Enfin, nous ferons de même avec les fonctions de communications point à point.

9.2.1 Les machines de test

Tous les tests de performance que nous allons présenter ont été réalisés sur deux architectures différentes. La première architecture est la grappe TERANOVA. Chaque nœud de cette grappe est une machine NUMA Bull NovaScale 5160 (voir 2.4.3) équipée de 16 processeurs Itanium2 Madison 1.6GHz. Ces nœuds sont dotés de 64Go de mémoire. La seconde architecture, nommée DALTON, est une architecture biprocesseur Xeon 2.66GHz sans HyperThreading avec 512Mo de mémoire.

9.2.2 Gestion mémoire

Pour évaluer les performances de notre politique d'allocation mémoire, nous nous sommes comparés à l'allocateur standard et à celui de la bibliothèque HOARD. En effet, HOARD est un concurrent de choix car c'est une bibliothèque d'allocation optimisée pour applications multithreads. Le test de performance que nous avons utilisé vient aussi de la bibliothèque HOARD. Il est nommé *passive-false*[10]. Ce test évalue les capacités d'un allocateur à éviter le faux partage que nous introduisons volontairement au départ. En effet, le thread principal va commencer par allouer de petits objets pour chaque thread. Ces objets vont constituer l'objet de départ de chaque thread. Ensuite, les threads vont successivement allouer un objet, écrire dans cet objet puis le libérer. Le test dispose de deux paramètres en plus du nombre de threads et de la taille de l'objet. Ces deux paramètres déterminent le nombre de fois où la succession allocation/écriture/libération est effectuée pour chaque thread. Le premier paramètre est le nombre d'itérations. Ce nombre est constant quel que soit le nombre de threads. Le second paramètre est le nombre de répétitions. Cette valeur est divisée par le nombre de threads. On effectue donc le même nombre de successions allocation/écriture/libération quel que soit le nombre de threads. Le test de *passive-false* est donc parfaitement extensible en théorie.

La figure 9.1 représente le temps d'exécution de ce cas test pour des tailles d'objet de 1 et de 512 octets. Pour le cas des objets de taille 1, nous effectuons $1\,000$ itérations sur chaque thread avec $1\,000\,000/nb\_thread$ répétitions par itération. En ce qui concerne le cas 512, nous effectuons $1\,000$ itérations par thread mais seulement $1\,000/nb\_thread$ répétitions par itération. Les résultats obtenus avec ces tests sont très intéressants. En effet, ils confirment tout d'abord l'intérêt d'un allocateur optimisé pour les applications multithreads et ce quelle que soit l'architecture. Les résultats sur architecture NUMA sont très intéressants car ils démontrent qu'un allocateur spécifique au multithreading ne suffit pas pour les architectures NUMA. Il faut en plus que cet allocateur soit optimisé NUMA. En effet, sur architecture NUMA, la différence entre HOARD et MPC est significative et peut atteindre un facteur supérieur à dix alors que sur architecture à accès mémoire

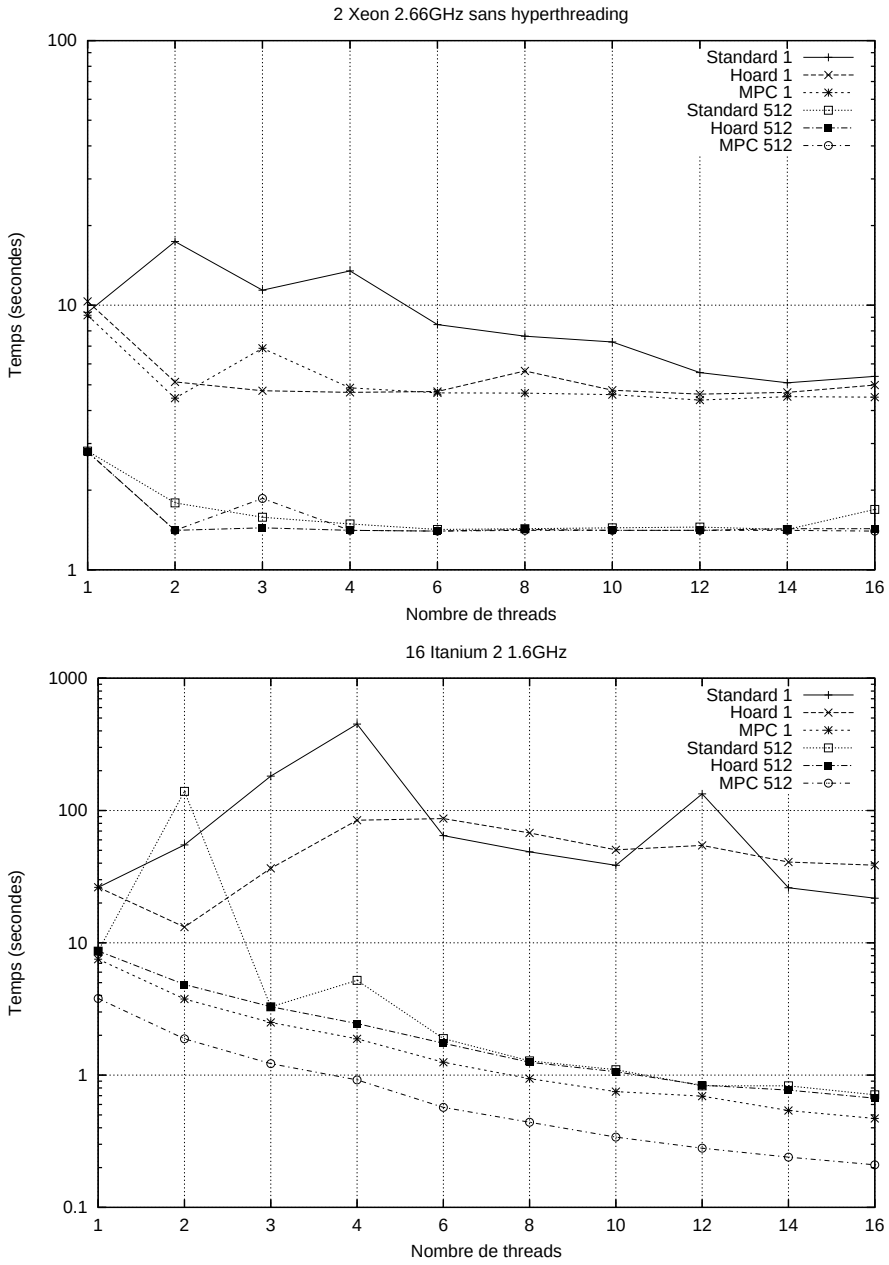


FIG. 9.1 – Comparaison des allocations mémoires

uniforme, les performances de HOARD et de MPC sont comparables.

Les tests d'évaluation de l'allocateur mémoire optimisé montrent l'intérêt de l'approche que nous avons choisie. En effet, cette approche nous permet d'assurer la portabilité des performances de la politique d'allocation optimisée pour le multithreading.

9.2.3 Communications collectives

Le test suivant va évaluer les opérations collectives de type barrière, réduction et diffusion. Comme nous l'avons vu précédemment, toutes ces opérations sont construites sur le même schéma (en fait c'est la même portion de code qui gère les trois opérations). Les résultats présentés ici ne concernent que la barrière, les autres opérations ayant un comportement similaire. Le test de communication collective va consister en la répétition (1 000 fois) d'une barrière. Le nombre de threads impliqués dans cette barrière va varier de 2 à 256 threads.

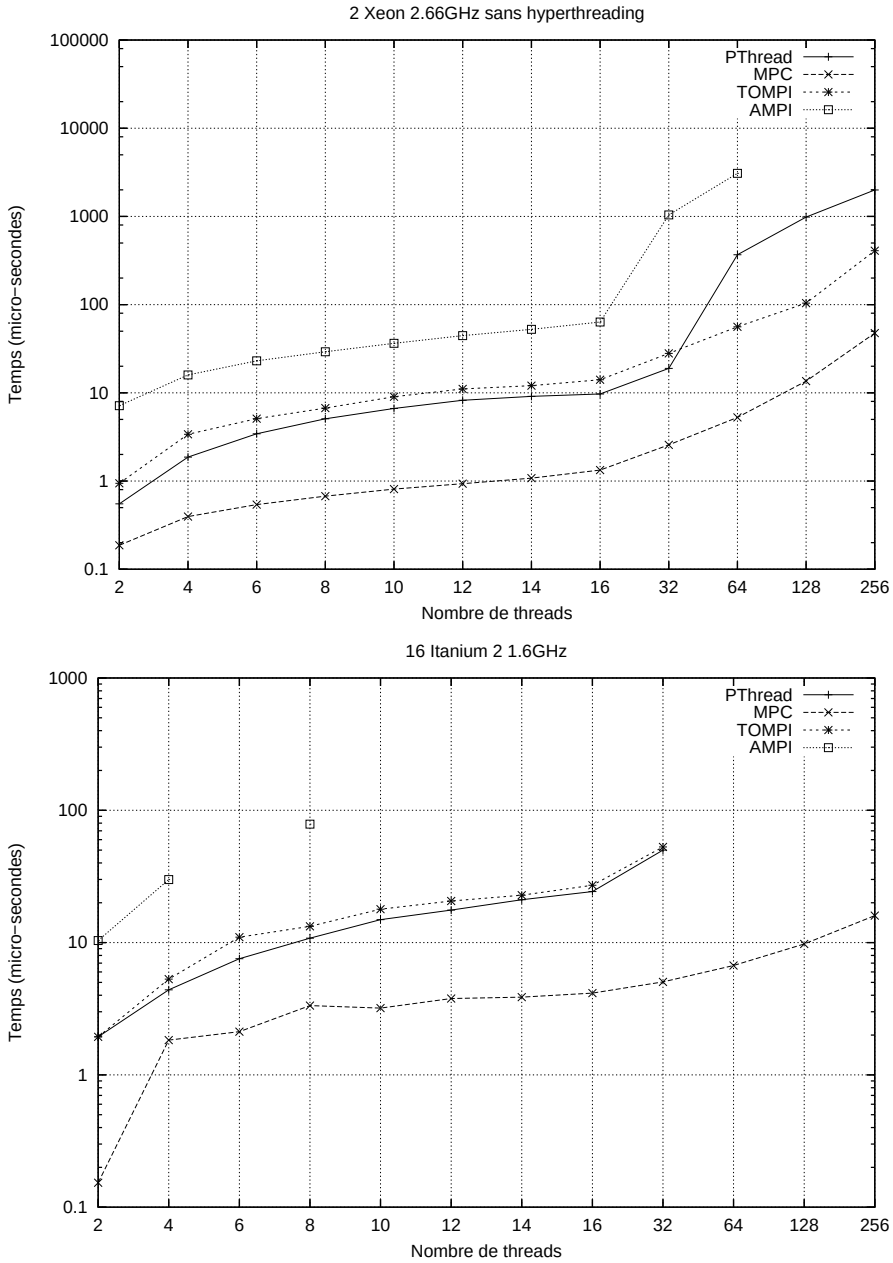


FIG. 9.2 – Comparaison des barrières en threads

Lors de ce test, nous comparons quatre approches. La première utilise les barrières de la bibliothèque de threads POSIX présente sur la machine. La seconde utilise MPC pour effectuer les barrières. La troisième utilise TOMPI et la quatrième AMPI qui sont des implémentations utilisant les threads de la norme MPI (voir 4.2.2). La figure 9.2 présente donc une comparaison des temps d'exécution d'une barrière en fonction du nombre de threads pour chaque approche. On constate que l'approche MPC offre toujours le meilleur temps d'exécution. On peut donc dire que les choix qui ont guidé la conception des opérations collectives se sont avérés judicieux. On remarque aussi que tous les tests AMPI n'ont pu être menés à terme sans doute à cause d'un problème d'implémentation dans AMPI. Enfin, les tests PThread et TOMPI sur Itanium n'ont pu être faits avec plus de 32 threads suite à un problème dans la bibliothèque de threads POSIX qui ne supportait pas plus d'une trentaine de threads au moment des tests. On constate donc un bon comportement général des performances de MPC sur les communications collectives.

9.2.4 Communications point à point

Le test que nous avons mis en place pour évaluer les performances des communications point à point est un test classique de *ping-pong* constitué d'une série d'aller-retour de messages. Les comparaisons de performances portent sur quatre implémentations de bibliothèques de communication par passage de messages. Nous allons comparer MPC à AMPI et TOMPI comme précédemment mais aussi avec une implémentation plus classique du standard MPI. Pour la machine DALTON, nous utilisons MPICH2 comme version classique de MPI et pour la machine TERANOVA, nous utilisons MPIBull qui est une version "constructeur" fortement optimisée pour les architectures NUMA Bull NovaScale.

Les performances obtenues pour les communications point à point en terme de latence et de débit, sont données par les figures 9.3 et 9.4. Sur le plan de la latence des communications, on constate que MPC est la meilleure implémentation sur la machine DALTON. En ce qui concerne la machine TERANOVA, les latences de MPC et de MPIBull sont comparables. MPC est donc une implémentation à faible latence en ce qui concerne les communications point à point car elle arrive même à concurrencer des implémentations "constructeur". Les résultats relatifs aux débits atteints lors des communications sont similaires à ceux des latences. On note aussi un pic des performances correspondant à la taille optimale de message pour la réalisation de la copie mémoire. On peut aussi noter une singularité dans la courbe MPIBull pour des tailles de message entre 4Ko et 64Ko. Cette singularité est sans doute le résultat d'un changement de politique en fonction de la taille des messages. En conclusion, on remarque que les performances de MPC en termes de latence et de débit sont très bonnes et sont comparables à des implémentations MPI "constructeur".

Les tests de comparaison des performances sur des opérations élémentaires permettent de confirmer que les choix faits dans la conception du modèle d'exécution de MPC ont été judicieux. En effet, les tests ont montré que MPC était la meilleure implémentation en terme de performance sur tous les tests élémentaires présentés. Néanmoins, la véritable évaluation va commencer avec les tests représentatifs d'applications réelles de calcul scientifique.

9.3 Applications représentatives

Nous allons maintenant évaluer les performances de MPC sur des exemples d'applications représentatives du calcul scientifique. Pour effectuer notre analyse, nous allons comparer MPC aux

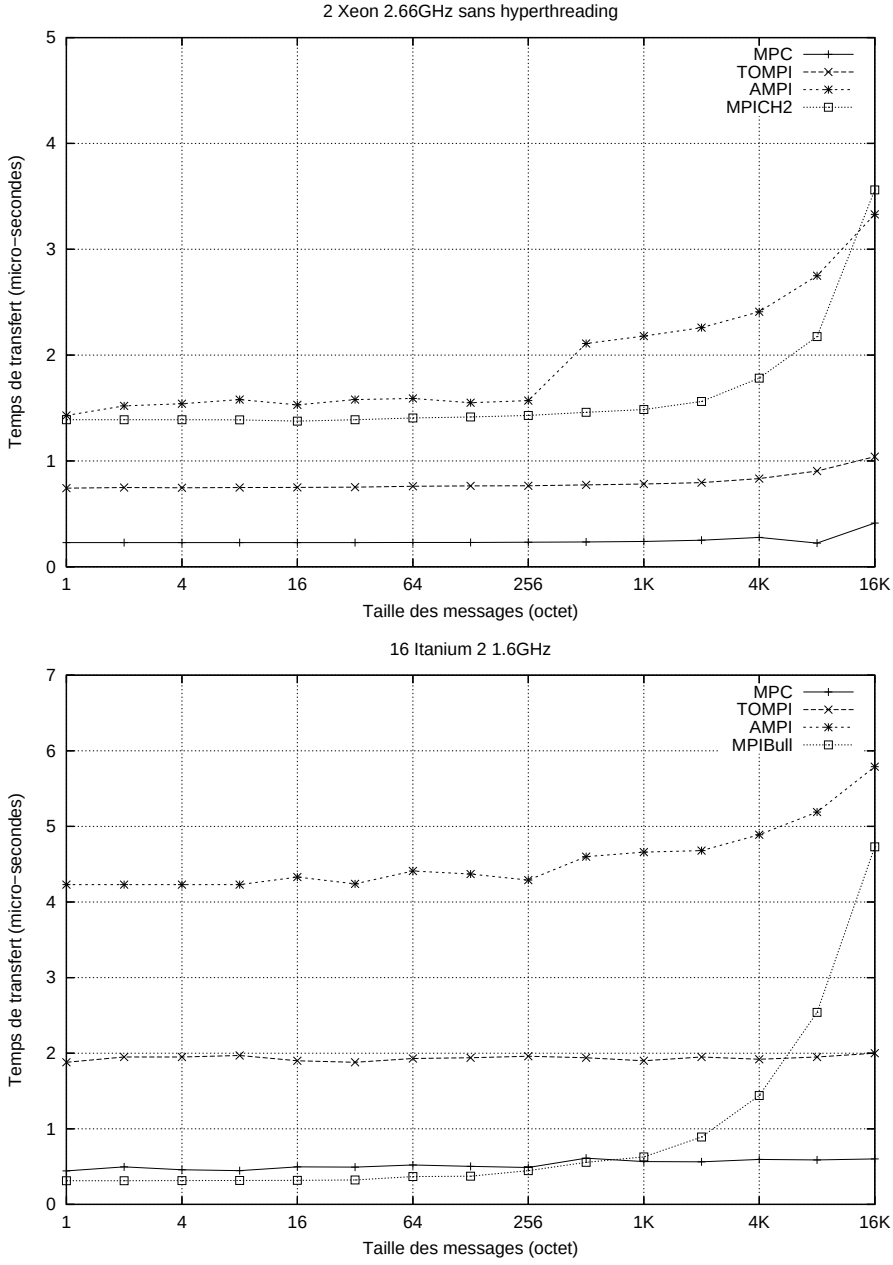


FIG. 9.3 – Comparaison des latences en mémoire partagée

implémentations de MPI disponibles pour chaque architecture. Les valeurs des temps d'exécution des implémentations MPI sont données à titre de référence. Il est important de noter que seuls les cas MPC vont avoir un nombre de tâches qui varie. Ce nombre va toujours être un multiple du nombre de processeurs présents sur la machine. Les exécutions MPI sont, quant à elles, à nombre de tâches constant et égal au nombre de processeurs. Cette batterie de tests va donc permettre d'évaluer l'implémentation de MPC mais aussi la pertinence de la méthode de surcharge ainsi que les gains qu'elle peut apporter. Nous avons pris le parti de ne plus effectuer la comparai-

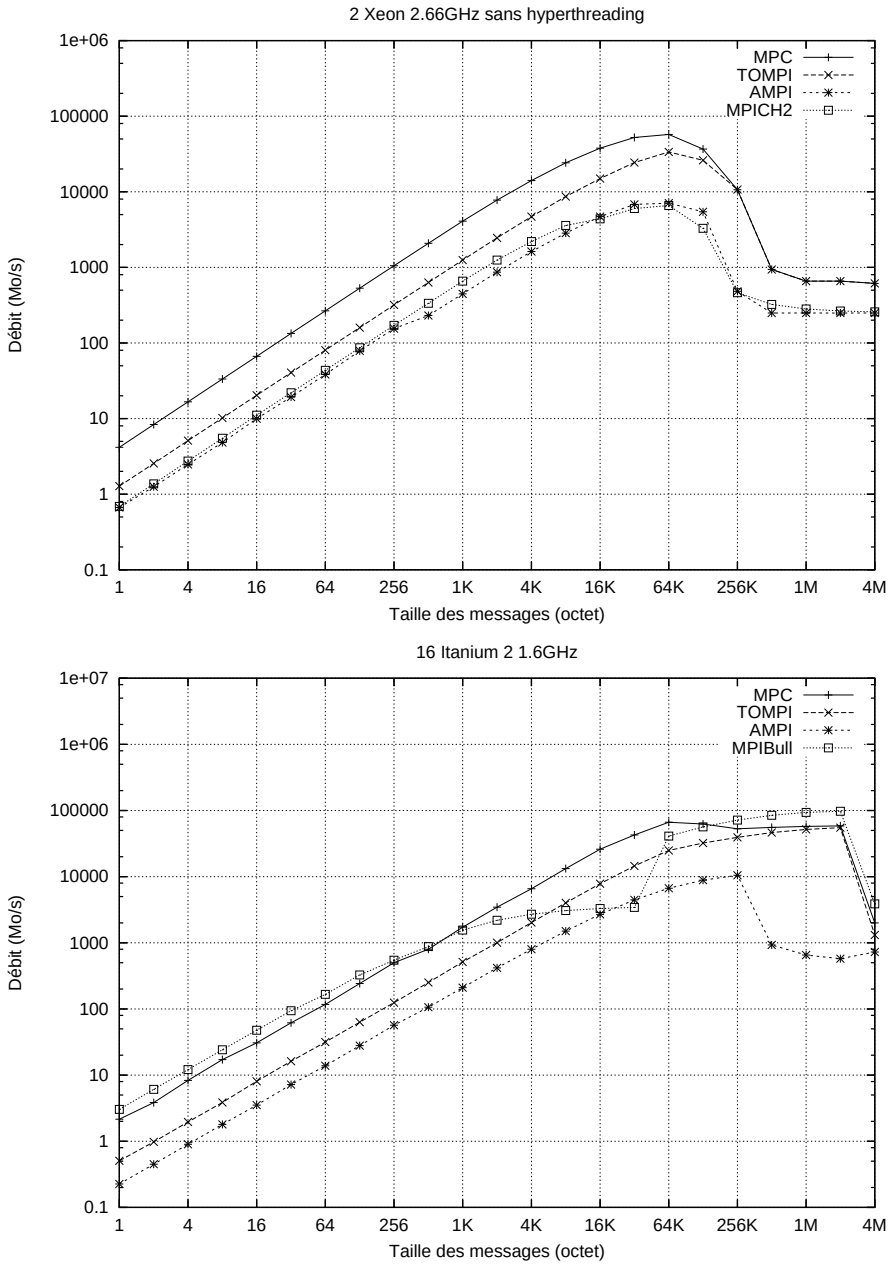


FIG. 9.4 – Comparaison des débits en mémoire partagée

son avec AMPI et TOMPI car la stabilité de ces implémentations ainsi que l'absence de certaines fonctionnalités de communication nous empêchent de réaliser certains tests.

9.3.1 Jacobi

Le cas test Jacobi est extrait des exemples fournis avec MPICH. Ce test consiste à calculer la matrice jacobienne d'une matrice donnée. Il est écrit suivant le paradigme de programmation

SPMD et suit un schéma itératif classique. Ce schéma consiste en la succession de communication des mailles fantômes puis de calcul sur la sous-matrice locale suivie d'une opération collective de type réduction pour évaluer si la condition d'arrêt a été atteinte.

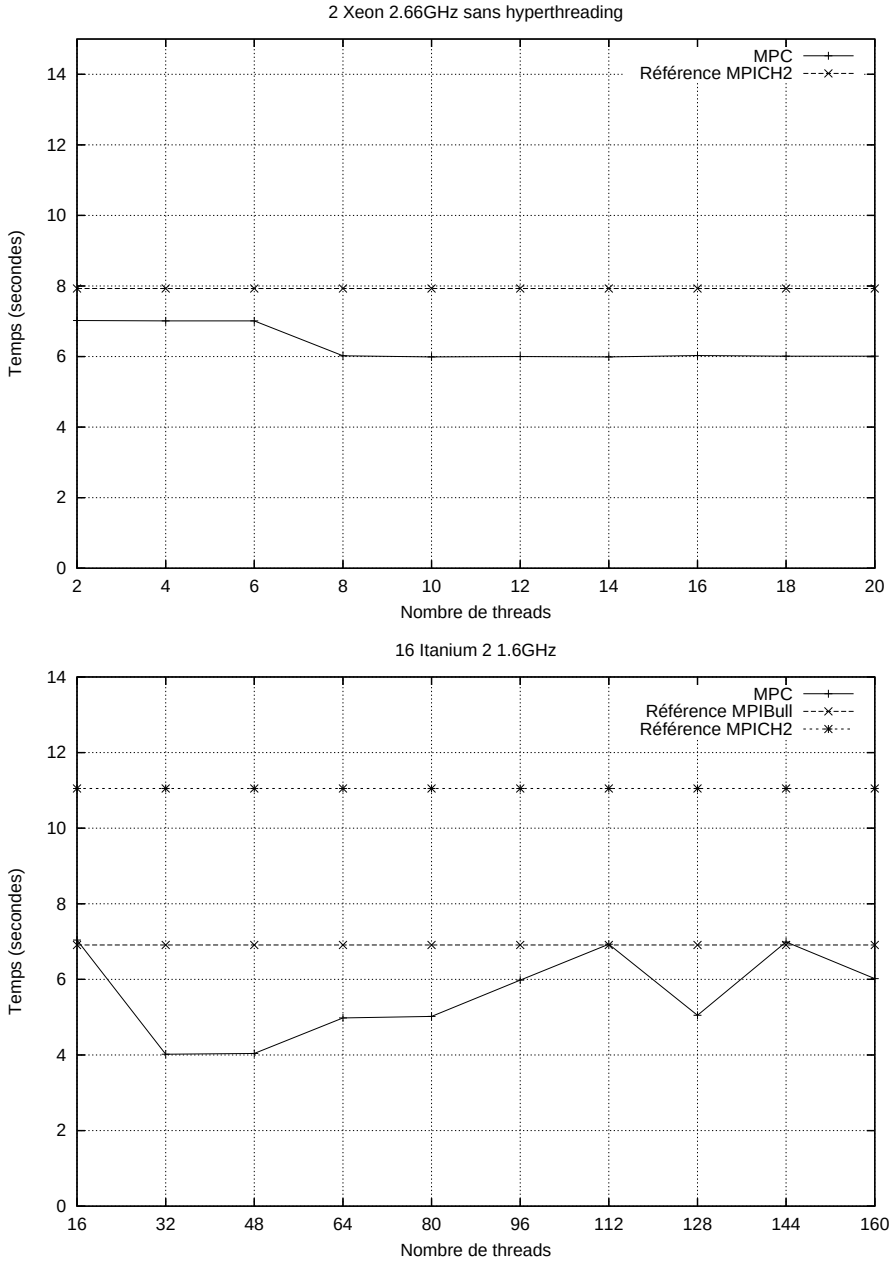


FIG. 9.5 – Comparaison des temps d'exécution du cas test Jacobi

La figure 9.5 présente le temps d'exécution d'un cas constitué d'une matrice globale $2\,048 \times 2\,048$ pour TERANOVA et 500×500 pour DALTON. Comme on peut le voir, l'approche de surcharge est particulièrement intéressante sur ce cas test puisqu'elle permet de gagner jusqu'à 14.7% sur la machine DALTON et jusqu'à 42.9% sur TERANOVA sans aucune modification du code uti-

lisateur. On note, de plus, que les temps d'exécution MPC dans une configuration une tâche par processeur sont inférieurs ou égaux à toutes les autres solutions. Sur cet exemple, MPC permet non seulement d'avoir un gain sur les autres implémentations avec une tâche par processeur mais aussi d'amplifier ce gain avec la méthode de surcharge qui est particulièrement inadaptée voir impossible avec une implémentation MPI classique.

9.3.2 Advection

Le cas test d'advection est une maquette de code de calcul 2D avec schéma d'advection décentré amont «UpWind», explicite en temps. Ce schéma résout l'équation suivante :

$$\begin{cases} \frac{\partial \rho}{\partial t} + v_x \frac{\partial \rho}{\partial x} + v_y \frac{\partial \rho}{\partial y} = 0 & \text{sur } \Omega = [0,1]^2 \\ \rho(0,x,y) = \rho_0(x,y) \\ \text{Conditions aux limites} \end{cases}$$

Ce cas test est codé en suivant le paradigme de programmation SPMD. Comme pour le test Jacobi, le cas d'advection suit un schéma itératif constitué d'une phase de communication suivie d'une phase de calcul et conclut par une réduction pour maintenir la valeur du temps physique uniforme entre tous les sous-domaines (voir pour plus de détails B.1).

La figure 9.6 présente le temps d'exécution d'un cas avec un domaine global de $2\,000 \times 1\,000$ mailles pour TERANOVA et 500×500 mailles pour DALTON. Comme on peut le voir, l'approche de surcharge apporte aussi sur ce cas car elle permet de gagner, en terme de temps d'exécution, jusqu'à 25.8% sur la machine DALTON et jusqu'à 23.6% sur TERANOVA. On note, de plus, que les temps d'exécution MPC sont inférieurs ou égaux à toutes les autres solutions sauf pour les valeurs initiales de MPC sur la machine DALTON.

9.3.3 Conduction

Le cas test de conduction est une maquette de code de calcul 2D avec schéma de conduction implicite en temps avec résolution par gradient conjugué préconditionné par la diagonale. Ce schéma résout l'équation suivante :

$$\begin{cases} \frac{\partial T}{\partial t} - div(K \bigtriangledown T) = f & (t,x,y) \in [0,+\infty[\times[0,1]^2 \\ T(0,x,y) = T_0(x,y) \\ \text{Conditions aux limites} \end{cases}$$

Ce cas test est codé en suivant le paradigme de programmation SPMD. Il se distingue des deux cas précédents par le fait qu'à chaque itération, on utilise la méthode de gradient conjugué. Cette dernière est une méthode itérative qui nécessite de faire des produits matrice $\times$ vecteur et des réductions (produit scalaire). Cette méthode engendre donc un grand nombre de messages lors de la phase de résolution du système, ainsi qu'un grand nombre de synchronisations. (voir pour plus de détails B.2)

La figure 9.7 présente le temps d'exécution d'un cas avec un domaine global de $4\,000 \times 1\,000$ mailles pour TERANOVA et 1500×500 mailles pour DALTON. Comme on peut le voir, l'approche de surcharge apporte aussi sur ce cas car elle permet de gagner, en terme de temps d'exécution, jusqu'à 14.8% sur la machine DALTON et jusqu'à 6% sur TERANOVA. Néanmoins, les gains sont moins importants du fait du schéma utilisé qui est très gourmand en communication dans la phase

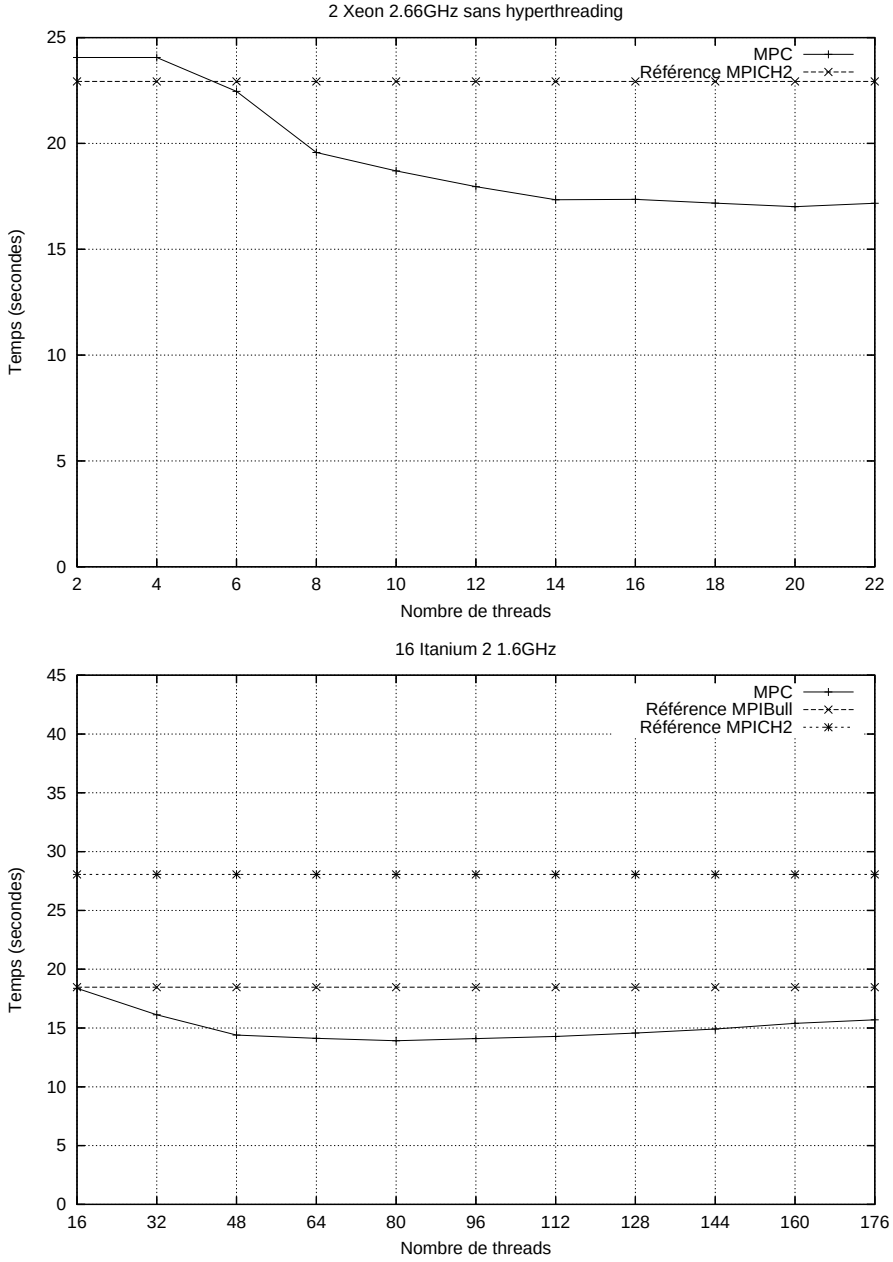


FIG. 9.6 – Comparaison des temps d'exécution du cas test Advection

de résolution par gradient conjugué. On note tout de même que les temps d'exécution MPC sont inférieurs ou égaux à toutes les autres solutions.

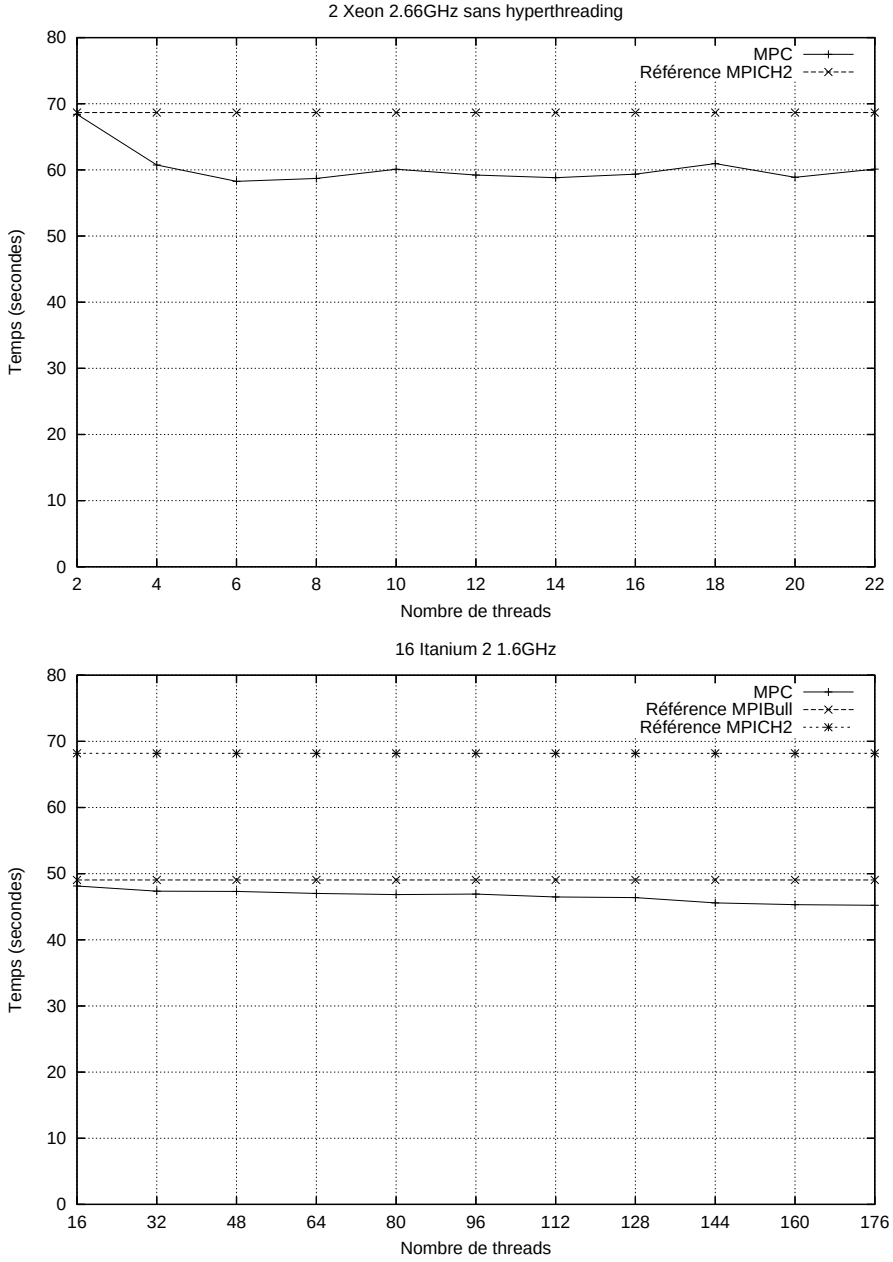


FIG. 9.7 – Comparaison des temps d'exécution du cas test Conduction

9.4 Applications de production

Nous allons conclure cette série d'évaluation de MPC par une évaluation sur des applications réelles. Ces applications sont à la base destinées à être utilisées en contexte MPI. Elles ont donc été modifiées pour pouvoir utiliser MPC. La principale difficulté de cette migration a été la suppression des variables locales qui rendaient certaines fonctions non compatibles avec une utilisation multithread. De plus, certaines fonctionnalités de MPC ont dû être désactivées dans certains cas

pour des raisons de performances ou de conflits avec la sémantique de MPC. Les évaluations vont porter sur quatre codes : un code Fortran, un code C#, un code AMR et un code équation des ondes ordre élevé. Pour chacune de ces évaluations, nous préciserons les fonctionnalités de MPC qui ont été désactivées ainsi que quelques résultats illustrant l'intérêt de l'utilisation de MPC.

9.4.1 Code Fortran de sismique : PRODIF

PRODIF est un code parallèle Fortran SPMD à maillage structuré qui résout le système d'équations de l'Elastodynamique (géophysique, sismologie) dans un milieu tridimensionnel. Ce programme utilise une méthode de différences finies d'ordre élevé, explicite en temps, pour calculer le déplacement sismique en tout point d'un maillage structuré. Des développements récents, réalisés lors d'une thèse, ont permis de passer d'un système en coordonnées cartésiennes à un système en coordonnées curvilignes. Cette évolution permet de maintenant prendre en compte tout relief dans le modèle numérique (via une déformation adaptée de celui-ci). Par ailleurs, la version en coordonnées cartésiennes s'est vu ajouter de nouvelles conditions aux limites absorbantes, basées sur la méthode PML (Perfectly Matched Layers) de Bérenger (1994), bien plus efficace que les méthodes d'éponge numérique (Cerjan et al, 1985) ou paraxiales (Higdon,1991) utilisées jusqu'alors.

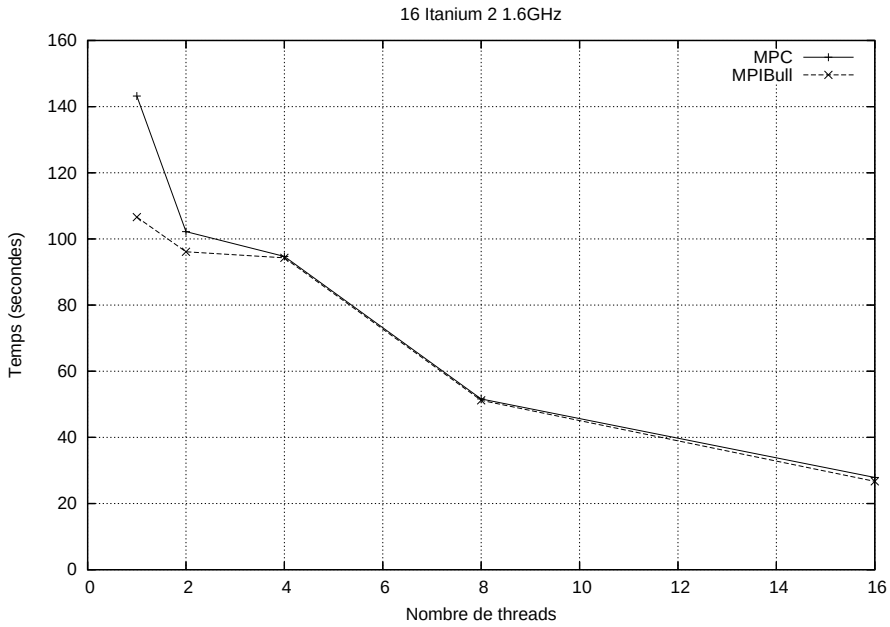


FIG. 9.8 – Tests d'extensibilité du code Fortran de sismique : PRODIF (4.7 millions de mailles)

Le code PRODIF a été testé sur plusieurs nœuds de la machine TERANOVA. La figure 9.8 présente les temps d'exécution de ces tests. PRODIF est un code Fortran bien optimisé qui ne tire pas vraiment avantage de la méthode de surcharge. Néanmoins, ce test est très intéressant car il est très rare de voir l'utilisation de threads pour la parallélisation de codes Fortran. Cette évaluation démontre surtout la possibilité d'utiliser MPC pour paralléliser des codes Fortran.

9.4.2 Code C# d'advection/striation

Ce test est une évaluation de l'interface C# de MPC. L'interface C# de MPC a nécessité un gros travail de la partie thread de MPC. En effet, Mono est une machine virtuelle qui utilise en interne des threads. Cette utilisation des threads de la bibliothèque Mono ajouté à ceux de MPC a posé de nombreux problèmes. Pour les résoudre, nous avons doté MPC d'une interface POSIX thread qui nous a permis de forcer Mono à utiliser les threads MPC pour son fonctionnement interne. Ce travail de création d'une interface POSIX thread pour MPC a été réalisé lors d'un stage. Le code C# d'advection est donc un code parallèle qui peut s'exécuter en environnement de grappe multiprocesseur grâce à l'intégration, au sein de MPC, de la machine virtuelle Mono qui est conçue pour être facilement embarquable dans une application. C'est une approche assez novatrice car elle montre que l'on peut faire du calcul scientifique parallèle avec un langage compilé à la volée comme on l'aurait fait avec un langage compilé comme Fortran ou C. De manière identique au code précédent, l'évaluation a montré une bonne extensibilité de la méthode et aucun gain par effet de cache n'a été constaté.

9.4.3 Code C++ AMR d'hydrodynamique : HERA

Depuis quelques années, des études sont engagées au CEA/DAM sur des codes de calcul utilisant des maillages AMR. HERA(Hydrodynamique Euler Raffinement Adaptatif)[37] est une plate-forme multi-physique AMR développée au CEA/DAM. Le logiciel simule des écoulements compressibles multifluides (domaine de la Dynamique Rapide) en plusieurs dimensions d'espace, avec couplage à des modèles physiques variés, comme des phénomènes thermiques de diffusion non linéaires. HERA est composé principalement de 300 000 lignes de C++, mais fait également appel à des langages interprétés comme Python ou C#. La méthode de parallélisation choisie est l'approche SPMD par décomposition de domaines. La parallélisation des schémas numériques par cette approche est classique : aux points de synchronisation des algorithmes, pour prendre en compte les contributions des sous-domaines voisins, des échanges de messages s'opèrent pour mettre à jour les bords de chaque sous-domaine. En revanche, l'approche AMR couplée à la décomposition de domaine induit des déséquilibrages de charge entre les tâches parallèles. Le maillage s'adaptant au cours de la simulation, les charges de travail entre les tâches vont évoluer au cours du calcul.

L'implémentation de MPC utilisée par le code AMR est une version bridée du fait de certaines incompatibilités du code HERA avec le multithreading. Cette incompatibilité fait que l'on ne peut utiliser plusieurs processeurs virtuels. Les tests présentés ici sont donc monoprocesseurs. De plus, le code HERA utilise des allocations par `realloc` successifs. Comme nous l'avons déjà mentionné, ce type d'allocation est pour le moment problématique (voir 7.2) pour notre allocateur optimisé. C'est pourquoi nous avons choisi de désactiver l'allocateur. Ceci a aussi pour effet de désactiver les méthodes de migration et de tolérance aux pannes.

La figure 9.9 représente les temps d'exécution pour deux cas tests sur un nœud de la machine TERA10(voir 2.4.3). Le premier cas test est un cas hydrodynamique explicite ordre 5 avec 1.5 millions de mailles (proche du cas test advection). Le deuxième cas test est un cas de conduction thermique implicite avec utilisation d'un solveur gradient conjugué préconditionné par Cholesky sur un maillage de 770 000 mailles. Pour ces deux cas tests, nous avons effectué une évaluation de la méthode de surcharge en monoprocesseur en utilisant une décomposition en sous-domaines suivant une dimension et suivant deux dimensions. Comme on peut le voir, la méthode de surcharge

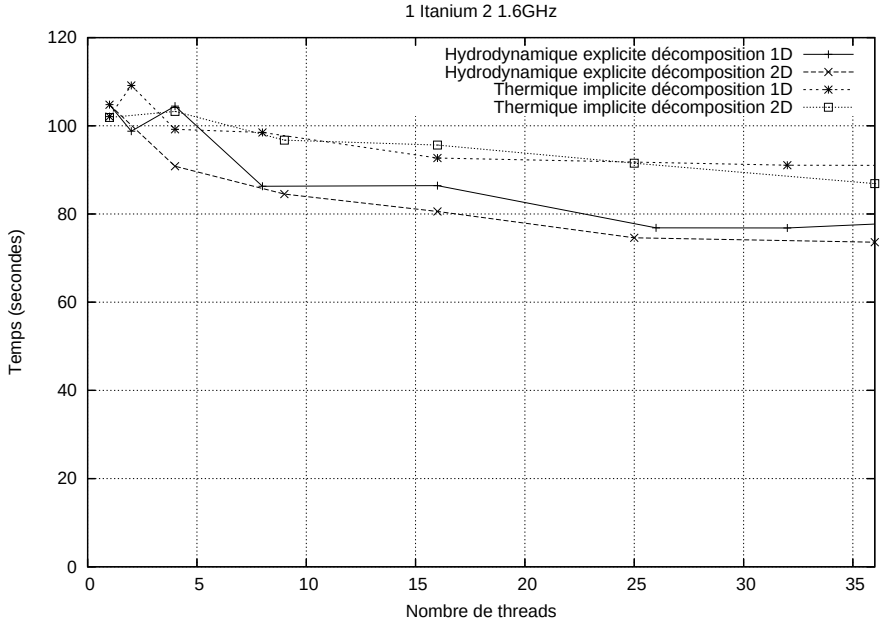


FIG. 9.9 – Cas AMR figé mono matériau

apporte un gain en performance de près de 30% sur le cas hydrodynamique et d'environ 10% sur le cas conduction thermique implicite. Le gain moindre dans le cas conduction thermique provient en partie du préconditionnement. En effet, ce dernier a une influence directe sur le nombre d'itérations nécessaires à la méthode de gradient conjugué. Or, le préconditionnement de Cholesky est dégradé par la parallélisation de la méthode. Cette dégradation est fonction du nombre de sous-domaines. Ceci implique que le nombre d'itérations de la méthode gradient conjugué va augmenter en fonction du nombre de sous-domaines. Il y a donc de plus en plus de calculs à faire au fur et à mesure que l'on surdécoupe le domaine de calcul. On peut donc dire que bien que l'on ait à effectuer une plus grande quantité de calculs, la méthode de surcharge permet tout de même un gain non négligeable.

9.4.4 Évaluation à grande échelle en calcul teraflopique : le code infrason Dragst-R

Le test présenté ici est une évaluation grande échelle de MPC avec un code réel. Ce code permet de simuler la propagation des ondes infrasonores en utilisant des schémas d'ordres élevés[52].

Ce test consiste en la simulation de propagation en trois dimensions d'ondes infrasonores sur une région grande comme huit fois la France et jusqu'à une altitude de 200km. Cette étude a été réalisée dans le cadre de la surveillance du traité d'interdiction complète des essais nucléaires(TICE). Le volume était découpé en 10 milliards de mailles autorisant une finesse de détails de 400m. Le modèle est basé sur les données recueillies lors d'une explosion chimique expérimentale non nucléaire réalisée au Nouveau-Mexique et pour laquelle le CEA était partenaire. Ce test a permis de tester MPC sur $1\,936$ processeurs des $8\,704$ processeurs Itanium2 de TERA10(voir 2.4.3) pendant plusieurs heures de calcul. Nous n'avons pas effectué des tests de performances de la surcharge et de la tolérance aux pannes lors de cette expérimentation grande échelle. Néanmoins, un cas similaire à plus petite échelle a permis de valider la méthode de tolérance aux pannes sur

quelques nœuds de la machine TERANOVA. Le code équations des ondes ayant été très optimisé au niveau de la gestion du cache et des communications, la méthode de surcharge ne permet pas de réels gains sur cette application.

L'évaluation présentée ici a permis de valider la robustesse de MPC dans une configuration utilisant un grand nombre de processeurs sur une durée de calcul élevée.

Ce chapitre a permis d'évaluer les performances de MPC sur des cas tests élémentaires couramment utilisés pour évaluer les bibliothèques de communication. Ces résultats ont montré les bonnes performances MPC. Ces tests, bien que classiques, ne reflètent pas l'utilisation réelle. Nous avons donc évalué MPC sur des cas tests représentatifs. Ces évaluations ont permis de confirmer les résultats obtenus précédemment mais aussi de montrer l'intérêt de la méthode de surcharge. La troisième partie de l'évaluation de MPC a porté sur des codes de calcul réels. Ces évaluations ont montré la robustesse de MPC ainsi que l'intérêt de l'approche de surcharge sur un code de calcul complexe comme le code HERA. Bien que ces évaluations aient confirmé certaines faiblesses au niveau de l'allocation mémoire, nous pouvons dire que la bibliothèque MPC s'est montrée à la hauteur de ses ambitions.

Chapitre 10

Conclusions et perspectives

10.1 Conclusions

Avec la diversification des plates-formes matérielles due à la recrudescence de processeurs multicœurs et/ou multithreads et au développement des machines multiprocesseurs à hiérarchie mémoire (NUMA), le visage du calcul scientifique hautes performances est en train de changer. En effet, les outils logiciels utilisés jusqu'alors, principalement des bibliothèques de communication, montrent leurs limites et l'on doit s'orienter vers de nouvelles méthodes qui tiennent compte de la complexité des architectures.

Les travaux réalisés dans cette thèse se situent dans le contexte du calcul scientifique hautes performances et ont été menés conjointement au laboratoire PNP (Pôle Numérique et Prospectives) au CEA/DAM Île de France et au sein du projet INRIA RUNTIME. Le laboratoire PNP a pour objectif l'élaboration de codes de calculs scientifiques avant-gardistes et hautes performances tandis que l'équipe RUNTIME a pour objectif l'étude, la conception et la réalisation de supports exécutifs destinés à servir d'intergiciels parallèles de haut niveau.

L'objectif de cette thèse était de concevoir des méthodes adaptées au calcul scientifique parallèle pour permettre d'améliorer les performances des codes de calculs sur les architectures actuelles et à venir de supercalculateurs.

La première partie de notre contribution est la proposition d'une architecture complète d'environnement de programmation destinée au calcul scientifique hautes performances. Le défi majeur que nous cherchions à relever était d'aboutir sur un outil de programmation riche et flexible permettant une exploitation optimale des multiples configurations matérielles possibles.

La définition de notre architecture a impliqué de devoir repenser la façon dont la parallélisation des codes était effectuée au sein des bibliothèques de communication tout en conservant une interface utilisateur proche de celles qui existent actuellement pour faciliter l'évolution des codes. Nous avons axé notre démarche autour des axes suivants :

- ordonnancement et placement des tâches de calcul ;
- gestion de la localité mémoire ;
- optimisation des applications ;
- équilibrage dynamique de charge par migration de tâche.

Ceci nous a permis d'encapsuler à bas niveau tous les aspects liés à la complexité de l'architecture sous-jacente et d'offrir une approche d'optimisation de codes.

Cette architecture a été mise en œuvre avec succès pour donner naissance à un environnement

de programmation doté d'une interface proche de celle du standard MPI et nommé MPC.

La bibliothèque MPC a été évaluée dans de nombreuses configurations et comparée aux autres environnements de programmation utilisant le même schéma de communication. Les résultats tirés de ces évaluations montrent le bien fondé de notre démarche de recherche. Les performances obtenues sur les différentes architectures à notre disposition sont de tout premier plan.

MPC a été évaluée sur une application Fortran de Dominique Rodrigues au CEA/DAM Île de France. Cette évaluation montre la faisabilité d'une application parallèle hautes performances basée sur le multithreading pour Fortran. Jusqu'alors, seul OpenMP permettait l'utilisation de threads pour le langage Fortran. Toutes les fonctionnalités de MPC n'ont pas pu être évaluées, néanmoins, MPC s'est montrée une alternative sérieuse aux versions optimisées constructeur de MPI.

MPC a été utilisée pour la parallélisation d'un code C#. Cette évaluation avait pour but de voir si l'utilisation d'un langage nécessitant une machine virtuelle est viable dans un environnement de calcul scientifique. MPC s'est avérée une plate-forme de choix pour l'implémentation d'une solution parallèle pour cette évaluation. L'intégration de la machine virtuelle Mono dans MPC a permis en outre de tester l'interface POSIX threads de MPC qui a été développée pour l'occasion. Cette évaluation permet de dire que MPC peut être efficacement couplée avec d'autres bibliothèques complexes utilisant le multithreading.

MPC a été utilisée pour réaliser une évaluation très grande échelle du code Dragst-R. Cette évaluation qui a utilisé le supercalculateur de CEA/DAM Île de France TERA10 a permis de mettre en évidence la robustesse de MPC dans les conditions réelles voir même extrêmes du calcul scientifique hautes performances. La version actuelle de MPC est donc un prototype fiable et performant pour une utilisation en calcul hautes performances.

Enfin, MPC est une des bibliothèques de communication utilisée dans la conception d'un code de calcul maillage adaptatif au CEA/DAM Île de France. Historiquement, c'est même pour ce code qu'a été pensée MPC. En effet, il regroupe toutes les difficultés liées aux grands codes de calcul. Ce code est tout d'abord fortement déséquilibré à cause de la méthode de raffinement adaptatif qu'il utilise. Ensuite, c'est un gros code (300 000 lignes de code) ayant un accès nécessitant de nombreuses indirections pour accéder aux données ce qui rend l'optimisation des accès en mémoire cache difficile. Enfin, il fait une utilisation très intensive des communications. Ce code est donc le candidat idéal pour l'évaluation d'une bibliothèque de communication. La totalité des fonctionnalités de MPC n'ont pas encore été testées, néanmoins, les évaluations préliminaires ont montré que MPC et ses méthodes d'optimisation sont une bonne approche pour l'optimisation des codes complexes.

10.2 Perspectives

Notre travail a permis de reconsidérer l'approche de parallélisation des codes de calcul scientifique. MPC permet de concevoir des applications de calcul scientifique très flexibles, performantes et tolérantes aux pannes pour un effort limité de portage si le code est déjà parallélisé avec MPI. C'est donc un réel apport pour la communauté des développeurs de codes.

MPC est une bibliothèque amenée à évoluer tout d'abord vers plus de robustesse et de performance. En effet, les différentes limitations décrites dans ce document devront être levées pour permettre une intégration de MPC dans les grands projets de code de simulations développés au CEA/DAM et en particulier le code hydrodynamique d'interaction laser plasma que nous avons

présenté. MPC devra donc passer d'un travail de thèse expérimental à une bibliothèque de communication utilisée dans un contexte industriel.

Lors des diverses présentations que nous avons faites, en particulier dans le cadre de coopération entre Bull et le CEA, MPC a retenu l'attention. Ceci a conduit à divers échanges avec les équipes de Bull chargées de la conception des systèmes d'exploitation et des architectures de supercalculateurs. Comme nous l'avons souvent mentionné, la gestion du placement mémoire sur ces architectures est primordiale pour obtenir de bonnes performances. Une des perspectives à court terme de ces échanges est donc la fourniture par Bull d'une primitive de migration de pages mémoire qui va permettre d'enrichir la politique de gestion mémoire de MPC.

De plus, MPC va faire l'objet de divers évaluations approfondies sur des architectures d'avant-garde dans le cadre du projet FAME2[2], projet structurant du pôle mondial de compétitivité SYSTEM@TIC PARIS-REGION.

A plus long terme, le modèle utilisé va être amené à évoluer pour tenir compte de nouvelles techniques d'optimisation de codes grâce notamment à l'extraction d'information par le compilateur ou bien en cours d'exécution. On peut citer l'intégration de méthodes comme la *Software-based Speculative Pre-computation* introduite par Intel dans son compilateur version 9.0[61, 60]. Cette optimisation génère un thread chargé du préchargement des données qui va s'exécuter de concert avec le thread chargé du calcul. Pour qu'un code utilisant MPC puisse bénéficier cette optimisation, il est nécessaire que les threads générés utilisent les primitives MPC de gestion des threads. Pour cela, il faudra adaptater de l'ordonnanceur de threads de MPC pour qu'il gère et place correctement le thread généré en fonction du thread de calcul qui lui est associé.

Bibliographie

[1] « CHARM++ ». `http://charm.cs.uiuc.edu/`.

[2] *Site de FAME2*. `http://www.fame2.org/`.

[3] « Site web de BULL. ». `http://www.bull.com/`.

[4] « Site web de Quadrics. ». `http://www.quadrics.com/`.

[5] « Site web TOP500 supercomputers. ». `http://www.top500.org/`.

[6] Bill ABT, Saurabh DESAI, David P. HOWELL, Inaky PEREZ-GONZALEZ et Dave MCCRACKEN. « Next Generation POSIX Threading Project ». `http://www-124.ibm.com/developerworks/oss/pthreads/`, 2002.

[7] NR ADIGA, G ALMASI, GS ALMASI, Y ARIDOR, R BARIK, D BEECE, R BELLOFATTO, G BHANOT, R BICKFORD, M BLUMRICH, AA BRIGHT et J. « An Overview of the BlueGene/L Supercomputer », 2002.

[8] Gabriel ANTONIU, Luc BOUGÉ et Raymond NAMYST. « An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System ». Dans *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 496–510, London, UK, 1999. Springer-Verlag.

[9] Frank BELLOSA et Martin STECKERMEIER. « The Performance Implications of Locality Information Usage in Shared-Memory Multiprocessors ». *Journal of Parallel and Distributed Computing*, 37(1) :113–121, 1996.

[10] Emery D. BERGER, Kathryn S. MCKINLEY, Robert D. BLUMOFE et Paul R. WILSON. « Hoard : A Scalable Memory Allocator for Multithreaded Applications ». Dans *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, novembre 2000.

[11] Emery D. BERGER, Benjamin G. ZORN et Kathryn S. MCKINLEY. « Composing high-performance memory allocators ». Dans *PLDI '01 : Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 114–124, New York, NY, USA, 2001. ACM Press.

[12] Franck CAPPELLO et Daniel ETIEMBLE. « MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks ». pages 51–51, 2000.

[13] Tony F. CHAN et Tarek P. MATHEW. Domain Decomposition Algorithms. Dans *Acta Numerica 1994*, pages 61–143. Cambridge University Press, 1994.

[14] Jamison D. COLLINS, Hong WANG, Dean M. TULLSEN, Christopher HUGHES, Yong-Fong LEE, Dan LAVERY et John P. SHEN. « Speculative precomputation : long-range prefetching of delinquent loads ». Dans *ISCA '01 : Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25, New York, NY, USA, 2001. ACM Press.

[15] L. DAGUM et R. MENON. « OpenMP : An industry-based API for Shared-Memory programming ». *IEEE Computational Science and Engineering*, 5(1) :46–55, 1998.

[16] M. de ICAZA. *The Mono Project.* `http://www.mono-project.com`.

[17] E. DEMAINE. « A Threads-Only MPI Implementation for the Development of Parallel Programs ». Dans *Proceedings of the 11th International Symposium on High Performance Computing Systems*, pages 153–163, juillet 1997.

[18] Jack DONGARRA, Steven HUSS-LEDERMAN, Steve OTTO, Marc SNIR et David WALKER. *MPI : The Complete Reference*. MIT Press, 1996.

[19] Ulrich DREPPER et Ingo MOLNAR. « The Native POSIX Thread Library for Linux ». `http://people.redhat.com/drepper/nptl-design.pdf`, janvier 2003.

[20] David DUREAU et Hervé JOURDREN. « Parallélisme et équilibrage de charge en hydrodynamique AMR ». *Revue Chocs*, (28) :51–60, octobre 2003.

[21] Ralf S. ENGELSCHALL. « GNU Portable Threads (Pth) ». `http://www.gnu.org/software/pth/`, 1999.

[22] Ralf S. ENGELSCHALL. « Portable Multithreading — The Signal Stack Trick for User-Space Thread Creation ». Dans *USENIX Annual Technical Conference*, pages 239–250, San Diego, California, USA, juin 2000. `http://www.usenix.org/publications/library/proceedings/usenix2000/general/engelschall.html`.

[23] M. FELDMAN. « Intel Threads Its Way to Parallel Programming ». `http://www.hpcwire.com/hpc/857248.html`, 2006.

[24] Y. FENG et E. BERGER. « A locality-improving dynamic memory allocator ». Dans *MSP '05 : Proceedings of the 2005 workshop on Memory system performance*, pages 68–77, New York, NY, USA, 2005. ACM Press.

[25] Michael J. FLYNN, Patrick HUNG et Kevin W. RUDD. « Deep-Submicron Microprocessor Design Issues ». *IEEE Micro*, 19(4) :11–22, 1999.

[26] Ian FOSTER, Carl KESSELMAN et Steven TUECKE. « The Nexus Approach to Integrating Multithreading and Communication ». *Journal of Parallel and Distributed Computing*, 37(1) :70–82, 1996.

[27] David GEER. « Industry Trends : Chip Makers Turn to Multicore Processors ». *Computer*, 38(5) :11–13, 2005.

[28] Al GEIST, Adam BEGUELIN, Jack DONGARRA, Weicheng JIANG, Robert MANCHEK et Vaidy SUNDERAME. *PVM : Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[29] J. GONNORD, P. LECA et F. ROBIN. « 60 mille milliards d'opérations par seconde ». *La Recherche*, 393, 2006.

[30] W. GROPP et E. LUSK. *The Mono Handbook.* `http://www.gotmono.com/docs/`.

[31] William GROPP, Ewing LUSK, Nathan DOSS et Anthony SKJELLUM. « High-performance, portable implementation of the MPI Message Passing Interface Standard ». *Parallel Computing*, 22(6) :789–828, 1996.

[32] William GROPP et Ewing L. LUSK. « Why Are PVM and MPI So Different ? ». Dans *PVM/MPI*, pages 3–10, 1997.

[33] Lance HAMMOND, Basem A. NAYFEH et Kunle OLUKOTUN. « A Single-Chip Multiprocessor ». *Computer*, 30(9) :79–85, 1997.

[34] Chao HUANG, Orion LAWLOR et L. V. KALÉ. « Adaptive MPI ». Dans *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, octobre 2003.

[35] Chao HUANG, Gengbin ZHENG, Sameer KUMAR et Laxmikant V. KALÉ. « Performance Evaluation of Adaptive MPI ». Dans *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, mars 2006.

[36] Dongming JIANG et Jaswinder Pal SINGH. « A Methodology and an Evaluation of the SGI Origin2000 ». Dans *Measurement and Modeling of Computer Systems*, pages 171–181, 1998.

[37] H. JOURDREN. « HERA : A hydrodynamics AMR Platform for multiphysics simulation ». Dans Tomasz (ed.) et al. PLEWA, éditeur, *Adaptive mesh refinement - theory and applications. Lecture Notes in Computational Science and Engineering*, volume 41, pages 283–294, 2005.

[38] Ronald N. KALLA, Balaram SINHAROY et Joel M. TENDLER. « IBM Power5 Chip : A Dual-Core Multithreaded Processor. ». *IEEE Micro*, 24(2) :40–47, 2004.

[39] Vivek KHERA, Jr. P R LAROWE et S C ELLIS. « An Architecture-Independent Analysis of False Sharing ». Rapport Technique, Durham, NC, USA, 1993.

[40] Poonacha KONGETIRA, Kathirgamar AINGARAN et Kunle OLUKOTUN. « Niagara : A 32-Way Multithreaded Sparc Processor ». *IEEE Micro*, 25(2) :21–29, 2005.

[41] Rakesh KUMAR, Norman P. JOUPPI et Dean M. TULLSEN. « Conjoined-Core Chip Multiprocessing ». Dans *MICRO 37 : Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 195–206, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Xavier LEROY. « The LinuxThreads Library ». `http://pauillac.inria.fr/~xleroy/linuxthreads/`, 1996.

[43] J. LIU, B. CHANDRASEKARAN, J. JIANG, S. KINI, W. YU, D. BUNTINAS, P. WYCKOFF et D. PANDA. « Performance Comparison of MPI Implementations over InfiniBand Myrinet and Quadrics », 2003.

[44] Deborah T. MARR, Frank BINNS, David L. HILL, Glenn HINTON, David A. KOUFATY, J. Alan MILLER et Michael UPTON. « Hyper-Threading Technology Architecture and Microarchitecture ». *Intel Technology Journal*, 2002.

[45] Bertrand MELTZ. « Simulation numérique téraflopique en hydrodynamique ». *Revue Chocs*, (28) :25–32, octobre 2003.

[46] M. M. MICHAEL, A. K. NANDA, B-H. LIM et M. L. SCOTT. « Coherence Controller Architectures for SMP-Based CC-NUMA Multiprocessors ». Dans *Proc. of the 24th Annual Int'l Symp. on Computer Architecture (ISCA'97)*, pages 219–228, 1997.

[47] K. MILFELD, C. GUIANG, A. PURKAYASTHA et J. BOISSEAU. « Exploring the Effects of Hyper-Threading on Scientific Applications ». Dans *Cray User Group 2003*, 2003.

[48] Gordon MOORE. « Cramming more components onto integrated circuits ». *Electronics*, 1965.

[49] Raymond NAMYST. « *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières* ». PhD thesis, Université de Lille 1, janvier 1997.

[50] Kunle OLUKOTUN et Lance HAMMOND. « The future of microprocessors ». *Queue*, 3(7) :26–29, 2005.

[51] Fabrizio PETRINI, Wu chun FENG, Adolfy HOISIE, Salvador COLL et Eitan FRACHTENBERG. « The Quadrics Network : High-Performance Clustering Technology ». *IEEE Micro*, 22(1) :46–57, 2002.

[52] S. Del PINO et H. JOURDREN. « Arbitrary high-order schemes for the linear advection and wave equation : application to hydrodynamics and aeroacoustics ». Dans *C. R. Math., Acad, Sci. Paris*, volume 342, pages 441–446, 2006.

[53] Sundeep PRAKASH et Rajive BAGRODIA. « MPI-SIM : Using Parallel Simulation to Evaluate MPI Programs ». Dans *Winter Simulation Conference*, pages 467–474, 1998.

[54] Jeffrey M. SQUYRES et Andrew LUMSDAINE. « A Component Architecture for LAM/MPI ». Dans *Proceedings, 10th European PVM/MPI Users' Group Meeting*, numéro 2840 dans Lecture Notes in Computer Science, pages 379–387, Venice, Italy, 2003. Springer-Verlag.

[55] V. S. SUNDERAM. « PVM : a framework for parallel distributed computing ». *Concurrency, Practice and Experience*, 2(4) :315–340, 1990.

[56] Josep TORRELLAS, Monica S. LAM et John L. HENNESSY. « False Sharing and Spatial Locality in Multiprocessor Caches ». *IEEE Transactions on Computers*, 43(6) :651–663, 1994.

[57] Dean M. TULLSEN, Susan EGGERS et Henry M. LEVY. « Simultaneous Multithreading : Maximizing On-Chip Parallelism ». Dans *Proceedings of the22th Annual International Symposium on Computer Architecture*, 1995.

[58] Theo UNGERER, Borut ROBIC et Jurij SILC. « A survey of processors with explicit multithreading. ». *ACM Comput. Surv.*, 35(1) :29–63, 2003.

[59] Aad J. van der STEEN. « Overview of Recent Supercomputers ». 2005. `http://phase.hpcc.jp/mirrors/top500/ORSC/`.

[60] H. WANG, G. HOFLEHNER, D. LAVERY, S. LIAO, P. WANG et J. SHEN. « Post-pass binary adaptation for software-based speculative precomputation ». Dans *Proceedings of the ACM SIGPLAN 2002*, 2002.

[61] Perry H. WANG, Jamison D. COLLINS, Hong WANG, Dongkeun KIM, Bill GREENE, Kai-Ming CHAN, Aamir B. YUNUS, Terry SYCH, Stephen F. MOORE et John P. SHEN. « Helper Threads via Virtual Multithreading On An Experimental Itanium 2 Processor-based Platform ». Dans *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[62] Wayne YAMAMOTO et Mario NEMIROVSKY. « Increasing superscalar performance through multistreaming ». Dans *PACT '95 : Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pages 49–58, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[63] Youhui ZHANG, Dongsheng WONG et Weimin ZHENG. « User-level checkpoint and recovery for LAM/MPI ». *SIGOPS Oper. Syst. Rev.*, 39(3) :72–81, 2005.

[64] Gengbin ZHENG, Chao HUANG et Laxmikant V. KALÉ. « Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++ ». Dans *OSR special issue on OSes and Runtimes for HEC systems*, to appear.

Annexe A

MPC

A.1 Interface MPC

```
/*************************************************************************
File :    mpc.h
Thu Jul 6 14:20:34 CEST 2006
Author : PERACHE Marc
Comments :
*************************************************************************/
#ifndef __mpc__H
#define __mpc__H

#ifdef __cplusplus
extern "C"
{
#endif

#include "mpcthread.h"
#include <stdlib.h>
#include <unistd.h>

/*Main function redefinition*/
#undef main
#define main mpc_user_main
  int MPC_Main (int argc, char **argv);
  int MPC_User_Main (int argc, char **argv);

/*Sleep functions redefinition*/
  unsigned int sctk_thread_sleep (unsigned int seconds);
#define  sleep sctk_thread_sleep
  int sctk_thread_usleep (unsigned int seconds);
#define usleep sctk_thread_usleep
  int sctk_thread_nanosleep (const struct timespec *req,
                             struct timespec *rem);
#define nanosleep  sctk_thread_nanosleep

  typedef int mpc_msg_count;
  typedef unsigned int mpc_pack_indexes_t;

  typedef unsigned int MPC_Comm;
```

```
  typedef struct
  {
    int task_nb;
    int *task_list;
  } MPC_Group_t;
  typedef MPC_Group_t *MPC_Group;

  extern MPC_Group_t mpc_group_empty;
  extern MPC_Group_t mpc_group_null;

#define MPC_GROUP_EMPTY &mpc_group_empty
#define MPC_GROUP_NULL ((MPC_Group)NULL)

#define MPC_STATUS_SIZE 80
#define MPC_REQUEST_SIZE 30

  typedef unsigned long MPC_Datatype;
  typedef void (*MPC_Op_f) (const void *, void *, size_t, MPC_Datatype);
  typedef void (MPC_User_function) (void *, void *, int *, MPC_Datatype *);

  typedef void (MPC_Handler_function) (MPC_Comm *, int *, ...);
  typedef MPC_Handler_function *MPC_Errhandler;

  typedef struct
  {
    MPC_Op_f func;
    MPC_User_function *u_func;
  } MPC_Op;
#define MPC_OP_INIT {NULL,NULL}

  typedef struct mpc_thread_message_header_s
  {
    int message_tag;
    MPC_Comm communicator;

    int source;
    int destination;

    int myself;
    int *request;
    size_t *req_msg_size;
    long rank;
    size_t msg_size;
  } mpc_thread_message_header_t;

  typedef struct
  {
    mpc_msg_count count;
    mpc_pack_indexes_t *begins;
    mpc_pack_indexes_t *ends;
  } mpc_default_pack_t;

  typedef struct
  {
    int completion_flag;
    void *msg;
```

```
    mpc_thread_message_header_t header;
    size_t msg_size;
    mpc_default_pack_t default_pack;
    int is_null;
  } MPC_Request;

  extern MPC_Request mpc_request_null;
#define MPC_REQUEST_NULL mpc_request_null
#define MPC_COMM_WORLD 0

#define MPC_SUCCESS 0
/* Communication argument parameters */
#define MPC_ERR_BUFFER   1  /* Invalid buffer pointer */
#define MPC_ERR_COUNT    2  /* Invalid count argument */
#define MPC_ERR_TYPE     3  /* Invalid datatype argument */
#define MPC_ERR_TAG      4  /* Invalid tag argument */
#define MPC_ERR_COMM     5  /* Invalid communicator */
#define MPC_ERR_RANK     6  /* Invalid rank */
#define MPC_ERR_ROOT     7  /* Invalid root */
#define MPC_ERR_TRUNCATE 14 /* Message truncated on receive */

/* MPC Objects (other than COMM) */
#define MPC_ERR_GROUP    8  /* Invalid group */
#define MPC_ERR_OP       9  /* Invalid operation */
#define MPC_ERR_REQUEST 19 /* Invalid mpc_request handle */

/* Special topology argument parameters */
#define MPC_ERR_TOPOLOGY 10 /* Invalid topology */
#define MPC_ERR_DIMS     11  /* Invalid dimension argument */

/* All other arguments.     This is a class with many kinds */
#define MPC_ERR_ARG      12  /* Invalid argument */

/* Other errors that are not simply an invalid argument */
#define MPC_ERR_OTHER  15  /* Other error; use Error_string */

#define MPC_ERR_UNKNOWN 13 /* Unknown error */
#define MPC_ERR_INTERN 16  /* Internal error code         */

/* Multiple completion has two special error classes */
#define MPC_ERR_IN_STATUS 17 /* Look in status for error value */
#define MPC_ERR_PENDING 18 /* Pending request */


#define MPC_NOT_IMPLEMENTED 49
#define MPC_ERR_LASTCODE 50

#define MPC_STATUS_IGNORE NULL
#define MPC_ANY_TAG -1
#define MPC_ANY_SOURCE -1
#define MPC_PROC_NULL -2
#define MPC_COMM_NULL -1
#define MPC_MAX_PROCESSOR_NAME 255

  typedef struct
  {
```

```
    int MPC_SOURCE;
    int MPC_TAG;
    int MPC_ERROR;
    int cancelled;
    mpc_msg_count count;
  } MPC_Status;
#define MPC_STATUS_INIT {MPC_ANY_SOURCE,MPC_ANY_TAG,MPC_SUCCESS,0}

#define MPC_CREATE_INTERN_FUNC(name) extern MPC_Op MPC_##name

    MPC_CREATE_INTERN_FUNC (SUM);
    MPC_CREATE_INTERN_FUNC (MAX);
    MPC_CREATE_INTERN_FUNC (MIN);
    MPC_CREATE_INTERN_FUNC (PROD);
    MPC_CREATE_INTERN_FUNC (LAND);
    MPC_CREATE_INTERN_FUNC (BAND);
    MPC_CREATE_INTERN_FUNC (LOR);
    MPC_CREATE_INTERN_FUNC (BOR);
    MPC_CREATE_INTERN_FUNC (LXOR);
    MPC_CREATE_INTERN_FUNC (BXOR);
    MPC_CREATE_INTERN_FUNC (MINLOC);
    MPC_CREATE_INTERN_FUNC (MAXLOC);

    /*TYPES*/
#define MPC_DATATYPE_NULL ((unsigned long)-1)
#define MPC_CHAR 0
#define MPC_BYTE 1
#define MPC_SHORT 2
#define MPC_INT 3
#define MPC_LONG 4
#define MPC_FLOAT 5
#define MPC_DOUBLE 6
#define MPC_UNSIGNED_CHAR 7
#define MPC_UNSIGNED_SHORT 8
#define MPC_UNSIGNED 9
#define MPC_UNSIGNED_LONG 10
#define MPC_LONG_DOUBLE 11
#define MPC_LONG_LONG_INT 12
#define MPC_PACKED 13
#define MPC_FLOAT_INT 14
/*      struct { float, int } */
#define MPC_LONG_INT 15
/*      struct { long, int } */
#define MPC_DOUBLE_INT 16
/*      struct { double, int } */
#define MPC_SHORT_INT 17
/*      struct { short, int } */
#define MPC_2INT 18
/*      struct { int, int }*/
#define MPC_2FLOAT 19
#define MPC_COMPLEX 20
/*      struct { float, float }*/
#define MPC_2DOUBLE_PRECISION 21
/*      struct { double, double }*/
#define MPC_LOGICAL 22
    /*Initialisation */
```

```
int MPC_Init (int *argc, char ***argv);
int MPC_Init_thread (int *argc, char ***argv, int required, int *provided);
int MPC_Initialized (int *flag);
int MPC_Finalize (void);
int MPC_Abort (MPC_Comm, int);

/*Topologie informations */
int MPC_Comm_rank (MPC_Comm comm, int *rank);
int MPC_Comm_size (MPC_Comm comm, int *size);
int MPC_Node_rank (int *rank);
int MPC_Node_number (int *number);
int MPC_Processor_rank (int *rank);
int MPC_Processor_number (int *number);
int MPC_Process_rank (int *rank);
int MPC_Process_number (int *number);
int MPC_Get_version (int *version, int *subversion);
int MPC_Get_multithreading (char *name, int size);
int MPC_Get_networking (char *name, int size);


/*Collective operations */
int MPC_Barrier (MPC_Comm comm);
int MPC_Bcast (void *buffer, mpc_msg_count count, MPC_Datatype datatype,
               int root, MPC_Comm comm);
int MPC_Allreduce (void *sendbuf, void *recvbuf, mpc_msg_count count,
                   MPC_Datatype datatype, MPC_Op op, MPC_Comm comm);
int MPC_Reduce (void *sendbuf, void *recvbuf, mpc_msg_count count,
                MPC_Datatype datatype, MPC_Op op, int root, MPC_Comm comm);
int MPC_Op_create (MPC_User_function *, int, MPC_Op *);
int MPC_Op_free (MPC_Op *);



/*P-t-P Communications */
int MPC_Isend (void *buf, mpc_msg_count count, MPC_Datatype datatype,
               int dest, int tag, MPC_Comm comm, MPC_Request * request);
int MPC_Ibsend (void *, mpc_msg_count, MPC_Datatype, int, int, MPC_Comm,
                MPC_Request *);
int MPC_Issend (void *, mpc_msg_count, MPC_Datatype, int, int, MPC_Comm,
                MPC_Request *);
int MPC_Irsend (void *, mpc_msg_count, MPC_Datatype, int, int, MPC_Comm,
                MPC_Request *);

int MPC_Irecv (void *buf, mpc_msg_count count, MPC_Datatype datatype,
               int source, int tag, MPC_Comm comm, MPC_Request * request);

int MPC_Wait (MPC_Request * request, MPC_Status * status);
int MPC_Waitall (mpc_msg_count, MPC_Request *, MPC_Status *);
int MPC_Waitsome (mpc_msg_count, MPC_Request *, mpc_msg_count *,
                  mpc_msg_count *, MPC_Status *);
int MPC_Waitany (mpc_msg_count count, MPC_Request array_of_requests[],
                 mpc_msg_count * index, MPC_Status * status);
int MPC_Wait_pending (MPC_Comm comm);
int MPC_Wait_pending_all_comm (void);

int MPC_Test (MPC_Request *, int *, MPC_Status *);
```

```
int MPC_Iprobe (int, int, MPC_Comm, int *, MPC_Status *);
int MPC_Probe (int, int, MPC_Comm, MPC_Status *);
int MPC_Get_count (MPC_Status *, MPC_Datatype, mpc_msg_count *);

int MPC_Send (void *, mpc_msg_count, MPC_Datatype, int, int, MPC_Comm);
int MPC_Bsend (void *, mpc_msg_count, MPC_Datatype, int, int, MPC_Comm);
int MPC_Ssend (void *, mpc_msg_count, MPC_Datatype, int, int, MPC_Comm);
int MPC_Rsend (void *, mpc_msg_count, MPC_Datatype, int, int, MPC_Comm);
int MPC_Recv (void *, mpc_msg_count, MPC_Datatype, int, int, MPC_Comm,
              MPC_Status *);

int MPC_Sendrecv (void *, mpc_msg_count, MPC_Datatype, int, int, void *,
                  mpc_msg_count, MPC_Datatype, int, int, MPC_Comm,
                  MPC_Status *);

/*Status */
int MPC_Test_cancelled (MPC_Status *, int *);

/*Gather */
int MPC_Gather (void *, mpc_msg_count, MPC_Datatype, void *, mpc_msg_count,
                MPC_Datatype, int, MPC_Comm);
int MPC_Gatherv (void *, mpc_msg_count, MPC_Datatype, void *,
                 mpc_msg_count *, mpc_msg_count *, MPC_Datatype, int,
                 MPC_Comm);
int MPC_Allgather (void *, mpc_msg_count, MPC_Datatype, void *,
                   mpc_msg_count, MPC_Datatype, MPC_Comm);
int MPC_Allgatherv (void *, mpc_msg_count, MPC_Datatype, void *,
                    mpc_msg_count *, mpc_msg_count *, MPC_Datatype,
                    MPC_Comm);

/*Scatter */
int MPC_Scatter (void *, mpc_msg_count, MPC_Datatype, void *, mpc_msg_count,
                 MPC_Datatype, int, MPC_Comm);
int MPC_Scatterv (void *, mpc_msg_count *, mpc_msg_count *, MPC_Datatype,
                  void *, mpc_msg_count, MPC_Datatype, int, MPC_Comm);

/*Alltoall */
int MPC_Alltoall (void *, mpc_msg_count, MPC_Datatype, void *,
                  mpc_msg_count, MPC_Datatype, MPC_Comm);
int MPC_Alltoallv (void *, mpc_msg_count *, mpc_msg_count *, MPC_Datatype,
                   void *, mpc_msg_count *, mpc_msg_count *, MPC_Datatype,
                   MPC_Comm);

/*Informations */
int MPC_Get_processor_name (char *, int *);

/*Groups */
int MPC_Comm_group (MPC_Comm, MPC_Group *);
int MPC_Group_free (MPC_Group *);
int MPC_Group_incl (MPC_Group, int, int *, MPC_Group *);
int MPC_Group_difference (MPC_Group, MPC_Group, MPC_Group *);

/*Communicators */
int MPC_Comm_create_list (MPC_Comm, int *list, int nb_elem, MPC_Comm *);
int MPC_Comm_create (MPC_Comm, MPC_Group, MPC_Comm *);
```

```
  int MPC_Comm_free (MPC_Comm *);
  int MPC_Comm_dup (MPC_Comm, MPC_Comm *);
  int MPC_Comm_split (MPC_Comm, int, int, MPC_Comm *);

  /*Error_handler */
#define MPC_MAX_ERROR_STRING 512
  void MPC_Default_error (MPC_Comm * comm, int *error, char *msg, char *file,
                          int line);
  void MPC_Return_error (MPC_Comm * comm, int *error, ...);
#define MPC_ERRHANDLER_NULL MPC_Default_error
#define MPC_ERRORS_RETURN MPC_Return_error
#define MPC_ERRORS_ARE_FATAL MPC_Default_error

  int MPC_Errhandler_create (MPC_Handler_function *, MPC_Errhandler *);
  int MPC_Errhandler_set (MPC_Comm, MPC_Errhandler);
  int MPC_Errhandler_get (MPC_Comm, MPC_Errhandler *);
  int MPC_Errhandler_free (MPC_Errhandler *);
  int MPC_Error_string (int, char *, int *);
  int MPC_Error_class (int, int *);

  /*Timing */
  double MPC_Wtime (void);
  double MPC_Wtick (void);

  /*Types */
  int MPC_Type_size (MPC_Datatype, size_t *);
  int MPC_Sizeof_datatype (MPC_Datatype *, size_t);
  int MPC_Type_free (MPC_Datatype * datatype);

  /*Requests */
  int MPC_Request_free (MPC_Request *);

  /*Scheduling */
  int MPC_Proceed (void);
  int MPC_Checkpoint (void);
  int MPC_Checkpoint_timed (unsigned int sec, MPC_Comm comm);
  int MPC_Migrate (void);
  int MPC_Restart (int rank);
  int MPC_Restarted (int *flag);

  /*Packs */
  int MPC_Open_pack (MPC_Request * request);
  int MPC_Default_pack (mpc_msg_count count,
                        mpc_pack_indexes_t * begins,
                        mpc_pack_indexes_t * ends, MPC_Request * request);
  int MPC_Add_pack (void *buf, mpc_msg_count count,
                    mpc_pack_indexes_t * begins, mpc_pack_indexes_t * ends,
                    MPC_Datatype datatype, MPC_Request * request);
  int MPC_Add_pack_default (void *buf, MPC_Datatype datatype,
                            MPC_Request * request);
  int MPC_Isend_pack (int dest, int tag, MPC_Comm comm,
                      MPC_Request * request);
  int MPC_Irecv_pack (int source, int tag, MPC_Comm comm,
                      MPC_Request * request);
  void *tmp_malloc (size_t size);
```

```
#ifdef __cplusplus

}

#endif

#endif
```

A.2 Exemples de migration de MPI vers MPC

A.2.1 Exemple de programme C

```
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    value=10;

    do {
        value--;
        if (rank == 0)
            MPI_Send( &value, 1, MPI_INT, rank + 1,
                      0, MPI_COMM_WORLD );
        else {
            MPI_Recv( &value, 1, MPI_INT, rank - 1, 0,
                      MPI_COMM_WORLD, &status );
            if (rank < size - 1)
                MPI_Send( &value, 1, MPI_INT, rank + 1,
                          0, MPI_COMM_WORLD );
        }
        printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);

    MPI_Finalize( );
    return 0;
}
```

Version MPI

```
#include <stdio.h>
#include "mpc.h"

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size;
    MPC_Status status;

    MPC_Init( &argc, &argv );
    MPC_Comm_rank( MPC_COMM_WORLD, &rank );
    MPC_Comm_size( MPC_COMM_WORLD, &size );

    value=10;

    do {
        value--;
        if (rank == 0)
            MPC_Send( &value, 1, MPC_INT, rank + 1,
                      0, MPC_COMM_WORLD );
        else {
            MPC_Recv( &value, 1, MPC_INT, rank - 1, 0,
                      MPC_COMM_WORLD, &status );
            if (rank < size - 1)
                MPC_Send( &value, 1, MPC_INT, rank + 1,
                          0, MPC_COMM_WORLD );
        }
        printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);

    MPC_Finalize( );
    return 0;
}
```

Version MPC

A.2.2 Exemple de programme Fortran

```
      program main
C
C
      implicit none
      include 'mpif.h'
      integer ierr,n,tag,status(MPI_STATUS_SIZE)
      integer size,rank,i
      integer errs,nrecv
      logical l(1000)
C
      call mpi_init(ierr)
      call mpi_comm_size(MPI_COMM_WORLD,size,ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,rank,ierr)
C
      n = 100
      do i=1,n
         l(i) = i .lt. n/2
      enddo
      tag = 27
      if (rank .eq. 1) then
         call MPI_Send(l,n,MPI_LOGICAL,0,
     $           tag,MPI_COMM_WORLD,ierr)
      else if (rank .eq. 0) then
         call MPI_Recv(l,n,MPI_LOGICAL,1,
     $           tag,MPI_COMM_WORLD,
     $           status,ierr)
      endif
C
      call mpi_finalize(ierr)
C
      end
```

Version MPI

```
      subroutine mpc_user_main
C
C
      implicit none
      include 'mpcf.h'
      integer ierr,n,tag,status(MPC_STATUS_SIZE)
      integer size,rank,i
      integer errs,nrecv
      logical l(1000)
C
      call mpc_init(ierr)
      call mpc_comm_size(MPC_COMM_WORLD,size,ierr)
      call mpc_comm_rank(MPC_COMM_WORLD,rank,ierr)
C
      n = 100
      do i=1,n
         l(i) = i .lt. n/2
      enddo
      tag = 27
      if (rank .eq. 1) then
         call MPC_Send(l,n,MPC_LOGICAL,0,
     $           tag,MPC_COMM_WORLD,ierr)
      else if (rank .eq. 0) then
         call MPC_Recv(l,n,MPC_LOGICAL,1,
     $           tag,MPC_COMM_WORLD,
     $           status,ierr)
      endif
C
      call mpc_finalize(ierr)
C
      end
```

Version MPC

Annexe B

Présentation détaillée des schémas

B.1 Advection

Nous considérons pour le schéma d'advection un maillage carré de N mailles. Nous avons donc $4\sqrt{N}$ mailles fantômes et $2\left(N^2+\sqrt{N}\right)$ faces. Lors de l'exécution du cas d'advection, nous devons résoudre le système d'équations suivant :

$$\underline{\text{Equation d'advection :}}\begin{cases} \frac{\partial \rho}{\partial t} + v_x \frac{\partial \rho}{\partial x} + v_y \frac{\partial \rho}{\partial y} = 0 \qquad \text{sur } \Omega = [0,1]^2 \\ \rho(0,x,y) = \rho_0(x,y) \\ \text{Conditions aux limites} \end{cases}$$

L'algorithme suivant précise, pour chaque type d'opération, le nombre d'opérations élémentaires mises en jeu. Ainsi nous pouvons bien voir les effets de l'augmentation de la taille du maillage. Cet algorithme est exécuté sur chaque sous-domaine.

Schéma décentré amont «UpWind», explicite en temps :

Initialisation de ρ_0 : N *affectations*
Tant que
- Calcul du pas de temps Δ_{t_k} :
  - $3N$ *Maximums*
  - $2N+1$ *Divisions*
  - $2N$ *Additions*
  - 1 *Minimum*
  - 1 *Réduction du minimum sur* Δ_{t_k}
- Calcul du flux aux faces :
  - $4\left(N^2+\sqrt{N}\right)$ *Multiplications*
  - $2\left(N^2+\sqrt{N}\right)$ *Maximums*
  - $2\left(N^2+\sqrt{N}\right)$ *Minimums*
  - $2\left(N^2+\sqrt{N}\right)$ *Additions*
- Affectation $\rho_{k+1} \leftarrow \rho_k$: N *affectation*

Calcul de ρ_{k+1}
- $2N$ *Additions*
- $2N$ *Divisions*
- $2N$ *Multiplications*
- $2N$ *Soustractions*

Calcul de la surcharge pour les mailles de la bande (concerne O(largeur bande $* N$) mailles)
- 1000 *Additions par maille*
- 500 *Divisions par maille*
- 500 *Multiplications par maille*
- 500 *Opérations Racine carrée par maille*

Mise à jour des mailles fantômes :
- 2 *Envois de* $\sqrt{N}$ *réels*
- 2 *Réceptions de* $\sqrt{N}$ *réels*

Affectation $\rho_k \leftarrow \rho_{k+1}$: N *affectation*

$t_{k+1} \leftarrow t_k + \Delta_{t_k}$

Fin Tant que

B.2 Conduction

Nous considérons pour le schéma de conduction un maillage carré de N mailles. Nous avons donc $4\sqrt{N}$ mailles fantômes et $2\left(N^2+\sqrt{N}\right)$ faces.Lors de l'exécution du cas de conduction, nous devons résoudre le système d'équations suivant :

$$\underline{\text{Equation de la chaleur :}} \begin{cases} \frac{\partial T}{\partial t} - div(K \bigtriangledown T) = f \qquad (t,x,y) \in [0,+\infty[\times[0,1]^2 \\ T(0,x,y) = T_0(x,y) \\ \text{Conditions aux limites} \end{cases}$$

L'algorithme suivant précise, pour chaque type d'opération, le nombre d'opérations élémentaires mises en jeu. Ainsi nous pouvons bien voir les effets de l'augmentation de la taille du maillage. Cet algorithme est exécuté sur chaque sous-domaine.

Schéma implicite en temps avec résolution par gradient conjugué préconditionné par la diagonale :

Initialisation de T_0 : N *affectations*

Tant que

Calcul du pas de temps Δ_{t_k}
- 3 *Multiplications*
- 1 *Minimum*
- 1 *Réduction du minimum sur* Δ_{t_k}

Construction Matrice et Second membre :
- $16\left(N^2+\sqrt{N}\right)$ *Multiplications*
- $16\left(N^2+\sqrt{N}\right)$ *Additions*
- N *Soustractions*
- N *Multadds*

Mise à jour des mailles fantômes

Résolution par gradient conjugué :
Initialisation :
1 *Produit Matrice Vecteur*
1 *Préconditionnement*
1 *Norme L2*
Par itération du gradient conjugué (au maximum N itérations) :
1 *Produit Matrice Vecteur*
3 *Produits Scalaire*
1 *Préconditionnement*
1 *Norme L2*
$2N$ *Multadds*
N *Multsubs*
3 *Divisions*
Mise à jour des mailles fantômes
Affectation $T_k \leftarrow T_{k+1}$: N *affectation*
$t_{k+1} \leftarrow t_k + \Delta_{t_k}$
Fin Tant que

Avec :
- Mise à jour des mailles fantômes :
  2 *Envois de* $\sqrt{N}$ *réels*
  2 *Réceptions de* $\sqrt{N}$ *réels*
- Produit martice vecteur :
  $O(4N)$ *Multadds*
  Mise à jour des mailles fantômes
- Produit Scalaire :
  N *Multadds*
  1 *Réduction somme sur* 1 *réel*
- Norme L2 :
  1 *Produit Scalaire*
  1 *Opération Racine carrée*
- Préconditionnement :
  N *Divisions*

Contribution à l'élaboration d'environnements de programmation dédiés au calcul scientifique hautes performances

Résumé : Dans le cadre du calcul scientifique intensif, la quête des hautes performances se heurte actuellement à la complexité croissante des architectures des machines parallèles. Ces dernières exhibent en particulier une hiérarchie importante des unités de calcul et des mémoires, ce qui complique énormément la conception des applications parallèles.

Cette thèse propose un support d'exécution permettant de programmer efficacement les architectures de type grappes de machines multiprocesseurs, en proposant un modèle de programmation centré sur les opérations collectives de communication et de synchronisation et sur l'équilibrage de charge. L'interface de programmation, nommée MPC, fournit des paradigmes de haut niveau qui sont implémentés de manière optimisée en fonction de l'architecture sous-jacente. L'environnement est opérationnel sur la plate-forme de calcul du CEA/ DAM (TERANOVA) et les évaluations valident la pertinence de l'approche choisie.

Mots-clés : Multithreading, calcul scientifique, hautes performances, NUMA.

Contributing to the design of Runtime Systems dedicated to High Performance Computing

Abstract : In the field of intensive scientific computing, the quest for performance has to face the increasing complexity of parallel architectures. Nowadays, these machines exhibit a deep memory hierarchy which complicates the design of efficient parallel applications.

This thesis proposes a programming environment allowing to design efficient parallel programs on top of clusters of multiprocessors. It features a programming model centered around collective communications and synchronizations, and provides load balancing facilities. The programming interface, named MPC, provides high level paradigms which are optimized according to the underlying architecture. The environment is fully functional and used within the CEA/DAM (TERANOVA) computing center. The evaluations presented in this document confirm the relevance of our approach.

Keywords : Multithreading, scientific computing, high performance computing, NUMA.