Exploitation efficace des architectures parallèles de type grappes de NUMA à l'aide de modèles hybrides de programmation

Jérôme Clet-Ortega

N° d'ordre: 4514

THÈSE

présentée

devant l'Université de Bordeaux 1

École doctorale de Mathématiques et Informatique

pour obtenir

le grade de : Docteur de l'Université de Bordeaux 1
Mention Informatique

par

Jérôme CLET-ORTEGA

Équipe d'accueil : Equipe projet Runtime - LaBRI/INRIA
Composante universitaire : LaBRI/Bordeaux 1

Titre de la thèse :

Exploitation efficace des architectures parallèles de type grappes de NUMA à l'aide de modèles hybrides de programmation

Soutenue le 18 avril 2012 devant la commission d'examen

M. :	Denis	Barthou	Professeur des Universités	Président
MM. :	Yves	Denneulin	Professeur des Universités	Rapporteurs
	Christian	Pérez	Directeur de recherche INRIA	
MM. :	David	Goudin	Directeur au CEA CESTA	Examinateurs
M. :	Raymond	Namyst	Professeur des Universités	Directeurs de thèse
	Guillaume	Mercier	Maître de Conférences	

Remerciements

Je tiens tout d'abord à remercier mes deux directeurs de thèse, Raymond Namyst et Guillaume Mercier qui m'ont permis de découvrir le monde de la recherche et n'ont cessé de me soutenir au cours de ces dernières années. Ce document a pu voir le jour grâce à la confiance qu'ils m'ont accordée. Ils ont su me transmettre leur savoir et leur motivation. Je ne cesserais pour ainsi dire jamais d'envier le talent pédagogique de Raymond qui a su éveiller mon intérêt pour le parallèlisme dès mon premier cours de PAP et dispose d'au moins une dizaine de métaphores pour expliquer un concept particulier. Citons ainsi le fameux robot et ses tiroirs représentant respectivement le micro-processeur et la mémoire.

Je remercie les rapporteurs de cette thèse, Yves Denneulin et Christian Pérez, pour l'intérêt qu'ils ont porté à mon manuscrit et pour les corrections avisées qu'ils ont su me proposer. Merci à eux ainsi qu'aux autres membres du jury, Denis Barthou et David Goudin, qui ont accepté de juger ces travaux.

J'adresse également mes remerciements à tous les membres, actuels et anciens, de l'équipe Runtime qui m'ont accueilli et soutenu tout au long de cette thèse. La bonne humeur qu'ils apportent chaque jour est une excellente source de motivation et permet de travailler dans de très bonnes conditions. Merci au bienveillant Pierre-André qui sous son air nonchalant (et son sourire narquois) dissimule un être attentionné. Merci à Marie-Christine pour sa gentillesse qui force l'admiration. Merci à Brice disponible à tout instant pour répondre à mes innombrables interrogations plus ou moins scientifiques. Merci à Olivier qui a encadré mon DEA et a toujours été de bon conseil. Merci à Nathalie, toujours joyeuse, pour m'avoir expliqué que le jus d'orange ne constitue pas le meilleur carburant pour un mac. Merci à Alexandre pour ses conseils avisés. Merci à Samuel pour les sessions «à la recherche du bug perdu »dans Marcel. Merci à Ludovic, grand défenseur du GNU devant l'éternel, pour son grand sourire. Merci à Emmanuel pour les discussions sur le placement. Merci à Yannick pour son travail dans Marcel. Merci à Ludovic pour son irrévérence face à son chef. Merci à Bertrand et sa compréhension de Bertha. Merci à Sylvain pour avoir limité ses questions sur l'avancée de ma rédaction. Merci à tous les jeunes qui prennent la relève dans l'équipe : François, Andra, Nicolas, Cyril, etc. Je remercie également les anciens doctorants qui ont fait partie de l'équipe et avec lesquels j'ai partagé de très bons moments : Elisabeth qui nous a expliqué le parallèlisme dans notre jeunesse, François qui a baigné lui aussi dans un monde hybride, François (pas le même) que je vais rejoindre bientôt, Stéphanie et son gâteau à la citrouille et Cédric qui a redonné vie à de nombreuses œuvres oubliées de la chanson française.

Un grand merci également aux résidents du A29 bis et aux membres du LaBRI. Merci notamment Abdou d'être passé aussi souvent dans mon bureau, nos discussions m'ont toujours fait avancer. Merci aux membres de l'agos-foot pour les parties endiablées

autour du ballon rond.

Je tiens à remercier tout particulièrement François, qui a partagé avec moi les mêmes bancs de la faculté et par la suite le même bureau durant ces années de thèse. Son amitié sans faille m'est chère et n'est pas étrangère à la réussite de mon doctorat. Je souhaite à chacun de trouver un ami aussi sincère.

Je remercie les membres de ma famille de leur soutien tout au long de la thèse et, pour ceux qui ont pu venir, de leur présence à ma soutenance. Merci à Michèle, Marie-Claude et Jacques pour la préparation de cet excellent pot de thèse.

Merci également à mes amis, Vivien, Gaëlle, Stéphane, Leslie, François, Cathy, Laurent, Rémi pour avoir eu le courage de m'écouter jusqu'au bout.

Enfin, je souhaite remercier Mathilde, mon épouse, pour son indéfectible soutien et son enthousiasme communicatif vis-à-vis de mes travaux. Elle m'épaule chaque jour et sans elle rien n'aurait été possible.

Résumé

Présente dans presque tous les domaines de recherche et de développement, la simulation numérique a offert de nouvelles perspectives pour les acteurs de ces domaines et les enjeux sont tout autant économiques que scientifiques. Les performances des simulations dépendant directement de la puissance offerte par le matériel, ce dernier fait l'objet de nombreuses évolutions avec lesquelles les développeurs d'applications doivent composer. Les systèmes de calcul actuels sont généralement des grappes de machines composés de nombreux processeurs à l'architecture fortement hiérarchique. L'exploitation efficace de ces systèmes constitue le défi majeur des implémentations de modèles de programmation tels MPI ou OpenMP. Une pratique courante consiste à mélanger ces deux modèles pour bénéficier des avantages de chacun. Cependant ces modèles n'ont pas été pensés pour fonctionner conjointement ce qui entraîne des problématiques qui affectent directement les performances.

Nous proposons dans cette thèse un environnement destiné à assister le développeur dans la programmation d'application de type hybride. Trois axes majeurs structurent notre proposition. Dans un premier temps, nous analysons la hiérarchie du système de calcul cible pour connaître l'agencement des cœurs et des unités mémoire (cache, nœud NUMA, etc). En s'appuyant sur ces informations nous contrôlons le parallélisme de chaque paradigme en dimensionnant les ressources d'exécution (processus et threads) selon l'architecture sous-jacente et en les placant judicieusement. Plutôt qu'une approche hybride classique, créant un processus MPI multithreadé par nœud, nous évaluons de façon automatique des solutions alternatives, avec plusieurs processus multithreadés par nœud, mieux adaptées aux machines de calcul modernes. Enfin, nous nous appuyons sur les traces d'exécution des programmes afin d'aider le développeur à comprendre les performances obtenues au travers du schéma de communication et du diagramme d'exécution.

Les gains obtenus sur des tests synthétiques et sur des applications scientifiques montrent que le contrôle de la granularité et du placement selon la hiérarchie de l'architecture du système de calcul permet d'optimiser l'exécution dans le cadre d'un mélange de deux modèles de programmation.

Table des matières

Introduction

Le calcul intensif numérique occupe une place significative dans le quotidien des acteurs du monde scientifique et industriel (chercheurs, ingénieurs, etc). De nombreux phénomènes qui vont de l'infiniment petit, comme le comportement des protéines intervenant dans les processus biologiques, jusqu'à l'infiniment grand, tel l'évolution de l'Univers, excèdent le domaine de l'observable. Seule la simulation numérique offre les moyens d'étudier à de telles dimensions, de manipuler le temps à grande échelle ou de gérer un grand nombre de contraintes. La puissance de calcul que requiert l'exécution des simulations numériques est conséquente et augmente de façon continue. Les modélisations doivent être toujours plus fines et calculées dans un temps raisonnable. Pour répondre à ces besoins croissants de puissance, les centres de calcul se sont équipés de machines performantes renouvelées régulièrement pour se maintenir dans la compétition.

Ces dernières décennies ont vu l'émergence de nouvelles catégories de machines capables de fournir une puissance toujours plus importante. Dans les années 70, plusieurs constructeurs (SGI, Cray, etc.) ont développé des systèmes, capables d'exécuter plusieurs flots de calcul en parallèle. Ces super-calculateurs ont dominé le marché jusqu'à la fin des années 80. Leur omniprésence s'est peu à peu estompée depuis le début des années 90 pour laisser la place à des systèmes moins onéreux et beaucoup plus extensibles : les grappes de calcul. Il s'agit de l'interconnexion par un réseau dédié d'un ensemble d'ordinateurs standards (appelés également *nœuds* dans ce contexte). Leur succès est principalement dû au développement de nouvelles technologies réseau désignées comme étant rapides grâce à un débit élevé et une latence très faible. L'excellent rapport performance/coût qu'offre les grappes de calcul a particulièrement séduit les intervenants du domaine à tel point que la plupart des calculateurs actuels sont des grappes.

Depuis leur apparition, les grappes de calcul ont subi de nombreuses évolutions. Les réseaux d'interconnexion sont devenus plus performants et il est courant de rencontrer plusieurs technologies d'interconnexion à l'intérieur de la même grappe. En outre, les nœuds, également à la base des performances des grappes, sont devenus plus perfectionnés, principalement au niveau des processeurs. Historiquement, la montée en fréquence et la complexification des composants des processeurs constituaient les seuls efforts des constructeurs pour augmenter la puissance de calcul d'une génération à l'autre. Aujourd'hui, les problèmes de dissipation thermique inhérants aux circuits imprimés obligent

les architectes à limiter la fréquence. Pour continuer à fabriquer des processeurs plus efficaces, les constructeurs profitent de l'amélioration continue de la finesse de gravure pour intégrer plusieurs unités de calcul sur une même puce et donner ainsi naissance aux processeurs multicœurs. Ces derniers constituent l'architecture usuelle des nœuds de calcul et peuvent d'ailleurs y être présents en plusieurs exemplaires dans chaque nœud. Le nombre d'unités de calcul s'est multiplié et une hiérarchie se dégage de leur agencement. Certains cœurs vont ainsi partager un ou plusieurs niveaux de cache, l'accès à des bancs mémoire, plusieurs interfaces réseau, etc. De ce fait, de nouvelles contraintes se créent en raison des différentes affinités qui peuvent exister entre certains cœurs. La programmation d'applications parallèles au-dessus de ces architectures est rendue plus complexe. Il ne s'agit plus d'exploiter un ensemble de processeurs symétriques.

Les modèles de programmation parallèles doivent faire face à cette évolution architecturale pour continuer d'exploiter efficacement les grappes de calcul. Le plus largement utilisé sur ce type d'architectures, le standard MPI, permet d'écrire des applications parallèles performantes et portables. Bien que de nombreuses recherches aient été réalisées pour améliorer l'efficacité des implémentations dans un environnement multicœur, l'écart entre les performances théoriques et celles réellement obtenues demeure important. L'interface MPI reste limitée en ce sens que le programmeur ne peut pas aisément exprimer les affinités existantes entre certains flots d'exécution de manière à les mettre en correspondance avec la hiérarchie sous-jacente. Afin d'y remédier, de plus en plus d'applications ont recourt à l'adjonction d'un second modèle de programmation mieux adapté aux architectures multicœurs. L'application est alors caractérisée par deux niveaux de parallélisme en analogie avec l'architecture composite des grappes de calcul. Dans la majorité des cas, c'est le langage OpenMP qui est employé à l'intérieur des nœuds de calcul, notamment en raison de sa simplicité de mise en place et de son efficacité. Ce mélange de modèles de programmation pose de nouveaux problèmes pour les implémentations dont les modèles n'ont pas été conçus pour fonctionner ensemble.

Cadre de la thèse et contribution

En raison de la multiplication du nombre de cœurs par nœud, la programmation hybride est devenue une pratique courante dans le domaine du calcul hautes-performances. La création d'une section parallèle OpenMP permet de définir une relation d'affinité entre les traitements parallèle et plus globalement de structurer l'exécution du code pour tirer profit des architectures multicœurs modernes. Etant donné que ces architectures ne sont pas planes, le mélange de modèles peut s'avérer plus fin que la mise en place d'un seul processus MPI multithreadé par nœud. Selon l'organisation interne de l'architecture ciblée, il peut s'avérer plus intéressant de générer plusieurs processus multithreadés par nœud, en fonction des ressources mémoires partagées entre les cœurs.

Les travaux présentés dans cette thèse se situent dans ce contexte. Nous proposons d'assister le développeur d'application parallèle dans l'utilisation d'un modèle de programmation hybride. À cette fin, nous fournissons un environnement constitué d'ou-

tils et de techniques de programmation pour exploiter efficacement les capacités des architectures modernes. Tout d'abord, nous utilisons un outil pour récupérer une représentation générique de la topologie matérielle du système ciblé. Il s'agit de connaître en détail les liens existants entre les unités de calcul (niveau de mémoire cache, banc mémoire, etc.) pour en ressortir une structure arborescente claire. Cette représentation permet alors de déterminer les différentes combinaisons de processus et de threads qui accordent les affinités des traitements logiciels avec celles des unités de calcul du matériel. Les informations topologiques servent également à placer les traitements parallèles en relation sur les unités de calcul adjacentes. L'objectif est de contrôler finement les deux niveaux de parallélisme pour optimiser le temps d'exécution global des applications. Enfin, nous nous appuyons sur des outils de traces d'exécution pour modéliser le comportement applicatif pour comprendre les performances, identifier les différentes sections de code critiques et visualiser les interactions entre les deux modèles.

Organisation du document

Le chapitre 1 retrace l'évolution des architectures matérielles des systèmes parallèles. La structure interne complexe des nœuds composant les grappes de calcul ainsi que les technologies réseau qui les relient y sont décrites. Le chapitre 2 présente une classification des modèles de programmation permettant d'exploiter ces architectures parallèles. Les limites de ces modèles sont abordées ainsi que le moyen d'y remédier en mélangeant deux types de modèles de programmation. Dans le chapitre 3, nous présentons la contribution de cette thèse en exposant les trois grands principes : l'analyse de l'architecture, le contrôle du parallélisme et la visualisation des traces d'exécution. Le chapitre 4 permet de valider notre proposition au travers d'expériences sur des micro-benchmarks et applications parallèles. Pour finir, nous concluons sur les travaux réalisés et discutons des perspectives qui s'en dégagent.

Chapitre 1

Les plates-formes de calcul parallèle

Depuis l'introduction des transistors en 1955 dans la conception des machines de calcul, le matériel informatique a énormément progressé, pour répondre aux besoins toujours croissants des scientifiques. Les applications qu'ils développent et utilisent nécessitent toujours plus de puissance, de mémoire. Les fabricants de processeurs, et plus généralement d'ordinateurs, font continuellement preuve d'ingéniosité afin de satisfaire au mieux ces besoins. La parallélisation est le fruit de cette recherche constante de puissance. C'est une technique qui consiste à mettre en commun plusieurs ressources de calculs pour réaliser de multiples traitements de données, de façon simultanée. Au cours du temps, la parallélisation au sein du matériel informatique a pris de nombreux aspects, avec plus ou moins de succès selon les cas, pour en arriver aux machines de calcul actuelles.

Ce chapitre s'efforce de présenter le paysage matériel du calcul parallèle qui constitue le domaine d'action que nous nous sommes fixés dans ces travaux. Nous y énumérons les grandes évolutions qui ont été apportées au niveau de la structure des systèmes parallèles, de l'architecture des processeurs qui les composent, ainsi que des réseaux d'interconnexion qui les relient.

1.1 Evolution des systèmes parallèles

La dynamique qui régit l'évolution du matériel informatique est telle que la puissance de calcul des ordinateurs des années 50, capables de réaliser plusieurs milliers d'opérations à la seconde, apparaît bien faible face à celle des téléphones portables de dernière génération qui effectuent des milliers de millions d'opérations flottantes à la seconde. Ce constat est d'autant plus flagrant lorsque l'on considère les machines de calcul de grande dimension dédiées aux scientifiques, qui sont devenues de véritables symboles de puissance pour les laboratoires et organismes du monde entier. Nous présentons dans cette section les évolutions structurelles de ces systèmes parallèles.

1.1.1 L'ère des supercalculateurs

La chronologie de l'évolution du matériel de calcul hautes-performances, commence par la domination des super-calculateurs depuis le milieu des années 60 jusqu'aux années 90. Durant cette période, l'architecture de ces machines a beaucoup évoluée. L'une des premières, le CDC 6600 [Tho80] était dotée d'un processeur central assisté par des processeurs périphériques destinés à réaliser les opérations d'entrées/sorties en parallèle. Puis, au cours des années 70, les constructeurs ont intégré des processeurs vectoriels capables d'appliquer une même instruction à un ensemble de données, favorisant ainsi les calculs s'appliquant à des tableaux de nombres, comme la manipulation d'images. Précisons d'ailleurs que cette vectorisation est un mécanisme intégré dans la plupart des processeurs actuels par le truchement de jeux d'instructions spécifiques : SSE apparu en 1999 sur le processeur Intel Pentium III [RPK00a], Altivec mis en place pour le processeur PowerPC par Motorola [Ful99]. Afin d'accroître les performances des supercalculateurs, plusieurs de ces processeurs vectoriels sont associés pour fonctionner en parallèle et constituent ainsi la génération suivante.

À l'approche des années 90, ces unités de calcul vectorielles ont laissé la place à des processeurs scalaires plus « simples », donc moins coûteux, mais plus nombreux, franchissant ainsi un palier supplémentaire dans la course aux performances. Ce type d'architectures est souvent désigné sous le terme de système massivement parallèle, le nombre de processeurs pouvant atteindre plusieurs centaines dans un premier temps, puis plus récemment plusieurs centaines de milliers.

Le principal objectif des concepteurs de ces supercalculateurs consiste à développer une machine dotée d'une puissance de calcul toujours plus importante que la génération de machines précédente afin de satisfaire les besoins croissants des applications scientifiques. Cependant, le coût relativement important de ces systèmes ne permet qu'à des organismes disposant de moyens suffisant de s'en procurer. En outre, le seul moyen de rehausser la puissance de calcul d'un supercalculateur est, du fait de son aspect propriétaire, d'acquérir un nouveau système plus performant. Ce manque d'évolutivité et le prix de ces solutions ont tous deux mis à mal l'emprise qu'elles avaient sur le monde du calcul parallèle. Bien que moins présents de nos jours, ces systèmes tendent à réapparaître : ils représentent 17% des machines de calcul les plus puissantes, repertoriées par le Top500 [Top]. Citons ainsi le calculateur Blue Gene/P [ABB+08] disposant de centaines de milliers de cœurs et atteignant le PetaFlop/s [1], et dont la prochaine génération est d'ores et déjà en cours de conception. De surcroît, en première place du recensement de Novembre 2011, nous retrouvons le K Computer qui affiche une performance record de près de 10 Petaflop/s.

1. Flop/s est un acronyme désignant le nombre d'opérations à virgule flottante par seconde, et représente la vitesse communément admise d'un système informatique. Ici nous parlons de 10^{15} opérations par seconde.

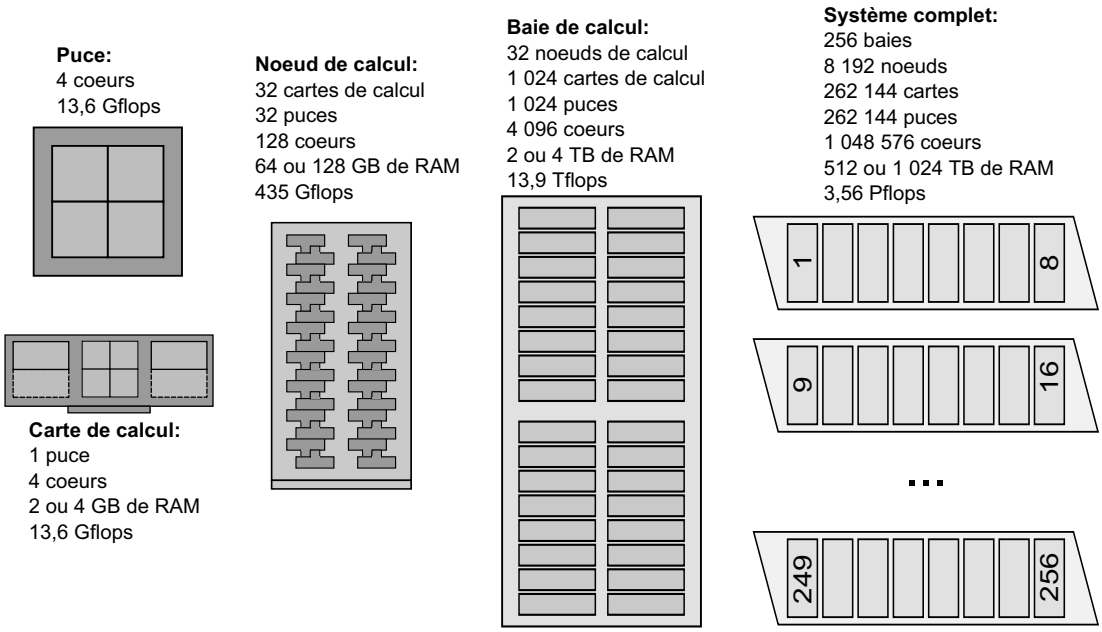


FIGURE 1.1 – Architecture du système Blue Gene/P

1.1.2 Le succès des grappes de calcul

1.1.2.1 Une puissance de calcul peu onéreuse et extensible

En marge des progrès réalisés dans les supercalculateurs, les micros-ordinateurs ont eux aussi connu de profondes mutations, se sont standardisés, et ont peu à peu conquis les entreprises, les laboratoires, les universités et les foyers. L'ensemble de ces stations de travail s'est avéré être une alternative intéressante pour les centres ne disposant pas de moyens financiers suffisants. En effet, en réunissant la puissance de calcul de chacune des ces machines grâce à un réseau d'interconnexion, il est possible d'exploiter les temps de cycles inutilisés. Le projet CONDOR [LLM88] en est une parfaite illustration. Toutefois cette démarche est plus un pis-aller qu'un réel moyen de calcul hautes-performances, les performances étant limitées à la fois par la disponibilité des machines et par les capacités du réseau d'interconnexion. Généralement il s'agissait du réseau Ethernet au-dessus duquel on utilisait des protocoles de communication gérant le transfert de données (TCP/IP). Ceux-ci étaient conçus pour fonctionner sur des lignes peu fiables et de longue distance : les constructeurs se sont alors concentrés sur le contrôle de flux et la reprise sur erreur.

Il a fallu attendre les années 90 pour voir apparaître des technologies réseau plus efficaces et plus fiables : le débit est ainsi passé de quelques Mégabits/s à plusieurs dizaines de Gigabits/s et la latence a atteint la microseconde. Certains ont vu une occasion d'améliorer les performances des réseaux de stations de travail comme en témoignent les

projets NOW de Berkeley [ACP95] et Beowulf [SSB+95] développé à l'origine par le CESDIS(*Center of Excellence in Space Data and Information Sciences*) en lien avec la NASA. Ces progrès étaient de plus en plus orientés vers des systèmes dédiés uniquement au calcul et ont engendré la naissance d'une nouvelle classe de systèmes : les grappes de calcul. À l'image des réseaux de stations de travail, il s'agit de réunir la puissance d'ordinateurs standards avec un réseau hautes-performances. Ainsi, le matériel est entièrement dédié, placé physiquement dans la même pièce (afin de minimiser les délais de transmission au maximum).

C'est une solution véritablement avantageuse en comparaison des supercalculateurs en raison du rapport performances/prix et des capacités d'extension avec des composants standards. L'ajout de nouveaux noeuds de calculs permet d'augmenter la puissance de calcul globale. La principale difficulté est de réussir à exploiter efficacement un tel système, là où les supercalculateurs disposent d'outils spécifiquement optimisés pour l'architecture. Malgré cela, les grappes se sont rapidement imposées dans le monde du calcul parallèle : au dernier recensement (c'est-à-dire en Novembre 2011), elles constituent plus de 80% du parc des machines les plus puissantes du monde (figure 1.2).

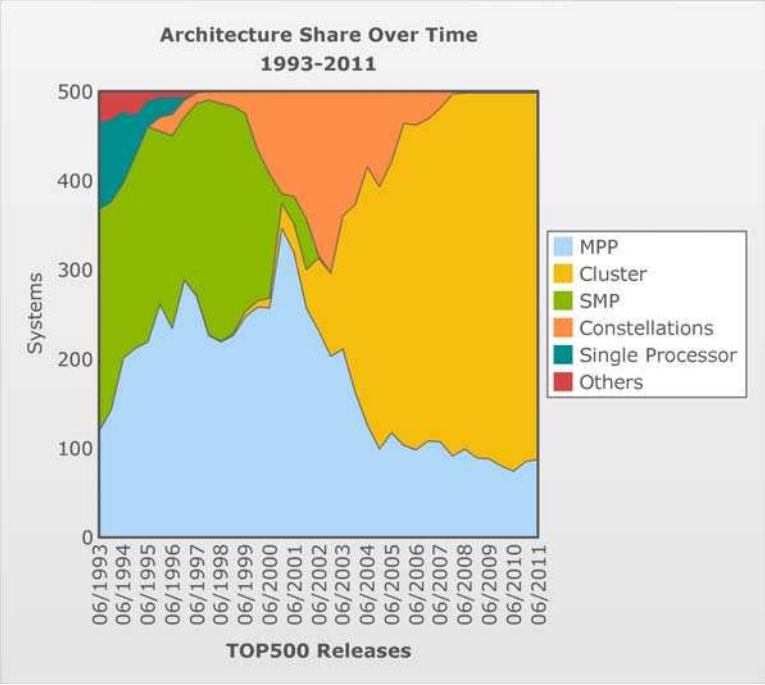


Figure 1.2 – Evolution des architectures du Top500

1.1.2.2 Les grappes de grappes

L'interconnexion de plusieurs grappes de calcul est apparu comme une conséquence naturelle du succès des grappes de calcul. L'idée s'inspire directement de cet assemblage d'ordinateurs standard pour créer une sorte de *méta-grappe* où les noeuds sont des grappes, reliées entre elles par un réseau à haut-débit. Ces grappes peuvent faire partie d'un même laboratoire de recherche ou bien de différents laboratoires à l'échelle nationale ou internationale.

De nouvelles problématiques se dégagent du fait de la forte hétérogénéité des composants, des systèmes d'exploitation et des réseaux d'interconnexion. Certaines grappes ont des nœuds reliés par un seul réseau hautes-performances, d'autres, et c'est de plus en plus courant, contiennent des noeuds disposant de plusieurs cartes réseau de technologies différentes. Par ailleurs, le graphe que forme l'union des grappes n'est pas forcément complet : certaines grappes ne sont pas directement liées entre elles. De plus, la nature du réseau reliant les grappes déterminera l'impact des communications inter grappes et par extension celui sur les performances applicatives. La figure 1.3 schématise une grappe de quatre grappes, chacune ayant un lien vers deux grappes voisines. Les outils logiciels permettant d'exploiter ce genre d'architectures doivent atteindre des objectifs de performances tout en restant portables. De même, la tolérance aux pannes et la gestion dynamique des processus deviennent beaucoup plus complexes.

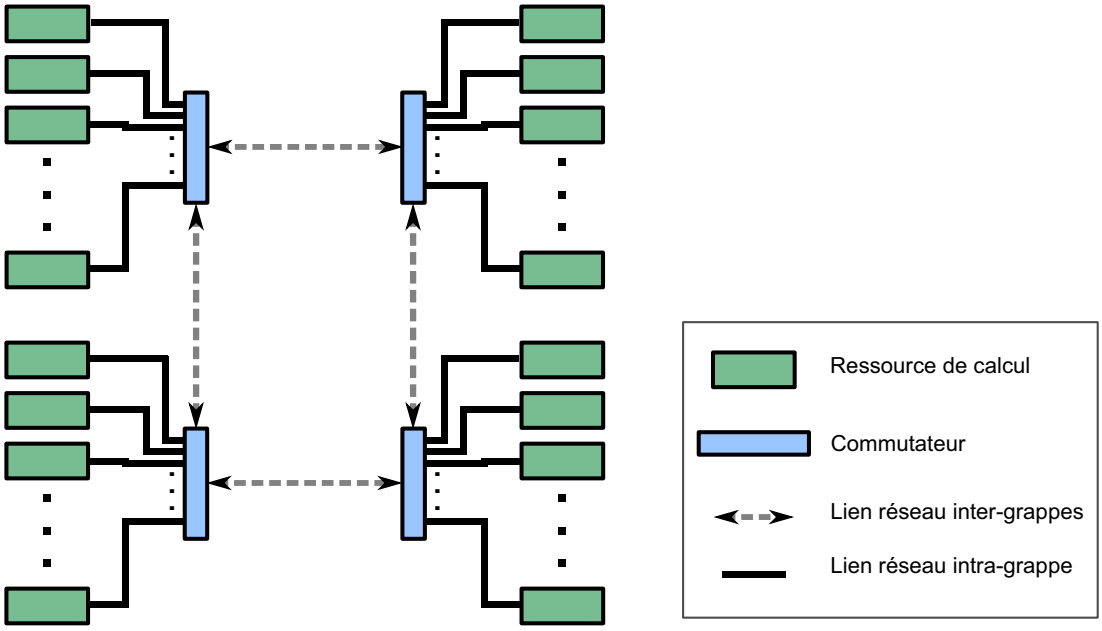


FIGURE 1.3 – Une grappe de grappes

1.1.2.3 Les grilles de calcul

Suite à la propagation des grappes de calcul, il était naturel de penser à franchir une étape supplémentaire dans la mutualisation des moyens informatiques dont disposent les laboratoires et universités. Cette approche consiste à relier plusieurs ressources de calcul, potentiellement distantes de plusieurs centaines de kilomètres, afin de disposer d'un grand système de calcul réparti. Les éléments reliés sont relativement variés : machines séquentielles, supercalculateurs, grappes de calcul ou baies de stockage. La structure d'une grille est par essence dynamique : de nouvelles ressources peuvent être ajoutées à tout moment, certaines peuvent être retirées temporairement (à cause d'une panne par exemple) ou définitivement. Les outils de gestion doivent fonctionner efficacement sur cet environnement hautement hétérogène, de grande dimension et pouvant subir des modifications au cours du temps.

Parmi les infrastructures présentes, citons le projet GRID'5000 [CCD+05, gri] reliant les grappes de calculs de nombreux laboratoires de recherche en France. Lancée en 2003, c'est une plate-forme expérimentale pour de nombreuses applications nécessitant une grande puissance de calcul et un vaste espace de stockage. Ainsi les sites de Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia et Toulouse sont interconnectés grâce au réseau RENATER [ren]. Celui -ci offre un débit de 10 GBit/s et une latence allant de 3 ms à 15 ms selon la distance séparant les noeuds de calcul. Chacun des sites offre des grappes de calcul basés sur des réseaux rapides à faible latence (aux alentours de quelques microsecondes). Actuellement GRID'5000 propose près de 3000 processeurs aux utilisateurs, ce qui correspond environ à 8500 coeurs. Et ces chiffres vont bientôt croître avec l'adjonction de nouveau sites à Reims, Luxembourg et Porto Alegre au Brésil.

1.1.3 Les réseaux d'interconnexion

1.1.3.1 Ethernet et la pile TCP/IP

Au cours des années 70, la nécessité d'échanger des informations entre différents ordinateurs géographiquement éloignés a permis la création et l'émergence de multiples technologies et de protocoles de communication. Parmi ces technologies, ETHERNET a su se distinguer de par sa simplicité d'interconnexion, son indépendance vis-à-vis des constructeurs et sa facilité d'évolution. Associée aux protocoles de communication TCP/IP, elle a rapidement conquis les structures informatiques mondiales pour finalement servir de base au vaste réseau qu'est Internet. L'objectif était de faire communiquer des noeuds distants de façon transparente, abstraction faite de la position des intervenants. Cette contrainte a orienté les efforts de développement vers la reprise sur erreur, la tolérance aux pannes, le routage dynamique des paquets de données et le contrôle de flux.

Il en résulte une pile logicielle relativement importante pour garantir ces nombreux services, et chaque couche de cette pile logicielle va adjoindre un bloc d'informations

aux paquets la traversant (codes correcteurs d'erreur, informations de routage, etc.). Par ailleurs, la transmission d'un message génère une copie mémoire des données utilisateurs en espace noyau en émission et une autre copie mémoire d'un tampon de l'espace noyau vers l'espace utilisateur en réception. Ces copies nécessitent l'intervention du processeur et ralentissent d'autant les performances globales du système. Ce mode de communication est désigné sous le terme PIO (*Programmed Input/Output*). La figure 1.4 représente le transfert d'un bloc de données utilisant le mode PIO : chaque message envoyé est découpé en une série de blocs d'octets qui sont lus par le processus avant d'être transférés sur la carte réseau. Le processeur est alors occupé tout le long du transfert. Relativement efficace pour les transferts de messages de petite taille, lorsque les messages sont de plus grande taille, l'impact est beaucoup plus conséquent. Afin d'y remédier, un autre mode de communication peut être employé pour soulager le processeur : le mode DMA (*Direct Memory Access*). C'est toujours lui qui initie la communication, cependant le transfert en lui-même est assuré par le contrôleur DMA de la carte réseau comme la figure 1.5 le décrit. Le processeur devient alors disponible pour réaliser d'autres opérations. Au final, la solution optimale pour minimiser les coûts des transmissions est un intermédiaire utilisant l'un ou l'autre de ces modes selon la quantité de données à envoyer comme en témoigne la courbe de la figure 1.6. D'autres mécanismes ont été mis en place afin de réduire les coûts de transferts. Certaines tâches, comme la vérification de la validité des données, ont été déléguées à la carte Ethernet. De plus, la carte a la possibilité de jouer le rôle d'accumulateur, afin de transférer un jeu de données en une fois plutôt que d'interrompre à plusieurs reprises le système d'exploitation.

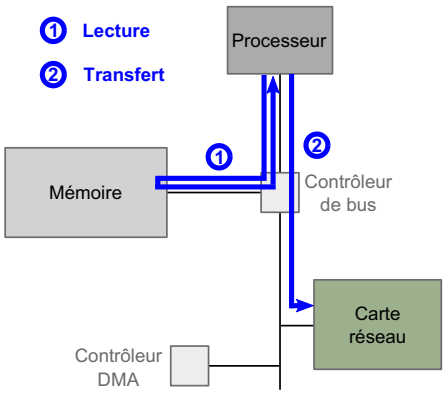


FIGURE 1.4 – Transfert en mode PIO

En dépit des problèmes de performances, les réseaux de type ETHERNET sont toujours présents dans les grappes de calcul, à un tel point que près de la moitié des machines du TOP500 en sont équipées. Des travaux ont été réalisés aussi bien au niveau du matériel qu'au niveau de la couche logicielle afin d'en améliorer les performances. D'une part, la technologie elle-même a connu plusieurs évolutions : à l'origine le débit était de 10 Mbit/s, pour ensuite atteindre 100 Mbit/s avec FAST ETHERNET, puis 1000 Mbit/s avec GIGABIT ETHERNET et enfin 10 Gbit/s grâce à la dernière génération ETHERNET

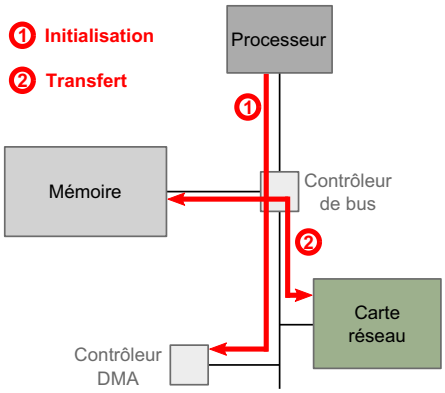


FIGURE 1.5 – Transfert en mode DMA

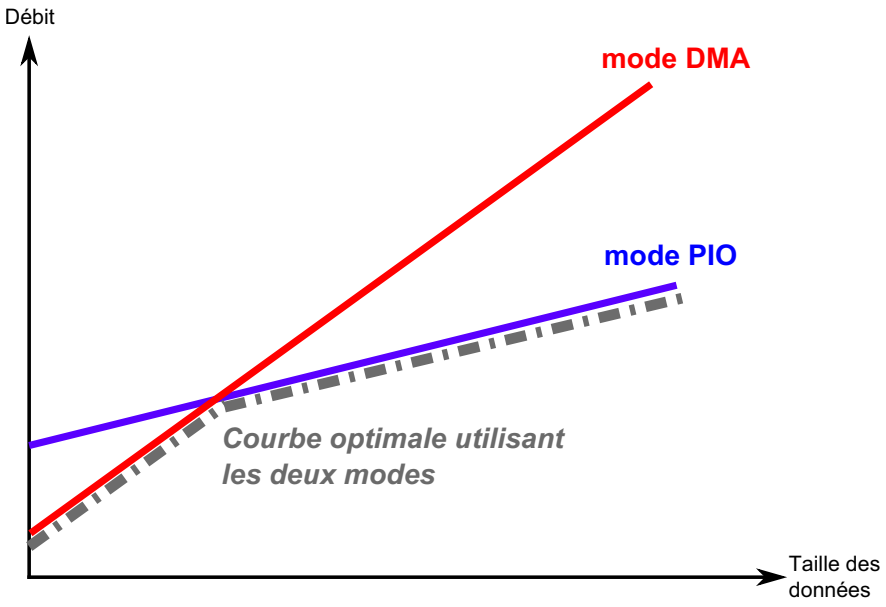


FIGURE 1.6 – Allure de la courbe de transfert de données selon le mode

10-GIGABIT. D'autre part, parallèlement à ces progrès, des mécanismes ont été développés pour optimiser les communications en modifiant la programmation embarquée sur la carte réseau. Pour la plupart, il s'agissait de court-circuiter le système d'exploitation, soit pour éviter des recopies mémoire intermédiaires [SWP01, KRS01, DDW06], soit pour éviter un appel système coûteux et dialoguer avec la carte réseau directement grâce à une projection mémoire en espace utilisateur [PF01]. Toutefois, ces modifications étaient réalisées en collaboration avec un constructeur particulier et donc limitées à un modèle particulier de carte ETHERNET. Dans un environnement où le type de carte ETHERNET peut changer d'une grappe de calcul à une autre c'est un réel problème.

1.1.3.2 Réseaux hautes-performances

Des recherches, fortement inspirées des matériels de communication intégrés dans les supercalculateurs, ont été menées et ont débouché sur des technologies d'interconnexion fiables (c'est-à-dire avec un taux de perte et un taux d'erreur beaucoup plus réduits) et plus compétitives. D'une part, les grappes de calcul sont en générales établies dans une même pièce, la connectique qui relie les cartes réseau entre-elles peut-être beaucoup plus réduite et plus coûteuse que celle des réseaux classiques. Par ailleurs, les constructeurs de telles cartes peuvent se permettre d'y intégrer une programmation (dans le logiciel embarqué) destinée à minimiser la latence et à maximiser le débit. Qui plus est, le nombre de noeuds qui composent une grappe est largement plus restreint que celui de l'Internet, le routage des paquets peut être effectué statiquement : on se dispense ainsi du mécanisme de décision dynamique, ce qui est un gain de temps certain sur le chemin critique. L'ensemble de ces optimisations et le matériel perfectionné ont conduit à une diminution sensible de la latence jusqu'à atteindre la microseconde, ainsi qu'à une amélioration conséquente du débit. Dans les paragraphes suivants, nous présentons les technologies d'interconnexion majeures présentes dans les machines de calcul actuelles.

Myrinet Le réseau d'interconnexion MYRINET [BCF+95] est une technologie réseau développé par la société MYRICOM [Myrb] depuis 1995. Depuis cette époque, de nombreuses générations de cartes ont été produites. La dernière en date, dénommée MYRI-10G, se compose d'un processeur RISC embarqué sur la carte et programmable. Ce processeur,appelé LANAI, est cadencé à 364,6 MHz et dispose d'une mémoire SRAM de 2 Mo. Grâce à un compilateur croisé, le firmware de la carte peut être facilement modifié (en langage C) et transmis dynamiquement par un utilitaire. Cela a formidablement contribué au succès de MYRINET et des protocoles réseaux adaptés ont vu le jour dans de nombreuses bibliothèques de communication telle BIP [PT98](*Basic Interface for Parallelism*), FM [PLC95] (*Fast Messages*) ou dans des implémentations du standard MPI [2] (*Message Passing Interface*).

Pour piloter la carte réseau, l'interface de communication MX (*Myrinet eXpress*)

2. Le standard MPI est une norme définissant une interface de programmation portable et performante pour le passage de messages

est fournie par le constructeur. Bien que considérée comme une interface de bas niveau, elle offre un ensemble de primitives, principalement non-bloquantes, similaires à celles du standard MPI. Les mécanismes de transfert sont adaptés en fonction de la taille des données et dissimulés au programmeur. Il est par exemple plus efficace de laisser le processeur piloter le transfert depuis la mémoire centrale vers la carte réseau (mode PIO) dans certains cas (pour des messages de petite taille), ou bien de déléguer ce travail à la carte réseau (mode DMA) dans d'autres. L'interface dispose également d'un puissant système de réception sélective permettant de faire correspondre les données transitant sur le réseau et les requêtes postées par l'application.

Les temps de latences obtenus de l'ordre de 2 μs et les débits allant jusqu'à 9,9 Gbit/s sur INTEL XEON X5460 cadencé à 3,6 GHz en font une technologie performante et très attrayante.

Infiniband Vers la fin des années 90, plusieurs grands constructeurs (dont IBM, Hewlett-Packard, SUN, Intel, etc.) ont mis en commun leurs connaissances pour définir une nouvelle norme d'interconnexion, destinée au départ à remplacer le bus d'entrées/sorties PCI, l'accès aux périphériques de stockage et les réseaux. Suite au retrait d'Intel qui a préféré concentrer ses efforts sur le remplaçant de PCI, PCI Express, les recherches ont été restreintes aux réseaux hautes performances.

Les spécifications matérielles et logicielles d'Infiniband étant publiques, plusieurs constructeurs (Voltaire, Topspin, Cisco) proposent leurs équipements réseau ainsi qu'une interface de communication respectant plus ou moins la norme logicielle : toutes les méthodes ne sont pas forcément implementées et leur syntaxe varie d'une implémentation à l'autre. Cette norme, intitulée Verbs, suit le paradigme de communication par accès mémoire distante : RDMA(*Remote Direct Memory Access*). Le principe est d'écrire ou de lire une donnée depuis l'espace utilisateur d'un processus d'une machine donnée vers l'espace utilisateur d'un autre processus qui s'exécute sur un noeud distant. Face à la multitude de distributions logicielles propriétaires, le projet open source Open Fabrics [Opea] propose une bibliothèque de communication unifiant les différentes interfaces et compatible avec toutes les cartes. Bien qu'il existe des interfaces de plus haut niveau, Verbs reste majoritairement utilisée car elle profite des capacités matérielles à effectuer du RDMA. La contre-partie est que les optimisations sont laissées à la charge de l'utilisateur : il doit gérer explicitement les zones mémoires pour les transferts RDMA, choisir le mode de transfert, etc. Une fois l'ensemble maîtrisé, l'utilisateur peut escompter obtenir une latence de moins de 1 μs et des débits allant jusqu'à 40 GBit/s. Grâce à ces performances, Infiniband a su se faire une place au sein des grappes de calcul du Top500 à tel point qu'au mois de Juin 2011, environ un tiers d'entre elles l'utilisait.

Gigabit Ethernet Ethernet est à la base des premiers réseaux de stations de travail et bien qu'il s'agisse d'un matériel destiné aux réseaux d'entreprises, au service de l'Internet ou des particuliers, la plupart des installations pour le calcul parallèle intensif

en sont équipées. De surcroît, pour nombre d'entre elles il s'agit du seul réseau d'interconnexion disponible. Des travaux ont donc été entrepris pour corriger les défauts d'Ethernet afin qu'il puisse concurrencer les autres technologies. D'un côté, certains fabricants ont modifié le matériel en intégrant par exemple un circuit capable de réaliser des transferts DMA. Cependant, au vu de la hausse de prix qui en résulte, Ethernet perd de son intérêt. Une autre voie consiste à se dispenser de la couche logicielle TCP/IP et de mettre en place des protocoles adaptés aux infrastructures hautes performances. Ainsi des protocoles tels FCoE (*Fiber Channel over Ethernet*) ou AoE (*ATA over Ethernet*) reposent directement sur Ethernet pour communiquer avec les supports de stockage, ce qui améliore significativement les performances des entrées/sorties. De la même manière, le court-circuitage de TCP/IP est une problématique répandue pour les bibliothèque de communication offrant une interface de haut-niveau tel que le passage de message. L'un des pionniers dans ce domaine est le projet GAMMA [CC00] qui a développé une couche de communication basée sur le paradigme Active Message [vECGS92] directement au-dessus d'Ethernet. Toutefois, la nécessité de modifier les pilotes Ethernet, si nombreux, et les problèmes d'interopérabilité avec la couche TCP/IP, ont été autant de freins au succès du projet. Des pistes similaires ont été empruntées, parmi lesquelles nous pouvons citer Open-MX [Gog11] qui vise à porter l'interface de communication MX (des réseaux Myrinet) sur les réseaux Ethernet. Contrairement aux autres solutions, celle-ci interagit avec les cartes réseaux au travers d'une interface haut-niveau du noyau, ce qui a l'avantage d'être générique. De plus, Open-MX affiche d'excellentes performances : sur un réseau Ethernet-10G, la latence réseau peut atteindre 5,6 μs et les débits obtenus sont de l'ordre de 10 Gbit/s.

Les progrès réalisés par les constructeurs de matériel réseau sont une des raisons majeures du succès des grappes de calcul. La réduction sensible de la latence qui a pu atteindre l'ordre de la microseconde et l'augmentation tout aussi remarquable du débit (plusieurs Go/s) résultent des nombreuses optimisations réalisées tant au niveau du matériel qu'au niveau des pilotes logiciels qui les contrôlent.

1.1.3.3 Mémoire partagée par réseau externe

Les grappes de calcul offrent une puissance de calcul très intéressante au regard de leur prix. Toutefois l'utilisateur doit composer avec l'hétérogénéité des technologies réseau présentes ainsi qu'avec les méthodes de transfert qui y sont liées. Plutôt qu'une approche logicielle offrant une abstraction de haut niveau, certains constructeurs ont tenté d'apporter une réponse matérielle à cette problématique. L'idée est de relier plusieurs machines par un réseau hautes performances puis de dissimuler l'interconnexion à l'utilisateur en donnant l'illusion d'une unique machine à mémoire partagée. L'objectif est de cumuler la puissance d'un ensemble de machines tout en simplifiant le travail du programmeur.

SGI NUMAlink Au milieu des années 90, le constructeur SGI [SGIa] (*Silicon Graphics Inc.*) propose un supercalculateur basé sur une architecture mémoire dite cc-NUMA (*cache coherent Non Uniform Memory Access*). Désignée sous le nom d'ORIGIN 2000 [LL97a], elle se compose d'un ensemble de noeuds reliés entre eux par un réseau d'interconnexion propriétaire, le NUMALINK. Chaque noeud contient un ou deux processeurs, la mémoire principale et une mémoire (le directory) destinée à connaître l'état des caches distants pour assurer la cohérence mémoire. Jusqu'à aujourd'hui, la technologie d'interconnexion NUMALINK a connu plusieurs évolutions. À chaque nouvelle génération de supercalculateurs, SGI a développé un réseau d'interconnexion plus performant : le débit double d'une version à l'autre. La dernière en date, NUMALINK 5 équipe la machine SGI ALTIX UV [SGIb] et offre un débit de 15 Go/s (en cumulant deux liens unidirectionnels de 7,5 Go/s). Si les performances de la technologie réseau se sont grandement améliorées au cours du temps, les bus mémoire ont également vu leur capacité et leur vitesse de transfert s'améliorer. La conséquence est qu'il est, proportionnellement parlant, beaucoup plus coûteux de communiquer avec les processeurs placés sur des noeuds distants qu'avec ceux appartenant à la même puce. Du point de vue de l'utilisateur, il s'agit d'une collection de processeurs disposant d'un vaste espace mémoire. Aucune information de localisation architecturale ne permet d'agencer les processus ou les threads pour minimiser les accès distants. Ce point est souvent critique pour nombre d'applications où les processus ou les threads doivent être proches des données sur lesquelles ils travaillent.

Dolphin SCI Initialement destiné à remplacer toute connectique à l'intérieur d'un ordinateur, le standard SCI [oEEE93](*Scalable Coherent Interface*) a subi beaucoup de changements par rapport au concept original et plusieurs versions ont été développées par différentes entreprises telles que DOLPHIN INTERCONNECT SOLUTIONS [Dol, LK99], CONVEX ou CRAY RESEARCH [Cra]. Parmi ces variantes, le constructeur DOLPHIN INTERCONNECT SOLUTIONS a conçu une carte PCI et PCI-EXPRESS qui permettent de mettre en place un réseau à capacité d'adressage. Grâce à ces cartes réseau, il est possible de relier les mémoire de processus distants et ainsi de construire des machines à mémoire partagée qui sont aisément extensibles : pour étendre l'espace mémoire disponible, il suffit de connecter une nouvelle machine. Le principe est de projeter certaines pages mémoire d'un processus distant dans l'espace d'adressage du processus local. Cela s'effectue facilement en utilisant la plage d'adresses correspondante à celle allouée à la carte SCI pour les opérations *load/store* usuellement employées et correspondant aux accès mémoire d'un programme. Chaque carte dispose d'une table de pages qui associe une plage d'adresse locale à une plage d'adresse distante. Lorsque la carte reçoit une opération appliquée à une adresse, la correspondance permet de retrouver le noeud distant et l'adresse relative sur ce noeud. La figure 1.7 montre une écriture distante réalisée par les cartes réseau SCI : le processus sur le noeud de gauche écrit la valeur 12 dans une variable stockée sur une page mémoire d'un processus distant ; le contrôleur du bus d'entrées/sorties redirige donc l'écriture vers la carte SCI qui transmet la modification à la carte du noeud distant, qui à son tour va l'écrire en mémoire. Par de

multiples aspects, c'est une solution avantageuse, toutefois garantir de bonnes performances implique de revoir à la baisse les critères de cohérence des données : l'utilisation de mémoire tampon en émission pour améliorer le débit global, et de mémoire cache en réception, pour grouper les accès à la mémoire, tendent à différer la transmission des informations de cohérence. Il s'agit donc de trouver un compromis entre la préservation de la validité des données sur l'ensemble des machines et l'efficacité globale du système.

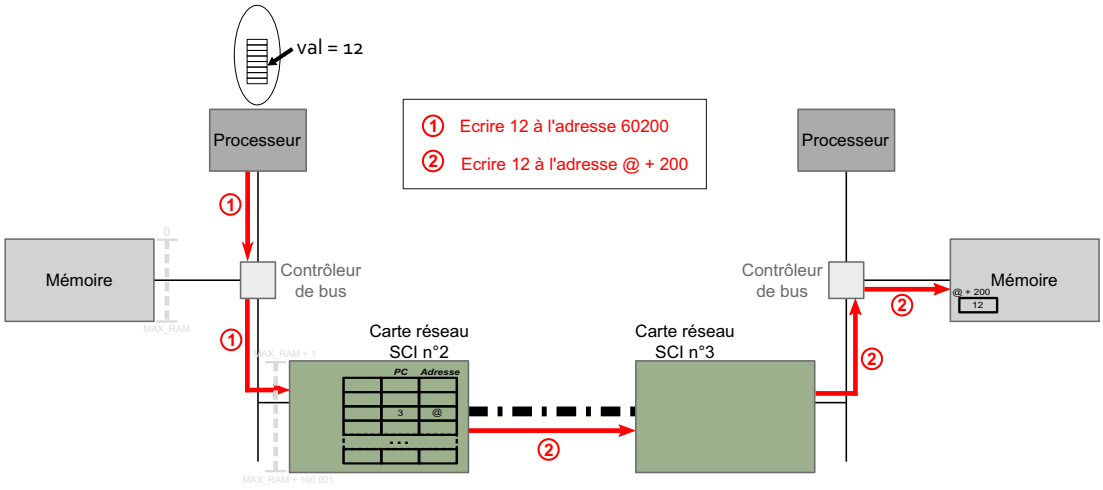


FIGURE 1.7 – Exemple d'écriture distante grâce au réseau SCI

En dépit des efforts réalisés pour améliorer les performances de ces solutions, les grappes de calcul continuent de dominer le monde du calcul hautes performances. La puissance de calcul qu'elles offrent est réellement intéressante au vu de leur prix, et ceci grâce aux recherches effectuées sur les technologies réseau. Cependant il ne faut pas exclure le rôle des noeuds dans ces grappes dont l'évolution est tout aussi importante.

1.2 Architecture des ordinateurs

Les systèmes de calcul actuels ont une structure complexe regroupant un ensemble d'unités de calcul. Que ce soit un supercalculateur doté d'un réseau d'interconnexion propriétaire ou bien une grappe de calcul reliant plusieurs ordinateurs par un réseau hautes performances, l'idée directrice est de faire fonctionner de concert un groupe de processeurs. À l'image de ces systèmes, les constructeurs de microprocesseurs tels INTEL ou AMD ont consacré leurs efforts à accélérer l'exécution d'instruction par tous les moyens possibles et ceci au cœur même des processeurs. Cette section détaille les progrès technologiques réalisés dans ce domaine qui ont mené aux processeurs d'aujourd'hui.

1.2.1 L'évolution au cœur des processeurs

Exprimée en 1965 et revue dix ans plus tard, la loi de Moore [Moo06, Moo75] postule un doublement du nombre de circuits tous les dix-huit mois, ce qui se traduit par des performances deux fois plus importantes concernant la mémoire et les processeurs. Les constructeurs ont eu recours à diverses techniques pour atteindre cet objectif, consistant le plus souvent à paralléliser les traitements.

1.2.1.1 La course à la fréquence

Dans un premier temps, la montée en fréquence des unités de traitement a été le fer de lance du développement des micro-processeurs. Ainsi les applications pouvaient bénéficier d'une accélération de la vitesse de traitement à chaque nouvelle génération du matériel sans modifier le code, en s'appuyant uniquement sur les mécanismes internes des processeurs (cf. figure 1.8).

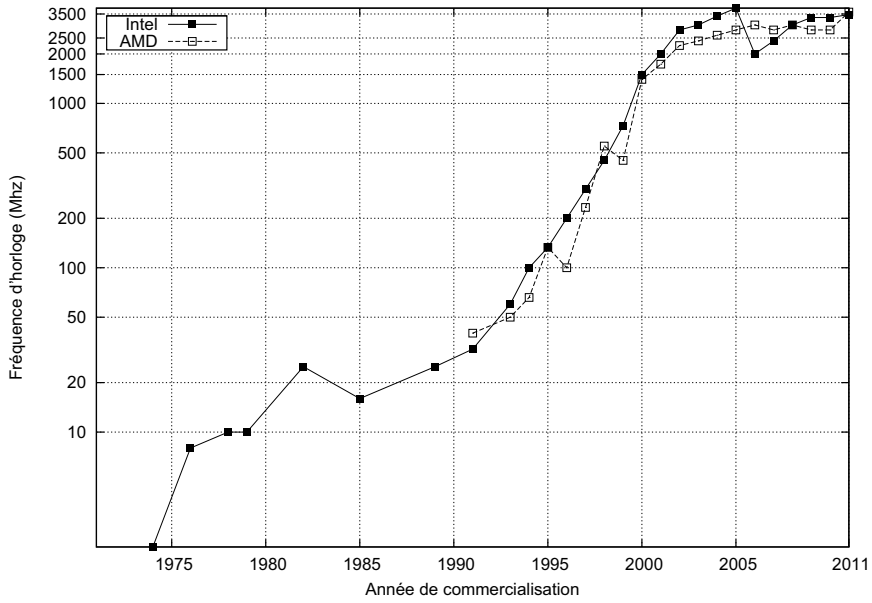


FIGURE 1.8 – Évolution de la fréquence des processeurs de 1971 à 2011

Depuis le milieu des années 80, les constructeurs font appel à une technique très simple et directement inspirée des chaînes de montage : le pipeline. Le principe est de décomposer le chemin du traitement d'une instruction en plusieurs étapes : chargement, décodage, exécution, etc. Chacune de ces étapes et réalisée par un circuit spécifique et indépendant des autres. Ainsi il n'est pas nécessaire d'attendre l'accomplissement d'une instruction pour commencer la suivante : un seul flot d'exécution se divise alors en plusieurs instructions réparties sur différentes unités fonctionnelles. La figure 1.9 donne

une représentation schématique d'un pipeline à trois étages où trois instructions peuvent s'exécuter de façon concurrente. Il faut noter que l'étage le plus lent va déterminer la fréquence d'horloge du processeur. De ce fait, certaines unités fonctionnelles sont dupliquées afin de réaliser les traitements spécifiques (calcul en virgule flottante par exemple) ou tout du moins nécessitant plus de cycles d'horloge que les autres, l'objectif étant de minimiser les situations bloquantes qui freinent la progression des instructions moins coûteuses.

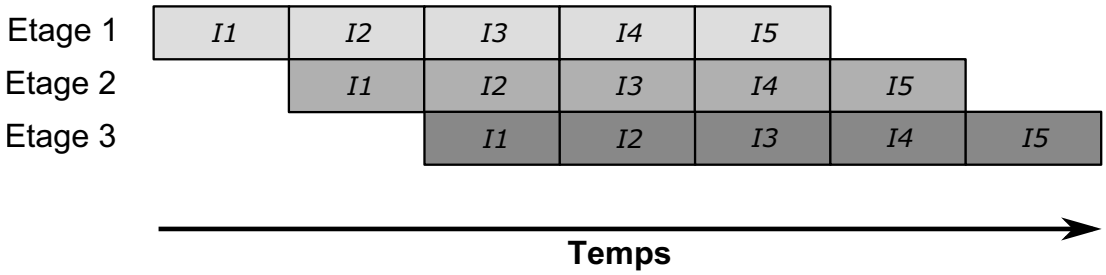


FIGURE 1.9 – Progression de cinq instructions dans un pipeline à trois étages

D'autres techniques comme la prédiction de branchement, la réorganisation dynamique d'instructions ou encore le renommage des registres sont autant d'optimisations qui minimisent le temps passé dans le pipeline et donc améliorent l'efficacité globale du processeur. Il subsiste toutefois certains ralentissement survenant dans le pipeline et qu'il est difficile de faire disparaître : l'impossibilité de mettre en concurrence deux instructions du fait d'une interdépendance ou encore le délai d'accès à la mémoire à cause d'un défaut de cache mettent à mal les performances du processeur.

1.2.1.2 Parallélisme de données

Une architecture à parallélisme de données se compose d'un certain nombre d'unités de calcul réalisant un traitement similaire sur un ensemble de données. Le parallélisme concerne alors les données et non plus les instructions.

Parmi ces architectures, citons les processeurs vectoriels spécialement créés et optimisés pour appliquer une opération élémentaire sur un vecteur de nombres. Des registres vectoriels sont ainsi proposés dans l'architecture CRAY [Rus78] pour éviter d'opérer directement en mémoire. D'autres constructeurs tels NEC [WM85, ATH90] ou FUTJISU [TKI85, UHYH90] ont également participé au développement des processeurs vectoriels dans le monde des supercalculateurs. Il s'agissait d'ailleurs de l'architecture dominante jusqu'au milieu des années 90. Et bien que nettement moins présente dans les années 2000, celle-ci a su prouver son efficacité au travers de l'EARTH SIMULATOR [HUYK04], constitué de 5120 processeurs vectoriels d'architecture NEC SX-6, qui a su rester première au classement mondial de novembre 2002 à juin 2004. Plus récemment, une seconde machine de calcul basée sur le processeur d'architecture NEC-SX 9

est classée 68ème au dernier classement du TOP500.

De nos jours, ce sont les processeurs scalaires qui dominent le marché cependant la plupart d'entre eux disposent de mécanismes issus des architectures vectorielles. Des instructions permettent d'appliquer une opération à des registres de grande taille. Chez INTEL, c'est le jeu d'instructions SSE [RPK00b] (successeur du MMX [PW96] et apparu avec le PENTIUM III) qui offre des registres de 128 bits capables d'accueillir 4 nombres flottants 32 bits. Le parallélisme généré est à une échelle plus réduite que les architectures vectorielles vues précédemment et leur application concerne surtout des traitements spécifiques comme l'encodage vidéo.

1.2.1.3 Le multithreading

Les constructeurs n'ont eu de cesse de réduire au maximum la taille des composants afin d'en rajouter autant que possible sur l'espace disponible. En s'appuyant sur une finesse de gravure qui s'accroît d'une année sur l'autre, on a pu rendre les pipelines plus complexes et y intégrer des étages supplémentaires pour maximiser le parallélisme. Bien que continuent les progrès en matière de gravure, un obstacle inévitable a stoppé la progression vers des fréquences plus élevées : la barrière thermique. Les composants ne pouvaient supporter de fonctionner à des températures trop élevées sans dommages physiques. Ainsi, INTEL prévoyait de développer le processeur TEJAS, basé sur un pipeline à 45 étages, et atteindre une fréquence de 7 Ghz, mais les problèmes de dissipation thermique ont eu raison de ces ambitions. Dans ces conditions, comment fabriquer de nouveaux processeurs plus performants que les anciens ?

Une solution envisagée a été de permettre à plusieurs threads matériels [3] d'accéder aux unités fonctionnelles de façon entrelacée, ce qui implique de gérer efficacement plusieurs contextes de threads en parallèle. Ainsi, il existe deux types de multithreading au sein des processeurs : le *Fine-grained Multithreading* (FMT) et le *Coarse-grained Multithreading* (CMT). Dans le premier cas, à chaque cycle, l'état du thread courant est sauvegardé puis l'on change de thread, pourvu qu'il y en ait un de prêt. De cette manière les temps d'accès à la mémoire sont recouverts par l'exécution d'un autre flux d'instructions. L'architecture Tera [HK97] permettait de gérer matériellement 128 contextes de threads en parallèle et n'utilisait pour ainsi dire pas de mémoire cache : l'arrivée des données dans le processeur devient alors prédictible. Le *Coarse-grained Multithreading* quant à lui, réalise des changements de contexte uniquement en réponse à certains événements, comme les accès à une mémoire distante dans un système à mémoire partagée ou suite à un défaut de cache. Le matériel gère donc beaucoup moins de contextes simultanément et le temps de changement de contexte peut atteindre plusieurs cycles sans que cela affecte les performances : 3 cycles étaient nécessaires au processeur PULSAR [BEKK00], sorti de chez IBM en 2000, pour passer d'un thread à un autre. Plus récemment, l'INTEL ITANIUM 2 [MS03], dénommé *Montecito*, s'appuie également sur le

3. Un thread matériel correspond à un ensemble de circuits constitué principalement par un jeu de registre de données et de contrôle ainsi que d'un contrôleur d'interruptions permettant de gérer le contexte d'un flot d'exécution

CMT et réalise un changement de contexte en 15 cycles.

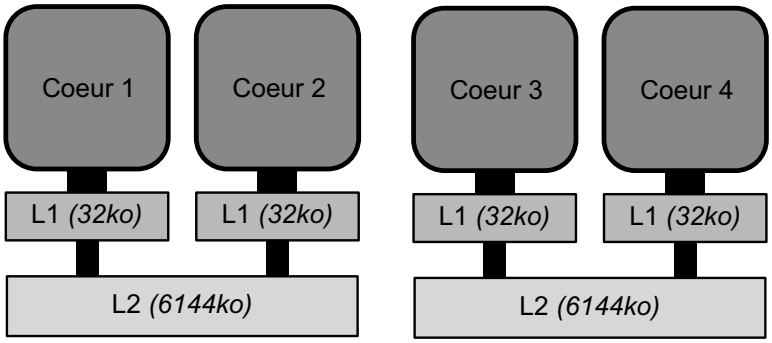
En 1995, l'équipe de H. Levy propose la technique du *Simultaneous Multithreading* [TEL95] (SMT). L'idée est de permettre l'exécution simultanée de plusieurs flots d'instructions indépendants sur un même pipeline. La plupart des ressources du processeur (mémoire cache, unités fonctionnelles, etc.) sont partagées, seul un ensemble de registres est dupliqué et le processeur est capable de récupérer des instructions provenant de plusieurs threads en un cycle. De ce fait, le système d'exploitation perçoit ces répliques comme autant de processeurs à disposition. L'implémentation du SMT la plus connue est l'*Hyperthreading* [KM03] de chez INTEL présente dans le modèle *Northwood* (XEON et PENTIUM 4) et qualifiée de modèle SMT à deux voies puisque deux flots d'instruction se partage le pipeline du processeur. Toutefois l'emploi d'une telle technologie n'est pas forcément synonyme de succès, car ce dernier va dépendre des calculs à réaliser pour les applications et de la disponibilité des unités fonctionnelles du processeur. Supposons que l'on veuille réaliser deux boucles de calcul flottant en parallèle : si deux unités de calcul flottant sont présentes, les deux calculs vont pouvoir s'effectuer en parallèle ; dans le cas contraire, les deux threads doivent se partager l'unique unité de calcul flottant et affecte le temps d'exécution. Dans le cadre de programmes moins triviaux, comme ceux du calcul scientifique, les résultats sont donc difficilement prévisibles.

1.2.1.4 L'avénement des multicœurs

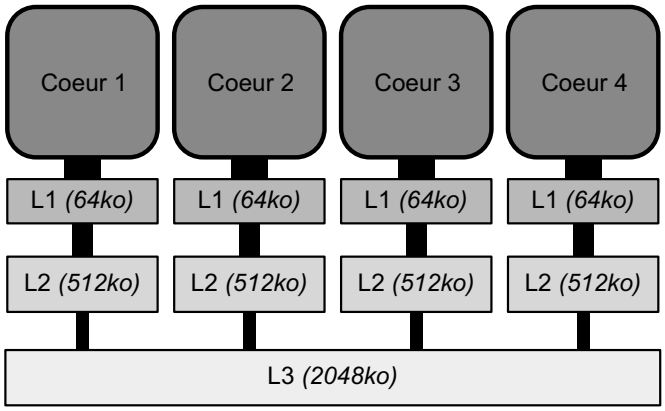
Comme nous l'avons dit précedemment, les progrès des techniques de gravure ont apporté aux constructeurs plus d'espace sur le silicium. Plutôt que de profiter de cet espace pour complexifier la structure interne du processeur, certains constructeurs ont pensé à dupliquer le processeur lui-même. Ainsi, disposées sur une même puce, plusieurs unités de calcul, surnommées cœurs, se partagent l'accès à une ou plusieurs mémoires cache et au bus de données. Ces cœurs sont identiques aux processeurs de la génération précédente : il s'agit des mêmes processeurs avec une taille de pipeline réduite. Le principe de construction de ces processeurs, dits *multicœurs*, est similaire d'un constructeur à l'autre. De plus, chaque coeur va disposer d'une mémoire cache dédiée et très rapide, dite de niveau 1, capable de stocker une petite quantité d'informations (64 ko pour les processeurs INTEL NEHALEM[4] et 128 ko pour les processeurs AMD OPTERON *Barcelona*[5]). La différence entre les modèles de processeurs multicœurs des constructeurs (INTEL, AMD, IBM, SUN) réside dans la disposition des cœurs et des mémoires cache de niveau supérieur (2 et 3). Ces dernières, de plus grande taille et moins rapides, sont en règle générale partagées entre les coeurs d'une même puce. Au travers de ces unités mémoire, ils peuvent ainsi partager des données auxquels ils accéderont plus rapidement qu'en mémoire principale. La figure 1.10 présente la structure de deux puces multicœurs. Sur la première (1.10(a)), chaque cœur du processeur INTEL XEON *Harpertown* dispose de son propre cache de niveau 1 et le cache de niveau 2 est commun aux quatre cœurs.

4. 32 ko pour les instructions et 32 ko pour les données
5. 64 ko pour les instructions et 64 ko pour les données

La seconde représente un processeur AMD Opteron *Barcelona* où les caches des deux premiers niveaux sont associés à chaque cœur et le cache de niveau 3 est partagé entre les quatre coeurs.



(a) Intel Xeon quadri-cœur *Harpertown E5420*



(b) AMD Opteron quadri-cœur *Barcelona 8347HE*

Figure 1.10 – Hiérarchie des caches de puces multicœurs

Actuellement les processeurs du marché grand public se déclinent en version bi-cœurs et quadri-cœurs, voire hexa-cœurs : le nombre de cœurs est même devenu un argument de vente aux dépens de la fréquence, reléguée à un rôle plus informatif de nos jours. Prenons pour exemple les derniers processeurs d'Intel, les *core i*, sont constitués de 2, 4 ou 6 cœurs. Pour des objectifs plus professionnels, les constructeurs proposent également des processeurs avec un plus grand nombre de cœurs, comme AMD avec son processeur Magny Cours [CKD+10] disposant de 6, 8 ou 12 cœurs. Le constructeur prévoit même sur sa dernière « feuille de route »(*roadmap*) des processeurs à 20 coeurs (processeur Terramar).

1.2.2 Des machines de plus en plus peuplées ...

En marge des progrès réalisés à l'intérieur même des processeurs, les constructeurs ont également fait évoluer l'architecture des machines de calcul. Ils ont eu l'idée de conjuguer la puissance de plusieurs processeurs en les interconnectant au sein d'une même machine. Ces architectures baptisées *multiprocesseurs* se scindent en deux grandes familles : les multiprocesseurs symétriques (SMP) et les multiprocesseurs à accès mémoire non-uniformes (NUMA).

1.2.2.1 Multiprocesseurs symétriques

Les systèmes monoprocesseurs « classiques »sont composés d'un processeur associé à une hiérachie de caches et connecté via un bus à la mémoire et aux autres périphériques d'entrées/sorties, comme l'illustre la figure 1.11. Un système multiprocesseur symétrique se construit en ajoutant un ou plusieurs processeurs (accompagné d'un cache) qui sont connectés au bus. Tous les processeurs accédent à la mémoire commune par ce bus et ce de façon uniforme : le temps d'accès est le même pour tous. Un exemple d'architecture SMP est visible en figure 1.12, où 4 processeurs sont reliés au bus.

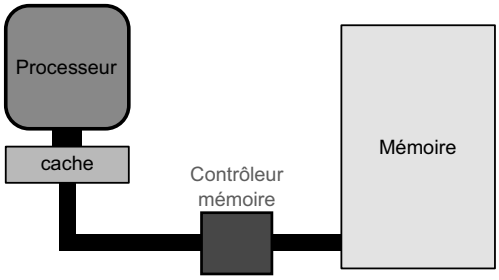


FIGURE 1.11 – Architecture monoprocesseur

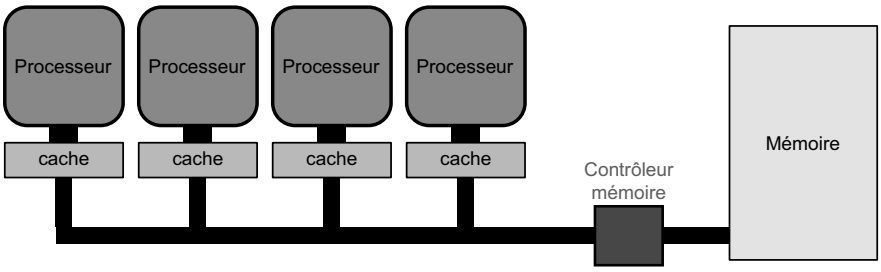


FIGURE 1.12 – Architecture SMP à quatre processeurs

En considérant la mémoire et son état, le système doit assurer une certaine cohérence quant aux données qui y sont stockées [AG96]. Prenons un exemple : soient deux

processeurs d'un système multiprocesseur exécutant en parallèle un code dans lequel ils accèdent à une même donnée ; lorsque l'un des deux processeurs modifie sa valeur (c'est-à-dire une zone mémoire dans son cache), la prochaine lecture de la donnée par le second processeur doit prendre en compte cette nouvelle valeur. Dans le cadre des architectures SMP, la cohérence des données sera assurée par le contrôleur de caches de chaque processeur. Celui-ci va « écouter »le bus (*bus snooping*) afin d'identifier toute requête liée aux données contenues dans son cache et réaliser les actions nécessaires (invalidation ou modification d'une ligne de cache).

La construction de ces systèmes multiprocesseurs symétriques est assez simple, cependant les accès à la mémoire principale étant sérialisés, le bus devient un goulot d'étranglement lorsque le nombre de processeurs accédant simultanément à des données en mémoire devient trop important. Et le trafic dû aux protocoles de cohérence de cache accentue cette contention. Par conséquent, les constructeurs limitent la quantité de processeurs des ces architectures. La majorité des SMP se composent de deux ou quatre processeurs, voire seize mais rarement plus.

1.2.2.2 Accès mémoire non-uniformes

Pour pallier les problèmes de passage à l'échelle des systèmes SMP les constructeurs ont développé des machines multiprocesseurs avec une mémoire répartie. Un groupe de processeurs sont ainsi reliés à un *banc mémoire* et l'ensemble forme un *nœud NUMA*(*Non-Uniform Memory Access*). Chaque noeud NUMA peut être assimilé à une architecture SMP dans laquelle chaque processeur accède uniformément au banc mémoire. Tous les noeuds NUMA sont reliés entre eux par un réseau d'interconnexion ou un commutateur, comme le montre la figure 1.13. Ainsi le temps que met le processeur d'un nœud NUMA pour accéder à une donnée stockée dans le banc mémoire d'un autre nœud NUMA est plus important que s'il accède à une donnée en mémoire locale (à son noeud). On définit alors le *facteur NUMA* qui correspond au rapport entre le temps d'accès à la mémoire locale et le temps d'accès à la mémoire distante. Plus ce facteur est élevé, plus l'impact du placement des données sur l'exécution des programmes sera important.

La cohérence de cache, dont nous parlions précédemment, est assurée par un circuit spécifique, appelé *répertoire*, associé à chaque noeud NUMA. Ce répertorie mémorise la liste des processeurs accédant aux données dont il a la charge (c'est-à-dire, celles du banc mémoire). Lorsqu'une de ces données est modifiée, le répertoire envoie des requêtes d'invalidation seulement aux processeurs dont les caches en contiennent une copie. Le trafic sur les bus est ainsi minimisé, ce qui permet de mieux passer à l'échelle.

Le recours aux machines NUMA s'est avéré une bonne solution pour construire des machines de plusieurs dizaines de processeurs performantes. Elles ont d'ailleurs connu un certain succès dans les années 90, notamment avec les machines SGI Origin [LL97b] ou IBM SP. Cependant, les grappes de calcul, d'un meilleur rapport performances/prix, les ont peu à peu supplanté, comme nous avons pu le voir en 1.1.2.

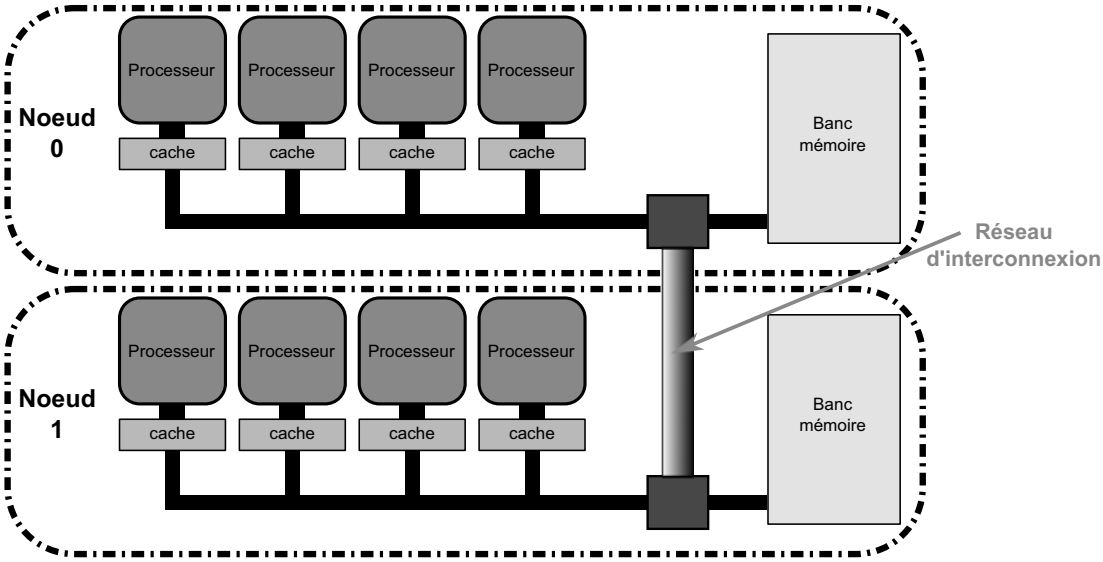


FIGURE 1.13 – Architecture NUMA avec deux nœuds à quatre processeurs

Plus récemment, l'architecture NUMA a fait son retour dans le domaine du calcul hautes-performances avec l'émergence des processeurs multicœurs. Jusqu'alors l'assemblage de plusieurs processeurs consistait à les connecter à un même bus mémoire. Face aux problèmes de contentions d'une telle solution, AMD a développé le réseau d'interconnexion HYPERTRANSPORT [AMD01]. Ainsi, les processeurs AMD OPTERON utilisent cette technologie pour se connecter à la mémoire, aux périphériques d'entrées/sorties et aux autres processeurs. Le dialogue entre ces entités est effectué par des communications point-à-point sur les liens HYPERTRANSPORT. La figure 1.14 montre l'architecture interne d'un processeur de type OPTERON qui dispose d'un lien direct avec la mémoire et de 3 autres liens. Le couple *{processeur - banc mémoire}* forme l'équivalent d'un nœud NUMA. Un processeur peut donc accéder à la mémoire d'un autre processeur au travers des différents liens d'interconnexion au prix d'une latence plus élevée que s'il s'agissait de son propre banc mémoire [AJR06].

Face au succès des processeurs OPTERON, INTEL a développé sa propre technologie d'interconnexion QUICKPATH INTERCONNECT (QPI) pour la conception de ses architectures les plus récentes telle que NEHALEM. Construite pour remplacer le bus mémoire (FSB : *Front Side Bus*), elle offre un débit théorique de 25,6 Gbit/s [Cor09] équivalent au débit offert par la technologie HYPERTRANSPORT dans sa dernière version (3.1). La structure des processeurs équipés de cette technologie est relativement similaire aux processeurs AMD utilisant HYPERTRANSPORT : un contrôleur mémoire permet au processeur (qui peut être multicœur) d'accéder directement à son banc mémoire et plusieurs liens QPI permettent de le connecter à d'autres processeurs ou à des périphériques d'entrées/sorties.

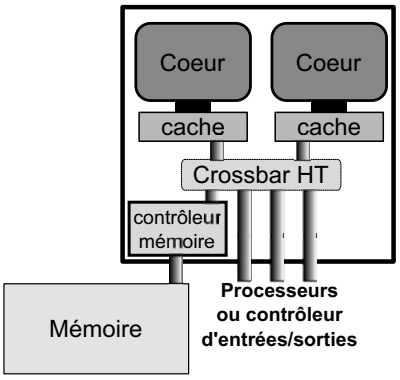


FIGURE 1.14 – Architecture d'un processeur Opteron à deux cœurs

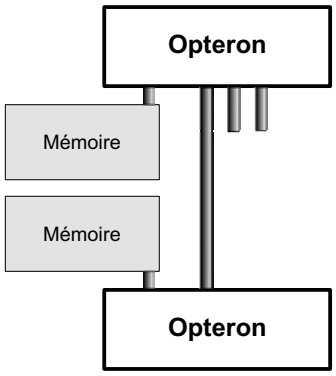


FIGURE 1.15 – Interconnexion de deux processeurs Opteron

Près de 78% des processeurs qui équipent les machines du TOP500 utilisent soit la technologie QPI soit la technologie HYPERTRANSPORT. Nous assistons ainsi au retour du NUMA dans les architectures de calculateurs et les stratégies de placement mémoire reviennent sur le devant de la scène avec des accès mémoires privilégiés pour les cœurs d'une même puce partageant une mémoire cache, pour les processeurs accédant à des données situées sur leur propre banc mémoire, etc.

1.2.3 ... et hétérogènes

1.2.3.1 L'attrait des accélérateurs graphiques

Ces dernières années, les processeurs graphiques ont suscité l'intérêt de la communauté scientifique. À l'origine ces processeurs étaient des accélérateurs spécialisés pour réaliser rapidement des calculs, soulageant ainsi les processeurs *généralistes*. Les besoins grandissant de l'industrie des jeux vidéo ont favorisé le développement de ces circuits qui sont devenus de véritables architectures parallèles capables de réaliser simultanément un grand nombre de traitements similaires. La puissance de calcul est nettement supérieure à celles qu'offre les processeurs multicœurs actuels. Pour exemple, le processeur graphique NVIDIA TESLA M2050 [Cor10] délivre une puissance théorique de 515 Gflop/s pour les calculs en double précision bien supérieure aux 109 Gflop/s qu'offre le processeur INTEL *Core i7 980XE*.

Aussi attrayante que puisse paraître une telle puissance, la programmation de ces accélérateurs nécessite une connaissance fine de leur architecture très particulière et les codes doivent être repensés pour fournir suffisament de parallélisme aux milliers de threads qui s'exécuteront. Les constructeurs offrent ainsi des interfaces de programmation bas-niveau : CUDA [Cud] pour NVIDIA et CAL [AMD10a] pour AMD, utilisées en lieu et place des APIs [6] graphiques standards. Bien que de nombreux efforts se portent

6. *Application Programming Interface*, c'est-à-dire une interface de programmation pour les appli-

sur l'écriture de noyaux de calcul efficaces [OLG+07, VD08], certaines initiatives proposent une approche unifiée afin de simplifier le travail du développeur d'applications. Le standard OpenCL [Khr11] offre un modèle de programmation, inspiré de CUDA, mais commun aux différentes technologies (processeurs généralistes et accélérateurs). Cependant l'interface reste assez bas niveau notamment au travers des mouvements de données explicites. Dans un registre de plus haut niveau, d'autres environnements de programmation comme HMPP [DBB07] ou StarSS [ABI+09, BPBL09] s'appuient sur des directives placées dans l'application pour exécuter certaines sections de codes sur les accélérateurs. L'utilisateur est soulagé des transferts mémoire et synchronisations, parfois au détriment des performances. Des solutions intermédiaires comme le support exécutif StarPU [ATNW11] propose une plate-forme pour l'exécution des tâches à la fois sur les accélérateurs et les processeurs multicœurs pour bénéficier de l'intégralité de la puissance de calcul. Plusieurs algorithmes ont déjà montré le gain significatif d'une telle solution [AAD+10, AAD+11a, AAD+11b].

L'utilisation des accélérateurs graphiques dans les applications scientifiques reste un défi pour nombre d'applications, cependant nul doute que celui-ci sera relevé au vu de la propagation de ces architectures dans le paysage du calcul intensif. En effet, trois des cinq premiers calculateurs du Top500 sont des machines hybrides utilisant à la fois des processeurs multicœurs et des accélérateurs graphiques : le *Tianhe-1A* du *National Supercomputing Center* à Tianjin, le *Nebulae* du *National Supercomputing Centre* à Shenzhen et le *Tsubame 2.0* du *GSIC Center* à Tokyo sont respectivement second, troisième et cinquième. Les gains en efficacité énergétique sont également impressionnants : au classement du Green500 [Gre], qui classe les calculateurs du Top500 en fonction de leurs performances par Watt, les trois machines précédentes font partie des vingts premières alors que le système *Jaguar* de l'*Oak Ridge National Laboratory*, classé troisième au Top500 et équipé de processeurs à six cœurs de type *Opteron*, n'atteint que la soixante-treizième place.

1.2.3.2 Les processeurs hybrides

Devant le succès rencontré par les accélérateurs graphiques dans le monde du calcul scientifique au gain significatif de performances pour les applications grâce à un développement de modèles et supports exécutifs performants, les constructeurs de processeurs ont intégrés des circuits accélérateurs graphiques directement sur la puce. Dans ce type d'architecture, les cœurs et l'accélérateur graphique partagent un même lien vers la mémoire principale. Des mécanismes de transfert par bloc sont également présents pour gérer les mouvements de données entre les cœurs et l'accélérateur, évitant ainsi d'utiliser le bus externe. Cela atténue les risques de contention et réduit le surcoût engendré par les transferts mémoire.

Dernière née du constructeur Intel, l'architecture *Sandy Bridge* [san10] correspond à l'association d'un accélérateur graphique et d'un processeur multicœur (avec deux,

cations

quatre ou six cœurs hyperthreadés) sur la même puce. Le dernier niveau de cache (à priori le troisième) est réparti en plusieurs blocs et partagé entre les cœurs et l'accélérateur graphique. De son côté, AMD propose l'architecture *Fusion* [AMD10b, DAF11] assez similaire à celle d'Intel. Bien que les cœurs et l'accélérateur partagent la même connexion vers la mémoire principale, celle-ci est divisée en plusieurs régions : des régions gérées par le système d'exploitation qui s'exécute sur les cœurs et des régions gérées par les applications utilisant l'accélérateur. C'est à la charge des développeurs d'applications ou de supports exécutifs de recouvrir les copies de blocs mémoire entre ces régions par des calculs impliquant d'autres blocs.

La complexité de ces processeurs de nouvelle génération va en grandissant notamment en raison de la forte hétérogénéité des unités de calcul qui les composent. Les constructeurs prévoient d'ailleurs de continuer sur cette lancée en intégrant toujours plus d'unités de calcul spécialisées aux côtés des cœurs généralistes.

1.3 Les calculateurs d'aujourd'hui et de demain

Les décennies de recherches entreprises depuis l'apparition de l'ordinateur ont profondément transformé la structure du matériel informatique. Ainsi, les technologies développées jusqu'à maintenant par les constructeurs ont mené aux architectures des calculateurs modernes. Les programmeurs doivent désormais composer avec l'ensemble de ces technologies car elles sont bien souvent rassemblées au sein d'un même calculateur.

L'émergence des technologies réseau hautes-performances a permis la démocratisation des grappes de calcul en lieu et place des supercalculateurs. Les noeuds de calcul qui les composent, tout d'abord mono-processeurs, ont évolué au niveau architectural. Les cœurs d'une même puce partagent un même accès à la mémoire (mémoire principale ou banc mémoire) à l'image des processeurs d'une architecture SMP. Plusieurs processeurs multicœurs sont reliés entre-eux par des réseaux d'interconnexion propriétaires très performants et chacun d'entre-eux dispose d'un accès privilégié à leur propre banc mémoire, marquant ainsi le retour des architectures NUMA. À côté de ces processeurs généralistes, les accélérateurs graphiques, intégrés dans la puce multicœur ou connectés au bus système comme tout périphérique, offrent une puissance de calcul impressionnante.

Toutes ces technologies imbriquées les unes dans les autres forment une topologie complexe et multiniveaux qui varie selon les plate-formes de calcul. De la considération de cette topologie vont dépendre les performances des applications. En effet, celle-ci est devenue si complexe qu'il est bien difficile d'atteindre les performances théoriques. En observant la figure 1.16, nous constatons que l'écart entre les performances théoriques et les performances effectives se creuse de plus en plus depuis ces dernières années. Et cette tendance risque vraisemblablement de s'accentuer avec les architectures des futurs processeurs hétérogènes annoncés par AMD et Intel et la forte croissance du nombre de processeurs par calculateur [Top].

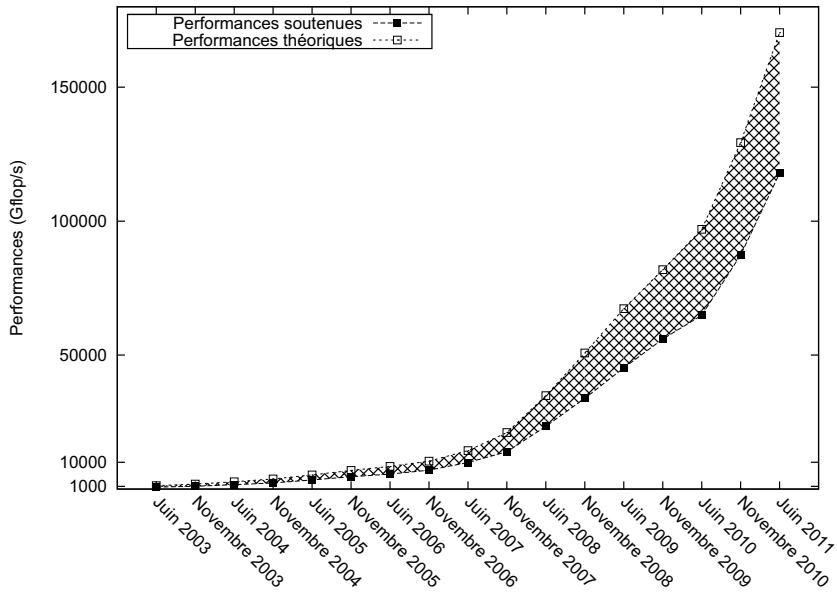


FIGURE 1.16 – Évolution de l'écart entre les moyennes des performances théoriques et des performances soutenues des calculateurs du TOP500

Les architectures caractéristiques des calculateurs modernes imposent de nouvelles contraintes au programmeur d'applications parallèles. Celui-ci doit choisir un modèle de programmation adapté qui puisse faire bénéficier à l'application des meilleures performances possibles.

Chapitre 2

Etat de l'art : comment les programme-t-on ?

Comme nous l'avons vu dans le chapitre précédent, les machines qui composent les grands calculateurs d'aujourd'hui se complexifient profondément de par leur structure hiérarchique. En outre, les processeurs de ces machines forment à une famille nombreuse et particulièrement hétérogène. Plongé dans cet univers complexe que forment les machines de calcul actuelles, le développeur d'applications doit se reposer sur un modèle de programmation adapté pour en extraire les meilleures performances. Ce chapitre répertorie les principaux modèles de programmation parallèle existants et présente des travaux menés sur ces modèles pour résoudre les problématiques causées par les architectures hiérarchiques.

2.1 Classification des modèles de programmation

Dès l'instant où la parallélisation intervient dans l'écriture d'un programme, il faut tout d'abord déterminer les blocs d'instructions qu'il est possible de rendre exécutables simultanément sur différents processeurs afin de découper les calculs complexes en fils d'exécution individuels. L'étape suivante consiste alors à choisir le modèle de programmation idoine, capable d'exprimer et d'exploiter le parallélisme de ce code. Ce choix est loin d'être évident, au vu du large panel de modèles existants. C'est pourquoi nous proposons ici de les classifier en deux groupes correspondant à leur façon de gérer la mémoire entre ces différents fils d'exécution.

2.1.1 Modèle en mémoire partagée

Les modèles de programmation par mémoire partagée se basent sur un ensemble de fils d'exécution qui évoluent de façon concurrente dans un même espace d'adressage. Les communications (lectures/écritures) sont réalisées implicitement, par l'intermédiaire de

la mémoire de la machine. De cette manière la tâche du programmeur est simplifiée en ce sens qu'il doit uniquement s'occuper du partage des tâches et potentiellement de la synchronisation en utilisant des verrous ou des sémaphores lors des accès à la mémoire partagée. Ce type de modèle est clairement adapté à des machines de type SMP, où plusieurs processeurs vont partager une mémoire physique commune. Par ailleurs avec une sous-couche logicielle ou par un dispositif matériel particulier, il est possible d'employer ce modèle en faisant apparaître la mémoire physiquement distribuée comme une seule mémoire partagée (mémoire virtuelle partagée). Cependant il est très difficile pour le programmeur d'obtenir des performances capables de concurrencer celles obtenues avec un modèle en mémoire distribuée, mieux adapté à de telles architectures.

2.1.1.1 Les bibliothèques de threads

Les bibliothèques de threads constituent une approche bas niveau, plus complexe à appréhender que des modèles de plus haut niveau mais offrant un contrôle beaucoup plus fin du parallélisme. Le programmeur dispose de primitives de création, gestion et destruction de threads pour réaliser toutes les optimisations possibles, au prix d'un important effort de développement. Nous présentons dans les paragraphes suivants les différentes catégories de bibliothèques différenciées selon le niveau d'implémentation de l'ordonnanceur de threads : dans le noyau du système, dans l'espace utilisateur ou dans les deux.

Bibliothèques de niveau noyau La mise en place de threads en espace noyau implique l'intervention du système d'exploitation. C'est son ordonnanceur qui va effectuer les opérations de gestion des threads. De cette façon, les threads peuvent être répartis sur plusieurs processeurs pour s'exécuter en concurrence. Les événements d'entrées/sorties potentiellement bloquants (par exemple une lecture sur le disque) suspendront uniquement le thread concerné, les autres pouvant être exécutés. Il s'agit de l'implémentation la plus simple. Cependant la gestion des flots parallèles est coûteuse puisque les primitives de création, d'attente et de synchronisation nécessitent des appels systèmes. D'ailleurs, les threads d'une telle implémentation sont souvent considérés comme des processus à mémoire partagée (« *Light-Weight Processes* »). En 1997, l'une des premières implémentations efficace, les *Linux Threads* [Ler99] de Xavier Leroy, offrait une primitive de création de thread équivalente à la création d'un processus à cela près que l'espace mémoire, le tableau des descripteurs de fichiers et le masque des signaux étaient partagés et non dupliqués. Plus tard, elle fut remplacée par la *Native POSIX Thread Library* [DM05] (*NPTL*), encore utilisée dans Linux.

Bibliothèques de niveau utilisateur À ce niveau, le noyau n'a pas connaissance des threads dans le processus, tout se passe en espace utilisateur. Il s'agit d'un certain nombre de threads utilisateurs fonctionnant au dessus d'un thread noyau. De ce fait, la gestion des threads est très efficace et l'ordonnanceur peut rapidement réaliser un

changement de contexte, c'est-à-dire bloquer l'exécution du thread courant et passer la main à un autre. Cela en fait un modèle de programmation flexible pouvant s'adapter aux besoins des applications. Ainsi, il est tout à fait possible de gérer la migration de threads particulièrement utile pour équilibrer la charge sans pour autant faire chuter les performances de la machine. Par ailleurs, c'est une solution portable car l'interface n'est pas dépendante du système d'exploitation sous-jacent. Toutefois, l'un des défauts majeurs de cette approche est qu'elle ne peut pas bénéficier de l'accélération matérielle offerte par une machine multiprocesseur. En effet, l'ordonnanceur ne voit que le thread noyau, dès lors deux threads utilisateurs du même processus ne pourront s'exécuter simultanément. Une conséquence fâcheuse est que lorsqu'un des threads effectue un appel système bloquant, tout le processus est bloqué : cela signifie qu'aucun autre thread ne peut prendre la main. Parmi les implémentations présentes, nous pouvons citer les *Green Threads* [Ber96] de SUN permettant d'utiliser le même ordonnanceur partout, indépendamment du système d'exploitation.

Bibliothèques mixtes Les implémentations mixtes utilisent plusieurs threads noyau qui exécutent des threads de niveau utilisateur. L'objectif est de bénéficier des avantages des deux solutions précédentes. L'ordonnanceur de niveau utilisateur dispose d'un ensemble de threads noyau sur lesquels affecter les threads utilisateur. De ce fait, l'exécution concurrente des traitements parallèles est tout à fait possible et, bien que moins rapide qu'une implémentation de niveau purement utilisateur, cette approche donne de très bonnes perfomances. Par ailleurs, le cas des appels systèmes bloquants évoqué précédemment peut s'avérer problématique si chacun des threads noyau est bloqué et qu'il reste des threads utilisateurs à ordonnancer. Heureusement, des solutions existent pour y remédier : par exemple en créant un nouveau thread noyau lors d'un appel à une primitive potentiellement bloquante. Néanmoins, les bibliothèques mixtes sont complexes à développer et à maintenir. Parmi les implémentations existantes, citons la solution d'IBM, *Next Generation POSIX Threads* [IBM], destinée à remplacer la *Linux Threads Library*, mais abandonnée plus tard au profit de la *NPTL*. La bibliothèque de threads *Marcel* de la suite logicielle PM2 [Nam97] utilise également un ordonnanceur à deux niveaux.

2.1.1.2 OpenMP

En octobre 1997, un consortium d'industriels et de constructeurs définit le standard OPENMP [Opeb] (*Open Multi Processing*), rassemblant ainsi certains efforts réalisés par le passé dans le domaine de la parallélisation basée sur des directives de compilation. L'interêt premier de ce modèle tient dans le peu d'efforts à fournir par le programmeur pour rapidement paralléliser un code écrit en C, C++ ou FORTRAN. L'ajout d'une directive OPENMP au niveau d'une boucle permet ainsi de répartir les indices de boucles entre les threads gérés par le support exécutif OPENMP. Plusieurs mots-clés placés avec les directives déterminent le type de répartition, la visibilité des variables (partagées ou privées) ou encore le nombre de threads d'une section parallèle. Certaines annotations

permettent également de synchroniser les threads ou de réaliser des exclusions mutuelles. L'exemple de la figure 2.1 présente le schéma d'exécution parallèle d'un programme OpenMP de type *Fork/Join* : le code s'exécute séquentiellement jusqu'à la rencontre d'une section parallèle ; une équipe de thread est alors générée (*Fork*) et exécute en parallèle le code imbriqué dans la section ; une fois leur travail terminé, les threads se synchronisent et terminent ou s'endorment (*Join*) jusqu'à la prochaine section parallèle.

```
int main()
{
  int var1, var2, var3, i;
  ...
  /* Code séquentiel */
  ...

#pragma omp parallel private(var1, var2) shared(var3)
  {
    /* Section parallèle exécutée par tous les threads */

    ...

  }

  ...
  /* Code séquentiel */
  ...

#pragma omp parallel for private(var1, var2) shared(var3)
  for (i = 0; i < 10000; i++)
    {
      /* Section parallèle exécutée par tous les threads
       * Les indices de boucle sont répartis entre tous les threads
       */

      ...

    }

  ...
  /* Code séquentiel */
  ...

  return 0;
}
```

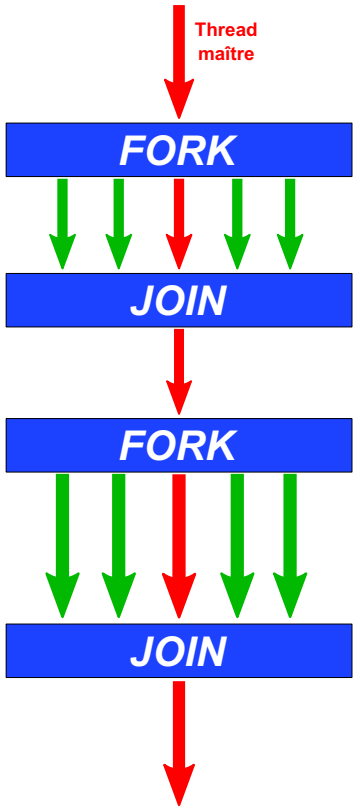


Figure 2.1 – Schéma d'éxécution d'un programme OpenMP

De plus la version 3.0 du standard [Ope08], publiée en 2008, apporte le concept de tâches, générées à l'intérieur d'une section parallèle. Chacune de ces tâches peut être exécutée par n'importe lequel des threads de la section. L'écriture de certains algorithmes s'en retrouve facilitée et s'avère plus performante [ACD+09]. C'est le cas notamment des codes récursifs générant plus de flots parallèles que de cœurs disponibles. L'utilisation des tâches permet d'éviter le surcoût de gestion de trop nombreux threads. La figure 2.2 décrit la parallélisation de l'algorithme de Fibonacci en utilisant le mécanisme des tâches OpenMP. Les deux appels récursifs peuvent être exécutés en parallèle. L'annotation `omp taskwait` permet de s'assurer que que les tâches sont finies avant de continuer l'exécution.

```
int fibonacci (int n)
{
  if (n <= 2)
    return n;
  else {
    int r1, r2;
    r1 = fibonacci(n-1);
    r2 = fibonacci(n-2);

    return r1 + r2;
  }
}
```

```
int fibonacci (int n)
{
  if (n <= 2)
    return n;
  else {
    int r1, r2;
#pragma omp task shared(r1,n)
    {
      r1 = fibonacci(n-1);
    }
    r2 = fibonacci(n-2);
#pragma omp taskwait
    return r1 + r2;
 }
}

...

#pragma omp parallel
#pragma omp single
{
  r = fibonacci(n);
}
```

FIGURE 2.2 – Parallélisation du code Fibonacci par des tâches OPENMP

C'est un modèle particulièrement adapté aux architectures de type SMP cependant le manque de contrôle sur la gestion des données bride les performances dans le cas des architectures NUMA par exemple. En effet, la norme ne définit rien pour exprimer une affinité entre threads et données, ou même pour répartir un espace mémoire sur la machine. En outre, la version 3.0, bien qu'elle étende le spectre des applications OPENMP, ne répond pas à ces problèmes de performance.

2.1.1.3 Cilk

Développée en 1994 au MIT, *Cilk* [FLR98] est une extension du langage C qui s'appuie sur un support d'exécution pour la programmation d'algorithmes parallèles. Le programmeur dispose de mot-clés spécifiques à rajouter dans le code pour en exposer le parallélisme : il définit certaines fonctions comme des tâches à exécuter en parallèle. Au démarrage du programme, le support exécutif crée quelques processus légers qui auront la charge d'exécuter les tâches *Cilk* définies par le programmeur. Chaque processus léger possède sa propre file de travaux qui contient les tâches *Cilk*. L'ajout de nouvelles tâches

se fait en tête de liste. Lorsque l'une de ces files devient vide, le processus léger peut voler une tâche depuis la queue de la file de travaux d'un autre processus léger. De cette manière, le vol de tâche ne nécessite aucune synchronisation entre les processus légers. Le coût de gestion des tâches étant très faible, *Cilk* propose un parallélisme à grain très fin efficace sur des architectures multiprocesseurs symétriques. Cependant, sur des architectures de type NUMA, les performances de *Cilk* peuvent s'avérer insatisfaisantes étant donné que le vol de travail ne tient pas compte d'une quelconque localité. De plus la parallélisation ne s'applique qu'à des algorithmes récursifs de type *Diviser pour régner*.

En 2006, le MIT vend la licence à la start-up *Cilk Arts*. Celle-ci commercialise une version de *Cilk* qui étend le modèle au langage C++ et rajoute le support pour l'exécution des boucles en parallèle, ainsi qu'un nouvel objet pour résoudre problèmes de contentions liés aux accès concurrents à des variables globales. Quelques années plus tard, Intel rachète la start-up et publie *Cilk Plus* [Int11]. Intégré à l'ensemble logiciel Intel *Building Blocks*,*Cilk Plus* simplifie le modèle en utilisant seulement trois mots-clés (`_Cilk_spawn`, `_Cilk_sync` et `_Cilk_for`) et rajoute des structures de données parallèles. La figure 2.3 présente l'exemple d'une parallélisation d'une fonction calculant la suite de Fibonacci. Les deux appels récursifs sont ainsi exécutés en parallèles et une synchronisation explicite est réalisée pour attendre leurs résultats.

```
int fibonacci (int n)
{
  if (n <= 2)
    return n;
  else {
    int r1, r2;
    r1 = fibonacci(n-1);
    r2 = fibonacci(n-2);

    return r1 + r2;
  }
}
```

```
int fibonacci (int n)
 {
   if (n <= 2)
     return n;
   else {
     int r1, r2;
     r1 = _Cilk_spawn fibonacci(n-1);
     r2 = fibonacci(n-2);
     _Cilk_sync;
     return r1 + r2;
   }
 }
```

Figure 2.3 – Parallélisation du code Fibonacci par Intel *Cilk Plus*

2.1.1.4 TBB

Le constructeur Intel a développé une bibliothèque C++ générique, dénommée *Thread Building Blocks*, [TBB, Rei07] pour écrire des programmes capables de profiter de la puissance des architectures multicœurs. L'interêt étant de simplifier autant que possible la tâche du programmeur concernant la parallélisation. En effet, à l'inverse d'une approche bas-niveau, la synchronisation, l'équilibrage de charge et l'optimisa-

tion des caches sont gérés par le support exécutif de TBB. L'utilisateur doit seulement spécifier les segments de code susceptibles de s'exécuter en parallèle. Chacun de ces segments sera alors considéré comme une tâche devant être ordonnancée sur les unités de calcul. Le programmeur manipule des objets, des itérateurs et des conteneurs C++ parallèles pour utiliser TBB : ce sont des outils usuels pour un programmeur C++. D'autre part, le support exécutif de TBB récupère les informations sur le matériel pour créer autant de threads qu'il existe de cœurs sur la machine. À l'image de Cilk, chaque thread dispose de sa propre file de tâches à exécuter et les problèmes d'équilibrage de charge sont réglés par un mécanisme de vol de travail. Le support exécutif prend soin de réaliser des vols de travails locaux (c'est-à-dire à des threads utilisant un même niveau de cache par exemple), dans la mesure du possible. Des interfaces de programmation de plus bas-niveau existent également : par exemple, il est possible de définir des priorités pour spécifier l'ordre d'exécution des tâches. De plus, TBB a été pensée pour être composable : cela signifie que plusieurs composants de calcul parallèles peuvent s'exécuter en parallèle sans nuire de façon notable aux performances. Toutefois, contrairement à OpenMP qui permet une parallélisation incrémentale, TBB nécessite des modifications majeures du code.

2.1.1.5 Modèles basés sur des machines à mémoire virtuellement partagée

L'exploitation des architectures à mémoire distribuée nécessite un travail supplémentaire de la part du programmeur en raison de la disjonction des multiples espaces d'adressages. Et la distribution des données utilisées par une application est une opération complexe qui, si elle est mal réalisée, peut mener à de graves chutes de performances. Les machines à mémoire virtuellement partagée (ou DSM :*Distributed Shared Memory*) offrent une alternative intéressante qui évite cette distribution explicite. Cela consiste à créer l'abstraction d'un unique espace d'adressage au-dessus de l'ensemble des nœuds de calcul interconnectés. Cette abstraction est mise en place et gérée soit par le système d'exploitation, soit par une bibliothèque particulière. Lorsqu'un processeur accéde une donnée qui n'est pas dans l'espace local, la ou les page(s) mémoire qui la contiennent migrent depuis la mémoire d'un nœud distant. Cette migration est assurée en interne par le système d'exploitation ou la bibliothèque, tout comme la cohérence et la consistance de la mémoire inhérentes à la gestion de multiples copies de données modifiables dans un même système.

Au-dessus de cet environnement, les programmeurs ont la possibilité d'utiliser un paradigme de programmation à mémoire partagée sans se soucier des échanges de données distants. Le passage à l'échelle (c'est-à-dire sur un réseau de machines, une grappe de calcul, etc.) d'un programme parallèle s'exécutant sur un seul nœud ne nécessite aucune réécriture du code et allège la charge du programmeur.

Intel Cluster OpenMP Conçu par Intel et commercialisé en 2006, *Cluster* OpenMP [Int06] est un système à mémoire virtuellement partagée permettant l'exécution de programmes OpenMP sur un ensemble de nœuds connectés entre-eux par un réseau.

Cluster OpenMP définit une nouvelle directive, `sharable` qui étend le langage (pour remplacer le mot-clé `shared`) et sert à identifier les variables partagés entre tous les threads dont la valeur doit être gardée cohérente dans le système entier. Pour maintenir cette cohérence mémoire, les variables *sharable* sont rassemblées sur des pages mémoire dédiées qui sont protégées par l'appel système `mprotect`. Lorsqu'une donnée *sharable* est modifiée par un processeur d'un nœud, *Cluster* OpenMP interdit l'accès en lecture et écriture aux pages correspondantes pour tous les autres nœuds. Ainsi lorsqu'un autre nœud accède à une donnée placée sur ces pages protégées, un signal SIGSGV est généré. *Cluster* OpenMP intercepte ce signal et envoie une requête de mise à jour au dernier nœud ayant modifié la page. Une fois les modifications reçues, la protection en lecture est supprimée et l'instruction qui avait généré la transmission du signal est exécutée à nouveau. Le maintien d'un tel niveau de cohérence est très coûteux : un accès mémoire distant peut être de quelques centaines à mille fois plus coûteux qu'un accès à une mémoire locale (cache, mémoire principale). En suivant un modèle de cohérence relâchée (maintenir la cohérence uniquement lors de l'utilisation de primitives de synchonisation comme les barrières ou les verrous) et en écrivant le programme de façon à maximiser la localité des données pour limiter les défauts de pages distantes, les peformances peuvent être notablement améliorées. Toutefois cela requiert un travail supplémentaire de la part du programmeur.

2.1.2 Modèle en mémoire distribuée

La seconde approche consiste à diviser la mémoire globale du programme en plusieurs blocs de données et à les affecter aux fils d'exécution, chacun se chargeant d'effectuer ses calculs sur ses propres données de façon concurrente. Le succès de ce modèle tient notamment à la démocratisation des structures distribuées en lieu et place des machines massivement parallèles. L'évolution des réseaux d'interconnexion est une des raisons majeures de ce revirement. En effet, les recherches réalisées dans ce domaine ont permis simultanément de réduire la latence et d'accroître la bande passante, et ce de façon significative. En marge de cette évolution architecturale, les demandes toujours croissantes des applications, spécialement en termes de mémoire, ont favorisé l'utilisation de ce modèle car l'ensemble des processus offre un espace d'adressage plus grand permettant de traiter des données de taille plus importantes. Deux grands types de paradigmes représentent le modèle distribué : le passage de messages pour lequel le programmeur doit gérer explicitement les mouvements de données et les langages utilisant un espace d'adressage global partagé qui offrent un plus haut niveau d'abstraction en masquant les transferts liés à des accès distants.

2.1.2.1 Le passage de message

La programmation par passage de messages entraîne la création de processus indépendants qui s'exécutent en parallèle et s'échangent des informations par envoi et réception de messages. Il est particulièrement adapté aux architectures à mémoire dis-

tribuée (les messages transitant par un réseau), mais peut tout à fait s'appliquer à celles à mémoire partagée (les messages étant transmis par une copie de mémoire à mémoire).

PVM La bibliothèque de communication Parallel Virtual Machine [SGDM94] (PVM) est issu d'un projet interne de l'Oak Ridge National Laboratory (ORNL) en 1989. Elle permet de rassembler un ensemble des machines hétérogènes d'un réseau en une unique machine virtuelle à mémoire distribuée reposant sur le passage de messages pour les communications. Des matériels à priori incompatibles, et fonctionnant sous des systèmes d'exploitation différents, peuvent ainsi communiquer. Deux parties constituent l'architecture de PVM : le processus démon `pvmd`, installé sur l'ensemble des hôtes, et une interface de programmation générique. Chacun des démons, une fois lancé, coordonne les différentes machines en gérant l'authentification des utilisateurs et les communications avec les autres processus démons PVM. L'interface de programmation offre une collection de primitives pour la création de tâches PVM, leur communication et leur synchronisation.

PVM est un modèle de programmation simple et portable qui s'est rapidement imposé auprès de la communauté scientifique. L'optimisation des performances n'étant pas l'objectif premier de PVM, le support de fonctionnalités avancées comme la dynamicité ou la gestion des appels non bloquants entraîne des surcoûts qui ont amené cette communauté à s'orienter vers des standards plus efficaces comme MPI.

MPI A l'époque où chaque vendeur de machine parallèle offrait sa propre bibliothèque de communication, un consortium d'industriels et d'équipes de recherche universitaires a décidé de mettre en commun les efforts réalisés lors des multiples projets de recherche de bibliothèques portables afin de proposer en 1993 une interface standard permettant de communiquer au dessus de tout type de réseau : le standard MPI [MPIa] est né. Quatre années plus tard, la norme MPI-2 [MPId] est venue compléter la version initiale. Ainsi la gestion dynamique des processus, le support des communications dites unilatérales (réalisant un accès direct à une mémoire distante sans intervention du processus distant) ou le support d'entrées/sorties parallèles ont été ajoutées. La prochaine version de MPI est en cours de discussion [MPIe] et apporte de sérieux changements offrant d'autres fonctionnalités telles que les communications collectives non-bloquantes, l'interface avec les outils externes (mesure de performances, débogage, etc.), les interactions avec d'autres modèles de programmation, notamment le multithreading.

Les applications MPI créent plusieurs processus qui peuvent exécuter le même programme (modèle SPMD :*Single Program Multiple Data*) ou un programme différent (modèle MPMD :*Multiple Program Multiple Data*) avec des données différentes. Au cours de l'exécution les processus s'échangent des messages suivant différents modes de communications : bloquant, non-bloquant, bufferisé, synchrone. La figure 2.4 en résume les effets. Ces modes étant composables, il est tout à fait possible de réaliser une communication non-bloquante et synchrone par exemple. Ces communications peuvent intervenir soit entre deux processus, soit entre un ensemble de processus par des méthodes

dites collectives. Dans le premier cas, les deux parties peuvent gérer les communications (bilatérale) ou une seule peut s'occuper de la transmission (unilatérale).

Modes de communications	Comportement
Envoi bloquant	La primitive se termine dès que l'emplacement mémoire utilisateur contenant les données à envoyer peut-être modifié.
Envoi non-bloquant	La primitive initialise l'opération d'émission et termine immédiatement sans se soucier du fait que les données à envoyer ont été recopiées en dehors du tampon mémoire utilisateur.
Envoi synchrone	La primitive d'envoi se termine lorsque la réception du message a commencée.
Envoi bufferisé	La terminaison de l'émission ne dépend pas de l'occurrence d'une réception correspondante.
Réception bloquante	La primitive se termine dès que l'emplacement mémoire utilisateur contient les données attendues.
Réception non-bloquante	La primitive initialise l'opération de réception et termine immédiatemment sans se soucier du fait que les données attendues aient été recopiées dans le tampon mémoire utilisateur.

FIGURE 2.4 – Liste des modes de communication du modèle MPI

Il existe de nombreuses implémentations du standard, cependant la plupart d'entre elles sont dérivées de l'une des deux implémentations libres MPICH2 [MPIf, Gro02] et OPEN MPI [OPEc, GFB+04a]. Ainsi les implémentations MPICH2-MX [Myra] (optimisée pour la technologie MYRINET), QUADRICS MPI (optimisée pour la technologie QSNET [PFH+02]), MVAPICH2 [NBCL] (optimisée pour la technologie INFINIBAND), BLUEGENE MPI [AAC+04] (optimisée pour les systèmes de type BLUEGENE) et INTEL MPI [INT] sont dérivées de l'implémentatonMPICH2. MPIBULL2 [Bul], dédiée aux serveurs de calcul BULL, est dérivée de l'implémentation OPEN MPI.

2.1.2.2 Espaces d'adressage globaux partagés

L'abstraction d'un espace d'adressage au-dessus d'un ensemble de machines interconnectées et sans mémoire commune est très attrayante pour les programmeurs d'applications qui cherchent à éviter la gestion des transferts de données explicites, propre aux modèles de programmation par passage de messages. Cependant les modèles comme OPENMP qui fonctionnent au-dessus d'une telle abstraction ne fournissent pas de mécanismes pour contrôler les placements des contextes d'exécution (processus, threads, etc.) et des données sur lesquels ils travaillent. De ce fait, les implémentations de ces modèles sur ces architectures à mémoire distribuée ne sont pas vraiment efficaces : chaque

opération de chargement ou enregistrement mémoire peut s'avérer être un transfert distant. Pour pallier ce problème, le paradigme de programmation par espaces d'adressage globaux partagés, ou PGAS (*Partitioned Global Address Space*), propose en plus de contrôler la localité mémoire. Pour ce faire, les références locales sont distinctes des références globales.

HPF *High Performance* FORTRAN [Lov93] (HPF) est un ensemble d'extensions au langage FORTRAN 90 défini par un groupe formé de plus de quarante organismes, au moment où les machines à mémoire distribuée commencaient à devenir populaires. Leur objectif était de répondre aux problèmes liés à l'écriture de programmes sensibles au placement des données en mémoire. Les directives de compilation de *High Performance* FORTRAN permettent ainsi de distribuer les données sur les processeurs, d'aligner les tableaux et d'indiquer une boucle comme devant être exécutée en parallèle. Une fois la distribution des données réalisées, chaque processeur travaille sur ses données locales. Lors d'un accès à une donnée non locale, une communication implicite est déclenchée. De même toute synchronisation est également implicite. En déléguant ainsi ces mécanismes au niveau du compilateur, HPF simplifie le travail du programmeur cependant les performances sont fortement dépendantes des optimisations réalisés par le compilateur.

Co-Array Fortran Créé dans les années 90, *Co-Array* FORTRAN [NR98] est une extension syntaxique au langage FORTRAN permettant de le convertir en un langage de programmation parallèle. Les programmeurs FORTRAN peuvent exprimer à la fois la distribution des tâches ainsi que la distribution des données. Un programme est ainsi répliqué un nombre fixe de fois, chaque réplique, également nommée image, ayant son propre jeu de données. Les programmeurs ont la possibilité de déclarer certaines variables comme étant accessibles aux autres images. L'accès (en lecture ou en écriture) distant étant réalisé de façon syntaxique en précisant l'image contenant la donnée visée. Cela simplifie les communications en ce sens qu'il n'y a pas d'envoi ou réception de message, cependant l'accès à une donnée distante reste explicite (identique aux communications unilatérale du standard MPI). L'extension supporte également les entrées/sorties en parallèle, optimisant les accès aux fichiers par l'ensemble des images. Enfin des primitives de synchronisation sont fournies : barrières globales ou limitées à un sous-ensemble d'images, sections critiques et mises à jour de variables partagées en mémoire globale. Cette extension de langage est maintenant intégré dans la norme Fortran 2008. Cependant *Co-Array* FORTRAN souffre de quelques faiblesses qui affectent les performances : par exemple, les synchronisations concernant les variables partagées interviennent pour l'ensemble des processus même si cela ne concerne qu'un sous-ensemble de ceux-ci. De plus, la spécification prescrit l'utilisation de synchronisations avant et après chaque appel de procédure, empêchant tout recouvrement de communication par du calcul. Pour y remédier, des chercheurs de l'université Rice proposent une implémentation du paradigme [JMCA+11] avec quelques fonctionnalités supplémentaires : des sous-ensembles de processus, une synchronisation plus fine, etc.

UPC *Unified Parallel C* (UPC) [CDC+99, UPC05] est une extension du langage C développée par un consortium et dont la première version a été publiée en 2001. L'extension s'appuie sur un unique espace d'adressage partitionné au sein duquel s'exécute un ensemble de threads. Chaque partition est associée à un thread particulier et le programmeur utilise les déclarations UPC pour répartir les données parmi les threads. Un thread et ses données associées sont alors placés sur le même nœud physique. Par défaut, les variables sont privées. Lorsque le programmeur les déclarent comme étant partagées, les variables scalaires sont stockées sur le thread 0 et les tableaux sont stockés selon la politique de répartition de type *round-robin*. Il est possible de définir une répartition par blocs cycliques. UPC propose également des primitives de synchronisation et de gestion mémoire (`upc_memcopy`, `upc_memget` et `upc_memput`). La cohérence mémoire des variables partagées est paramétrable : stricte qui synchronise la mémoire pour chaque accès et relâchée qui ne le fait qu'aux points de synchronisation du programme (barrières, etc). La distribution du calcul est réalisée par rapport aux informations d'affinités des données exprimées par le programmeur. Par exemple, dans le programme de la figure 2.5, l'indice `j` de la boucle sera traité par le thread associé à la variable partagée `t3[j]`. Les implémentations les plus actives de UPC sont *Berkeley UPC* [BER] et *GNU UPC* [GNU]. Il est important de noter que les accès distants impliquent des transferts de données entre processus. Pour optimiser l'exécution de l'application , il est nécessaire d'aggréger autant que possible les données à transférer en utilisant les primitives de gestion mémoire. Par ailleurs, UPC ne permet pas la définition de sous-ensemble de threads, qui pourraient être utilisés pour le partage de tâches au sein d'un nœud multiprocesseur et/ou multicœur.

```
#define N 1000 * THREADS

shared int t1[N], t2[N], t3[N];

int main() {
  int i;
  upc_forall (i = 0; i < N; i++; &t3[i])
    t3[i] = t1[i] + t2[2];
}
```

FIGURE 2.5 – Exemple d'utilisation de l'extension de langage UPC

XcalableMP Le langage XCALABLEMP [XCA] utilise des annotations de compilation (à la façon d'OPENMP) pour définir un parallélisme de tâches et une distribution des données. Apparu en fin d'année 2010, il s'agit d'un modèle très récent fortement influencé par les modèles HPF et CO-ARRAY FORTRAN. Un code C, C++ ou FORTRAN augmenté de ces annotations peut s'exécuter sur un ensemble de nœuds interconnectés par un réseau suivant un modèle de type SPMD. Sur chaque nœud, un programme doté

d'un seul thread démarre et s'exécute. Par défaut, les données déclarées sont allouées dans chaque nœud et référencées localement par les threads du nœud. Les directives permettent de distribuer des données spécifiques et de partager le travail. Un exemple de code utilisant XcalableMP est donné en figure 2.6. La première directive déclare la variable `p` comme un ensemble de quatre nœuds parmi l'ensemble des nœuds disponibles. Les deux suivantes créent une distribution d'un tableau fictif de taille `YMAX` sur les quatre nœuds. La quatrième répartit le tableau global `array` entre les espaces mémoire de ceux-ci. La répartition des indices de boucle est réalisée par la dernière annotation XcalableMP. D'autres politiques de répartition sont envisageables suivant les besoins applicatifs (une recopie dans tous les espaces mémoire locaux par exemple). En outre, l'extension permet également au programmeur d'avoir une vision locale des données, à la manière de Co-Array Fortran. Un processus peut ainsi accéder explicitement aux données d'un autre processus distant.

```
int array[XMAX][YMAX];

#pragma xmp nodes p(4)
#pragma xmp template t(YMAX)
#pragma xmp distribute t(block) on p
#pragma xmp align array[i][*] with t(i)

int main() {
  int i, j, res;
  res = 0;

#pragma xmp loop on t(i) reduction(+:res)
  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++) {
      array[i][j] = i*j;
      res += array[i][j];
    }
}
```

Figure 2.6 – Exemple d'utilisation de l'extension de langage XcalableMP

Il est important de noter que toute action de communication ou de synchronisation entre les nœuds se fait par les directives, ce qui permet au développeur d'applications une meilleur compréhension du comportement du programme afin d'en améliorer les performances. Cette extension de langage s'avère prometteuse cependant elle est encore jeune. Ainsi, la question de l'exploitation des multiples cœurs d'un même nœud n'est pas clairement définie : pour le moment, la solution envisagée est une interaction avec un autre modèle de programmation tel OpenMP.

Les langages de programmation PGAS offrent une réponse au besoins des applications quant à la distinction des accès à la mémoire distante et des accès à la mémoire locale. Ils permettent au programmeur de définir une notion d'affinité entre les données partagées et les contextes d'exécution (processus, threads). Cependant, ces langages souffrent de quelques défauts, notamment concernant l'exploitation des processeurs multicœurs. En effet, le programmeur ne dispose pas de mécanismes suffisamment précis pour exprimer l'affinité au sein d'un même noeud en considérant leur structure hiérarchique et complexe. Dans l'attente d'une amélioration de ce côté-là, il est nécessaire d'explorer d'autres solutions, plus efficaces.

2.2 Le modèle hybride

À l'ère des réseaux de machines uni-processeurs, le modèle privilégié, principalement de par ses performances, était le passage de messages (MPI, PVM). Peu à peu, les architectures multiprocesseurs se sont démocratisées et ont intégré les grappes de calcul, apportant par là-même un degré de hiérarchie supplémentaire. Les utilisateurs ont tout d'abord continué d'employer les modèles de programmation par échange de messages afin de bénéficier des performances d'un ensemble de nœuds (et ne pas se limiter à un seul). Dans un second temps, certains se sont interrogés sur les surcoûts engendrés par une telle solution, notamment les transmissions « réseau » à l'intérieur d'un même nœud. Éprouvés par le passé, les paradigmes de programmation en mémoire partagée sont apparus plus appropriés pour les architectures de type SMP. Ainsi, l'idée de réunir deux modèles de programmation, chacun étant adapté à une structure matérielle particulière, est venue tout naturellement.

2.2.1 Les limites des modèles actuels

L'emploi d'un unique modèle de programmation est beaucoup plus simple du point de vue du programmeur. En utilisant le modèle MPI par exemple, celui-ci peut s'appuyer sur un standard éprouvé depuis de nombreuses années, dont les grandes implémentations font l'objet d'améliorations régulières, notamment en contexte multicœur lors de ces dernières années. Cependant les évolutions architecturales des systèmes, comme la croissance du nombre d'unités de calcul, font que ce modèle rencontre des difficultés à garantir de bonnes performances. Les autres paradigmes de programmation montrent également leurs limites face aux machines de calcul modernes.

2.2.1.1 Difficultés de passage à l'échelle

Une des grandes difficultés pour un programmeur d'applications utilisant le paradigme du passage de messages, tel MPI, réside dans la répartition des données parmi l'ensemble des processus. La situation que l'on cherche le plus souvent à éviter est un important déséquilibre de la charge : certains processus finiront plus tôt que d'autres et

deviendront inactifs alors qu'ils pourraient exécuter une partie du travail des autres. Les applications irrégulières utilisant un raffinement de maillage dynamique illustre parfaitement ce cas : les processus travaillent chacun sur une zone particulière du domaine de calcul total, toutefois chacune d'entre-elles contient une quantité de travail différente. Le rééquilibrage nécessite alors la mise en place de mécanismes comme le vol de travail [DOS+07, RLP11] qui impliquent des migrations dynamiques de données entre processus, particulièrement coûteuses en temps. Et plus la machine cible dispose de nombreuses unités de calcul, plus le déséquilibre est prononcé et requiert de multiples migrations de données.

Par ailleurs, le standard MPI fournit tout un panel de primitives de communication dites *collectives* (c'est-à-dire qui impliquent l'ensemble des processus) fréquemment utilisées dans les applications, notamment dans leur version irrégulières qui permettent à l'utilisateur de transmettre des données de taille spécifique pour chaque processus impliqué. Ces primitives prennent un ou plusieurs arguments correspondant à des tableaux de taille équivalente au nombre de processus. Ainsi, pour un million de processus chaque tableau occupera un espace mémoire de 4 Mo sur chaque processus. Il faut toutefois noter que la plupart du temps, ces communications collectives irrégulières ne servent à envoyer des données qu'à un sous-ensemble des processus applicatifs (par exemple dans le cas d'une décomposition de domaine seuls les processus voisins communiquent entre-eux) : l'utilisateur spécifie une taille de valeur nulle pour les processus non concernés afin d'éviter d'envoyer des messages inutiles sur le réseau. L'implémentation MPI doit tout de même parcourir le tableau dans sa totalité pour identifier les processus auxquels transmettre réellement des données. Le coût de ce parcours dépendant du nombre de processus de l'application, sur des systèmes à grande dimension l'impact est plus important comme en témoigne les mesures effectuées par P. Balaji et al. [BBG+09]

Un autre paramètre à prendre en compte pour les applications parallèles est leur consommation mémoire. En effet, lorsque la taille des données allouées par les processus dépasse les capacités de la mémoire principale, certaines de ces données sont placées sur un espace mémoire particulier du disque (la mémoire *swap*). L'accès au disque étant beaucoup plus coûteux que l'accès à la mémoire principale, les développeurs optimisent leur code soit en réduisant les allocations mémoire pour rester dans les limites de la mémoire principale (algorithmes *in-core*), soit en contrôlant les entrées/sorties pour en réduire l'impact sur les performances (algorithmes *out-of-core*). En plus des allocations mémoire de l'application, une implémentation du standard MPI utilise différentes structures de données nécessaires à son bon fonctionnement : informations sur l'état du processus, tableau de correspondance entre identifiants MPI et identifiants de processeurs, zone mémoire pour la réception de messages inattendus [1], etc. L'occupation mémoire de certaines de ces structures dépend directement du nombre de processus MPI impliqués. D. Goodell et al. ont ainsi examiné l'utilisation mémoire du logiciel MPICH2 et ont réalisé plusieurs optimisations pour réduire cette consommation [GGZT11]. La tendance est la même pour les messages inattendus. Pour limiter l'impact sur la mémoire, une

1. Lorsque l'envoi intervient avant que la réception correspondante ne soit postée, les données doivent être stockées temporairement du côté récepteur par l'implémentation MPI

solution peut être de mettre en place d'un système de contrôle de flux comme cela a été fait pour les systèmes IBM Blue Gene [FCLA06].

La problématique du passage à l'échelle se pose également pour le modèle en mémoire partagé OpenMP, pour lequel le nombre de défauts de cache mémoire peut augmenter très rapidement avec l'accroissement du nombre de threads. En effet, étant donné que le standard ne fournit pas de mécanismes pour exprimer les affinités entre threads, si plusieurs threads placés sur des cœurs distants (ne partageant pas de mémoire cache) accédent à une donnée sur une même ligne de cache alors il est plus que probable qu'ils s'échangeront régulièrement le contenu de cette ligne de cache au travers des caches.

2.2.1.2 Modèles à mémoire partagée sur architectures distribuées

L'abstraction du système de calcul fournie par les modèles basée sur une mémoire partagée est un réel avantage par rapport aux modèles à mémoire distribuée. Effectivement, le programmeur est soulagé du travail de distribution des données et de la gestion des communications explicites nécessaire pour partager des données entre les processus [Sod05]. Toutefois cette abstraction, bien utile pour le programmeur, s'avère dommageable pour les performances des applications sur des systèmes de calcul dont l'architecture est fortement hiérarchique. L'absence d'information sur la structure matérielle sous-jacente et sur son organisation fait que l'on ne peut s'assurer de rassembler les threads travaillant sur les mêmes données ou même de placer les threads près des emplacements mémoire où sont stockées les données auxquelles ils accèdent. De ce fait, sur une grappe de calcul, tout accès mémoire ou synchronisation entre les threads est susceptible de générer un transfert par le réseau, plus coûteux qu'un accès à une zone mémoire locale. Qui plus est, la gestion d'une unique mémoire partagée implique le maintien d'une cohérence mémoire (plus ou moins forte) qui génère un important trafic : invalidation, migration, etc. Et ce trafic affecte directement l'exécution de l'application.

2.2.1.3 D'autres modèles non standards

Les modèles de programmation à espaces d'adressage globaux partagés se présentent comme une alternative intéressante aux autres paradigmes. En effet, ils ont l'avantage de permettre la mise en place d'une distribution de données et de distinguer les accès locaux et les accès distants. À la différence des modèles par passage de messages, l'accès à une donnée distante est relativement similaire à l'accès à une donnée locale utilisée usuellement par les langages standards (C, Fortran, etc.) sans être pour autant totalement transparent. Prenons un exemple : dans Co-Array Fortran les accès distants se font en spécifiant le numéro de l'image sur laquelle est la donnée désirée. L'apprentissage de quelques nouvelles règles et annotations suffit au programmeur Fortran pour utiliser cette extension syntaxique.

Aussi prometteurs que ces modèles de programmation semblent être, ils souffrent de certaines limitations. La plus notable d'entre-elles concerne le schéma de distribution

des données et des threads. Lors des accès distants, rien ne permet de distinguer les données placées sur le noeud local de celles situées sur un autre noeud. De plus, les synchronisations s'effectuent de façon globale, c'est-à-dire qu'elles nécessitent l'intervention de l'ensemble des processus de l'application. Une distinction plus fine, adaptée à la structure des grappes de calcul actuelles fait défaut. En effet, la création de sous-ensembles de processus travaillant sur des données similaires ou proches permettrait de les faire correspondre aux groupes de processeurs ou de cœurs d'un même noeud de calcul et par là-même de rendre l'exécution de l'application plus efficace. Des travaux dans ce sens ont déjà été réalisés pour améliorer le langage Co-Array Fortran. Pour d'autres langages comme XcalableMP, l'exploitation des architectures multiprocesseurs et multicœurs reste encore à l'étude. Parmi les solutions envisagées, la plus courante consiste à utiliser un autre modèle de programmation, qui assiterait le paradigme PGAS. En général, les modèles conseillés sont les modèles de programmation à mémoire partagée OpenMP ou Pthread.

2.2.2 Pourquoi mélanger deux modèles de programmation ?

Les modèles de programmation existants, notamment les standards MPI et OpenMP font l'objet de réflexions régulières qui visent à repenser leur interface et la façon de les implémenter. Ces réflexions s'appuient sur des besoins exprimés par les utilisateurs, des retours d'expériences de chercheurs du domaine et sont souvent liées aux progrès matériels des plates-formes de calcul. Toutefois les évolutions d'un paradigme de programmation ne répondent pas toujours à toutes les attentes. Les développeurs d'applications ont constaté l'écart entre les performances théoriques des systèmes de calcul modernes et celles obtenues en utilisant un unique modèle de programmation. Afin de réduire cet écart la conjugaison de deux modèles a été envisagée. Au travers de cette méthode, on cherche à offrir une combinaison du parallélisme à gros grain du paradigme distribué comme le passage de messages (par exemple, pour une décomposition de domaine) avec le parallélisme à grain fin offert par une approche partagée. Le premier permet de paralléliser au dessus de l'ensemble des nœuds et le second à l'intérieur de chaque nœud.

2.2.2.1 Bénéficier des avantages des deux types de modèles

Le mélange de deux modèles de programmation a pour objectif de tirer profit des performances de chaque modèle et de remédier aux problèmes rencontrés sur certaines architectures ou dans des conditions particulières. Ainsi, l'idée est de pallier les limites du passage à l'échelle des implémentations du standard MPI face à un trop grand nombre de processus sur des nœuds composés d'un grand nombre de cœurs. En utilisant un modèle mixte, le nombre de processus MPI diminue jusqu'à une borne inférieure correspondant au nombre de nœuds de calcul (un processus par nœud). La tendance actuelle étant à une multiplication des cœurs dans un même nœud de calcul, cette programmation *hybride*

réduit le trafic réseau, particulièrement important lors des communications collectives, et par là-même diminue les risques de contention.

Par ailleurs, l'utilisation d'un modèle de programmation par mémoire partagée au-dessus d'une grappe de calcul nécessite de mettre en place une abstraction mémoire juste au-dessus du matériel. Le mélange de deux paradigmes de programmation se substitue à cette solution et crée une hiérarchie à deux niveaux (processus et threads) que l'on peut faire correspondre à l'architecture des systèmes de calcul. Les accès distants sont donc réalisés explicitement (par l'appel à une primitive de communication) et donc distincts des accès à la mémoire locale. À partir de là, les programmeurs peuvent optimiser leur code en considérant ce facteur de distance. Ainsi, il est courant de *recouvrir* les communications réalisées par un accès à des informations d'un ou plusieurs autres processus par des calculs multithreadés opérant des accès locaux.

2.2.2.2 Gérer finement l'empreinte mémoire

La gestion de l'espace mémoire est une des grandes difficultés lors de l'écriture d'un programme. Cet espace étant limité, les développeurs d'applications parallèles tentent d'en optimiser autant que possible l'occupation. Des outils sont proposés pour réduire l'impact mémoire des programmes : par exemple, la bibliothèque d'allocation mémoire *SBLLmalloc* [BdSS+11] détecte automatiquement les blocs mémoire identiques et les rassemble en une seule copie placée en mémoire partagée. Les synchronisations pour les accès concurrents sont réalisés grâce à des verrous de la norme POSIX. L'emploi d'un modèle hybride de programmation tel MPI + OpenMP permet d'arriver au même résultat sans nécessiter de structures de données supplémentaires ou de mise en place de mécanismes d'interception. Ainsi en générant un processus MPI par nœud de calcul multicœur, les données d'un même nœud sont partagées entre les threads OpenMP plutôt que d'être dupliquées dans les espaces d'adressage de plusieurs processus.

Une catégorie d'opérations dénommées *stencil* illustre très bien cette situation. Celles-ci consistent un ensemble de boucles imbriquées parcourant des grilles de points en réalisant en chaque point un calcul de voisinage. Avec un modèle MPI, chaque processus travaille sur une partie des grilles et doit régulièrement communiquer avec les processus modifiant les zones voisines de la sienne. Une optimisation courante consiste à ne communiquer avec son voisin qu'au bout d'un certain nombre d'itérations. Cela implique de recopier les valeurs d'un certain nombre [2] de cases entourant la zone de travail du processus : on les appelle aussi les *cellules fantômes* (*ghost cells*). La figure 2.7 présente un exemple de découpage. En utilisant un modèle de programmation hybride, le nombre de zones de calcul est réduit, le découpage étant fait à un plus gros grain, et l'espace mémoire total alloué pour les cellules fantômes est plus petit.

D'autre part, la réduction du nombre de processus par nœud qu'implique la combinaison de deux modèles de programmation réduit fortement la quantité de mémoire

2. En fait, cela correspond au nombre d'itérations avant d'échanger des données avec les autres processus, et ce dans chacune des directions

allouée pour les structures de données utilisées par l'implémentation MPI. Les communications collectives irrégulières sont alors plus efficaces, comme certains des mécanismes internes aux implémentations du standard.

2.2.2.3 Equilibrer la charge de calcul

Lorsque l'on considère un programme parallélisé avec le standard MPI, la répartition des données est un point critique. C'est un travail difficile pour des algorithmes de nature irrégulière et une mauvaise répartition peut gravement altérer les performances globales de l'application. Dans ces cas, le mélange de deux modèles de programmation peut soulager, dans une certaine mesure, ce travail qui incombe au programmeur. En effet, le partage des tâches entre les processus du modèle distribué est réalisé à un grain plus grossier et minimise l'impact d'un déséquilibre de charge entre les processeurs. Chaque processus divise son travail en plusieurs tâches qui sont distribuées aux threads du modèle en mémoire partagée. De cette façon, si certains threads terminent la tâche qui leur a été assignée plus rapidement que d'autres, des mécanismes de rééquilibrage de charge peuvent intervenir pour fournir une partie du travail total restant aux threads inactifs : redistribution de l'ensemble des tâches non terminées à tous les threads, vol de travail entre threads, etc. Ces mécanismes sont beaucoup moins coûteux dans le cadre d'un modèle en mémoire partagée étant donné que les données à récupérer sont situées dans le même espace d'adressage. Par exemple, pour une boucle de calcul exécutée en parallèle par plusieurs threads, un vol de travail pourrait consister à récupérer un certain nombre d'indices de boucle. Bien entendu, la dimension d'un noeud de calcul défini la limite de ce type de rééquilibrage. Ainsi, si tous les threads d'un processus MPI terminent leur tâche avant les autres, le rééquilibrage de charge doit intervenir entre processus et l'on revient à des mécanismes de migration de données dont l'impact sur le temps d'exécution global est plus important.

2.3 Discussion

Plusieurs raisons expliquent l'engouement général pour les modèles de programmation hybrides. Certains cherchent à exécuter leur application utilisant un modèle à mémoire partagée sur des systèmes de dimension plus importante. D'autres éprouvent les limites des modèles de programmation à mémoire distribuée et cherchent un moyen de se rapprocher des performances théoriques des machines de calcul modernes. Or, la combinaison de ces deux modèles nécessite un effort de réflexion et un travail particulier pour être efficace.

2.3.1 Les efforts des standards

L'association de deux modèles de programmation ne concerne pas seulement les programmeurs d'applications parallèles. En effet, c'est également une des préoccupations

des développeurs d'implémentations des modèles. Ainsi le forum MPI [MPIb], qui référence toutes les étapes du processus de standardisation du standard (spécifications des différentes versions de la norme, compte-rendus des réunions, etc.), affiche une section dédiée à l'intégration de mécanismes pour faciliter la programmation hybride [MPIc]. Ils cherchent ainsi à améliorer le standard MPI pour supporter plus efficacement les modèles tels OpenMP, Pthread, TBB, OpenCL, CUDA, UPC ou encore Co-Array Fortran.

2.3.1.1 Les niveaux de multithreading

Il est important de noter que les spécifications des normes MPI 2 et MPI 3 ne définissent pas le support des threads par une implémentation MPI comme étant obligatoire. De plus, l'utilisation des threads et des appels aux primitives de communication varie selon le degré de multithreading géré par l'implémentation. À l'initialisation de l'environnement MPI, l'utilisateur peut connaître le niveau de multithreading pour savoir comment coupler l'exécution des threads avec les appels communicants.

MPI_THREAD_SINGLE	Un seul thread peut s'exécuter au sein du processus.
MPI_THREAD_FUNNELED	Le processus peut être multithreadé mais l'utilisateur doit s'assurer que seul le thread principal effectue les appels aux méthodes MPI.
MPI_THREAD_SERIALIZED	Le processus peut être multithreadé et plusieurs threads peuvent faire appel aux méthodes MPI mais un seul à la fois (c'est-à-dire pas de façon concurrente avec les autres threads du processus).
MPI_THREAD_MULTIPLE	Le processus peut être multithreadé et plusieurs threads peuvent appeler les primitives MPI, sans aucune restriction.

De nombreuses implémentations du standard MPI offrent désormais un support du multithreading au plus haut-niveau (MPI_THREAD_MULTIPLE). Cependant c'est une tâche complexe qui requiert de faire des choix pour suivre les spécifications de la norme et pour conserver une exécution correcte (au sens de la validité du résultat) souvent aux dépens des performances. Ainsi la norme précise que dans une implémentation supportant le multithreading, tous les appels aux méthodes MPI sont réentrants. Cela signifie que deux threads qui s'exécutent peuvent appeler des primitives MPI de façon concurrente et le résultat sera équivalent à un ordre d'exécution séquentielle des appels. D'autre part, dans le cas des appels potentiellement bloquants, seul le thread

appelant sera bloqué, permettant à un autre thread de s'exécuter. Cela nécessite des mécanismes supplémentaires à intégrer à l'implémentation. Les recherches à ce sujet sont toujours d'actualité [BBG+10, TG09].

Les implémentations MPICH2 et MVAPICH2 supportent le multithreading, et ce assez efficacement. C'est également le cas pour OpenMPI, toutefois ce dernier souffre de quelques limitations (pas de support du multithreading pour certains réseaux). MiMPI (*Multithread Implementation of MPI*) [GCC99], TOMPI (*Thread-Only MPI*) [Dem97] et TMPI (*Thread MPI*) [TY01] sont des implémentations MPI qui utilise des threads comme support pour les rangs MPI. Ce type d'implémentation ne supporte pas vraiment le multithreading selon la norme au sens où les threads d'un même processus MPI ne doivent pas être adressables séparément.

2.3.1.2 Interopérabilité de MPI avec les threads

Dans les derniers documents du groupe de travail MPI concernant la programmation par mélange de modèles, une proposition concerne la mise en place d'un mécanisme de création d'équipes de threads. Ces équipes de threads représentent des groupes logiques de threads définis à partir de l'ensemble des threads s'exécutant dans l'application. Ainsi l'application donne temporairement le contrôle de tout ou une partie des threads dont elle dispose à l'implémentation MPI. Cette dernière peut alors répartir les différentes opérations dont elle a la charge entre les threads.

Cette fonctionnalité, encore au stade de la discussion, fournit un début de réponse aux problèmes de composabilité de programmes parallèles que se posent les utilisateurs et les développeurs d'implémentations à l'heure actuelle. En effet, prenons l'exemple d'une application utilisant le langage OpenMP et appelant des fonctions d'une bibliothèque elle-même parallélisée à l'aide de threads (comme la bibliothèque MPI ou GoToBLAS). Il est indispensable de dimensionner les threads affectés à l'application et ceux affectés à la bibliothèque afin d'éviter qu'ils rentrent en compétition pour l'accès à l'ensemble des processeurs du système de calcul.

2.3.2 Un travail de composition

Les paradigmes de programmation dont dispose l'utilisateur pour écrire le code d'une application parallèle sont nombreux et variés. Chacun présente ses avantages et inconvénients selon différents critères comme la facilité d'apprentissage et de développement, ou encore les performances obtenues. Parmi eux, les modèles MPI et OpenMP connaissent un large succès depuis déjà quelques années et évoluent régulièrement pour rester efficaces. Toutefois, les architectures des systèmes de calcul modernes ont mis ces standards à rude épreuve et bien que leurs performances soient encore bonnes, l'écart entre les performances théoriques et les performances effectives se creuse. Nous avons constaté que ces modèles ne sont plus vraiment adaptés aux architectures hiérarchiques actuelles (grappes de calcul multicœurs, etc).

Le mélange de deux modèles de programmation est une solution couramment envisagée pour se rapprocher de la puissance théorique des machines. Apparu au temps où les réseaux de machines multiprocesseurs symétriques dominaient le marché, la programmation hybride consistait à utiliser le modèle à mémoire distribuée pour les interactions inter-nœuds et celui à mémoire partagée au sein de chaque nœud. Cependant, les architectures ne sont plus aussi symétriques comme nous l'avons vu dans le premier chapitre. Des affinités existent entre certains cœurs des processeurs et une hiérarchie complexe décrit la structure des sytèmes de calcul modernes. Il est necéssaire de récolter des informations à ce sujet et de réfléchir à une combinaison de modèles plus fine, adaptée à la machine sous-jacente.

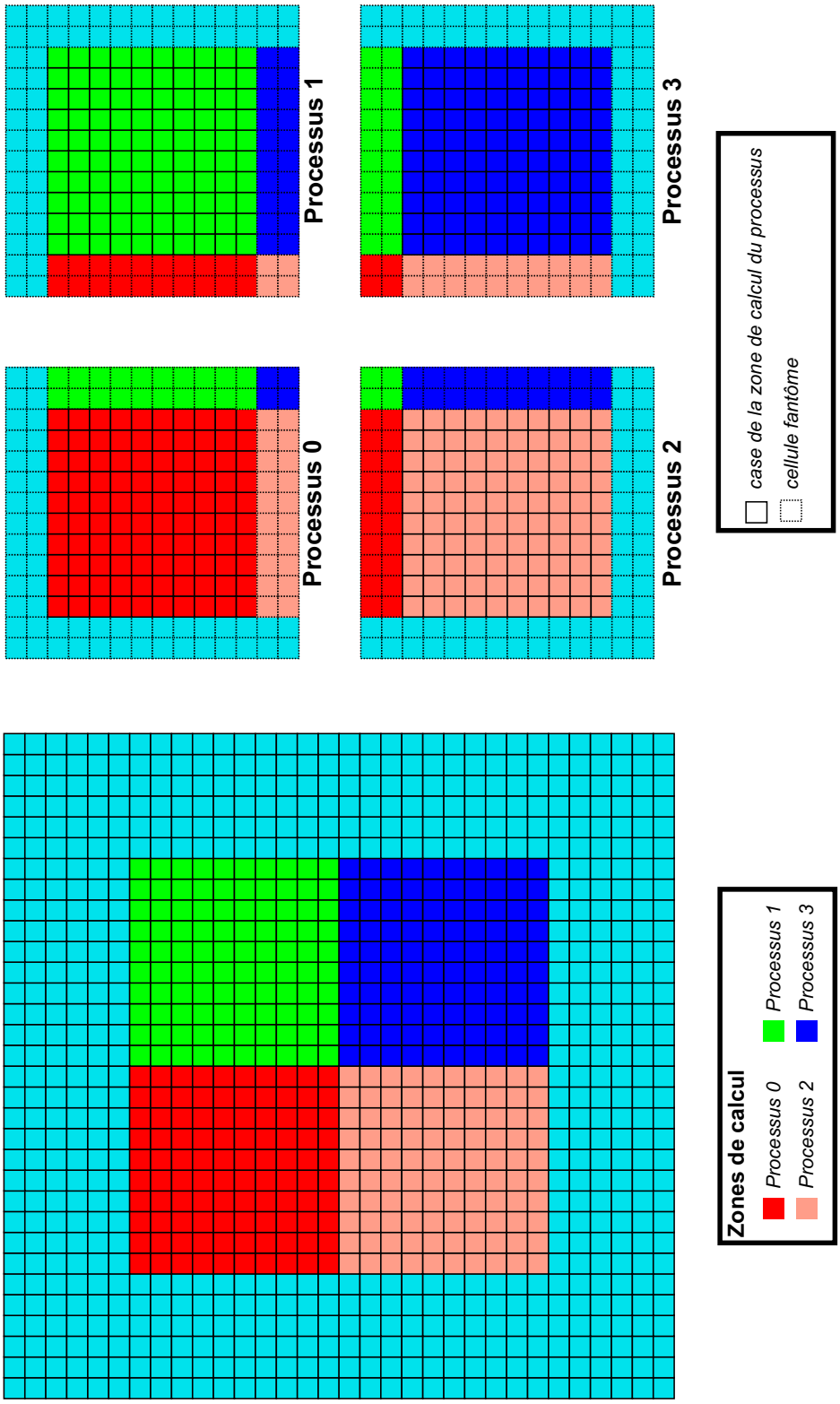


FIGURE 2.7 – Exemple de partage de grilles de points entre processus MPI

Chapitre 3

Contribution

Nous proposons dans ce chapitre un environnement propre à assister le développeur d'applications parallèles dans l'utilisation d'un modèle hybride de programmation. Notre objectif est d'étudier l'architecture cible pour optimiser l'exécution de l'application en contrôlant le parallèlisme offert par chacun des deux modèles. Nous nous appuyons également sur des outils de visualisation pour étudier les performances obtenues.

3.1 Un environnement pour la programmation hybride

Notre proposition consiste en un ensemble d'outils et de techniques de programmation sur lesquels peuvent s'appuyer les utilisateurs et développeurs d'applications parallèles désireux d'exploiter au mieux les capacités matérielles des machines de calcul actuelles. Dans cette section nous présentons schématiquement les trois aspects qui constituent notre approche.

3.1.1 Récolter les informations sur l'architecture du matériel

Avant toute chose, si l'on revient sur les raisons qui ont poussé les développeurs d'applications parallèles à mélanger deux types de modèles de programmation, la complexité architecturale des machines arrive en tête. La compréhension de l'agencement des processeurs et de la mémoire, la position des cartes de communication, la taille et les différents niveaux de caches, sont autant d'informations essentielles pour améliorer les performances des applications. En respectant l'affinité entre les multiples fils d'exécution d'un programme, les communications se font plus efficacement. Par exemple, si deux threads accèdent à une même donnée, il sera bien plus intéressant qu'ils la partagent sur une même mémoire cache (L2 ou L3) plutôt que de passer par la mémoire principale ou même de transiter par un réseau d'interconnexion. Un autre cas est celui où certains threads ont besoin d'accéder intensivement à la mémoire et au même instant : les placer sur une puce séparée, chacun disposant d'un lien avec la mémoire et profitant d'une

bande passante maximale, se révèle être une bonne stratégie. Par ailleurs, il est également important de connaître la taille des mémoires caches, afin d'optimiser les accès effectués par l'application et tâcher de minimiser les défauts d'accès à ces caches [1]. En résumé, plus nous disposons d'informations concernant le matériel, plus il sera possible d'optimiser l'exécution de l'application pour une architecture particulière.

3.1.2 Contrôler le parallélisme des deux modèles

Une fois la structure interne des machines de calcul connue, il faut utiliser toutes les informations qui ont pu être récoltées pour établir un couplage des modèles de programmation en adéquation avec l'architecture sous-jacente. En effet, mélanger deux modèles de programmation est loin d'être simple.

En premier lieu il s'agit de dimensionner correctement les ressources d'exécution mises en jeu par chacun des deux paradigmes par rapport aux traitements applicatifs à réaliser et aux unités de calcul présentes. L'intuition première serait d'employer d'un côté une programmation par mémoire partagée sur les groupes d'unités de calcul ayant accès à un module mémoire commun (par exemple l'ensemble des processeurs d'un ordinateur de type SMP qui ont un accès uniforme à une même mémoire) et d'opter pour le modèle distribué pour faire communiquer ces différents groupes. Par exemple, en considérant un modèle de programmation mixte MPI + OpenMP, on créera autant de processus MPI qu'il y a de nœuds de calcul. Au sein de chacun de ces processus MPI, un ensemble de threads OpenMP va se répartir le travail du processus et s'exécuter sur les différents processeurs du nœud. Les échanges d'informations se font entre les nœuds par transfert de messages sur le réseau et par variables partagées en mémoire entre les processeurs d'un même nœud. Toutefois, cette répartition n'est pas obligatoirement gage de performances optimales. En effet, certaines situations peuvent se révéler problématiques : par exemple, lorsque le nombre de cœurs dans une machine devient relativement élevé. Dans ce cas, il n'est pas rare de constater de graves chutes de performances lorsque de trop nombreux threads s'exécutent dans un même espace d'adressage. Par ailleurs, les architectures devenant de plus en plus hiérarchiques, les communications inter-cœurs sont potentiellement plus coûteuses lorsque ces derniers sont éloignés physiquement (sur des puces différentes, sur des bancs NUMA différents, etc). L'impact est d'autant plus important pour des applications dont les performances dépendent directement de la latence et du débit mémoire (applications de type *memory-bound*).

En second lieu, le contrôle du niveau de parallélisme permet de réguler et de maîtriser les échanges effectués dans le cadre du modèle distribué. Ainsi, le modèle par passage de messages illustre parfaitement la situation : en limitant le nombre de processus MPI qui s'exécutent sur le calculateur, nous limitons également le nombre de messages qui vont transiter sur le réseau. De cette manière, il est possible de minimiser,

1. Un défaut de cache (ou *cache miss*) se produit lorsque le processeur demande une donnée qui n'est pas stockée en mémoire cache ou qui est invalide. Une requête est alors effectuée pour rapporter une copie de ces données depuis la mémoire principale.

voire d'éviter d'éventuelles contentions réseau, et donc d'améliorer les temps d'exécution des applications. Ceci va notamment dépendre de l'application et de son schéma de communication : les communications MPI pouvant intervenir uniquement au début et à la fin, ou bien tout au long du calcul, ou encore de façon irrégulière. La taille des messages peut également varier selon l'application. Tous ces paramètres font qu'il n'est pas aisé ni intuitif de trouver un dimensionnement idéal.

Il faut donc d'une part réfléchir à un dimensionnement plus adapté aux architectures actuelles, et d'autre part prendre en compte les besoins applicatifs pour garantir une utilisation optimale des réseaux d'interconnexion et des modules mémoire partagés entre les unités de calcul.

3.1.3 Analyser l'exécution

Nous l'avons constaté, la complexité des architectures matérielles est désormais plus prononcée que jamais et la multitude des schémas applicatifs du monde scientifique entraîne une difficulté pour paramétrer correctement les outils de contrôle du parallélisme hybride sans exécution préalable. Ainsi, nous proposons de récupérer un certain nombre d'informations afin de mieux appréhender et comprendre les performances obtenues en utilisant un mélange de deux modèles de programmation. Nous nous appuyons sur un collecteur de traces d'exécutions afin de connaître à la fois le schéma de communication du modèle distribué et le comportement des tâches du modèle partagé. Grâce à ces informations, nous pouvons expliquer les écarts de performances entre plusieurs répartitions de granularité threads/processus.

3.2 L'architecture de la machine : une appréhension nécessaire

Plutôt que de considérer une machine de calcul comme une simple suite de processeurs accompagnés d'une certaine quantité de mémoire, notre solution permet une compréhension plus fine de l'architecture matérielle, cette compréhension forme la clé de voûte d'une utilisation optimale des calculateurs actuels.

3.2.1 La représentation du matériel par le système d'exploitation

Jouant le rôle d'intermédiaire entre le matériel informatique et les applications, le système d'exploitation gère l'accès aux ressources physiques (processeurs, mémoire, etc.) et offre une interface plus simple à utiliser et plus abstraite que celle du matériel. Il revient donc au système d'exploitation de fournir l'ensemble des détails architecturaux à l'utilisateur. Nous allons donc examiner quels sont les moyens qu'offre un système d'exploitation à l'utilisateur pour accéder à ces informations et si celles-ci sont suffisantes pour l'assister dans la combinaison de modèles de programmation.

3.2.1.1 Un accès difficile aux informations matérielles

Il est difficile pour un utilisateur quelconque de connaître avec précision les détails de la machine sur laquelle il désire exécuter son application. Si l'on considère les plate-formes expérimentales destinées aux scientifiques de tous domaines, les outils de soumission de tâches fournissent généralement certains détails comme le nombre d'unités de calculs à disposition (cœurs/processeurs), la quantité de mémoire vive ou les réseaux d'interconnexion dans le cas d'une grappe de calcul. Bien qu'utiles, ces informations sont loin d'être suffisantes pour optimiser l'exécution des applications sur les architectures modernes. En effet, rien ne nous dit comment sont agencés les cœurs : sont-ils sur la même puce ou sur des puces différentes ? Partagent-ils des caches de niveau L2 ou plus ensemble ? Si oui, de quelle taille ? Sont-ils connectés ensemble par un bus d'interconnexion d'une façon particulière ? Toutes les réponses à ces interrogations sont autant d'aides aux décisions de répartition et de placement des processus et threads utilisés par un modèle hybride de programmation, ainsi qu'aux décisions d'allocation mémoire.

Pour obtenir ces réponses, chaque système d'exploitation dispose d'une interface particulière. Ainsi sur les systèmes de type UNIX, il existe un système de fichiers particulier, généré dynamiquement au démarrage, qui contient diverses informations liées aux processus et au matériel. Parmi celles-ci, le fichier `/proc/cpuinfo` permet de connaître les informations sur les processeurs de la machine. Toutefois la figure 3.1 montre combien il est difficile d'en restituer une image claire et utilisable concrètement. Plus récemment, les systèmes Linux récents (à partir du noyau 2.6) offrent une interface plus structurée et plus complète pour accéder aux objets du noyau (par exemple le répertoire `/sys/devices/system/cpu/cpu0/topology/` contient l'ensemble des fichiers décrivant la topologie du premier processeur : son identifiant, celui de la puce auquel il appartient, etc.), mais l'extrapolation vers un schéma architectural clair reste peu évidente. Sur d'autres systèmes, comme Darwin, l'interface `sysctl` permet d'accéder aux paramètres du noyau comme la taille des caches ou encore la liste des cœurs d'une même puce : là encore l'obtention de ces informations est une procédure complexe pour le non spécialiste. Les méthodes de récupération d'informations matérielles étant aussi nombreuses que les systèmes d'exploitation, le travail de l'utilisateur désireux d'évaluer les performances d'une application sur plusieurs calculateurs peut s'avérer conséquent car non portable.

3.2.1.2 Une vision parfois biaisée

Outre l'aspect non portable des méthodes d'accès aux informations matérielles, celles-ci n'entrent pas toujours en accord avec la vision que l'on pense avoir de la machine. Ainsi, il est fréquent de constater que la simple numérotation des multiples cœurs d'un ordinateur est bien différente de celle qu'un utilisateur peut imaginer. Prenons l'exemple d'une machine dotée de 2 processeurs Intel *Xeon Nehalem X5550* à 4 cœurs, chacun des processeurs disposant d'un banc mémoire. La figure 3.2 montre d'un

```
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 26
model name : Intel(R) Xeon(R) CPU           X5560  @ 2.80GHz
stepping : 5
cpu MHz : 1596.000
cache size : 8192 KB
physical id : 1
siblings : 4
core id : 0
cpu cores : 4
apicid : 16
initial apicid : 16
fpu : yes
fpu_exception : yes
cpuid level : 11
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
 fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl pni
 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr dca sse4_1 sse4_2 popcnt lahf_lm ida
bogomips : 5585.99
clflush size : 64
cache_alignment : 64
address sizes : 40 bits physical, 48 bits virtual
power management:

processor : 1
vendor_id : GenuineIntel
cpu family : 6
model : 26
model name : Intel(R) Xeon(R) CPU           X5560  @ 2.80GHz
stepping : 5
cpu MHz : 1596.000
cache size : 8192 KB
physical id : 0
siblings : 4
core id : 0
cpu cores : 4
apicid : 0

...
```

FIGURE 3.1 – Exemple de contenu du fichier /proc/cpuinfo

côté une numérotation qui pourrait paraître logique à tout utilisateur et de l'autre la numérotation effective du système d'exploitation. Cette dernière repose sur une stratégie de répartition mémoire dans laquelle on suppose que les processus ou threads qui seront ordonnancés sur les cœurs de la machine vont chacun accéder à des zones mémoire différentes. Le placement réalisé par l'ordonnanceur du noyau se faisant dans l'ordre croissant, si l'on dispose de 4 processus par exemple, chacun d'entre eux disposera d'un banc mémoire sur lequel seront stockés les données qu'il utilise. Dans le cadre général d'un système d'exploitation qui doit gérer un ensemble de processus indépendants créés par différents utilisateurs, cette stratégie est tout à fait viable. C'est également le cas pour certains applications scientifiques dont le modèle mémoire correspond. Cependant, nombre de calculs impliquent désormais le multithreading, et par là-même le partage de l'espace d'adressage. Comment dès lors se douter que le thread 1 doit être placé sur le cœur numéroté 4 étant donné qu'il partage des données avec le thread 0 placé sur le cœur numéroté 0 ?

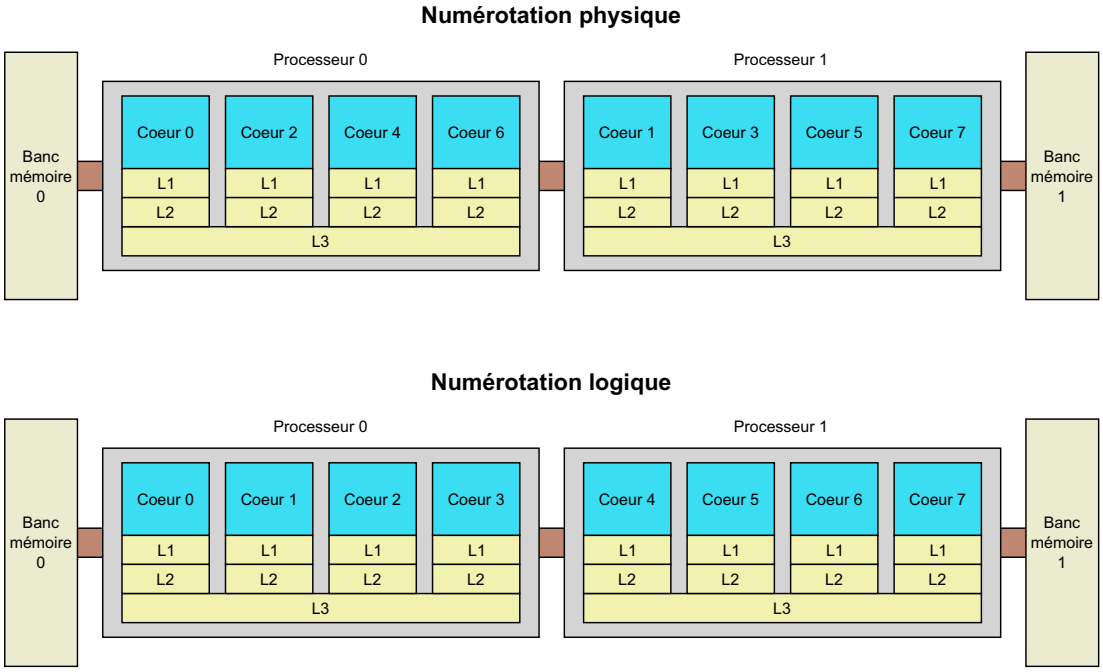


FIGURE 3.2 – Architecture Intel Xeon de 2 processeurs à 4 cœurs

Comme nous l'avons vu, il est tout à fait possible de connaître le nombre de processeurs/cœurs, ainsi que les relations entre ceux-ci (cache, banc mémoire, etc.) et par extension la structure hiérarchique de toute machine de calcul. Mais le cheminement est souvent complexe, parfois même trompeur. Nous pouvons nous interroger sur l'existence d'un outil offrant une interface claire, fonctionnant sur des systèmes différents et offrant une représentation simple et précise. Nous pouvons citer le logiciel *libtopology* [lib] dont le but est de découvrir la topologie matérielle des systèmes Linux uniquement. Outre

le fait qu'un seul système est supporté, ce projet n'est plus maintenu depuis quelques années. Dans le même registre, la bibliothèque PLPA (*Portable Linux Processor Affinity*) [PLP] fut intégrée au projet Open MPI pour permettre à son lanceur de processus de distribuer les processus sur les ressources de calcul diponibles. L'inconvénient de cette solution provient de son manque de précision quant à certains détails architecturaux : les caches partagés sont ignorés et cela peut générer de graves chutes de performances car pour certains processeurs, des caches ne sont partagés qu'entre un sous ensemble de cœurs d'une même puce. Par ailleurs, PLPA n'offre pas d'abstraction pour les nœuds mémoire alors que les machines NUMA sont pourtant répandues. Récemment l'interface PLPA a été réimplémentée au dessus d'un outil, plus performant capable de donner une vision générique et complète : hwloc.

3.2.2 Vers une abstraction générique

L'outil hwloc [BCOM+10] (*Portable Hardware Locality*) est un logiciel qui rassemble les informations concernant les processeurs, les caches, la mémoire et d'autres composants, et est capable de les transmettre aux applications et supports exécutifs par une abstraction hiérarchique.

3.2.2.1 Vers une représentation portable

hwloc a été conçu en s'appuyant sur le fait que les architectures actuelles et futures sont et seront fortement hiérarchiques. En effet, à l'image de la structure des machines d'aujourd'hui composées de plusieurs puces intégrant plusieurs cœurs potentiellement multithreadés, nous avons choisi d'exposer l'architecture matérielle comme un arbre de ressources. hwloc consiste en une bibliothèque de programmation et une série d'outils en ligne de commande. Le schéma 3.3 correspond à la sortie graphique de la commande `lstopo` exécutée sur une machine équipée de deux processeurs quad-cœur *Xeon E5345*. Nous observons ainsi l'arbre dont les feuilles correspondent aux unités de calcul, contenu dans les cœurs : dans le cas présent cela correspond physiquement au même composant mais dans d'autres cas les unités de calcul pourraient être des threads matériels composant les cœurs. Au troisième étage de l'arbre on retrouve les caches du premier niveau : chaque cœur dispose d'un cache L1. Ensuite, viennent les quatre caches L2, chacun partagé par deux cœurs. L'étage suivant correspond aux puces : sur notre exemple il y en a deux. Enfin la racine de l'arbre correspond à la machine elle-même. L'ensemble des relations entre les structures de données de hwloc correspondantes est décrit sur la figure 3.4. Sur celui-ci, chaque objet contient les valeurs des différents champs (profondeur dans l'arbre, index logique, etc.) et les flèches montrent quelles structures les pointeurs référencent (fils, parent, etc).

Sur d'autres architectures, un niveau supplémentaire peut apparaître : il s'agit du nœud mémoire. En effet, avec la démocratisation de l'architecture NUMA, il est nécessaire de prendre en considération le placement mémoire. D'ailleurs, pour les machines disposant d'un grand nombre de nœuds NUMA, comme les systèmes SGI Altix [SGIb],

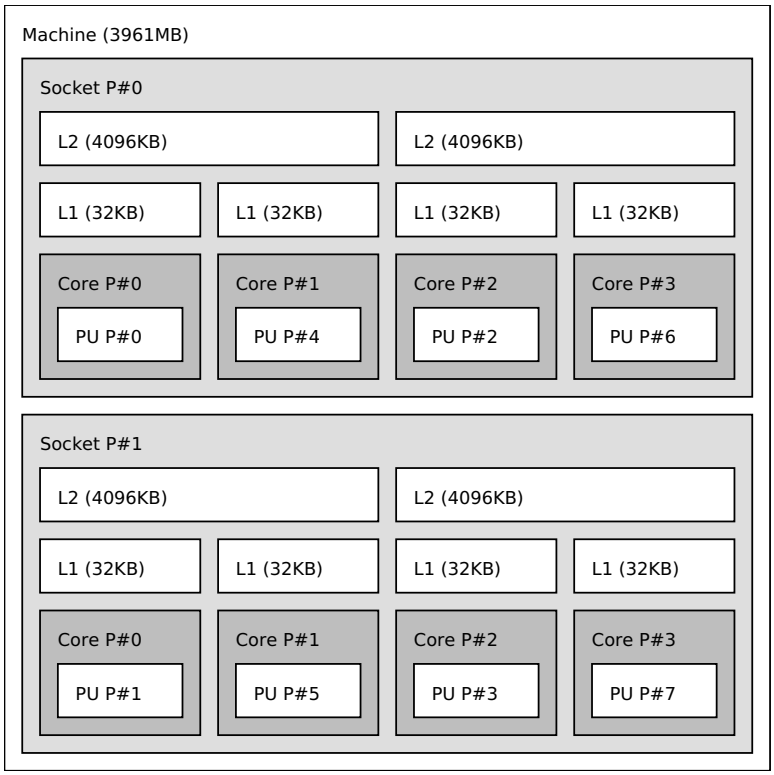


FIGURE 3.3 – Architecture de 2 processeurs à 4 cœurs modélisée par HWLOC

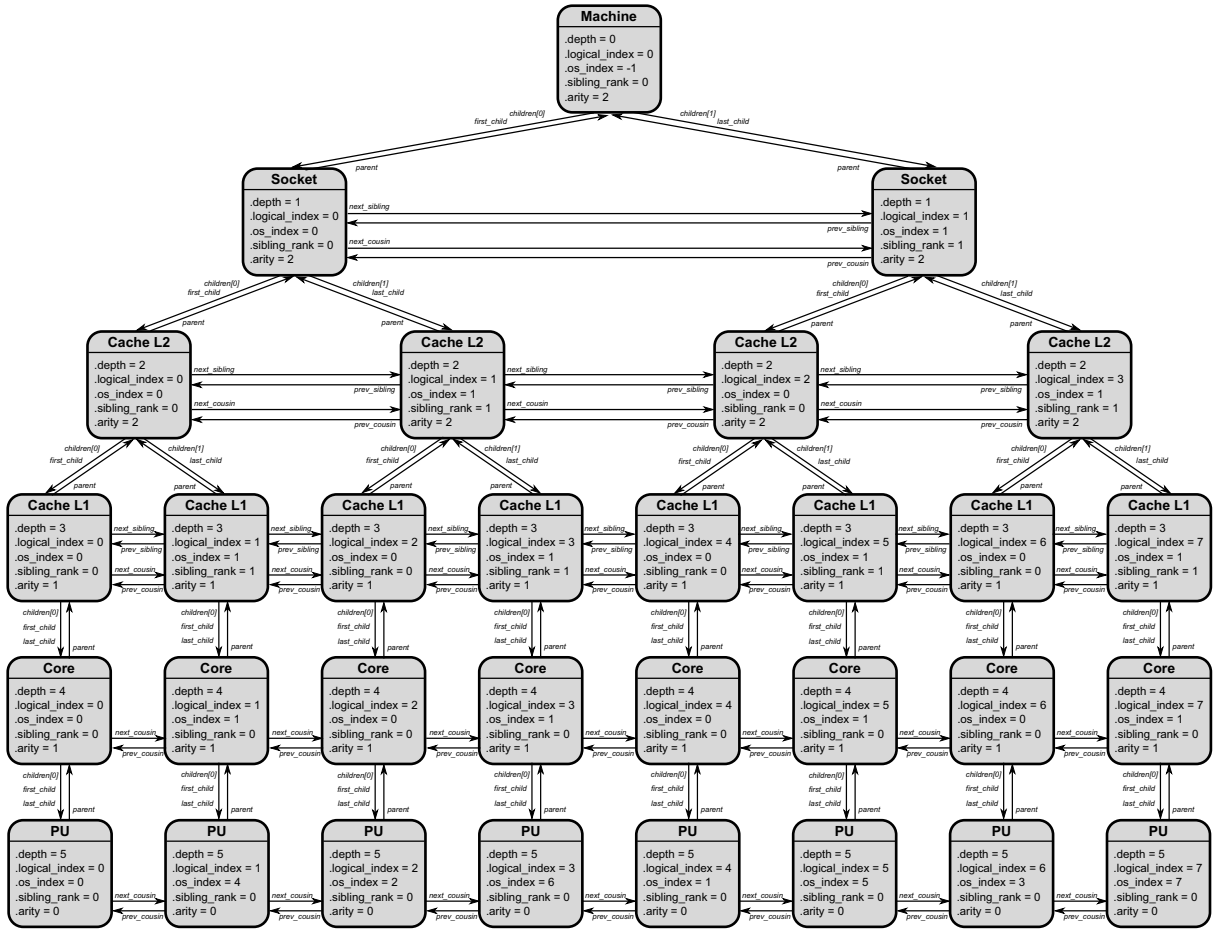


FIGURE 3.4 – Diagramme des relations entre les objets de HWLOC

HWLOC a la capacité de traduire les informations de la matrice des distances entre les nœuds (celle-ci étant fournie par le système d'exploitation) pour en déduire une organisation hiérarchique formelle.

La puissance de HWLOC réside dans sa capacité à s'adapter à tout type de machine, même celles qui ne sont pas encore produites. En effet, nous sommes partis du fait que des machines asymétriques (avec moins de cœurs sur certaines puces) ou hétérogènes (plusieurs sortes de processeurs dans une même architecture) verront le jour dans un futur plus ou moins proche. De ce fait, même si les calculateurs actuels ont une architecture symétrique, l'arbre construit par HWLOC est constitué d'objets génériques qui contiennent un type. Cette absence d'hypothèse sur une structure pre-déterminée soit concernant le type des objets existants (comme les cœurs ou les sockets), ou bien leur position relative dans la hiérarchie, accorde la possibilité d'employer HWLOC sur les architectures à venir. En effet, rien ne garantit que les constructeurs n'ajouteront pas de nouveaux types de ressources, ne déplaceront pas des caches en dehors des processeurs, ou ne modifieront pas la profondeur de certains composants.

3.2.2.2 Une riche interface de programmation

HWLOC rassemble les informations sur le matériel sous-jacent au démarrage. Il utilise les moyens fournis par le système d'exploitation pour le faire : par le système de fichier `sysfs` sous Linux, ou en appelant des fonctions d'une bibliothèque spécifique sous AIX, DARWIN, OSF, SOLARIS ou WINDOWS. À partir de ces informations, HWLOC réalise une représentation graphique ou textuelle, comme présentée en 3.5. Il est également possible de paramétrer l'outil pour obtenir une sortie au format XML, réutilisable par la suite pour éviter d'effectuer à nouveau un parcours topologique.

```
depth 0: 1 Machine (type #1)
 depth 1: 4 NUMANodes (type #2)
  depth 2: 4 Sockets (type #3)
   depth 3: 4 Caches (type #4)
    depth 4: 16 Caches (type #4)
     depth 5: 16 Caches (type #4)
      depth 6: 16 Cores (type #5)
       depth 7: 16 PUs (type #6)
```

FIGURE 3.5 – Sortie textuelle de HWLOC à propos d'une architecture à 4 processeurs AMD Opteron de 4 cœurs

Le logiciel offre un ensemble d'outils s'exécutant en ligne de commande. La commande `lstopo` dont nous avons déjà pu voir une sortie graphique précédemment. Elle affiche la topologie hiérarchique du système sur lequel elle est exécutée. Elle est également capable d'afficher les processus attachés à une partie de la machine. En effet, grâce

à la commande `hwloc-bind` il est possible de lier des processus à des éléments matériels spécifiques. Par exemple, l'exécution de l'opération `hwloc-bind socket:2 ./test` va exécuter l'application `./test` sur les unités de calcul de la deuxième puce de la machine.

Le logiciel bénéficie également d'une puissante bibliothèque écrite en langage C. Grâce à celle-ci, l'utilisateur récupère les informations topologiques sur tous les sytèmes d'exploitation supportés. Chacun d'entre eux fournit effectivement une interface spécifique qui, en plus de ne pas être portable sur d'autres systèmes, varie selon leur concept : certains utilisent des itérateurs pour parcourir les ressources et d'autres non, certains gèrent les objets de manière uniforme, etc. La bibliothèque C de HWLOC offre une abstraction de ces interfaces destinées à un système d'exploitaiton particulier dans une API portable. Cette dernière se compose de deux couches, de haut et de bas niveau, destinées à tirer profit des avantages de ces interfaces spécifiques. La couche de bas niveau, plus détaillée, s'adresse au programmeur avancé, et l'autorise a traverser l'arbre des ressources, de suivre les pointeurs vers les parents, les enfants ou les frères pour retrouver l'information désirée en utilisant les attributs topologiques comme la profondeur ou l'index. La figure 3.6 présente le code en langage C d'un programme qui parcourt la topologie du système sur lequel il s'exécute et l'affiche. Quant à la partie de haut niveau, elle offre des méthodes génériques pour trouver des ressources correspondant à certaines propriétés. Une fois ces informations obtenues l'application ou le support exécutif va adapter son comportement en fonction des caractéristiques du matériel sous-jacent (caches, nombre de cœurs, etc).

3.2.2.3 Intégrée dans de nombreux logiciels du domaine

Grâce à ses nombreuses capacités issues d'une puissante interface de programmation, HWLOC a su se rendre extrémement utile, voire même indispensable. En effet, de nombreux logiciels reposent sur cet outil pour appréhender les caractéristiques du matériel et en tirer profit. En effet, nous l'avons dit, la compréhension des architectures modernes est la clé de voûte dans la recherche de performances.

Les implémentations du standard MPI s'appuient sur un lanceur de processus pour répartir les multiples instances de code applicatif sur les unités de calcul à disposition. Ce même lanceur se doit de connaître en détails la structure du matériel sous-jacent pour placer correctement les processus MPI. Ainsi le projet Open MPI a profité de l'apparition de HWLOC pour tout d'abord porter son propre gestionnaire d'affinité PLPA au dessus, puis finalement a procédé à son remplacement complet. HWLOC fait désormais partie intégrante de la suite logicielle Open MPI. De même, du côté des implémentations MPICH2 et MVAPICH2, la plate-forme Hydra est le gestionnaire de processus qui s'appuie sur HWLOC pour contrôler leur affectation et attachement aux ressources de calcul.

Enfin, n'oublions pas de citer la bibliothèque de threads développées au sein de l'équipe RUNTIME, qui fut à l'origine de libtopology, la toute première version du logiciel HWLOC. La bibliothèque Marcel crée un ensemble de processeurs virtuels, ordonnés

```
#include <hwloc.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    int depth, levels, topodepth;
    unsigned i;
    char string[128];
    hwloc_topology_t topology;
    hwloc_obj_t obj;

   /* Allocation et initialisation de l'objet représentant la topologie */
    hwloc_topology_init(&topology);

   /* Réaliser la détection de la topologie */
    hwloc_topology_load(topology);

    /* Obtenir la profondeur de la topologie */
    topodepth = hwloc_topology_get_depth(topology);

    /* Parcours de la topologie à la manière d'un tableau, depuis le niveau 0
       (niveau du système) jusqu'au niveau le plus bas (le niveau des processeurs) */
    for (depth = 0; depth < topodepth; depth++) {
        printf("*** Objets du niveau %d\n", depth);
        for (i = 0; i < hwloc_get_nbobjs_by_depth(topology, depth); i++) {
            hwloc_obj_snprintf(string, sizeof(string), topology,
                       hwloc_get_obj_by_depth(topology, depth, i), "#", 0);
            printf("Index %u: %s\n", i, string);
        }
    }

    /* Destroy topology object. */
    hwloc_topology_destroy(topology);

    return 0;
}
```

FIGURE 3.6 – Exemple de programme C utilisant l'interface de HWLOC pour afficher la topologie du système

selon la numérotation logique de HWLOC, et sur lesquels un ensemble de threads noyaux vont être attachés.

3.2.3 De l'importance du placement des processus et des threads

Il apparaît que le logiciel HWLOC est un outil portable qui offre une interface générique utiles à de nombreuses bibliothèques et applications qui s'exécutent sur les calculateurs modernes. Les informations détaillées qu'il délivre sur le matériel et les mécanismes, tels l'attachement mémoire, permettent des optimisations et mènent à des gains de performances qu'il serait difficile d'obtenir autrement.

3.2.3.1 Exploiter la localité des données

En considérant les architectures NUMA dont chaque banc mémoire est associé à un groupe de processeurs multicœurs, le positionnement d'une donnée par rapport à l'unité de calcul qui va l'utiliser pendant l'exécution est un facteur déterminant dans l'amélioration des performances applicatives. L'impact se révèle d'autant plus important pour les programmes qui accèdent intensivement à la mémoire.

Le langage OpenMP, par exemple, offre un ensemble de directives de compilation dont le but est de réaliser le partage de données et des tâches à effectuer entre plusieurs threads. Particulièrement adapté aux machines de type SMP, il est plus difficile d'atteindre des performances équivalentes, proportionnellement parlant, sur les processeurs multicœurs actuels. Des alternatives ont alors vu le jour pour combler ce manque et tenter de tirer avantage des processeurs modernes. L'une d'entre elles, ForestGOMP [Bro10], étend le standard OpenMP en reliant les threads selon leur affinité. À chaque section parallèle rencontrée il crée une structure récursive appellée *Bulle* qui contient les threads de la section. Tout ceci génère un arbre de threads à partir d'un programme OpenMP. La difficulté consiste alors à mettre en correspondance cet arbre avec une représentation, également arborescente, de la topologie matérielle, fournie par HWLOC. L'une des stratégies déployées par ForestGOMP consiste à placer sur des cœurs proches (partageant une même mémoire cache) les threads qui travaillent sur les mêmes données. L'extension obtient ainsi de meilleurs performances que le support exécutif OpenMP de GCC (GOMP) [Nov06, GOM, BFG+10].

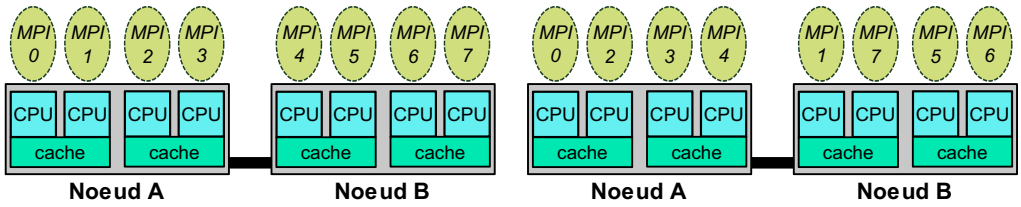
Dans le cadre d'un programme élaboré par un mélange de deux types de modèles, tel MPI et OpenMP, il convient de placer les threads travaillant sur les mêmes données sur des cœurs voisins. Et le modèle hybride va faciliter l'atteinte de cet objectif. En effet, le principe du modèle est de créer un certain nombre de processus MPI qui vont chacun partager le travail à effectuer à un ensemble de threads OpenMP. Ces derniers sont donc censés utiliser des données proches et même en partager certaines. Il suffit alors d'affecter chaque processus à un groupe d'unités de calcul adjacentes dans la hiérarchie de la machine fournie par HWLOC. En effet, les accès mémoire réalisés par les threads d'un même processus auront tout intérêt à se faire à partir d'une même puce pour bénéficier d'une latence mémoire optimale. Bien que l'on puisse espérer qu'une politique

répartissant les threads sur toute la machine de calcul permette de maximiser l'utilisation de la bande passante en évitant autant que possible les contentions mémoire, cela ne se confirme que dans le cas d'une machine peu chargée. Lorsque tous les processeurs sont occupés, il est préférable de réaliser des accès locaux pour obtenir les meilleures performances pour l'application.

3.2.3.2 Impact du placement dans les communications

Autre pilier du modèle hybride, les bibliothèques de communication du standard MPI se doivent d'exploiter au mieux les architectures modernes. Les implémentations récentes telles Open MPI ou MPICH2 offrent un niveau de performances extrêmement satisfaisant sur les architectures multicœurs. Les développeurs de ces implémentations ont choisi de distinguer les communications intra nœuds et les communications inter nœuds, pour optimiser les transferts dans les deux cas [BMG07, GFB+04b]. En utilisant les mécanismes de partage mémoire, tels `shmem` ou `mmap`, il est possible de transférer des données d'un processus à un autre au sein d'un même nœud. Ainsi, plutôt que de réaliser des transferts traversant des couches de protocoles réseau, susceptibles d'user de tampons mémoire, et donc coûteux en temps, les messages destinés aux processus du même nœud sont directement copiés dans leur mémoire. Afin de franchir un pas supplémentaire dans la course aux performances, les processus doivent être placés avec précaution sur les cœurs. Car, aussi performant que puisse être une bibliothèque MPI, si l'objectif est de tirer la quintessence des machines de calcul pour l'exécution d'une application, il est nécessaire de définir une politique de placement tenant compte des caractéristiques matérielles et du schéma de communication applicatif.

Chaque application utilisant MPI suit un schéma de communication qui lui est propre. La norme fournit tant de types de primitives différentes, de part leur mode de transfert (bloquant ou non, bufferisé, etc.), l'identité et le nombre de processus concernés (communications unilatérales, bilatérales ou collectives), qu'il n'est pas possible de déduire immédiatement le placement idéal qui optimisera le délai d'exécution des programmes. L'étude du schéma de communication permet de connaître les interactions entre les processus et par là-même de déduire des liens plus ou moins forts entre ceux-ci. Dans [MCO09], nous avons choisi la quantité de données échangées entre les processus comme critère discriminant. En calculant cette valeur pour chaque couple de processus, nous pouvons déterminer le schéma de communication. Ce schéma peut être représenté par une matrice (c'est-à-dire un graphe) où les indices correspondent aux identifiants des processus et les valeurs de la matrice la quantité de donnée échangéee entre deux processus. L'étape suivante consiste à établir une représentation matricielle du matériel sous-jacent à partir des informations topologiques fournies par hwloc : les valeurs de la matrice sont les cœfficients de distance entre les unités de calcul. Au final, la problématique de placement se résume à effectuer une correspondance statique entre deux graphes (représentés par les matrices). L'opération de correspondance est un problème de théorie des graphes (un plongement) que nous avons choisi de résoudre grâce au logiciel Scotch [Fra08, (F.94] qui applique un algorithme de bi-partitionnement récur-



(a) Placement par défaut

(b) Placement privilégiant les affinités entre les processus MPI

Processus MPI	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	0							
1	20	0						
2	2700	20	0					
3	1125	30	750	0				
4	750	30	1125	2700	0			
5	30	1125	30	20	20	0		
6	30	750	30	20	20	2700	0	
7	20	2700	20	30	30	750	1125	0

(c) Nombre de messages transmis entre les processus

FIGURE 3.7 – Placement des processus d'une application sur deux nœuds à quatre coeurs

sif. Bien entendu, la résolution du placement optimal est largement inférieur à celui de l'exécution de l'application. Grâce aux commandes d'attachement fournies par HWLOC, chaque processus est affecté à un cœur proche des cœurs sur lesquels s'exécuteront les processus avec lesquels il aura le plus d'interactions. Par ce placement, on escompte maximiser le nombre de communications intra nœuds, moins coûteuses que celles passant par un réseau d'interconnexion, et ainsi améliorer les performances globales de l'application.

3.3 Vers une automatisation du dimensionnement

Lorsque l'on emploie un modèle de programmation hybride, c'est en général en développant ou en réactualisant une application pour qu'elle puisse s'approcher des performances théoriques des machines de calcul modernes. Le principe est simple : combiner un modèle qui fonctionne bien sur des architectures à mémoire partagée avec un autre particulièrement performant dans le cadre des grappes de calcul. Cependant associer deux paradigmes de programmation, tels les standards MPI et OPENMP, est une opération plus complexe qu'il n'y paraît. Nombreux sont ceux qui supposent que le meilleur moyen de combiner ces deux modèles consiste à utiliser les threads OPENMP pour répartir le travail entre les unités de calcul d'un nœud et d'employer le passage de messages uniquement pour les communications entre les cœurs de nœuds différents.

Au lieu de se restreindre à un cadre aussi rigide, nous proposons ici une approche plus

fine et mieux adaptée. Le dimensionnement du nombre des threads et des processus qui les contiennent doit se faire en prenant en compte plus de paramètres comme la structure hiérarchique des nœuds de calcul ou encore les limites des supports d'exécution des modèles combinés.

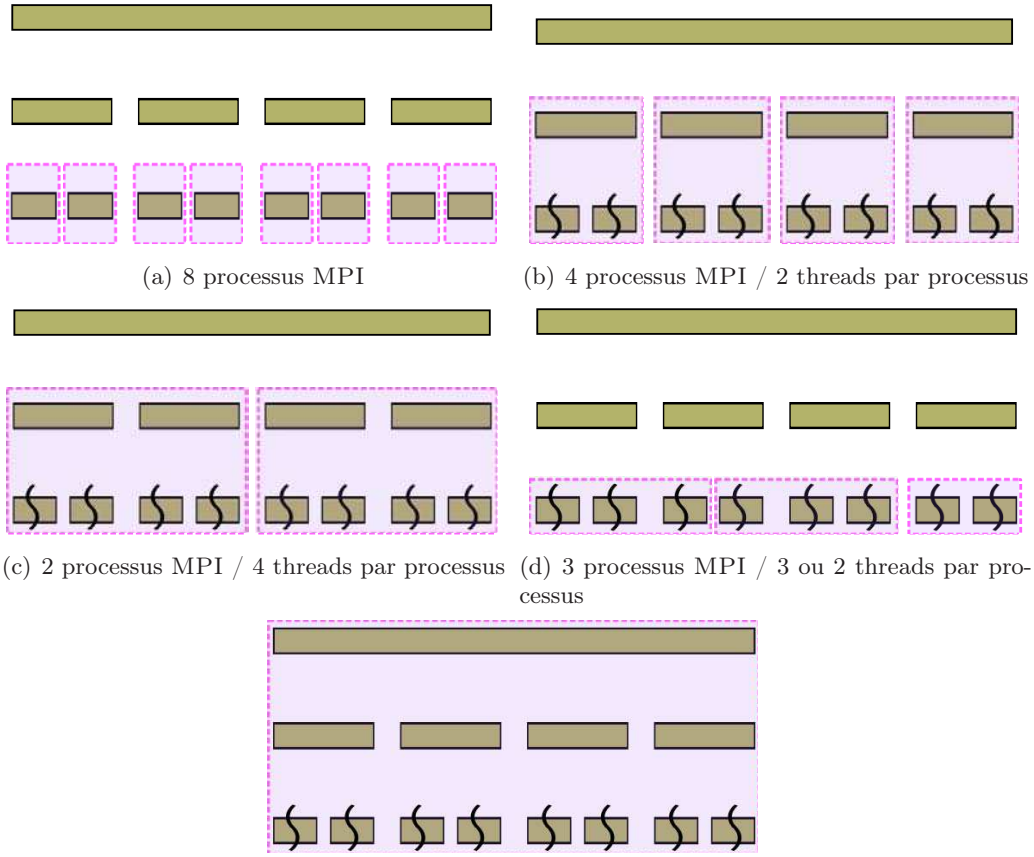
3.3.1 Accorder la distribution avec la hiérarchie de l'architecture

Dans le cadre d'un modèle hybride, on serait tenté de laisser le support exécutif OpenMP se charger de la répartition des threads sur l'intégralité des cœurs de chacun des nœuds (en utilisant un seul processus MPI par nœud). Cependant, rien dans le standard ne permet d'exprimer le fait que certains threads travailleront sur les mêmes données ou même de définir les affinités entre des threads et des données. Une possibilité serait de modifier le support exécutif OpenMP lui-même et plus précisément la partie qui gère l'ordonnancement pour y intégrer l'affinité entre threads, comme le logiciel Forest GOMP évoqué précédemment. Toutefois, c'est une solution qui requiert de sérieuses modifications du support exécutif implémentant OpenMP.

Au lieu de modifier un standard, nous proposons ici de confiner les threads travaillant sur des données proches ou similaires sur des ensembles pertinents d'unités pour gérer les affinités, et ceci au niveau utilisateur. Aucune modification interne de la bibliothèque MPI ou OpenMP n'est alors nécessaire. Il s'agit d'utiliser judicieusement les outils de ces deux standards pour contrôler le dimensionnement des unités d'exécution propres à chacun des modèles. Et bien qu'une répartition utilisant un processus MPI par nœud (et autant de threads qu'il y a de cœurs disponibles à l'intérieur du processus) ou une répartition avec un processus MPI par cœur (donc un unique thread par processus) soient les solutions généralement choisies, il nous semble approprié de regarder des répartitions qui correspondent mieux à la hiérarchie de chacun des nœuds du calculateur.

De la structure arborescente donnée par l'analyse de hwloc concernant une machine, nous pouvons en extraire plusieurs paliers correspondants aux mémoires partagées (caches, bancs mémoires, etc.). Chacun de ces paliers va décrire un agencement pour les processus MPI et les threads OpenMP. Notre outil enregistre les différentes combinaisons réalisables en partant du palier le plus bas (autant de processus MPI qu'il y a d'unités de calcul) pour ensuite remonter d'un niveau jusqu'à atteindre le palier le plus élevé (un seul processus MPI sur le nœud de calcul). Entre ces deux extrèmes, il existe nombre de répartitions, certaines plus incongrues que d'autres. La figure 3.8 schématise certaines d'entre elles sur sur une machine théorique, imaginée pour l'exemple, et composée de huit unités de calcul. Selon cette figure, les élements de chaque paire d'unité de calcul partagent une même mémoire (un même niveau de cache par exemple). La topologie admet donc trois paliers que nous pourrions assimiler respectivement aux cœurs, aux caches et enfin à la machine. La répartition 3.8(b) dispose quatre processus exécutant deux threads chacun, en pariant sur le fait que les échanges entre threads au sein de chaque processus seront réalisés principalement par la mémoire cache et optimiseront l'exécution de l'application. Une autre répartition comme la 3.8(c) pourrait être également envisagée car elle rassemblerait quatre threads par processus, cependant

l'analyse de l'architecture nous laisse penser qu'aucun gain ne sera obtenu étant donné que les quatre threads ne partagent, de façon exclusive, aucune mémoire commune. Quant à la répartition 3.8(d), elle constitue également une possibilité mais est incohérente au niveau d'un quelconque partage mémoire, ce qui risque de desservir notre objectif d'amélioration des performances.



(a) 8 processus MPI

(b) 4 processus MPI / 2 threads par processus

(c) 2 processus MPI / 4 threads par processus

(d) 3 processus MPI / 3 ou 2 threads par processus

(e) 1 processus MPI / 8 threads par processus

FIGURE 3.8 – Répartitions des processus et threads sur une architecture virtuelle composée de 8 cœurs

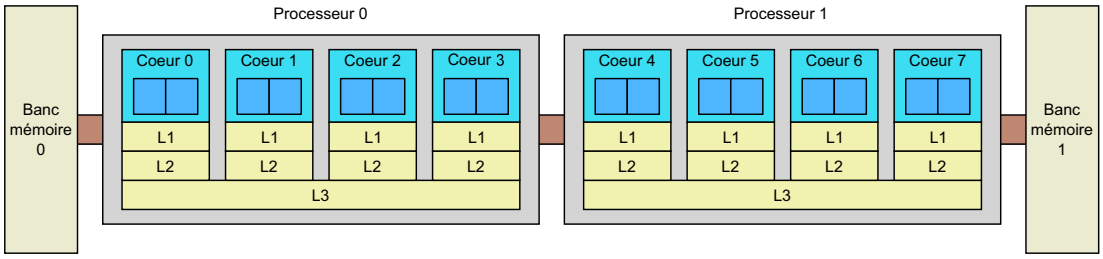
La figure 3.9 montre les multiples agencements calculés par notre algorithme sur une architecture composée de deux processeurs INTEL *Xeon* quadri-cœur hyperthreadés. Selon les caractéristiques topologiques de cette architecture, 4 distributions ont été établies. Il est ainsi possible de placer un processus MPI par thread matériel, ou bien de créer un processus MPI par cœur contenant deux threads chacun. On peut également profiter du fait que les quatre caches de niveau 3 permettent à quatre cœurs une communication plus rapide (qu'en passant par la mémoire principale) pour créer seulement deux processus MPI contenant deux threads et attachés à ces paires de cœurs. La der-

nière répartition consiste à faire entièrement confiance au support exécutif OpenMP en lui laissant l'intégralité du nœud de calcul : un seul processus MPI contient alors seize threads.

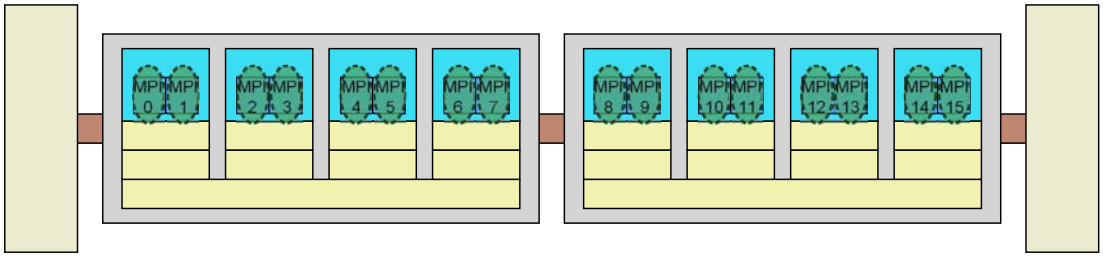
L'outil que nous proposons permet de déterminer automatiquement les multiples répartitions qu'il convient de prendre en considération en s'appuyant sur les informations récoltées par hwloc. Chacune d'entre-elles est alors évaluée et comparée aux autres pour obtenir la meilleure façon de mélanger deux modèles de programmation. La figure 3.10 présente la commande à exécuter, ainsi que les résultats de son évaluation sur l'architecture que nous venons de voir. L'utilisateur doit seulement préciser l'emplacement du fichier exécutable de son programme pour obtenir ces informations. Notre outil va alors analyser l'implémentation MPI, et principalement la commande `mpiexec` afin de transmettre les paramètres corrects : en effet, toutes les implémentations fournissent cette commande mais son utilisation va différer de l'une à l'autre. Une étude de l'ensemble des nœuds de la plate-forme de calcul sur laquelle l'utilisateur veut exécuter le programme est ensuite réalisée. Si rien n'est spécifié à ce propos, ce sera la machine locale qui sera choisie. Grâce à la bibliothèque hwloc, nous savons que nous disposons de seize unités de calcul. Étant donné que l'utilisateur n'a pas désiré tester toutes les distributions réalisables, quatre distributions en ressortent. Elles correspondent bien à celles de la figure 3.9. Il faut toutefois noter que notre outil ici va afficher les numéros des processeurs tels que le système les voit, alors que sur le dessin de la figure précédente nous avons optée pour une numérotation dite « logique ». Par la suite, chaque distribution est testée, évaluée puis comparée aux autres. Ici la distribution qui a été la plus efficace en durée d'exécution a été celle utilisant un seul processus MPI et 16 threads OpenMP.

3.3.2 Réduire le coût des communications

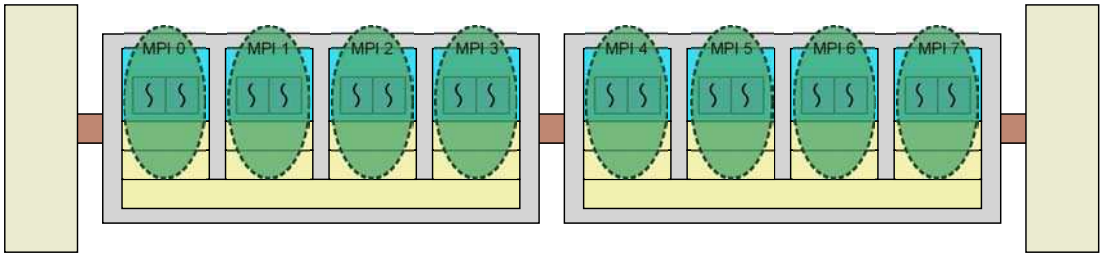
Dans un modèle de programmation hybride, les performances dépendent autant de la façon dont sont ordonnancés les flux d'éxécution du modèle à mémoire partagée que de l'efficacité des communications intervenants entre les instances exécutives du modèle distribué. L'impact des communications sur le temps d'exécution des applications est suffisamment important pour que de nombreux travaux aient été entrepris ces dernières années dans le développement des bibliothèques de communication telles que les implémentations MPI. Le principal objectif est de réduire à son plus bas niveau les délais induits par la couche logicielle et ainsi améliorer les échanges de messages. Toutefois, sous certaines conditions une bibliothèque de communication peut rencontrer des difficultés à exercer son rôle de manière efficace. Lorsque de nombreux processus s'exécutent simultanément au sein d'une même application, la bibliothèque alloue une quantité significative de mémoire liées à ses mécanismes internes et doit gérer un grand nombre de structures : une liste de requêtes en attente de disponibilité de la carte réseau, une zone mémoire pour les messages inattendus, etc. Qui plus est, plus de processus implique plus de messages circulant sur le réseau et le surcoût engendré, notamment pour les communications collectives influe directement sur les performances de l'application. Nous



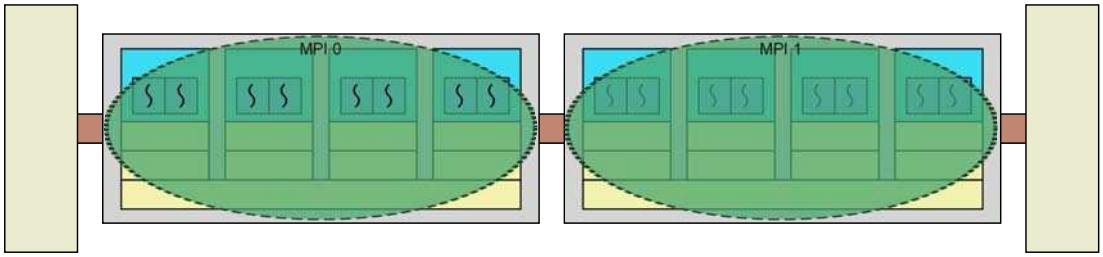
(a) Architecture à 2 processeurs Intel Xeon X5550



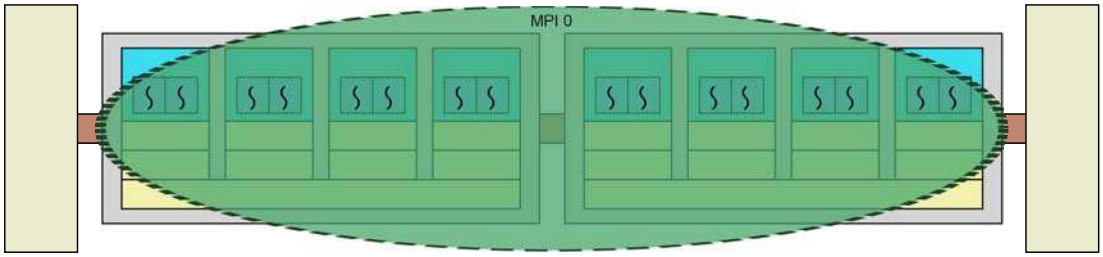
(b) 16 processus MPI



(c) 8 processus MPI / 2 threads par processus



(d) 2 processus MPI / 8 threads par processus



(e) 1 processus MPI / 16 threads par processus

FIGURE 3.9 – Les 4 répartitions calculées à partir d'une architecture à deux processeurs à quatre cœurs hyperthreadés

```
$>~: ggrun --measure -v ./hello-hybrid
Measure mode enabled : np, ppn, omp, test, debug and graphic options will be ignored.
checking if mpiexec is available... yes
Seeking topology of node localhost
checking if hwloc-calc is available... yes0,8,1,9,2,10,3,11,4,12,5,13,6,14,7,15(16 cpus found)
checking if all-dist option was given... no
checking if hwloc-info is available... yes
Computing distribution for 1 process(es) :
Distributing 1 process(es) on node localhost(16core(s) per process)
Adding a process on 16 cores (0, 8, 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15)

Computing distribution for 2 process(es) :
Distributing 2 process(es) on node localhost(8core(s) per process)
Adding a process on 8 cores (0, 8, 1, 9, 2, 10, 3, 11)
Adding a process on 8 cores (4, 12, 5, 13, 6, 14, 7, 15)

Computing distribution for 8 process(es) :
Distributing 8 process(es) on node localhost(2core(s) per process)
Adding a process on 2 cores (0, 8)
Adding a process on 2 cores (1, 9)
Adding a process on 2 cores (2, 10)
Adding a process on 2 cores (3, 11)
Adding a process on 2 cores (4, 12)
Adding a process on 2 cores (5, 13)
Adding a process on 2 cores (6, 14)
Adding a process on 2 cores (7, 15)

Computing distribution for 16 process(es) :
Distributing 16 process(es) on node localhost(1core(s) per process)
Adding a process on 1 cores (0,)
Adding a process on 1 cores (8,)
Adding a process on 1 cores (1,)
Adding a process on 1 cores (9,)
Adding a process on 1 cores (2,)
Adding a process on 1 cores (10,)
Adding a process on 1 cores (3,)
Adding a process on 1 cores (11,)
Adding a process on 1 cores (4,)
Adding a process on 1 cores (12,)
Adding a process on 1 cores (5,)
Adding a process on 1 cores (13,)
Adding a process on 1 cores (6,)
Adding a process on 1 cores (14,)
Adding a process on 1 cores (7,)
Adding a process on 1 cores (15,)

                        Global        localhost           Timing            Ratio
+------------+    +------------+   +------------+   +------------+   +------------+
Total:                                                 78.294039 ms
MPI:               1 process(es)    1 process(es)              n/a
GOMP:               16 thread(s)     16 thread(s)              n/a
+------------+    +------------+   +------------+   +------------+   +------------+
Total:                                                166.761875 ms
MPI:               2 process(es)    2 process(es)              n/a
GOMP:                8 thread(s)      8 thread(s)              n/a
+------------+    +------------+   +------------+   +------------+   +------------+
Total:                                                232.054949 ms
MPI:               8 process(es)    8 process(es)              n/a
GOMP:                2 thread(s)      2 thread(s)              n/a
+------------+    +------------+   +------------+   +------------+   +------------+
Total:                                                714.226007 ms
MPI:              16 process(es)   16 process(es)              n/a
GOMP:                1 thread(s)      1 thread(s)              n/a
+------------+    +------------+   +------------+   +------------+   +------------+
Best distribution was with 1 process(es). (78.294039 ms)
```

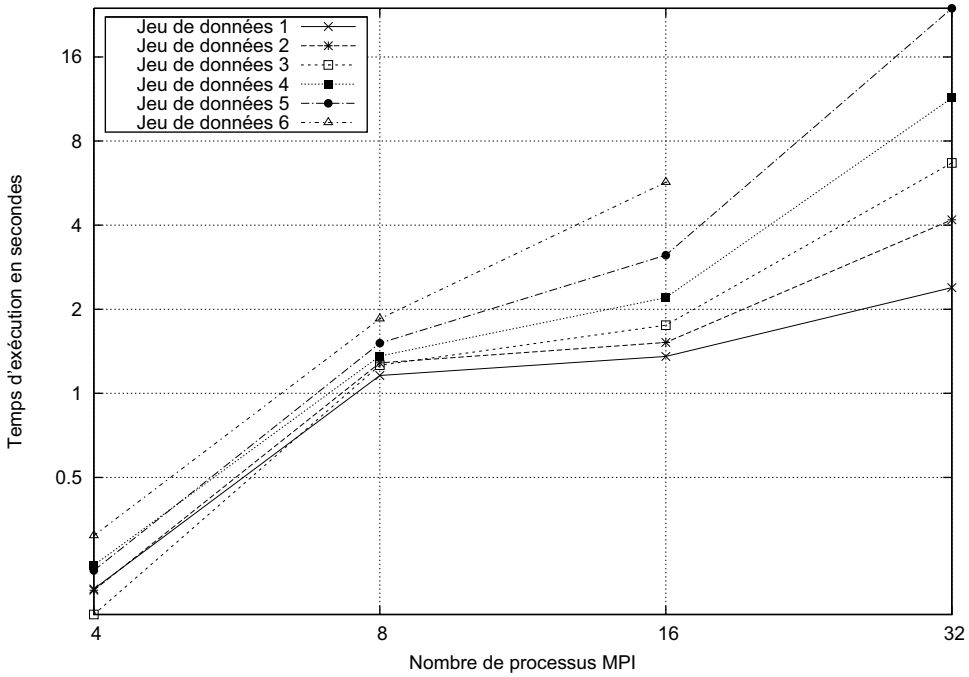
FIGURE 3.10 – Calcul automatique de la meilleure répartition pour un programme donné

avons mis en évidence ce phénomène au travers d'un programme effectuant une série de communications collectives sur un ensemble variable de processus. Sur 4 nœuds disposant chacun de deux processeurs Intel Xeon Nehalem quadri-cœurs. Nous mesurons le temps d'exécution d'une boucle de 128 appels à la primitive *MPI_Alltoall* : chaque processus transmet un message d'une taille τ à tous les autres et en reçoit un de même taille en provenance de chacun des autres. La figure 3.11 fournit les valeurs de τ pour les 6 différents jeux de données que nous avons sélectionnés. Les deux implémentations libres du standard MPI, Open MPI et MVAPICH2, sont évaluées dans le cadre de ce test. Le temps d'exécution de notre programme est mesuré pour successivement 4, 8, 16 et 32 processus : ces valeurs correspondent aux différentes distributions possibles sur cet ensemble de quatre nœuds. Bien entendu, les 4, 8, 16 et 32 processus disposent respectivement de 8, 4, 2 et 1 cœur(s) chacun. Nous aurions tout à fait pu simuler un calcul quelconque en créant une section parallèle OpenMP (qui se chargerait d'une multiplication matricielle par exemple) avant chaque communication MPI. Cependant nous nous concentrons ici sur les performances des communications. La figure 3.12 affiche les courbes du temps d'exécution du programme. Nous voyons clairement que l'augmentation du nombre de processus génère un surcoût de plus en plus important : jusqu'à plus de 20 secondes pour 32 processus. De plus, pour des messages de grande taille ce phénomène s'accentue. Alors que le programme continue de s'exécuter dans un temps raisonnable pour 4 ou 8 processus, ce quel que soit la taille des messages, lorsque 32 processus s'échangent les données, le temps d'exécution est presque 50 fois supérieur.

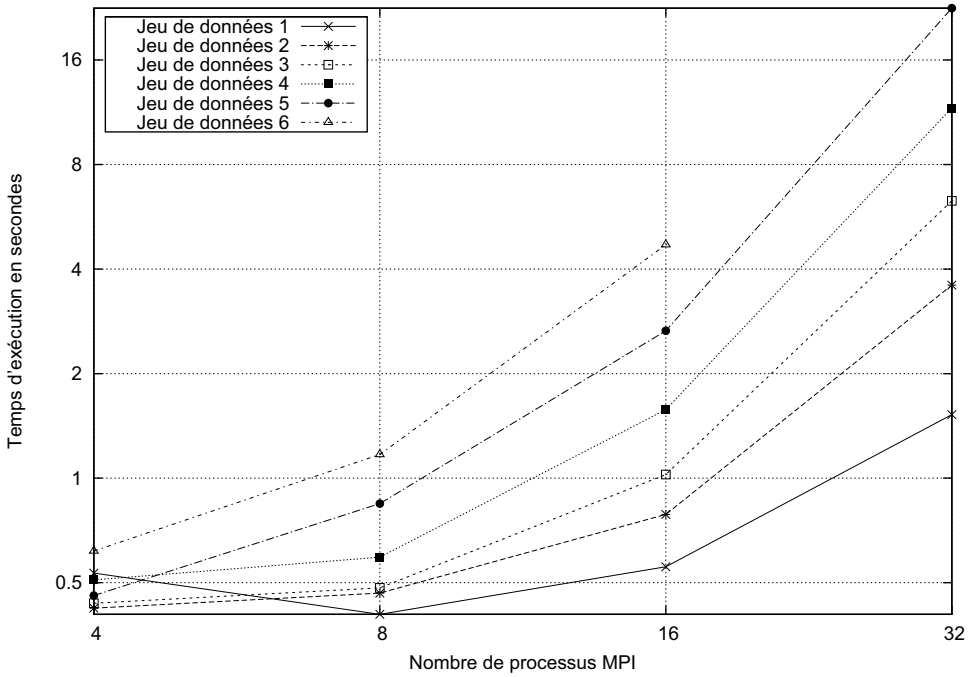
	Jeu de données 1	Jeu de données 2	Jeu de données 3	Jeu de données 4	Jeu de données 5	Jeu de données 6
Taille des messages pour 4 processus	8,48ko	33,91ko	67,81ko	135,63ko	271,25ko	
Taille des messages pour 8 processus	1,82ko	7,27ko	14,53ko	29,06ko	58,13ko	116,25ko
Taille des messages pour 16 processus	434 octets	1,70ko	3,39ko	6,78ko	13,56ko	27,13ko
Taille des messages pour 32 processus	105 octets	420 octets	840 octets	1,64ko	3,28ko	6,56ko
Quantité totale de données transférées	**101,72ko**	**406,88ko**	**813,75ko**	**1,59Mo**	**3,18Mo**	**6,36Mo**

Figure 3.11 – Taille des données échangées pour 6 jeux de données

La capacité de dimensionnement que nous offre la programmation hybride permet



(a) OpenMPI



(b) Mvapich2

Figure 3.12 – Impact du nombre de processus sur les performances de la bibliothèque MPI

de résoudre cette problématique en régulant le nombre de processus impliqués. La duplication mémoire est ainsi limitée et le partage mémoire vient remplacer les transferts de messages. Grâce à notre outil, l'utilisateur peut réguler le nombre de processus MPI selon sa convenance et le nombre de threads contenus dans chaque processus est modifié en conséquence (par défaut, il est équivalent au nombre de cœurs disponible par processus). Si pour une raison quelconque on désire ne pas utiliser tous les cœurs de la machine, il est également possible de préciser le grain de multithreading désiré.

3.4 Analyse du comportement des applications

Au-delà des possibilités de dimensionnement des applications et des décisions réalisées par notre outil, aiguillées par les informations topologiques du matériel de calcul récoltées, la compréhension des performances du modèle hybride ne peut se faire sans apport supplémentaire. Pour une application donnée, il ne suffit pas de trouver une répartition particulière qui fonctionne plus efficacement que les autres. Il faut aussi en connaître les raisons et expliquer les problèmes de performances rencontrés par ces répartitions alternatives. Cela vient-il d'un trop grand nombre de synchronisations entre des threads ? Le réseau est-il en surcharge ? La mémoire fait-elle défaut ? Nous proposons donc de récupérer les informations qui nous seront nécessaires pour répondre à ces interrogations.

3.4.1 Capter les informations

Afin de minimiser l'impact que pourrait avoir la collecte de traces sur le temps d'exécution des programmes nous avons choisi de récolter les informations grâce au logiciel EZTRACE [TRF+11]. Il présente l'avantage d'être peu invasif et ne nécessite aucune modification ou recompilation du code, contrairement à d'autres solutions comme VAMPIRTRACE ou SCALATRACE [NRM+09]. Par ailleurs, le système de traces de EZTRACE est réellement générique car il ne s'appuie sur aucune implémentation particulière du standard MPI. Par conséquent, l'utilisateur disposera du choix de l'implémentation qui correspondra le mieux à ses attentes.

3.4.1.1 Interception des appels

La première tâche effectuée par le système de traces correspond à l'enregistrement de tous les événements notables survenant au cours de l'exécution de l'application. EZTRACE délègue cette tâche à la bibliothèque FXT [DNW05]. Le surcoût apporté à une application est tout à fait négligeable. FXT intrumente le code grâce à quelques macros, très courtes, qui ne pertubent que très peu l'application. Ces macros enregistrent tout un ensemble d'informations qui seront exploitées par des outils d'analyse *post-mortem*. La synchronisation interne à FXT, nécessaire lorsque plusieurs tâches instrumentées s'exécutent en parallèle, est assurée par des opérations atomiques du processeur ce qui

permet d'observer le comportement du support exécutif (MPI ou OpenMP) sans interférer avec son ordonnancement. Un grand nombre d'événements sont préenregistrés dans les entêtes de la bibliothèque FxT : changement de contexte de threads, interruptions, etc. L'ajout de nouveaux types d'événements se fait très facilement par les primitives de l'interface.

De plus, l'instrumentation du code applicatif ne requiert aucune modification. Le principe d'Eztrace consiste à ouvrir les bibliothèques dynamiques appropriées (MPI, OpenMP, pthread) pour ensuite surcharger, c'est-à-dire redéfinir, l'ensemble des primitives que l'on désire instrumenter. On intègre alors dans cette nouvelle définition les macros FxT. À chaque invocation d'une fonction instrumentée, ce sera la fonction redéfinie qui sera appelée et qui modifiera les compteurs de FxT, en plus d'exécuter le code originel de la fonction. Si la fonction n'est pas instrumentée par Eztrace, le code de la fonction, tel qu'il est écrit dans la bibliothèque initiale, sera utilisé. Dans le cas de la bibliothèque MPI, toutes les méthodes de l'interface de programmation sont interceptées et on ajoute une trace décrivant l'action réalisée, avec certains détails spécifiques à l'événement et le moment où cet événement est intervenu. Le code de la figure 3.13 montre comment Eztrace redéfinit le code de la fonction *MPI_Send*. La première macro, `EZTRACE_EVENT3(...)`, enregistre le début de l'exécution de la fonction *MPI_Send* avec la taille du message envoyé, l'identifiant de l'émetteur, celui de son destinataire, l'étiquette (*tag*) et le communicateur. L'appel à la méthode de la bibliothèque MPI est ensuite réalisé, sans aucune modification des paramètres. La seconde macro permet d'enregistrer la date de fin d'exécution de l'appel. Le mécanisme est similaire pour l'ensemble des primitives de la bibliothèque MPI. Etant donné que l'interface fait partie du standard, Eztrace n'est pas lié à une quelconque implémentation, et fonctionne donc avec n'importe laquelle d'entre-elles.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,int dest, int tag,
             MPI_Comm comm)
{
  int size;
  /* retrieve the size of the datatype so that we can compute */
  /* the message length */
  MPI_Type_size(datatype, &size);
  EZTRACE_EVENT3 (FUT_MPI_START_SEND, count*size, dest, tag);
  int ret = libMPI_Send(buf, count, datatype, dest, tag, comm);
  EZTRACE_EVENT0 (FUT_MPI_STOP_SEND);
  return ret;
}
```

Figure 3.13 – Surcharge de la fonction *MPI_Send* par Eztrace

En ce qui concerne le standard OpenMP, on s'appuie sur une implémentation particulière : GNU OpenMP (GOMP). Etant donné que le standard décrit les directives

de compilation qui annotent le code, les fonctions appelées à chaque rencontre de telles annotations dans le code font partie des mécanismes internes à l'implémentation de la bibliothèque OpenMP. Ainsi nous savons que la directive `omp parallel` va générer l'appel à la méthode `GOMP_parallel_start(...)` par le thread principal, qui créera les threads de son équipe pour se répartir l'exécution du code imbriqué dans la section parallèle (référencé par le pointeur de fonction passé en paramètre). Il en va de même pour toutes les autres annotations OpenMP : synchronisations, opérations de réduction, etc. Eztrace peut facilement décrire le schéma d'exécution des programmes OpenMP en précisant les moments pendant lesquels un thread a commencé et fini la tâche qu'on lui avait incombé. Par exemple, le motif *Fork and join* du modèle OpenMP ressort clairement grâce à ces traces.

3.4.1.2 Les directives OpenMP

Nous référençons dans cette partie l'ensemble des méthodes de l'implémentation GNU OpenMP interceptées par Eztrace, et nous décrivons pour chacune les informations récupérées.

libGOMP_parallel_start(...) : Cette fonction est appelée à chaque rencontre d'une nouvelle section parallèle. Les paramètres contiennent un pointeur vers la sous-fonction à exécuter (c'est-à-dire le code contenu dans la région parallèle qui suit la directive OpenMP), un pointeur vers une structure utilisée pour transmettre les données vers et depuis la sous-fonction, ainsi que le nombre de threads qui se partageront le travail. Eztrace enregistre le début de la section parallèle par le thread principal ainsi que la création (ou le réveil) des autres threads qui vont composer l'équipe. Tous les threads de la section passent dans l'état actif.

libGOMP_parallel_loop_{static, runtime, dynamic, guided}_start(...) : C'est la fonction liée à la rencontre d'une section destinée à paralléliser une boucle de calcul. Le nom de la fonction varie selon la politique de distribution des indices de boucle aux threads. Pour le moment, les informations récoltées par Eztrace sont les mêmes que celles d'une simple section parallèle.

libGOMP_parallel_end(...) : Cette méthode est appelée dès qu'une section parallèle se termine, et ce par tous les threads de la section. Chacun de ces threads attend que tous les autres aient fini leur travail, c'est-à-dire qu'ils appelent tous `libGOMP_parallel_end`. Au terme de cette barrière de synchronisation, tous les threads, mis à part le thread principal, basculent dans l'état endormi : ils sont en attente d'être réveillés pour une potentielle réutilisation lors de la prochaine section parallèle. Eztrace passe tous les threads de l'équipe, sauf le thread principal, dans l'état inactif.

libGOMP_critical_start(...) : La fonction est exécutée dès qu'un thread rencontre la directive `critical`. Celle-ci agit comme un verrou de synchronisation. Si un autre thread de l'équipe est actuellement dans cette même région critique, alors le premier thread se bloque jusqu'à l'autre en soit sorti. S'il n'y aucun thread dans la section critique, le thread exécute le code imbriqué dans la section. Le rôle d'EZTRACE est de faire passer ou non le thread en état bloqué, selon les cas décrits précedemment. Si le thread entre dans la section critique, cette information est également enregistrée par notre système de trace.

libGOMP_critical_end(...) : A la fin d'une section critique définie par la directive `critical`, cette fonction relâche le verrou de synchronisation lié, ce qui permet à un autre thread, potentiellement en attente d'y entrer. EZTRACE enregistre cet état de fait et l'état du thread passe du statut *En section critique* au statut *Actif*.

Bien que l'ensemble des directives de compilation OPENMP ne soient pas interceptées par EZTRACE, celles qui sont interceptées sont suffisantes pour couvrir de nombreuses applications. Pour s'en convaincre, il suffit de lire les résultats de l'outil OPENMPSPY [PKMW11] dont les auteurs ont réalisé une étude empirique sur une quarantaine d'applications du domaine scientifique afin de ressortir des statistiques concernant notamment les mots-clés OPENMP utilisés. Sur l'ensemble de ces applications analysées, c'est la directive `critical` qui reste le plus employée pour réaliser des synchronisations. Quand à la construction `task`, apparue récemment dans le standard, elle reste encore peu utilisée mais nul doute qu'elle sera de plus en plus présente. Le logiciel EZTRACE prendra alors en compte cette construction ainsi que les directives qui y sont liées, et ce dans un futur proche.

3.4.1.3 Les primitives MPI

MPI_Init/MPI_Init_thread : Appelée en début de programme par chaque processus, la primitive d'initialisation MPI alloue les différentes structures nécessaires au bon fonctionnement du processus. EZTRACE l'utilise pour enregistrer la date de création du processus et son identifiant dans le communicateur global. Par ailleurs, c'est dans cette fonction que EZTRACE définit le fichier de traces où enregistrer les événements : il en existe un par processus MPI pour minimiser les problèmes de synchronisation.

Les communications point-à-point : L'ensemble des méthodes de communications point-à-point sont appelées par l'application pour envoyer un message d'un processus à un autre. Ils différent de par leur mode de communication : bufferisé, bloquant, non-bloquant ou synchrone. Chaque message contient un certain nombre de données d'un type particulier (caractère, entier, flottant, etc.). EZTRACE récupère la taille du message, l'identifiant de l'émetteur, celui du destinataire, l'étiquette (ou tag) et le communicateur.

Ces informations permettront ensuite de faire correspondre chacun des envoi à une réception. Par ailleurs le processus passe dans l'état *en émission* ou *en réception* pendant ce temps. Les appels aux primitives de test de complétion sont également mesurées et enregistrées.

Les communications collectives : Ces fonctions permettent de transférer un jeu de données depuis un ou plusieurs processus vers un ou plusieurs processus. De la même façon que pour les communications point-à-point, les différentes informations de quantité de données, d'émetteurs et de récepteurs est enregistrée par EZTRACE pour déterminer par la suite les échanges de données produits par ces communications collectives.

Les communications unilatérales : Ces transferts permettent d'envoyer ou de récupérer messages en ne faisant intervenir qu'une seule entité (émetteur ou récepteur), contrairement aux communications point à point. EZTRACE collecte également les identifiants des deux processus concernés ainsi que la quantité de données transmises. La différence avec une communication point à point tient dans le fait qu'il n'y a pas de correspondance à faire avec un autre appel sur le processus distant.

MPI_Finalize : Cet appel permet de garantir à l'application que chaque processus a exécuté son travail avant de terminer l'un d'entre eux. Pour EZTRACE cela signifie la fin de l'exécution du programme.

MPI_Barrier : La fonction synchronise tous les processus du communicateur. L'état des processus dans EZTRACE passent de l'état *Actif* à l'état *Bloqué* tant que l'ensemble des processus n'a pas effectué cet appel.

MPI_Comm_{size, rank} : Ce sont uniquement des fonctions d'informations destinées à l'application. EZTRACE les enregistre mais ne les utilise pas.

Au final, nous disposons des informations sur l'ordonnancement des threads du modèle en mémoire partagé (OPENMP) et des informations de communication du modèle distribué (MPI). Grâce à EZTRACE, elles sont récoltées en même temps et rassemblées dans un même fichier de trace. C'est un grand avantage lorsque l'on désire comprendre le déroulement d'un programme utilisant un modèle de programmation hybride. Souvent chaque modèle nécessite un outil de trace et de visualisation spécifique. Dans notre cas, les interactions entre les deux types d'entités (processus et threads) se voient directement grâce à la modélisation des ces séries d'événements.

3.4.2 Modéliser l'exécution d'un programme

La deuxième phase du mécanisme de trace de EZTRACE consiste à analyser les informations recueillies, dites « post-mortem», puis à les interpréter pour donner en sortie

un graphique représentatif du comportement des processus et des threads durant l'intégralité de l'exécution du programme. À cette fin, nous profitons du fait que les traces enregistrées par le logiciel EZTRACE sont convertibles au format approprié. Ainsi, la commande `eztrace_convert` lit un ensemble de traces issues d'un ou plusieurs processus d'une même application et les rassemble en un seul fichier de type *Paje*. PAJE est un environnement de visualisation interactif qui peut gérer une grande quantité de traces de manière efficace. Ce format est utilisé par l'outil de visualisation VITE qui va afficher le diagramme de Gantt de l'application. Nous obtenons alors une représentation temporelle de l'exécution des programmes. Dans notre cas, où ces programmes emploient les deux modèles de programmation MPI et OPENMP nous obtenons le diagramme d'ordonnancement des threads à l'intérieur des processus MPI ainsi que le schéma de communication entre ces processus.

3.4.2.1 Ordonnancement

La représentation graphique des traces d'exécution d'une application nous permet d'observer l'ordonnancement des threads OPENMP qui s'exécuteront en concurrence au sein de chaque processus. Grâce aux informations récoltées par FXT nous connaissons les dates auxquelles les threads OPENMP vont être créés ou réveillés et celles auxquelles les threads termineront les tâches qui leur sont dévolues. De cette manière, le schéma de type *Fork and join* du paradigme OPENMP apparaît clairement. Les parties où tous les threads travaillent et celles où ils sont inactifs s'enchaînent sur le diagramme. Cela signifie que nous sommes capables de dire si l'application a passé plus de temps à exécuter du code séquentiel que du code parallèle ou non. De plus, nous voyons si les threads finissent à peu près au même instant leur travail ou s'il existe des différences notables entre les temps de terminaison. Si c'est le cas, cela traduit un déséquilibre de charge et selon la proportion de ces différences l'impact sur les performances globales de l'application sont plus ou moins importants. La figure 3.14 donne le diagramme d'exécution d'un programme OPENMP affiché par VITE sur un processeurs à quatre cœurs. Ce programme se compose de trois boucles de calcul identiques. La première et la troisième sont parallélisées grâce à une directive OPENMP et la seconde est effectué séquentiellement. Chaque ligne du dessin représentent l'état d'un thread du programme et leur couleur varie en fonction de l'état du thread au cours du temps. Dans un premier temps on voit sur le dessin que seul le thread principal est actif : il s'agit de l'initialisation du programme. Ensuite le thread lance la création des threads de son équipe : les flèches montrent l'action de création. Les 4 threads passent en section parallèle pour effectuer les tours de boucle qu'on leur a affecté. Au terme de son travail chaque thread passe en état *Bloqué* et attend que les autres aient fini la leur. Dès lors la fin de la section parallèle est annoncée par les flèches partant de chaque thread vers le thread principal. Ce dernier repasse alors en état *Actif* et continue sur une tâche séquentielle. On voit qu'il n'atteint la prochaine boucle parallèle que plus tard (après avoir réalisé la boucle séquentielle) et qui se présente de la même façon que la première.

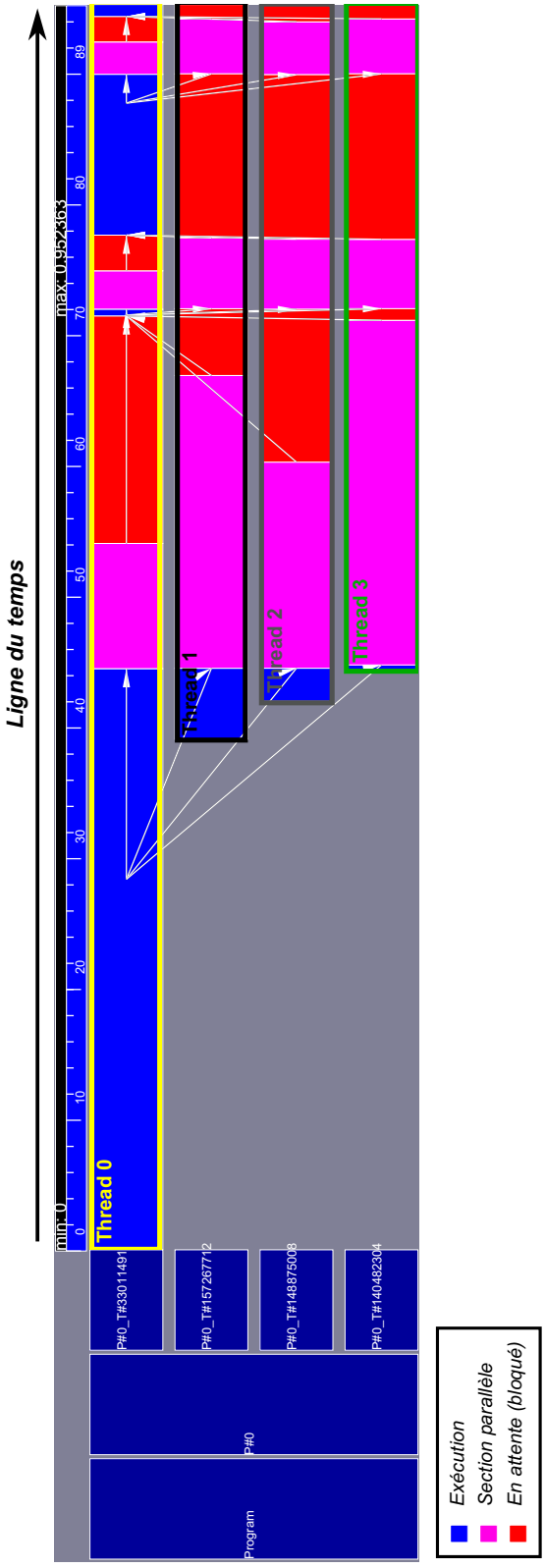


FIGURE 3.14 – Visualisation des traces d'exécution d'un programme OPENMP

3.4.2.2 Schéma de communication

De la même façon que pour les threads du modèle OpenMP, la représentation graphique des traces MPI donne une image claire de l'activité des processus et de leurs interactions. Étant donné que tout appel aux méthodes de l'interface de programmation est intercepté et référencé par Eztrace, le diagramme dessiné par Vite montre toutes les communications entre les processus MPI ainsi que la durée de chacune d'entre elles. Chaque communication est représentée par une flèche partant d'un point de la frise d'état d'un processus vers un point de celle d'un autre processus. L'état des processus (en cours de communication ou non) est indiqué par une couleur spécifique à chaque cas. La figure 3.15 montre un exemple de visualisation d'un programme MPI. Il s'agit d'une série d'échanges de messages entre deux processus (*ping-pong*), chaque transmission est précédée d'une boucle de calcul. Les périodes de calcul et les périodes d'échange de messages s'alternent et il est facile de déterminer à vue d'œil si le temps que passe l'application à dialoguer avec la bibliothèque MPI est plus ou moins important que celui qu'il passe à effectuer des calculs.

En plus de ces informations visuelles, le module de statistiques d'Eztrace fournit un jeu d'informations concernant la bibliothèque MPI. Nous avons accès au nombre de messages transmis entre les processus et grâce à une matrice de communication (indicée par les identifiants des processus MPI [2]) nous savons quels processus dialoguent plus (ou moins) que les autres. Cette matrice peut également contenir, non plus la quantité de messages, mais le volume de données échangées. Il s'agit d'autant de critères susceptibles de nous aiguiller pour le choix du placement et du dimensionnement optimaux.

3.4.3 Comprendre les performances

Comprendre les raisons d'échec ou de succès d'une répartition particulière est essentiel dans le cadre des applications de calcul scientifique. Il faut être capable de mettre en évidence les sections de codes critiques pour les performances pour ensuite les identifier sur la représentation graphique des traces d'éxécution. Dans cette sous-partie, nous proposons de référencer les problèmes que peuvent rencontrer les applications qui utilisent un modèle hybride de progammation lorsqu'elles s'exécutent et d'en expliciter les causes probables. Nous illustrerons ceci au travers de quelques cas typiques.

3.4.3.1 Le runtime OpenMP

Le support exécutif OpenMP est présent dans chaque processus MPI pour répartir le travail entre les threads et les ordonnancer sur la machine. Comme nous l'avons vu précédemment l'une des premières approches de programmation mélangeant les deux modèles MPI et OpenMP consiste à créer un seul processus par nœud et de s'en remettre aux threads OpenMP pour exécuter le code applicatif sur l'intégralité des

2. Ces identifiants correspondent aux rangs des processus dans le communicateur *MPI_COMM_WORLD*

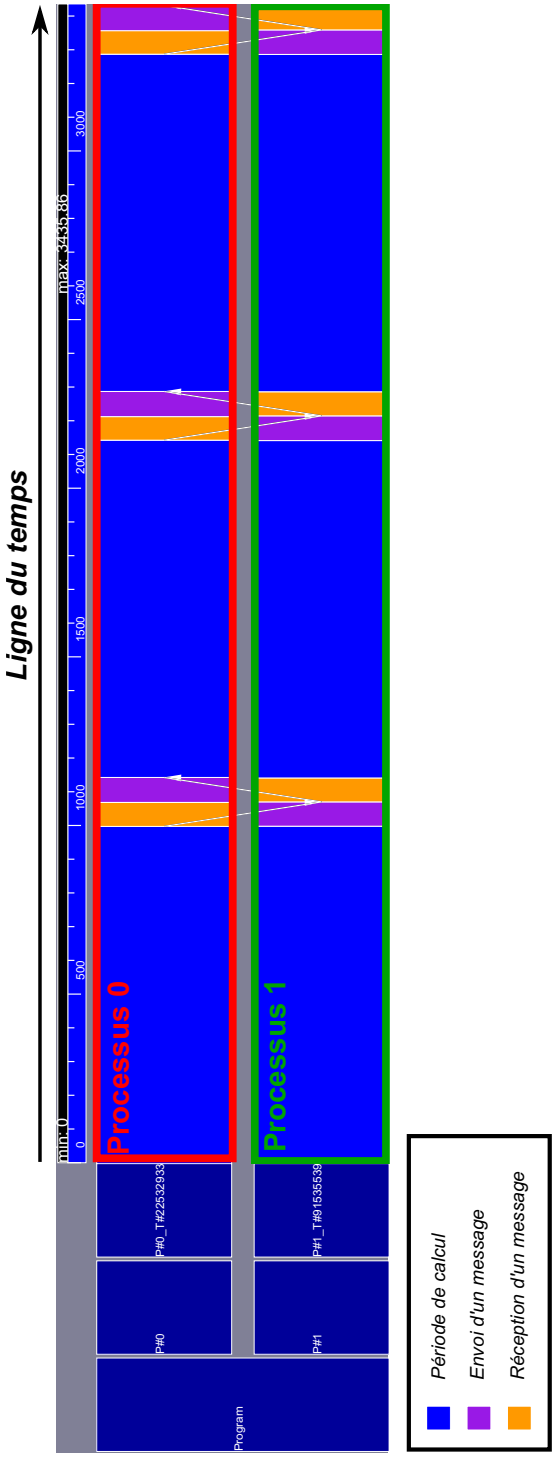


FIGURE 3.15 – Visualisation des traces d'exécution d'un programme MPI

unités de calcul du nœud. Bien que cela semble être une bonne idée, cette décision peut s'avérer dommageable pour les performances. En effet, lorsque le support exécutif doit ordonnancer tous les threads d'une section parallèle il ne connaît pas avec précision les affinités qui peuvent exister entre eux. Ce critère n'entrera pas en compte lors du placement des threads sur les unités de calcul. Or, la structure hiérarchique des machines modernes où les facteurs de distance sont très importants nous oblige à les considérer avec attention. De plus, sur des machines à mémoire partagée de grande dimension, la gestion des threads devient problématique pour le support OpenMP. La création de nouveaux threads à la rencontre de la première section parallèle et leur réveil pour les sections suivantes sont beaucoup plus coûteuses en temps. Le même effet se produit pour les synchronisations des threads, telles les barrières implicites à la fin de chaque section parallèle qui vont mettre à mal les performances globales de l'application. Avec un nombre de threads élevé, le temps passé en moyenne par chaque thread dans la fonction de synchronisation devient plus important, ce qui retarde d'autant la reprise de l'exécution du code applicatif. En effet, lors de la rencontre d'une barrière de synchronisation, les threads passent en attente de tous les autres threads de l'équipe. Plus la taille de cette équipe est importante plus on augmente les chances de trouver des threads terminant moins rapidement que les autres, surtout dans le cadre des architectures fortement hiérarchiques où un accès mémoire distant s'avère très coûteux (les threads partageant des données contiguës, ces accès peuvent être récurrents).

3.4.3.2 La durée des communications

Un second critère déterminant est la valeur relative du temps passé à transférer des données entre les processus. Lorsque celui-ci est trop important cela peut s'avérer fortement dommageable pour l'application. Dans nombre d'algorithmes utilisant les deux paradigmes de programmation (MPI ou OpenMP) plusieurs étapes de calculs exécutées en parallèle s'enchaînent et sont entrecoupées de communication servant à transmettre les résultats locaux à tous les autres (ou au moins à un sous-ensemble de processus). De ce fait, de l'efficacité du transfert réalisé après une étape vont dépendrent le début des calculs de l'étape suivante. L'impact est immédiat sur les performances applicatives.

3.4.3.3 Exemple d'exécution problématique

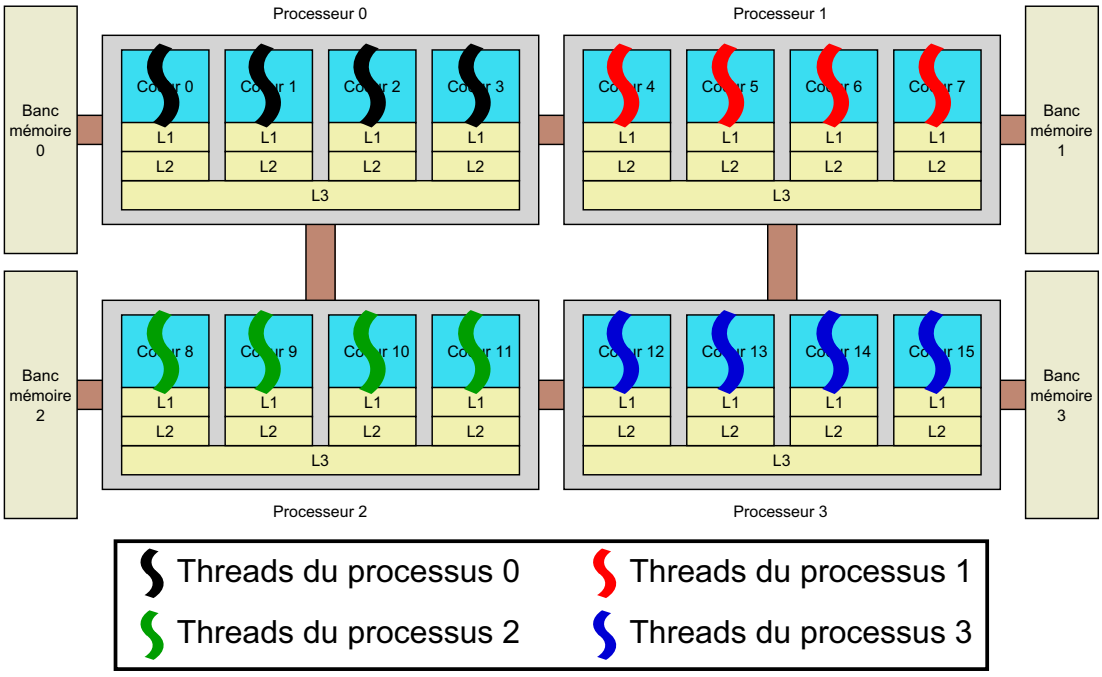
Pour illustrer ce genre de problématique, nous avons choisi de visualiser l'exécution d'un programme de test que nous appelerons *Alpha* et qui emploie un modèle hybride de programmation. Il présente six boucles parallèles OpenMP qui réalisent plusieurs opérations (multiplications, additions, soustractions) pour modifier les valeurs d'un tableau de grande taille à chacun des tours de boucle (Les threads se répartissent les indices de boucle et chaque thread modifient les valeurs d'une section contiguë de ce tableau). Entre les deux premières boucles une communication MPI s'effectue entre chaque paire de processus.

La plate-forme d'exécution est une machine à quatre processeurs AMD *Opteron* quadri-cœurs. Chaque processeur dispose d'un banc mémoire NUMA auquel il a un accès privilégié par rapport aux autres. Pour cette expérimentation, nous fixons le nombre de processus à quatre. Etant donné qu'il y a en tout 16 cœurs, cela signifie que chaque processus crée quatre threads à la rencontre d'une section parallèle OpenMP. Nous décidons de comparer les performances et le comportement de l'exécution du programme pour deux différentes distributions des processus et des threads sur les multiples unités de calcul. Elles sont toutes deux présentées sur la figure 3.16. La première consiste à rassembler sur le même banc NUMA les threads d'un même processus MPI pour profiter de la localité mémoire. La seconde, quant à elle, répartit les threads sur les différents bancs mémoire : chaque processus dispose d'un thread sur chaque banc NUMA. Bien que cette dernière répartition puisse sembler pertinente au sens de l'amélioration des performances dans certains cas (pour favoriser le débit mémoire par exemple), nous supposons que le temps d'exécution sera beaucoup plus important pour celle-ci. En effet, tous les threads accèdent au tableau de l'espace d'adressage de leur processus simultanément et la machine est pour ainsi dire « chargée » : c'est-à-dire que la totalité des cœurs est occupée.

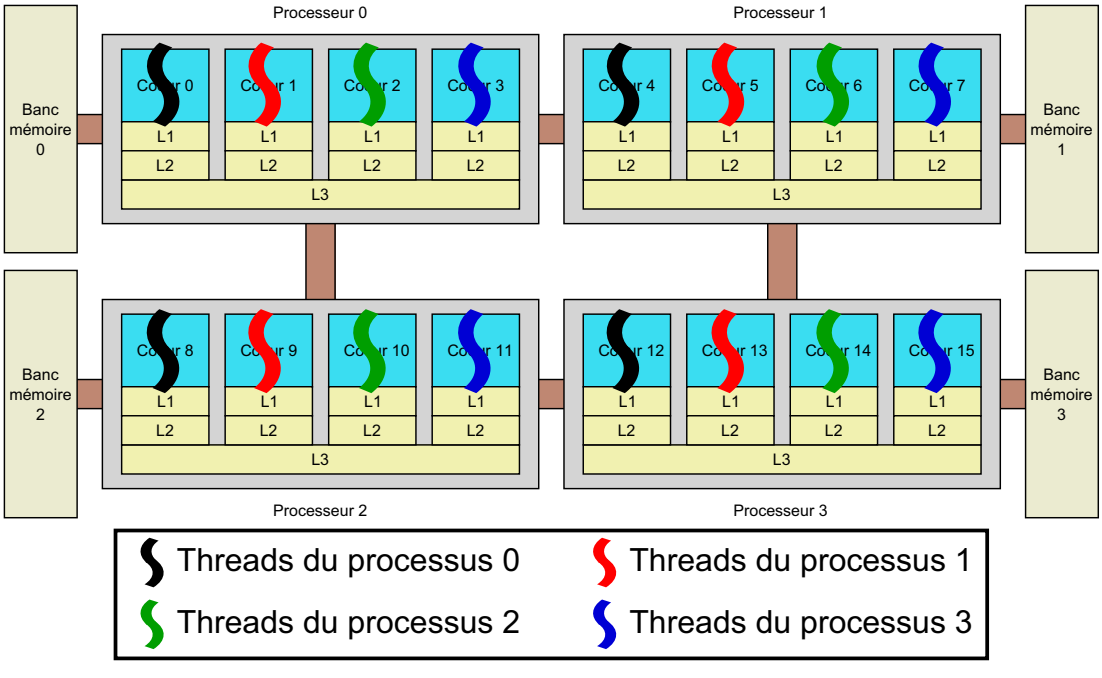
Les temps de calcul obtenus confirment notre supposition puisque celui de la deuxième répartition équivaut au double de celui de la répartition privilégiant la localité mémoire. Ces résultats s'expliquent facilement grâce aux traces d'exécution présentées en figure 3.17. En effet, pour chaque processus MPI, on y voit clairement l'initialisation et la terminaison des sections parallèles ainsi que les périodes d'activité des différents threads dans ces sections parallèles, représentées par les zones de couleur rose. Lorsqu'un thread a terminé la tâche qui lui a été donnée il passe en état bloqué (zones de couleur rouge) et attend la terminaison des autres threads du processus. Alors que dans le cas de la répartition A, les threads terminent à peu près au même moment, le schéma d'exécution de la répartition B montre un autre comportement. Certains threads terminent leur travail bien avant les autres et doivent attendre que ces derniers aient également complété le leur. De plus, si l'on compare le temps minimal d'exécution du travail d'un thread entre les deux répartitions, il est plus important pour B que pour A. Ces problèmes de performance s'expliquent facilement : dans la répartition A, les threads d'un même processus partagent une même mémoire cache (de niveau 3), ce qui n'est pas le cas pour B, où les threads génèrent de nombreux défauts de cache et s'échangent régulièrement les pages mémoires où sont stockées les valeurs des tableaux auxquels ils accèdent. Par ailleurs, les tableaux étant alloués sur des bancs mémoires particuliers, les threads placés sur les cœurs du même noeud que ces mémoires sont privilégiés par rapport aux autres.

3.5 Discussion

L'évolution des architectures des plates-formes de calcul a révélé les limites des modèles de programmation populaires tel que MPI ou OpenMP. Il est très difficile, voire parfois impossible, d'en tirer la quintessence en employant un unique modèle. À notre

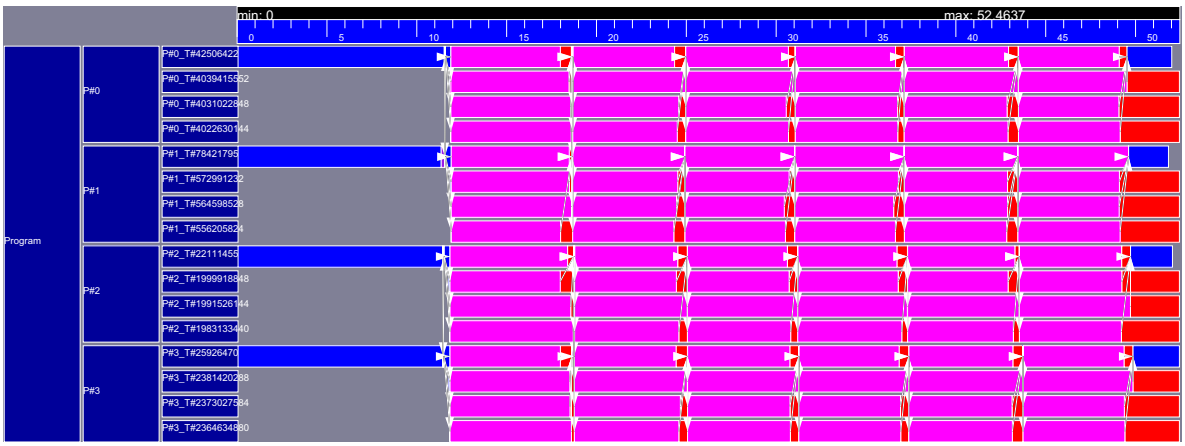


(a) Répartition A

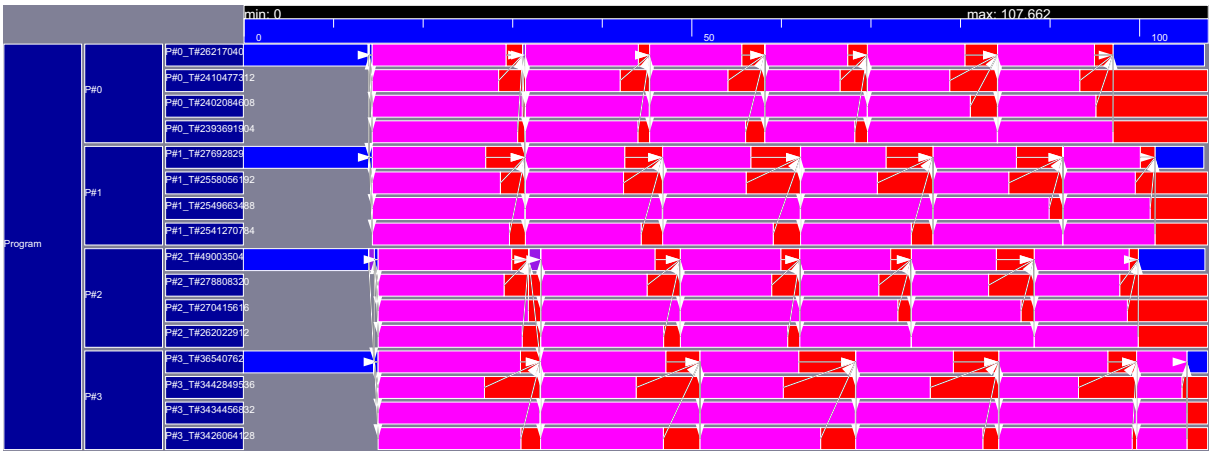


(b) Répartition B

FIGURE 3.16 – Les deux répartitions évaluées sur une architecture à quatre processeurs quadri-cœurs pour le test *Alpha*



(a) Répartition A



(b) Répartition B

FIGURE 3.17 – Visualisation de l'exécution du programme de test Alpha

sens, le modèle de programmation hybride offre un début de réponse qui étend l'expressivité de ces paradigmes pour s'adapter aux machines modernes. Cependant, afin d'atteindre ces objectifs de performance, il convient de contrôler finement le parallélisme fourni par chaque modèle. C'est pourquoi nous proposons un outil capable de déterminer la combinaison de parallélisme optimale pour une application donnée selon la topologie matérielle de l'architecture sous-jacente. En prime, l'outil retranscrit le schéma d'exécution et de communication de l'application pour aider à la compréhension des performances et fournit des statistiques sur l'exécution. L'utilisateur peut alors comprendre plus précisément pourquoi une combinaison particulière n'est pas efficace et tenter, si cela est possible, d'améliorer la parallélisation hybride dans l'application. Certaines parties de notre solution sont encore à la charge de l'utilisateur, notamment l'interprétation des résultats graphiques pour détecter les contentions, les problèmes de synchronisation, une mauvais répartition de charge, etc. Néanmoins, nous nous dirigons vers une solution entièrement automatisée dont l'objectif est de déterminer le meilleur moyen d'exécuter une application mélangeant deux modèles de programmation.

Pour aller un peu plus loin, notre solution pourrait prendre en considération d'autres critères pour raffiner la compréhension de l'application, voire même écarter certaines combinaisons du parallélisme hybride. À terme, il s'agirait de prendre la décision à l'avance, plutôt que d'évaluer différentes exécutions, coûteuses en temps, et parfois non envisageables pour certaines applications ou architectures : sur la machine TERA100 par exemple, le volume de traces à rassembler peut être conséquent. Dans un premier temps, nous pouvons envisager de récupérer les informations sur les défauts de cache : une quantité trop importante, proportionnellement aux autres combinaisons, signifierait que le nombre de threads par processus est trop élevé. À cette fin, nous envisageons d'intégrer les compteurs matériels PAPI pour augmenter la précision de notre démarche. Pour aller plus loin, la connaissance des besoins en quantité mémoire des applications,voire même le schéma d'accès, en fonction du nombre de processus nous permettrait de faire une correspondance avec le matériel mémoire partagé entre les unités de calcul et d'en déduire une sélection plus fine de combinaison des modèles. Le développement de micro-benchmarks, est également envisagé pour assister notre outil dans ses choix. Par une série d'exécution et d'évaluation des performances et limites des implémentations MPI, notamment au niveau du nombre de processus, l'outil pourrait décider de retirer des combinaisons qui ne tiendraient pas la charge.

Actuellement, notre solution s'est consacrée aux modèles MPI et OpenMP. Nous envisageons d'appliquer la même approche sur d'autres modèles hybrides. La tendance actuelle est de mélanger le standard MPI avec un paradigme de programmation en mémoire partagée : les Pthreads, le langage Cilk, Intel TBB, etc. Nous pensons, comme nous l'avons constaté pour OpenMP, qu'il est audacieux de supposer que le support exécutif du modèle à mémoire partagée va gérer plus efficacement les calculs si on lui confie l'intégralité de chaque nœud. Énormément de critères rentrent en ligne de compte et souvent c'est une valeur médiane de dimensionnement qui va garantir les meilleures performances à l'application.

Chapitre 4

Evaluation

Nous avons mené différentes expériences destinées à valider notre proposition sur les modèles hybrides de programmation et nous les présentons dans ce chapitre. Tout d'abord, grâce à plusieurs benchmarks, nous évaluons l'impact des paramètres tels le positionnement des processus et threads sur l'architecture cible, ainsi que leurs dimensionnements respectifs. À l'aide de notre environnement, nous expérimentons les répartitions possibles sur plusieurs systèmes pour les applications NAS développés par la NASA.

4.1 Plate-formes expérimentales

Nous détaillons ici les caractéristiques matérielles des systèmes de calcul qui nous ont servi de plates-formes d'expérimentation : deux machines de calcul, dénommées *Kwak* et *Bertha*, ainsi qu'une grappe de calcul du projet Plafrim.

4.1.1 Configuration matérielle

4.1.1.1 Kwak

La machine *Kwak* est constituée de quatre nœuds NUMA. Chacun de ces nœuds détient un processeur AMD *Opteron* à quatre cœurs de type *Barcelona* cadencés à 1,9 GHz et un banc mémoire de 8 Go. Les cœurs d'un processeur disposent de deux mémoires caches privées de 64 ko et de 512 ko (de niveau 1 et 2) et partagent une troisième mémoire cache de 2048 ko (de niveau 3). Des liens *HyperTransport* connectent les nœuds NUMA entre eux comme présenté en figure 4.1. Le temps d'accès à une donnée par un cœur dépend de la position relative de la mémoire contenant la donnée et du cœur qui réalise l'accès. Par exemple, une requête d'accès en lecture ou en écriture à une donnée du banc mémoire 3 devra traverser deux liens *HyperTransport* et sera plus coûteux qu'un accès à une donnée stockée en mémoire locale.

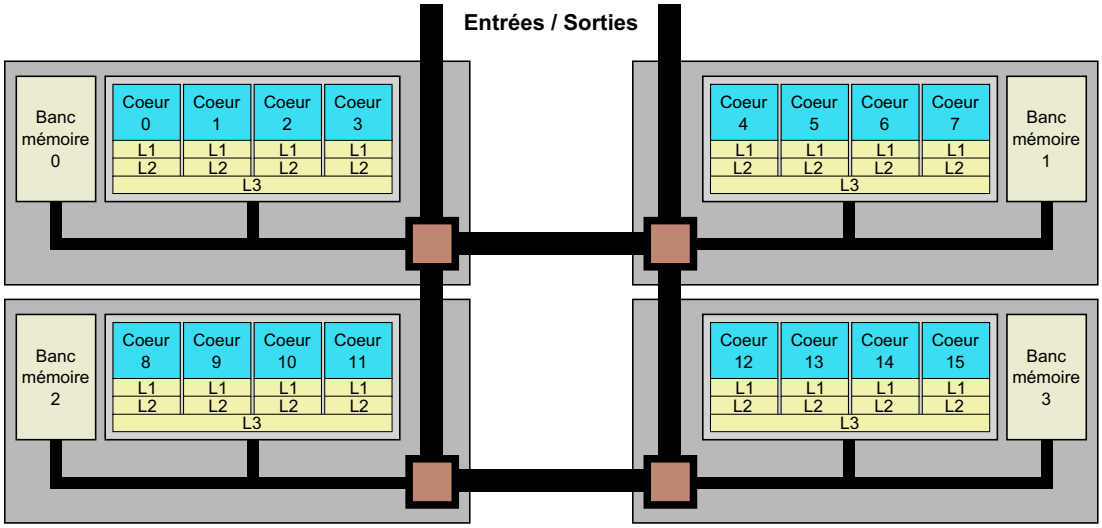


FIGURE 4.1 – Architecture de la machine *Kwak*

4.1.1.2 Bertha

La machine *Bertha* est un assemblage de quatre serveurs IBM *x3950M2*. Chaque serveur est composé d'un banc mémoire de 48 Go de mémoire RAM et de quatre processeurs INTEL *Xeon Dunnington X7460* à six cœurs cadencés à 2,67 GHz. Chaque cœur accède à une mémoire cache privée de 32 ko et de niveau 1. Les mémoires caches de niveau 2 (de 3072 ko) sont partagées entre chacune des trois paires de cœurs et l'ensemble des cœurs partagent 16 Mo de mémoire cache de niveau 3. Un processeur est détaillé en figure 4.2 et la figure 4.3 présente le système complet. Nous avons donc affaire à une architecture à quatre nœuds NUMA, comme pour la machine *Kwak*, mais offrant un total de 96 cœurs.

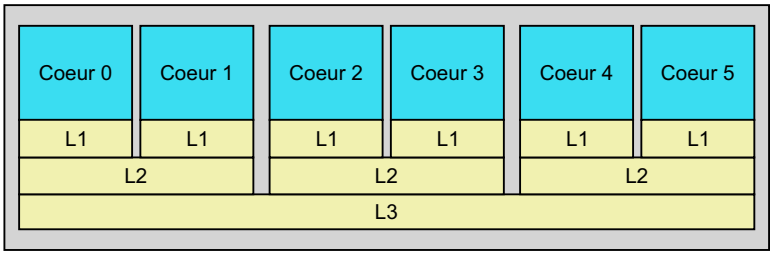


FIGURE 4.2 – Processeur INTEL *Xeon Dunnington X7460* à six cœurs

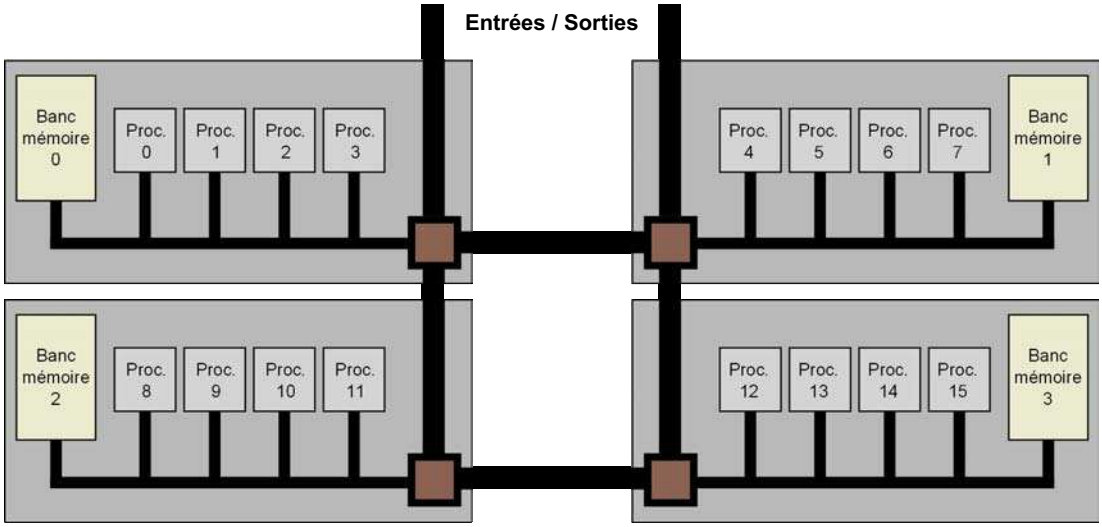


FIGURE 4.3 – Architecture de la machine *Bertha*

4.1.1.3 Fourmi

Nous désignons sous le terme *Fourmi* une grappe de calcul présente dans l'infrastructure de la plate-forme interactive du projet *Plafrim* [Pla]. L'objectif de ce projet est de mutualiser les ressources de calcul entre l'*Institut de Mathématiques de Bordeaux* (IMB), le *Laboratoire Bordelais de Recherche en Informatique* (LABRI) et le *Centre de Recherche INRIA Bordeaux - Sud Ouest*.

En tout, 68 nœuds composent la grappe *Fourmi* et sont inter-connectés par un réseau *Infiniband* de type QDR (*Quad Data Rate*) avec un débit nominal de 40 Gbits/s. L'agencement des nœuds est présenté en figure 4.5. Chaque nœud contient deux processeurs XEON quadri-cœur hyperthreadés *Nehalem X5550* (voir la figure 4.4). Un banc mémoire de 12 Go est attaché à chaque processeur. Les caches de niveau 1 et 2 sont privés à chaque cœur et ont respectivement une taille de 32 ko et 256 ko. Une mémoire cache de troisième niveau est partagée entre les cœurs d'une même puce et peut contenir 8192 ko de données.

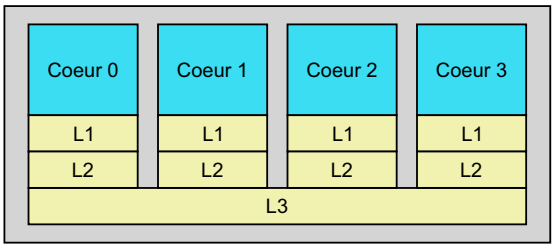


FIGURE 4.4 – Processeur INTEL *Xeon* à quatre cœurs hyperthreadés *X5550*

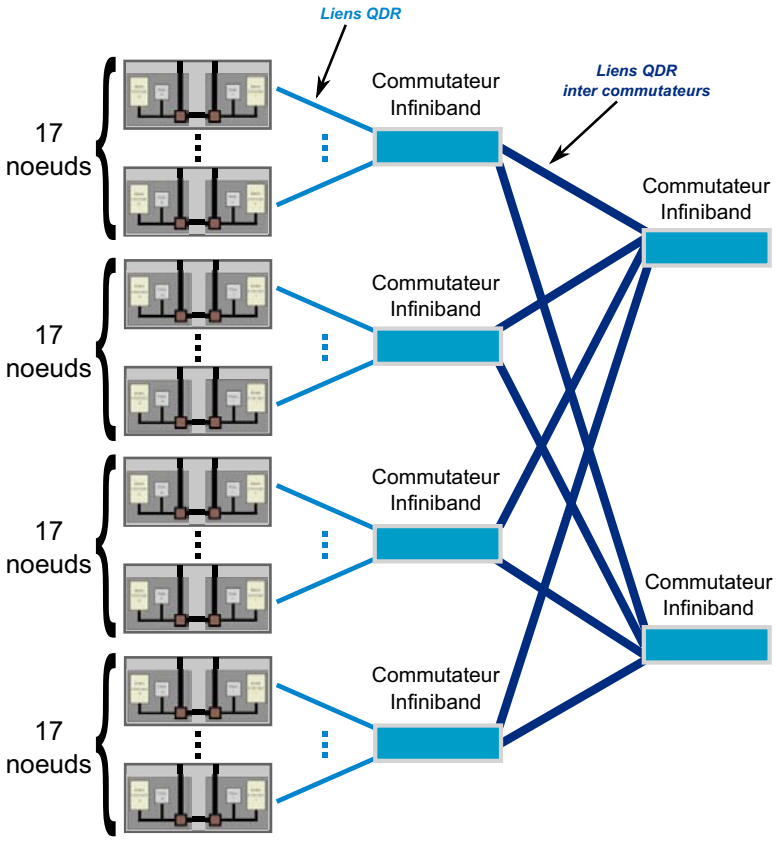


FIGURE 4.5 – Infrastructure de la grappe de calcul *Fourmi*

4.1.2 Comparaison logicielle

Afin de valider les résultats obtenus par les répartitions calculées grâce à notre environnement, nous avons choisi plusieurs bibliothèques de communication du standard MPI-2.

4.1.2.1 MPICH2

La bibliothèque MPICH2 est l'une des plus utilisées et constitue la base de nombreuses implémentations spécialisées du standard, que ce soit par rapport au réseau (MVAPICH2 pour INFINIBAND, MPICH-MX pour MYRINET), à l'architecture du processeur (INTEL MPI) ou des modèles de calculateurs (BLUEGENE, CRAY). La pile logicielle de MPICH2 est relativement complexe comme le montre la figure 4.6.Les développeurs ont choisi une structure par couches qui permet de mettre en place un pilote réseau à différents niveaux, plus ou moins proches du matériel. Nous avons utilisé le *channel* NEMESIS [BMG07] qui est un sous-système de communication qui permet

de différencier les transferts intra-nœuds et les transferts inter-nœuds. Pour le premier cas il utilise des mécanismes de mémoire partagée et dans le second cas, des pilotes réseaux sont employés. Les expériences réalisées sur les machines *Kwak* et *Bertha* ont été réalisées avec la version 1.2.1 de MPICH2.

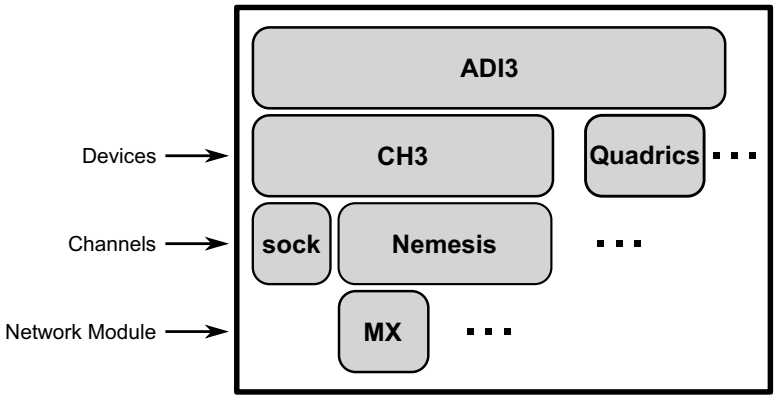


FIGURE 4.6 – Pile logicielle de MPICH2

4.1.2.2 Open MPI

Développée et maintenue par un consortium qui regroupe des instituts académiques, des organismes de recherche et des industriels, l'implémentation OPEN MPI est issue des multiples expériences apportées par de nombreuses bibliothèques de communication (FT-MPI, LA-MPI, LAM-MPI et PACX-MPI). La structure de l'implémentation est basée sur des composants décrits en figure 4.7. De la même façon que pour MPICH2, OPEN MPI dispose d'un composant pour la gestion des communications intra-nœuds par des mécanismes de partage mémoire : BTL SM (*Byte Transfer Layer Shared Memory*). C'est celui qui est utilisé pour les expériences sur les machines *Kwak* et *Bertha*. Pour les tests réalisés sur la grappe de calcul *Fourmi*, nous avons également utilisé le module BTL OPENIB pour le réseau INFINIBAND. La version de OPEN MPI employée était la 1.5.3.

4.1.2.3 MVAPICH2

MVAPICH2 est basée sur l'implémentation générique MPICH2 dont les développeurs se concentrent sur l'exploitation efficace des réseaux employant le RDMA comme INFINIBAND. MPICH2 a d'ailleurs cessé le maintien de l'exploitation de ce type de réseaux. Pour ces expériences nous avons utilisé la version 1.6 de MVAPICH2.

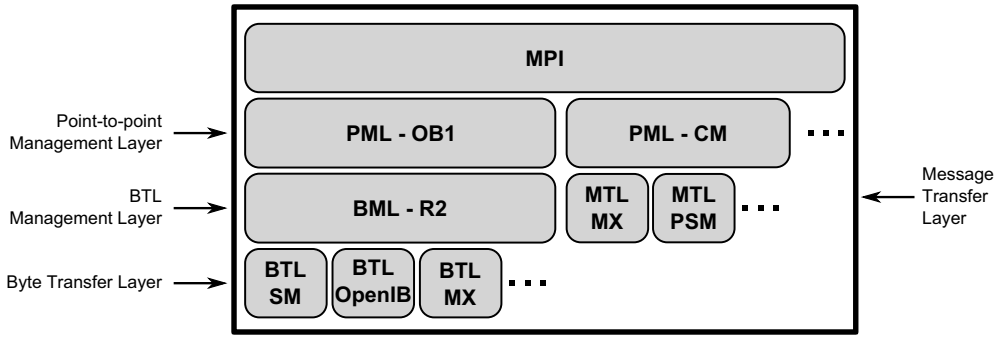


FIGURE 4.7 – Pile logicielle de OPEN MPI

4.2 Micro-benchmarks

Nous évaluons ici l'impact des politiques de placement et de dimensionnement sur des tests courts inspirés de la suite de microbenchmarks EPCC [BEA09] pour les codes de programmation hybrides. L'aspect concernant la consommation mémoire est également pris en compte.

4.2.1 Placement

Comme nous l'avons souligné dans le chapitre précédent, le placement des threads par rapport aux données sur lesquelles ils travaillent a une grande incidence sur les performances globales des applications. Et concernant les communications entre les processus, le positionnement de ces derniers affecte également le temps d'exécution des applications. Les expériences dont nous présentons les résultats ici-même ont pour objectif de mettre en évidence l'impact de ces facteurs dans le cadre d'une application utilisant un modèle de programmation hybride.

La figure 4.8 présente le schéma de calcul et de communication du microbenchmark que nous évaluons ici. Cela consiste en l'échange d'un vecteur de données (de taille T) entre deux processus. Ce vecteur est tout d'abord modifié par les threads du processus *0* (chaque thread modifie un sous-vecteur de taille $\frac{T}{nb\_threads}$, $nb\_threads$ correspondant au nombre de threads par processus) puis transmis au processus *1* qui se charge de lire ces données puis de les modifier pour enfin les retransmettre au processus *0*. Nous mesurons le temps à la fois pour la transmission du message avec les primitives MPI et pour les lectures et écritures effectuées par les threads OPENMP : le coût des communications inter-threads et celui des communications inter-processus sont tous deux pris en compte. Les mesures sont effectuées pour une centaine de tours de boucle et l'on calcule le temps moyen. Nous faisons varier la taille T du vecteur (et donc des messages envoyés) de 64 octets à 64 Mo.

Le deuxième benchmark, *Pingpong-Thread_Multiple*, a été mis en place pour évaluer

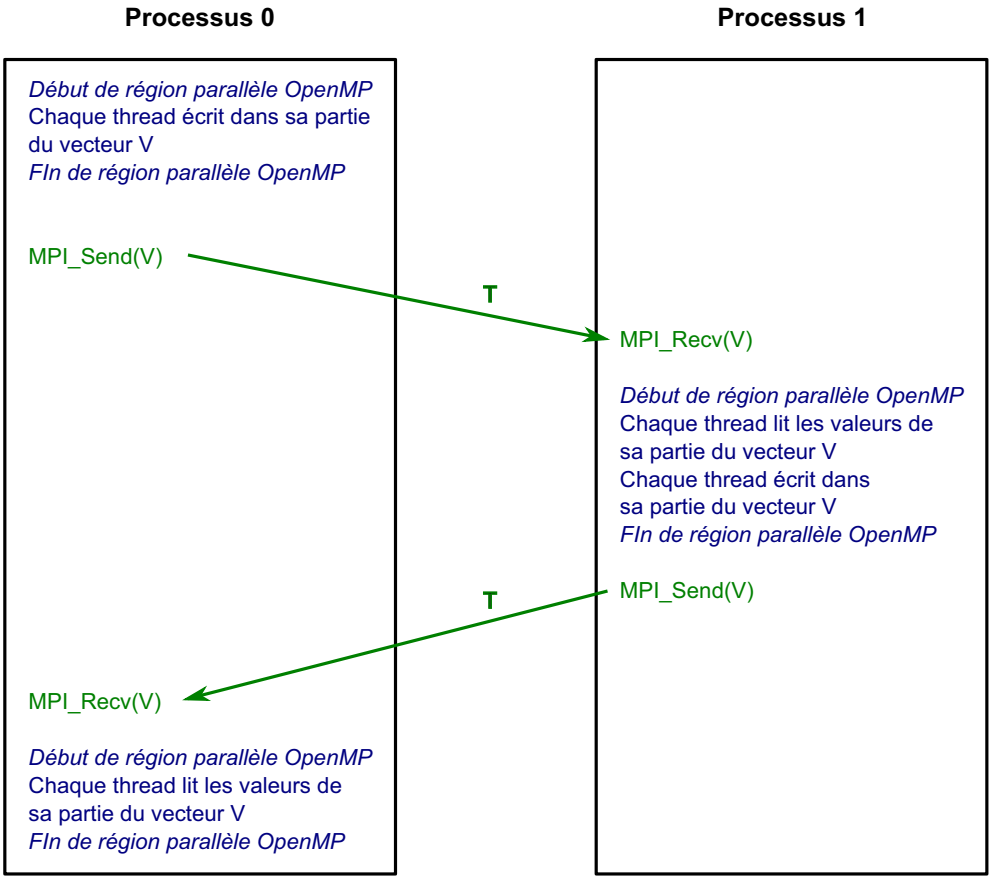


FIGURE 4.8 – Schéma de fonctionnement du microbenchmark *Pingpong-Thread_Single*

l'impact du placement lorsque plusieurs threads décident de réaliser des communications et des calculs de façon concurrente. Son schéma d'exécution est présenté en figure 4.8. En pratique, le processus *0* crée une section parallèle *OpenMP* dans laquelle chaque thread écrit des valeurs dans une partie d'un vecteur de taille T. Une fois que sa partie du vecteur a été mis à jour, ce thread envoie les valeurs qu'il a modifié à un thread du processus *1*. Ce dernier reçoit les données, les lit, les modifie pour finalement les transmettre au thread qui les lui avait transmis. Une fois ces données transmises, le thread du processus *0* les lit et arrive au terme de la section parallèle. Le programme termine lorsque tous les threads ont atteint ce point. Ainsi, $nb\_threads$ messages de taille $\frac{T}{nb\_threads}$ seront transmis du processus *0* au processus *1* et pareillement du processus *1* au processus *0*. Et l'ensemble du vecteur sera lu et modifié par chaque processus. De la même manière que précédemment, nous faisons varier la taille des données calculées et transmises de 64 octets à 64 Mo.

Les deux benchmarks de type *Pingpong* impliquent la création de deux processus MPI à l'intérieur desquels s'exécutent quatre threads OpenMP. En observant les résul-

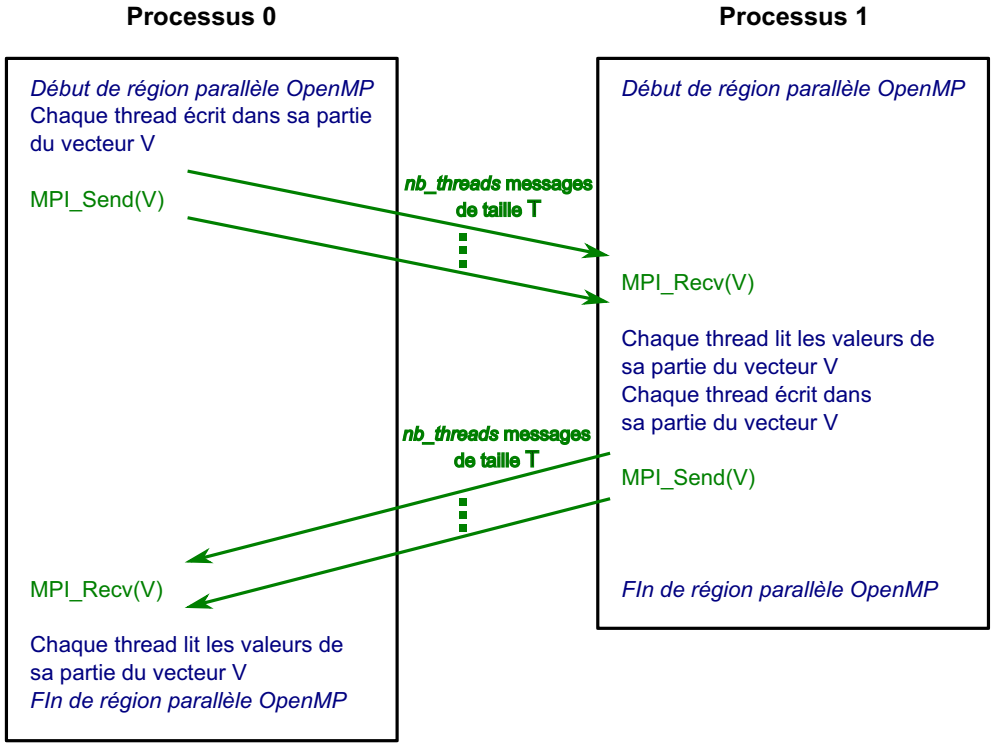


FIGURE 4.9 – Schéma de fonctionnement du microbenchmark *Pingpong-Thread_Multiple*

tats des temps d'exécution des ces deux programmes, nous pouvons mettre en évidence l'importance que revêt le placement des threads et des processus au sein des systèmes de calcul modernes. Pour exemple, la machine *Bertha* présente une architecture fortement hiérarchique sur laquelle nous avons choisi de tester trois différentes politiques de placement. La politique A_1 consiste à placer les 2 processus sur le même nœud NUMA afin qu'ils profitent de l'accès à un banc mémoire commun. À l'inverse, la politique A_2 éloigne les deux processus en les placant sur deux nœuds NUMA distincts. Quant à la politique A_3, elle répartit les threads de chaque processus sur chaque nœud NUMA. Nous espérons ainsi que les threads qui communiquent ensemble dans le test *Pingpong-Thread_Multiple* le fassent plus efficacement grâce au différents niveaux de cache et au banc mémoire qu'ils partagent. Toutefois la figure 4.11 nous montre que la meilleure politique de placement est celle où les deux processus sont sur le même nœud NUMA et la politique A_3 donne les plus mauvaises performances. Comme pour le benchmark *Pingpong-Thread_Single* (figure 4.10), les modifications effectuées par les threads sur le vecteur génèrent plus de trafic pour des accès distants et plus de défauts de cache, qui ne peuvent pas être compensés par la proximité des threads communiquant entre eux via MPI.

Les figures 4.12 et 4.13, montre que les différences sont moins prononcées entre la

politique qui rassemble les threads d'un même processus sur un nœud NUMA et celle qui les répartis sur l'ensemble des nœuds NUMA. Cependant elles sont toujours présentes et en faveur de la première politique.

Pour finir, les courbes de la figure 4.14 montrent que le placement des processus, en opposition à celui des threads, a aussi un impact sur les performances globales de l'application. En rapprochant les processus qui dialoguent entre eux, les communications sont plus rapides grâce aux mécanismes de transfert intra-nœud mis en place. C'est un paramètre à prendre en compte comme nous l'avons déjà souligné [MCO09].

4.2.2 Dimensionnement

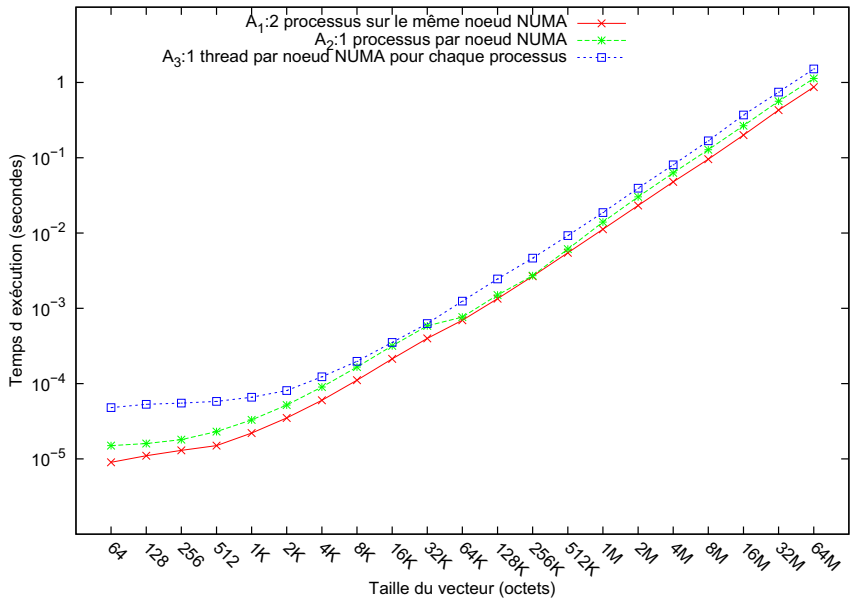
Dans le cadre d'un modèle de programmation hybride, le contrôle du parallélisme dans chacun des deux modèles est un élément important qui affecte directement les performances applicatives. Ce dimensionnement permet d'adapter la structure parallèle de l'application à l'architecture hiérarchique des sytèmes de calcul. C'est ce que nous cherchons à mettre en valeur au travers des expériences présentées ici. Pour chaque architecture, nous avons mis en évidence plusieurs distributions correspondant aux multiples niveaux de leur hiérarchie.

Le microbenchmark *Reduction* réalise une opération de réduction (en l'occurence, une addition) sur l'ensemble des processus MPI et récupère le résultat dans la mémoire du processus *0*. Etant donné que la norme OpenMP ne permet pas d'appliquer une opération de réduction sur un ensemble de tableaux de données, cette opération est réalisée « à la main ». Ainsi chaque processus détient $nb\_threads$ vecteurs d'une taille donnée T. Pour chaque indice i des vecteurs, les valeurs stockées à la position i des vecteurs sont additionnées et le résultat est affecté dans l'élément à la position i du premier vecteur. Pour cela, chaque thread effectue cette opération pour la partie du vecteur dont il a la charge. Une fois qu'ils ont tous terminé, chaque processus va transmettre son vecteur résultant pour qu'il soit additionné aux autres et le résultat sera stocké dans la mémoire du processus *0*.

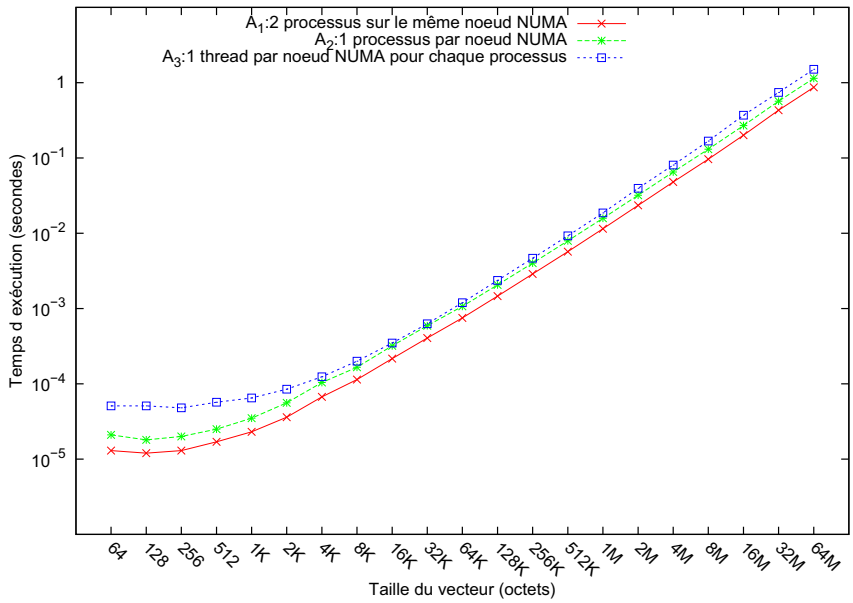
Un second microbenchmark réalise une évaluation du partage d'une donnée entre les threads et les processus. Le test *Broadcast* initialise un vecteur de données sur le processus *0* puis le transmet à tous les autres processus. Par la suite, tous les threads de tous les processus lisent l'ensemble des valeurs du vecteur.

Un dernier microbenchmark, dénommé *Barrier*, évalue l'impact des mécanismes de synchronisation fournis par les deux standards MPI et OpenMP. Ce programme crée tout d'abord une première barrière dans une section parallèle OpenMP pour synchroniser les threads entre eux, puis grâce au mot-clé `master`, le thread principal de chaque processus réalise un appel à la primitive `MPI_Barrier` pour synchroniser tous les processus. Une fois que tous les threads et tous les processus ont effectué la synchronisation, ils reprennent leur exécution et le programme termine.

Sur la machine *Bertha*, les résultats des expériences (figures 4.16 et 4.19) montrent que le meilleur ratio $\frac{nb\_processus}{nb\_threads}$ ne se retrouve pas parmi les cas extrèmes (c'est-à-dire

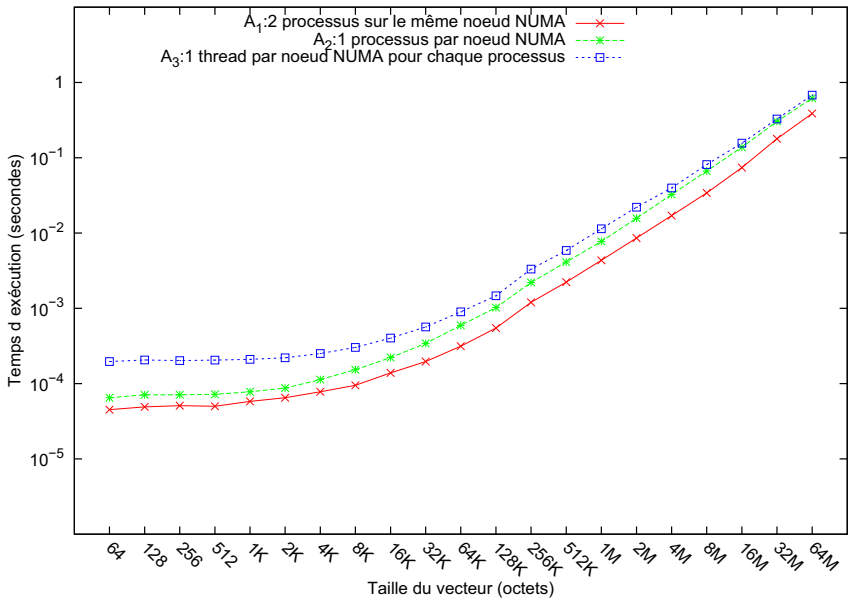


(a) MVAPICH

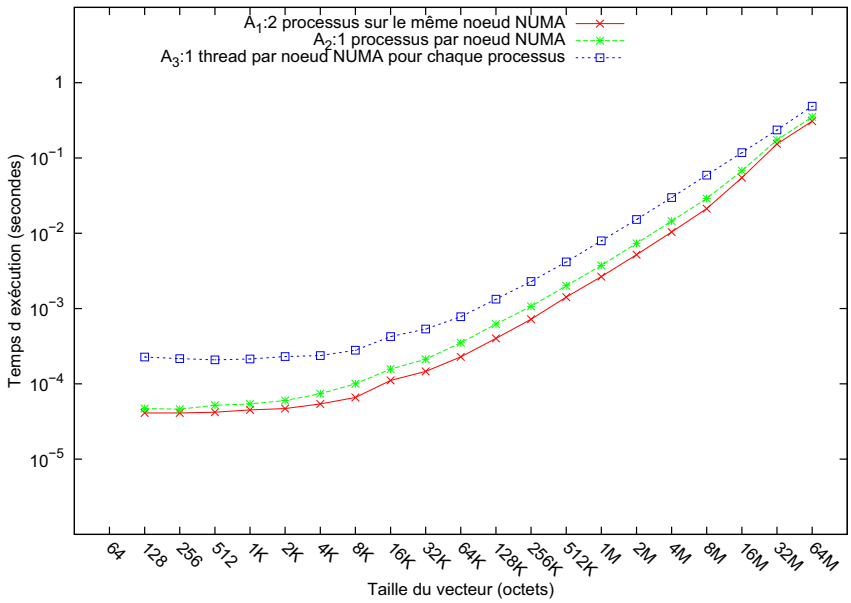


(b) OPENMPI

FIGURE 4.10 – *Pingpong-Thread_Single* : Impact du placement de 2 processus à 4 threads sur la machine *Bertha*

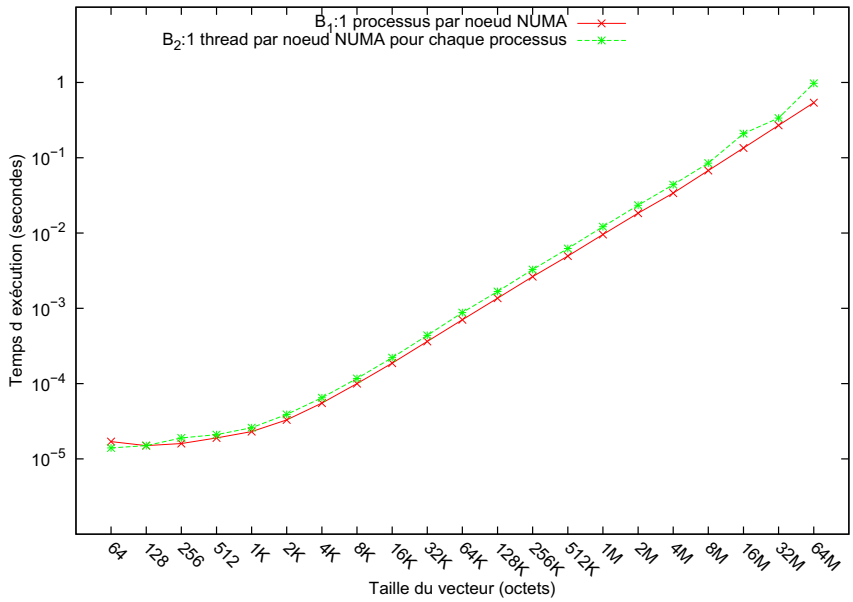


(a) MVAPICH

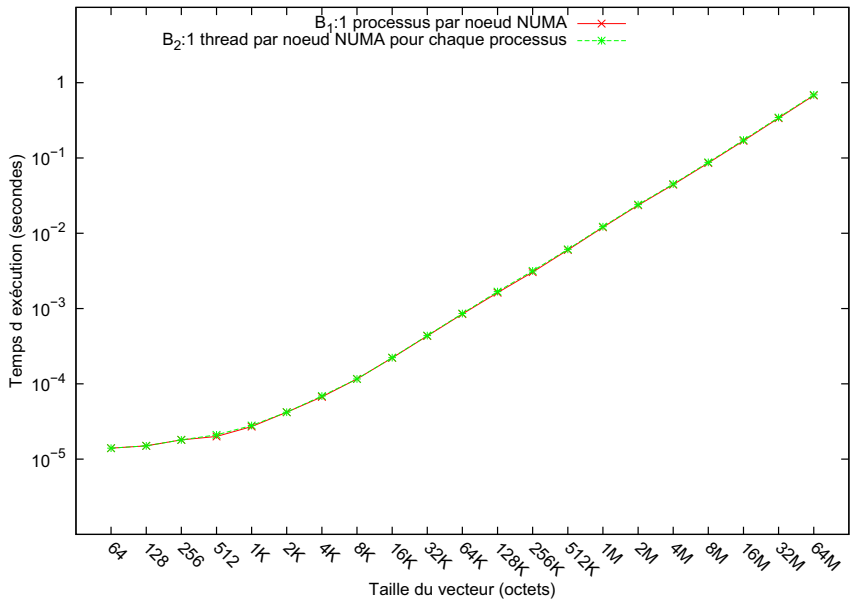


(b) OPENMPI

FIGURE 4.11 – *Pingpong-Thread_ Multiple* : Impact du placement de 2 processus à 4 threads sur la machine *Bertha*

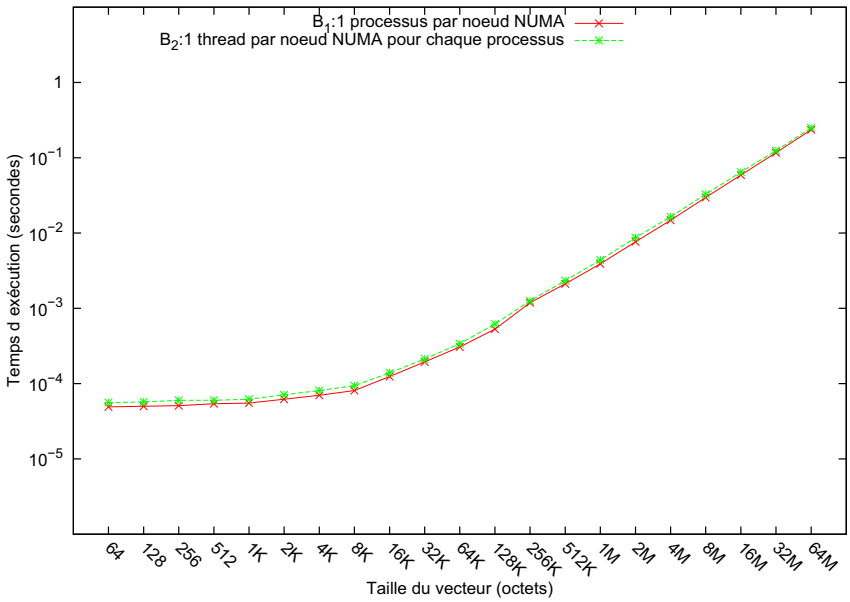


(a) MVAPICH

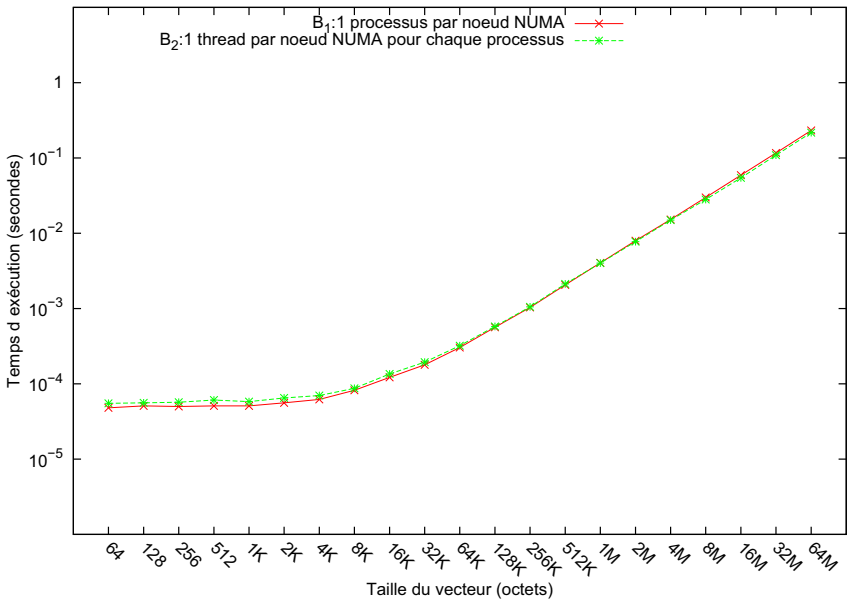


(b) OPENMPI

FIGURE 4.12 – *Pingpong-Thread_Single* : Impact du placement de 2 processus à 4 threads sur la machine *Kwak*

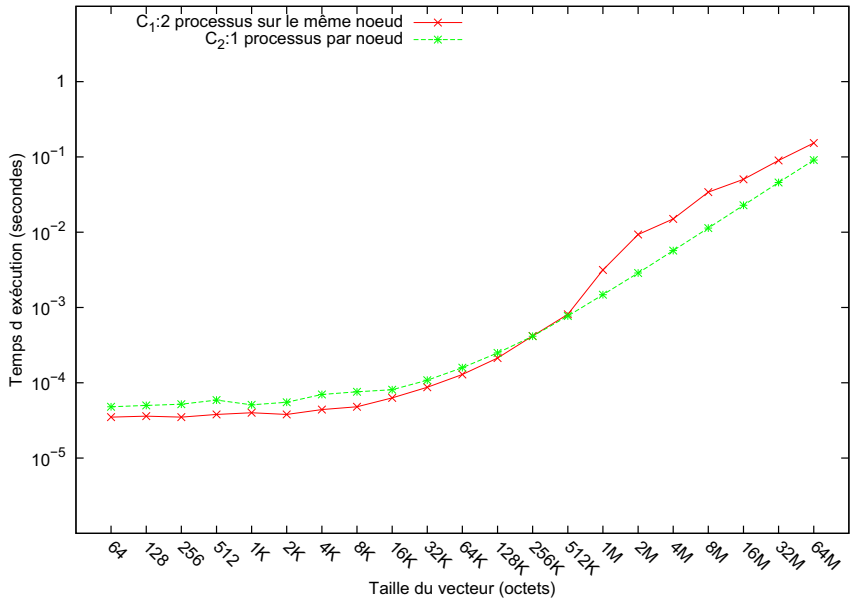


(a) MVAPICH

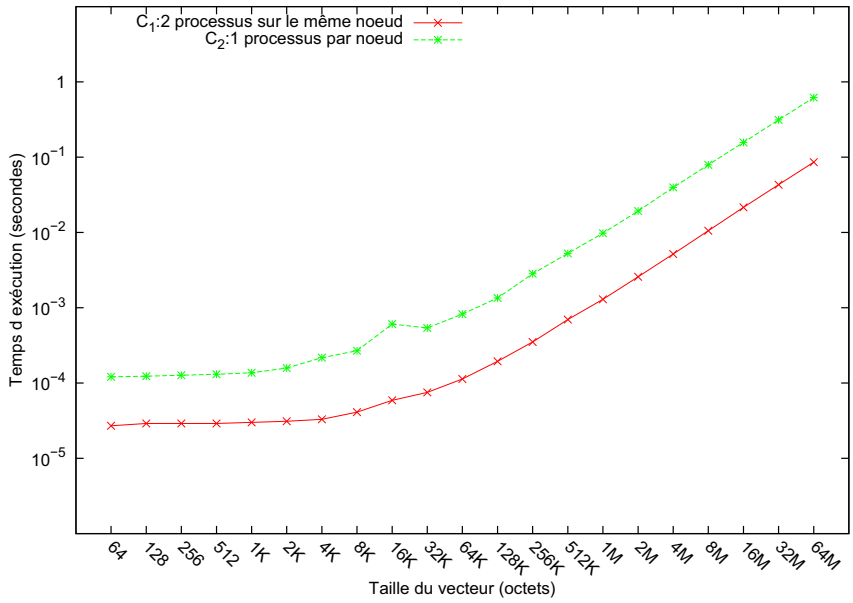


(b) OPENMPI

FIGURE 4.13 – *Pingpong-Thread_ Multiple* : Impact du placement de 2 processus à 4 threads sur la machine *Kwak*



(a) MVAPICH



(b) OPENMPI

FIGURE 4.14 – *Pingpong-Thread_ Multiple* : Impact du placement de 2 processus à 4 threads sur la grappe de calcul *Fourmi*

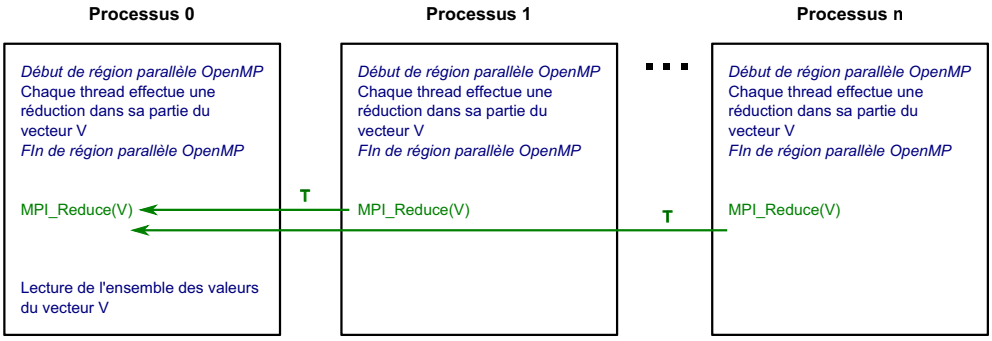


FIGURE 4.15 – Schéma de fonctionnement du microbenchmark *Reduction*

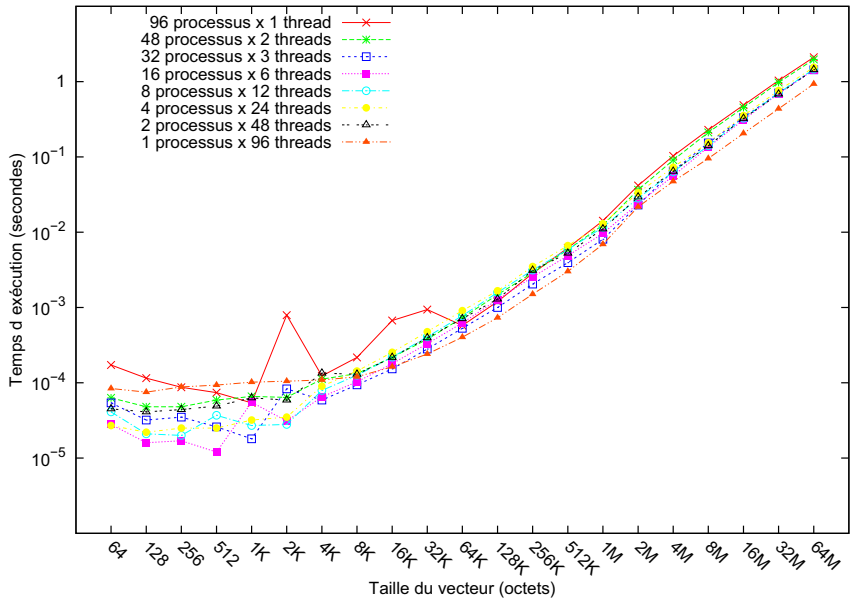
autant de threads que de cœurs, ou autant de processus que de cœurs). En effet, les courbes montrent qu'une approche intermédiaire, comme seize processus utilisant chacun six threads est une répartition des plus prometteuses. Les six threads s'exécutent sur six cœurs d'une même puce et partagent le cache de niveau 3. Les expériences réalisées avec les tests de synchronisation (figure 4.22(a)) et de diffusion d'une donnée (figure 4.19) confirment cette situation. De plus, nous pouvons voir que pour la distribution $\frac{1\ processus}{96\ threads}$ met à mal le support exécutif OPENMP lorsqu'il s'agit de synchroniser les 96 threads ensemble (près de quatre fois plus de temps que l'approche $\frac{16\ processus}{6\ threads}$ pour MPICH2 par exemple), ou même de partager une donnée. La bibliothèque de communication MPI est également en difficulté lorsqu"il s'agit de gérer 96 processus sur *Bertha* car les structures de données (pour les communications, la synchronisation, etc ..) occupent plus de mémoire et les mécanismes qui y sont liés ont un travail plus important à réaliser que dans le cas des distributions utilisant des processus multithreadés (donc avec un nombre plus restreint de processus MPI).

Les résultats des expériences sur la machine *Kwak* (figures 4.17, 4.19 et 4.22(b)) sont moins catégoriques : les distributions utilisant un processus par cœur ou un processus sur toute la machine donnent des performances tout à fait correctes. Toutefois, les distributions placant un processus à quatre threads par nœud NUMA ou un processus à huit threads sur chaque paire de nœud NUMA donne globalement les meilleures performances.

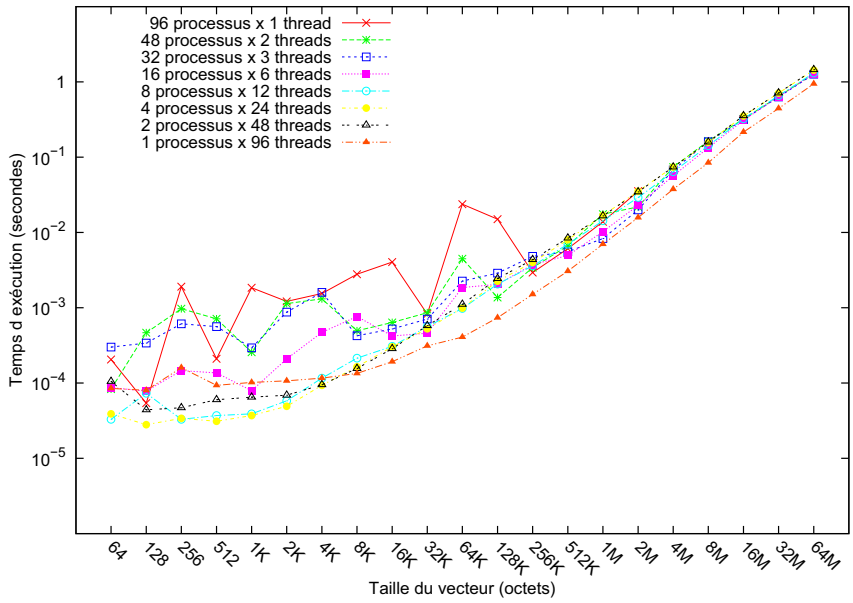
En revanche, lorsque nous expérimentons les distributions sur seize nœuds de calcul de la plate-forme *Fourmi* (figures 4.18 4.21 et 4.22(d)), il est plus efficace de laisser le support exécutif OPENMP gérer l'ensemble des cœurs d'un nœud plutôt que d'employer des distributions intermédiaires. De surcroît, la bibliothèque MVAPICH délivre de très mauvaises performances pour ces distributions.

4.2.3 Gain mémoire

Un modèle de programmation hybride permet d'adapter le code à l'architecture. Il est alors possible d'utiliser un modèle à mémoire partagée à l'intérieur d'une même nœud

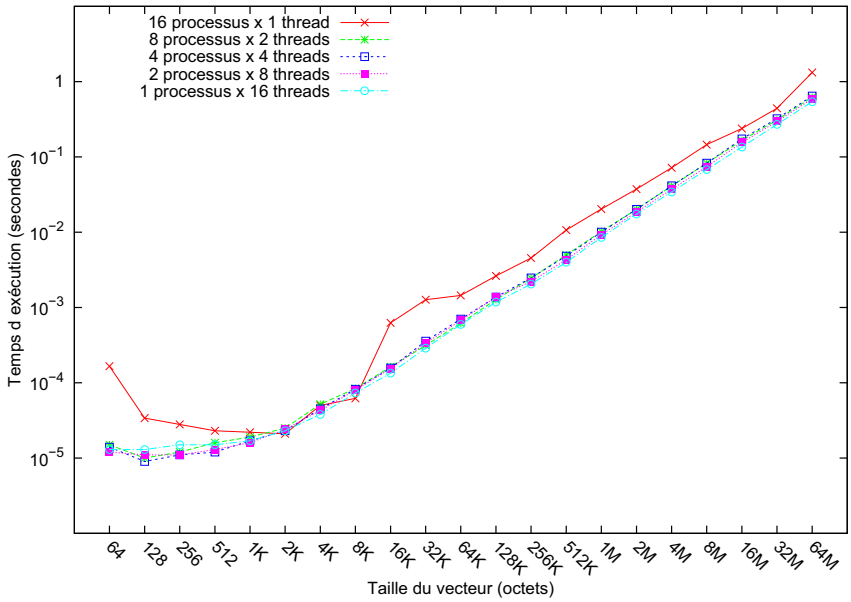


(a) MPICH2

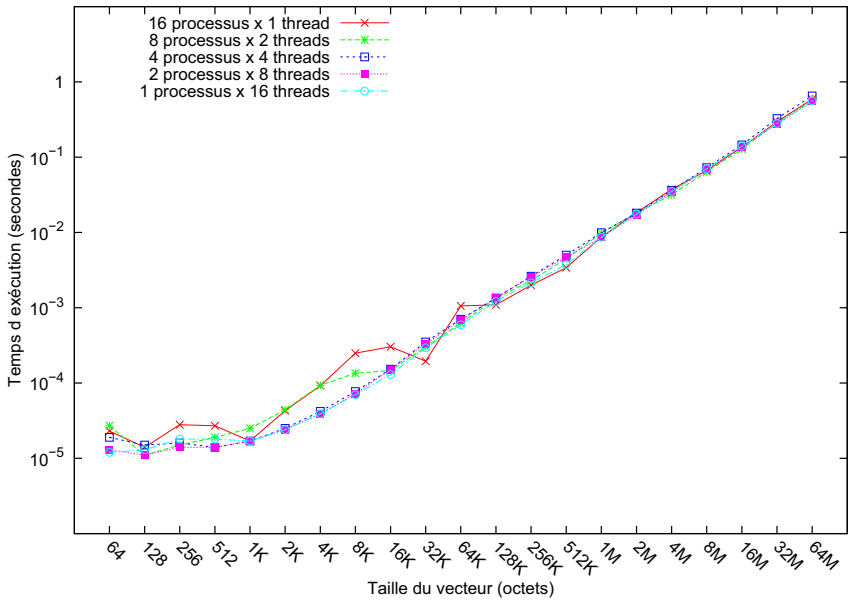


(b) OPENMPI

FIGURE 4.16 – Temps d'exécution du benchmark *Reduction* sur la machine *Bertha* pour plusieurs distributions.

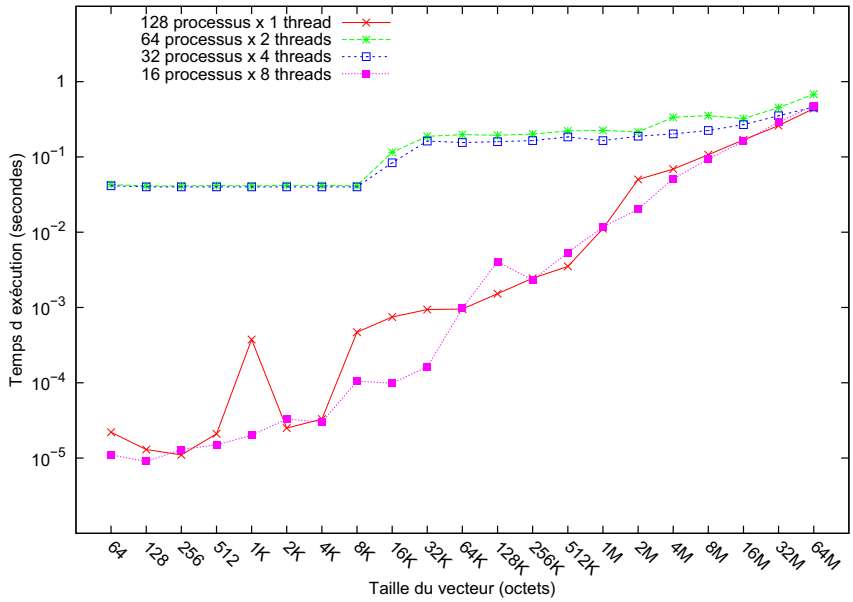


(a) MPICH2

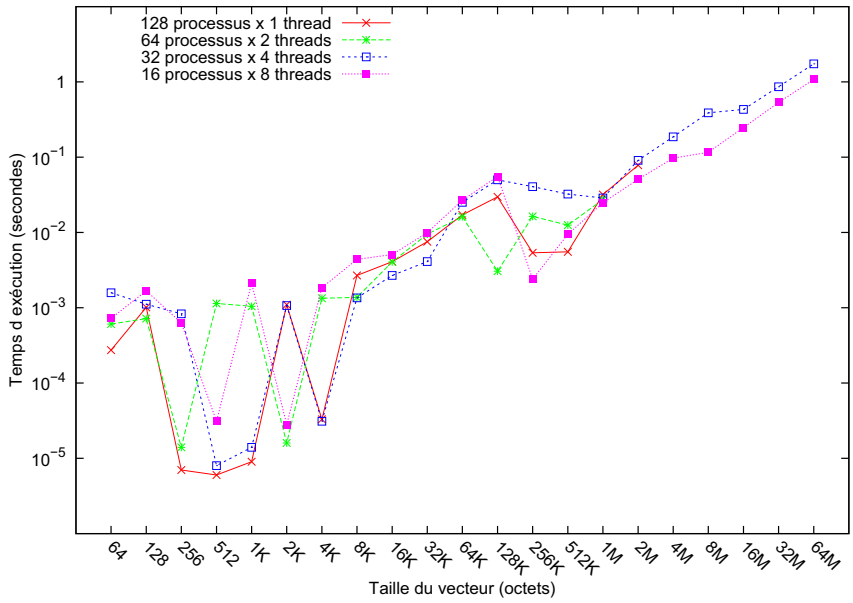


(b) OPENMPI

FIGURE 4.17 – Temps d'exécution du benchmark *Reduction* sur la machine *Kwak* pour plusieurs distributions.

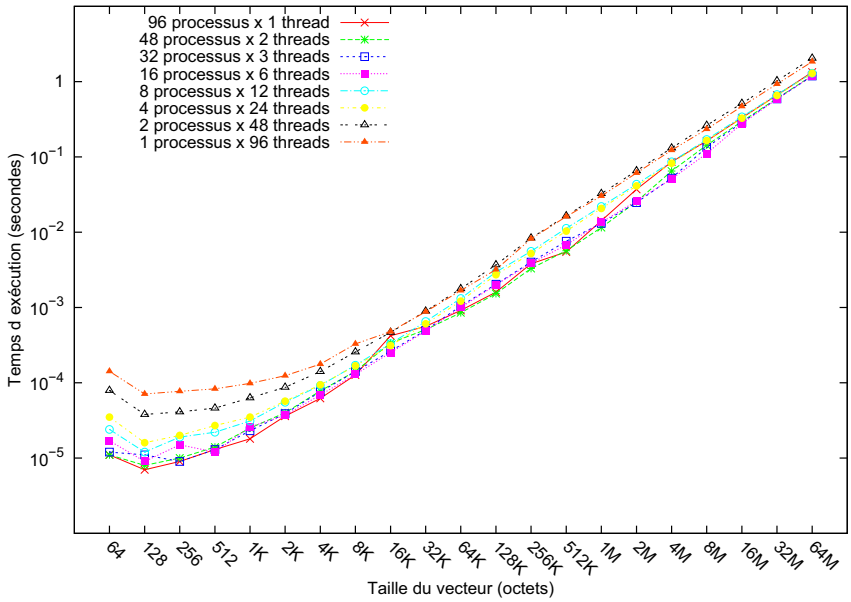


(a) MVAPICH2

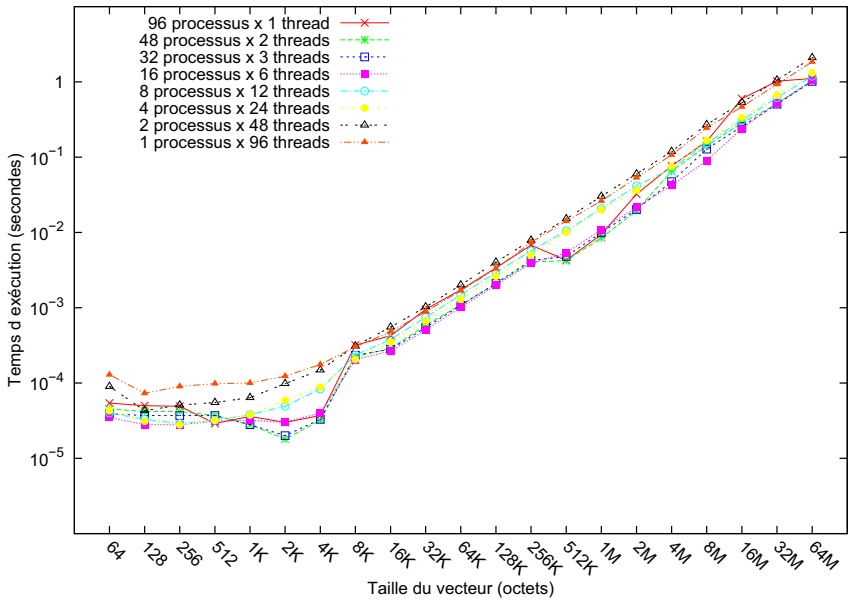


(b) OPENMPI

FIGURE 4.18 – Temps d'exécution du benchmark *Reduction* sur 16 nœuds de la grappe *Fourmi* pour plusieurs distributions.

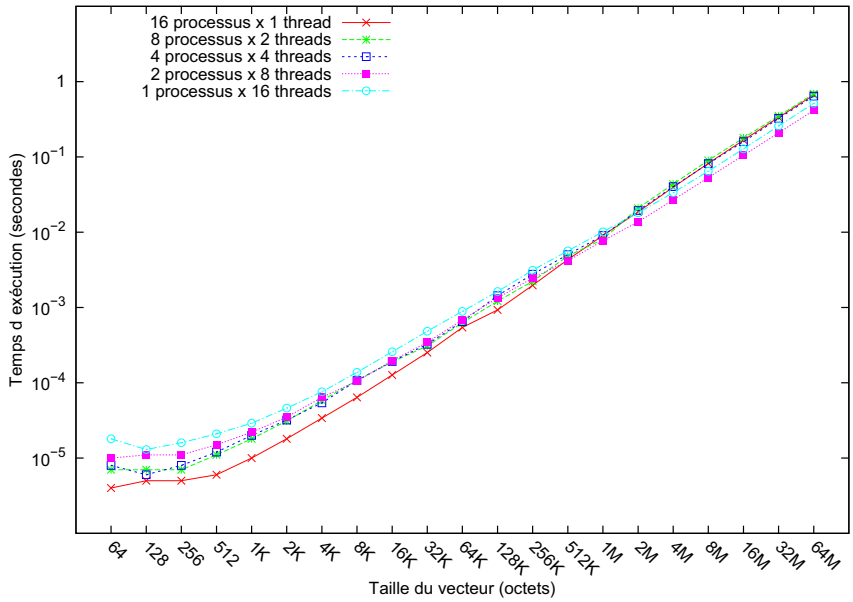


(a) MPICH2

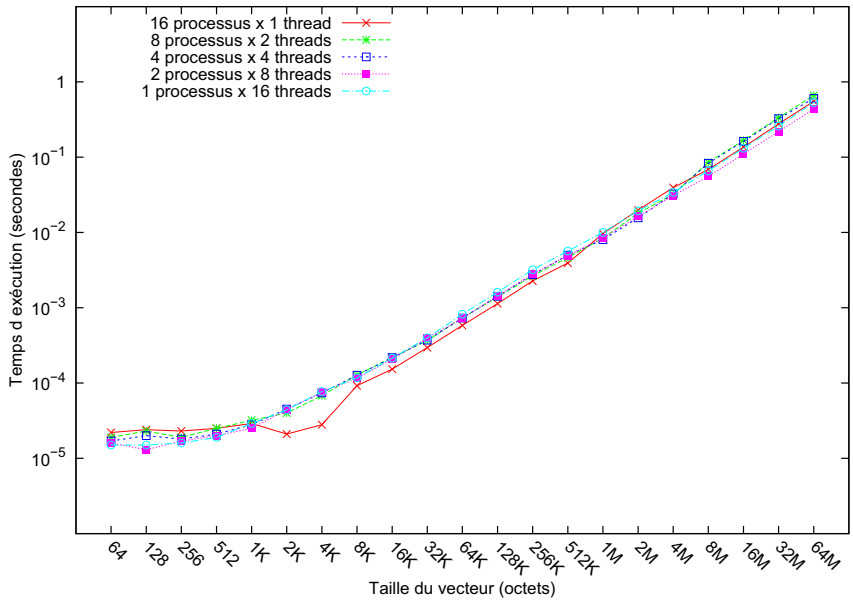


(b) OPENMPI

FIGURE 4.19 – Temps d'exécution du benchmark *Broadcast* sur la machine *Bertha* pour plusieurs distributions.

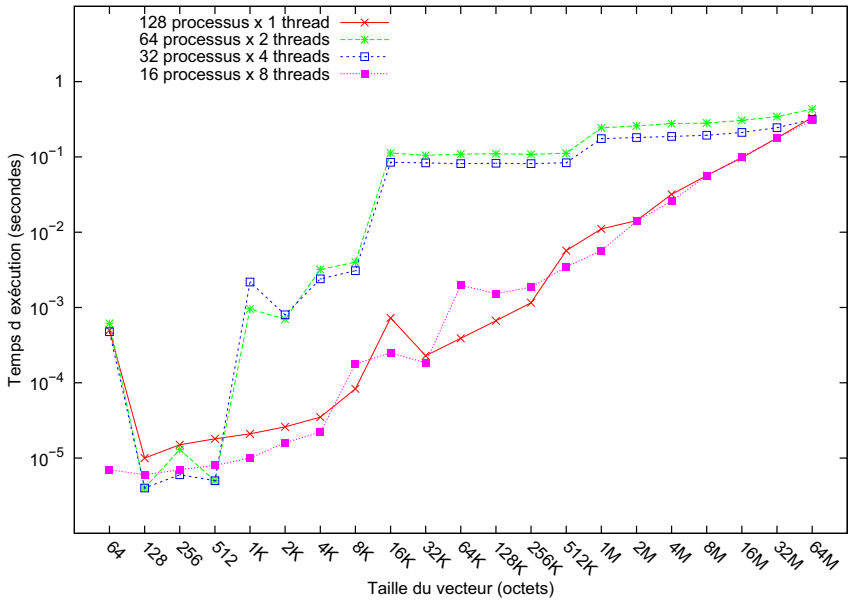


(a) MPICH2

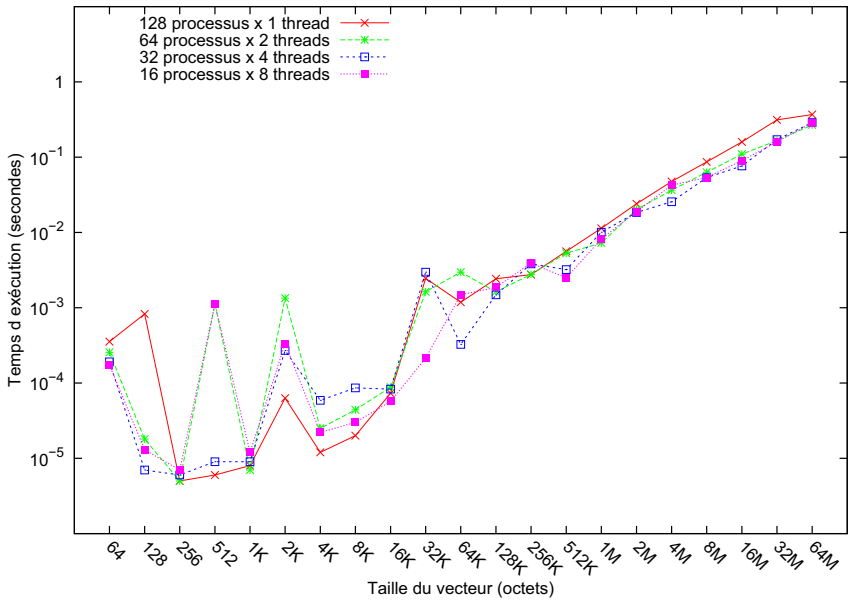


(b) OPENMPI

FIGURE 4.20 – Temps d'exécution du benchmark *Broadcast* sur la machine *Kwak* pour plusieurs distributions.



(a) MVAPICH2



(b) OPENMPI

FIGURE 4.21 – Temps d'exécution du benchmark *Broadcast* sur 16 nœuds de la grappe *Fourmi* pour plusieurs distributions.

Processus × Threads	MPICH2	OpenMPI
96 × 1	75	127
48 × 2	48	88
32 × 3	39	57
16 × 6	**34**	44
8 × 12	45	**40**
4 × 24	53	51
2 × 48	80	81
1 × 96	140	141

(a) *Bertha*

Processus × Threads	MPICH2	OpenMPI
16 × 1	**11**	20
8 × 2	**11**	15
4 × 4	12	12
2 × 8	12	**11**
1 × 16	17	17

(b) *Kwak*

Processus × Threads	MPICH2	OpenMPI
16 × 1	15	144
8 × 2	43699	79
4 × 4	41277	53
2 × 8	**6**	**42**

(c) 2 nœuds de calcul de la grappe *Fourmi*

Processus × Threads	MPICH2	OpenMPI
128 × 1	47	475
64 × 2	68420	654
32 × 4	44914	727
16 × 8	**32**	**469**

(d) 16 nœuds de calcul de la grappe *Fourmi*

FIGURE 4.22 – Temps d'exécution (en μs) du benchmark *Barrier* pour plusieurs distributions

et d'éviter de dupliquer des données. Chaque thread du modèle à mémoire partagée accède directement à la mémoire du nœud sans générer de communication explicite.

Prenons l'exemple d'une décomposition de domaines, utilisée par de nombreuses applications et qui permet de répartir les données entre les processus. Afin de rendre l'application plus performante les développeurs utilisent souvent une technique qui consiste à dupliquer une partie des données (les bords de la zone de travail du processus) des processus voisins dans le domaine pour pouvoir calculer plusieurs itérations de boucle avant de réaliser les communications. Ces zones mémoire situées aux frontières occupent un espace mémoire supplémentaire qu'il est nécessaire d'allouer dans l'espace d'adressage de chaque processus. Et cet espace mémoire croît en fonction du nombre de processus MPI. La figure 4.23 présente un exemple de décomposition d'un domaine entre `n` processus et suivant une seule dimension : chaque processus détient une zone de données de taille D x D x D/N et deux tampons mémoire servant à stocker une partie des données voisines de la zone.

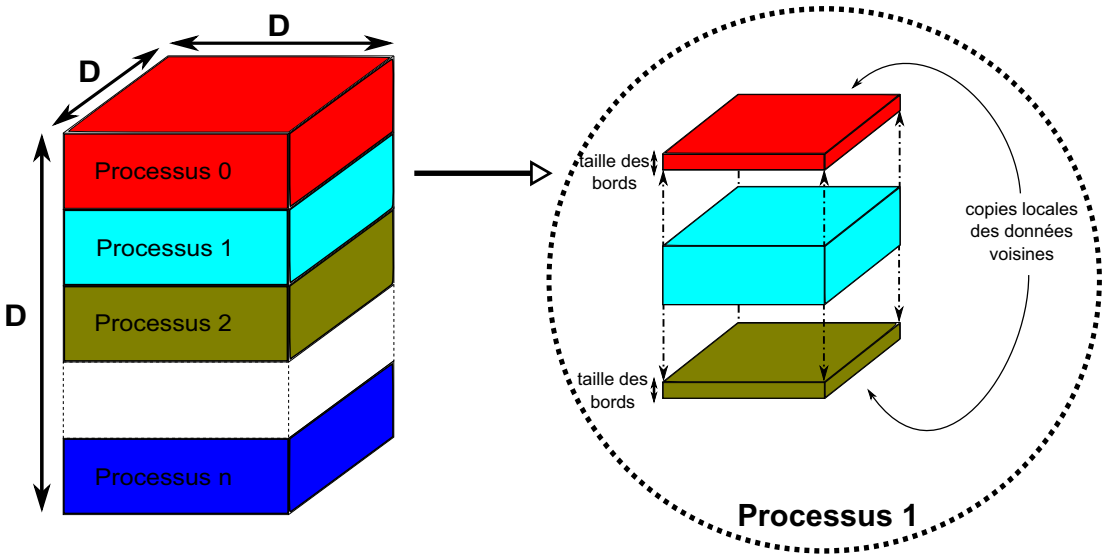


FIGURE 4.23 – Décomposition d'un domaine à trois dimensions entre n processus

Le couplage d'un paradigme de programmation à mémoire partagée et du modèle de programmation MPI permet de réduire la consommation mémoire de la décomposition de domaine. Chaque processus va gérer une partie du domaine de taille plus importante que dans le cadre d'un modèle unique. Ce sous-domaine sera alors réparti entre plusieurs threads qui n'ont pas besoin de tampons mémoire pour connaître l'état des données proches de la zone qu'ils traitent (sauf ci cela concerne des données gérées par d'autres processus MPI). En effet, tous les threads d'un même processus peuvent accéder directement aux zones des threads voisins.. L'allure de la courbe de la figure 4.24 montre le pourcentage de mémoire gagnée par l'emploi d'un modèle de programmation à mémoire partagée par rapport à celui utilisant uniquement des processus MPI. L'éva-

luation est réalisée pour un unique nœud de calcul disposant de 4 à 128 cœurs de calcul et pour une taille des copies de bord de la zone variant de 2 à 16. Nous supposons que ce nœud de calcul dispose de 64 Go de mémoire.

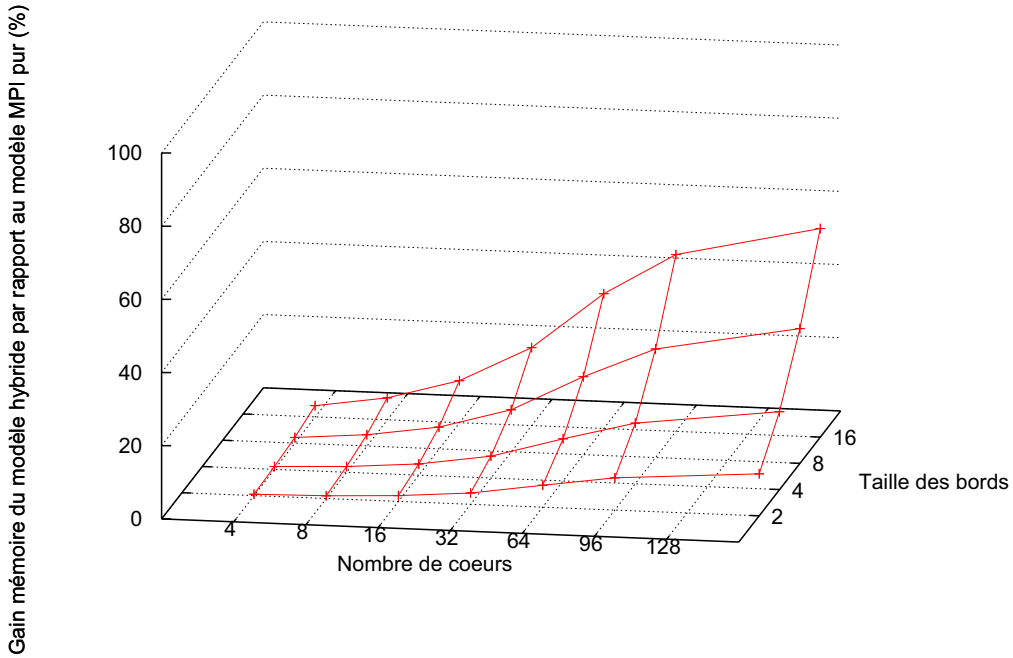


FIGURE 4.24 – Gain mémoire d'un modèle de programmation hybride par rapport à un modèle MPI pur pour une décomposition de domaine en trois dimensions

Nous voyons clairement que le gain mémoire croît avec le nombre de cœurs du nœud de calcul et la taille des tampons mémoire pour les bords. Nous nous limitons ici à un modèle hybride dans lequel il y a un processus MPI par nœud. Bien entendu nous pouvons extrapoler ces résultats pour des versions intermédiaires (c'est-à-dire avec plusieurs processus multithreadés par nœud) et les gains en mémoire seraient également notables.

4.3 NAS

Mis à disposition depuis déjà plusieurs dizaines d'années par la NASA, la suite de benchmarks NAS [BBB+91] propose un jeu de programmes représentatifs des calculs réalisés dans les applications scientifiques courantes. Ces benchmarks sont basés sur les calculs de la dynamique des fluides et travaillent sur un jeu de donnée particulier. La

taille de ce jeu de données est paramétrable en sélectionnant la *classe* du benchmark (du plus petit au plus grand : *S*, *W*, *A*, *B*, *C*, *D*, *E*, *F*).

Les versions 2 et 3 des applications NAS utilisent respectivement les modèles de programmation MPI et OpenMP pour implémenter leur calcul. Depuis quelques années, une version dite *multizone* de certains de ces benchmarks est disponible pour évaluer les performances des modèles de programmation hybrides. Le domaine de calcul est découpé en zones qui sont traitées en parallèles. Les applications affichent deux niveaux de parallélisme : un parallélisme à gros grain qui divise le domaine en zones et le parallélisme à grain fin invoqué à l'intérieur de chaque zone.

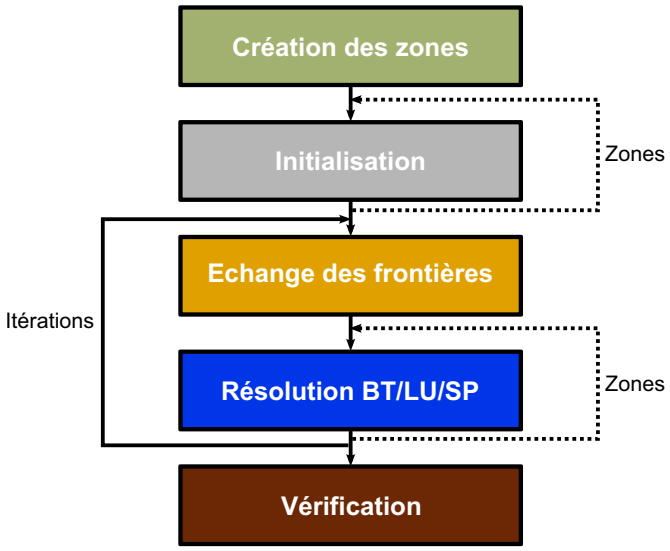


FIGURE 4.25 – Schéma d'exécution des applications NAS multizones

Les trois applications qui ont été implémentées en versions multizones sont *BT* (*Block Tri-diagonal*), *LU* (*Lower-Upper Symmetric Gauss-Seidel* et *SP* (*Scalar Penta-diagonal*). Toutes les trois résolvent des équations dans une domaine en trois dimensions. La figure 4.25 présente le schéma d'exécution de l'application. Pour un certain nombre d'itérations, les équations sont résolues indépendamment dans chaque zone, et après chaque itération, les zones s'échangent les valeurs des frontières avec leurs voisins immédiats. Les zones sont créées en découpant le domaine total suivant les 2 axes x et y. Les applications NAS diffèrent selon la taille de ces zones et bien entendu les calculs qu'ils réalisent à l'intérieur de chaque zone.

Dans l'implémentation hybride (MPI et OpenMP) des benchmarks multizones, on assigne à chaque processus MPI un ensemble de zones (au minimum de cardinal égal à un). À l'intérieur de chaque zone, les threads OpenMP du processus se répartissent les indices des boucles externes. Pour SP-MZ et BT-MZ, il s'agit principalement de boucles sur la troisième dimension (z). Concernant LU-MZ, ce sont des boucles sur la dimension y, qui offre une parallélisation plus efficace que z (la taille croît beaucoup

plus dans la seconde dimension lorsque le domaine grandit).

4.3.1 LU-MZ

Dans le benchmark LU-MZ, pour toutes les classes d'application, le domaine est découpé en seize zones (quatre zones en x et quatre zones en y). C'est la taille de ces zones qui varie selon la classe choisie, cependant toutes les zones ont la même taille. La répartition de charge est donc facilitée pour LU-MZ. Par ailleurs, le nombre de zones étant fixé à seize, le nombre de processus est limité à ce même nombre. Ainsi, pour nos expériences sur cette application, nous avons restreint le nombre de nœuds de la grappe *Fourmi* utilisés à deux. Cela nous permet d'évaluer les performances d'un modèle utilisant uniquement MPI. La machine *Bertha* disposant de 96 cœurs ne fait pas partie des architectures testées pour ce benchmark, le modèle utilisant un thread par processus (et utilisant tous les cœurs) n'est pas disponible.

Classe	Coord.	# zones	Taille
C	x	4	120
	y	4	80
	z	1	28
D	x	4	408
	y	4	304
	z	1	34

FIGURE 4.26 – Taille (en nombre de points) des zones pour LU-MZ

En observant les résultats obtenus par LU-MZ sur deux nœuds de calcul à huit cœurs (figure 4.27), nous constatons que l'utilisation d'un modèle hybride avec un processus par nœud et huit threads par processus est la meilleure stratégie. En effet, le nombre de zones étant relativement restreint, la quantité de travail au sein de chaque zone est suffisamment importante pour que les threads OpenMP offrent une bonne accélération par rapport à une version n'utilisant qu'un seul thread par zone. Dans la distribution $\frac{2\ processus}{8\ threads}$, les deux processus détiennent chacun huit zones traitées successivement par leurs huit threads alors que dans la distribution $\frac{16\ processus}{1\ thread}$, les seize processus traitent chacun l'unique zone qui leur est affectée et cela séquentiellement. Bien entendu notre environnement permet d'évaluer automatiquement les différentes distributions réalisables et d'en ressortir la plus performante.

4.3.2 SP-MZ

Comme c'est le cas pour le benchmark LU-MZ, le domaine de calcul de SP-MZ est découpé en zones de tailles identiques entre elles. Les charges de travail de chaque processus sont donc équilibrées. Toutefois le nombre de ces zones croît avec la taille du domaine. De ce fait, contrairement à LU-MZ, nous pouvons expérimenter l'application

	Classe C		Classe D	
Processus × Threads	MPICH2	OpenMPI	MPICH2	OpenMPI
16 × 1	334,18	341,34	10787,27	9631,26
8 × 2	334,64	338,85	12735,55	8305,56
4 × 4	334,22	337,41	7797,43	7426,86
2× 8	239,16	240,39	**7236,56**	**7253,42**
1× 16	**209,01**	**209,08**	7906,7	8000,09

(a) *Kwak*

	Classe C		Classe D	
Processus × Threads	MVAPICH2	OpenMPI	MVAPICH2	OpenMPI
16 × 1	115,29	116,03	2287,66	2427,95
8 × 2	255,54	110,6	4397,7	2275,69
4 × 4	165,6	79,74	3413,9	2307,64
2× 8	**78,38**	**79,47**	**2013,44**	**2024,5**

(b) 2 nœuds de la grappe *Fourmi*

FIGURE 4.27 – Temps d'exécution (en secondes) de *LU-MZ* en fonction du nombre de processus MPI et du nombre de threads OpenMP par processus

sur un plus grand nombre de nœuds de la grappe *Fourmi*. En outre, nous pouvons évaluer toutes les distributions calculées par notre outil sur la machine *Bertha*.

Classe	Coord.	# zones	Taille
C	x	16	30
	y	16	20
	z	1	28
D	x	32	51
	y	32	38
	z	1	34

FIGURE 4.28 – Taille (en nombre de points) des zones pour SP-MZ

Les temps d'exécution de SP-MZ, autant ceux relatifs aux seize nœuds de la grappe *Fourmi* que ceux de la machine *Bertha*, montrent que le modèle générant un processus par cœur (donc sans multithreading) est le plus efficace parmi l'ensemble des distributions réalisables. Cela s'explique aisément : contrairement à l'application LU-MZ le nombre de zones évolue avec la taille du domaine de calcul (256 pour la classe C et 1024 pour la classe D). Le domaine étant de la même taille entre benchmarks d'une même

classe, la taille de chaque zone est beaucoup plus petite pour SP-MZ que pour LU-MZ. La charge de calcul dans chaque zone n'est pas suffisante pour que les threads d'une section parallèle puissent fournir une accélération efficace. Grâce à notre outil de visualisation, les traces générées montrent clairement cette problématique. En effet, le schéma d'exécution partiel exposé en figure 4.30 montre que pour la distribution $\frac{1\ processus}{96\ threads}$ la terminaison des sections parallèles génèrent beaucoup plus de synchronisations pour une même durée. Par ailleurs, il ne faut pas négliger l'impact du partage mémoire lié aux calculs multithreadés dans chaque zone. Le cas de la machine *Bertha* illustre parfaitement les effets néfaste que cela peut avoir : lorsqu'une zone est partagée entre tous les cœurs du système le temps d'exécution est multiplié par un facteur pouvant atteindre 14.

	Classe C	
Processus × Threads	MPICH2	OpenMPI
96 × 1	-	-
48 × 2	**27,5**	**27,43**
32 × 3	28,58	28,35
16 × 6	30,12	29,52
8 × 12	53,97	55,12
4 × 24	77,65	79,23
2 × 48	163,81	168,6
1 × 96	391,15	400,23

(a) *Bertha*

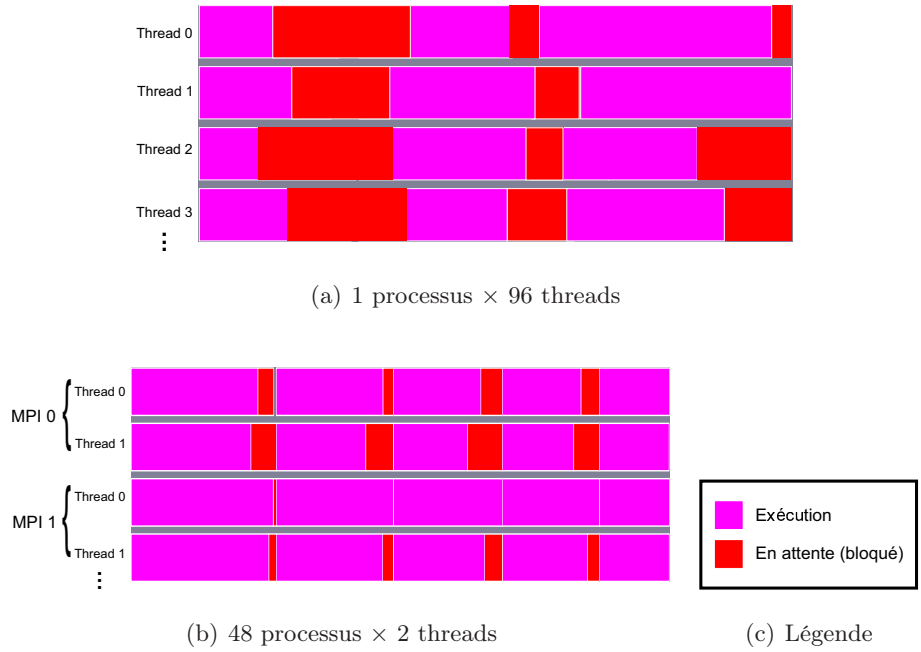
	Classe C		Classe D	
Processus × Threads	MVAPICH2	OpenMPI	MVAPICH2	OpenMPI
128 × 1	**20,76**	21,37	**313,44**	**224,53**
64 × 2	79,69	**17,71**	571,98	237,26
32 × 4	56,2	21,53	584,46	295,19
16 × 8	36,28	26,67	687,1	409,69

(b) 16 nœuds de la grappe *Fourmi*

Figure 4.29 – Temps d'exécution (en secondes) de *SP-MZ* en fonction du nombre de processus MPI et du nombre de threads OpenMP par processus

4.3.3 BT-MZ

De la même manière que pour SP-MZ, plus la taille du domaine augmente, plus le nombre de zones augmente. La différence pour ce cas réside dans l'irrégularité de la taille des zones. En partant du point d'origine, la taille des zones, selon x et selon y,



(a) 1 processus × 96 threads

(b) 48 processus × 2 threads

(c) Légende

FIGURE 4.30 – Visualisation de traces d'exécution de SP-MZ sur la machine *Bertha*

s'accroît successivement (cf. figure 4.31). Le ratio entre la plus grande et la plus petite zone est d'environ 20. Ainsi, le parallélisme externe devient irrégulier. L'équilibrage de charge est plus complexe que pour les deux autres benchmarks.

Les résultats des expériences réalisées sur la machine *Bertha* tendent à privilégier une distribution intermédiaire, contrairement à ceux des deux autres applications NAS multizones. En effet, les distributions $\frac{32\ processus}{3\ threads}$ (deux processus par puce multicoeur) et $\frac{16\ processus}{6\ threads}$ (un processus par puce multicoeur) donnent d'excellentes performances. Le format de découpage des zones du domaine est une des raisons de cet état de fait. Cette fois-ci les processus traitent chacun un ensemble de zones de tailles irrégulières. La quantité de travail est donc plus ou moins importante selon les zones. Lorsque les processus sont monothreadés, il y a de fortes chances qu'un déséquilibre se crée et que les processus ne terminent pas (approximativement) au même moment. L'utilisation de processus multithreadés permet de réduire ce déséquilibre puisque le partage de travail se fait à un grain plus grossier. Une distribution employant un seul processus générant autant de threads que de cœurs devrait donc donner les performances optimales. Cependant il ne faut pas oublier l'impact du partage de données dans une hiérarchie que nous avons évoqué précédemment.

Classe	Coord.	# zones	Tailles
C	x	16	13, 14, 15, 18, 19, 21, 23, 26, 28, 31, 35, 38, 43, 47, 52, 57
	y	16	8, 10, 10, 12, 12, 14, 16, 17, 19, 21, 23, 26 28, 31, 35, 38
	z	1	28
D	x	32	22, 23, 24, 25, 26, 28, 29, 31, 32, 34, 35, 37, 39, 41, 43, 45, 48, 49, 52, 55, 58, 60, 63, 67, 70, 73, 77, 81, 85, 88, 94, 98
	y	32	16, 17, 18, 19, 20, 20, 22, 23, 24, 25, 26, 28, 29, 30, 33, 33, 35, 37, 39, 41, 43, 45, 47, 50, 52, 54, 58, 60, 63, 66, 70, 73
	z	1	34

FIGURE 4.31 – Taille (en nombre de points) des zones pour BT-MZ

	Classe C		ClasseD	
Processus × Threads	MPICH2	OPENMPI	MPICH2	OPENMPI
96 × 1	43,53	42,65	742,96	735,43
48 × 2	24,29	24,88	619,66	622,38
32 × 3	**19,98**	**21,79**	562,71	567,44
16 × 6	21,2	23,11	**493,18**	**494,02**
8 × 12	34,17	36,29	595,78	592,92
4 × 24	50,32	52,24	857,71	853,13
2 × 48	88,18	89,59	1441,12	1438,85
1 × 96	199,28	203,33	2989,53	3000,38

FIGURE 4.32 – Temps d'exécution (en secondes) de *BT-MZ* sur la machine *Bertha* en fonction du nombre de processus MPI et du nombre de threads OPENMP par processus

4.4 Bilan de l'évaluation

Dans ce chapitre nous avons présenté l'évaluation de l'impact du placement et du dimensionnement dans le cadre d'un modèle de programmation hybride. Dans un premier temps, nous avons montré les résultats d'exécution de différents microbenchmarks représentatifs des programmes mêlant les deux modèles de programmation MPI et OPENMP. Il en ressort qu'il est tout aussi important de veiller à placer correctement les threads d'un même processus afin qu'ils partagent un même niveau de mémoire cache que de prendre soin à répartir les processus de façon à respecter leurs relations d'inter-affinité. Par ailleurs, le contrôle du nombre de processus et du nombre de threads par processus nous a permis de constater que sur des architectures de grande dimension (de nombreux cœurs par nœud) et fortement hiérarchiques l'emploi de plusieurs processus

multithreadés par nœud se révéle une solution particulièrement adaptée et efficace.

Les expériences réalisées avec la suite des benchmarks parallèles NAS nous ont permis d'obtenir des résultats similaires sur des applications typiques du calcul scientifique. En outre, ce sont à la fois l'architecture des systèmes de calcul et le parallélisme, régulierou irrégulier, exprimé par l'application qui déterminent le dimensionnement du nombre de processus et du nombre de threads optimal. Nous nous appuyons sur la visualisation des traces d'exécution pour expliquer plus précisément les performances obtenues.

Conclusion

Animés par les besoins soutenus des grandes applications scientifiques et industrielles, les recherches et développements des grands fabricants de processeurs et constructeurs de calculateurs refaçonnent régulièrement l'architecture des machines du domaine du calcul intensif. Les supercalculateurs ont ainsi cédé la place aux grappes de machines monoprocesseurs, qui elles-mêmes ont évolué vers des grappes de machines multiprocesseurs et multicœurs. Chaque nœud de ces grappes est désormais une véritable machine parallèle dont la structure hiérarchique expose plusieurs niveaux de parallèlisme. Ainsi, les cœurs partagent entre eux une hiérarchie de caches et sont reliés aux cœurs d'autres puces par un réseau d'interconnexion, formant par là même des nœuds NUMA. Cette topologie matérielle complexe crée des relations d'affinités entre les unités de calcul qui nécessitent d'être prises en considération pour optimiser l'exploitation des calculateurs modernes.

De nombreux modèles de programmation existent pour exploiter ces calculateurs, chacun étant adapté à un type d'architectures particulier. Les paradigmes de programmation en mémoire partagée (Pthreads, OpenMP, Intel TBB, etc.) tirent efficacement parti des machines multiprocesseurs symétriques et des architectures NUMA. De l'autre côté les modèles de programmation en mémoire distribuée (MPI, Co-Array Fortran, UPC) sont tout à fait adaptés aux grappes de calculs dont la mémoire est physiquement répartie entre les différents nœuds qui la compose. Bien que des efforts aient été réalisés pour améliorer les performances des modèles distribués dans un contexte de grappes multicœurs, leur structure hiérarchique n'est pas vraiment prise en compte. De ce constat et dans l'attente d'un paradigme de programmation plus adapté, la combinaison de deux modèles de chaque type est venu tout naturellement pour réunir à la fois les atouts du paradigme du distribué et de la mémoire partagée.

L'interface de communication MPI et le langage OpenMP s'avèrent être les modèles de programmation parallèle les plus répandus et sont tous deux représentatifs de leur catégorie. Des efforts ont été réalisés pour gérer leur utilisation conjointe, notamment au niveau du standard MPI qui définit différents degrés de liberté pour l'emploi des threads et des primitives de communication. En dépit de ces efforts, les modèles mixtes peinent à supplanter le modèle unique car les applications suivent le plus souvent un schéma de parallélisation rigide et inapproprié au relief des architectures multicœurs. Les unités de calcul sont considérées comme étant symétriques et ne sont sujettes à aucune différenciation selon leurs affinités respectives.

Contribution

Dans cette thèse, nous avons proposé un ensemble d'outils et de de techniques de programmation destinés à assister le développeur d'applications parallèles dans l'emploi de la combinaison de deux paradigmes de programmation. L'objectif est de tirer parti des niveaux de parallélisme que l'on peut exprimer pour l'adapter à l'architecture des calculateurs modernes.

Dans un premier temps nous avons exprimé le besoin d'une connaissance approfondies de la structure des machines de calcul utilisées, ce qui nous a mené à participer au développement de l'outil Hwloc pour récolter les informations topologiques sur le matériel. Nous obtenons une représentation générique du matériel ciblé sous forme d'une structure arborescente dont les feuilles correspondent aux unités de calcul. Chaque niveau de l'arbre correspond au partage d'une ressource commune (mémoire cache, banc mémoire) et nous permet de définir une relation d'affinité particulière.

Nous utilisons ces informations pour définir différentes politiques de dimensionnement du nombre de processus et de threads par processus. L'idée est de faire correspondre la structure du programme à la hiérarchie du matériel sous-jacent. Les threads d'un même processus sont alors placés sur des unités de calcul adjacentes et partageant des ressources mémoire. La localité mémoire et la localité spatiale sont respectées pour optimiser l'exécution globale de l'application. Par ailleurs, le contrôle du nombre de processus MPI permet de maîtriser l'impact des communications circulant au sein du calculateur.

Le dernier point de notre contribution concerne la visualisation des traces d'exécution grâce au logiciel Eztrace. Celui-ci intercepte les différents appels liés aux deux modèles de programmation MPI et OpenMP. Nous retraçons ainsi le schéma d'exécution des applications hybrides pour en comprendre les performances.

Le résultats que nous avons obtenus montrent que la combinaison des modèles MPI et OpenMP doit être finement contrôlée pour s'adapter à la hiérarchie de l'architecture et au parallélisme exprimé par l'application. Il s'agit de trouver un compromis pour équilibrer la charge et ne pas être pénalisé par la gestion d'un trop grand nombre de threads ou de processus.

Perspectives

Ces travaux ouvrent de nombreux perspectives à court, moyen et long termes.

Vers une compréhension plus fine du comportement applicatif

Dans un premier temps nous envisageons de collecter davantage d'informations concernant le comportement de l'application, notamment comme nous l'avons évoqué précédemment concernant les défauts de cache. Dans l'état actuel, nous nous appuyons

sur les temps d'exécution afin d'évaluer la meilleure répartition entre threads et processus sur le système de calcul. La visualisation du schéma d'exécution de l'application au travers du logiciel Vite permet de mieux appréhender les temps d'activité et d'inactivité des threads ainsi que les conséquences sur l'exécution globale de l'application. Toutefois cela requiert un travail manuel de la part de l'utilisateur. L'utilisation de compteurs matériels grâce à l'interface PAPI [pap] permettra ainsi de déterminer l'impact des partages de données liés au multithreading : une grande quantité de défauts de cache indiquerait que le threads travaillent sur les mêmes données mais ne partagent pas de mémoire cache (ou celle-ci n'est pas suffisante). C'est un indicateur à prendre en compte pour comprendre les performances relatives des répartitions entre threads et processus dans le cadre d'un modèle hybride de programmation.

Une relation étroite entre l'implémentation MPI et le support exécutif OpenMP

Bien que le multithreading soit supporté par les implémentations les plus couramment utilisées, la plupart des applications écrites à la fois avec MPI et OpenMP utilisent un schéma de programmation relativement similaires. Les données qui représentent l'espace de travail sont réparties entre les différents processus puis chaque processus crée un ensemble de threads qui sera en charge du travail local au processus. Les échanges de données entre processus sont réalisés en dehors des sections parallèles. Chaque dialogue inter-processus implique une synchronisation entre les threads et ce pour chaque processus impliqué dans la communication. Ces synchronisations génèrent un temps d'attente plus ou moins important pour tous les threads (sauf un) et laisse autant de processeurs inoccupés. L'utilisation des primitives de communication MPI à l'intérieur des sections parallèles permet d'y remédier. Chacun des threads transmet une partie des données aux autre processus dès qu'ils le peuvent et ne bloquent pas la progression des autres threads. Cependant l'envoi consécutif de plusieurs messages génèrent un trafic réseau important et il peut s'avérer plus efficace qu'un seul message de plus grande taille. C'est ce que proposent Preissl et al. [PWL+11] en utilisant un langage de type PGAS pour agréger un certaine quantité de données dans un même buffer transmise par la suite aux autres processus. C'est une solution tout à fait envisageable pour le modèle de programmation MPI. Grâce à l'une de stratégies de la bilbiothèque de communication NewMadeleine développée dans l'équipe Runtime, un ensemble de petits messages transmis par l'application peut être aggrégé au sein d'un seul puis envoyé au processus distant.

Bibliographie

[AAC+04] George Almási, Charles Archer, José Castaños, C. Erway, Philip Heidelberger, Xavier Martorell, José Moreira, Kurt Pinnow, Joe Ratterman, Nils Smeds, Burkhard Steinmacher-burow, William Gropp, and Brian Toonen. Implementing mpi on the bluegene/l supercomputer. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 833–845. Springer Berlin / Heidelberg, 2004. pages 44

[AAD+10] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, 09 2010. pages 31

[AAD+11a] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In *9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11)*, Sharm El-Sheikh, Egypt, june 2011. pages 31

[AAD+11b] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *25th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2011)*, Anchorage, Alaska, USA, 5 2011. pages 31

[ABB+08] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. Early evaluation of IBM BlueGene/P. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 23 :1–23 :12, Piscataway, NJ, USA, 2008. IEEE Press. pages 10

[ABI+09] E Ayguadé, Rosa M Badia, Francisco D Igual, J Labarta, Rafael Mayo, and Enrique S Quintana-Ortí. An extension of the starss programming model for platforms with multiple gpus. *EuroPar 2009*, pages 851–862, 2009. pages 31

[ACD+09] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20 :404–418, 2009. pages 38

[ACP95] T.E. Anderson, D.E. Culler, and D. Patterson. A case for now (networks of workstations). *Micro, IEEE*, 15(1) :54 –64, feb 1995. pages 12

[AG96] S.V. Adve and K. Gharachorloo. Shared memory consistency models : a tutorial. *Computer*, 29(12) :66 –76, dec 1996. pages 27

[AJR06] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring Thread and Memory Placement on NUMA Architectures : Solaris and Linux, UltraSPARC/F irePlane and Opteron/HyperTransport. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, Bangalore, India, December 2006. pages 29

[AMD01] Inc. Advanced Micro Devices. HyperTransport Technology I/O Link, A High-Bandwidth I/O Architecture, July 2001. pages 29

[AMD10a] AMD. *AMD Computer Abstraction Layer (CAL) Programming Guide*, December 2010. pages 30

[AMD10b] AMD. Amd fusion family of apus technology overview. White Paper, 2010. pages 32

[ATH90] D. Akihiro, W. Toshihiko, and N. Hideki. Packaging technology for the nec sx-3/sx-x supercomputer. In *Electronic Components and Technology Conference, 1990. ., 40th*, pages 525 –533 vol.1, may 1990. pages 23

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, 23 :187–198, February 2011. pages 31

[BBB+91] D H Bailey, E Barszcz, J T Barton, R L Carter, T A Lasinski, D S Browning, L Dagum, R A Fatoohi, P O Frederickson, and R S Schreiber. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3) :63–73, 1991. pages 118

[BBG+09] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing L. Lusk, Rajeev Thakur, and Jesper Larsson Träff. Mpi on a million processors. In *PVM/MPI*, pages 20–30, 2009. pages 49

[BBG+10] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. Fine-grained multithreading support for hybrid threaded mpi programming. *IJHPCA*, 24 :49–57, 2010. pages 55

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet : A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1) :29–36, 1995. pages 17

[BCOM+10] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc : a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pages 180–186, Pisa, Italia, February 2010. IEEE Computer Society Press. pages 65

[BdSS+11] Susmit Biswas, Bronis R. de Supinski, Martin Schulz, Diana Franklin, Timothy Sherwood, and Frederic T. Chong. Exploiting data similarity to reduce memory footprints. In *IPDPS*, pages 152–163. IEEE, 2011. pages 52

[BEA09] J. Mark Bull, James P. Enright, and Nadia Ameer. A microbenchmark suite for mixed-mode openmp/mpi. In *IWOMP'09*, pages 118–131, 2009. pages 100

[BEKK00] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6) :885 –898, nov. 2000. pages 24

[BER] Berkeley unified parallel c (upc) project. `http://upc.lbl.gov/`. pages 46

[Ber96] Daniel J. Berg. Java threads - a white paper. Technical report, Sun Microsystems, 1996. pages 37

[BFG+10] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP : an efficient OpenMP environment for NUMA architectures. *International Journal on Parallel Programming, Special Issue on OpenMP ; Guest Editors : Matthias S. Müller and Eduard Ayguadé*, 38(5) :418–439, 2010. pages 71

[BMG07] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in mpich2 using the nemesis communication subsystem. *Parallel Computing*, 33(9) :634 – 644, 2007. pages 72, 98

[BPBL09] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Exploiting locality on the cell/b.e. through bypassing. In *Proceedings of the 9th International Workshop on Embedded Computer Systems : Architectures, Modeling, and Simulation*, SAMOS '09, pages 318–328, Berlin, Heidelberg, 2009. Springer-Verlag. pages 31

[Bro10] François Broquedis. *De l'exécution d'applications scientifiques OpenMP sur architectures hiérarchiques.* PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, December 2010. pages 71

[Bul] Bull. *Bullx Cluster Suite - Application Developer's Guide.* `2.1-MPIBull2`. pages 44

[CC00] Giuseppe Ciaccio and Giovanni Chiola. GAMMA and MPI/GAMMA on gigabitethernet. In *Proceedings of 7th EuroPVM-MPI conference*, pages 129–136, Balatonfured, Hongrie, September 2000. pages 19

[CCD+05] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000 : A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society. pages 14

[CDC+99] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to upc and language specification. Technical report, IDA Center for Computing Sciences, may 1999. pages 46

[CKD+10] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30 :16–29, March 2010. pages 26

[Cor09] Intel Corp. An Introduction to the Intel QuickPath Interconnect, January 2009. pages 29

[Cor10] Nvidia Corp. Tesla M2050/M2070 GPU Computing Module Supercomputing At 1/10th The Cost, 2010. pages 30

[Cra] Cray Inc. `http://www.cray.com`. pages 20

[Cud] Nvidia Cuda Zone. `http://www.nvidia.com/cuda`. pages 30

[DAF11] Mayank Daga, Ashwin Aji, and Wu-chun Feng. On the Efficacy of a Fused CPU+GPU Processor for Parallel Computing. In *Symposium on Application Accelerators in High-Performance Computing*, Knoxville, Tennessee, USA, July 2011. pages 32

[DBB07] Romain Dolbeau, S Bihan, and F Bodin. *HMPP : A hybrid multi-core parallel programming environment*, pages 1–5. 2007. pages 31

[DDW06] Dennis Dalessandro, Ananth Devulapalli, and Pete Wyckoff. iwarp protocol kernel space software implementation. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 274–274, Washington, DC, USA, 2006. IEEE Computer Society. pages 17

[Dem97] Erik Demaine. A threads-only mpi implementation for the development of parallel programs. In *In : Proceedings of the 11th International Symposium on High Performance Computing Systems*, pages 153–163, 1997. pages 55

[DM05] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. Technical report, Tech. Rep., RedHat, Inc, 2005. pages 36

[DNW05] Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In *EuroPar*, Lisbonne, Portugal, September 2005. pages 81

[Dol] Dolphin Interconnect Solutions. `http://www.dolphinics.com`. pages 20

[DOS+07] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P Sadayappan, and Chau-Wen Tseng. Dynamic load balancing of unbalanced computations using message passing. *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007. pages 49

[(F.94] Pellegrini (F.). Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proceedings of SHPCC'94, Knoxville*, pages 486–493. IEEE, May 1994. pages 72

[FCLA06] M. Farreras, T. Cortes, J. Labarta, and G. Almasi. Scaling mpi to short-memory mpps such as bg/l. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 209–218, New York, NY, USA, 2006. ACM. pages 50

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998. pages 39

[Fra08] François Pellegrini. Scotch *and* LibScotch 5.1 *User's Guide*. ScAlApplix project, INRIA Bordeaux – Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800, August 2008. `http://www.labri.fr/perso/pelegrin/scotch/`. pages 72

[Ful99] S. Fuller. Motorola's altivec technology. *Networking & Computing Core Technology*, 1999. Motorola Inc. pages 10

[GCC99] F. García, A. Calderón, and J. Carretero. Mimpi : A multithread-safe implementation of mpi. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697, pages 674–674. Springer Berlin / Heidelberg, 1999. pages 55

[GFB+04a] Edgar Gabriel, Graham Fagg, George Bosilca, Thara Angskun, Jack Dongarra, Jeffrey Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph Castain, David Daniel, Richard Graham, and Timothy Woodall. Open mpi : Goals, concept, and design of a next generation mpi implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241, pages 353–377. Springer Berlin / Heidelberg, 2004. pages 44

[GFB+04b] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI : Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. pages 72

[GGZT11] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. Scalable memory use in mpi : A case study with mpich2. In Yiannis Cotronis, Anthony Danalis, Dimitrios Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in*

Computer Science, pages 140–149. Springer Berlin / Heidelberg, 2011. pages 49

[GNU] Gnu unified parallel c (gnu upc). `http://www.gccupc.org/`. pages 46

[Gog11] Brice Goglin. High-Performance Message Passing over generic Ethernet Hardware with Open-MX. *Elsevier Journal of Parallel Computing (PARCO)*, 37(2) :85–100, February 2011. pages 19

[GOM] The gnu openmp project. `http://gcc.gnu.org/projects/gomp/`. pages 71

[Gre] The Green500 Supercomputing Sites. `http://www.green500.org`. pages 31

[gri] Grid'5000. `http://www.grid5000.fr/`. pages 14

[Gro02] William Gropp. Mpich2 : A new start for mpi implementations. In Dieter Kranzlmüller, Jens Volkert, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2474, pages 37–42. Springer Berlin / Heidelberg, 2002. pages 44

[HK97] M. Howard and A. Kopser. Design of the tera mta integrated circuits. In *Gallium Arsenide Integrated Circuit (GaAs IC) Symposium, 1997. Technical Digest 1997., 19th Annual*, pages 14 –17, oct 1997. pages 24

[HUYK04] Shinichi Habata, Kazuhiko Umezawa, Mitsuo Yokokawa, and Shigemune Kitawaki. Hardware system of the earth simulator. *Parallel Comput.*, 30 :1287–1313, December 2004. pages 23

[IBM] IBM. Next generation posix threading. pages 37

[INT] Message passing interface (mpi) library from intel. `http://software.intel.com/en-us/articles/intel-mpi-library/`. pages 44

[Int06] Intel. Extending openmp to clusters. White Paper, 2006. pages 41

[Int11] Intel. Intel cilk plus specification, 2011. `http://software.intel.com/en-us/articles/intel-cilk-plus-specification/`. pages 40

[JMCA+11] Guohua Jin, J. Mellor-Crummey, L. Adhianto, W.N. Scherer, and Chaoran Yang. Implementation and performance evaluation of the hpc challenge benchmarks in coarray fortran 2.0. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1089 –1100, may 2011. pages 45

[Khr11] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1*, 6 January 2011. pages 31

[KM03] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *Micro, IEEE*, 23(2) :56 – 65, march-april 2003. pages 25

[KRS01] Christian Kurmann, Felix Rauch, and Thomas Stricker. Speculative defragmentation leading gigabit ethernet to true zero-copy communication. *Cluster Computing*, 4 :7–18, 2001. pages 17

[Ler99] Xavier Leroy. The linux threads library, 1999. `http://gallium.inria.fr/~xleroy/linuxthreads/`. pages 36

[lib] libtopology. http://libtopology.ozlabs.org/. pages 64

[LK99] Marius Christian Liaaen and Hugo Kohmann. Dolphin sci adapter cards. In *SCI : Scalable Coherent Interface, Architecture and Software for High-Performance Compute Clusters*, pages 71–87, London, UK, 1999. Springer-Verlag. pages 20

[LL97a] James Laudon and Daniel Lenoski. The sgi origin : a ccnuma highly scalable server. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 241–251, New York, NY, USA, 1997. ACM. pages 20

[LL97b] James Laudon and Daniel Lenoski. The sgi origin : a ccnuma highly scalable server. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 241–251, New York, NY, USA, 1997. ACM. pages 28

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988. pages 11

[Lov93] David B. Loveman. High performance fortran. *IEEE Concurrency*, 1 :25–42, 1993. pages 45

[MCO09] Guillaume Mercier and Jérôme Clet-Ortega. Towards an efficient process placement policy for mpi applications in multicore environments. In *EuroPVM/MPI*, volume 5759 of *Lecture Notes in Computer Science*, pages 104–115, Espoo, Finland, September 2009. Springer. pages 72, 103

[Moo75] Gordon E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13, 1975. pages 22

[Moo06] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *Solid-State Circuits Newsletter, IEEE*, 20(3) :33–35, 2006. pages 22

[MPIa] The message passing interface (mpi) standard. http://www.mcs.anl.gov/research/projects/mpi/. pages 43

[MPIb] Message passing interface (mpi) forum. http://www.mpi-forum.org/. pages 54

[MPIc] Mpi 3.0 hybrid programming working group. http://meetings.mpi-forum.org/mpi3.0_hybrid.php. pages 54

[MPId] Mpi-2 : Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html. pages 43

[MPIe] Mpi 3.0 standardization effort. http://meetings.mpi-forum.org/MPI_3.0_main_page.php. pages 43

[MPIf] Mpich2 : High-performance and widely portable mpi. http://www.mcs.anl.gov/research/projects/mpich2/. pages 44

[MS03] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *Micro, IEEE*, 23(2) :44 – 55, march-april 2003. pages 24

[Myra] Myricom. Mpich2-mx. `http://www.myricom.com/support/downloads/mx/mpich-mx.html`. pages 44

[Myrb] Myricom Inc. `http://www.myri.com`. pages 17

[Nam97] Raymond Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Université de Lille 1, January 1997. pages 37

[NBCL] The Ohio State University Network-Based Computing Laboratory. Mvapich2 (mpi-2 over openfabrics-ib, openfabrics-iwarp, psm, udapl and tcp/ip. `http://mvapich.cse.ohio-state.edu/overview/mvapich2/`. pages 44

[Nov06] D. Novillo. Openmp and automatic parallelization in gcc. *GCC Developers' summit*, june 2006. pages 71

[NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17 :1–31, August 1998. pages 45

[NRM+09] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. Scalatrace : Scalable compression and replay of communication traces for high-performance computing. *J. Parallel Distrib. Comput.*, 69 :696–710, August 2009. pages 81

[oEEE93] Institute of Electrical and CORPORATE Electronics Engineers, Inc. Staff. *IEEE Standard for Scalable Coherent Interface, Science : IEEE Std. 1596-1992*. IEEE Standards Office, New York, NY, USA, 1993. pages 20

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1) :80–113, 2007. pages 31

[Opea] Open Fabrics. `http://www.openfabrics.org`. pages 18

[Opeb] Openmp application program interface - version 3.0. pages 37

[OPEc] Openmpi : Open source high performance computing. `http://www.open-mpi.org/`. pages 44

[Ope08] The openmp api specification for parallel programming - version 3.0, 2008. `http://www.openmp.org`. pages 38

[pap] Performance application programming interface (papi). `http://icl.cs.utk.edu/papi/`. pages 129

[PF01] Ian Pratt and Keir Fraser. Arsenic : A user-accessible gigabit ethernet interface. In *IN PROCEEDINGS OF IEEE INFOCOM*, pages 67–76, 2001. pages 17

[PFH+02] F. Petrini, Wu-Chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network : high-performance clustering technology. *Micro, IEEE*, 22 :46–57, 2002. pages 44

[PKMW11] Victor Pankratius, Fabian Knittel, Leonard Masing, and Martin Walser. Openmpspy : Leveraging quality assurance for parallel software. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 124–135, Berlin, Heidelberg, 2011. Springer-Verlag. pages 84

[Pla] Plafrim. `https://plafrim.bordeaux.inria.fr/`. pages 97

[PLC95] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations : Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, volume 2, pages 1528–1557, San Diego, California, December 1995. Available from `http://www.c3.lanl.gov/PAL/publications/papers/Pakin1995:FM.pdf`. pages 17

[PLP] Portable Linux Processor Affinity. `http://www.open-mpi.org/projects/plpa`. pages 65

[PT98] Loic Prylli and Bernard Tourancheau. Bip : a new protocol designed for high performance networking on myrinet. In *In Workshop PC-NOW, IPPS/SPDP98*, pages 472–485. Springer-Verlag, 1998. pages 17

[PW96] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *Micro, IEEE*, 16(4) :42 –50, aug 1996. pages 24

[PWL+11] Robert Preissl, Nathan Wichmann, Bill Long, John Shalf, Stephane Ethier, and Alice Koniges. Multithreaded global address space communication techniques for gyrokinetic fusion applications on ultra-scale platforms. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 78 :1–78 :11, New York, NY, USA, 2011. ACM. pages 129

[Rei07] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. pages 40

[ren] Renater : Réseau national de télécommunications pour la technologie, l'enseignement et la recherche. `http://www.renater.fr/`. pages 14

[RLP11] Kaushik Ravichandran, Sangho Lee, and Santosh Pande. Work stealing for multi-core hpc clusters. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852, pages 205–217. Springer Berlin / Heidelberg, 2011. pages 49

[RPK00a] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming simd extensions on the pentium iii processor. *Micro, IEEE*, 20(4) :47 –57, jul/aug 2000. pages 10

[RPK00b] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming simd extensions on the pentium iii processor. *Micro, IEEE*, 20(4) :47 –57, jul/aug 2000. pages 24

[Rus78] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21 :63–72, January 1978. pages 23

[san10] Intel sandy bridge. Intel Processor Roadmap, 2010. pages 31

[SGDM94] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The pvm concurrent computing system : Evolution, experiences, and trends. *PARALLEL COMPUTING*, 20 :531–546, 1994. pages 43

[SGIa] Silicon Graphics Inc. `http://www.sgi.com`. pages 20

[SGIb] Altix uv : The world's fastest supercomputer. `http://www.sgi.com/products/servers/altix/uv/`. pages 20, 65

[Sod05] A. C. Sodan. Message-passing and shared-data programming models - wish vs. reality. *High Performance Computing Systems and Applications, Annual International Symposium on*, pages 131–139, 2005. pages 50

[SSB+95] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF : A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I :11–14, Oconomowoc, WI, 1995. pages 12

[SWP01] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. Emp : Zero-copy os-bypass nic-driven gigabit ethernet message passing. *SC Conference*, 0 :49, 2001. pages 17

[TBB] Intel threading building blocks for open source. `http://threadingbuildingblocks.org`. pages 40

[TEL95] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading : Maximizing on-chip parallelism. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 392 –403, june 1995. pages 25

[TG09] Rajeev Thakur and William Gropp. Test suite for evaluating performance of multithreaded mpi communication. *Parallel Comput.*, 35 :608–617, December 2009. pages 55

[Tho80] James E. Thornton. The cdc 6600 project. *Annals of the History of Computing*, 2(4) :338 –348, oct.-dec. 1980. pages 10

[TKI85] Hiroshi Tamura, Sachio Kamiya, and Takahiro Ishigai. Facom vp-100/200 : Supercomputers with ease of use. *Parallel Computing*, 2(2) :87 – 107, 1985. pages 23

[Top] Top500 Supercomputing Sites. `http://top500.org`. pages 10, 32

[TRF+11] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. EZTrace : a generic framework for performance analysis. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Newport Beach, CA, États-Unis, May 2011. Poster Session. pages 81

[TY01] Hong Tang and Tao Yang. Optimizing threaded mpi execution on smp clusters. In *IN PROC. OF 15TH ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING*, pages 381–392. ACM Press, 2001. pages 55

[UHYH90] N. Uchida, M. Hirai, M. Yoshida, and K. Hotta. Fujitsu vp2000 series. In *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth*

IEEE Computer Society International Conference., pages 4 –11, feb-2 mar 1990. pages 23

[UPC05] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005. pages 46

[VD08] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, Piscataway, NJ, USA, 2008. IEEE Press. pages 31

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages : a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Int'l Symp. on Computer Architecture*, Gold Coast, Australia, May 1992. pages 19

[WM85] T. Watari and H. Murano. Packaging technology for the nec sx supercomputer. *Components, Hybrids, and Manufacturing Technology, IEEE Transactions on*, 8(4) :462 – 467, dec 1985. pages 23

[XCA] Xcalablemp : Directive-based language extension for scalable and performance-aware parallel programming. `http://www.xcalablemp.org/`. pages 46

Table des figures