N° d'ordre : 3870

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **François TRAHAY**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

De l'interaction des communications et de l'ordonnancement de threads au sein des grappes de machines multi-cœurs

---

Soutenue le : 13 Novembre 2009

Après avis des rapporteurs :

M. Franck Cappello	Directeur de Recherche INRIA
M. Jean-François Méhaut .	Professeur des Universités

Devant la commission d'examen composée de :

M. Luc BOUGÉ	Professeur des Universités	Examinateur
M. Franck CAPPELLO	Directeur de Recherche INRIA	Rapporteur
M. Alexandre DENIS	Chargé de Recherche INRIA ..	Directeur de thèse
M. Olivier GLÜCK	Maître de conférence	Examinateur
M. Jean-François MÉHAUT	Professeur des Universités	Rapporteur
M. Raymond NAMYST ...	Professeur des Universités	Directeur de thèse
M. Jean ROMAN	Professeur des Universités	Président

– 2009 –

Thèse réalisée au sein de l'Équipe-Projet INRIA Runtime au LaBRI.

INRIA Bordeaux – Sud-Ouest
Batiment A29
351 cours de la Libération
F-33405 Talence Cedex

LaBRI
Unité Mixte de Recherche CNRS (UMR 5800)
351 cours de la Libération
F-33405 Talence Cedex

De l'interaction des communications et de l'ordonnancement de threads au sein des grappes de machines multi-cœurs

Résumé : La tendance actuelle des constructeurs pour le calcul scientifique est à l'utilisation de grappes de machines dont les nœuds comportent un nombre de cœurs toujours plus grand. Le modèle de programmation basé uniquement sur MPI laisse peu à peu la place à des modèles mélangeant l'utilisation de threads et de MPI. Ce changement de modèle entraîne de nombreuses problématiques car les implémentations MPI n'ont pas été conçues pour supporter les applications multi-threadées.

Dans cette thèse, afin de garantir le bon fonctionnement des communications, nous analysons ces problèmes et proposons un module logiciel faisant interagir l'ordonnanceur de threads et la bibliothèque de communication. Ce gestionnaire d'entrées/sorties générique prend en charge la détection des événements du réseau et exploite les multiples unités de calcul présentes sur la machine de manière transparente. Grâce à la collaboration étroite avec l'ordonnanceur de threads, le gestionnaire d'entrées/sorties que nous proposons assure un haut niveau de réactivité aux événements du réseau. Nous montrons qu'il est ainsi possible de faire progresser les communications réseau en arrière-plan et donc de recouvrir les communications par du calcul. La parallélisation de la bibliothèque de communication est également facilité par un mécanisme d'exportation de tâches capable d'exploiter les différentes unités de calcul disponible tout en prenant en compte la localité des données.

Les gains obtenus sur des tests synthétiques et sur des applications montre que l'interaction entre la bibliothèque de communication et l'ordonnanceur de threads permet de réduire le coût des communications et donc d'améliorer les performances d'une application.

Mots-clés : Calcul intensif, communications réseau, supports d'exécution, threads, multi-cœur.

About the interactions between communication and thread scheduling in clusters of multicore machines

Abstract : The current trend of constructors for scientific computation is to build clusters whose node include an increasing number of cores.The classical programming model that is only based on MPI is being replaced by hybrid approaches that mix communication and multi-threading. This evolution of the programming model leads to numerous problems since MPI implementations were not designed for multi-threaded applications.

In this thesis, in order to guarantee a smooth behavior of communication, we study these problems and we propose a software module that interact with both the threads scheduler and the communication library. This module, by working closely with the thread scheduler, allows to make communication progress in the background and guarantees a high level of reactivity to network events, even when the node is overloaded. We show that this permits to make communication progress in the background and thus to overlap communication and computation. The parallelization of the communication library is also made easier thanks to a task onloading mechanism that is able to exploit the available cores while taking data locality into account.

The results we obtain on synthetic application as well as real-life applications show that the interaction between the thread scheduler and the communication library allows to reduce the overhead of communication and thus to improve the application performance.

Keywords : High Performance Computing, High Performance Networking, runtime systems, threads, multicore.

Remerciements

Me voici donc rendu à la fin de cette grande aventure qu'est le doctorat et, comme le veut l'usage dans ce genre de situation, je remercie tous ceux qui m'ont aidé/soutenu/supporté pendant ma thèse. Cela représente un grand nombre de personnes et je ne pourrai pas citer tout le monde, je vais donc tâcher d'en oublier le moins possible.

Je souhaite tout d'abord remercier Raymond Namyst et Alexandre Denis pour m'avoir donné une chance de rejoindre le monde de la recherche et du parallélisme et pour m'avoir soutenu pendant les quatre ans qu'ont durés mon stage de Master recherche et mon doctorat. Je tiens également à remercier les membres de mon jury, Jean Roman, Luc Bougé, Olivier Glück et tout particulièrement Jean-François Méhaut et Franck Cappello qui ont accepté de relire ce document. Enfin, je remercie ceux qui ont pris du temps pour relire toutes ces pages et pour m'avoir fourni des conseils parfois avisés.

Je remercie l'ensemble de l'équipe Runtime, ceux qui m'ont accueilli ainsi que ceux qui prennent la relève. Merci aux "permanents" de #runtime pour m'avoir proposé des noms de logiciels tous plus débiles les uns que les autres ; merci à Élisabeth – ma co-bureau pendant 3 ans – et Cécile pour leurs discussions entre filles – parfait exercice de concentration – ainsi qu'à Gonnet qui, malgré son inadaptation flagrante aux tâches administratives, peut par sa parole vous faire découvrir des solutions à des bugs coriaces. Merci à PAW, le vieux sage, pour ses conseils souvent avisés et sa constante irrévérence ; merci à Sam pour avoir répondu très patiemment à toutes mes questions sur Marcel ; merci à Guigui pour m'avoir guidé dans le code de MPICH2 et pour sa DVDthèque nanardesque ; merci à Brice Goglin pour les réponses à mes questions concernant Myrinet ; merci à Nathalie pour avoir débarrassé mes articles en anglais d'un grand nombre de fautes de grammaire ; merci à Olivier pour ses relectures attentives ; merci à Marie-Christine pour son incroyable gentillesse. Je remercie également la relève de l'équipe : Broq et Jéjé – les inséparables –, Steph pour m'avoir fait découvrir le gâteau à la citrouille et Diak qu'on ne voit pas souvent mais auquel on pense quand même. Merci également à Ludo pour m'avoir fait découvrir NixOS. Ah... J'allais oublier Mateo que je remercie pour nous avoir consciencieusement signalé des bugs bien que parfois imaginaires. Merci aux autres Scallala Nico, Dams, Abdou et toute leur bande de mangeurs de galettes.

Et puisque les week-ends n'ont pas tous été passés à coder, un grand merci à Christophe et Amélie pour leurs soirées barbecue ; merci à Stan et aux autres Castors Joyeux pour les trop nombreuses soirées passées à arpenter un monde virtuel ; merci également à Camille pour les soirées Tequila.

Enfin, merci à toute ma famille pour m'avoir soutenu depuis toujours et pour avoir fait découvrir quelques spécialités angevines aux bordelais ; merci à la belle-famille pour avoir également participé au pot de thèse. Et merci à Juliette qui m'a supporté pendant toute cette thèse, même à l'approche des deadlines et pendant les très longues journées^W soirées passées devant un écran.

Table des matières

Table des figures

Chapitre 1

Introduction

La simulation numérique est, depuis les origines de l'informatique, un des piliers de la démarche scientifique. De nombreuses disciplines, que ce soit en climatologie, en géologie, en biologie moléculaire, ou en astrophysique, se reposent sur l'outil informatique pour modéliser les phénomènes étudiés. La mise à disposition de grands moyens de calcul permet la modélisation toujours plus fine des phénomènes physiques et l'expérimentation réelle laisse peu à peu la place à la simulation dans la démarche de recherche. Quels que soient les domaines, les besoins en puissance de calcul grandissent continuellement du fait des modélisations toujours plus fines. Afin d'assouvir ces besoins, les grands centres de calcul se sont très tôt dotés de calculateurs puissants. Une course à la puissance de calcul s'est ainsi lancée : la quantité de données à traiter a poussé les grands organismes – Département de l'Énergie aux États-Unis, Commissariat à l'Énergie Atomique (CEA) en France, etc. – à investir massivement dans l'achat de calculateurs toujours plus puissants. Cette course à la puissance a permis, au cours des années 1970, de développer des super-calculateurs dont l'architecture permet l'exécution de plusieurs flots simultanément. Ces calculateurs ont alors régné et ont connu un âge d'or dans les années 1980. Le coût important des super-calculateurs et leur non-extensibilité ont poussé la communauté scientifique à se tourner vers un nouveau type d'architecture : les grappes de calcul. Ces calculateurs sont constitués de machines classiques connectées par un réseau dédié. Avec le développement de technologies d'interconnexion performantes, les grappes de PC ont gagné en performances et ont permis de disposer d'une grande puissance de calcul pour un coût réduit. Ce rapport performance/prix allié à une capacité d'extension du calculateur ont rendu les grappes de plus en plus populaire au point qu'une grande majorité des calculateurs sont aujourd'hui des grappes [TOP]. Les technologies relativement courantes utilisées dans les machines composant les grappes de calcul ont récemment radicalement changé leur architecture. En quelques années, les processeurs multi-cœurs se sont propagés au point que les nœuds constituant les grappes d'aujourd'hui sont de véritables mini super-calculateurs.

La grande hétérogénéité des composants des grappes, que ce soit en terme de processeurs ou de technologies réseau, a rapidement poussé la communauté scientifique à concevoir des outils capables d'exploiter efficacement les grappes quel que soit le matériel sous-jacent. Ainsi, les concepteurs de réseaux d'interconnexion se sont alliés à des équipes académiques afin de mettre au point une interface standard permettant de communiquer à travers n'importe quel type de réseau. Grâce au standard MPI ainsi conçu, les applications peuvent s'appuyer sur des fonctionnalités communes aux diverses technologies réseau et assurer leur portabilité. Depuis la conception de MPI, de nombreuses implémentations ont vu le jour et, malgré les abstractions du standard, les performances délivrées sont très proches des performances brutes des réseaux.

L'évolution de l'architecture des grappes remet aujourd'hui l'interface MPI en question. L'introduction de processeurs multi-cœurs montre les limites de ce modèle et depuis quelques années, on observe une hybridation des applications qui commencent à mélanger les communications basées sur MPI avec d'autres modèles de programmation. Ce changement vers un modèle hybride pose de nombreux problèmes pour les implémentations MPI qui doivent s'adapter pour supporter ces nouveaux usages.

Objectifs et contributions

Nous identifions dans ce document les problèmes posés par la combinaison de communications et de multi-threading et concevons des outils capables d'exploiter efficacement les technologies réseau modernes quel que soit l'environnement d'exécution. En étudiant les problèmes posés par les évolutions du matériel utilisé dans les grappes de calcul, nous analysons les impacts des architectures matérielles actuelles et futures sur l'implémentation d'une bibliothèque de communication moderne adaptée aux utilisations hybrides.

La gestion des environnements combinant multi-threading et communication est une tâche complexe car elle nécessite une grande connaissance de l'état du système, que ce soit la topologie de la machine ou l'occupation des processeurs. Alors que l'ordonnanceur connaît précisément ces caractéristiques, l'état des réseaux d'interconnexion lui est inconnu. Nous proposons donc un modèle logiciel, nommé PIOMAN, chargé de gérer les interactions nécessaires entre la bibliothèque de communication et l'ordonnanceur de threads. Nous verrons comment ce gestionnaire d'entrées/sorties, de par sa collaboration étroite avec les bibliothèques de threads et de communication, prend en charge les problèmes liés au multi-threading pour que les communications restent efficaces, même lorsque les contraintes exercées par les threads de l'application sont fortes. Les mécanismes proposés par PIOMAN permettent aux bibliothèques de communication de sous-traiter une partie des traitements des communications. Nous avons modifié la bibliothèque de communication NEWMADELEINE afin qu'elle exploite les mécanismes fournis par PIOMAN. Il en résulte une bibliothèque de communication capable d'exploiter les différents processeurs présents sur la machine. Outre la détection des événements provenant du réseau qui peut être réalisée par les processeurs laissés inutilisés par l'application, les traitements plus généraux ont été parallélisés afin de réduire le coût des communications sur les performances des applications. L'exploitation des capacités de gestion des communications de PIOMAN a été validé par de vraies applications et nous avons également modifié MPICH2, une des implémentations MPI les plus répandues, afin de bénéficier du traitement parallèle des communications fourni par PIOMAN. Ainsi, les communications sont gérées à tous les niveaux en prenant en compte le multi-threading.

Organisation du document

Ce document est organisé autour de trois grandes parties. Nous présentons d'abord le contexte de travail. Les évolutions du calcul hautes performances, que ce soit d'un point de vue matériel ou applicatif, sont décrites dans le chapitre 2. Nous exposons ensuite dans le chapitre 3 les principaux problèmes liés aux évolutions des grappes de calcul. Enfin, le chapitre 4, nous décrivons un état de l'art des solutions proposées pour résoudre ces problèmes. Nous présentons également les implémentations MPI les plus courantes et les mécanismes qu'elles mettent en œuvre pour gérer ces problèmes. La deuxième partie de ce document est dédiée à notre proposition. Nous décrivons dans le chapitre 5 les mécanismes que nous proposons d'utiliser et nous présentons dans le chapitre 6 les détails de l'implémentation de ces

solutions dans PIOMAN, NEWMADELEINE et MPICH2. Enfin, nous validons notre démarche dans la troisième partie. Les mécanismes implémentés sont évalués dans le chapitre 7 afin de déterminer leur efficacité. Finalement, nous concluons ce document et discutons des perspectives dans le chapitre 8.

Publications

Les travaux que nous présentons dans ce document ont donné lieu à plusieurs publications dans des conférences. Ces publications concernent :
- l'architecture et les mécanismes mis en œuvre dans le gestionnaire d'entrées/sorties PIOMAN [1, 7, 9] ;
- la parallélisation de la bibliothèque de communication NEWMADELEINE et les interactions entre NEWMADELEINE et PIOMAN [2, 3, 4, 5, 8] ;
- l'intégration de PIOMAN dans le moteur de progression de MPICH2-NEMESIS [6].

Conférences internationales avec comité de lecture

[1] François TRAHAY, Alexandre DENIS, Olivier AUMAGE et Raymond NAMYST. « Improving Reactivity and Communication Overlap in MPI using a Generic I/O Manager ». Dans *EuroPVM/MPI'07 : Recent Advances in Parallel Virtual Machine and Message Passing Interface* Paris, septembre 2007.

[2] François TRAHAY, Élisabeth BRUNET, Alexandre DENIS et Raymond NAMYST. « A multithreaded communication engine for multicore architectures ». Dans *CAC 2008 :The 8th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, avril 2008.

[3] Élisabeth BRUNET, François TRAHAY et Alexandre DENIS. « A Multicore-enabled Multirail Communication Engine ». Dans *Proceedings of the IEEE International Conference on Cluster Computing*, Tsukuba, Japon, septembre 2008. note : poster session.

[4] François TRAHAY et Alexandre DENIS. « A scalable and generic task scheduling system for communication libraries ». Dans *Proceedings of the IEEE International Conference on Cluster Computing*, New Orleans, LA, septembre 2009.

[5] François TRAHAY, Élisabeth BRUNET et Alexandre DENIS. « An analysis of the impact of multithreading on communication performance ». Dans *CAC 2009 :The 9th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2009*, Rome, Italie, mai 2009.

[6] Guillaume MERCIER, François TRAHAY, Darius BUNTINAS et Élisabeth BRUNET. « NewMadeleine : An Efficient Support for High-Performance Networks in MPICH2 ». Dans *Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, Rome, Italie, mai 2009.

Conférences nationales avec comité de lecture

[7] François TRAHAY. « PIOMan : un gestionnaire d'entrées-sorties générique ». Dans *RenPar'18 : 18ème Rencontres Francophones du Parallélisme*, Fribourg, Suisse, février 2008.

[8] François TRAHAY. « Bibliothèque de communication multi-threadée pour architectures multi-cœurs ». Dans *RenPar'19 : 19ème Rencontres Francophones du Parallélisme*, Toulouse, septembre 2009.

Divers

[9] François TRAHAY. « Gestion de la réactivité des communications réseau ». Mémoire de DEA. Université Bordeaux 1, juin 2006.

Chapitre 2

Évolution des calculateurs parallèles

Sommaire

La course à la puissance de calcul engendrée par des applications scientifiques toujours plus gourmandes a fait évoluer le paysage du calcul intensif. Depuis le milieu des années 1990, l'architecture des calculateurs a radicalement changé, passant d'un modèle massivement parallèle à des structures distribuées. Toutefois, l'augmentation de la puissance de calcul délivrée par les calculateurs ne vient pas seulement de l'évolution de l'architecture. Les composants informatiques (processeurs, bus mémoire, cartes réseau, etc.) bénéficient sans cesse d'innovations technologiques améliorant leurs performances.

La principale amélioration ayant touché les grappes de calcul concerne les réseaux d'interconnexion. Les réseaux "classiques" basés sur la technologie ETHERNET sont généralement remplacés par des réseaux développés spécifiquement pour le calcul intensif tels que MYRINET, QUADRICS ou INFINIBAND. Ces réseaux offrent des performances bien meilleures que les réseaux de type ETHERNET : les latences sont désormais de l'ordre de la microseconde et les débits mesurés atteignent plusieurs gigaoctets par seconde.

Parmi les différents facteurs ayant permis une augmentation générale des performances, l'évolution des processeurs est la plus impressionnante. Du fait de la gravure de plus en plus fine des processeurs et des problèmes de dissipation thermique rendant difficile l'augmentation de la fréquence, les fondeurs ont exploité l'espace disponible en "collant" plusieurs processeurs sur une même puce. Ces processeurs multi-cœurs se sont répandus et perfectionnés : augmentation du nombre de cœurs, utilisation de caches partagés, etc.

L'objectif de ce chapitre est d'exposer le contexte dans lequel nous avons travaillé au cours de cette thèse. Les grappes de calcul sont tout d'abord présentées et les raisons de leur popularité sont détaillées. Les technologies réseaux modernes et les différentes solutions utilisées aujourd'hui dans les grappes de PC sont ensuite étudiées. L'évolution des architectures matérielles des calculateurs ainsi que les changements apparus au sein des processeurs sont présentés avant d'analyser l'évolution des modèles de programmation utilisés dans le paysage du calcul scientifique.

2.1 La domination des grappes de PC dans le paysage du calcul intensif

Le développement de la simulation numérique a entraîné une course à la puissance de calcul afin de toujours calculer plus précisément et plus rapidement. Cette course à l'équipement des grands consommateurs de puissance de calcul a permis de faire évoluer le paysage du calcul intensif de manière spectaculaire : en 10 ans, la puissance de calcul des calculateurs a été multipliée par 1000 [TOP]. Pour permettre une telle augmentation, il a fallu faire évoluer l'architecture des calculateurs : du fait des limitations techniques et des coûts de fabrication, les super-calculateurs ont cédé la place à des architectures distribuées. Les grappes de calcul qui se sont ainsi popularisées consistent en un ensemble de machines relativement standardes reliées entre elles par un réseau d'interconnexion. Les machines sont généralement toutes identiques afin de constituer une grappe homogène.

Le concept de grappe de calcul est relativement vieux, mais ce type de calculateur n'a commencé à se répandre qu'à la fin des années 1990. Le développement de technologies réseaux hautes performances telles que MYRINET a permis de réduire considérablement l'inconvénient principal par rapport aux super-calculateurs : le coût de l'interconnexion entre les processeurs. Grâce à ces technologies réseaux, les communications entre machines se font en un temps de l'ordre de la microseconde. Les coûts de communication sur un super-calculateur étant du même ordre de grandeur que pour une grappe de machines, l'intérêt des super-calculateurs est réduit.

L'amélioration des technologies d'interconnexion couplée à un rapport coût/performance particulièrement intéressant ont permis une adoption rapide des grappes de PC. Les centres de calcul ont donc pu s'équiper de solutions performantes à moindre coût. L'engouement pour les grappes de calcul est tel que ce type d'architecture représente aujourd'hui plus de 80 % des calculateurs les plus puissants au monde là où, il y a 10 ans, 95 % des machines étaient des super-calculateurs [TOP].

2.2 Interfaces de communication hautes performances pour grappes de calcul

Le développement rapide du protocole TCP/IP dans les années 1980 et sa domination pendant des années a poussé la communauté scientifique à améliorer continuellement ses performances. Les travaux ont pendant longtemps porté sur les capacités des cartes ETHERNET sur lesquelles se base généralement TCP/IP. Les débits obtenus sont alors passés de 10 Mbit/s à 100 Mbit/s (FASTETHERNET), puis 1000 Mbit/s (GIGAETHERNET) pour atteindre aujourd'hui 10 Gbit/s avec ETHERNET 10 G. Malgré les améliorations en terme de performances, les réseaux ETHERNET restent pénalisés par la pile logicielle TCP/IP généralement utilisée. En effet, TCP et IP sont des protocoles destinés à Internet et fournissent donc des mécanismes permettant de corriger le manque de fiabilité et de gérer les réseaux longue distance. Les la-

tences médiocres (plusieurs dizaines de microsecondes) obtenues avec de telles configurations pénalisent un grand nombre d'applications scientifiques très sensibles aux performances du réseau.

Le développement dans les années 1980 de technologies réseau spécifiques dites *"hautes performances"* a permis de réduire considérablement le coût des communications dans les applications scientifiques. Le développement de ces réseaux rapides fut fortement inspiré par la conception des réseaux d'interconnexion internes aux super-calculateurs : la localité des grappes de calcul, généralement situées au sein d'une même salle, permet de s'affranchir des contraintes liées aux communications longue distance dont souffre TCP/IP et ETHERNET (tolérance aux pertes, contrôle de congestion, etc.) Les solutions basées sur des technologies spécifiques et des protocoles de communication adaptés au calcul intensif se sont par la suite développés, apportant continuellement des gains de performances. Les latences obtenues aujourd'hui sont de l'ordre de la microseconde et les débits atteignent plusieurs gigaoctets par seconde.

2.2.1 Paradigmes de communication

L'exploitation des technologies réseau hautes performances a mené à la définition de nouveaux modes de communication. Alors que les réseaux ETHERNET sont généralement basés sur des interfaces de type SOCKET, les réseaux rapides ont entraîné le développement de communications par passage de message ou par accès direct à la mémoire distante.

Les communications par **passage de message** sont caractérisées par des échanges de données au cours desquels participent l'émetteur et le récepteur. Les deux parties doivent appeler des primitives d'envoi ou de réception de message pour participer à l'échange. Généralement, ces primitives sont disponibles sous plusieurs formes : primitives bloquantes/non-bloquantes et primitives synchrones/asynchrones. Lorsqu'une communication synchrone se termine, l'émetteur est assuré que le récepteur a commencé à recevoir le message. Ce type de communication permet une synchronisation implicite entre l'émetteur et le récepteur. À l'inverse, les communications asynchrones ne fournissent aucune garantie quant à l'état de l'autre participant à l'échange de données. Les primitives non-bloquantes initient les communications mais ne garantissent pas que le message ait été envoyé/reçu lorsque l'application reprend la main. Il est alors nécessaire de tester la terminaison de la requête par une scrutation ou d'attendre sa terminaison pour s'assurer de l'envoi ou de la réception du message. À l'inverse, les primitives bloquantes garantissent l'émission ou la réception du message.

Un aspect important du modèle de communication par passage de message concerne la détection de la terminaison d'une requête. Lorsque l'application teste la terminaison d'une requête réseau, une *scrutation* est effectuée afin d'interroger la carte réseau (voir Figure 2.1). Quand l'utilisateur utilise une primitive bloquante, la bibliothèque de communication sous-jacente peut scruter la carte réseau jusqu'à la terminaison de la requête réseau – il s'agit alors d'une *attente active* – ou attendre que la carte réseau envoie une interruption signalant la fin de la communication (voir Figure 2.2).

Le comportement des communications par accès direct à la mémoire distante (**RDMA** pour *Remote Direct Memory Access*) est radicalement différent. Chaque machine met à disposition des autres des zones dans son espace mémoire. Lors de l'initialisation, les informations concernant les zones réservées sont échangées afin que chaque nœud connaisse les adresses des données. Les échanges de données se font alors par lecture/écriture à distance. Ce paradigme de communication est donc résolument asynchrone puisque l'initiateur de la communication est l'unique intervenant.

La terminaison d'une communication est habituellement détectée par scrutation : lorsqu'une requête se termine, la carte réseau peut modifier une variable en mémoire (que ce soit dans la mémoire locale ou

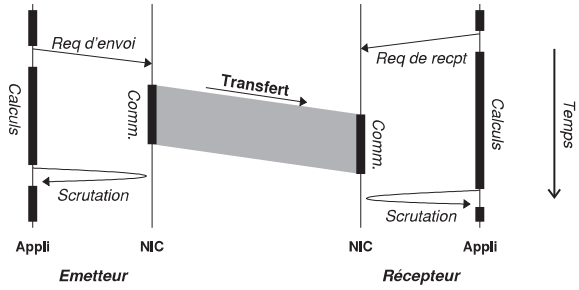

FIGURE 2.1 – *Détection de la terminaison d'un message par scrutation.*

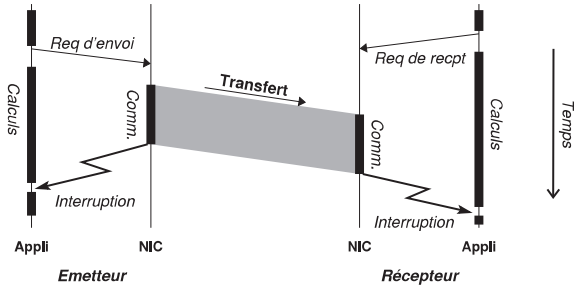

FIGURE 2.2 – *Détection de la terminaison d'un message grâce à un appel bloquant utilisant une interruption.*

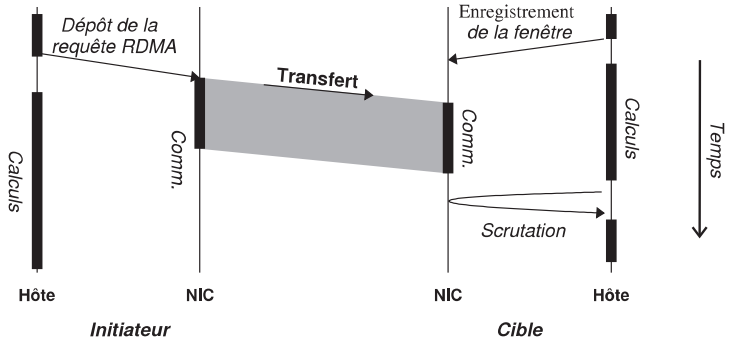

FIGURE 2.3 – *Communication par accès direct à la mémoire distante.*

dans la mémoire distante) pour signaler la fin de la communication. L'application peut alors consulter cette variable pour déterminer si une lecture/écriture demandée s'est terminée. Ce mécanisme s'applique également sur le nœud cible qui peut détecter qu'une autre machine a bien lu ou écrit certaines données. La Figure 2.3 illustre cette technique de détection des communications. Il est à noter que puisque la scrutation n'interroge pas directement la carte réseau mais une variable mémoire, le coût d'une telle scrutation est considérablement réduit par rapport à la scrutation effectuée dans un paradigme de passage de message. La détection de la terminaison d'une lecture ou d'une écriture est également possible en demandant à la carte réseau de générer une interruption. Le comportement est alors similaire au mécanisme d'interruption utilisé pour le passage de message.

2.2.2 Mécanismes clés pour améliorer les performances du réseau

Les différences de performances entre les réseaux à base de TCP/IP et les réseaux rapides peuvent être expliquées par plusieurs facteurs : l'absence de mécanisme assurant l'intégrité des communications longue distance avantage certes les réseaux hautes performances, mais ne peut pas à elle seule expliquer cette différence de performances. Nous présentons ici les mécanismes généralement employés dans les grappes de calcul et qui influencent le plus les performances du réseau.

Communications en espace utilisateur. L'accès à un périphérique est généralement réservé au système d'exploitation : quand une application souhaite accéder à ce périphérique, elle doit effectuer un

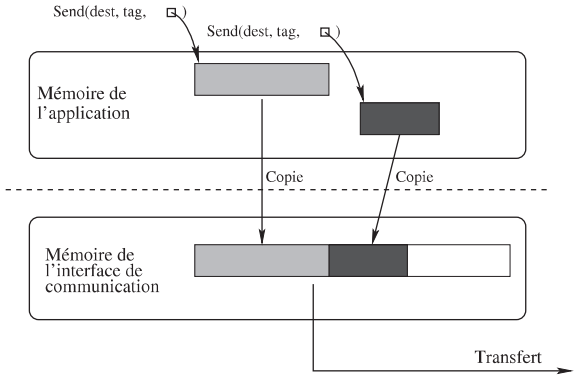

FIGURE 2.4 – *Copies intermédiaires en émission.*

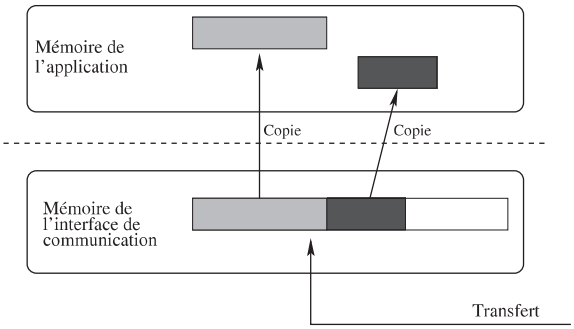

FIGURE 2.5 – *Copies intermédiaires en réception.*

appel système qui se chargera de l'opération. Ainsi, le système d'exploitation peut fournir une interface standard quel que soit la marque ou le modèle. Le passage obligé par le système d'exploitation permet également de s'assurer que l'opération effectuée sur le périphérique n'est pas illégale. L'envoi ou la réception d'un message *via* l'interface SOCKET du protocole TCP/IP nécessite donc un appel système pour dialoguer avec la carte réseau. Afin d'éviter le surcoût entraîné par le passage de l'espace utilisateur à l'espace noyau, des techniques d'*OS-bypass* ont été développées. Ces mécanismes de *communication en espace utilisateur* requièrent l'ajout d'une extension au noyau fournie par l'interface de communication. Au lancement de l'application, la mémoire de la carte réseau est projetée dans l'espace d'adressage du processus. L'application peut alors dialoguer avec la carte réseau sans passer par le système d'exploitation.

Lors de la création de l'interface de communication U-NET [EBBV95] – qui fut l'une des premières à utiliser ce mécanisme au milieu des années 1990 – le surcoût des appels systèmes était du même ordre de grandeur que la latence du réseau. Les communications en espace utilisateur étaient donc particulièrement bien adaptées. Même si ce surcoût est aujourd'hui réduit à une centaine de nanosecondes, l'amélioration des performances des réseaux fait que ce surcoût représente toujours une partie non négligeable de la latence. L'avenir de ce type de mécanisme dans le domaine des réseaux rapides est donc incertain.

Transferts de données entre la carte réseau et l'hôte. Une partie importante du coût des communications est due aux transferts de données entre la mémoire de la machine et la mémoire de la carte réseau. Les recopies mémoire sont parfois nécessaires pour garantir le fonctionnement des communications : le protocole de communication peut nécessiter d'ajouter des entêtes aux données ou de regrouper les données éparses en mémoire pour former des fragments contigus (voir Figure 2.4). En réception, les recopies surviennent principalement lorsque des données sont reçues alors que l'application n'a pas encore précisé où les stocker (voir Figure 2.5). Dans ce cas, la carte réseau copie les données reçues dans un tampon intermédiaire et lorsque l'application devient prête, les données sont déplacées à leur emplacement final.

Le coût de la recopie mémoire étant directement lié à la taille des données manipulées, il est important d'éviter les copies inutiles lorsque la taille des messages devient grande. Les recopies mémoire ne peuvent alors plus bénéficier des effets de cache et les performances se détériorent. L'utilisation d'un protocole de *rendez-vous* permet d'éviter ces copies. La Figure 2.6 présente les étapes d'un *rendez-vous* : l'émetteur commence par envoyer une *demande de rendezvous* (1) signalant que les données sont prêtes

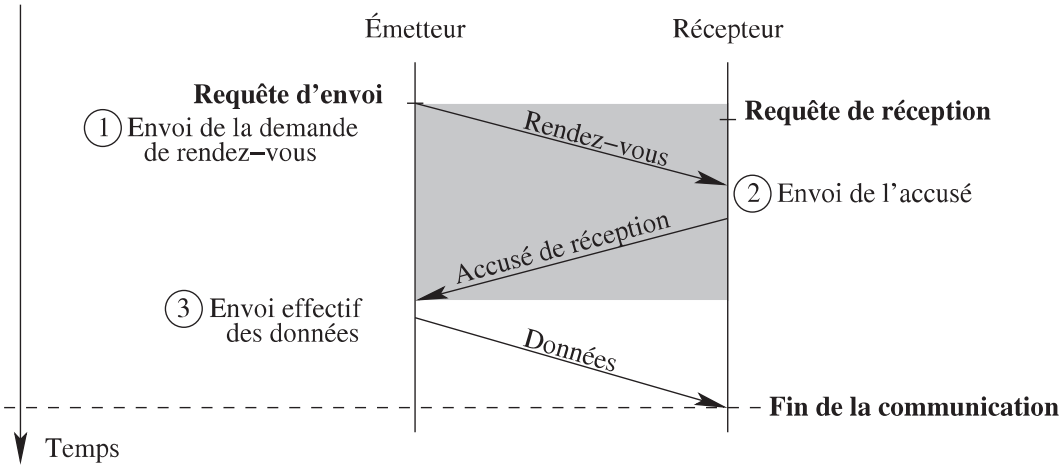

FIGURE 2.6 – *Protocole du rendezvous.*

à être envoyées. Lorsque le récepteur est prêt – c'est à dire lorsque la demande de *rendez-vous* est reçue et que l'application a précisé l'adresse mémoire à laquelle doivent être stockées les données – un *accusé de réception* est renvoyé (2). Lorsque l'émetteur reçoit cet accusé, il peut envoyer les données (3) qui seront directement reçues à l'adresse indiquée par le récepteur.

Ce mécanisme évite donc les recopies en réception – puisque la carte réseau peut écrire directement à l'adresse indiquée – mais aussi en émission : les entêtes nécessaires aux communications (les numéros de message, le type de message, etc.) peuvent être transmises en même temps que la demande de *rendez-vous*. L'utilisation d'un protocole de *rendez-vous* est toutefois coûteuse du fait de l'échange de messages préliminaires. Comme l'illustre la Figure 2.7 un compromis doit donc être trouvé entre payer le coût d'un *rendez-vous* et celui de la recopie des données. Du fait du faible coût de la recopie pour les petites tailles de données, ce mécanisme est utilisé pour les petits messages. Au-dessus d'un certain seuil, la recopie devient trop chère et le protocole de *rendez-vous* est alors utilisé. Par exemple, l'interface de communication bas niveau MX [Myr03] (décrite dans la section 2.2.3) a un seuil de *rendez-vous* fixé à 32 Ko.

Utilisation de transferts mémoire directs (DMA). Les échanges de données entre la machine hôte et la carte réseau sont inévitables et ont un impact non-négligeable sur les performances du réseau, notamment sur le débit. Le mécanisme de base, nommé PIO (*Programmed Input/Output*) utilise le processeur pour copier les données de la mémoire à la carte. Comme l'illustre la Figure 2.8, le processus copie les données vers la mémoire, dans une zone où la mémoire de la carte réseau est projetée. Bien que très efficace pour les transferts de données de petite taille, cette méthode souffre de la consommation excessive de processeur. En effet, la copie d'un grand volume de données par PIO monopolise le processeur pendant très longtemps. Afin d'augmenter le débit lors de la copie de données, il est maintenant courant d'utiliser des techniques d'accès direct à la mémoire appelées DMA (*Direct Memory Access*). Contrairement au mode PIO qui utilise le processeur pour copier les données, les transferts DMA se basent sur un contrôleur externe au processeur (voir Figure 2.9). Ici, le processeur envoie une requête à la carte réseau qui décide de récupérer les données par DMA. Le contrôleur DMA de la carte réseau envoie donc une requête au moteur DMA du contrôleur mémoire. Ce dernier se charge alors de la copie des données en arrière-plan [HP03] et le processeur n'est sollicité qu'à la fin du transfert. Ce mécanisme permet de réduire très fortement l'occupation du processeur. L'utilisation du contrôleur DMA pour transférer les

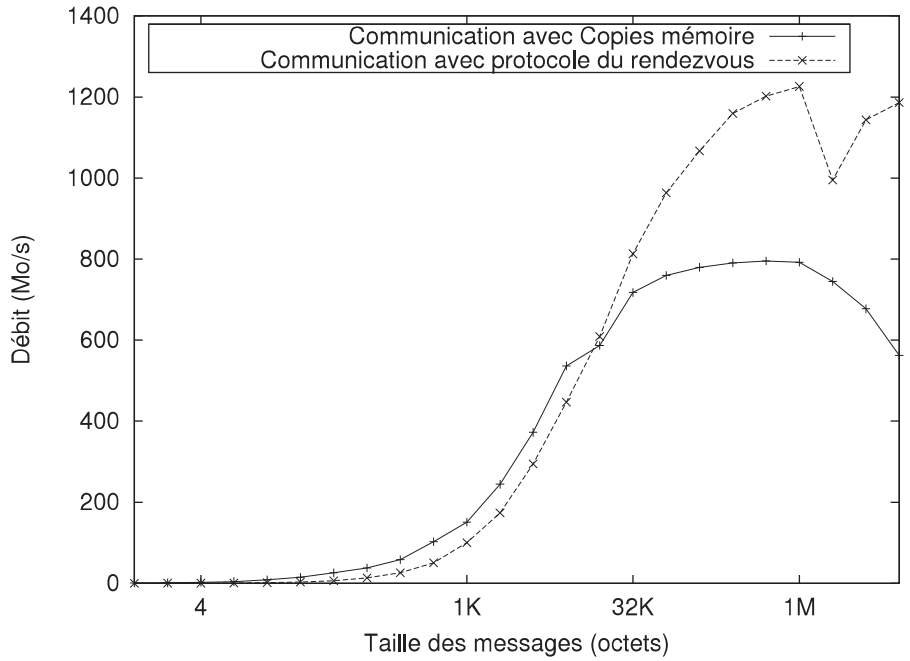

FIGURE 2.7 – *Débits mesurés sur* INFINIBAND. *Les transferts impliquant des copies mémoire sont efficaces jusqu'à un certain seuil (ici 16 ko). Ensuite, le protocole du rendezvous s'avère plus efficace.*

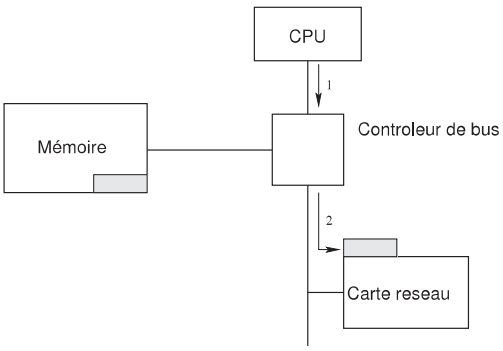

FIGURE 2.8 – *Transfert de données de la mémoire à la carte réseau par PIO.*

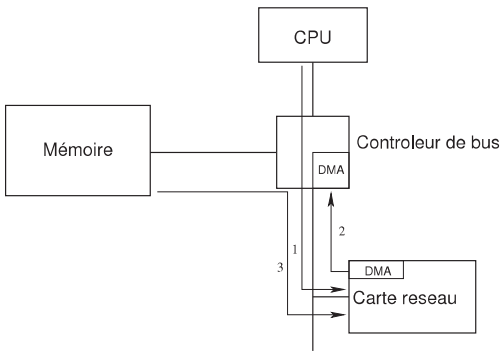

FIGURE 2.9 – *Transfert de données de la mémoire à la carte réseau par DMA.*

données de la mémoire à la carte réseau souffre toutefois d'un surcoût constant introduit par l'initialisation et la terminaison du transfert. Le mode PIO est donc généralement utilisé pour transférer les données de petite taille alors que les gros messages utilisent le mode DMA.

Au final, le programmeur de bibliothèques de communication a à sa disposition plusieurs méthodes pour transférer des données d'un nœud d'une grappe à un autre : transfert par PIO, par DMA, ou par DMA en utilisant un protocole de *rendez-vous*. En fonction des performances de chaque méthode, il faut donc choisir laquelle utiliser pour chaque échange de données. Les bibliothèques de communication bas niveau telles que MX cachent généralement ce choix et ne fournissent aux programmeurs qu'une seule méthode de communication. L'interface de communication choisit alors en interne la méthode à utiliser en fonction de la taille des données à transférer. D'autres bibliothèques de communication comme OFED [Opea] ou SISCI [Dol] exposent ce choix au programmeur.

2.2.3 Pilotes de réseaux hautes performances actuels

Nous présentons dans cette section les principales technologies réseau utilisées de nos jours afin d'en comprendre leurs caractéristiques et leur comportement. Cette liste de réseaux hautes performance n'est pas exhaustive et nous ne décrivons ici que les plus développés : MYRI-10G, QSNET II, INFINIBAND et ETHERNET. Les interfaces de communication généralement associées à ces technologies sont également présentées.

MX/MYRINET La société MYRICOM [Myr95] produit depuis 1995 un des leaders des réseaux rapides pour grappes de calcul nommé MYRINET [BCF+95]. Plusieurs familles de carte MYRINET ont vu le jour. Depuis 2005, MYRICOM distribue sa dernière génération de carte : MYRI-10G. Ces cartes réseau sont équipées d'un processeur RISC (nommé LANAI) tournant à 333 MHz, de 2 Mo de mémoire SRAM et d'un moteur DMA. Les spécifications libres des premières cartes réseau de MYRICOM ainsi que la facilité de reprogrammation ont rendu possible de nombreux travaux académiques sur des protocoles réseaux adaptés aux cartes MYRINET, notamment dans VMMC (*Virtual Memory-Mapped Communication* [DBL97]), FM (*Fast Messages* [PKC97]) ou BIP (*Basic Interface for Parallelism* [PT98]).

Actuellement, l'interface de communication bas-niveau permettant d'exploiter les cartes MYRINET se nomme *Myrinet eXpress* (MX) [Myr03]. La sémantique proposée par cette interface est très proche de celle du standard MPI : MX propose principalement des routines non-bloquantes qui cachent complètement les méthodes de transferts (présentées dans la section 2.2.2) utilisées en interne en fonction de la taille des messages. Cette sélection automatique du mode de communication ainsi que l'implémentation de mécanismes tels que l'enregistrement mémoire, la gestion des *rendez-vous* ou la progression des communications font de MX une bibliothèque de communication. MX supporte également les communications vectorielles et fournit un puissant système d'*appariement* (*Matching*). Cette technologie exhibe des latences de 2,1 μs et des débits de 9,9 Gbit/s sur un INTEL XEON X5460 cadencé à 3,16 GHz.

ELAN/QSNET II La société QUADRICS [Qua03] propose depuis 2003 des cartes QSNET II [BAPM] utilisables avec l'interface de communication par accès direct à la mémoire distante ELAN ou avec l'interface TPORT (*Tagged message Port*) qui se base sur ELAN. Le processeur puissant qui équipe ces cartes permet un traitement rapide des requêtes de communication, ainsi les performances en latence sont proches de la microseconde. Les cartes QSNET II intègrent également une grande quantité de mémoire (64 Mo de SDRAM), un mécanisme permettant de faire progresser les communications en arrière-plan

(un thread de progression s'exécute sur le processeur de la carte réseau) et un moteur DMA. En terme de débit, QUADRICS a innové en embarquant une MMU (*Memory Management Unit*) dans ses cartes réseau. Les adresses virtuelles sont donc traduites directement en adresses réelles. Les transferts DMA sont ainsi optimisés puisqu'il n'est plus nécessaire de punaiser les pages en mémoire virtuelle. L'absence d'enregistrement des pages mémoire a un impact important sur les débits obtenus et QSNET II atteint les 900 Mo/s. Le débit est ici limité par la fréquence du bus mémoire de la machine.

L'intégration d'une MMU dans la carte réseau n'est pas sans poser certains problèmes. Ainsi, le nombre de processus pouvant exploiter la carte simultanément est limité, ce qui devient problématique lorsque le nombre de processeurs augmente. De plus, les modifications à apporter au noyau pour gérer cette MMU supplémentaire sont importantes et les noyaux comportant autant de modifications sont parfois mal accueillis par les administrateurs des grandes plates-formes de calcul. L'avenir de cette technologie est d'autant plus incertain que, malgré des prix élevés, QUADRICS n'a pas su faire évoluer ses produits. En effet, QSNET II n'est disponible que sur des bus PCI-X et non PCI EXPRESS, limitant ainsi les débits atteignables. Enfin, le développement de la génération suivante de cartes QSNET, bien qu'entamé depuis quelques années, est au point mort depuis la fermeture de la société QUADRICS.

OFED/INFINIBAND INFINIBAND est une norme créée à la fin des années 1990 par un consortium de grands constructeurs de matériel informatique (parmi lesquels IBM, SUN, INTEL, HEWLETT-PACKARD, etc.) afin de définir l'architecture d'interconnexion du futur. Le but était de remplacer le bus PCI, mais également des systèmes d'accès au stockage ou au réseau [Inf00]. Suite au retrait d'INTEL du projet pour lancer PCI EXPRESS, les travaux se sont recentrés sur les réseaux hautes performances, que ce soit comme système de communication pour les grappes de calcul ou pour accéder aux systèmes de stockage [Pfi01].

Du fait des spécifications matérielles publiques, les équipements réseaux sont distribués par plusieurs constructeurs dont VOLTAIRE, TOPSPIN, ou CISCO. Pendant longtemps, les constructeurs ont distribué leur propre couche logicielle pour exploiter leur matériel, chaque constructeur proposant une interface de programmation spécifique. Le projet OPEN FABRICS [Opea] qui propose une implémentation libre est en passe d'unifier ces différentes versions de l'interface INFINIBAND. Bien qu'étant principalement destiné à l'exploitation de réseaux INFINIBAND, le projet OPEN FABRICS s'est récemment diversifié en exploitant également les réseaux de type RDMA sur IP grâce à des technologies comme IWARP [DDW06]. L'interface de communication VERBS est généralement utilisée pour exploiter les réseaux INFINIBAND. Cette interface suit le paradigme de communication par RDMA, exploitant ainsi les capacités des cartes réseau. D'autres interfaces de plus haut niveau comme DAPL [VRC+03] peuvent également être utilisées sur INFINIBAND.

Contrairement à MX, l'interface VERBS est de très bas-niveau et donc plus difficile à utiliser. Les traitements effectués par les autres interfaces telles que MX sont ici à la charge de l'utilisateur : enregistrement des zones mémoire, choix de la méthode de transfert, etc. Toutefois, une fois ces opérations maîtrisées, INFINIBAND affiche des latences de l'ordre de 1 μs et des débits atteignant 1,8 Go/s. Ces performances font d'INFINIBAND l'un des réseaux les plus utilisés aujourd'hui : près d'un tiers des grappes représentées au TOP500 en est équipé.

***/ETHERNET** Malgré les efforts fournis par les constructeurs de réseaux rapides, la technologie réseau dominante dans les grappes de calcul reste ETHERNET. Ce type de cartes réseau généralement bon marché équipe la quasi-totalité des grappes et la plupart du temps, seul le réseau ETHERNET est dis-

ponible. Hormis quelques modifications matérielles telles que les transferts DMA, certains mécanismes issus des réseaux rapides sont désormais employés couramment par ETHERNET. Le coût de la pile logicielle TCP/IP est le principal inconvénient de cette technologie et des efforts ont porté sur l'utilisation de protocoles alternatifs par ETHERNET.

La couche TCP est souvent évitée comme couche de transport pour les systèmes de stockage au profit de solutions comme FCoE [JD08] (*Fibre Channel over Ethernet*) ou AoE [Cas05] (*ATA over Ethernet*) qui s'appuient directement sur ETHERNET. Il en résulte un gain de performance grâce au court-circuitage de la pile logicielle TCP. Cette pile logicielle peut également être évitée dans les applications utilisant un paradigme de passage de message. À la fin des années 1990, le projet GAMMA [CC97] a ouvert la voie à l'utilisation directe d'ETHERNET. Toutefois, la modification des pilotes ETHERNET nécessaire à GAMMA a rendu sa diffusion difficile : la grande diversité des pilotes ETHERNET ainsi que les problèmes d'interopérabilité entre les paquets venant de la couche TCP/IP et ceux venant de GAMMA limitent la gamme de matériels supportés. Le projet OPEN FABRICS a ensuite développé IWARP [DDW06] permettant d'exploiter les capacités d'accès direct à la mémoire à distance de certaines cartes ETHERNET. Toutefois, le coût de telles cartes réseaux enlève tout l'intérêt d'utiliser des cartes ETHERNET. Le succès d'IWARP fut donc mitigé. Le protocole MX développé par MYRICOM a lui aussi été porté sur ETHERNET, mais avec une approche différente. En effet, les liens utilisés par les réseau MYRI-10G sont physiquement compatibles avec les liens ETHERNET. Aussi, dans un souci de standardisation de l'interface MX, une bibliothèque de communication nommée OPEN-MX a été créée afin de porter MX sur ETHERNET [Gog08]. Il est donc possible d'utiliser l'interface de communication bas niveau MX à la fois sur les réseaux MYRINET et ETHERNET. À la différence des autres techniques visant à porter un protocole sur ETHERNET, OPEN-MX se base sur la couche haute des pilotes réseau du noyau. Cette couche étant générique, toutes les cartes ETHERNET, peu importe leurs capacités, sont exploitables. L'interface OPEN-MX offre désormais de bonnes performances : la latence réseau peut atteindre de 5,8 μs et les débits mesurés atteignent les 10 Gbit/s (sur un réseau ETHERNET 10-Gigabit).

2.2.4 Vers une interface de communication standard

Malgré les excellentes performances délivrées par les interfaces de communication bas-niveau, leur utilisation par un programmeur lambda reste complexe. Les interfaces étant dédiées à un type de carte réseau, il devient difficile de développer une application à la fois portable et exploitant toutes les capacités du réseau. De plus, du fait des changements de technologie réseau relativement fréquents, la maintenance d'une telle application est problématique si de nouvelles interfaces de communication apparaissent. La disparité des interfaces de bas-niveau a suscité de nombreuses recherches visant à développer des bibliothèques de communication génériques capables d'exploiter efficacement une large gamme de technologies réseau. Parmi tous ces travaux, on peut citer FAST MESSAGE [PKC97], VMI [PP02], PM [TSH+00] ou MADELEINE [ABD+02]. En plus de l'abstraction des interfaces de communication, ces bibliothèques fournissent généralement certaines fonctionnalités destinées à certains types d'applications. Ainsi, bien que certaines se restreignent à fournir une interface générique de type passage de message (PM [THIS97] par exemple), d'autres proposent des modèles plus complexes permettant l'accès à la mémoire de manière transparente grâce à un modèle de mémoire partagée distribuée, ou encore l'appel de procédures à distance (MADELEINE [BNM98]).

Malgré les excellentes performances de ces bibliothèques et les fonctionnalités avancées qu'elles offrent, aucune ne s'est imposée et leur utilisation directement par une application reste aujourd'hui marginale. En effet, les développeurs d'applications se sont tournés vers une solution pouvant être moins perfor-

mante, mais apportant des garanties en termes de portabilité et de fonctionnalités. Cette solution fut développée au début des années 1990 par un consortium composé des principaux acteurs de la scène du calcul intensif : constructeurs, industriels et universitaires se sont réunis pour mettre au point un solution standard pour le développement d'applications parallèles. Il en a résulté la création de l'interface de communication MPI (*Message Passing Interface*), orientée passage de message, au milieu des années 1990 [Mes94]. Quelques années plus tard, MPI-2 [Mes96] est venu corriger quelques lacunes et étendre certains aspects. Ainsi, les communications collectives ou le support des communications *"One-Sided"* – c'est-à-dire les communications permettant un accès direct à une mémoire distante – ont été ajoutées au standard. La troisième version de MPI est en cours de discussion et le mécanisme de communications collectives devrait encore être étendu et les *One-Sided* améliorées.

MPI définit un ensemble de fonctionnalités et de spécifications complètement génériques. Le comportement de chaque fonction de l'interface est décrit et se veut indépendant de la technologie réseau sous-jacente. Depuis la création du standard, de nombreuses implémentations de MPI ont vu le jour : des implémentations libres permettant d'exploiter une large gamme de technologies réseau ou des implémentations commerciales destinées à certains réseaux. Des versions optimisées pour la plupart des technologies réseau ont également été développées : MPICH2-MX [Myr] pour les réseaux MYRINET, QUADRICS MPI [Qua] pour QSNET et MVAPICH2 [HSJ+06] pour les réseaux INFINIBAND. Les grands constructeurs de grappes de calcul fournissent généralement une implémentation de MPI optimisée pour leurs plates-formes. Ainsi, des implémentations comme BLUEGENE MPI [AAC+03] ou MPIBULL2 [Bul] bien que dérivées de versions génériques sont hautement optimisées pour le matériel utilisé dans les grappes vendues par leurs constructeurs. Des implémentations génériques de MPI permettent toutefois d'obtenir de bonnes performances pour une large gamme de technologies réseau ou d'architectures processeur. Par exemple, les performances obtenues par OPEN MPI [GWS05] ou MPICH2 [mpi07] ne sont que très légèrement dégradées par rapport à celles obtenues avec une implémentation spécifique à une technologie réseau.

2.3 Évolution générale de l'architecture des machines

Depuis quelques années, le grand public a vu apparaître les machines parallèles grâce aux processeurs multi-cœurs popularisés par les fondeurs INTEL et AMD. En conséquence, une part grandissante des développeurs d'applications découvre aujourd'hui la programmation parallèle et les problèmes de synchronisation entre threads. Pourtant, la programmation parallèle règne au sein de la communauté du calcul scientifique depuis de longues années. Il en a résulté de nombreux travaux qui permettent aujourd'hui de simplifier grandement le développement d'applications parallèles, notamment grâce à des outils comme OPENMP. Toutefois, la communauté scientifique est aujourd'hui confrontée à des problématiques bien plus complexes telles que la hiérarchisation des architectures.

2.3.1 La perpétuelle complexification des processeurs

Les progrès acquis en finesse de gravure par les constructeurs ont permis une complexification étonnante des processeurs. La place libérée sur les puces par cette miniaturisation a permis de doter les processeurs de mécanismes extrêmement évolués tels que la prédiction de branchements, d'améliorer les capacités de calcul flottant ou d'augmenter la taille de la mémoire cache. Le développement de telles optimisations est toutefois coûteux et les techniques telles que la prédiction de branchement ne peuvent plus être

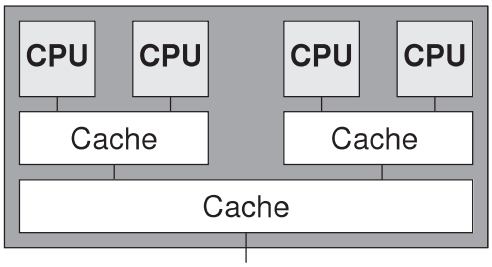

FIGURE 2.10 – *Puce multi-cœur.*

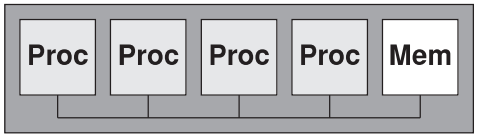

FIGURE 2.11 – *Exemple d'architecture SMP.*

raffinées. Afin d'augmenter encore la puissance de calcul, les fondeurs n'ont désormais d'autre choix que de graver plusieurs processeurs sur une même puce.

Puisque plusieurs processeurs sont gravés sur une même puce, il est courant de leur faire partager un cache. Ainsi, les communications par mémoire partagée s'effectuent beaucoup plus rapidement. Les puces embarquant plus de deux cœurs utilisent ainsi parfois une hiérarchie de cache, comme celle décrite Figure 2.10.

L'engouement pour cette technologie multi-cœur est telle qu'aujourd'hui tous les grands constructeurs proposent des puces bicœurs, quadricœurs, voire octocœurs. L'évolution actuelle des processeurs semble mener à une course au nombre de cœurs : des puces équipées de 16 cœurs sont déjà prévues et certains étudient des processeurs embarquant plusieurs dizaines ou centaines de cœurs [HBK06, VHR+08].

2.3.2 Des machines monoprocesseurs aux machines NUMA

Alors qu'au début de l'ère de l'informatique, les machines monoprocesseurs régnaient, les années 1960 ont vu apparaître les premières machines multiprocesseurs. L'architecture SMP (*Symmetric MultiProcessing*) consiste à relier plusieurs processeurs à un même bus mémoire, comme le montre la Figure 2.11. L'utilisation de plusieurs processeurs permettant d'augmenter simplement la puissance d'un calculateur, cette technique fut employée dans la plupart des super-calculateurs pendant 30 ans. Le bus mémoire est partagé par tous les processeurs. En conséquence, une augmentation du nombre de processeurs implique une augmentation de l'utilisation du bus qui, rapidement, devient saturé. Une solution courante consiste à soulager le bus mémoire en répartissant les processeurs sur plusieurs bancs mémoire. Comme le montre la Figure 2.12, sur ces architectures **NUMA** (*Non-Uniform Memory Access*), la mémoire est répartie sur les différents **nœuds** reliés entre eux par un réseau d'interconnexion. Les processeurs peuvent donc accéder à l'ensemble de la mémoire, mais de manière non-uniforme : les temps d'accès dépendent de l'emplacement mémoire auquel un processeur accède. La latence d'accès à une donnée distante sera plus élevée que celle pour une donnée locale. Les architectures NUMA permettent donc aux calculateurs d'héberger toujours plus de processeurs. Par exemple, les machines SGI ALTIX peuvent aujourd'hui comporter plus de 1000 processeurs [WRRF]. La complexification des applications qui doivent

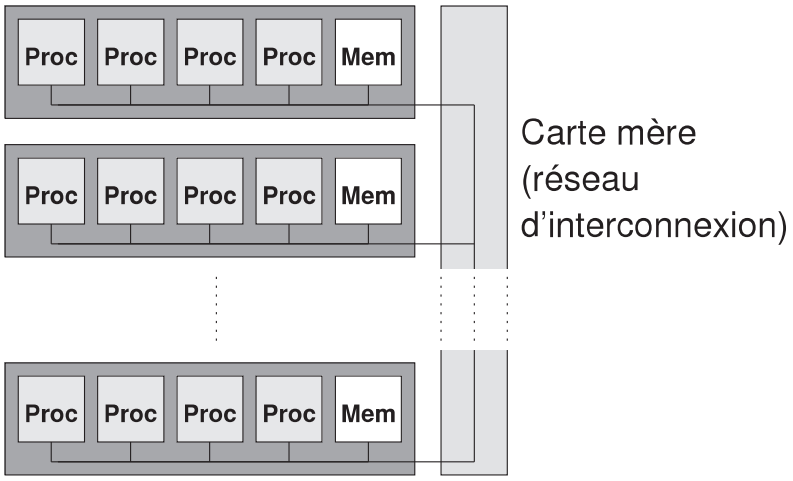

FIGURE 2.12 – *Exemple d'architecture NUMA.*

maintenant gérer la localité des données est le prix à payer pour cette augmentation de la puissance de calcul.

Alors que les architectures NUMA ont pendant longtemps été réservées aux super-calculateurs, ce type de technologie arrive aujourd'hui dans les machines constituant les grappes de calcul. L'augmentation de la puissance des processeurs et l'incorporation de plusieurs cœurs dans chaque puce entraînent une utilisation intensive du bus mémoire. Afin de réduire la contention sur ce bus, AMD utilise pour ses processeurs OPTERON la technologie HYPERTRANSPORT. Comme le décrit la Figure 2.13, les machines comportant plusieurs processeurs OPTERON sont alors attachées à différents bancs mémoire [KMAC03]. Cette architecture NUMA permet de répartir les accès mémoire et ainsi de limiter la contention sur les bus mémoire. La technologie HYPERTRANSPORT implique également un accès non-uniforme aux bus d'entrées/sorties. En effet, ceux-ci sont attachés à un des nœuds NUMA et l'accès à un périphérique d'entrées/sorties (une carte réseau par exemple) peut donc nécessiter la traversée d'un ou plusieurs liens HYPERTRANSPORT. Cela peut avoir un impact important sur les performances des entrées/sorties, comme le montrent les travaux sur les effets NUIOA [MG07]. Récemment, INTEL a adopté une approche similaire pour sa dernière génération de processeurs. La technologie QPI (QUICKPATH INTERCONNECT) [Sin08] permet de lier plusieurs processeurs de type NEHALEM.

2.3.3 Évolution des grappes de calcul

Pendant longtemps, les super-calculateurs ont consisté en une accumulation de processeurs reliés entre eux par un ou plusieurs bus mémoire. L'augmentation de la puissance des processeurs et l'amélioration des bus mémoire permettaient alors de construire des machines toujours plus puissantes et ainsi d'assouvir les besoins en puissance de calcul. Toutefois, la conception de ces calculateurs souffrait des coûts élevés engendrés par le développement d'un réseau d'interconnexion toujours plus complexe. Maintenir la cohérence mémoire sur ce type de machine lorsque le nombre de processeurs augmente est en effet une tâche extrêmement délicate.

Avec le développement de technologies réseau dont les performances ne cessent de s'améliorer, les super-calculateurs ont progressivement disparu au profit des grappes de PC au point qu'aujourd'hui plus de 80 % des calculateurs sont des grappes de calcul comme le montre la Figure 2.14. Cette évolution a été

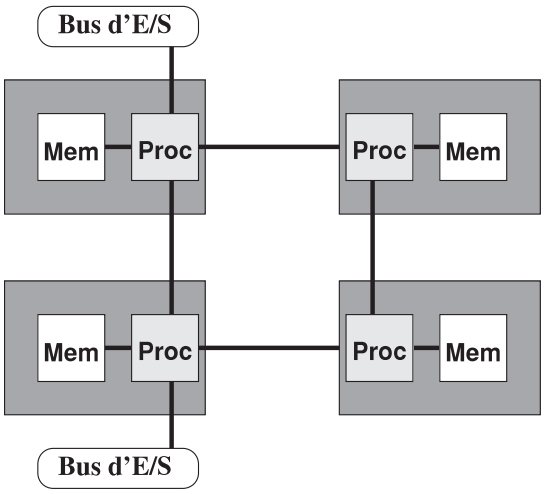

FIGURE 2.13 – *Processeurs* OPTERON *reliés par la technologie* HYPERTRANSPORT.

rendue possible par les innovations technologiques et par la conception de réseaux hautes performances tels que MYRINET à partir de la fin des années 1990. À cette époque, les grappes étaient majoritairement constituées de nœuds embarquant quelques processeurs – généralement deux – se partageant l'accès à une carte réseau hautes performances.

Depuis quelques années, l'architecture des nœuds composant les grappes de calcul a évolué grâce à l'apparition des processeurs multi-cœurs. Les nœuds de calcul comportent encore un nombre limité de processeurs, mais ceux-ci possèdent plusieurs cœurs de calcul. Le nombre d'unités de calcul dans chaque nœud augmente donc et les nœuds comportant 8 cœurs ou plus se démocratisent. Cette augmentation du nombre de cœurs et donc du nombre de flots de communication implique une utilisation accrue du réseau qui peut devenir le goulet d'étranglement du système. Pour cette raison, certaines grappes intègrent désormais plusieurs cartes réseau hautes performances dans chaque nœud. La bande-passante réseau totale est donc accrue. La bande-passante mémoire est également soumise à rude épreuve. Les architecture NUMA telles que les technologies HYPERTRANSPORT ou QPI sont donc de plus en plus utilisées pour répartir les accès aux données sur plusieurs bancs mémoire. Le T2K OPEN SUPERCOMPUTER [Nak08] installé à l'Université de Tokyo est un excellent exemple d'une telle architecture : chaque nœud comporte quatre processeurs OPTERON QUAD-CORE reliés par HYPERTRANSPORT. Les seize cœurs se partagent l'accès à quatre cartes réseau MYRI-10G.

L'évolution actuelle du matériel tend vers une augmentation massive du nombre de cœurs par processeur et l'utilisation quasi-généralisée d'architectures NUMA. Il en résultera une forte hiérarchisation des grappes de calcul : les cœurs– potentiellement *hyperthreadés* – seront regroupés dans des puces à hiérarchie de caches appartenant à des nœuds NUMA. Les grappes peuvent également être décomposées en *"sous-grappes"* afin de réduire la contention sur le réseau. Une architecture similaire fortement hiérarchique est d'ores et déjà présente dans les calculateurs BLUE GENE/P [AUW08]. De plus, la tendance actuelle à l'intégration d'accélérateurs – que ce soit une carte graphique programmable ou un processeur hétérogène comme le Cell/BE équipant le super-calculateur ROADRUNNER [BDH+08] – et les performances obtenues laissent présager d'une utilisation accrue de ces techniques. Toutefois, ces accélérateurs, bien qu'extrêmement efficaces pour certaines classes d'applications, ne sont pas une solution ultime au calcul scientifique. Ils ne devraient donc se développer que sur certaines plates-formes.

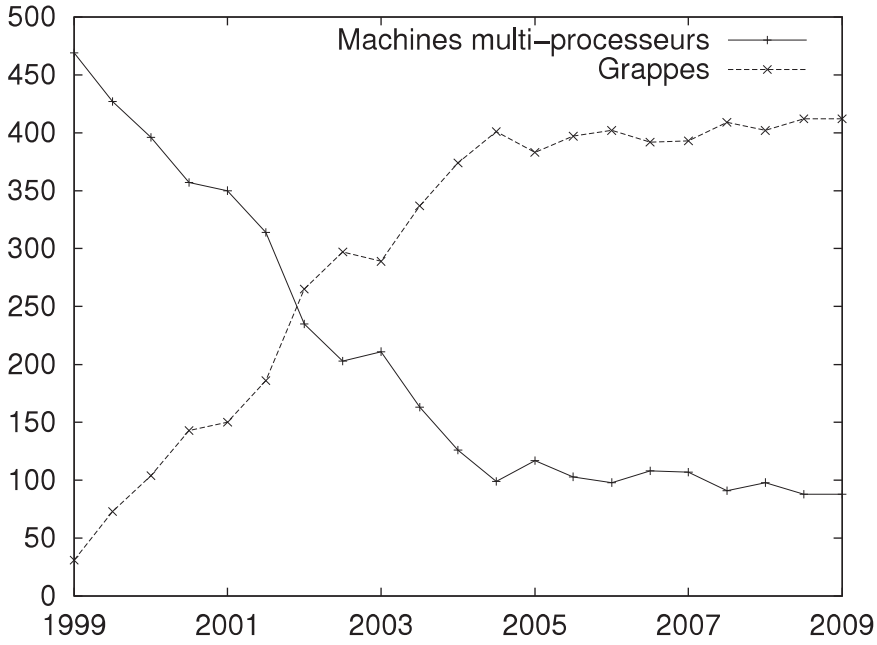

FIGURE 2.14 – *Évolution de l'architecture des plus puissants calculateurs [TOP].*

2.4 Évolution des modèles de programmation

Depuis maintenant presque 20 ans, le calcul scientifique est largement dominé par les interfaces à base de passage de message comme MPI. L'interface standardisée de MPI et la grande diversité des implémentations qui permettent d'exploiter efficacement une large gamme de matériels ont permis une adoption massive du standard par la communauté scientifique. L'utilisation classique de MPI consiste à lancer un processus sur chaque cœur de la grappe et les faire communiquer *via* l'implémentation MPI. La Figure 2.15 illustre ce type d'utilisation de MPI. Les messages sont ici échangés par le réseau si les processus sont placés sur des nœuds différents ou en utilisant un segment de mémoire partagée entre les processus dans le cas contraire. Le mécanisme de communication par échange de messages en mémoire partagée permet de profiter de la localité des processus et d'atteindre des débits élevés ainsi que des latences de quelques centaines de nanosecondes sur des machines modernes. Cette différence de performances entre les transferts réseau et les communications en mémoire partagée a entraîné de nombreux travaux sur le placement des processus MPI. Ainsi, des logiciels comme SCOTCH [CP08] tentent de rapprocher les processus communiquant beaucoup en les plaçant sur les même nœuds d'une grappe. Le surcoût lié aux communications peut alors être considérablement réduit [MCO09].

À l'heure où le nombre d'unités de calcul par nœud augmente rapidement, ce modèle de programmation commence à être remis en question. En effet, les communications en mémoire partagée, bien que très efficaces, souffrent de problèmes de passage à l'échelle. L'espace mémoire nécessaire aux communications en mémoire partagée entre les processus d'un même nœud augmente fortement lorsque le nombre de processus par nœud grandit. À l'initialisation, chaque processus MPI doit allouer un segment de mémoire à partager avec chaque autre processus local. Lorsque le nombre de processus augmente, l'espace mémoire nécessaire aux communications locales augmente donc quadratiquement.

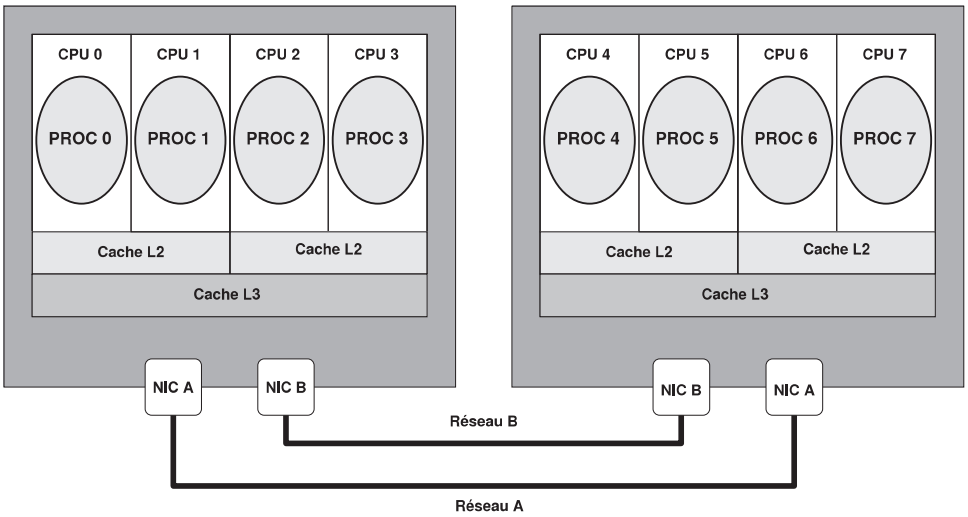

FIGURE 2.15 – *Utilisation "classique" de MPI : un processus MPI est lancé sur chaque unité de calcul.*

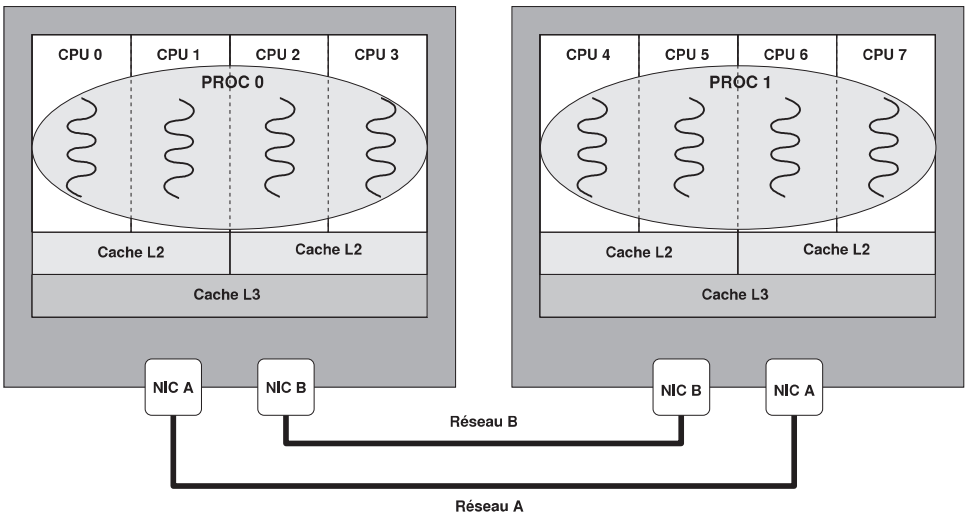

FIGURE 2.16 – *Exemple d'utilisation de MPI avec des threads : un processus MPI est lancé sur chaque processeur et chaque processus exécute un thread par cœur.*

Une solution à ce problème de passage à l'échelle est de réduire le nombre de processus MPI par nœud. La Figure 2.16 présente un exemple de solution : un processus MPI est lancé sur chaque processeur et les différents cœurs sont exploités grâce à des threads. Cette solution hybride qui mélange des threads et des processus MPI peut par exemple se baser sur l'interface de programmation OpenMP [Opeb]. Outre la résolution des problèmes de passage à l'échelle, les solutions hybrides mélangeant threads et processus MPI permettent une granularité plus fine : les systèmes à base de tâches [ACD+08] et le vol de travail sont facilités.

Bien que de telles solutions mélangeant processus MPI et threads se développent depuis près de 10 ans [SB01, CE00], l'impact des threads sur les communications MPI reste problématique. En effet, le standard MPI a été développé dans l'optique d'une utilisation par des programmes monothreadés. L'arrivée tardive de la notion de threads dans MPI et le manque d'intérêt que la plupart des bibliothèques de communication portent aux applications multi-threadées font persister les problèmes liés aux threads. Nous présentons dans le chapitre suivant quelques unes de problématiques dues à l'utilisation de bibliothèques de communication par des applications multi-threadées.

Chapitre 3

Impact des processeurs multi-cœurs sur les bibliothèques de communication

Sommaire

Au cours des 15 dernières années, l'utilisation de grappes de machines s'est fortement développée au point que 80 % des calculateurs sont aujourd'hui des assemblages de machines standard. Cette démocratisation a été rendue possible par les progrès en matière de technologies réseau hautes performances et par la création du standard MPI. Ce standard parfaitement adapté aux grappes traditionnelles a donné naissance à de nombreuses implémentations exploitant efficacement les différentes technologies réseau. Les latences et débits obtenus pour des opérations de type communication point-à-point et les algorithmes employés pour les communications collectives font des implémentations MPI d'aujourd'hui des bases solides pour le développement d'applications scientifiques.

Toutefois, le standard MPI a été conçu pour des plates-formes très peu hiérarchiques : les grappes de machines comportant une ou deux unités de calcul étaient ciblées. Par conséquent, les différents processus MPI ne sont pas hiérarchisés. Avec le développement des processeurs multi-cœurs, ce modèle devient caduque. L'architecture des grappes est aujourd'hui fortement hiérarchique : les machines comportent plusieurs processeurs embarquant chacun plusieurs cœurs partageant ou non des mémoires cache. L'utilisation de processus pour exploiter les unités de calcul d'une telle grappe n'est donc pas forcément adaptée. En effet, les relations entre les différentes unités de calcul ne sont pas homogènes – les échanges de données entre deux cœurs partageant un cache sont fortement réduits par rapport aux échanges entre deux machines – et les processus doivent donc être hiérarchisés.

L'utilisation de threads pour exploiter les différents cœurs d'une machine permet de calquer les domaines traités par les processus sur l'architecture hiérarchique de la grappe. Les threads d'un processus se partagent un domaine – et donc les données qui lui sont propres – et sont localisés sur un même niveau de la hiérarchie (généralement le même nœud NUMA ou la même machine). Toutefois, l'intégration des threads dans les processus MPI pose problème à une majorité d'implémentations MPI ou plus généralement aux bibliothèques de communication. Bien que la norme MPI définisse des fonctionnalités pour le support d'applications multi-threadées, bien peu d'implémentations les supportent correctement. Pour les quelques versions ayant implémenté ces mécanismes, ils ne sont malheureusement que très légèrement testés et il n'est pas rare de voir une application multi-threadée s'arrêter à cause d'une erreur dans la bibliothèque de communication.

Des progrès sont donc encore à faire en terme de support des applications multi-threadées. Plus généralement, le développement des processeurs multi-cœurs rend indispensable l'exploitation des multiples unités de calcul par les bibliothèques de communication. Pour être efficace et réduire le coût des communications, celles-ci doivent donc tirer profit du multi-threading et non l'endurer. Ce chapitre a pour but d'exposer les principaux problèmes liés aux évolutions des grappes de calcul, notamment dus à l'augmentation du nombre de cœurs par nœud. Nous étudions tout d'abord la progression des communications et analysons les problèmes de performance pouvant en découler. Nous présentons ensuite les pièges à éviter pour fournir un support des applications multi-threadées dans une bibliothèque de communication. Enfin, nous exposons les opportunités de répartition de la charge de calcul engendrée par les bibliothèques de communication.

3.1 Recouvrement des communications par le calcul

Les performances de certaines applications s'exécutant sur des grappes de calcul dépendent fortement de la vitesse des échanges de messages. Le temps nécessaire à la transmission d'un message n'étant généralement pas compressible à l'infini, le recouvrement des communications par du calcul permet de réduire l'impact des échanges de messages sur le temps d'exécution d'une application. Cette section décrit les mécanismes généralement employés pour cacher le coût des communications ainsi que les problèmes qu'impliquent ces techniques.

3.1.1 Utilisation de primitives non-bloquantes pour cacher le coût des communications

Le recouvrement des communications par du calcul est généralement réalisé à partir de primitives de communication non-bloquantes telles que `MPI_Isend` ou `MPI_Irecv` (ou leurs équivalents implémentés dans des bibliothèques de communication). La communication est alors simplement initialisée et on redonne la main à l'application. Ainsi, l'échange de messages peut avoir lieu pendant que l'application poursuit ses calculs. Ce type de mécanisme permet donc de cacher le coût des communications et ainsi de réduire le temps d'exécution global d'un grand nombre d'applications [SBKD06, SSB+08, BU04]. On peut ici définir un taux de recouvrement permettant d'évaluer la quantité de communication pouvant être effectuée en arrière-plan. Ce taux τ est généralement défini par :

$$\tau = \frac{T_{calcul}}{T_{calcul} + T_{com}}$$

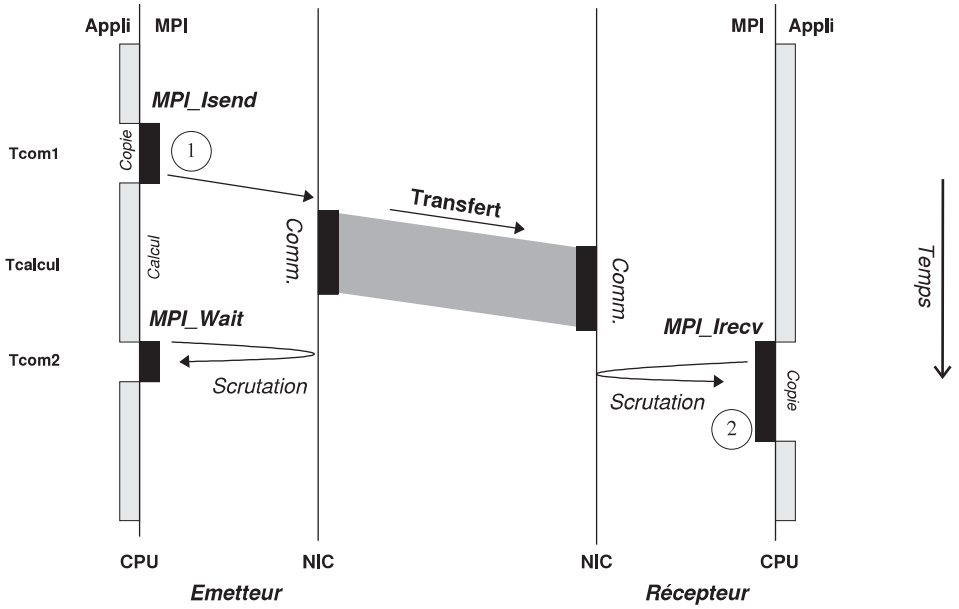

FIGURE 3.1 – *Recouvrement des communications par le calcul.*

où T_{calcul} représente le temps passé à calculer pendant une période de temps donnée et T_{com} le temps passé dans la bibliothèque de communication pendant ce laps de temps. Ainsi, un *taux de recouvrement* égal à 1 signifie que les échanges de messages se font sans aucun surcoût pour l'application qui peut passer son temps à calculer. À l'inverse, un taux de recouvrement nul signifie que la bibliothèque de communication monopolise le processeur et que l'application ne peut pas calculer pendant la communication. On cherchera donc à maximiser ce taux de recouvrement, par exemple en réduisant l'occupation du processeur pour traiter les communications.

Du point de vue de la bibliothèque de communication, le recouvrement est plus complexe. Comme nous l'avons montré dans la section 2.2.2, les interfaces de communications proposent plusieurs modes de communication (transfert des données par copie ou par DMA, protocole de *rendez-vous*, etc.) permettant d'exploiter au mieux le matériel. Ainsi, la transmission de messages de petite taille nécessite généralement une copie. Ces opérations, bien qu'elles permettent d'atteindre des vitesses de transfert élevées, souffrent d'une forte occupation du processeur. Le recouvrement des communications par du calcul pour des messages de petite taille s'en trouve perturbé, comme le montre la Figure 3.1. En effet, le temps passé dans la bibliothèque de communication pour transférer les données jusqu'à la carte réseau est élevé du fait de la copie. Ces opérations coûteuses en temps processeur peuvent survenir du côté de l'émetteur (1) lors de la copie des données de la mémoire de la machine vers la carte réseau. Si le récepteur n'est pas prêt lorsque les données arrivent, celles-ci sont stockées à un emplacement mémoire temporaire. Quand l'application demande les données, il faut donc les copier à leur emplacement final (2), monopolisant le processeur pour une période potentiellement longue.

Bien que certaines primitives de communication soient non-bloquantes, elles peuvent monopoliser le processeur pendant un long moment, réduisant le *taux de recouvrement*. Le développeur d'application, cherchant à recouvrir les communications par du calcul n'obtiendra donc pas forcément les performances espérées.

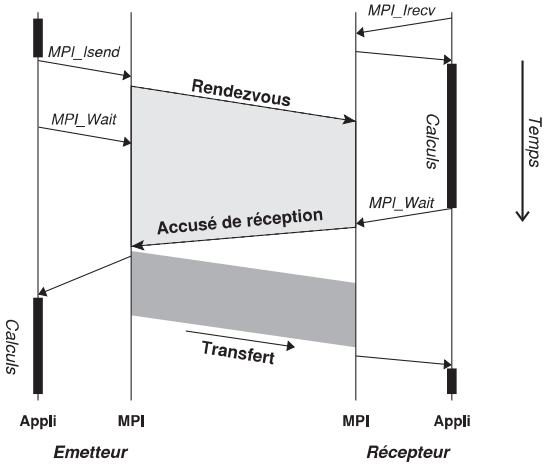

FIGURE 3.2 – *Illustration des problèmes de recouvrement pour les messages nécessitant un rendezvous. L'émetteur est ici bloqué en attendant que le récepteur termine son calcul et consulte la bibliothèque de communication.*

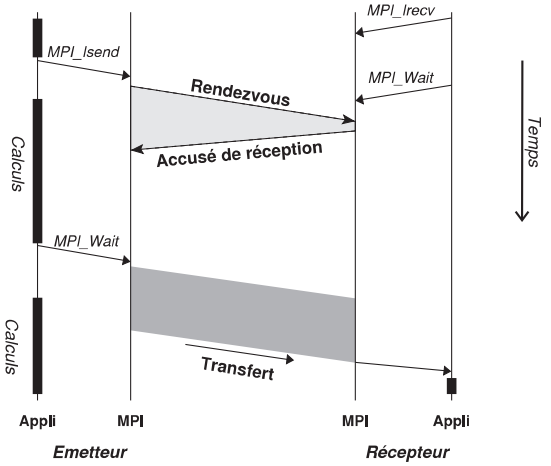

FIGURE 3.3 – *Illustration des problèmes de recouvrement pour les messages nécessitant un rendezvous. Le récepteur doit attendre que l'émetteur ait terminé son calcul pour recevoir les données.*

3.1.2 Progression des communications dans le protocole du *rendez-vous*

Le recouvrement des communications par le calcul, s'il est problématique pour les messages transmis par copie, l'est encore plus pour le mode *rendez-vous*. Les temps de transfert de ce mode étant plus élevés, un mauvais taux de recouvrement est ici beaucoup plus dommageable pour l'application. Le protocole de *rendez-vous* présenté dans la section 2.2.2 est utilisé afin d'éviter les copies mémoire inutiles, mais également pour limiter la taille des tampons mémoire réservés aux messages inattendus. Le protocole de *rendez-vous* peut poser certains problèmes lorsque l'application utilise des primitives de communication non-bloquantes en espérant recouvrir les communications et le calcul [BRU05].

Ainsi, le taux de recouvrement obtenu dépend fortement de la progression des communications en arrière-plan. La Figure 3.2 décrit la cause de ce manque de performances. Lorsque l'émetteur s'apprête à envoyer un message de grande taille, une demande de *rendez-vous* est transmise au récepteur. Lorsque cette demande est reçue, l'accusé de réception correspondant n'est envoyé que quand l'application consulte la bibliothèque de communication. Le *rendez-vous* ne progresse alors que lorsque le calcul se termine. Ce manque de progression implique un mauvais taux de recouvrement du côté récepteur, mais également une attente prolongée du côté de l'émetteur qui doit attendre l'accusé de réception pour envoyer les données et donc terminer la communication. Comme le montre la Figure 3.3, un problème similaire a lieu lorsque l'émetteur tente de recouvrir ses communications par du calcul. L'appel à la primitive de communication non-bloquante initie le *rendez-vous* et la main est rendue à l'application qui peut calculer. L'accusé de réception envoyé par le récepteur n'est ici détecté que lorsque l'application appelle la bibliothèque de communication pour attendre la fin de l'échange de message. Le récepteur se trouve donc bloqué à attendre les données tant que l'émetteur calcule.

Le problème du recouvrement des communications par du calcul pour des messages de tailles importantes est grandement dû au manque de **réactivité** aux événements du réseau des bibliothèques de communication. La notion de réactivité désigne la capacité à réagir rapidement à un événement. Ainsi, une bibliothèque très réactive sera capable de détecter et réagir très rapidement à la terminaison d'une com-

munication alors qu'un système peu réactif ne réagira que tardivement à un événement.

Le manque de réactivité des bibliothèques de communication entraîne donc des problèmes de recouvrement des communications par du calcul du fait de l'incapacité des implémentations à réagir aux messages de contrôle en arrière-plan. Le standard MPI ne définit aucune règle concernant la progression des communications. Aussi, les implémentations MPI adoptent-elles des comportements différents – certaines utilisent des threads pour faire progresser les communications, d'autres nécessitent que l'application teste la terminaison d'une communication à l'aide de `MPI_Test` pour que le *rendez-vous* progresse.

3.2 Gestion de la concurrence

Du fait du développement massif des grappes contenant des processeurs multi-cœurs, de plus en plus d'approches mélangent les threads aux paradigmes de communication de type passage de message. Les bibliothèques de communication utilisées pour de telles approches nécessitent donc un certain niveau de tolérance aux threads et aux problématiques en découlant : accès concurrents, risques d'interblocage, etc. Malgré les spécifications du standard MPI, la plupart des implémentations ont été conçues pour des utilisations résolument mono-threadées et leur tolérance aux threads est généralement très légère. Cette section présente les problèmes rencontrés au sein des bibliothèques de communication lorsque celles-ci sont utilisées dans des environnement multi-threadés.

3.2.1 Accès concurrents à la bibliothèque de communication

Le support des applications multi-threadées dans une bibliothèque de communication est une opération délicate tant les utilisateurs sont habitués aux hautes performances du réseau. L'ajout de mécanismes de protection doit donc être opéré avec précaution afin de réduire l'impact du verrouillage sur les performances brutes. La plupart des implémentations MPI étant à l'origine conçues pour des applications n'utilisant pas de threads, le développement de solutions permettant les accès concurrents nécessite un travail minutieux *a posteriori* afin d'éviter les interblocages dus au verrouillage.

L'implémentation d'une bibliothèque de communication supportant les applications multi-threadées requiert généralement une analyse poussée des structures de données utilisées afin d'isoler celles risquant d'être modifiées par plusieurs threads simultanément [GT07]. Sans mécanisme de protection, de telles structures de données peuvent contenir des données invalides suite à des modifications concurrentes. Une fois que les structures de données potentiellement problématiques ont été protégées par des verrous, il reste encore à s'assurer que les interfaces de communication bas-niveau soient appelées correctement. En effet, certains pilotes de carte réseau ne supportent pas les accès concurrents. Il faut donc sérialiser les appels à ces pilotes, généralement à l'aide de verrous.

Enfin, un grand nombre d'implémentations MPI sont disponibles pour plusieurs technologies réseau. Ainsi, OPEN MPI ou MPICH2 permettent d'utiliser indifféremment des réseaux MYRINET, TCP/IP ou INFINIBAND. Cette abstraction des interfaces réseau implique une complexification de la pile logicielle de ces implémentations : MPICH2 adopte une approche par couches et les pilotes réseaux peuvent être inclus dans la couche *Device*, *Channel* ou dans un module réseau (voir Figure 3.4). La structure de OPEN MPI est basée sur des composants et les pilotes réseaux sont généralement implémentés dans un composant MTL (*Message Transfer Layer*) ou BTL (*Byte Transfer Layer*) comme le montre la Figure 3.5.

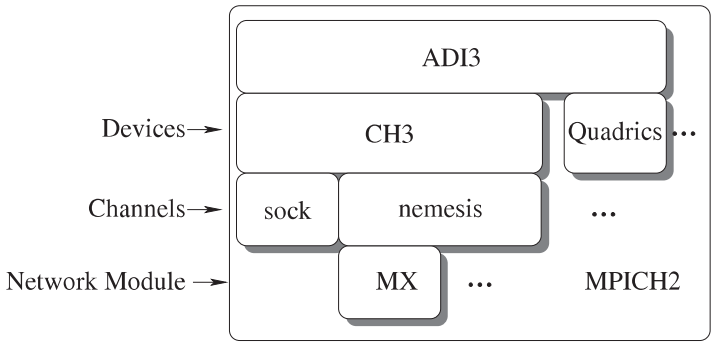

FIGURE 3.4 – *Pile logicielle de* MPICH2.

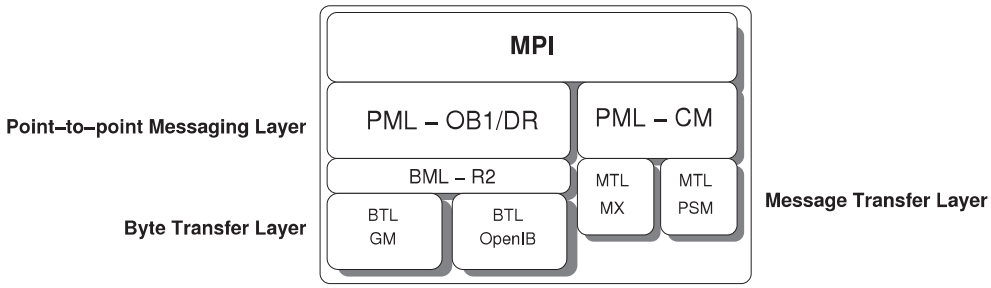

FIGURE 3.5 – *Pile logicielle de* OPEN MPI.

Le verrouillage des structures de données pouvant être modifiées de manière concurrente dépend donc du module réseau utilisé et le travail d'analyse doit donc être fait pour tous les pilotes réseau. Cette tâche fastidieuse explique pourquoi la plupart des implémentations MPI ne supporte les accès concurrents que pour un nombre restreint de modules réseaux.

3.2.2 Efficacité et performances du verrouillage

Les mécanismes permettant de protéger les structures de données contre les accès concurrents ont un impact sur les performances brutes du réseau. La Figure 3.6 illustre ce problème : en activant les mécanismes de protection, la latence observée du réseau augmente (ici, d'environ 300 ns). Cette détérioration des performances – due à l'utilisation de verrous pour protéger les structures de données – reste modérée et l'impact sur le temps d'exécution d'une application sera généralement négligeable. Toutefois, certaines applications sont très sensibles à la latence du réseau et une détérioration de 10 % comme ici peut entraîner une dégradation sensible des performances [WADJ+05].

Si l'impact des mécanismes de protection reste limité dans le cas mono-threadé, il peut devenir important lorsque l'application utilise plusieurs threads pour communiquer. La Figure 3.7 montre la latence moyenne mesurée quand plusieurs threads communiquent de manière concurrente. Outre le fait que les threads doivent s'attendre mutuellement pour accéder à la carte réseau – ce qui a un impact sur la latence observée – ils doivent également s'attendre mutuellement pour entrer dans les sections critiques protégées par des verrous. Il en résulte une contention sur les verrous qui pénalise fortement les performances de l'application. Cette contention n'entre pas seulement en jeu lorsque plusieurs threads tentent d'émettre ou de recevoir des données, mais également à chaque fois que des primitives de communication sont appelées de manière concurrente. Un exemple frappant de ce type de problème

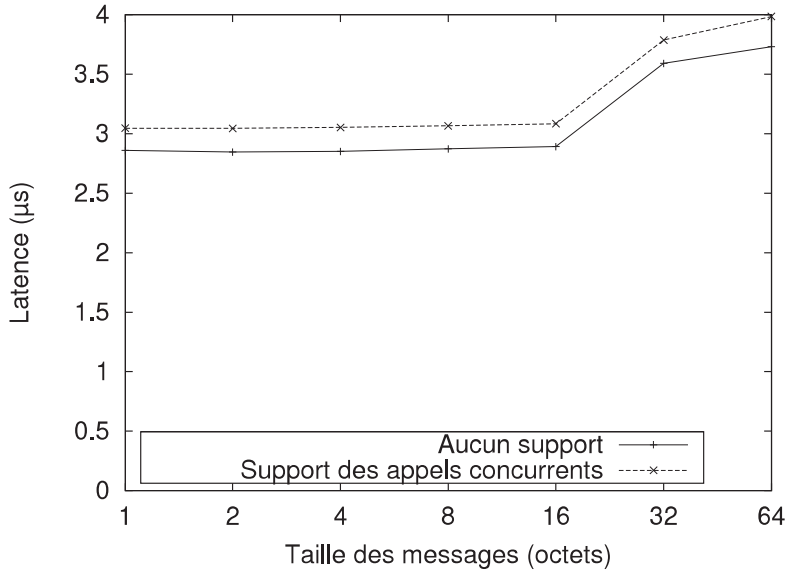

FIGURE 3.6 – *Surcoût introduit par le support des accès concurrents dans* OPEN MPI *sur* MX.

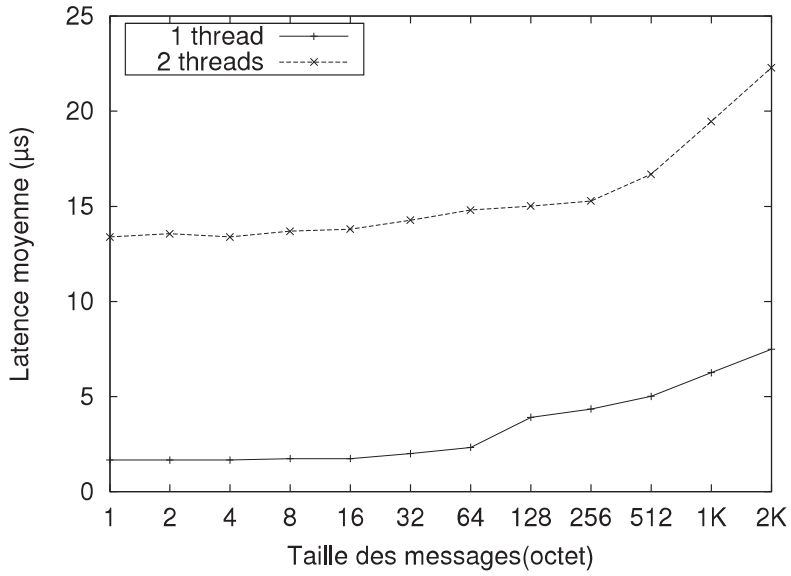

FIGURE 3.7 – *Performances de MVAPICH sur* INFINIBAND *lorsque plusieurs threads communiquent simultanément.*

concerne les performances des fonctions permettant d'attendre la terminaison d'une requête. Ces fonctions sont généralement implémentées de façon à scruter le réseau jusqu'à ce que la requête soit terminée. Ce comportement, bien qu'extrêmement efficace dans un environnement mono-threadé, peut poser de gros problèmes de performances dès lors que plusieurs threads attendent de manière concurrente. La contention pour accéder à l'interface de communication ainsi qu'aux structures de données entraîne une compétition entre les threads et détériore ainsi très fortement les performances.

3.3 Exploitation efficace des ressources

Le développement des processeurs multi-cœurs dans les grappes de calcul entraîne une augmentation du nombre d'unités de calcul par nœud depuis maintenant 5 ans. Alors qu'auparavant la plupart des grappes étaient équipées de machines embarquant 2 unités de calcul, il devient maintenant courant de disposer de nœuds comportant 8 cœurs. Pendant cette même période, la bande-passante délivrée par des réseaux a également augmenté. Toutefois, les performances en latence des réseaux n'ont que faiblement changé et l'augmentation du nombre de cœurs se partageant une carte réseau entraîne une contention. L'utilisation de plusieurs cartes réseau par machine permet de limiter cette contention, mais n'est pas toujours suffisante. Il devient donc de plus en plus important d'optimiser ces flux de communication afin de réduire l'impact du réseau sur les performances des applications. Des approches agrégeant les flux de communication [Bru08a] ou les distribuant sur plusieurs réseaux [ABMN07] sont des premiers pas vers une réduction du coût des communications mais l'utilisation massive de processeurs multi-cœurs et l'introduction d'approches mélangeant passage de messages et threads ouvrent la voie à de nouvelles optimisations.

3.3.1 Vers de nouvelles opportunités

L'augmentation du nombre de cœurs et l'utilisation de threads, outre leur impact sur la quantité de données à échanger, ont des conséquences sur la synchronisation. En effet, plus le nombre de threads est élevé, plus les primitives de synchronisation telles que les barrières prennent de temps. Les structures de données partagées entre les threads sont généralement protégées par des verrous et lors de l'accès à une structure protégée un thread peut donc potentiellement se bloquer. Ainsi, l'augmentation du nombre de cœurs par nœud peut entraîner une utilisation moins intensive des processeurs et des *"trous"* peuvent se former dans l'ordonnancement des tâches.

Les synchronisations tendent donc à réduire l'efficacité des calculateurs et du fait de l'augmentation du nombre de cœurs par nœud– et donc des synchronisations – leur impact sur les performances devient problématique. La réduction du nombre et de la durée des synchronisations ou l'utilisation des *trous* laissés dans l'ordonnancement augmenteraient le taux d'occupation utile (*i.e.* servant à faire progresser les calculs ou les communications) des processeurs.

3.3.2 Le problème du traitement séquentiel des communications

L'utilisation massive de processeurs multi-cœurs dans les grappes et le développement d'applications mélangeant communications et threads peut dans certains cas mener à des déséquilibres de la charge de travail entre les différentes unités de calcul d'un même nœud. Par exemple, quand certains threads

sont bloqués à cause d'une primitive de synchronisation (barrière, attente d'un message provenant du réseau, etc.), d'autres font appel à la bibliothèque de communication afin d'envoyer des données à un autre nœud. Ce type de situation peut poser problème si, comme c'est généralement le cas, la bibliothèque de communication traite les flux de données de façon séquentielle. Par exemple, la plupart des implémentations de MPI traitent la primitive d'envoi non-bloquante `MPI_Isend` séquentiellement : la requête MPI est tout d'abord enregistrée, le message est copié dans la mémoire de la carte réseau et la requête est transmise au réseau. Ainsi, le message est envoyé le plus tôt possible et le destinataire peut le recevoir rapidement. Toutefois, le traitement séquentiel des flux de communication peut accentuer le déséquilibre de la répartition de la charge au sein d'un nœud. Certaines opérations telles que les copies mémoire ou l'enregistrement de pages mémoire peuvent être très coûteuses en temps et monopoliser un cœur alors que d'autres unités de calcul sont inutilisées.

3.4 Bilan

L'évolution récente de l'architecture des grappes de calcul qui embarquent aujourd'hui des processeurs multi-cœurs a entraîné un changement dans les modèles de programmation employés. Le modèle MPI se voit maintenant mélangé aux threads afin d'exploiter plus efficacement les architectures hiérarchiques des grappes. Toutefois, ce changement de paradigme implique de nombreux problèmes dus aux interactions entre threads et communication. La gestion de la concurrence dans les bibliothèques de communication ou le traitement séquentiel des communications restent aujourd'hui problématiques. Le recouvrement des communications par le calcul, même s'il n'est pas spécifique aux environnements multi-threadés est d'autant plus important que le nombre d'unités de calcul augmente et que le coût des communications devient critique.

Chapitre 4

État de l'art

Sommaire

Comme nous l'avons vu dans le chapitre précédent, la tendance actuelle à l'utilisation massive de processeurs multi-cœurs et le mélange des communications avec les threads ont fait émerger de nouvelles problématiques. En effet, la conception de la plupart des bibliothèques de communication est orientée vers un usage résolument mono-programmé et l'utilisation de threads dans les applications peut avoir un impact important sur les performances du réseau. Nous présentons donc dans ce chapitre un survol des bibliothèques de communication actuelles et passées ainsi que les travaux ayant porté sur les problématiques que nous avons décrites dans le chapitre précédent.

4.1 Recouvrement des communications par le calcul

Le coût des communications est un des principaux problèmes qui réduisent les performances des applications scientifiques. En effet, même si les performances des réseaux ont été fortement améliorées au cours des dernières années, la transmission de données entre plusieurs machines reste très coûteuse. Le

recouvrement des communications par le calcul est une opportunité intéressante pour réduire le surcoût engendré par les transferts de données. L'utilisation de primitives de communication non-bloquantes fournies par la très grande majorité des bibliothèques de communication est donc un moyen simple pour améliorer les performances globales d'une application. Toutefois, comme nous l'avons vu dans la section 3.1.2, l'implémentation de mécanismes permettant le recouvrement des communications par du calcul n'est pas simple. Du fait des techniques employées dans les bibliothèques de communication pour améliorer les performances du réseau (protocole de *rendez-vous*, transferts de données entre la carte réseau et la mémoire de l'hôte en utilisant intensivement le processeur, etc.), le recouvrement efficace des communications par du calcul n'est pas toujours obtenu. En effet, la transmission de gros messages nécessite généralement un protocole de *rendez-vous* pour éviter les recopies inutiles. Le recouvrement n'est atteint de manière efficace que si la détection des messages de ce *rendez-vous* et la réponse à ces derniers sont effectuées rapidement. Nous présentons dans cette section les mécanismes utilisés à cette fin dans les bibliothèques de communication.

4.1.1 Utilisation de threads de progression

La progression du *rendez-vous* permettant de recouvrir le calcul et les communications nécessite de pouvoir détecter l'arrivée (et y réagir) d'un message – la demande de *rendez-vous* ou l'accusé de réception – pendant que l'application calcule. Une méthode simple pour détecter les messages de *rendez-vous* est d'utiliser un *thread de progression* en plus des threads de l'application. Ce *thread de progression* passe son temps à attendre une interruption – signalant un événement – provenant de la carte réseau. Lorsqu'un message arrive par le réseau, le thread est réveillé et peut répondre s'il s'agit d'un message de *rendez-vous*. Ainsi, l'application peut calculer normalement et les messages nécessitant un *rendez-vous* progressent en arrière-plan de manière transparente. Comme le thread de progression est en attente d'un événement la plupart du temps, son occupation du processeur est très faible, ce qui permet d'obtenir un taux de recouvrement élevé. De nombreuses bibliothèques de communication utilisent aujourd'hui des *threads de progression* pour assurer le recouvrement des communications par du calcul. Ainsi, OPEN MPI tire partie de ce mécanisme pour ses pilotes réseau TCP [WGC+04], QUADRICS [YWGP05] et INFINIBAND [SWG+06]. L'interface de communication bas-niveau MX [Myr03] ou MPICH-MADELEINE [AMN01] utilisent également des threads pour faire progresser les communications en arrière-plan.

L'utilisation d'un *thread de progression* est probablement le moyen le plus naturel pour obtenir un recouvrement des communications par du calcul. Toutefois, pour que les communications progressent rapidement, le thread supplémentaire doit être ordonnancé dès que la carte détecte l'arrivée de données. Si l'application exploite toutes les unités de calcul, l'ordonnanceur de thread peut ne donner la main au thread de communication qu'au bout d'un certain temps. Pour qu'il soit ordonnancé le plus rapidement possible, il est possible de lui donner une priorité élevée [HL08]. Cette technique n'est toutefois pas infaillible car la priorité du thread de progression n'est qu'une indication pour l'ordonnanceur : si un grand nombre de threads sont prêts à s'exécuter, le thread de communication peut avoir à attendre son tour. Outre ces problèmes de priorité, l'utilisation de threads de progression souffre d'un surcoût élevé. En effet, le thread attend une interruption de la carte réseau pour se réveiller. Le coût induit par le traitement de cette interruption se répercute sur les communications. Enfin, il faut également ajouter le coût des changements de contexte nécessaire à l'ordonnancement du thread de progression.

Afin de résoudre ces problèmes de priorité des threads de progression tout en exploitant les grappes de machines multi-processeurs, il est également possible de dédier un processeur au traitement des commu-

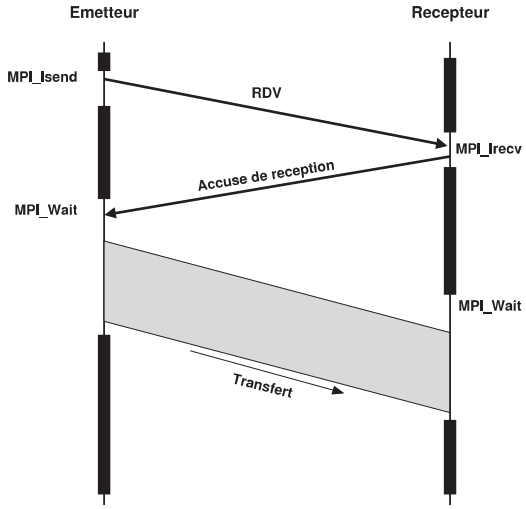

FIGURE 4.1 – *Protocole du rendezvous sur une interface de communication par RDMA. Les échanges de messages se font par RDMA-Write.*

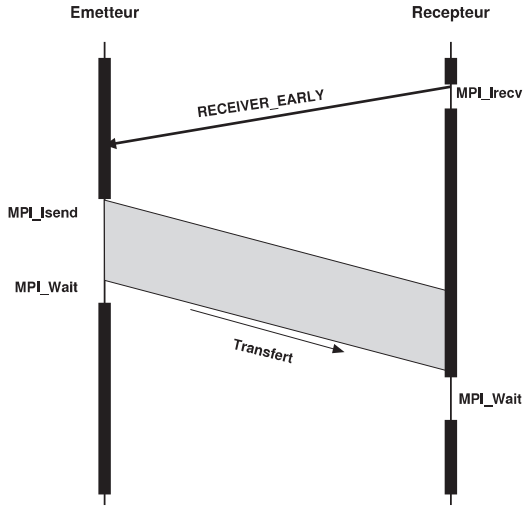

FIGURE 4.2 – *Le protocole du rendezvous peut être simplifié si le récepteur est prêt avant l'émetteur.*

nications [BGSP94]. Le thread chargé de détecter les communications peut alors utiliser le mode scrutation de l'interface réseau afin de réduire le coût de la détection au minimum. Le taux de recouvrement est ici très bon puisque les communications peuvent être totalement recouvertes (le seul surcoût est dû aux communications entre les processeurs). Le recouvrement des communications par du calcul n'est donc généralement atteint efficacement que lorsqu'au moins une unité de calcul est disponible [HSGL08]. Toutefois, ce mécanisme nécessite de dédier un processeur aux communications et donc de perdre une partie de la puissance de calcul qui pourrait être utilisée pour faire progresser les calculs. Ainsi, si le nœud possède 4 cœurs, dédier l'un d'entre eux aux communications résultera en une baisse de 25 % de la puissance de calcul exploitable par l'application. Bien que cette réduction de la puissance de calcul disponible soit aujourd'hui trop importante, on peut penser que ce type de mécanisme pourra être utilisé dans quelques années, sur des machines possédant des dizaines de cœurs : sur une machine à 64 cœurs, l'impact du sacrifice d'un cœur pour les communications représente moins de 2 % de la puissance de calcul totale. Les applications nécessitant une progression des communications en arrière-plan pourraient donc raisonnablement utiliser un tel mécanisme.

4.1.2 Utilisation des fonctionnalités avancées des cartes réseau

Du fait du surcoût des threads de progression dû au traitement des interruptions et aux changements de contexte, la communauté scientifique a cherché d'autres solutions au problème du recouvrement des communications par le calcul. De nombreux travaux ont porté sur un protocole de *rendez-vous* adapté au paradigme de communication RDMA, notamment pour les cartes INFINIBAND. Comme le montre la Figure 4.1, l'implémentation naïve du protocole de *rendez-vous* sur un réseau RDMA est la même que sur un réseau à base de passage de message : en utilisant des *écritures distantes* (RDMA-Write), l'émetteur envoie la demande de *rendez-vous*, le récepteur répond par un accusé de réception et l'émetteur peut alors écrire directement dans la mémoire du récepteur. Cette méthode est la même que le protocole de *rendez-vous* décrit dans la section 2.2.2 et souffre donc des mêmes problèmes : si l'émetteur

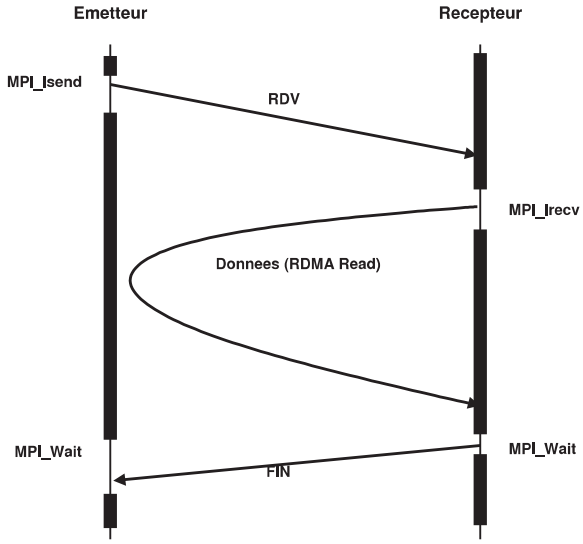

FIGURE 4.3 – *Optimisation du protocole du rendezvous pour les réseaux supportant le RDMA : le récepteur peut lire directement les données grâce à un RDMA-Read.*

ou le récepteur ne détectent pas les messages du *rendez-vous*, le transfert des données est retardé. Rashti *et al.* [RA08] ont montré comment optimiser ce protocole lorsque le récepteur est en avance par rapport à l'émetteur : le récepteur peut alors initier le *rendez-vous* et l'émetteur peut envoyer directement les données (voir Figure 4.2). Ce mécanisme permet d'éviter les problèmes de progression du *rendez-vous* du côté du récepteur lorsque celui-ci est en avance par rapport à l'émetteur. Toutefois, cette solution ne résout pas les problèmes de progression dans les autres cas.

Pour résoudre ces problèmes de recouvrement des communications par du calcul, des protocoles alternatifs ont été proposés. Tout comme le *rendez-vous*, ces protocoles visent à permettre des transferts de messages sans recopie des données. En s'appuyant sur des cartes réseau qui permettent des accès directs à la mémoire distante, un protocole à base de *lecture distante* (RDMA-Read) a été conçu [SJCP06]. Ce protocole illustré par la Figure 4.3 n'implique l'émetteur qu'à l'initialisation de l'échange : une demande de *rendez-vous* est tout d'abord envoyée. Lorsque le récepteur la reçoit, il effectue une lecture des données à distance (RDMA-Read) afin de récupérer les données. Quand le transfert de données se termine, le récepteur le signale à l'émetteur pour lui signifier la fin de la communication. Ainsi, l'émetteur n'est impliqué qu'au tout début du protocole et ne peut pas gêner le récepteur. Toutefois, le recouvrement de la communication par du calcul n'est réellement possible que du côté de l'émetteur : si le récepteur est occupé à calculer lorsqu'il reçoit la demande de *rendez-vous*, la lecture des données à distance ne sera effectuée que lors de l'appel à `MPI_Wait` (ou, pour une autre bibliothèque de communication, à la fonction correspondante). Ce problème de progression des communications du côté du récepteur peut être contourné si la carte réseau sait générer une interruption lors de la réception d'un message [SJCP06]. Le calcul qu'effectue le récepteur est alors interrompu et la lecture à distance peut être lancée. L'utilisation d'interruptions permet ici d'assurer le recouvrement des communications par du calcul à la fois du côté de l'émetteur et du côté du récepteur. Toutefois, les surcoûts engendrés par le traitement de l'interruption et par les changements de contexte sont prohibitifs et aucune implémentation MPI n'utilise actuellement cette technique.

Une caractéristique intéressante de certaines cartes réseau modernes est qu'elles sont programmables. En effet, la plupart des cartes réseau hautes performances embarquent des processeurs relativement

puissants – une carte Myri-10G possède un processeur RISC cadencé à 333 MHz – qui peuvent être reprogrammés. Ainsi, de nombreux travaux ont porté sur le déport du traitement d'un protocole sur la carte réseau [Ang01, WJPR04]. Par exemple, les cartes réseaux QSNETII sont équipées d'un processeur dédié à la progression des communications qui est donc entièrement gérée par le matériel. L'utilisation de cartes réseau programmables pour gérer le *rendez-vous* sans intervention du processeur de la machine a également été étudiée [MZOR02]. Par exemple, un mécanisme nommé TUPLEQ a été implémenté dans MVAPICH2, l'implémentation MPI pour réseaux INFINIBAND [KSP09]. TUPLEQ se base sur le concept de files de réception XRC (*eXtended Reliable Connection*) présent dans les cartes INFINIBAND de dernières générations. Une liste de réception – nommée TUPLEQ – est créée pour chaque *n-uplet* (*tag*, *rang*, *identifiant de communicateur*) et, à partir de ces files, le mécanisme d'appariement est géré par la carte réseau. Ainsi, lorsqu'un message doit être émis et que le récepteur n'est pas encore prêt (*i.e.* la requête de réception correspondante n'a pas encore été déposée), la carte réseau de l'émetteur attend que le récepteur soit prêt. La synchronisation entre l'émetteur et le récepteur nécessaire aux transferts sans recopie est donc entièrement gérée par les cartes réseaux et les communications sont donc recouvertes par du calcul. Toutefois, ce mécanisme souffre de quelques problèmes qui réduisent son champ d'application. Outre le matériel très spécifique nécessaire (seules les cartes INFINIBAND CONNECTX proposent les connexions XRC), l'utilisation de TUPLEQ par n'importe quelle application est gênée par la gestion complexe des réceptions dont la source ou le tag se sont pas spécifiés (`MPI_ANY_SOURCE` ou `MPI_ANY_TAG`). En effet, dans ce type de situation, le mécanisme de TUPLEQ doit être désactivé pour revenir à un mode de communication classique entraînant un surcoût non-négligeable.

4.1.3 Bilan des mécanismes de progression des communications

Bien que les communications non-bloquantes et le besoin de recouvrir les transferts réseau par du calcul soient apparus au début des années 1990, les travaux portant sur ces problématiques n'ont pas abouti à des solutions satisfaisantes. Les bibliothèques de communication optant pour les threads de progression permettent le recouvrement de manière portable. Toutefois, l'utilisation de threads implique un surcoût dû au traitement des interruptions et aux changements de contexte. De plus, les applications surchargeant la machine (en lançant plus de threads qu'il n'y a de processeurs) risquent de gêner les threads de progression et réduire leur efficacité. Toutefois, l'augmentation du nombre de cœurs par machine devrait permettre dans un avenir proche de dédier un certain nombre de cœurs au traitement des communications afin d'augmenter les taux de recouvrement perçus par les applications. Les solutions exploitant certaines capacités avancées des cartes réseau, notamment le déport de protocole, souffrent quant à elles d'une faible portabilité : les cartes réseau permettant ces mécanismes sont peu répandues. De plus, bien que ces techniques permettent de recouvrir les communications par du calcul, leur champ d'application reste limité comme le montrent les problèmes rencontrés par TUPLEQ.

4.2 Gestion des flux de communication concurrents

Comme nous l'avons vu dans la section 2.4, l'augmentation du nombre de cœurs par nœud de calcul dans les grappes rend l'exécution d'un processus MPI par cœur de plus en plus problématique. Les modèles hybrides mélangeant threads et communications se sont récemment beaucoup développés, comme le montrent les travaux de l'ASC SEQUOIA [Sea08]. Toutefois, l'utilisation de MPI par des applications multi-threadées change radicalement la donne. En effet, les bibliothèques de communication doivent

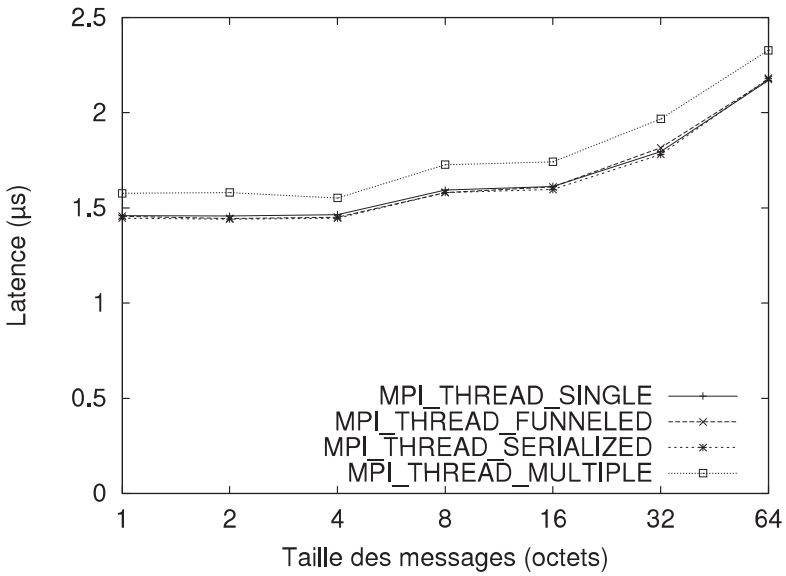

FIGURE 4.4 – *Impact du niveau de support des threads demandé à l'initialisation sur les performances de MVAPICH2 sur InfiniBand avec un seul thread. Seul `MPI_THREAD_MULTIPLE` implique un surcoût.*

maintenant supporter les accès concurrents. Dans cette section, nous présentons ce que dit le standard MPI à propos des accès concurrents et analysons les différentes implémentations qui en découlent.

4.2.1 Le point sur les implémentations MPI

Bien que la première version du standard MPI ne précisait rien concernant l'usage de threads par les applications, l'émergence de solutions hybrides a mené à la prise en compte du multi-threading dans la seconde version de MPI. Ainsi, la norme définit plusieurs niveaux de support :

- le mode `MPI_THREAD_SINGLE` signifie qu'un seul thread s'exécute dans le processus MPI.
- le mode `MPI_THREAD_FUNNELED` signifie que le processus peut être multi-threadé, mais seul le thread principal peut faire appel à des primitives MPI.
- le mode `MPI_THREAD_SERIALIZED` signifie que le processus peut être multi-threadé et que tous les threads peuvent faire appel aux primitives MPI, mais pas de manière concurrente. Ainsi, les appels à MPI doivent être "sérialisés" au niveau de l'application, par exemple en utilisant un mécanisme d'exclusion mutuelle.
- le mode `MPI_THREAD_MULTIPLE` signifie que le processus MPI peut utiliser des threads qui sont libres de faire appel à MPI, y compris de manière concurrente. L'application n'a donc pas a gérer d'exclusion mutuelle pour accéder aux primitives MPI.

Les trois premiers niveaux de support des applications multi-threadées ont peu d'implications sur les implémentations MPI : si l'application utilise des threads, elle garantit que les appels aux primitives MPI se feront un par un. En revanche, le niveau `MPI_THREAD_MULTIPLE` a un impact fort sur l'implémentation MPI. Ici, les threads de l'application peuvent accéder à la bibliothèque de communication de manière concurrente et des mécanismes de protection contre les accès concurrents doivent donc être ajoutés. Comme le montre la Figure 4.4, ces mécanismes de protection ont un impact sur les performances des communications. Les premiers niveaux de support des threads forcent les applications à gérer la

concurrence à la place de l'implémentation MPI. Bien que pour certaines applications, cette gestion ne soit pas une contrainte, la plupart des programmes multi-threadés sont fortement simplifiés s'ils peuvent accéder à la bibliothèque de communication de manière concurrente. Par la suite, nous ne considérons donc que le niveau `MPI_THREAD_MULTIPLE`.

Les implémentations MPI supportant ce niveau au moins partiellement sont aujourd'hui nombreuses. Certaines implémentations telles que MPICH-MADELEINE [AMN01] supportent les appels concurrents pour l'ensemble des pilotes réseau disponibles, mais la plupart des implémentations génériques ne proposent le niveau `MPI_THREAD_MULTIPLE` de façon performante que pour leur pilote réseau TCP : MPICH2 [mpi07], OPEN MPI [GWS05] et MVAPICH2 [HSJ+06]. Le support des threads pour les autres technologies réseau est alors généralement plus basique. Par exemple, bien que OPEN MPI supporte officiellement les applications multi-threadées utilisant ses pilotes MYRINET ou INFINIBAND, les piles logicielles concernées sont très peu testées et les erreurs ne sont donc pas rares lorsque les applications utilisent des threads. MVAPICH2 se comporte de façon similaire avec son pilote INFINIBAND : bien qu'affichant de bonnes performances pour les applications multi-threadées, il n'est pas rare d'obtenir des erreurs causées par les accès concurrents.

4.2.2 Mécanismes de protection internes aux bibliothèques de communication

Le support des applications multi-threadées par les bibliothèques de communication nécessite, la plupart du temps, l'ajout de mécanismes d'exclusion mutuelle. Ces mécanismes permettent de s'assurer que les structures de données internes ne sont pas modifiées par plusieurs threads simultanément, ce qui risquerait de les mettre dans un état incohérent. L'ajout de ces mécanismes de protection peut être relativement simple en utilisant un verrou global : lorsqu'un thread accède à la bibliothèque de communication, le verrou est pris et n'est relâché que lorsque l'application ressort de la bibliothèque de communication. Si cette méthode permet effectivement de supporter les accès concurrents, les performances obtenues sont généralement médiocres. En effet, les appels à la bibliothèque de communication sont sérialisés et les threads souhaitant communiquer doivent donc attendre que les autres threads aient relâché le verrou. Avec l'augmentation du nombre de threads se partageant l'accès au réseau, cette attente devient de plus en plus longue et les performances offertes par ce mécanisme sont donc mauvaises.

Pour gérer plus efficacement les accès concurrents, un travail complexe d'analyse doit être effectué afin d'identifier les portions de code pouvant poser problème [SPH96, GT07]. Il est alors possible d'utiliser un mécanisme de *verrous à grain fin* qui consiste en plusieurs verrous pour les différentes parties de la bibliothèque de communication. Ainsi, l'application peut effectuer plusieurs appels à la bibliothèque de communication simultanément : les threads ne seront bloqués que s'ils tentent d'accéder à une même structure de données. Bien que cette méthode permette plusieurs appels concurrents sans compromettre les structures de données, les implémentations actuelles ne sont pas toutes exemptes de problème. Dans certains cas, ces problèmes peuvent être dus à l'utilisation d'interfaces réseau de bas niveau qui ne sont pas protégées correctement. En effet, toutes les interfaces de communication de bas-niveau ne supportent pas les accès concurrents et il faut donc les protéger au même titre que les structures de données internes à la bibliothèque de communication.

Le principe des *verrous à grain fin* consiste à réduire la taille des *sections critiques* afin de limiter la durée pendant laquelle un verrou est pris. Balaji *et al.* [BBG+08] ont ainsi implémenté plusieurs mécanismes de verrouillage dans MPICH2, avec chacun une granularité plus ou moins grande. Le verrou global peut ainsi être remplacé par des verrous protégeant des objets – les verrous sont ici pris lorsque l'on doit manipuler ces objets – ou par des algorithmes ne nécessitant pas de verrous (*lock-free*

algorithms). Ces méthodes permettent de réduire fortement la taille de la section critique et offrent de bonnes performances lorsque plusieurs threads accèdent à la bibliothèque de communication de manière concurrente.

Le prix à payer pour cette amélioration des performances est une analyse complexe de la bibliothèque de communication pour isoler les structures partagées ou la conception d'algorithmes au sein de la bibliothèque de communication permettant d'accéder aux structures de données sans verrouillage. De plus, ce travail doit être effectué pour chacun des pilotes réseau de la bibliothèque de communication. Par conséquent, la plupart des implémentations de MPI sachant exploiter plusieurs technologies réseau ne supportent les accès concurrents que pour les réseaux fortement répandus (généralement TCP). Le problème du support des applications multi-threadées par les bibliothèques de communication n'est pas seulement dû à un manque de finition dans les implémentations mais est beaucoup plus profond. Les bibliothèques de communication n'ont pas été conçues pour supporter le multi-threading et l'application de *rustines* n'est pas suffisante. Le support des accès concurrents nécessite de profonds remaniements des algorithmes utilisés au sein des bibliothèques de communication préexistantes. Aussi, il faut penser au multi-threading dès la conception de la bibliothèque de communication faute de quoi les modifications à apporter pour gérer ce problème seront massives et risquent d'avoir un impact important sur les performances obtenues dans le cas mono-threadé.

4.3 Solutions intégrant les communications et le calcul

Avec l'apparition des premières grappes de machines multi-processeurs il y a 15 ans, l'idée d'utiliser des threads pour exploiter les unités de calcul est apparue. Aussi, plutôt que d'ajouter quelques rustines aux bibliothèques de communication préexistantes pour les adapter à ce nouveau paradigme, de nouvelles plates-formes destinées à ces architectures ont été conçues. Le pari est ici de remplacer le paradigme de passage de message par un modèle plus adapté au mélange de multi-threading et de communication. Ce nouveau paradigme, basé sur les *appels de procédure à distance* (*Remote Procedure Call*, RPC), permet d'invoquer des phases de calcul à distance en passant par le réseau. De nombreux travaux ont porté sur ce modèle et ont débouché sur des bibliothèques complètes intégrant des mécanismes de communication, mais également des bibliothèques de threads. Nous présentons ici les différentes approches ayant été étudiées.

4.3.1 La notion de pointeur global

Plutôt que de combiner les communications de type passage de message et les threads, Foster *et al.* ont proposé une approche basée sur des pointeurs globaux (*Global Pointers*, GP) et des requêtes de services distants (*Remote Service Requests*, RSR). L'implémentation de ces mécanismes, nommée NEXUS [FKT94], permet de gérer les messages asynchrones, la création ou destruction de threads à distance et un modèle de mémoire globale [FKT96]. Le but est ici d'exploiter efficacement des grappes de calcul avec des applications parallèles irrégulières. Les phases de calcul de ce type d'application peuvent générer de nombreux threads et les communications se font entre threads. Dans un paradigme de passage de message, les messages sont dirigés vers un processus et il est généralement nécessaire d'utiliser un tag particulier pour chaque thread avec lequel on souhaite communiquer. Le mécanisme de pointeurs globaux de NEXUS permet de remédier à ce problème en fournissant des connexions virtuelles entre threads. Un autre inconvénient des communications par passage de message pour les applications fortement irrégulières

réside dans le fait que les communications impliquent une synchronisation du côté du récepteur. Ce comportement peut être problématique pour ce type d'application, notamment lorsque le but d'une communication est de créer un nouveau thread pour la traiter. Le mécanisme de requêtes de services distants, couplé aux pointeurs globaux, permet de gérer ce cas de figure.

En proposant un paradigme alternatif au passage de message classique, NEXUS a permis d'intégrer communications et calcul. Cette bibliothèque de communication a servi de brique de base de l'environnement GLOBUS [FK97].

4.3.2 Sélection automatique du mode d'interrogation du réseau dans PANDA

L'environnement de programmation PANDA [BRH+93], développé à l'Université Libre d'Amsterdam, fournit des abstractions permettant de gérer à la fois les interfaces de communication et les threads. De manière similaire à NEXUS, PANDA propose également un mécanisme d'*appel de procédures à distance*. Du fait du contrôle de l'ordonnancement et des communications, PANDA peut utiliser la méthode de détection d'un événement réseau (*i.e.* une scrutation ou un appel bloquant basé sur des interruptions) la plus adaptée à la situation [LRBB96]. En effet, bien que la scrutation offre de très bonnes performances quand un cœur est inutilisé, elle devient inefficace lorsque toutes les unités de calcul sont exploitées. L'idée de PANDA est donc la suivante : lorsqu'un processeur est inoccupé, les interruptions sont désactivées et les requêtes réseau en cours sont détectées par scrutation. Si tous les cœurs sont utilisés, les interruptions sont activées et l'arrivée d'un message par le réseau va donc interrompre un thread de calcul qui pourra traiter la communication. Pour implémenter le mécanisme de RPC, PANDA utilise un système d'*upcall* lors de la réception d'un message : une fonction de traitement de message est exécutée [BBH+97]. Ce mécanisme est similaire à celui des *Active Messages* [ECGS92]. En faisant collaborer l'ordonnanceur de threads et la bibliothèque de communication, PANDA assure donc une bonne réactivité aux communications tout en gérant les accès concurrents.

4.3.3 Intégration des communications dans l'ordonnancement

Namyst *et al.* [NM95a] ont conçu en 1995 le système PM^2 (*Parallel Multi-threaded Machine*), un environnement destiné aux applications parallèles pour les architectures distribuées. Le but de PM^2 est de fournir une plate-forme à la fois performante et portable. En effet, PM^2 est capable d'exploiter une très large gamme de configurations, que ce soit en terme de systèmes d'exploitation (SOLARIS, SUNOS, LINUX, etc.), d'architectures processeur (SPARC, X86, ALPHA, etc.) ou de technologies réseau (VIA, BIP, SBP, SCI et TCP). Ces configurations sont exploitées efficacement grâce à la combinaison d'une bibliothèque de threads nommée MARCEL [NM95b] et d'une bibliothèque de communication nommée MADELEINE [BNM98]. Outre les fonctionnalités classiques d'une bibliothèque de threads, PM^2 fournit des mécanismes avancés permettant de créer des threads à distance ou de migrer les threads d'une machine à l'autre. Pour cela, PM^2 s'appuie à la fois sur MARCEL et sur les mécanismes d'*appel de procédure à distance* (*Remote Procedure Call*, RPC) de MADELEINE.

À la manière de NEXUS, PM^2 propose donc un modèle de programmation à base d'appel de procédure à distance. En plus de la bibliothèque de threads, PM^2 fournit une bibliothèque de communication puissante supportant les accès concurrents. MADELEINE est également capable de grouper plusieurs requêtes réseau afin d'éviter que plusieurs threads scrutent une même interface réseau de manière concurrente.

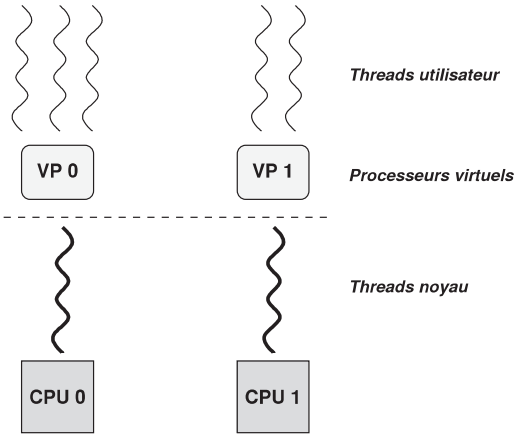

FIGURE 4.5 – *Dans une bibliothèque de threads de niveau utilisateur comme* MARCEL*, chaque processeur virtuel repose sur un thread noyau. Plusieurs threads utilisateur peuvent s'exécuter sur un même processeur virtuel.*

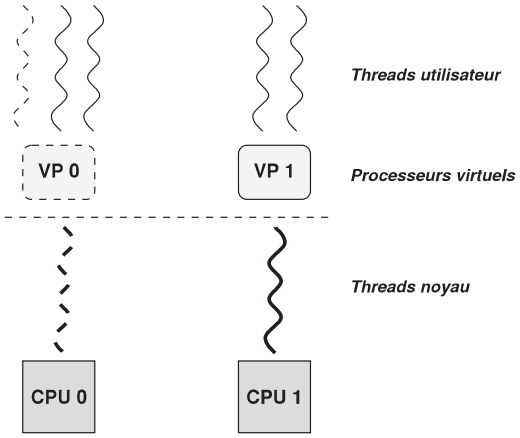

FIGURE 4.6 – *Lorsqu'un thread de niveau utilisateur effectue un appel bloquant, le thread noyau se bloque, empêchant les threads de niveau utilisateur partageant le processeur virtuel de s'exécuter.*

MARCEL_EV Les limites de PM^2 ont poussé à la conception d'un mécanisme de détection des événements d'entrées/sorties au sein de la bibliothèque de thread MARCEL [NM95b]. Ce mécanisme, décrit comme un *serveur de détection d'événements*, permet aux bibliothèques de communication ou aux systèmes de gestion des entrées/sorties de sous-traiter la détection de certains événements – tels que la terminaison d'une requête réseau – à l'ordonnanceur de threads [BDN02]. Le choix de l'intégration des mécanismes de détection des événements dans l'ordonnanceur de threads est dicté par la forte connaissance du système qu'a la bibliothèque de threads. De plus, l'ordonnanceur sait quels threads s'exécutent sur quels processeurs et peut donc adapter la méthode de détection (*i.e.* scrutation ou appel bloquant) en conséquence.

Contrairement au fonctionnement de PANDA qui lie très fortement la bibliothèque de communication et l'ordonnanceur de threads, le serveur de détection des événements est hautement générique et peut donc être utilisé par n'importe quelle bibliothèque de communication. Pour cela, le serveur se base sur des *fonctions de rappel* (*Callbacks*) fournies à l'initialisation et qui décrivent la façon de détecter un événement. La bibliothèque de communication peut fournir plusieurs *fonctions de rappel* pour les différentes méthodes de détection : scrutation d'un événement ou de tous les événements, appel bloquant pour détecter un événement ou n'importe quel événement, etc. Ensuite, lorsque l'application attend la fin d'une communication, la détection de l'événement associé – la complétion d'une certaine requête réseau – est laissée au serveur de détection. Les fonctions de rappel sont alors exécutées directement par l'ordonnanceur de threads lorsqu'un processeur est inutilisé ou lors d'un signal d'horloge. Ce mécanisme assure donc une fréquence de scrutation garantie : le temps de réaction à un événement réseau est borné par la durée du *quantum de temps* alloué aux threads. La durée de ce *quantum* dépend du système, mais est généralement de l'ordre de la dizaine de millisecondes.

Ce temps de réaction borné est un premier pas vers intégration des communications et des threads, mais reste trop élevé pour de nombreuses applications. Le serveur de détection d'événements fournit donc un mécanisme de gestion des appels bloquants permettant une plus grande réactivité. L'implémentation d'un tel mécanisme est toutefois problématique dans un ordonnanceur de niveau utilisateur comme MARCEL. Comme le montre la Figure 4.5, les *threads de niveau utilisateur* s'exécutent sur des *pro-*

cesseurs virtuels (*Virtual Processor*, VP). À chaque processeur virtuel est associé un *thread noyau*. Cela permet de gérer entièrement l'ordonnancement des threads depuis l'espace utilisateur, sans interaction avec l'espace noyau. Bien que ce type d'ordonnanceur soit très performant – le coût d'un changement de contexte est par exemple fortement réduit par rapport à un changement de contexte entre threads noyau – il devient problématique lorsque l'application effectue des appels bloquants. En effet, comme le système d'exploitation n'a pas connaissance des threads de niveau utilisateur, le thread noyau – et donc le processeur virtuel tout entier – est bloqué, empêchant par la même occasion les autres threads de niveau utilisateur partageant le processeur virtuel de s'exécuter (voir Figure 4.6).

Pour résoudre le problème des appels bloquants dans les ordonnanceurs de threads de niveau utilisateur, Anderson *et al.* ont proposé le modèle des SCHEDULER ACTIVATIONS [ABLL91]. Ce modèle fait collaborer l'ordonnanceur des threads de niveau utilisateur avec celui gérant les threads noyau. Ainsi, quand un thread s'apprête à se bloquer, le noyau peut prévenir l'espace utilisateur et ainsi ne bloquer que le thread de niveau utilisateur incriminé. L'implémentation de cette technique dans le noyau LINUX, nommée LINUX ACTIVATIONS [DNR00a], et son utilisation dans MARCEL ont permis une grande réactivité du serveur d'événements. Toutefois, les SCHEDULER ACTIVATIONS nécessitent l'implémentation de mécanismes complexes dans le noyau, ce qui pose des problèmes de performances, mais également de maintenance. Bien que les SCHEDULER ACTIVATIONS aient été, pendant un temps, intégrées à certains noyaux (notamment NETBSD [Wil02] ou FREEBSD [EE00]), ces problèmes ont mené à l'abandon de ces implémentations.

4.3.4 Bilan des solutions intégrant communications et multi-threading

L'intégration des communications et du calcul dans un même modèle de programmation est une première étape pour des communications efficaces en milieu multi-threadé. Les modèles de programmation basés sur les *appels de procédure à distance* fournissent un paradigme adapté au multi-threading dans les grappes de calcul. Toutefois, la nécessité de changer de modèle de programmation est contraignante pour le développeur qui doit faire migrer son application d'un modèle basé sur le passage de message à un modèle de RPC. Si ce type de modèle de programmation peut sembler plus naturel au niveau de l'application, il pose cependant les mêmes problématiques que les modèles à base de passage de message au niveau de la bibliothèque de communication. Les problèmes de réactivité aux communications ou de progression des communication restent présents.

Le travail conjoint de la bibliothèque de communication et de l'ordonnanceur de threads fournit des pistes de travail intéressantes concernant les problèmes de réactivité. En adaptant la méthode de détection des événements en fonction de l'état du système, le temps de réaction est minimisé tout en offrant de bonnes performances lorsque l'application n'utilise pas de thread. Toutefois, ces mécanismes sont aujourd'hui relativement anciens et ne sont pas forcément adaptés aux architectures modernes. L'augmentation du nombre de cœurs et du nombre de cartes réseau par nœud pose ici des problèmes de passage à l'échelle : les modèles centralisés présentés dans cette section souffrent de problèmes de performances sur de telles architectures.

4.4 Bilan et analyse de l'existant

Le développement des grappes de calcul et l'utilisation de processeurs multi-cœurs ont radicalement changé le paysage du calcul hautes performances au cours des dernières années. Malgré les nombreux

travaux concernant les interactions entre communications et threads, aucune solution réellement satisfaisante n'a émergé. Nous analysons ici les solutions proposées et tentons d'expliquer les raisons de l'absence actuelle de solution afin d'en retirer des enseignements.

4.4.1 Les raisons de l'absence actuelle de solution efficace

Le développement de bibliothèques de communication extrêmement efficaces pour les réseaux rapides modernes a permis une adoption très large des grappes de calcul. La généricité de ces implémentations et leurs performances toujours plus proches des performances des capacités des réseaux sous-jacents ont rendu les développeurs d'application plus exigeants. L'introduction d'applications multi-threadées a posé un énorme problème pour la conception et le développement de ces bibliothèques de communication : comment exploiter efficacement les réseaux rapides en gérant plusieurs flux de communication simultanément ?

L'adoption massive du standard MPI et le culte des performances brutes du réseau (latence, débit, nombre de messages échangés par seconde, etc.) sont des contraintes fortes pour le développement de nouvelles solutions. Il est, en effet, difficile de concevoir des mécanismes permettant de gérer efficacement les communications et les threads sans s'écarter du standard MPI ni détériorer les performances brutes du réseau. Il a donc fallu prendre du recul, concevoir des solutions *ad hoc* (le modèle RPC par exemple), expérimenter et tirer des conclusions afin de distinguer les mécanismes efficaces de ceux pouvant poser problème.

De plus, en analysant les bibliothèques de communication modernes, un constat s'impose : la plupart sont relativement anciennes et leur développement a débuté lorsque les problèmes liés au multi-threading n'étaient que très peu développés. Lors de leur conception, le but était d'offrir de bonnes performances à des applications mono-threadées pour une large gamme de technologies réseau. L'utilisation de telles bibliothèques par des applications multi-threadées a donc nécessité des modifications telles que l'ajout d'un mécanisme de verrouillage, l'ajout de threads de progression, etc. Les modifications apportées pour supporter les applications multi-threadées doivent être peu invasives et ne pas détériorer les performances de la bibliothèque de communication dans le cas mono-threadé. Ainsi, les nouvelles fonctionnalités ne sont implémentées que sous la forme de rustines et les possibilités sont alors limitées.

Lors de la conception du standard MPI-2 sont apparus les différents niveaux de gestion des accès concurrents présentés dans la section 4.2.1. Du fait du manque d'application exploitant les niveaux les plus élevés de support des accès concurrents, les premières implémentations de MPI-2 ne proposaient pas le niveau `MPI_THREAD_MULTIPLE`. Nous entrons ici dans un cercle vicieux : les développeurs de bibliothèques de communication rechignent à implémenter un support des threads pour quelques applications seulement. Outre la complexité de la tâche, cela risquerait d'engendrer des surcoût y compris lorsque l'application n'utilise pas de thread. Par conséquent, les développeurs d'application évitent de concevoir des applications multi-threadées puisque les bibliothèques de communication ne sont pas adaptées.

Au final, le support des applications multi-threadées dans les bibliothèques de communication reste faible, obligeant les développeurs d'application à gérer eux-même les problèmes dus aux thread en ajoutant des mécanismes d'exclusion mutuelle autour des primitives de communication ou en insérant des appels à la bibliothèque de communication dans les phases de calcul pour faire progresser les communications.

Toutefois, cette situation commence à changer avec le développement des processeurs multi-cœurs dans

les grappes de PC. La nécessité du multi-threading dans les applications pousse les développeurs à concevoir des mécanismes efficaces pour supporter les applications multi-threadées, comme le montrent les travaux réalisés autour du projet SEQUOIA [Sea08].

4.4.2 Enseignements retenus

L'étude des bibliothèques de communication existantes et l'analyse des raisons expliquant leur manque de performance dans un environnement multi-threadé nous ont permis d'en retirer des enseignements. Nous voyons ici les points clés qu'il faut garder en tête lors de la conception d'une bibliothèque de communication moderne.

Nous l'avons vu, la majorité des bibliothèques de communication existantes ont été conçu à une époque où les applications mélangeant communications et threads étaient très peu nombreuses. L'ajout de mécanismes permettant de gérer les problèmes liés au multi-threading dans des bibliothèques de communication optimisées pour les environnements mono-threadés a donné des résultats médiocres. Les problématiques du multi-threading – que ce soit le support d'applications multi-threadées ou la progression des communications – doivent être pris en compte dès la conception de la bibliothèque de communication faute de quoi les modifications à apporter pour ajouter des mécanismes de protection des données ou de progression des communications sont trop importantes et entraînent une complexification du code et une dégradation des performances.

Malgré les nombreux travaux ayant porté sur les modèles de programmation hybrides mélangeant passage de message et multi-threading, la plupart des bibliothèques de communication reste peu adaptée à ce type de paradigme. Les accès concurrents sont rarement maîtrisés et la progression des communications en arrière-plan est souvent inexistante. Par conséquent, les applications multi-threadées développent des techniques pour pallier ces manquements : thread de progression dans l'application, ajout de primitives de progression (`MPI_Test` par exemple) au milieu des phases de calcul, mécanismes d'exclusion mutuelle protégeant les primitives de communication ou threads dédiés aux communications, etc.

Bien que ces techniques permettent de contourner les limitations des bibliothèques de communication, elles posent des problèmes de portabilité : le développement de tels mécanismes dans l'application nécessite de réinventer la roue dans chaque application alors que leur implémentation dans une bibliothèque de communication permettrait de disposer de mécanismes efficaces une fois pour toute. De plus, le développement de ces techniques dans la bibliothèque de communication permet de profiter de la grande connaissance du matériel exploité, notamment des caractéristiques de la technologie réseau sous-jacente. Ainsi, l'application accéderait à la bibliothèque *"naturellement"* : sans se soucier des accès concurrents et uniquement pour échanger des données avec les autres machines. La bibliothèque de communication n'est en fait pas seulement une abstraction des interfaces réseau sous-jacentes, elle fournit des fonctionnalités nécessaires au bon fonctionnement des communications : progression des communications, mécanisme d'appariement, communications collectives, etc. L'implémentation dans l'application de techniques palliant les manquements des bibliothèques de communication pose également des problèmes plus fonctionnels : l'ajout de threads de communication complexifie le code, l'insertion de primitives de communication dans les phases de calcul peut entraîner des défauts de cache, etc.

Une pile logicielle dans laquelle les différents éléments se contentent de fournir les fonctionnalités pour lesquelles ils ont été conçus – l'application traite un problème, la bibliothèque de communication gère les échanges de données par le réseau, etc. – permet donc de simplifier le développement. Cela évite également que les développeurs d'applications n'implémentent des mécanismes préexistant de manière

non-optimale. La bibliothèque de communication a donc à sa charge tous les aspects des communications (appariement, gestion des flux de communication concurrents, progression des *rendez-vous*, etc.) et l'application se borne à calculer et à transmettre des messages en passant par la bibliothèque de communication.

D'une manière générale, les problèmes rencontrés par les bibliothèques de communication du fait des applications multi-threadées sont étroitement liés à l'ordonnancement des threads. Par exemple, l'ordonnanceur joue un rôle important dans la réactivité aux événements réseau. La collaboration entre la bibliothèque de communication et l'ordonnanceur de threads permet donc des communications plus efficaces. Le fait que l'ordonnanceur connaisse parfaitement la charge des processeurs de la machine et que la bibliothèque de communication connaisse l'état des cartes réseau permet à cette collaboration de dégager une vision globale du système. Grâce à cette vue de l'état général de la machine, des stratégies intelligentes – prenant en compte à la fois l'utilisation des processeurs et l'état des réseaux – peuvent être développées.

Chapitre 5

Pour une prise en compte des communications dans l'ordonnancement

Sommaire

Les chapitres précédents nous ont permis de relever les principales problématiques posées par les applications multi-threadées qui utilisent des bibliothèques de communication. Comme nous l'avons vu, les implémentations actuelles ne parviennent pas à résoudre ces problèmes de manière satisfaisante. Nous présentons donc ici les mécanismes clés que nous proposons d'utiliser pour exploiter efficacement les grappes de calcul modernes. Nous commençons par une présentation de l'architecture générale avant de détailler les différents mécanismes mis en œuvre.

5.1 Pour une bibliothèque de communication adaptée au multi-threading

Le chapitre précédent nous a montré les approches utilisées par les bibliothèques de communication existantes pour gérer les interactions entre multi-threading et communication. Nous avons tiré des enseignements de cette analyse et proposons une architecture qui soit une meilleure solution à ces problèmes. Nous commençons par énoncer les contraintes à respecter lors de la conception d'une telle architecture. Nous montrons ensuite les axes directeurs avant de présenter l'architecture générale du projet.

5.1.1 Démarche

Comme nous l'avons vu au cours des chapitres précédents, le développement des approches mélangeant multi-threading et communications pose de nombreuses problématiques aux concepteurs de bibliothèques de communication. La bibliothèque de communication que nous concevons doit donc supporter les applications multi-threadées et exploitant efficacement les multiples cœurs disponibles. Cette bibliothèque de communication doit satisfaire les points suivants :

1. **Gestion des accès concurrents :** l'augmentation du nombre de cœurs par nœud rend les approches hybrides mélangeant communications et multi-threading de plus en plus intéressantes. La bibliothèque de communication doit donc supporter les accès concurrents de manière efficace afin de pouvoir gérer les applications multi-threadées. Outre la protection des structures de données, la bibliothèque doit arbitrer efficacement l'accès aux interfaces de communication notamment lors des attentes concurrentes.
2. **Progression des communications :** comme nous l'avons vu dans la section 3.1.2, l'utilisation par l'application de primitives de communication non-bloquantes permet de cacher le coût des communications. Toutefois, ce recouvrement des communications par du calcul nécessite souvent que les *rendez-vous* progressent en arrière-plan. La bibliothèque de communication doit donc fournir des mécanismes à la fois génériques et performants permettant de faire progresser les communications.
3. **Exploitation des processeurs multi-cœurs :** avec l'augmentation du nombre de cœurs par nœud, les synchronisations prennent de plus en plus de temps et laissent donc des *"trous"* dans l'occupation des processeurs. Afin de réduire le coût des transferts réseau, une bibliothèque de communication moderne doit être capable d'exploiter les multiples cœurs d'une machine en parallélisant le traitements des communications.

Outre ces contraintes propres aux grappes de machines multi-cœurs, la bibliothèque de communication doit également fournir les fonctionnalités d'une implémentation "classique". Ainsi, les diverses technologies réseau modernes doivent être exploitées efficacement et la bibliothèque de communication doit fournir une abstraction des différentes interfaces de communication bas-niveau sur lesquelles elle repose.

5.1.2 Axes directeurs

En nous basant sur l'expérience retirée des projets présentés au cours du chapitre précedent, nous allons à contre-courant de certaines idées reçues sur lesquelles s'appuient les implémentations classiques.

Les environnements multi-threadés ne sont pas incompatibles avec des communications performantes. En effet, les problèmes de performance des applications multi-threadées sont principalement dues à une mauvaise gestion des accés concurrents par la bibliothèque de communication. Pourtant, des projets comme PM^2 ont montré que multi-threading et communications performantes ne sont pas incompatibles. Les mécanismes de protection des données sont indispensables et peuvent dégrader les performances, mais ce surcoût peut être compensé par une meilleure gestion des différents flux de communication.

Pour des communications efficaces en milieu multi-threadé, l'ordonnanceur de thread et la bibliothèque de communication doivent collaborer étroitement. Les approches faisant interagir l'ordonnanceur de threads et la bibliothèque de communication telles que PANDA ou le serveur de détection d'événements de PM^2 montrent des améliorations notables dans la gestion des communications dans un environnement multi-threadé. Les problèmes de réactivité des communications sont en effet fortement liés à l'ordonnancement des threads et des interactions fortes entre la bibliothèque de threads et la bibliothèque de communications sont indispensables au fonctionnement efficace du réseau lorsque l'application est multi-threadée.

5.1.3 Architecture générale

Afin de concevoir une bibliothèque de communication efficace pour les environnements multi-threadés, nous proposons de faire collaborer étroitement l'ordonnanceur de threads et la bibliothèque de communication. Un *gestionnaire d'entrées/sorties générique* est chargé de gérer les interactions entre l'ordonnanceur de threads et les autres composants de la pile logicielle, que ce soit la bibliothèque de communication ou une bibliothèque chargée des entrées/sorties sur disque. L'architecture générale de la pile logicielle ainsi obtenue est décrite par la Figure 5.1. L'utilisation d'un module spécialisé dans la gestion des entrées/sorties permet de simplifier les mécanismes de progression de la bibliothèque de communication et de la bibliothèque d'entrées/sorties. Ce module est distinct de la bibliothèque de threads pour des raisons principalement conceptuelles. En effet, la détection des événements ne fait pas partie des fonctionnalités attendues d'une bibliothèque de threads. Le module de détection prend en charge la progression des communications et des entrées/sorties. Nous ne considérons ici que les bibliothèques de communication, mais les mécanismes décrits sont également applicables à des bibliothèques d'entrées/sorties au sens large.

Bibliothèque de communication. Le module chargé des communications réseaux ou de la gestion des entrées/sorties sur disque rassemble les requêtes de l'application et les traite (optimisation des messages, répartition sur les différents périphériques disponibles, etc.) La détection des événements (terminaison d'une requête réseau ou d'une lecture disque par exemple) ainsi que la progression des communications sont déléguées au gestionnaire d'entrées/sorties.

Afin d'exploiter efficacement les multiples cœurs disponibles, le traitement des communications est parallélisé sous forme de tâches. La bibliothèque de communication fournit les différentes tâches à exécuter au gestionnaire d'entrées/sorties qui se charge de les répartir sur l'ensemble de la machine.

Bibliothèque de threads. L'ordonnancement des threads est réalisé par la bibliothèque de threads qui met également en œuvre des primitives de synchronisation à disposition de l'application et des

autres modules de la pile logicielle. Afin d'exploiter efficacement les cœurs de calcul, la bibliothèque de threads met à disposition du gestionnaire d'entrées/sorties des informations sur la charge de la machine, notamment concernant la disponibilité des différentes unités de calcul, le nombre de threads prêts à s'exécuter, etc. De plus, l'ordonnanceur de threads profite des processeurs inutilisés pour appeler le gestionnaire d'entrées/sorties afin qu'il puisse traiter les communications en cours.

Gestionnaire d'entrées/sorties. La détection des événements d'entrées/sorties – tels que la terminaison d'une communication – et la parallélisation de la bibliothèque de communication sont prises en charge par le gestionnaire d'entrées/sorties. Ce dernier rassemble les différents événements à détecter ainsi que les tâches à traiter et les répartit sur les unités de calcul de la machine. Le gestionnaire d'entrées/sorties fournit donc deux services à la bibliothèque de communication : un service de détection des événements afin de gérer la progression des communication et un service de gestion de tâches permettant de paralléliser le traitement des communications. Pour fournir ces services, le gestionnaire d'entrées/sorties prend en compte les informations de la bibliothèque de communication (type de réseau utilisé, placement du thread associé à la communication, etc.) et de l'ordonnanceur de threads (disponibilité des différents cœurs de la machine, etc.) En retour, des directives sont envoyées à l'ordonnanceur de threads afin de donner la main rapidement aux threads qui viennent d'être réveillés. De plus, les traitements (que ce soit la détection d'un événement ou l'exécution d'une tâche) sont effectués en prenant en compte la localité des données. Ainsi, les communications entre les différents cœurs de la machine sont optimisées et peuvent bénéficier des caches partagés par exemple.

5.2 Intégration des communications dans l'ordonnanceur de threads

La progression des communications permet de recouvrir les transferts réseau par du calcul et donc de réduire l'impact des communications sur les performances d'une application. La qualité de la progression des communications dépend fortement du temps de réaction aux événements réseau. Nous présentons dans cette section les différents mécanismes impliqués dans la collaboration entre l'ordonnanceur de threads et la bibliothèque de communication et permettant d'améliorer cette réactivité aux événements. Le gestionnaire d'entrées/sorties que nous proposons ici prend en charge la détection des événements du réseau afin de faire progresser les communications. Il sert donc de *moteur de progression* pour la bibliothèque de communication qui peut lui déléguer cette tâche.

5.2.1 Sous-traiter la détection des événements de communication

Du point de vue de la bibliothèque de communication, la détection des événements du réseau – notamment la terminaison d'une requête – est une tâche fastidieuse dans un environnement multi-threadé. En effet, la bibliothèque de communication n'a pas la main sur l'ordonnancement des threads et ne connaît pas la charge des processeurs de la machine. Ainsi, les décisions prises concernant la détection des événements dépendent d'hypothèses sur la charge de la machine qui sont invérifiables par la bibliothèque de communication. Par exemple, en supposant qu'un des processeurs est inutilisé, la bibliothèque de communication va scruter le réseau activement. Cette méthode, bien que très efficace si un processeur est libre, peut se révéler contre-productive si toutes les unités de calcul sont utilisées. La gestion de la détection des événements directement dans la bibliothèque de communication semble donc être peu efficace.

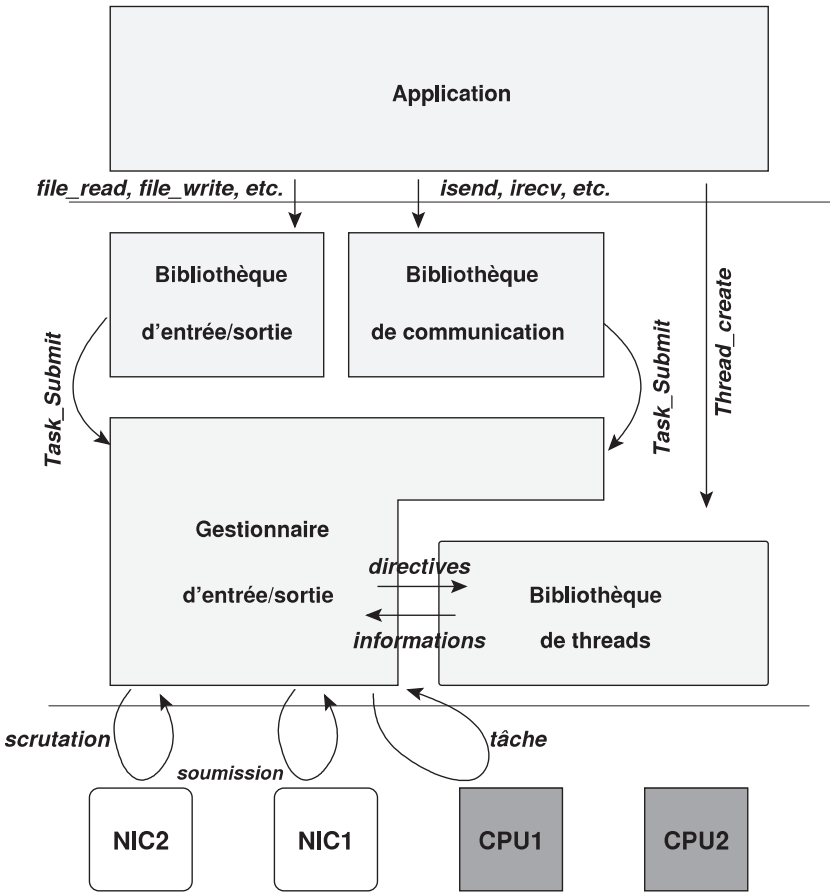

FIGURE 5.1 – *Architecture générale de la pile logicielle.*

Nous proposons donc de déléguer la détection des événements à un module logiciel séparé de la bibliothèque de communication. Ce module, en interaction avec l'ordonnanceur de thread, est plus à même de gérer les problèmes liés au multi-threading. Ainsi, la conception d'une bibliothèque de communication s'en trouve simplifiée et l'on peut se concentrer sur les fonctionnalités propres aux réseaux (optimisation des communications, gestion de plusieurs réseaux simultanément, multiplexage, etc.)

Pour cela, la bibliothèque de communication doit fournir au système de détection des événements des méthodes permettant d'accéder aux réseaux. À la manière du serveur de détection des événements de MARCEL [BDN02], des fonctions de rappel (*Callbacks*) sont définies à l'initialisation de la bibliothèque de communication. Ces fonctions permettent d'exploiter les différents modes de détection fournis par les interfaces de communication de bas-niveau. La bibliothèque de communication définit donc différents *callbacks* pour effectuer une scrutation sur une ou plusieurs requêtes réseau ainsi que des fonctions détectant la terminaison d'une ou plusieurs requêtes grâce à un appel bloquant. Une fois que ces *fonctions de rappel* sont spécifiées au module de détection d'événements, la bibliothèque peut soumettre une requête à ce dernier afin de déléguer la détection de la terminaison d'une communication réseau. L'événement attendu est ensuite détecté en arrière-plan de manière transparente. En utilisant les différentes fonctions de rappel fournies par la bibliothèque de communication, le module de détection des événements interroge le réseau jusqu'à ce que l'événement se produise. Le *callback* détectant l'événement peut alors réagir en conséquence en répondant à la demande de *rendez-vous*, en réveillant un thread qui attend la fin de la communication, etc.

La gestion de plusieurs liens réseau différents entre plusieurs machines (également appelée *multirail hétérogène*) est bien sûr possible. La bibliothèque de communication doit alors définir les différents *callbacks* pour chaque technologie réseau exploitée. Lors de la soumission d'une requête au module de détection des événements, le type de réseau à exploiter est alors ajouté.

5.2.2 Collaboration avec l'ordonnanceur de threads

Du point de vue du module de détection des événements, la progression des communications peut s'avérer difficile à réaliser de manière efficace. Comme nous l'avons montré dans la section 4.1.1, l'utilisation de threads dédiés à la détection des événements peut aider dans certains cas, mais pose des problèmes de performances. En effet, si le système est surchargé (*i.e.* il y a plus de threads que d'unités de calcul), le thread de progression risque de n'être ordonnancé que rarement et le temps de réaction à un événement réseau peut devenir élevé. De plus, lorsque des processeurs sont libres, l'utilisation de threads de progression risque d'entraîner des changements de contexte dont les coûts se répercutent directement sur le temps de réaction. D'une manière générale, la détection des événements doit distinguer deux types de systèmes :

- **Les systèmes chargés** pour lesquels tous les cœurs de la machine sont exploités par des threads. Ce type de configuration nécessite de détecter les événements en arrière-plan pendant que les threads de l'application calculent. Les méthodes de détection par scrutation sont ici peu efficaces car elles nécessitent d'interrompre les calculs fréquemment. Les méthodes de détection utilisant des interruptions sont par contre les plus adaptées pour délivrer un temps réaction rapide.
- **Les systèmes sous-utilisés** pour lesquels certains processeurs sont inutilisés. Les unités de calcul inutilisées peuvent alors servir à la détection des événements sans pénaliser les threads de l'application. Les méthodes de détection basées sur des interruptions ne sont pas très efficaces ici du fait du surcoût engendré par l'invocation du traitant d'interruption. Les méthodes de scrutation sont par contre parfaitement adaptées à cette situation : les processeurs inutilisés peuvent faire une attente active et ainsi

détecter très rapidement l'événement réseau.

L'application peut passer d'une configuration à l'autre au cours de son exécution – par exemple, lors d'une synchronisation, un système surchargé peut devenir sous-utilisé, avant de redevenir surchargé lorsque la synchronisation se termine – et il est donc important de pouvoir adapter la méthode de détection à la configuration actuelle du système. La collaboration étroite avec l'ordonnanceur de threads permet de prendre en compte la charge de la machine lors du choix de la méthode de détection. En effet, la bibliothèque de threads expose ces informations au module de détection des événements.

Toutefois, le surcoût engendré par un appel bloquant basé sur une interruption est conséquent (il est du même ordre de grandeur que la latence du réseau) et ce type de méthode de détection doit donc être évité lorsque cela est possible. La scrutation étant généralement très peu coûteuse, on utilisera de préférence ce mode de détection. Afin de détecter rapidement les événements provenant du réseau, il convient d'appeler les *callbacks* de scrutation fréquemment. Étant donné le faible coût des méthodes de scrutation (généralement de l'ordre de la centaine de nanosecondes) celles-ci peuvent être appelées très fréquemment sans que cela n'ait d'impact sur les performances des phases de calcul.

Exploitation des points-clés de l'ordonnancement. La bibliothèque de threads ordonnance donc le module de détection des événements à chaque fois qu'une opération d'ordonnancement est nécessaire (lors d'un signal d'horloge ou d'un changement de contexte principalement). Le module de détection exécute alors les *callbacks* de chaque requête en attente. Lorsqu'un processeur est inutilisé (ou *idle*), l'ordonnanceur de threads entre dans une boucle qui consulte les listes de threads pour en trouver un à exécuter. On profite de cette boucle pour appeler le module de détection : tant qu'aucun thread n'est prêt à s'exécuter sur ce processeur, les fonctions de rappel servant à détecter les événements sont appelées. L'étroite collaboration entre le module de détection des événements et l'ordonnanceur de threads permet donc d'appeler fréquemment les *callbacks* fournis par la bibliothèque de communication. Les processeurs inutilisés sont exploités afin de minimiser le temps de réaction et les appels au module de détection lors des changements de contexte et des signaux d'horloge permettent de garantir une fréquence de scrutation minimum. Ainsi, la bibliothèque de communication est assurée de détecter un événement avec un temps de réaction borné par la durée d'un *quantum de temps* (typiquement 10 ms).

5.2.3 Gestion de la réactivité sur un système chargé

Bien que les méthodes de scrutation présentées dans la section précédente permettent de détecter rapidement les événements lorsque l'ordonnanceur a la main, elles souffrent d'un problème de réactivité lorsque tous les processeurs sont constamment utilisés. Le temps de réaction, borné à la durée d'un *quantum de temps* (*timeslice*), peut s'avérer beaucoup trop élevé pour les applications sensibles à la latence. Il est alors nécessaire d'utiliser les fonctions de rappels basées sur des interruptions que la bibliothèque de communication fournit. Malgré le surcoût engendré par ces méthodes de détection, le temps de réaction ainsi obtenu est largement inférieur à celui obtenu par scrutation dans ce contexte précis.

L'utilisation de méthodes bloquantes peut toutefois être problématique dans le module de détection des événements. En effet, celui-ci s'exécute directement dans le contexte de la bibliothèque de communication ou de l'ordonnanceur. Un appel bloquant risque alors de bloquer l'ordonnanceur de thread ou la bibliothèque de communication. Il convient donc de prendre certaines précautions et d'éviter les appels bloquants dans un contexte non maîtrisé. Nous proposons d'utiliser des threads supplémentaires dédiés aux appels bloquants. Ces threads, endormis la plupart du temps, sont réveillés pour effectuer un appel

bloquant à la place d'un autre thread afin que ce dernier ne soit pas bloqué. Ainsi, les threads de l'application ne sont pas bloqués par les *callbacks* bloquants fournis au module de détection des événements.

Le principe de fonctionnement de ces threads dédiés aux appels bloquants est décrit par la Figure 5.2. Une réserve de threads est créée à l'initialisation du module de détection des événements (1) afin de réduire le risque de souffrir du coût de création d'un thread pendant l'exécution de l'application. Ces threads supplémentaires sont bloqués la plupart du temps en attendant une commande. Ils ne consomment donc pas de temps processeur et ne gênent pas les threads de l'application. Lorsqu'une fonction de rappel bloquante doit être exécutée, un des threads supplémentaires est réveillé et l'identifiant de la requête à traiter lui est transmise (2). Lorsque le thread supplémentaire est ordonnancé, il peut exécuter le *callback* bloquant et ainsi attendre l'événement souhaité (3). Un thread de l'application peut alors reprendre son calcul sur le processeur libéré par le thread supplémentaire (4). Lorsque la carte réseau détecte la communication attendue, elle envoie une interruption et le thread de l'application est interrompu. Le thread bloqué en attente de cet événement est alors réveillé et il peut traiter la communication (5). En traitant l'événement, le thread supplémentaire peut signaler à l'application la fin d'une communication ou effectuer un traitement de communication (répondre à un *rendez-vous* par exemple). Lorsque le *callback* se termine, le thread supplémentaire rejoint les autres threads supplémentaires en attente de nouvelles requêtes et un thread de l'application – pas forcément celui qui vient d'être préempté – peut reprendre son calcul.

Ce mécanisme permet donc au module de détection des événements d'exécuter des fonctions de rappel bloquantes sans gêner les threads de l'application. Toutefois, un problème de priorité peut survenir si le nombre de threads actifs est supérieur au nombre d'unités de calcul. Le thread supplémentaire risque alors de ne pas être réveillé dès que l'événement est détecté, entraînant un temps de réaction plus long. En fonction des politiques d'ordonnancement du système d'exploitation, le comportement sera différent : certaines stratégies d'ordonnancement vont donner immédiatement la main au thread réveillé, alors que d'autres se contentent de remettre le thread dans la liste des threads prêts à s'exécuter et recalculer la priorité des threads pour en choisir un. Lorsque cela est possible, il est donc important de choisir une stratégie permettant un ordonnancement rapide du thread. Si ce type de stratégie d'ordonnancement n'est pas disponible, l'augmentation de la priorité des threads supplémentaires permet de donner une indication à l'ordonnanceur. Ainsi, même si le thread réveillé ne sera pas toujours ordonnancé immédiatement, l'ordonnanceur aura quand même tendance à lui donner la main rapidement.

Si l'utilisation de threads supplémentaires permet d'assurer un certain niveau de réactivité, le surcoût engendré par le traitement des interruptions et les changements de contexte a un impact certain sur le temps de réaction. Il convient donc de n'utiliser ce mécanisme que dans certains cas. D'une manière générale, les appels bloquants ne doivent être utilisés que lorsque le mécanisme de scrutation entraînerait un temps de réaction élevé, par exemple lorsque tous les processeurs sont occupés par des threads de l'application. Concevoir un mécanisme de décision "parfait" (*i.e.* qui choisit toujours la méthode la plus adaptée) est impossible car cela nécessite de connaître le déroulement futur des communications. La décision doit donc être prise en fonction d'une heuristique qui, en fonction de l'état de la machine, choisit une des méthodes disponibles. Outre l'occupation des processeurs, cette heuristique prend en compte les directives de la bibliothèque de communication qui, dans certains cas, peut savoir s'il vaut mieux utiliser la scrutation plutôt qu'un appel bloquant sur un thread supplémentaire. Cette heuristique reste imparfaite et nécessiterait un approfondissement afin de prendre en compte des directives données par l'application qui connaît plus précisément les schémas de communication.

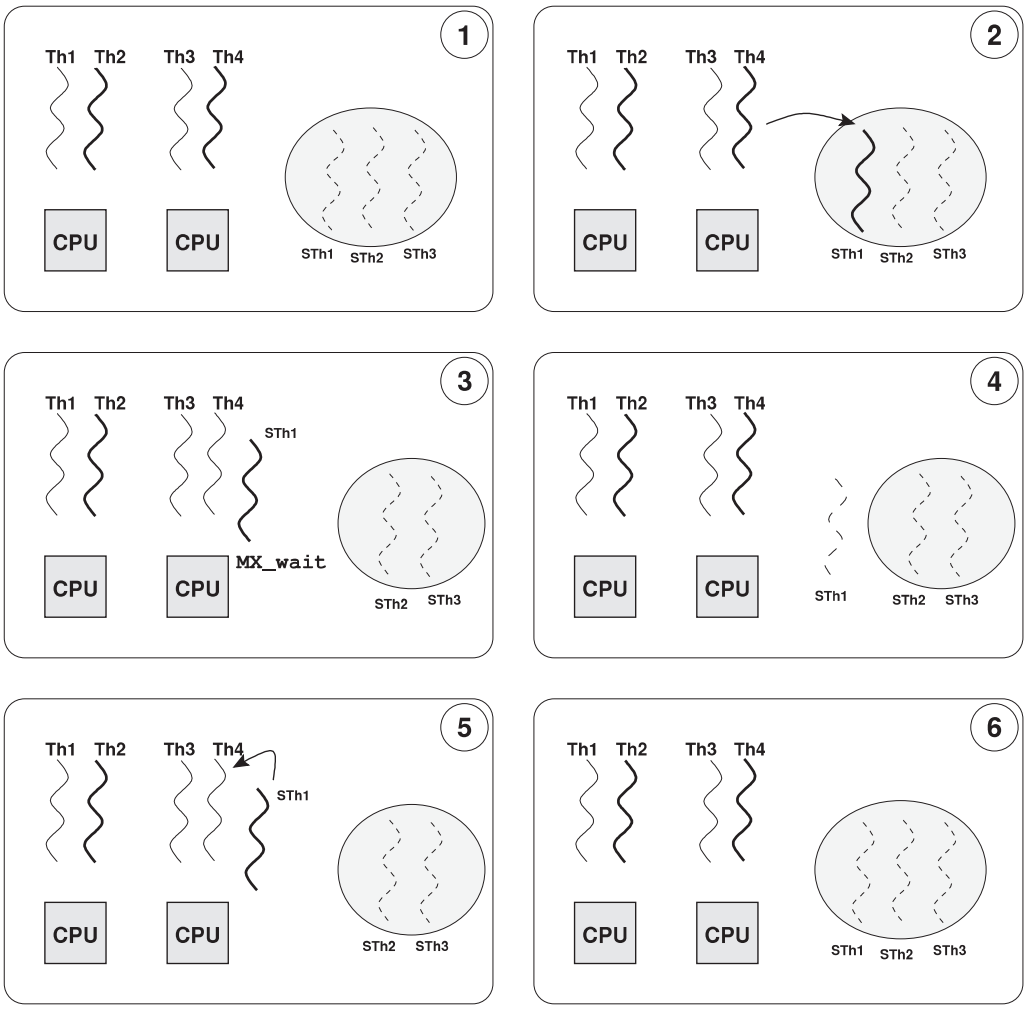

FIGURE 5.2 – *Déroulement d'un appel bloquant exporté.*

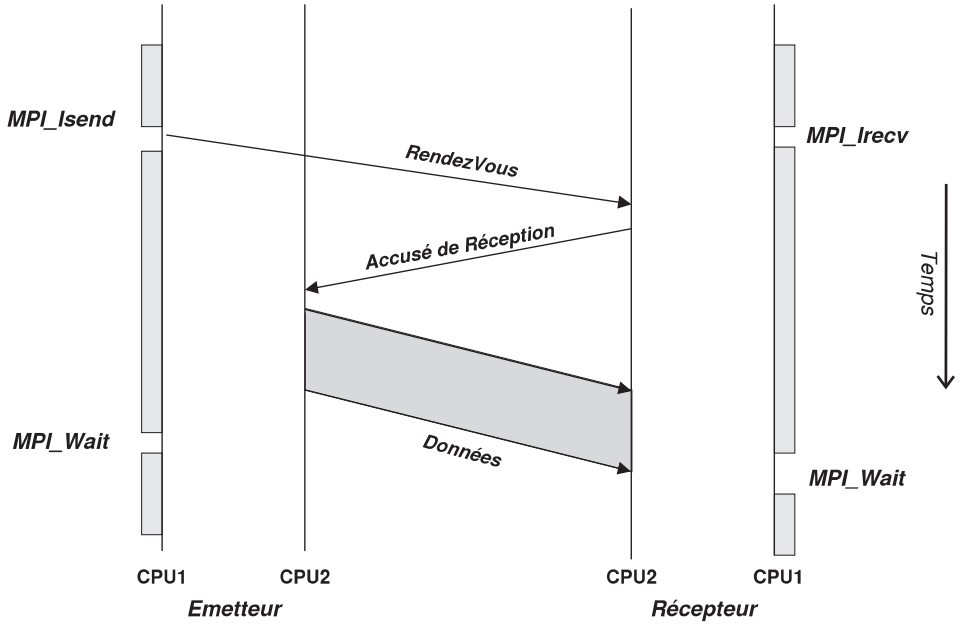

FIGURE 5.3 – *Progression des communication en arrière-plan grâce au* moteur de progression.

5.2.4 Progression des communications en arrière-plan

Le module de détection des événements permet de réagir rapidement à la terminaison d'une requête réseau et peut donc être utilisé par la bibliothèque de communication afin de faire progresser les *rendez-vous* en arrière-plan. Une progression efficace des *rendez-vous* permet aux applications exploitant les primitives non-bloquantes des bibliothèques de communication de cacher le coût des communications en les recouvrant par du calcul. Nous proposons d'utiliser le module de détection des événements comme un *moteur de progression* pour la bibliothèque de communication qui délégue alors la progression des communications au module de détection.

L'utilisation d'un moteur de progression externe à la bibliothèque de communication permet de gérer de manière transparente les communications : les messages sont soumis aux réseaux et leur progression est assurée en arrière-plan. Le fonctionnement de cette délégation de la progression des communications est illustré par la Figure 5.3. Lorsque l'application initie l'échange de message, la demande de *rendez-vous* est envoyée directement. La détection de l'accusé de réception est ensuite laissée au *moteur de progression* et l'application peut calculer sans se soucier de la progression de la communication. Lorsque l'arrivée l'accusé de réception est détectée, la fonction de rappel se charge d'y répondre directement. La progression des communications est assurée en définissant des fonctions de rappel capable de détecter un événement et d'y réagir. Le comportement est similaire du côté du récepteur : une fois la requête de réception soumise au réseau, la détection de l'arrivée de la demande de *rendez-vous* est gérée par le module de détection qui peut répondre pendant que l'application calcule. La bibliothèque de communication et l'application voient donc les communications progresser de manière transparente sans qu'elles aient à intervenir.

Le moteur de progression s'appuie sur le module de détection des événements pour traiter les événements provenant du réseau. Lorsque la bibliothèque de communication soumet un message à faire progresser, l'événement associé – l'arrivée d'une demande de *rendez-vous*, d'un accusé de réception, etc. – est

transmis au module de détection. Les *callbacks* fournis à l'initialisation sont alors utilisés en fonction de l'état de la machine : si des processeurs sont inutilisés, l'événement est détecté par scrutation, sinon les *callbacks* bloquants sont exécutés. Ainsi, lorsqu'une *fonction de rappel* détecte l'événement attendu, elle peut traiter le *rendez-vous* en soumettant au réseau une réponse appropriée (un accusé de réception si l'on vient de recevoir une demande de *rendez-vous*, les données à transférer si l'on a reçu un accusé de réception).

En s'appuyant sur le module de détection des événements, nous proposons donc un moteur de progression permettant la progression des communications en arrière-plan. L'exploitation des cœurs inutilisés permet d'assurer une détection rapide des messages constituant le *rendez-vous* grâce aux fonctions de scrutation fournies par l'application. Lorsque la machine est surchargée (*i.e.* toutes les unités de calcul sont utilisées par l'application), les méthodes de détection basées sur les interruptions envoyées par la carte réseau permettent une progression des communications, certes moins efficace que lorsqu'un cœur est inutilisé, mais suffisamment performante pour que les communications soient recouvertes par du calcul de l'application.

5.3 Gestion des flux de communication concurrents

Le développement des architectures multi-cœurs dans les grappes de calcul et l'augmentation du nombre de cœurs par nœud ont poussé de nombreux développeurs d'applications à délaisser le modèle de programmation traditionnel qui consiste à lancer un processus par unité de calcul. Les approches mélangeant communications et multi-threading se sont répandues récemment et les bibliothèques de communication doivent donc faire face à de nouvelles problématiques. Le support des applications multi-threadées par les bibliothèques de communication est aujourd'hui quasiment obligatoire. Nous présentons donc dans cette section les mécanismes de protection que nous proposons d'utiliser à la fois dans le module de détection des événements et dans la bibliothèque de communication. Nous présentons également la façon dont sont gérées les attentes concurrentes.

5.3.1 Protection contre les accès concurrents

Le développement massif des applications se basant sur un modèle hybride mélangeant communications et multi-threading fait du support des accès concurrents un problème très important pour les bibliothèques de communication. En effet, si l'application utilise plusieurs threads pour accéder au réseau, ces derniers risquent de modifier simultanément les mêmes structures de données (la liste des requêtes réseau à traiter par exemple) ou d'utiliser de manière concurrente une interface réseau non-protégée. Il convient donc de prévoir des mécanismes de protection performants empêchant que les modifications concurrentes de structures de données ne posent problème et arbitrant les accès aux interfaces réseau de bas niveau lorsque celles-ci ne supportent pas les accès concurrents.

5.3.1.1 Verrous à gros grain

Le mécanisme le plus simple pour assurer l'intégrité des données au sein de la bibliothèque de communication lorsque plusieurs threads y accèdent simultanément consiste à utiliser un verrou à gros grain. Ce mécanisme, décrit par la Figure 5.4, assure une exclusion mutuelle entre les threads. Les threads de l'application prennent le verrou à chaque accès à la bibliothèque de communication et ne le relâchent

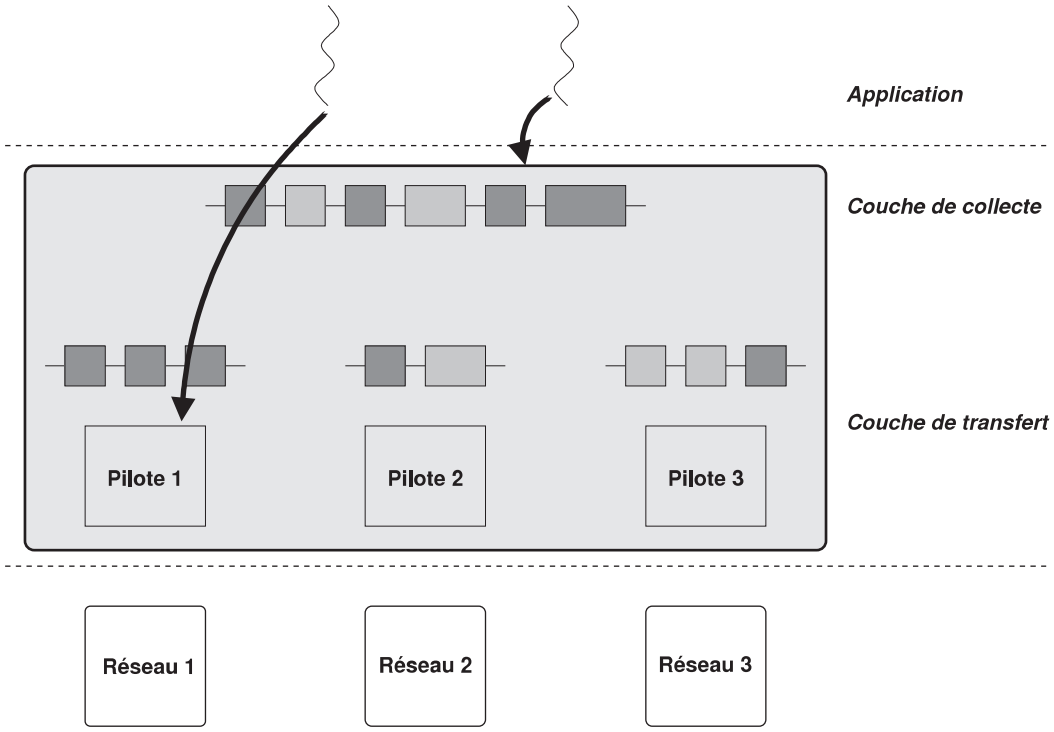

FIGURE 5.4 – *Verrou à gros grain protégeant l'accès à la bibliothèque de communication. La zone grisée correspond à la portée du verrou.*

qu'en sortant. Ainsi, lorsque plusieurs threads accèdent simultanément à la bibliothèque, seul le premier à prendre le verrou pourra traiter sa communication et les autres devront attendre que le verrou soit relâché pour accéder au réseau.

Le surcoût engendré par l'utilisation de ce mécanisme dans une bibliothèque de communication est très réduit lorsque l'application utilise peu de threads. Le surcoût correspond alors au temps nécessaire à prendre et relâcher un verrou, ce qui ne représente que quelques dizaines de nanosecondes sur une machine moderne. Toutefois, le verrou global à toute la bibliothèque de communication s'avère peu performant lorsque l'application utilise plusieurs threads pour communiquer comme le montrent les mesures présentées dans la Figure 3.7 page 29. Ce mécanisme n'est donc à utiliser que lorsque l'application exécute plusieurs threads, mais que ceux-ci n'accèdent que rarement à la bibliothèque de manière concurrente. Les applications soumettant la bibliothèque de communication à une forte concurrence requièrent donc des techniques plus évoluées afin que les performances ne se dégradent pas.

5.3.1.2 Verrous à grain fin

Les problèmes de performance rencontrés avec le mécanisme de verrou à gros grain sont dus à la taille de la *section critique*. En effet, le verrou global protège des accès concurrents la bibliothèque de communication toute entière, même lorsque les threads n'accèdent pas aux mêmes structures de données. Pour améliorer les performances obtenues lorsque plusieurs threads accèdent de manière concurrente à la bibliothèque de communication, il convient donc de réduire la taille de la *section critique*. L'utilisation de verrous séparés pour les différentes parties de la bibliothèque de communication permet d'éviter

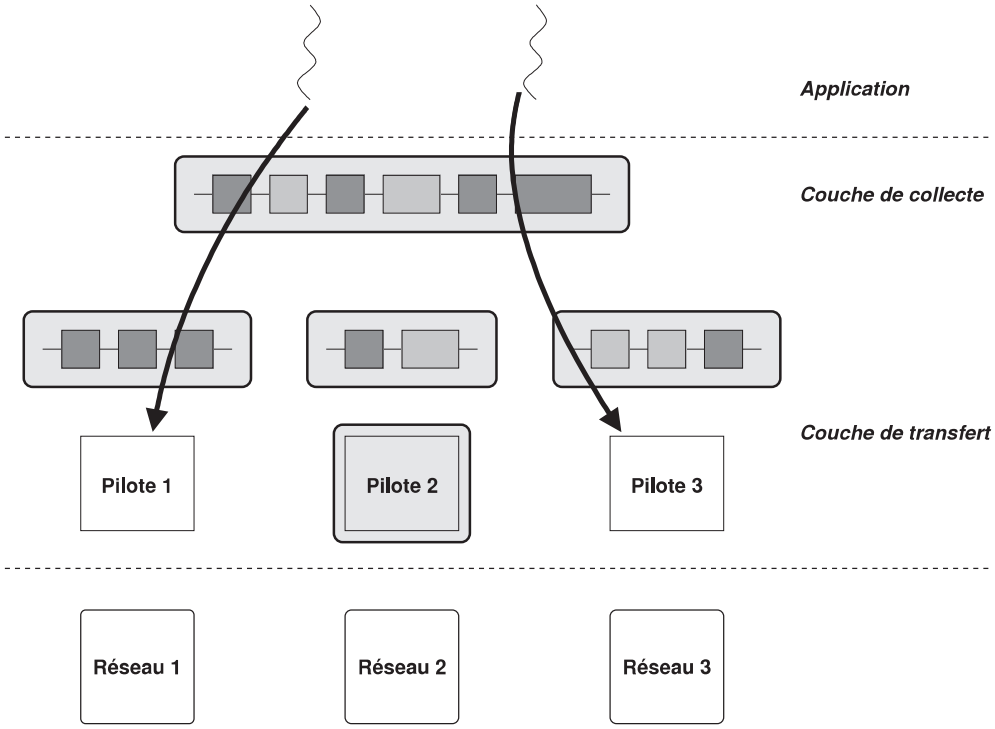

FIGURE 5.5 – *Verrous à grain fin protégeant l'accès à la bibliothèque de communication. Les zones grisées correspondent à la portée des verrous.*

la contention sur les verrous lorsque les threads accèdent à des fonctions distinctes. Le fonctionnement de verrous à grain fin dans une bibliothèque est décrit par la Figure 5.5 : les différentes structures de données partagées sont chacunes protégées par un verrou afin d'éviter les problèmes de contention. L'accès aux interfaces de communication de bas niveau non protégées nécessite également la prise d'un verrou. Dans une bibliothèque de communication, les structures de données partagées nécessitant une attention particulière sont généralement des listes de requêtes réseau à traiter. Le mécanisme de verrous à grain fin implique alors de prendre un verrou pour accéder à une liste (que ce soit pour la modifier ou pour la consulter) et de le relâcher une fois l'opération terminée. Ainsi, si plusieurs threads tentent d'accéder à une liste de manière concurrente, leurs accès seront séquentialisés et l'intégrité de la liste sera conservée.

Le mécanisme de verrouillage à grain fin est bien adapté aux applications accédant intensivement à la bibliothèque de communication depuis plusieurs threads. La petite taille des sections critiques permet alors aux différents threads de traiter leurs communications respectives en parallèle. Toutefois, les applications nécessitant des mécanismes de protection mais qui ne font que rarement des appels concurrents risquent de souffrir du surcoût du verrouillage à grain fin sans en tirer partie. Pour ce type d'applications, le mécanisme de verrouillage à gros grain est alors un choix plus judicieux puisque le surcoût sur les performances brutes du réseau est réduit. Il convient donc de choisir le type de verrouillage en fonction des besoins de l'application afin de tirer les meilleures performances de la bibliothèque de communication. Si une sélection statique – au moment de la compilation du programme – est simple à mettre en œuvre, il serait intéressant que l'application puisse choisir "à la volée" le type de verrouillage à utiliser. Cela permettrait par exemple de bénéficier des avantages des différents mécanismes pour chaque partie de l'application : les phases au cours desquelles les accès concurrents sont rares pourrait utiliser le ver-

rouillage à gros grain, alors que le verrouillage à grain fin pourrait servir lorsque les threads accèdent intensivement de manière concurrente à la bibliothèque de communication.

5.3.2 Attentes concurrentes

Si les mécanismes de protection sont indispensables pour assurer l'intégrité des structures de données de la bibliothèque de communication, ils ne permettent pas à eux seuls d'obtenir de bonnes performances pour des applications multi-threadées. Par exemple, la gestion des attentes concurrentes – c'est-à-dire le comportement de la bibliothèque de communication lorsque plusieurs threads attendent la fin de leurs communications respectives simultanément – peut influencer énormément les performances de l'application. Nous présentons ici les différents mécanismes pouvant être implémentés dans une bibliothèque de communication afin de gérer ces attentes concurrentes. Ces mécanismes sont généralement implémentés dans les primitives d'attente de fin de communication (`MPI_Wait` par exemple). Nous nous intéressons donc tout particulièrement aux différentes façons d'implémenter des fonctions d'attente.

Attente active. La plupart des bibliothèques de communication implémentent et utilisent les primitives d'attente à l'aide d'un mécanisme d'*attente active* : lorsqu'un thread souhaite attendre qu'une communication se termine, la primitive appelée scrute l'interface de communication jusqu'à ce que l'événement attendu soit détecté. Ce mécanisme offre de très bonnes performances dans un environnement monothreadé grâce au faible coût des méthodes de scrutation. Toutefois, ce mécanisme est problématique lorsque plusieurs threads attendent des communications de manière concurrente. Outre la contention engendrée par ces scrutations concurrentes, certains effets mémoire peuvent provoquer des déséquilibres dans l'application lorsqu'elle est exécutée sur des machines NUMA. En effet, les primitives de synchronisation utilisées dans la bibliothèque de communication peuvent alors concentrer les prises de verrou sur un seul nœud NUMA, empêchant ainsi les threads s'exécutant sur les autres nœuds d'acquérir le verrou [Lam05]. Les mécanismes d'attente active introduisent également des problèmes au niveau de l'ordonnancement des threads. En effet, si plusieurs threads attendent de manière concurrente la terminaison de leurs communications respectives, les processeurs sur lesquels les threads s'exécutent sont alors sous-utilisés. Plutôt que d'exécuter le même traitement – la scrutation du réseau – sur plusieurs processeurs simultanément, ces derniers pourraient servir à exécuter des threads de l'application.

Attente passive. Du point de vue de l'ordonnanceur de threads, les threads en attente de la terminaison d'une communication devraient donc se bloquer et laisser le processeur aux autres threads de l'application. Les primitives d'attente implémentées sous forme d'*attente passive* permettent de ne pas occuper de processeur lors de l'attente de la terminaison d'une communication. Les threads sont ici bloqués grâce à une primitive de synchronisation bloquante (une condition ou un sémaphore généralement). La détection de l'événement attendu au niveau du réseau permet ensuite de débloquer les threads en attente qui peuvent alors être ordonnancés. Ce mécanisme permet donc de libérer les ressources de calcul et de les laisser aux threads de calcul de l'application qui peut alors faire progresser ses calculs plus rapidement. De plus, les attentes concurrentes ne produisent pas ici de contention puisque les threads sont bloqués. Toutefois, le mécanisme d'attente passive a l'inconvénient de présenter des surcoûts importants dus aux changements de contexte. En effet, les threads en attente sont désordonnancés lorsqu'ils atteignent la primitive bloquante. Un deuxième changement de contexte a ensuite lieu lorsque la communication se termine et que l'ordonnanceur rend la main au thread bloqué. Ces changements de contexte coûteux ont un impact direct sur les performances du réseau observée par l'application.

Attente mixte. En combinant les mécanismes d'attente active et passive, il est possible de ne conserver que leurs avantages respectifs. Cette combinaison appelée *attente mixte* consiste à effectuer une attente active pendant un certain temps et à basculer vers une attente passive si au bout de ce laps de temps la communication n'est pas terminée. Ainsi, si la communication se termine rapidement, les performances obtenues sont celles de l'attente active. Si le thread doit attendre longtemps la fin de la communication, le basculement vers une attente passive permet de libérer le processeur – et ainsi faire progresser les threads de calculs de l'application – et le surcoût dû aux changements de contexte est amorti par la durée de la communication. En effet, si le thread attend la terminaison d'une communication pendant 1 ms, le coût du changement de contexte (généralement de l'ordre de quelques centaines de nanosecondes ou quelques microsecondes) devient négligeable. Ce type d'attente mixte est similaire au mécanisme de verrou adaptatif (*"fixed-spin lock"*) [KLMO91] et nécessite donc les mêmes précautions. La principale difficulté est de déterminer la durée au-delà de laquelle le thread doit se bloquer. Si celle-ci est trop courte, les threads se bloqueront presque immédiatement et le surcoût des changements de contexte risquent de détériorer les performances de l'application. Si la durée est trop longue, les threads passeront leur temps en attente active, empêchant les autres threads de calculer et entraînant de la contention. La durée idéale pendant laquelle le thread doit scruter le réseau dépend du comportement de l'application et des performances du système. Le compromis entre le surcoût de l'attente passive et la sous-utilisation du processeur due à l'attente active doit être arbitré par les besoins de l'application. Il faut évaluer la part de surcoût sur les communications qui est acceptable par l'application. À partir de ce surcoût et du coût d'un changement de contexte, on peut alors calculer la durée de scrutation à l'aide de la formule :

$$Surcoût = \frac{C_{chgt\_contexte}}{T_{scrutation}}$$

La durée de scrutation choisie peut varier d'une application à l'autre : si la latence du réseau influe beaucoup sur les performances de l'application, le seuil de passage au mode attente passif sera relevé afin de privilégier la scrutation. À l'inverse, lorsque la latence a peu d'impact sur les performances, la réduction de la durée de scrutation permet d'ordonnancer les autres threads rapidement et donc de faire progresser les calculs. Fixer une durée de scrutation est un problème difficile qui nécessite une analyse de l'application. En considérant qu'un surcoût de 5 % est acceptable pour une majorité d'applications, nous choisissons une durée de scrutation correspondant à ce surcoût. Toutefois, ce choix est forcément mauvais pour certaines applications et il serait nécessaire d'étudier plus profondément ce problème. Une solution adaptative consistant à analyser les communications et le temps nécessaire à leur terminaison permettrait de choisir une durée de scrutation automatiquement afin de minimiser l'impact de cette attente sur les performances de l'application.

5.4 Traitement des communications en parallèle

L'augmentation du nombre de cœurs par nœuds dans les grappes modernes a entraîné de nombreuses contraintes aux bibliothèques de communication telles que le support des applications multi-threadées ou la gestion de multiples interfaces de communication simultanément. Les nombreux cœurs des machines modernes ouvrent pourtant de nouvelles perspectives permettant de réduire le coût des communications. Il est par exemple possible de paralléliser les traitements de la bibliothèque de communication afin d'exploiter au mieux les nœuds d'une grappe. Nous présentons dans cette section des mécanismes permettant de traiter les communications en parallèle afin de mieux répartir la charge de calcul et de réduire le coût des communications.

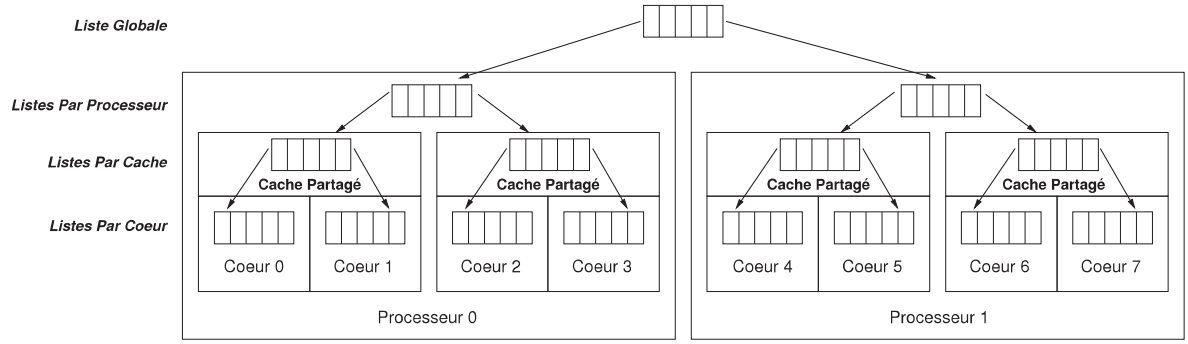

FIGURE 5.6 – *Listes hiérarchiques appliquées à la topologie d'une machine.*

5.4.1 Mécanisme d'exportation de tâches

Comme nous l'avons décrit dans la section 5.1.3, le gestionnaire d'entrées/sorties fournit un service de détection d'événements permettant de faire progresser les communications en arrière-plan, et un service de gestion de tâches permettant de paralléliser simplement les traitements de communication. Pour cette parallélisation, nous proposons d'utiliser une approche basée sur des tâches pouvant être exécutées sur n'importe quel cœur de la machine. Le mécanisme de tâche est entièrement géré par le gestionnaire d'entrées/sorties et la bibliothèque de communication peut donc concentrer ses efforts sur le traitement des communications. La collaboration étroite entre l'ordonnanceur de threads et le gestionnaire d'entrées/sorties permet la conception d'un mécanisme d'exportation de tâches efficace. Tout comme le module de détection fournit un service de détection des événements à la bibliothèque de communication, le *gestionnaire de tâches* offre un service d'exportation de tâches de communication. La différence entre ces tâches et les threads fournis par la bibliothèque de threads vient de la durée très courte des tâches (moins de quelques dizaines de microsecondes) et de leurs faibles contraintes du point de vue du gestionnaire de tâches. Par exemple, les tâches ne peuvent pas se bloquer et ne peuvent donc être exécutées directement sur la pile d'un thread.

L'interface du gestionnaire de tâches permet à la bibliothèque de communication de définir des tâches en précisant les processeurs sur lesquelles celles-ci peuvent être exécutées. Ce mécanisme permet de prendre en compte la localité des données. Ainsi, un thread qui soumet une tâche peut spécifier un processeur (ou un groupe de processeurs) proche du cœur sur lequel il s'exécute pour bénéficier des effets de cache lors du traitement des données par la tâche.

Le gestionnaire de tâches tire parti des masques de processeurs fournis par la bibliothèque de communication en adoptant une approche hiérarchique pour le traitement des tâches. Un arbre de listes de tâches est donc défini et est appliqué à la topologie de la machine. Cette hiérarchie de *listes de tâches*, illustrée par la Figure 5.6, permet de réduire la contention lors de l'accès aux listes et permet un meilleur passage à l'échelle quand le nombre de cœurs augmente. Lors de la soumission d'une tâche, le masque de processeurs est étudié et la tâche est insérée dans la liste correspondant au masque. Ainsi, une tâche dont le masque de processeurs ne contient qu'un seul cœur sera placée dans une des listes du plus bas-niveau alors qu'une tâche pouvant s'exécuter sur n'importe quel processeur sera placée dans la liste globale. Outre les effets de cache que cette approche peut apporter, la contention lors des accès aux listes de tâches est ici réduite. En effet, une liste de tâche correspondant à un seul cœur ne sera modifiée que par un nombre restreint de processeurs.

Faisant partie du gestionnaire d'entrées/sorties, le gestionnaire de tâches collabore étroitement avec l'or-

Algorithm 1 Task_Schedule

```
for Queue = Per_Core_Queue to Global_Queue do
  for Task in Queue do
    run(Task)
    if OptionRepeatIsSet(Task) then
      Enqueue(Queue, Task)
    end if
    Task ⇐ Get_Task(queue)
  end for
end for
```

donnanceur de threads qui lui donne la main à certains moments-clés (lorsqu'un processeur est inutilisé, lors d'un signal d'horloge ou d'un changement de contexte, etc.) Les tâches à traiter sont alors exécutées en suivant l'algorithme 1 : les tâches de la liste locale (correspondant au cœur sur lequel s'exécute le thread courant) sont traitées. Lorsque toutes les tâches de la liste ont été exécutées, la liste du niveau supérieur est sélectionnée et ses tâches sont traitées. La sélection des tâches dans une liste se fait grâce à l'algorithme 2 : le contenu de la liste est tout d'abord évalué sans prendre de verrou afin d'éviter toute contention inutile. Si la liste comporte des tâches à traiter, le verrou est pris et le contenu de la liste est revérifié. Ce mécanisme permet d'éviter les *"race conditions"* tout en réduisant les surcoûts dus au verrouillage puisque le verrou n'est pris que quand la liste n'est pas vide. L'utilisation de listes hiérarchiques permet donc de conserver l'affinité des tâches – seuls les cœurs spécifiés à la création de la tâche peuvent l'exécuter – tout en réduisant la contention sur les verrous des listes.

Algorithm 2 Get_Task(Queue)

```
Result ⇐ NULL
if Not_Empty(Queue) then
  LOCK(Queue)
  if Not_Empty(Queue) then
    Result ⇐ Dequeue(queue)
  end if
  UNLOCK(Queue)
end if
return Result
```

5.4.2 Décomposer le traitement des communications

Les bibliothèques de communication telles que OPEN MPI ou MPICH2 ne prennent pas en compte les architectures multi-cœurs pour traiter les communications. En effet, la plupart des traitements de communication sont réalisés de manière séquentielle. Il est alors très difficile de répartir les opérations de communication sur les différentes unités de calcul disponibles. Nous proposons d'adopter une approche basée sur le multi-threading afin de réduire le coût des communications et d'exploiter au mieux les processeurs multi-cœurs équipant les machines modernes. Pour cela, le traitement d'une communication doit être vu comme une suite d'opérations. Par exemple, comme le montre la Figure 5.7, l'envoi de données nécessite (a) d'enregistrer la requête, (b) de soumettre la requête au réseau et (c) d'attendre la terminaison de la communication. Ces opérations ne doivent pas nécessairement s'effectuer sur un seul

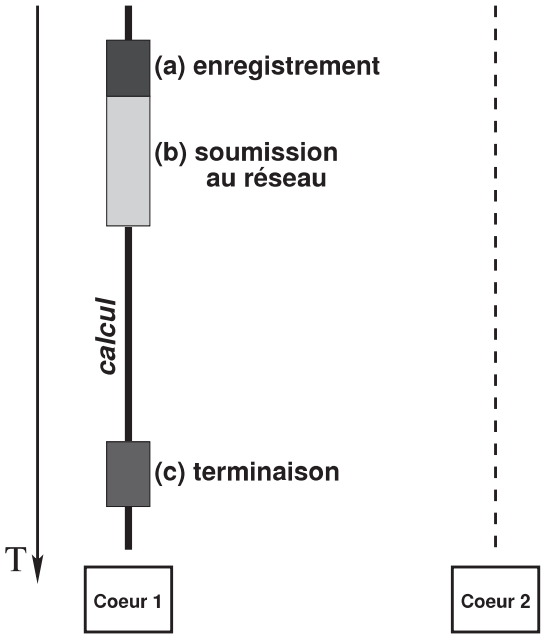

FIGURE 5.7 – *Traitement séquentiel d'une communication.*

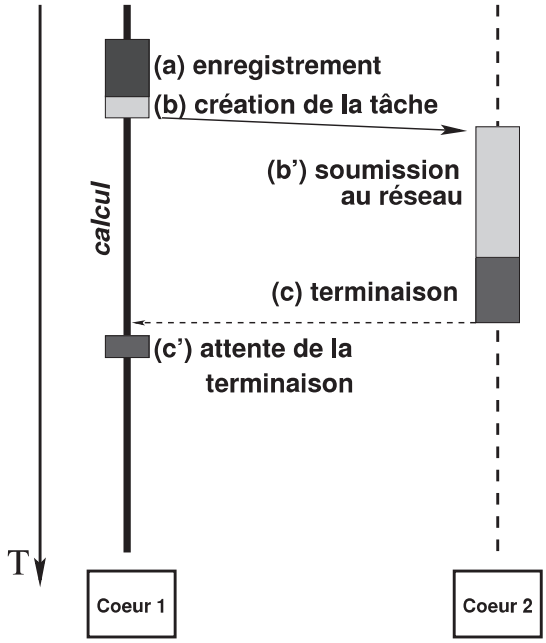

FIGURE 5.8 – *Traitement d'une communication en parallèle.*

cœur, la seule contrainte est de respecter les dépendances entre les tâches. Il est donc possible d'exploiter les cœurs libres de la machine pour exécuter certaines de ces opérations, comme le montre la Figure 5.8.

Grâce à la décomposition des traitements, les primitives de communication non-bloquantes peuvent être exécutées en parallèle et permettent de recouvrir les communications par du calcul. En effet, les primitives non-bloquantes se contentent d'enregistrer la requête et de soumettre une *tâche* pour le reste du traitement. Si un cœur est inutilisé, le gestionnaire de tâches peut alors se charger de soumettre la requête au réseau et d'attendre la terminaison de celle-ci. Si toutes les unités de calcul sont utilisées par l'application, l'exécution de la tâche est retardée jusqu'à ce que l'ordonnanceur donne la main au gestionnaire de tâches ou lorsque l'application attend la fin de la communication.

La décomposition des traitements de communication alliée au gestionnaire de tâches permet donc de paralléliser les opérations de communication. L'exploitation des multiples cœurs de la machine pour le traitement des communications est un moyen efficace de réduire le coût des communications et l'utilisation des cœurs inactifs permet de répartir la charge entre les processeurs. En effet, les cœurs occupés par l'application sont *"épaulés"* par les unités de calcul inutilisées et le traitement des communications peut être accéléré. Ce mécanisme nécessite toutefois que la quantité de calcul soit suffisante pour recouvrir la tâche. En effet, l'exportation d'une tâche sur un autre processeur a un coût du fait des communications entre les cœurs et de problèmes de cache. Ainsi, un traitement peut prendre plus de temps à s'exécuter sur un processeur distant que sur le processeur local. Il convient donc de calculer suffisament longtemps pour recouvrir complètement les communications.

5.4.3 Utilisation de plusieurs réseaux simultanément

Le mécanisme de parallélisation des traitements de communication offre de nouvelles possibilités pour réduire l'impact des communications sur les performances de l'application. Le recouvrement des communications par du calcul permet de cacher le surcoût des communication. Toutefois, l'utilisation de multi-threading permet également de réduire les temps de transfert. Nous proposons donc d'exploiter le mécanisme de tâches présenté dans la section 5.4.1 et le découpage des traitements de communication pour améliorer les temps de transfert par le réseau.

L'exploitation de plusieurs réseaux simultanément est un moyen efficace de réduire la durée de transfert d'un message [ABMN07]. Ce mécanisme qui consiste à couper un message en plusieurs segments envoyés chacun sur un réseau différent permet d'augmenter le débit pour les gros messages. Toutefois, cette technique est peu efficace pour les messages de taille réduite. En effet, la soumission de messages nécessitant une copie est coûteuse en temps processeur et, comme le montre la Figure 5.9, les soumissions aux réseaux sont séquentialisées. Le temps de transfert alors obtenu est de la forme :

$$T(longueur) = T_{Réseau1}\left(\frac{longueur}{2}\right) + T_{Réseau2}\left(\frac{longueur}{2}\right) + \varepsilon$$

Où $T_X(l)$ représente le temps de transfert d'un message de longueur l sur un réseau X et ε représente le sucoût lié au découpage du message. Les temps de transfert obtenus sont alors similaires à ceux obtenus en transférant toutes les données sur un seul réseau.

En utilisant le mécanisme de tâches et en parallélisant les traitements de communication, le temps de transfert d'un message nécessitant une recopie peut être réduit. Comme l'illustre la Figure 5.10, l'utilisation de tâches pour la soumission des segments de message permet d'éviter la séquentialisation des envois. Les segments sont alors envoyés simultanément depuis plusieurs cœurs. Le temps de transfert ainsi obtenu est alors de la forme :

$$T(longueur) = MAX\left(T_{Réseau1}\left(\frac{longueur}{2}\right), T_{Réseau2}\left(\frac{longueur}{2}\right)\right) + T_{Tâche}$$

Où $T_{Tâche}$ représente le temps nécessaire à la création d'une tâche et à son exécution. Le temps de transfert peut donc être réduit si $T_{Tâche}$ reste faible par rapport aux temps de transfert. Bien que cette assertion soit fausse pour les petits messages, elle peut être valable pour les messages de taille moyenne. Pour ce type de messages, l'utilisation de plusieurs réseaux simultanément peut réduire le coût des communications sur les performances de l'application.

Lorsque tous les cœurs de la machine sont occupés, ce mécanisme ne peut pas s'appliquer. Il convient alors de transmettre les données en n'utilisant qu'un seul réseau. De plus, ce mécanisme ne se limite pas à l'utilisation de deux réseaux simultanément, mais peut être appliqué à un nombre quelconque d'interfaces réseau. La seule contrainte est qu'il faut disposer d'autant de cœurs libres que de réseaux que l'on souhaite exploiter.

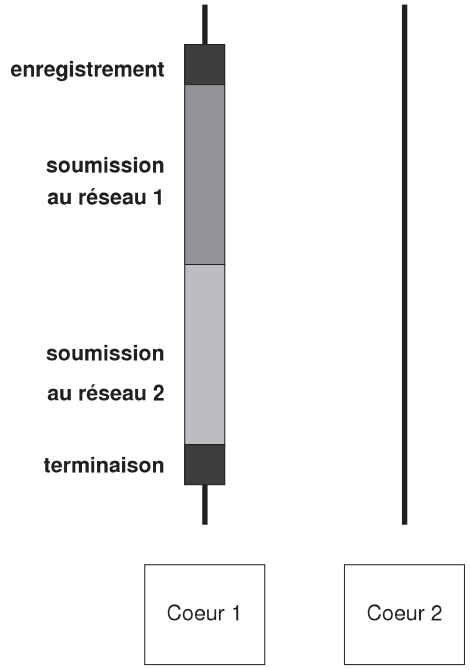

FIGURE 5.9 – *Envoi d'un message sur deux réseaux simultanément en n'utilisant qu'un cœur.*

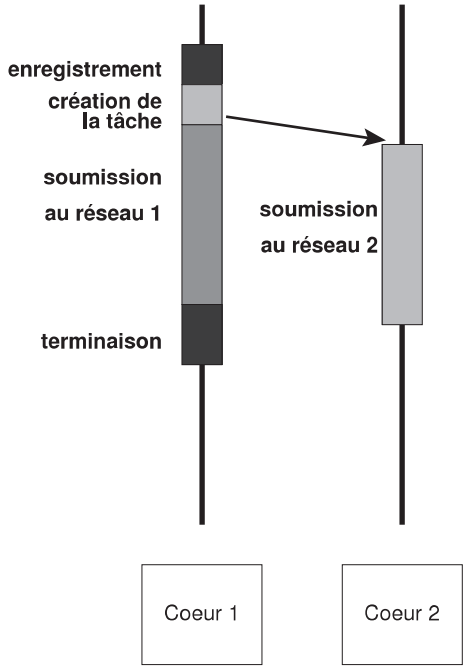

FIGURE 5.10 – *Envoi d'un message sur deux réseaux simultanément en utilisant deux cœurs.*

5.5 Bilan de la proposition

Nous avons présenté dans ce chapitre l'architecture d'une pile logicielle adaptée aux plates-formes matérielles modernes et aux modèles de programmation mélangeant communication et multi-threading. Cette architecture logicielle repose sur les interactions entre la bibliothèque de communication et l'ordonnanceur de threads. La gestion de ces interactions est laissée à un module logiciel dédié au multi-threading dans les communications. Nous avons montré les possibilités offertes par ce module de *détection des événements*, notamment comment les problèmes de réactivité aux communications et la progression des communication en arrière-plan pouvaient être gérés.

Nous avons également étudié le support des applications multi-threadées et proposé plusieurs mécanismes de protection adaptés aux différents types d'applications. Le support des applications multi-threadées ne se résume pas à des mécanismes de protection : nous avons analysé l'implémentation des primitives d'attente fournies par les bibliothèques de communication. À partir de cette étude, nous avons proposé un mécanisme d'*attente mixte* permettant de faire progresser les calculs de l'application tout en limitant les surcoûts liés aux changements de contexte.

Enfin, nous avons présenté un *gestionnaire de tâches* adapté aux bibliothèques de communication. Ce mécanisme offre la possibilité de traiter les tâches de communication en parallèle et d'exploiter les cœurs inutilisés. Nous avons montré comment ce gestionnaire de tâches peut être utilisé pour améliorer le recouvrement des communications par du calcul et pour réduire le coût des communications en exploiter plusieurs réseaux simultanément.

Le modèle faisant collaborer l'ordonnanceur de thread avec la bibliothèque de communication que nous

proposons est conçu pour gérer les différentes problématiques posées par les modèles de programmation modernes. Si les notions présentées ici peuvent être mises en œuvre de manière relativement simple, l'optimisation des performances nécessite la mise au point de quelques mécanismes que nous détaillons dans le chapitre suivant.

Chapitre 6

Élements d'implémentation : le gestionnaire d'événements PIOMAN et son utilisation dans NEWMADELEINE

Sommaire

Nous présentons dans ce chapitre l'implémentation des concepts proposés dans le chapitre précédent. La majorité du développement a été conduite dans la suite logicielle PM^2. Nous commençons donc par décrire brièvement cette suite logicielle et ses différents modules logiciels. Nous présentons ensuite PIOMAN, le gestionnaire d'entrées/sorties qui implémente le mécanisme de détection des événements et le système de tâches légères. La bibliothèque de communication NEWMADELEINE est ensuite détaillée, notamment la partie en charge du multi-threading. Enfin, nous présentons MPICH2/NEWMADELEINE, l'implémentation MPICH2 se basant sur NEWMADELEINE pour gérer les communications.

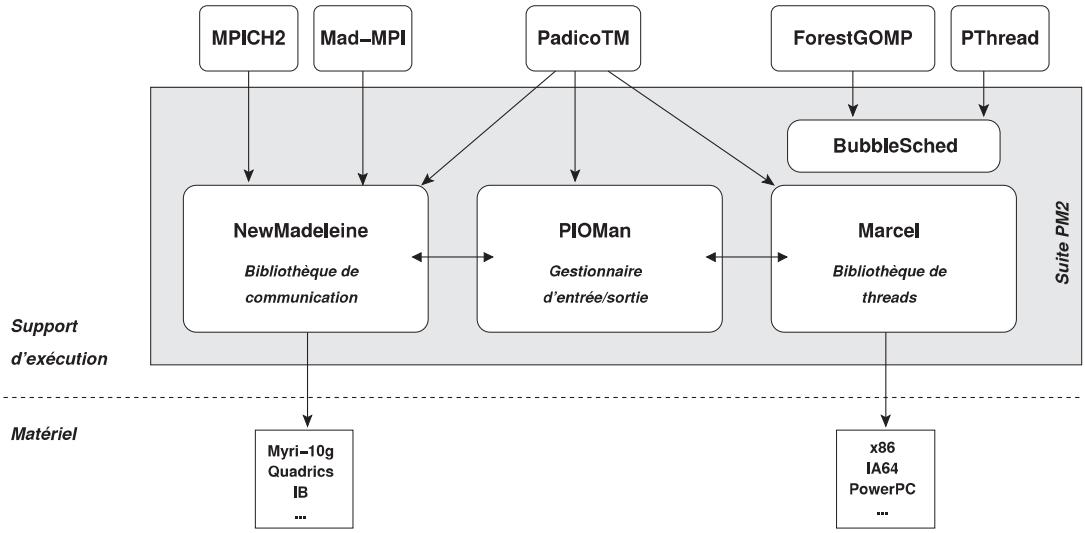

FIGURE 6.1 – *Suite logicielle PM*^{2}.

6.1 La suite logicielle PM^2

Une grande partie du développement réalisé au cours de cette thèse l'a été au sein de la suite logicielle PM^2 [NM95a]. L'environnement de programmation PM^2 a initialement été conçu au LIFL de Lille et s'est poursuivi à l'École normale supérieure de Lyon. Il s'agissait à l'origine d'un environnement de programmation distribué fondé sur le paradigme LRPC (*Lightweight Remote Procedure Call*) s'appuyant sur l'ordonnanceur de threads de niveau utilisateur MARCEL [NM95b] et la bibliothèque de communication MADELEINE [BNM98]. Si les notions sur lesquelles se base PM^2 n'ont pas changé, la paradigme LRPC est aujourd'hui remplacé par une approche basée sur le passage de messages. Pour cela, PM^2 s'appuie sur la bibliothèque de communication NEWMADELEINE [Bru08a] et sur la bibliothèque de threads de niveau utilisateur MARCEL [mar07]. Comme l'illustre la Figure 6.1, ces deux bibliothèques interagissent par le biais du gestionnaire d'entrées/sorties PIOMAN.

Le développement effectué au cours de cette thèse a principalement porté sur l'implémentation de PIOMAN et sur la partie de NEWMADELEINE gérant le multi-threading. Nous présentons dans cette section la bibliothèque de communication NEWMADELEINE et la bibliothèque de threads MARCEL.

6.1.1 La bibliothèque de communication NEWMADELEINE

NEWMADELEINE est une bibliothèque de communication pour réseaux hautes performances développée au cours de la thèse d'Élisabeth Brunet [Bru08b]. NEWMADELEINE est capable d'exploiter efficacement la plupart des technologies réseau modernes (MYRINET, INFINIBAND, QSNET, TCP, etc.) et supporte l'utilisation simultanée de plusieurs technologies réseau différentes (*multirail hétérogène*) [ABMN07].

Contrairement aux bibliothèques de communication classiques, NEWMADELEINE applique dynamiquement des stratégies d'ordonnancement et d'optimisation. Différentes stratégies d'optimisation (agrégation de messages, répartition des messages sur plusieurs réseaux, etc.) sont définies et peuvent agir en fonction de l'état des cartes réseaux. En effet, NEWMADELEINE base son activité sur celle des cartes réseaux : lorsque ces dernières sont occupées à transmettre des données, les messages déposés par

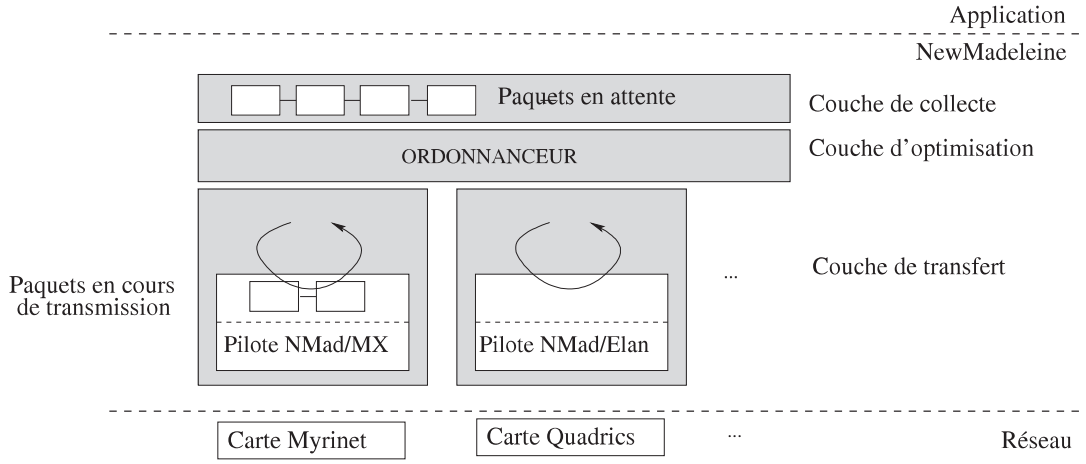

FIGURE 6.2 – *Architecture de la bibliothèque de communication* NEWMADELEINE.

l'application sont accumulés. Lorsque les cartes réseau sont inutilisées, NEWMADELEINE invoque une stratégie d'optimisation et génère un paquet à transmettre par le réseau. NEWMADELEINE profite donc des moments pendant lesquels les cartes réseau sont occupées pour accumuler des messages de l'application et ainsi avoir un large choix d'optimisations. Comme l'illustre la Figure 6.2, l'architecture de NEWMADELEINE comporte 3 couches :

La couche de collecte. Les messages que l'application soumet à la bibliothèque de communication sont accumulés dans la couche de collecte. Les informations permettant d'identifier les messages (numéro de séquence, numéro de tag, etc.) sont alors ajoutées aux informations concernant le message (adresse des données, taille du message, expéditeur ou destinataire, etc.) La requête ainsi formée est alors stockée dans une des listes de requêtes.

La couche d'optimisation. Lorsqu'une carte réseau devient libre, les différentes stratégies d'optimisation examinent les listes de requêtes de la couche de collecte et forment un paquet optimisé à soumettre au réseau. Les messages provenant de plusieurs flux de communication peuvent alors être combinés pour former un même paquet. Le paquet généré est une combinaison d'une ou plusieurs requêtes vers une même destination. La couche d'optimisation est également en charge de la gestion des protocoles nécessaires au bon fonctionnement des communications (création des messages de contrôle du *rendez-vous* par exemple).

La couche de transfert. Chaque technologie réseau supportée est exploitée grâce à un pilote spécifique dans la couche de transfert. Ce pilote fournit aux couches supérieures de NEWMADELEINE des fonctions permettant de soumettre une requête au réseau, de vérifier la terminaison d'une requête, etc. Les pilotes exposent également des informations sur les capacités des réseaux. Ces indications peuvent porter sur la nature des échanges (en flux, par message, etc.), sur la capacité du réseau à traiter des blocs de données épars en mémoire, etc.

NEWMADELEINE dispose également d'un mécanisme d'échantillonnage qui permet d'évaluer les performances réelles d'un réseau. À l'initialisation de l'application, les différents réseaux et leurs méthodes de transfert sont évalués et les performances obtenues sont stockées en mémoire. Ainsi, lorsqu'un paquet

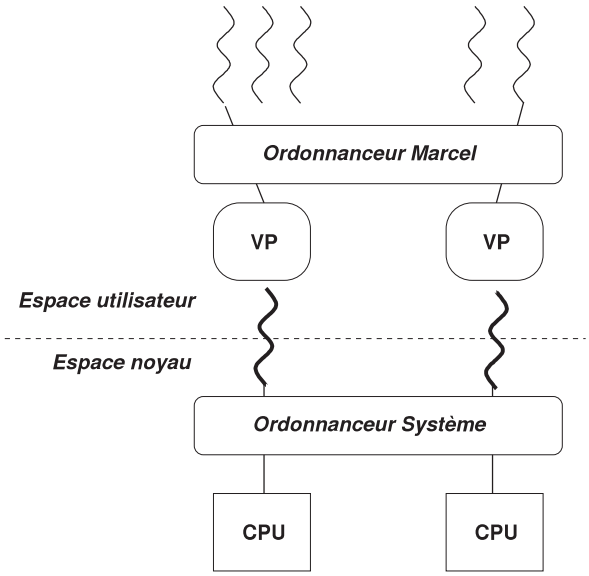

FIGURE 6.3 – *Ordonnanceur de threads de niveau utilisateur.*

de données doit être envoyé par un réseau, NEWMADELEINE peut consulter les performances mesurées et estimer le temps que prendra le transfert réseau. Les stratégies d'optimisation de NEWMADELEINE peuvent ainsi adapter leur comportement aux caractéristiques de la machine.

6.1.2 La bibliothèque de threads MARCEL

La bibliothèque de threads MARCEL [NM95b] a initialement été conçue par Raymond Namyst pour l'environnement de programmation distribué PM^2. MARCEL propose un mécanisme de threads de niveau utilisateur. Ces threads sont des entités propres à MARCEL et l'ordonnanceur de threads du système d'exploitation n'en a pas connaissance. MARCEL fait des appels à `setjmp` et `longjmp` pour gérer les changements de contexte entre les threads MARCEL, ce qui permet d'obtenir de très bonnes performances en comparaison avec le coût des changements de contexte entre des threads noyau. De plus, MARCEL fournit une interface unifiée pour exploiter des machines multi-processeurs variées en terme de systèmes (notamment GNU/Linux, BSD, AIX, Irix et OSF) ou d'architectures (notamment `x86`, `x86_64`, Itanium, PowerPC, Alpha, Mips ou Sparc).

Pour exploiter les machines multi-cœurs, MARCEL utilise le concept de *Processeur Virtuel* (*Virtual Processor* ou VP) illustré par la Figure 6.3. Pour chaque processeur disponible, MARCEL crée un thread de niveau noyau (*LightWeight Process* ou LWP) et le fixe sur un processeur. Ce thread noyau devient alors un *Processeur Virtuel* sur lequel peuvent s'exécuter plusieurs threads de niveau utilisateur. Ainsi, l'ordonnancement des threads est entièrement réalisé en espace utilisateur, ce qui permet de maîtriser totalement l'exécution des threads sans interaction avec le système d'exploitation.

Au cours de sa thèse, Samuel Thibault a proposé et implémenté l'interface de programmation BUBBLESCHED [Thi07]. La notion de *bulle* a permis d'exprimer la nature structurée du parallélisme : les threads sont regroupés dans des bulles en fonction de leurs affinités et une hiérarchie de threads est ainsi créée. L'ordonnanceur tente alors de placer les threads d'une bulle sur des processeurs proches afin de bénéficier d'effets de cache ou de pouvoir placer les données sur lesquelles travaillent les threads

sur un nœud NUMA proche. L'interface BUBBLESCHED propose également des outils permettant de développer des ordonnanceurs dédiés, efficaces et portables.

Le support OPENMP du compilateur GCC, GOMP, a été étendu pour utiliser BUBBLESCHED. Cette extension, nommée FORESTGOMP [BDT+08] et développée au cours de la thèse de François Broquedis, permet d'exprimer la nature hiérarchique des sections parallèles imbriquées grâce aux bulles. MARCEL propose également une couche de compatibilité POSIX afin de bénéficier des performances des ordonnanceurs à bulles pour une large gamme d'applications.

6.2 Le gestionnaire d'entrées/sorties PIOMAN

Alors que les éléments logiciels présentés précédemment préexistaient, le gestionnaire d'entrées/sorties dont nous détaillons ici les points-clés de l'implémentation a été entièrement développé au cours de nos travaux. Afin de gérer les interactions entre NEWMADELEINE et MARCEL, nous avons implémenté le gestionnaire d'entrées/sorties PIOMAN qui prend en charge les problèmes de multi-threading pour la bibliothèque de communication. Ainsi, NEWMADELEINE peut concentrer ses efforts sur la gestion et l'optimisation des requêtes réseau et le code n'est pas complexifié par une gestion de la progression des communications ou par la parallélisation des traitements. Nous présentons ici quelques points-clés du fonctionnement interne de PIOMAN.

6.2.1 Collaboration avec MARCEL

Comme nous l'avons exposé dans la section 5.2.2, les interactions entre la bibliothèque de communication et l'ordonnanceur de threads sont gérées par PIOMAN. Le développement de NEWMADELEINE est ainsi simplifié puisque les principaux problèmes liés au multi-threading sont gérés dans un module séparé. Le développement de PIOMAN nécessite toutefois de travailler étroitement avec l'ordonnanceur de threads afin d'assurer une forte réactivité aux événements provenant du réseau. Pour cela, MARCEL donne la main fréquemment à PIOMAN, de telle sorte que les tâches (détection des événements ou tâche soumise par la bibliothèque de communication) puissent être exécutées. Ainsi, des appels à PIOMAN sont insérés à certains points-clés de l'ordonnancement : dans la boucle *idle* – afin que PIOMAN puisse exploiter les processeurs inutilisés – lors des changements de contexte et dans le code traitant les signaux d'horloge. La fréquence de scrutation est alors garantie : dans le pire des cas, PIOMAN est ordonnancé à chaque signal d'horloge.

Afin d'améliorer la réactivité aux événements réseau, PIOMAN utilise les fonctions de rappel bloquantes fournies par NEWMADELEINE. Ce mécanisme, bien que généralement efficace, est problématique lorsque l'on utilise des threads de niveau utilisateur. En effet, comme l'illustre la Figure 6.4, lorsqu'un tel thread exécute un appel système bloquant, le système d'exploitation bloque le thread noyau tout entier. Le *processeur virtuel* associé est alors bloqué et les threads de niveau utilisateur s'exécutant sur ce VP ne peuvent plus s'exécuter tant que le système ne réordonnance pas le thread noyau. Un appel système bloquant risque donc de ralentir considérablement l'application en la privant d'un des processeurs. Mais un appel bloquant peut avoir des conséquences beaucoup plus graves puisqu'il peut entraîner une situation d'inter-blocage. En effet, l'événement attendu par le thread qui se bloque (la réception d'un message par le réseau par exemple) peut nécessiter l'exécution d'un autre thread. On peut penser au cas où un thread T_1 envoie un message dont la réponse est envoyée au thread T_2 s'exécutant sur le même processeur

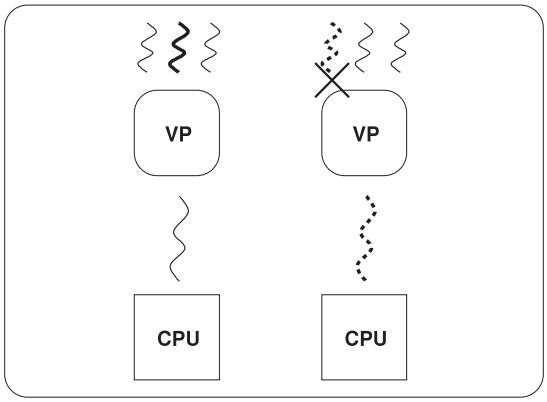

FIGURE 6.4 – *Déroulement d'un appel bloquant sur un thread de niveau utilisateur.*

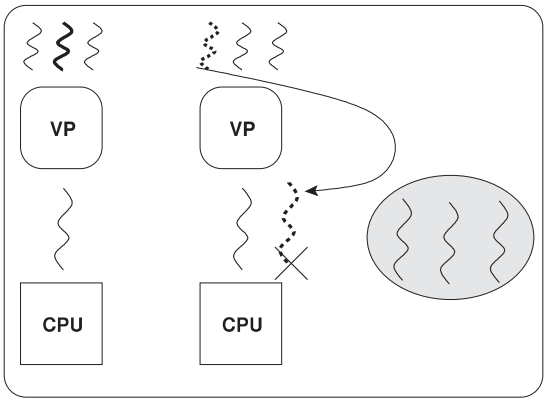

FIGURE 6.5 – *Exportation d'un appel bloquant sur un système de threads de niveau utilisateur.*

virtuel. Dans ce cas, si T_2 effectue un appel bloquant pour attendre la réponse, T_1 risque de ne pas pouvoir envoyer le message. L'utilisation d'appels système bloquants est donc à proscrire lorsque plusieurs threads de niveau utilisateur se partagent un processeur virtuel. L'implémentation de mécanismes – tels que les SCHEDULER ACTIVATIONS – faisant collaborer l'ordonnanceur de threads de niveau utilisateur avec l'ordonnanceur du système est une solution à ce type de problèmes. Toutefois, l'implémentation de cette solution est complexe et les performances obtenues sont fortement pénalisées par le surcoût de ce mécanisme [DNR00b].

Les problèmes d'appels bloquants dans un système de threads de niveau utilisateur peuvent être résolus en appliquant un mécanisme similaire à celui décrit dans la section 5.2.3. Le mécanisme utilisé par PIOMAN est décrit par la Figure 6.5 : afin de ne pas bloquer de processeur virtuel, les appels bloquants sont exécutés sur un thread noyau supplémentaire. Pour cela, PIOMAN crée une réserve de threads noyau (LWP). Ces LWP ne sont pas utilisés par MARCEL pour créer des processeurs virtuels et ils passent leur temps à attendre une commande. Ainsi, lorsqu'un thread doit effectuer un appel bloquant, un des LWP supplémentaires est réveillé et la requête à traiter lui est transmise. Le LWP peut alors exécuter la fonction de rappel bloquante et le thread de niveau utilisateur ne bloque pas le processeur virtuel. Les autres threads de niveau noyau peuvent alors être ordonnancés par MARCEL. Lorsque l'événement attendu survient, le LWP supplémentaire est ordonnancé par le système d'exploitation, l'événement peut être traité et le thread utilisateur est réveillé. Le LWP se rendort ensuite et rejoint la réserve de LWP supplémentaires.

L'utilisation de LWP supplémentaires dans PIOMAN a un effet intéressant sur la réactivité aux événements provenant du réseau. En effet, les mécanismes de priorité des threads noyaux sont ici avantageux. Comme nous l'avons vu dans la section 5.2.3, il n'est pas toujours possible de choisir une politique d'ordonnancement permettant de donner la main au LWP supplémentaire immédiatement après l'occurrence de l'événement. On peut alors donner une priorité élevée au thread afin que l'ordonnanceur du système ait tendance à l'ordonnancer rapidement. Lorsque l'application utilise une bibliothèque de thread de niveau noyau (la bibliothèque NPTL par exemple), les threads de l'application sont au même niveau que les threads chargés des appels bloquants. Ainsi, si l'application exécute N threads, le système d'exploitation doit choisir d'ordonnancer un thread parmi les $N+1$ threads disponibles (N threads de l'application et un thread chargé des appels bloquants). Si l'application lance un grand nombre de threads, le thread de communication risque de ne pas être ordonnancé rapidement, malgré

```
/* Initialise une requête */
int piom_req_init(piom_req_t req);

/* Soumet une requête à PIOMan.
 * À partir de maintenant, la requête peut être détectée à n'importe
 * quel moment */
int piom_req_submit(piom_server_t server, piom_req_t req);

/* Annule une requête.
 * Les threads en attente de la requête sont réveillés et ret_code est
 * retourné */
int piom_req_cancel(piom_req_t req, int ret_code);

/* Attend la fin d'une requête */
int piom_req_wait(piom_req_t req, piom_wait_t wait,
                  piom_time_t timeout);

/* Teste la terminaison d'une requête */
int piom_test(piom_req_t req);
```

FIGURE 6.6 – *Interface de détection des événements de* PIOMAN.

sa haute priorité. Avec un ordonnanceur de threads de niveau utilisateur comme MARCEL, la situation est différente : même si l'application lance N threads, le système n'en voit que M (le nombre de processeurs). L'ordonnanceur du système choisit alors de donner la main à un thread parmi $M+1$ (M threads noyau et un LWP chargé des appels bloquants). Ainsi, même si l'application utilise un grand nombre de threads (de niveau utilisateur), la probabilité que le thread chargé de la détection des événements soit ordonnancé reste constante.

Puisque PIOMAN peut gérer les méthodes de détection par scrutation et par appel bloquant, il convient d'utiliser la méthode la plus adaptée au contexte. PIOMAN choisit le mode de détection qui minimise le temps de réaction. La méthode de détection à choisir dépend fortement de la charge de la machine : lorsque des processeurs sont inutilisés, la scrutation permet de détecter rapidement un événement. À l'inverse, si tous les processeurs sont occupés, la fonction de rappel bloquante est généralement plus adaptée. La bibliothèque de communication peut bien sûr donner des indications à PIOMAN lors de la soumission d'une requête : NEWMADELEINE peut spécifier qu'une requête doit être traitée par une méthode bloquante uniquement ou par une méthode de scrutation. Ainsi, lorsqu'une requête doit être traitée, PIOMAN applique les spécifications de NEWMADELEINE. Si aucune indication n'est donnée, PIOMAN évalue la charge de la machine en recherchant des processeurs inactifs. Dans le cas où MARCEL ne signale aucun processeur inactif, un LWP supplémentaire est réveillé afin d'exécuter la fonction de rappel bloquante. Si certains processeurs sont inactifs, la requête est alors placée dans une liste de requêtes à traiter afin que ces processeurs puissent appeler la fonction de rappel associée.

6.2.2 Interface de détection des événements

L'interface standard de détection des événements de PIOMAN, présentée dans la figure 6.6, est directement héritée de l'interface du serveur d'événements implémenté dans MARCEL. La bibliothèque de communication commence par initialiser une requête et spécifie ses différentes options (type de callback

```
/* Initialise un sémaphore */
void piom_sem_init(piom_sem_t *sem, int initial);
/* Verrouille un sémaphore */
void piom_sem_P(piom_sem_t *sem);
/* Déverrouille un sémaphore */
void piom_sem_V(piom_sem_t *sem);

/* Initialise une condition */
void piom_cond_init(piom_cond_t *cond, uint8_t initial);
/* Met à jour le masque d'une condition et réveille les threads en
 * attente de cette condition */
void piom_cond_signal(piom_cond_t *cond, uint8_t mask);
/* Attend qu'une condition soit vérifiée */
void piom_cond_wait(piom_cond_t *cond, uint8_t mask);
/* Teste l'état d'une condition */
int  piom_cond_test(piom_cond_t *cond, uint8_t mask);
```

FIGURE 6.7 – *Interface de détection des événements par condition de* PIOMAN.

à utiliser, priorité de la requête, etc.) La requête est ensuite soumise à PIOMAN qui peut alors appeler les fonctions de rappel spécifiées. Enfin, la bibliothèque de communication peut se bloquer en attendant la fin d'une requête, ou utiliser une primitive non-bloquante. Ainsi, les appels aux primitives de communication depuis l'application peuvent être redirigés vers l'interface de PIOMAN. Par exemple, lorsque l'application fait appel à une primitive de réception non-bloquante (`MPI_Irecv`), la bibliothèque de communication soumet la requête au réseau et délègue la détection de la terminaison de la requête à PIOMAN.

Toutefois, cette interface nécessite que les requêtes réseau correspondent aux requêtes de l'application. Si la bibliothèque de communication effectue des traitements sur les requêtes de l'application et génère des requêtes réseau différentes, cette interface est peu adaptée. Par exemple, les stratégies d'ordonnancement de NEWMADELEINE effectuent des optimisations sur les messages à transmettre et une requête de l'application peut correspondre à plusieurs requêtes réseau (notamment lorsqu'un message est coupé et transmis sur plusieurs réseaux simultanément). Plusieurs requêtes de l'application peuvent également être agrégées et transmises en une seule fois par le réseau. Pour gérer ce découplement entre les requêtes de l'application et celles du réseau rends complexe l'utilisation d'une interface faisant correspondre un événement réseau à une requête de l'application. Nous proposons donc d'utiliser l'interface décrite par la figure 6.7.

Cette interface est basée sur des mécanismes de sémaphores et de conditions. Ainsi, après avoir initialisé un sémaphore ou une condition, la bibliothèque de communication soumet une requête à PIOMAN (en utilisant `piom_req_submit`) et l'application attend directement la fin de l'envoi ou de la réception d'un message. NEWMADELEINE se charge alors des optimisations sur les messages et génère des requêtes réseau. Lorsque toutes les requêtes réseau d'un message sont terminées, NEWMADELEINE réveille les threads en attente. Le découplage de la couche de transfert et de la couche de collecte de NEWMADELEINE devient ainsi naturel du point de vue des fonctions d'attente.

```
typedef enum
{
  PIOM_LTASK_OPTION_NULL = 0,
  PIOM_LTASK_OPTION_REPEAT = 1
} piom_ltask_option;

/* Crée une tâche et l'initialise */
void piom_ltask_create (piom_ltask *task,
                        piom_ltask_func * func_ptr,
                        void* func_args,
                        piom_ltask_option options,
                        piom_ltask_cpuset mask);

/* Soumet une tâche au gestionnaire de tâches.
 * À partir de maintenant, la tâche peut être exécutée à n'importe quel moment */
void piom_ltask_submit (piom_ltask *task);

/* Attend la terminaison d'une tâche */
void piom_ltask_wait (piom_ltask *task);

/* Teste la terminaison d'une tâche */
int piom_ltask_test (piom_ltask *task);

/* Marque la tâche comme étant terminée */
void piom_ltask_set_completed(piom_ltask *task);
```

FIGURE 6.8 – *Interface de programmation du gestionnaire de tâches.*

6.2.3 Mécanisme d'exportation de tâches

Le gestionnaire de tâches décrit dans la section 5.4.1 est implémenté dans PIOMAN et bénéficie donc de son étroite collaboration avec MARCEL. Outre la possibilité de traiter des tâches lors de certains points-clés de l'ordonnancement –lorsqu'un processeur est inutilisé ou lors d'un signal d'horloge notamment–, le gestionnaire de tâche bénéficie de la grande connaissance qu'a MARCEL de la topologie de la machine. Ainsi, des listes de tâches sont créées à chaque niveau de topologie.

L'interface du gestionnaire de tâches est décrite dans la Figure 6.8. La bibliothèque de communication qui utilise ce mécanisme définit des *tâches* par une fonction de rappel et l'argument à passer à cette fonction. Une fois la tâche définie et soumise au gestionnaire, ce dernier se charge d'exécuter la fonction de rappel spécifiée sur l'un des cœurs de la machine. La bibliothèque de communication peut alors tester ou attendre la terminaison d'une tâche. Lors de la création d'une tâche, NEWMADELEINE peut spécifier l'option `LTASK_OPTION_REPEAT` pour signifier que la tâche est répétitive. Le gestionnaire de tâches exécutera alors la tâche tant que celle-ci ne sera pas marquée comme étant terminée. Une tâche soumise par la bibliothèque de communication comporte également un masque de processeurs sur lesquels peut s'exécuter la fonction. Ce mécanisme permet de prendre en compte la localité des données. Ainsi, un thread qui soumet une tâche peut spécifier un processeur (ou un groupe de processeurs) proche du cœur sur lequel il s'exécute pour bénéficier des effets de cache lors du traitement des données par la tâche.

6.3 NEWMADELEINE : une bibliothèque de communication multi-threadée

Si l'utilisation de PIOMAN pour la progression des communications de NEWMADELEINE a nécessité très peu de développement, la parallélisation de NEWMADELEINE est une tâche plus complexe. Ainsi, une grande partie du développement dans NEWMADELEINE a porté sur la gestion du multi-threading, que ce soit le support des applications utilisant des threads ou l'utilisation de threads à l'intérieur de la bibliothèque de communication pour paralléliser le traitement des communication. Nous présentons ici l'implémentation des principaux mécanismes traitant du multi-threading dans NEWMADELEINE.

6.3.1 Progression des communications dans NEWMADELEINE

Comme nous l'avons décrit dans la section 6.1.1, NEWMADELEINE adopte une architecture en 3 couches. La progression des communication est assurée par la couche d'optimisation : comme le montre la figure 6.9, lorsque l'application dépose un message à la couche de collecte, cette dernière appelle la couche d'optimisation et les informations (cible de l'échange, description des données, etc.) sont collectées dans une structure de données nommée *packet wrapper* ((1) sur la figure). En fonction de la stratégie d'optimisation, ces informations peuvent être ajoutées à un *packet wrapper* préexistant (lorsque plusieurs messages sont agrégés notamment) ou elles peuvent être insérées dans un nouvel objet. Le *packet wrapper* est ensuite stocké dans une liste en attendant qu'une carte réseau se libère.

Pour détecter la libération d'une carte réseau, NEWMADELEINE utilise le mécanisme de tâches fourni par PIOMAN. Une tâche répétitive est lancée sur les différents cœurs de la machine. Ainsi, les cœurs inactifs passent leur temps à rechercher des cartes réseau libres. Lorsque l'une d'elles est détectée, la stratégie d'optimisation est invoquée afin de fournir à la couche de transfert un paquet de données à transmettre ((2) sur la figure). Lors de la soumission au réseau, la terminaison de la requête est vérifiée. Si la requête n'est pas terminée, la détection de l'événement associé est alors déléguée à PIOMAN qui

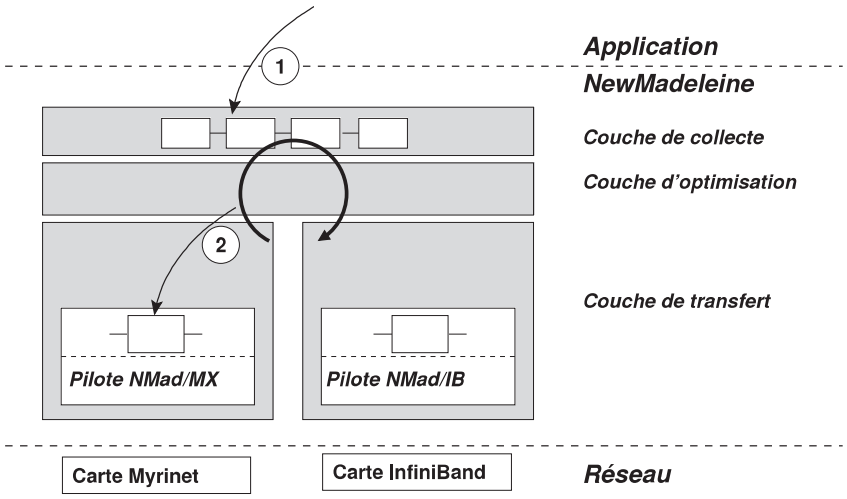

FIGURE 6.9 – *Cheminement d'une requête de l'application jusqu'au réseau.*

exécute les fonctions de rappel fournies sur les différents processeurs disponibles. Lorsque la fonction de rappel détecte la terminaison de la requête, une fonction de la couche de collecte est appelée afin de notifier à l'application la fin d'un transfert réseau. Cette fonction consulte la liste des requêtes de l'application en attente et détermine lesquelles sont terminées. Les conditions associées à ces requêtes sont alors réveillées.

Le découplage de la couche de collecte et des autres couches de NEWMADELEINE peut donc être géré par l'utilisation conjointe de fonctions de rappel et de conditions. Les mécanismes implémentés dans NEWMADELEINE sont semblables à ceux proposés dans ACTIVE MESSAGES [ECGS92] et permettent d'adopter une approche événementielle des communications.

6.3.2 Gestion des accès concurrents

Afin que NEWMADELEINE supporte les accès concurrents des applications multi-threadées, nous avons implémenté deux mécanismes de protection différents, chacun ayant une granularité particulière. Ainsi, les applications employant intensivement les appels concurrents à NEWMADELEINE peuvent bénéficier de la granularité du mécanisme de verrouillage à grains fins, les autres applications pouvant se tourner vers une solution ayant un impact négligeable sur les performances brutes du réseau.

Verrouillage à gros grains Du fait du découplage de la couche de collecte et de la couche de transfert dans NEWMADELEINE, le mécanisme de verrouillage à gros grains implique que le verrou global est pris plusieurs fois sur le chemin critique. En effet, comme le montre la figure 6.10, le verrou est pris lors la soumission de la requête par l'application et lorsque PIOMAN tente de détecter un événement réseau ou lors de la soumission d'un message à une interface réseau. Au final, entre le dépôt du message par l'application et sa soumission au réseau, le verrou est pris deux fois.

Verrouillage à grains fins Pour réduire la taille des sections critiques, le verrouillage à grains fins dans NEWMADELEINE est essentiellement constitué de verrous protégeant les différentes listes. Ainsi, la liste des requêtes déposées par l'application et les listes (une par interface réseau) de requêtes de

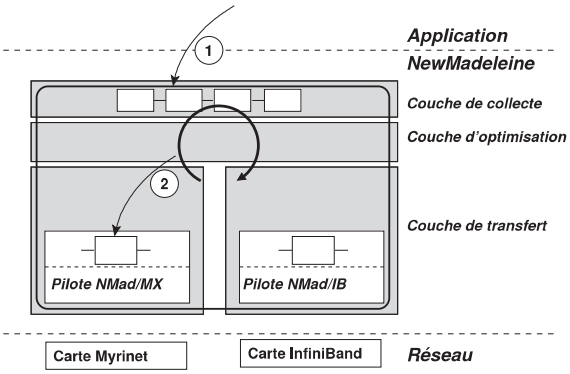

FIGURE 6.10 – *Portée du verrou global de* NEWMADELEINE.

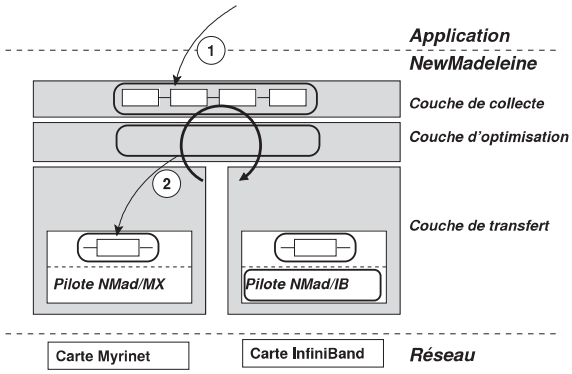

FIGURE 6.11 – *Portées des verrous à grain fin dans* NEWMADELEINE.

la couche de transfert sont toutes protégées par des verrous différents. Les algorithmes de parcours de liste utilisés dans NEWMADELEINE ont été optimisés afin de réduire la durée pendant laquelle le verrou protégeant une liste est pris. Les verrous sont ainsi pris uniquement le temps d'ajouter ou d'enlever un élément de la liste, réduisant la taille de la section critique et la contention sur les verrous. Certains pilotes réseau permettant d'exploiter des technologies réseaux sont également protégés afin d'éviter les accès concurrents lorsque l'interface de communication de bas-niveau ne le permet pas. Enfin, la couche d'optimisation est également protégée afin d'éviter que plusieurs threads ne tentent d'ordonnancer les messages de manière concurrente.

Afin d'éviter une complexification du code source et pour gérer simplement les différentes configurations (environnement mono-threadé, utilisation de threads MARCEL ou utilisation de PThreads), NEWMADELEINE utilise des primitives de verrouillage génériques. En fonction de l'environnement, ces primitives sont définies dans PIOMAN par des primitives de synchronisation MARCEL, des primitives PThread ou par des actions nulles. De plus, comme les verrous sont généralement pris pour une période très courte (généralement moins de quelques centaines de nanosecondes), utiliser des primitives d'exclusion mutuelle classique (*mutex*) risque de se révéler pénalisant. En effet, quand un thread tente de prendre un verrou et échoue, il y a de grande chance pour que ce verrou soit relâché très prochainement, le changement de contexte induit par un mutex serait alors très coûteux. Les verrous sont donc implémentés sous forme de verrous rotatifs (*spinlocks*). Ainsi, lorsqu'un thread tente de prendre un verrou qui est tenu par un autre thread, il retentera sa chance jusqu'à l'obtention du verrou – ce qui arrive généralement rapidement – sans changement de contexte coûteux.

6.3.3 Traitement des communications en parallèle

La parallélisation des traitements est une partie importante du travail effectué dans NEWMADELEINE. Le système de tâches développé dans PIOMAN est ici un outil essentiel pour paralléliser simplement les traitements de communication. Une première étape dans cette parallélisation consiste à découper les traitements. Par exemple, l'envoi d'un message par le réseau est composé de 3 étapes :

1. l'**enregistrement** de la requête soumise par l'application à la couche de collecte. Cette étape est relativement rapide (généralement moins de quelques centaines de nanosecondes).
2. la **soumission** de la requête au réseau. Cette étape nécessite de copier les données jusqu'à la carte réseau, ce qui peut prendre beaucoup de temps (jusqu'à plusieurs dizaines de microsecondes).

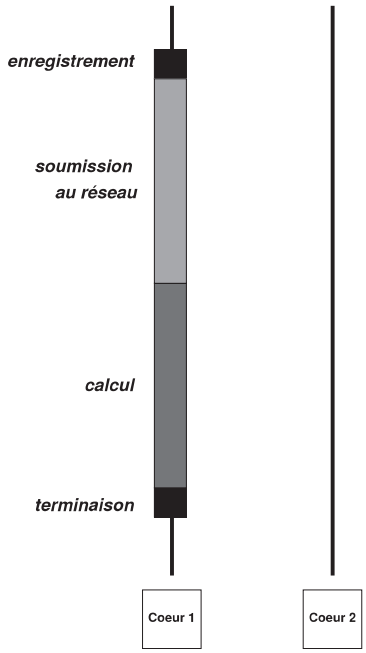

FIGURE 6.12 – *Envoi séquentiel d'un message sur le réseau.*

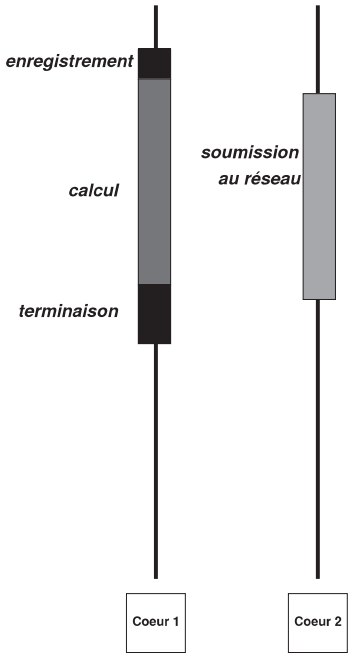

FIGURE 6.13 – *Envoi d'un message sur le réseau en utilisant un cœur inactif.*

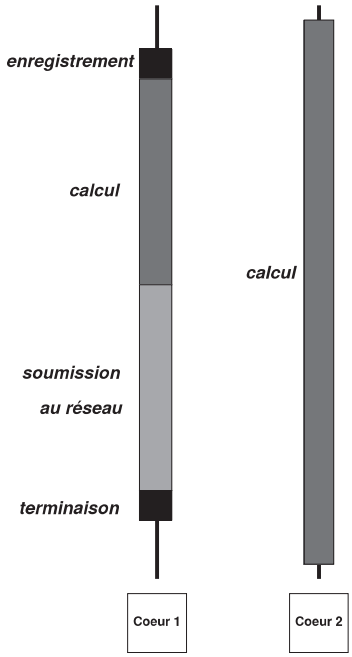

FIGURE 6.14 – *Envoi retardé d'un message sur le réseau quand tous les cœurs sont utilisés.*

3. la **détection** de la fin du transfert réseau.

Ces différentes opérations peuvent être exécutées sur plusieurs processeurs, la seule contrainte étant de les exécuter l'une après l'autre. NEWMADELEINE utilise donc le mécanisme de tâches fourni par PIOMAN pour répartir ces traitements sur les processeurs libres. Ainsi, lorsque l'application dépose un message à transmettre, celui-ci est enregistré – il serait contre-productif de déléguer à un autre processeur un enregistrement qui ne nécessite que quelques instants – et sa soumission est déléguée grâce au mécanisme de tâches. Ainsi, lorsqu'un processeur est inactif, il peut traiter la tâche et donc soumettre la requête au réseau. La détection de la terminaison de la requête est ensuite déléguée à PIOMAN. Ainsi, plutot que de traiter la communication de manière séquentielle – comme l'illustre la figure 6.12 – la soumission du message au réseau peut être faite depuis un processeur inactif, comme le montre la figure 6.13. Si aucun processeur n'est disponible, la tâche est exécutée au moment où l'application appelle la primitive d'attente (voir figure 6.14). Ce découpage des traitements des communications permet donc d'exploiter les cœurs inutilisés de la machine pour transférer les données en arrière-plan. L'application peut alors voir ses communications être recouvertes par du calcul.

Le découpage des traitements est également exploité pour paralléliser les opérations de communication de NEWMADELEINE. En alliant la stratégie d'optimisation qui découpe les messages pour les transférer sur plusieurs réseaux simultanément et le mécanisme de tâches fourni par PIOMAN, NEWMADELEINE peut paralléliser l'envoi de messages. En effet, lorsqu'une carte réseau se libère, une stratégie d'optimisation est invoquée afin de générer une nouvelle requête réseau. NEWMADELEINE détermine alors le nombre de cœurs inutilisés. Comme l'illustre la figure 6.15, en se basant sur le mécanisme d'échantillonnage, NEWMADELEINE calcule l'optimisation la plus adaptée :

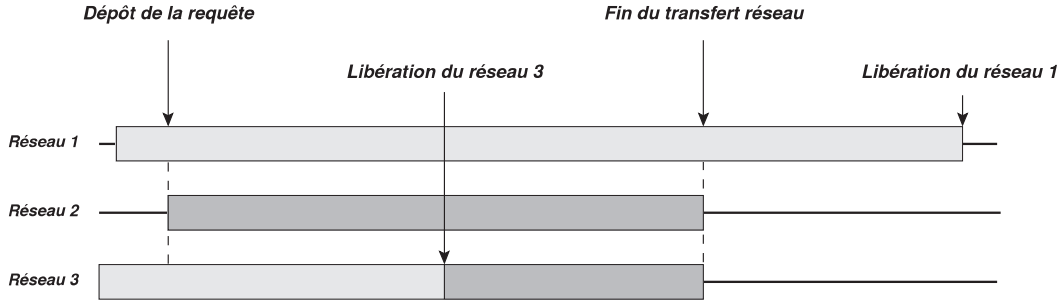

FIGURE 6.15 – *Répartition des données sur les réseaux en utilisant le mécanisme de prédiction de* NEWMADELEINE.

– Si plusieurs cœurs sont disponibles, NEWMADELEINE tente de découper le message en prenant en compte toutes les cartes réseau. NEWMADELEINE essaie alors de découper le message de façon à minimiser le temps de transfert. En prenant en compte les capacités des réseaux sous-jacents, NEWMADELEINE estime la durée d'un transfert sur chacune de ces interfaces réseau. Si un réseau est actuellement occupé, la date de libération de la ressource est ajoutée à la durée du transfert sur ce réseau. Au final, NEWMADELEINE répartit les données sur les différentes cartes réseau de telle sorte que tous les transferts réseau se terminent en même temps. Pour chaque réseau sélectionné, une tâche PIOMAN est créée et le masque de processeurs associé est fixé à l'un des cœurs inactifs.
– Si un seul cœur est disponible, NEWMADELEINE estime le temps de transfert du message sur chaque réseau. Là encore, une pénalité est ajoutée aux cartes réseau occupées. Une requête est ensuite créée afin d'envoyer le message sur le réseau sélectionné. La tâche PIOMAN dédiée à ce transfert se voit fixée sur le cœur inactif.
– Si aucun cœur n'est disponible, une tâche est créée pour soumettre le message au réseau. Cette tâche est exécutable sur n'importe quel processeur. Le réseau sélectionné est, là encore, choisi de manière à minimiser le temps de transfert.

Ainsi, lorsque les ressources sont disponibles (processeurs ou cartes réseau inactifs), le traitement des communications est parallélisé. Si les ressources sont limitées, NEWMADELEINE se contente de traiter les communications en arrière-plan lorsque cela est possible.

6.4 MPICH2/NEWMADELEINE

Depuis 2008, l'équipe associée entre l'équipe Runtime et le Radix Lab (Argonne National Laboratory, États-Unis), nous a conduit à l'intégration de NEWMADELEINE et PIOMAN dans MPICH2. Nous présentons ici les grandes lignes de cette intégration avant de détailler les mécanismes de progression des communications implémentés.

6.4.1 Architecture générale de MPICH2-NEMESIS

MPICH2 [mpi07] est une implémentation libre du standard MPI-2 parmi les plus populaires. Comme nous l'avons vu dans la section 3.2.1, l'architecture de MPICH2 s'organise en différentes couches. Plusieurs possibilités sont offertes lors de l'ajout d'un nouveau type d'interface réseau. L'implémentation d'un nouveau *device* ADI3 (*Abstract Device Interface*) mène généralement aux meilleures performances

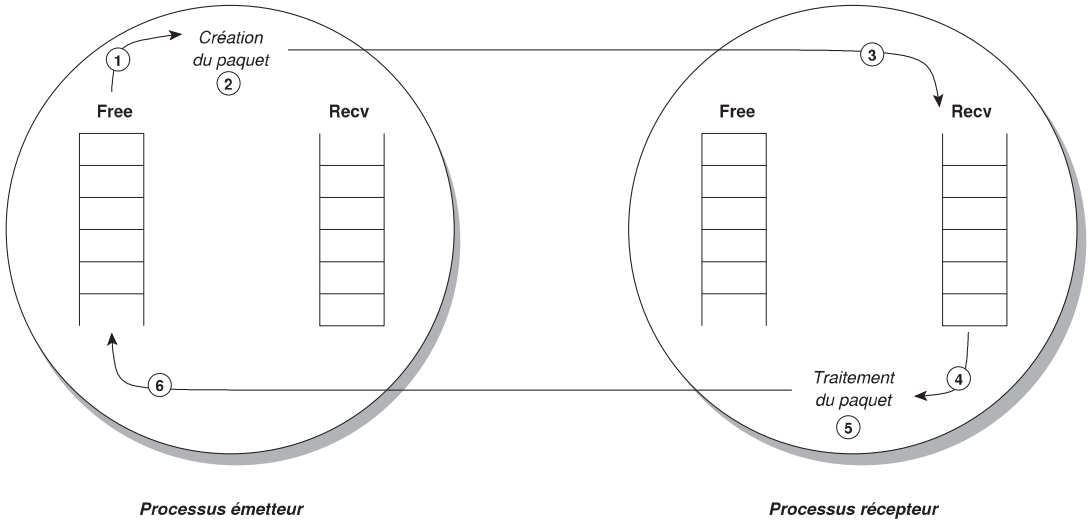

FIGURE 6.16 – *Files d'éléments dans* NEMESIS.

puisque les appels de l'application à MPI sont quasiment immédiatement transformés en appels à l'interface de communication. Le coût de développement d'un pilote réseau est toutefois important du fait du nombre de routines à implémenter. Cette solution est généralement réservée aux interfaces de communication élaborées dont le comportement est proche de celui de MPI. Ainsi, l'interface MPICH2-MX [Myr] (l'implémentation MPICH2 se basant sur le pilote MX de MYRICOM) ou le module permettant d'exploiter les réseaux QSNET avec MPICH2 [Qua03] sont implémentés sous la forme de module ADI3.

Une autre approche consiste à développer un nouveau *channel* CH3. Il s'agit ici d'implémenter un ensemble restreint de routines (une douzaine au total). Ainsi, CH3 compte plusieurs *channels* comme SHM (*shared memory channel*), SSM (*socket and shared memory channel*), SSHM (*scalable shared memory channel*) et NEMESIS qui se focalisent sur les communications en mémoire partagée. NEMESIS ne se limite pas aux communications intra-nœuds et propose un support aux communications inter-nœud grâce à un système de *modules réseau*. NEMESIS est actuellement le *channel* le plus performant en mémoire partagée parmi tous les *channels* de MPICH2, mais également parmi les autres implémentations MPI [BMG07]. Ces performances ont promu NEMESIS *channel* officiel de MPICH2. Les communications en mémoire partagée sont gérées par NEMESIS grâce à un système de files pouvant être modifiées de manière concurrente sans verrouillage. Chaque processus possède une file de réception (*Receive Queue*) et une ou plusieurs files d'éléments libres (*Free Queues*) allouées dans un segment de mémoire partagé avec les autres processus. La figure 6.16 montre le fonctionnement des communications intra-nœuds : l'émetteur (1) retire un élément de la *Free Queue* et y inclut le message à transmettre(2). Cet élément est ensuite ajouté à la file de réception du processus récepteur(3). Lorsque le récepteur détecte le message, il retire l'élément de sa file de réception (4) et traite le message (5). Finalement, l'élément est ajouté à la *Free Queue* de l'émetteur.

Afin d'améliorer encore les performances en mémoire partagée, NEMESIS utilise également un mécanisme de boîte aux lettres basé sur des *fastbox*. Une *fastbox* est un tampon mémoire auquel est ajouté un *marqueur* permettant de savoir si la *fastbox* est vide ou pleine. À l'initialisation, NEMESIS crée une *fastbox* par couple de processus connectés. Ainsi, si la *fastbox* du destinataire est vide, un processus émetteur peut y copier un message et modifier le *marqueur*. Le récepteur vérifie d'abord la *fastbox* avant d'examiner sa file de réception. Au final, ce mécanisme permet d'améliorer significativement les

performances des communications en mémoire partagée.

6.4.2 Intégration de NEWMADELEINE dans MPICH2

MPICH2 permet d'obtenir d'excellentes performances pour les communications en mémoire partagée. Les communications inter-nœuds sont également efficaces et restent proches des performances brutes du réseau. De plus, MPICH2 implémente les communications collectives de manière efficace. Toutefois, les optimisations sur les flux de communication ou l'utilisation de plusieurs réseaux simultanément n'est pas possible dans MPICH2. NEWMADELEINE, de son côté, sait appliquer des stratégies d'optimisation à des flux de communication et peut exploiter plusieurs réseaux – potentiellement différents – simultanément de manière efficace. Grâce à PIOMAN, NEWMADELEINE peut traiter les communications en parallèle en exploitant les processeurs multi-cœurs. Bien que NEWMADELEINE fournisse une interface MPI – appelée MAD-MPI–, cette dernière reste limitée : les communications en mémoire partagée sont impossible, les algorithmes de communication collectives ne sont pas parfaitement optimisés, etc.

Il paraît donc naturel de combiner les deux bibliothèques de communication afin de bénéficier des algorithmes de communication collective et des performances de MPICH2 en mémoire partagée ainsi que des stratégies d'optimisation et de la gestion du multirail de NEWMADELEINE. L'intégration de NEWMADELEINE dans MPICH2 a été réalisée grâce aux modules réseau de NEMESIS. En effet, outre les communications en mémoire partagée, NEMESIS est également capable d'exploiter diverses technologies réseau grâce aux modules réseau implémentés. Ces modules fournissent un ensemble de fonctions restreint. Ces fonctions se résument à une méthode d'initialisation (`net_module_init`), une fonction d'envoi de données (`net_module_send`), une fonction de détection (`net_module_poll`) et une fonction de terminaison (`net_module_finalize`). La création d'un nouveau module réseau destiné à NEWMADELEINE, réalisée par Guillaume Mercier, a donc consisté en l'implémentation de ces quelques fonctions en les faisant reposer sur NEWMADELEINE. Nous ne présentons pas ici les détails de cette intégration, mais ceux-ci ont fait l'objet d'une publication qui décrit les principaux mécanismes mis en œuvre [MTBB09]. Bien que MPICH2 inclue des mécanismes de protection permettant aux applications d'y accéder de manière concurrente, quelques portions du code de MPICH2/NEWMADELEINE ne sont pas protégées. Il n'est donc pas possible, à l'heure actuelle, de bénéficier du mode de protection `MPI_THREAD_MULTIPLE` avec cette implémentation.

6.4.3 Détection des événements d'entrées/sorties

Afin de bénéficier de la progression des communications dans MPICH2, nous avons également mis au point un mécanisme permettant la détection des entrées/sorties – que ce soit les entrées/sorties en mémoire partagée ou celles utilisant le réseau – Il s'agit ici de fournir un mécanisme complémentaire à la détection des événements provenant du réseau. Nous avons donc développé un système de boîtes aux lettres dont le fonctionnement est proche de celui des *fastboxes* : chaque processus MPI crée une zone mémoire partagée permettant de signaler qu'un message lui a été transmis – que ce soit dans la file de réception ou dans une *fastbox* – Cette zone mémoire est en fait un sémaphore contrôlé par PIOMAN. Ainsi, comme le montre la figure 6.17, lors de la transmission d'un message en mémoire partagée, le processus émetteur ajoute le message à la file de réception (1a) ou dans une *fastbox* (1b) et déclenche le sémaphore (2). Le récepteur peut alors détecter l'arrivée d'un message en scrutant la zone mémoire contrôlée par PIOMAN. Dès que le sémaphore est débloqué, NEMESIS peut vérifier l'état des *fastboxes* ainsi que sa file de réception à la recherche du message.

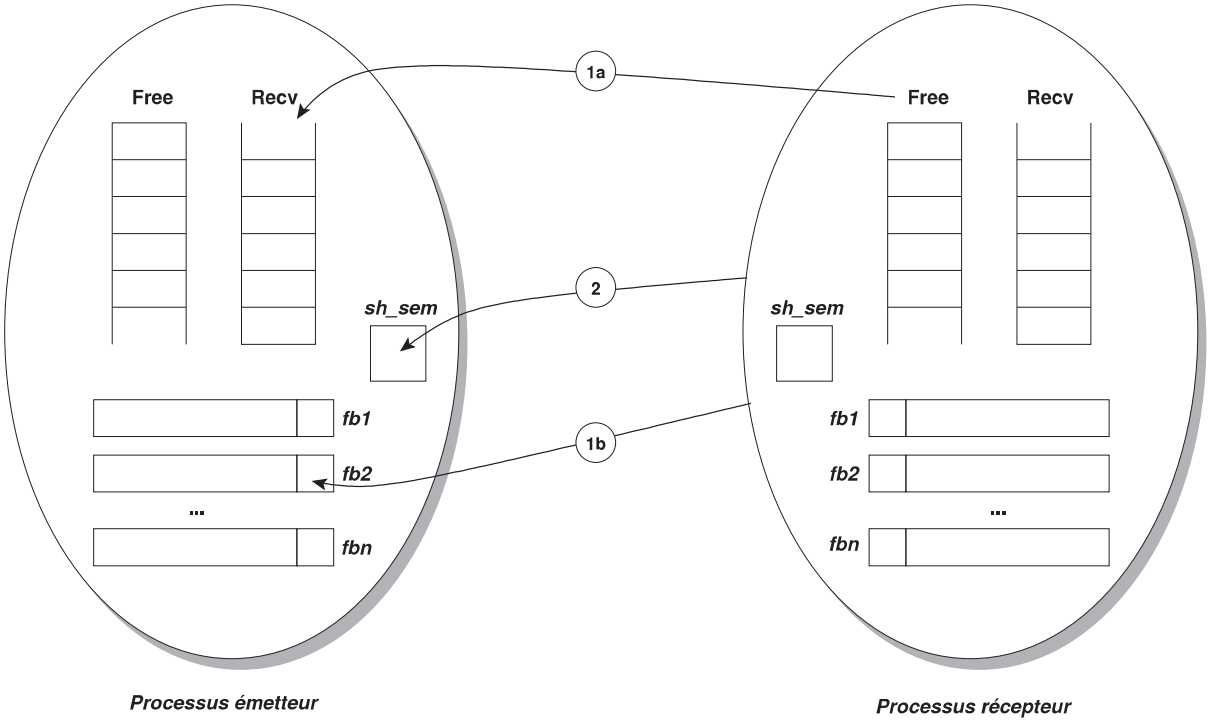

FIGURE 6.17 – *Transmission d'un message en mémoire partagée grâce au mécanisme de sémaphore dans* NEMESIS.

Ce mécanisme de boîte aux lettres prend tout son sens lorsque l'on utilise conjointement NEMESIS et NEWMADELEINE. Il est en effet possible de spécifier à NEWMADELEINE le sémaphore à utiliser pour notifier les événements provenant du réseau. La progression des communications – qu'elles soient intra-nœuds ou inter-nœuds– est donc déléguée à PIOMAN qui, en fonction du type de requête, exécute les fonctions de rappel (s'il s'agit d'une communication inter-nœud) ou scrute le sémaphore (s'il s'agit d'une communication en mémoire partagée). Le système de progression des communications de NEMESIS a également été modifié afin de tirer parti de la progression fournie par PIOMAN. Les boucles d'attente active qui interrogent tour à tour le réseau et la mémoire partagée ont été remplacées par des appels bloquants aux sémaphores fournis par PIOMAN. Ainsi, la détection des événements est entièrement gérée par PIOMAN et, d'une manière similaire aux mécanismes décrits dans la section 5.3.2, les fonctions d'attente de MPICH2 permettent de bloquer les threads afin de laisser du temps processeurs aux autres threads de l'application.

6.5 Bilan de l'implémentation

Nous avons présenté dans ce chapitre l'implémentation des différents mécanismes décrits au chapitre précédent. L'implémentation vise les différents modules de la pile logicielle : des mécanismes de protections ont été ajouté à la bibliothèque de communication, l'ordonnanceur de threads a été modifié afin d'aider à la progression des communications, un gestionnaire d'entrées/sorties a été ajouté afin de gérer les interactions entre la bibliothèque de communication et l'ordonnanceur, et le moteur de progression des communications dans MPICH2 a été modifié afin d'exploiter les différents cœurs présents. Tout au

long de l'implémentation, nous avons tenté de concevoir des algorithmes permettant d'obtenir de bonnes performances pour les communications quels que soient les environnements d'exécution – application mono-programmées ou multi-threadées, machines surchargée ou sous-utilisées, etc. – Il convient donc de tester les différents mécanismes implémentés afin de vérifier leur fonctionnement et d'évaluer leurs performances.

Chapitre 7

Évaluations

Sommaire

Dans ce chapitre, nous présentons les expériences que nous avons menées afin d'évaluer PIOMAN et NEWMADELEINE. Dans un premier temps, nous présentons les plates-formes d'expérimentation que nous avons utilisées. Des applications simplifiées nous permettront ensuite de vérifier les performances des mécanismes développés. Nous présentons ensuite les résultats obtenus avec des programmes mettant en œuvre des schémas de communication plus élaborés afin d'observer le comportement de PIOMAN et NEWMADELEINE. Enfin, nous évaluons MPICH2/NEWMADELEINE avec des applications réelles afin de vérifier que les comportements observés pour des tests synthétiques se retrouvent bien dans des applications et améliorent les performances.

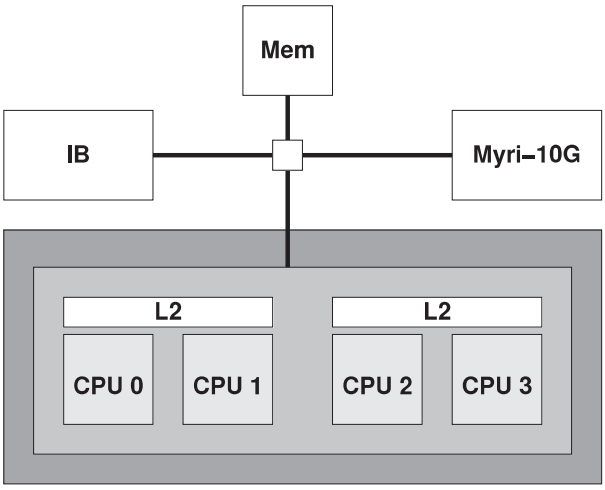

FIGURE 7.1 – *Topologie des machines de la grappe* JOE.

7.1 Plate-forme d'expérimentation

7.1.1 Configuration matérielle

Les résultats présentés dans ce chapitre ont été mesurés sur deux grappes de calcul. La première grappe, nommée JOE, est composée de deux machines équipées chacune d'un processeur quadri-cœur INTEL XEON X5460 cadencé à 3,16 GHz dont la topologie est présentée dans la figure 7.1. Les machines possèdent 4 Go de mémoire et utilisent Linux (version 2.6.29). Les nœuds sont équipés de cartes MYRI-10G (avec le pilote MX 1.2.7) et de cartes INFINIBAND CONNECTX (MT25418, utilisant le pilote OFED 1.3.1).

La seconde grappe, nommée BORDERLINE, est composée de 10 nœuds comportant chacun 8 cœurs cadencés à 2,6 GHz. L'architecture des machines est présentée dans la figure 7.2 : chaque machine comporte 4 processeurs bi-cœur AMD OPTERON 8218. Chaque processeur est attaché à 8 Go de mémoire. La machine comporte donc 4 nœuds NUMA. Les machines sont équipées de cartes MYRI-10G (avec le pilote MX 1.2.7) et de cartes INFINIBAND CONNECTX (MT25418, utilisant le pilote OFED 1.2) et utilisent Linux (version 2.6.22). Les cartes MYRINET et INFINIBAND sont connectées respectivement aux nœuds NUMA 0 et 1.

7.1.2 Éléments de comparaison

Afin de pouvoir comparer les résultats obtenus avec NEWMADELEINE et PIOMAN, d'autres bibliothèques de communication ont également été utilisées pour les tests présentés dans ce chapitre. Les programmes employés tout au long de ce chapitre utilisent le standard MPI et les bibliothèques de communication sélectionnées implémentent donc le standard MPI-2 :

MPICH2 Le RADIX LAB (Argonne National Laboratory, États-Unis) développe l'une des implémentations MPI libres parmi les plus utilisées : MPICH2 [mpi07]. Cette bibliothèque de communication est générique et est à la base de plusieurs implémentations MPI spécialisées, que ce soit pour

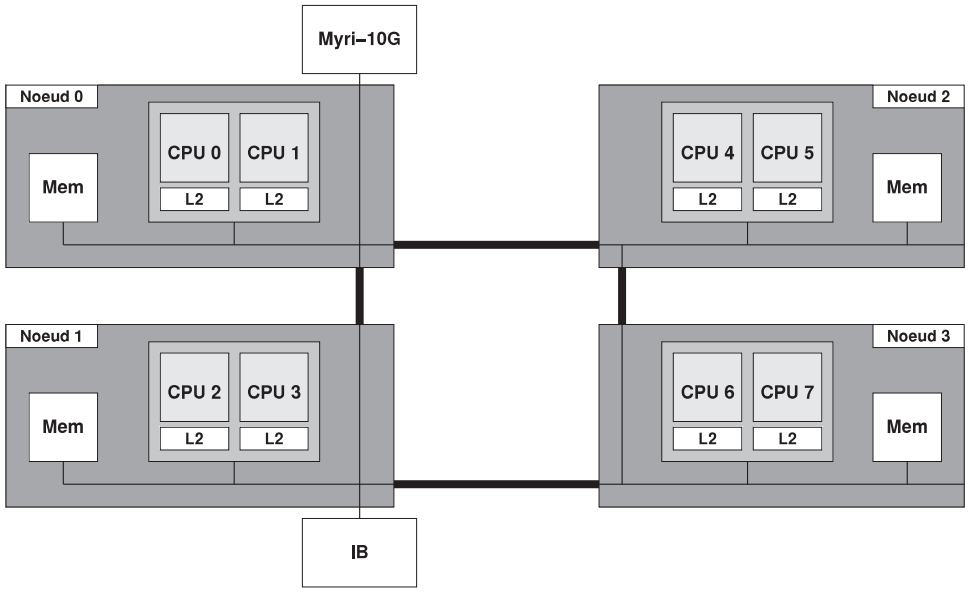

FIGURE 7.2 – *Topologie des machines de la grappe* BORDERLINE.

des technologies réseau (MVAPICH2 ou MPICH-MX par exemple), des types de processeurs (INTEL MPI) ou des types de super-calculateurs (pour BLUEGENE, CRAY, SICORTEX, etc.) La pile logicielle de MPICH2 est complexe : comme nous l'avons précisé dans la section 3.2, une technologie réseau peut être implantée à plusieurs niveaux. Nous avons utilisé le module réseau MX du *channel* NEMESIS. Ce module réseau permet de bénéficier des optimisations implémentées dans les couches supérieures de MPICH2 (sur les opérations collectives notamment) tout en gardant des performances proches de celles de l'interface de communication bas-niveau MX. Les mesures présentées dans ce chapitre ont été mesurées avec la version 1.1.1 de MPICH2.

OPEN MPI Cette implémentation du standard MPI est développée par un consortium regroupant des académiques (Université de l'Indiana, Université du Tennessee, Université de Dresden, etc.), des industriels (Cisco, IBM, SUN, etc.) et des organismes de recherche (Los Alamos National Laboratory, INRIA, Oak Ridge National Laboratory, etc.) Le projet est une mise en commun de l'expérience apportée par l'implémentation de plusieurs bibliothèques de communication (FT-MPI, LA-MPI, LAM/MPI, et PACX-MPI). Chacune de ces implémentations excellait dans un ou plusieurs domaines, le but d'OPEN MPI [GWS05] est de garder les bonnes idées des différents projets afin de créer une implémentation MPI performante dans tous les domaines. Tout comme pour MPICH2, l'implémentation de pilotes réseau pour OPEN MPI peut se faire à plusieurs niveaux. Comme nous l'avons décrit dans la section 3.2, un pilote réseau peut être implanté en tant que module MTL (*Message Transfer Layer*) ou en tant que module BTL (*Byte Transfer Layer*). Au cours des expériences présentées dans ce chapitre, nous avons utilisé le module BTL MX (pour les réseaux MYRINET) et le module BTL OPENIB (pour les réseaux INFINIBAND) de la version 1.3.3 de OPEN MPI.

MVAPICH2 L'Université de l'Ohio développe une bibliothèque de communication haute performance pour réseaux INFINIBAND. MVAPICH2 [HSJ+06] est en fait dérivé de l'implémentation générique MPICH2 et les modifications apportées à MPICH2 lui permettent d'exploiter efficacement les

réseaux à base de RDMA comme INFINIBAND. Outre les performances brutes du réseau, MVAPICH2 fournit des opérations collectives optimisées pour les machines multi-cœurs. Au cours des expériences présentées dans ce chapitre, nous avons utilisé la version 1.4 rc1 de MVAPICH2 qui est basée sur MPICH2 1.0.8p1.

Les résultats obtenus avec NEWMADELEINE présentés dans ce chapitre sont donc comparés aux résultats obtenus avec les trois implémentations MPI les plus utilisées. MPICH2 et OPEN MPI sont utilisés pour les tests sur réseau MYRINET, tandis que MVAPICH2 et OPEN MPI sont utilisés pour les tests exploitant le réseau INFINIBAND. Afin que les comparaisons aient un sens, les programmes utilisés adoptent tous l'interface MPI. Outre MPICH2/NEWMADELEINE, l'interface MPI de NEWMADELEINE, nommée MAD-MPI, est également utilisée.

7.2 Micro-Benchmarks

Afin d'évaluer le comportement de NEWMADELEINE et PIOMAN, nous présentons dans cette section une série de tests synthétiques. Le but est ici de déterminer les éventuels surcoûts et apports des mécanismes présentés dans les chapitres précédents. Les résultats présentés ici ont tous été obtenus sur la grappe JOE.

7.2.1 Impact des mécanismes implémentés sur les performances brutes

Afin d'évaluer l'impact de PIOMAN sur les performances brutes du réseau, nous présentons tout d'abord des tests de latence et de débit. Nous plaçons ici NEWMADELEINE et PIOMAN dans une situation où ils ne peuvent que subir les communications sans les optimiser ni profiter des éventuels cœurs inutilisés. Il s'agit ici d'évaluer le surcoût de la gestion du multi-threading pour des applications mono-programmées. Pour cela, nous utilisons le programme NETPIPE [SMG96] qui effectue une série d'aller-retour sur le réseau et qui mesure la durée moyenne afin de calculer la latence et le débit perçus par l'application lorsqu'elle communique.

7.2.1.1 Influence des mécanismes de protection

Nous évaluons tout d'abord l'impact qu'ont les mécanismes de protection contre les accès concurrents implémentés dans NEWMADELEINE. Pour cela, nous comparons les résultats obtenus avec la version de NEWMADELEINE sans mécanisme de protection et la version supportant les accès concurrents. La détection des événements est ici entièrement gérée par NEWMADELEINE et seuls les mécanismes de verrouillage sont ajoutés. Les modules logiciels MARCEL et PIOMAN sont donc chargés en mémoire, même s'ils ne sont pas utilisés –aucun thread n'est lancé et la progression des communications est assurée par NEWMADELEINE. Les latences mesurées sur la grappe JOE pour les réseaux INFINIBAND et MYRINET sont présentées dans les Figures 7.3 et 7.4.

INFINIBAND. Alors que la latence obtenue avec la version de NEWMADELEINE ne supportant pas les accès concurrents est à 2,05 μs, le mécanisme de verrouillage à gros grain affiche une latence à 2,19 μs. Ce surcoût de 150 ns correspond à deux fois le temps nécessaire à la prise et le relâchement du verrou global. En effet, le verrou global est pris deux fois sur le chemin critique : lorsque l'application

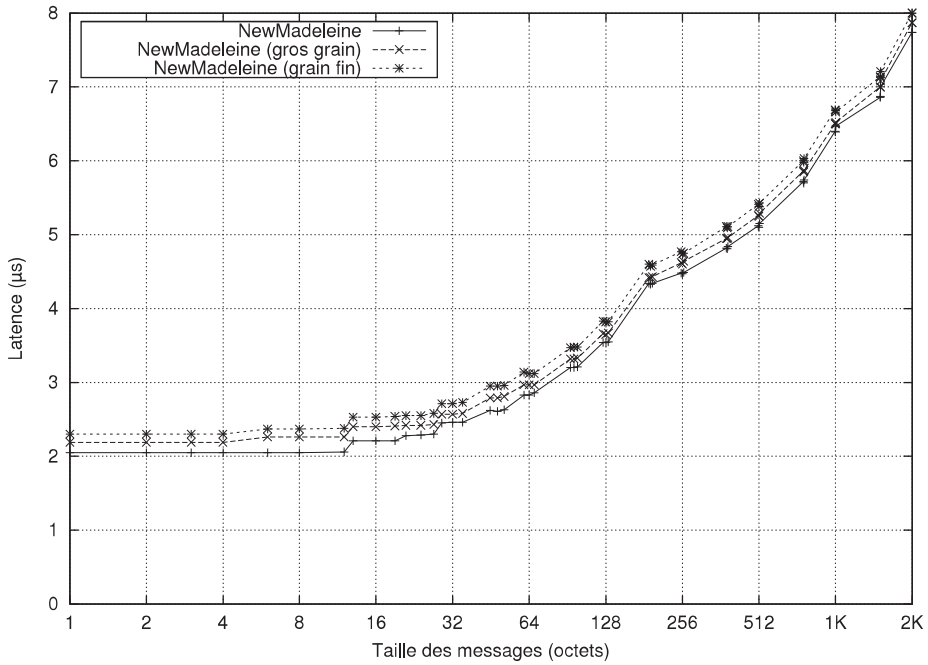

FIGURE 7.3 – *Impact des mécanismes de protection sur la latence du réseau* INFINIBAND

dépose une requête à la couche de collecte, et lorsque le paquet correspondant est soumis au réseau. Le coût correspondant à la prise et le relâchement d'un verrou (environ 70 ns) se retrouve donc deux fois. La latence obtenue avec le mécanisme de verrouillage à grain fin est de 2,30 μs, ce qui représente un surcoût légèrement plus élevé (250 ns). Cela est dû au nombre plus important de verrous à prendre et relâcher. En effet, les verrous sont pris trois fois sur le chemin critique : le verrou de la couche de collecte lorsque l'application dépose un message et lorsque la stratégie d'optimisation est appelée, et le verrou protégeant la liste des requêtes d'un réseau lorsque le paquet est soumis au pilote.

MYRINET. Le test de latence sur le réseau MYRINET montre des résultats similaires : la version ne supportant pas les accès concurrents affihce une latence de 2,62 μs alors que la latence obtenue avec le verrouillage à gros grain est de 3 μs. Le surcoût est ici plus élevé que pour le réseau INFINIBAND à cause du nombre plus élevé de scrutations nécessaires pour détecter la terminaison d'une requête. En effet, MYRINET est légèrement moins rapide que INFINIBAND et le nombre de scrutations (et donc de verrouillages) est plus élevé pour MYRINET. Le mécanisme de verrouillage à grain fin permet d'obtenir une latence à 3,10 μs, ce qui représente un surcoût supplémentaire de 100 ns par rapport au verrouillage à gros grain.

Les mécanismes de verrouillage implémentés dans NEWMADELEINE ont donc un impact réduit sur les performances brutes du réseau. Le surcoût introduit par les mécanismes de protection reste constant et l'impact sur les débits est donc négligeable.

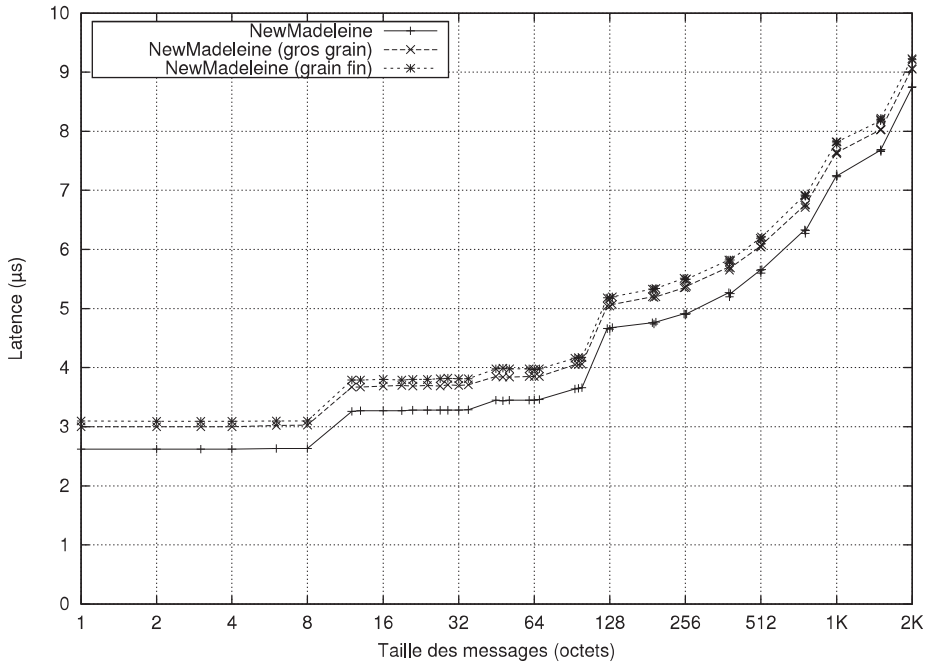

FIGURE 7.4 – *Impact des mécanismes de protection sur la latence du réseau* MYRINET

7.2.1.2 Délégation de la détection des événements

Nous évaluons ici l'impact de la délégation de la détection des événements. Pour cela, nous comparons les résultats obtenus avec une version de NEWMADELEINE supportant les accès concurrents (le mécanisme de verrouillage à grains fins) et une autre version de NEWMADELEINE similaire, mais qui délègue la détection des événements réseau à PIOMAN. Enfin nous comparons ces résultats à ceux obtenus avec d'autres implémentations MPI. Dans tous les cas, les mécanismes de protection sont activés – *i.e.* MPI est initialisé avec le mode `MPI_THREAD_MULTIPLE` – afin que les comparaisons aient un sens.

INFINIBAND. Comme le montre la figure 7.5, lorsque NEWMADELEINE se charge de la détection, la latence obtenue sur le réseau INFINIBAND est de 2,30 μs. Lorsque PIOMAN gère la progression des communications, la latence est de 2,47 μs. Ce surcoût est principalement dû aux mécanismes de verrouillage internes à PIOMAN. Les performances obtenues avec PIOMAN montrent une différence d'environ 700 ns par rapport aux performances délivrées par MVAPICH2 (1,63 μs) et par OPEN MPI (1,74 μs). Les débits présentés dans la figure 7.6 sont également légèrement différents. En effet, OPEN MPI et MVAPICH2 maintiennent un *"cache d'enregistrement"* qui permet de ne pas ré-enregistrer une zone mémoire en vue d'un transfert par DMA. Ce mécanisme n'est pas implémenté dans NEWMADELEINE et les débits mesurés – 1262 Mo/s ou 1268 Mo/s avec PIOMAN– sont donc légèrement inférieurs à ceux obtenus avec OPEN MPI– 1294 Mo/s – et MVAPICH2– 1330 Mo/s.

MYRINET. La latence obtenue avec NEWMADELEINE est de 3,10 μs, comme le montre la figure 7.7. L'utilisation de PIOMAN pour la détection des événements permet d'obtenir une latence de 3,31 μs. Les performances sont proches de celles obtenues avec OPEN MPI (3,01 μs de latence). La latence mesurée avec MPICH2 (2,64 μs) est, elle, environ 650 ns plus basse que celle obtenue avec PIOMAN. Les

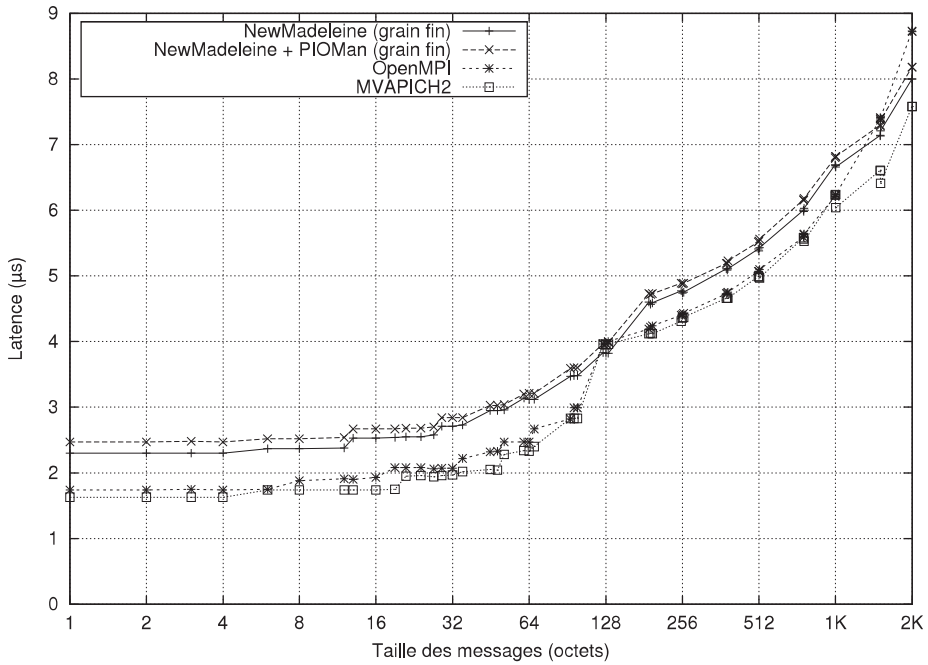

FIGURE 7.5 – *Impact de* PIOMAN *sur la latence du réseau* INFINIBAND

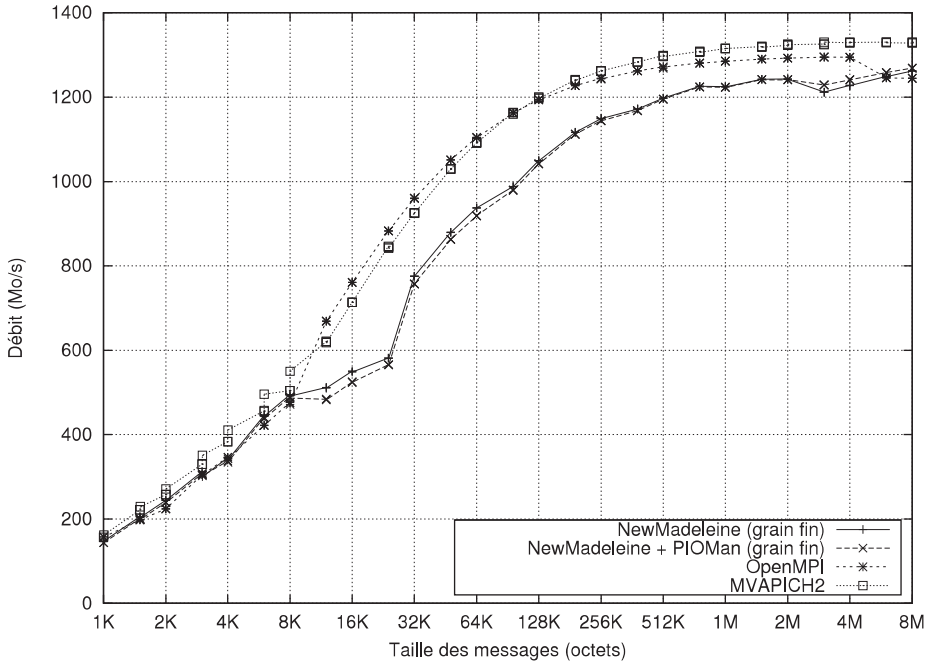

FIGURE 7.6 – *Impact de* PIOMAN *sur le débit du réseau* INFINIBAND

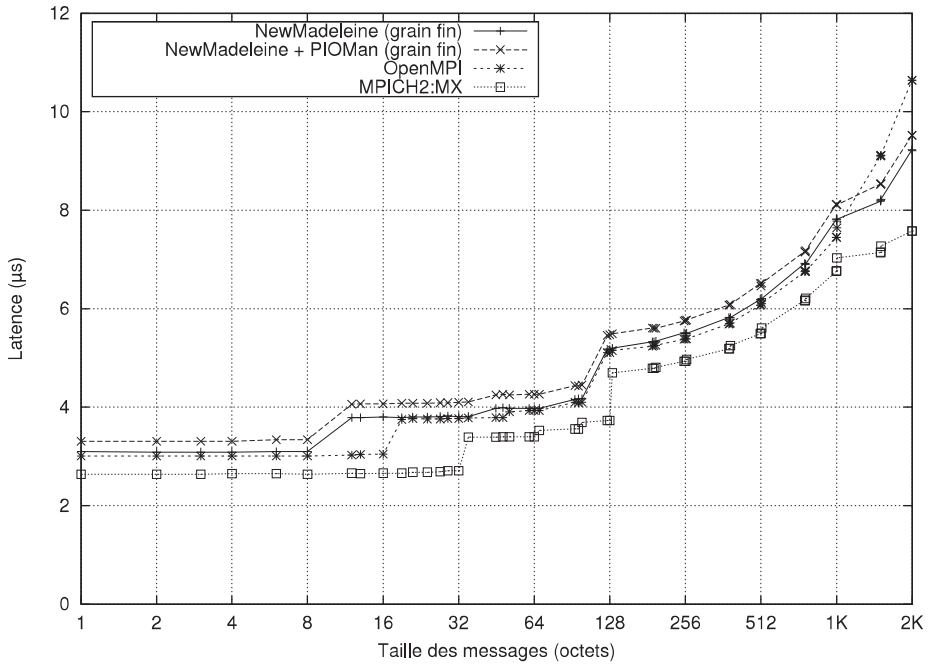

FIGURE 7.7 – *Impact de* PIOMAN *sur la latence du réseau* MYRINET

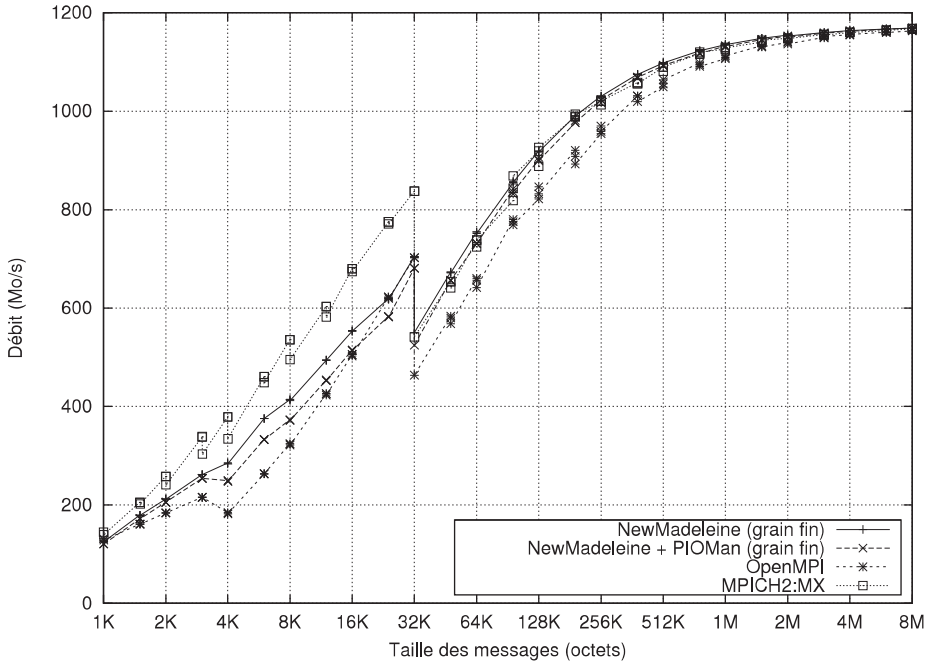

FIGURE 7.8 – *Impact de* PIOMAN *sur le débit du réseau* MYRINET

débits, présentés dans la figure 7.8, sont tous relativement semblable (entre 1164 et 1168 Mo/s). Cela est dû à l'interface de communication MX qui n'expose pas les différents mécanismes de transmission par le réseau tels que l'enregistrement mémoire ou les transfert par DMA. Ainsi, seules de légère variations différencient les implémentations MPI testées.

7.2.1.3 Impact de PIOMAN dans MPICH2

Nous évaluons ici l'impact des mécanismes de progression des communications implémentés dans MPICH2. Pour cela, nous comparons les résultats obtenus avec MPICH2/NEWMADELEINE aux résultats délivrés par la version utilisant PIOMAN comme moteur de progression.

Les résultats des tests de latence pour les réseaux MYRINET et INFINIBAND sont présentés dans les figures 7.9 et 7.10. Le surcoût introduit par l'utilisation de PIOMAN est élevé (environ 2.5 μs) et est causé par de nombreux mécanismes de protection. L'implémentation de mécanismes de protection moins pénalisant est un travail qui reste à faire afin de réduire à quelques de nano-secondes le surcoût de l'utilisation de PIOMAN dans NEMESIS. Les surcoûts mesurés sont constants et n'ont donc qu'une influence négligeable sur les débits.

Une partie importante du travail effectué lors de l'intégration de PIOMAN fut la détection des événements d'entrées/sorties en mémoire partagée. La figure 7.11 montre les résultats du test de latence en mémoire partagée. Les processus MPI sont ici lancés sur une même machine et de telle sorte qu'ils s'exécutent sur deux cœurs partageant un cache. Nous observons que l'utilisation de PIOMAN pour la détection de message intra-nœuds entraîne un surcoût de moins de 50 ns. Les débits mesurés, présentés dans la figure 7.12, sont relativement similaires. Le débit obtenu avec MVAPICH2 est très réduit par rapport aux débits de MPICH2 ou de OPEN MPI. Cela peut s'expliquer par l'utilisation que fait MVAPICH2 de la carte réseau INFINIBAND pour les communications locales, là où OPEN MPI et MPICH2 utilisent un segment de mémoire partagée.

7.2.2 Communications concurrentes

Cette partie a pour but d'évaluer l'efficacité des mécanismes de protection implémentés dans NEWMADELEINE. Nous utilisons ici des programmes multi-threadés accédant intensivement à la bibliothèque de communication de manière concurrente. Pour cette raison, la version de MPICH2 NEMESIS utilisant NEWMADELEINE et PIOMAN n'est pas évaluée ici car cette pile logicielle ne supporte pas l'utilisation de threads par l'application. OPEN MPI ne figure pas dans les résultats présentés ici car, malgré le support des accès concurrents affiché, cette implémentation génère systématiquement des erreurs pour ces tests. En effet, bien que des mécanismes de protection aient été implémentés dans OPEN MPI, ceux-ci n'ont été que très peu testés et les erreurs sont très fréquentes [Bar09, Squ09].

7.2.2.1 Impact du verrouillage

Nous évaluons ici l'impact des mécanismes de protection lorsque plusieurs threads accèdent de manière concurrente à la bibliothèque de communication. Pour cela, nous utilisons le programme de test `concurrent_ping` qui effectue des *"ping-pong"* de manière concurrente. Les deux processus lancent chacun un nombre défini de threads. Les paires de threads communiquent alors entre eux et on mesure le temps de transfert moyen pour un message de 8 octets.

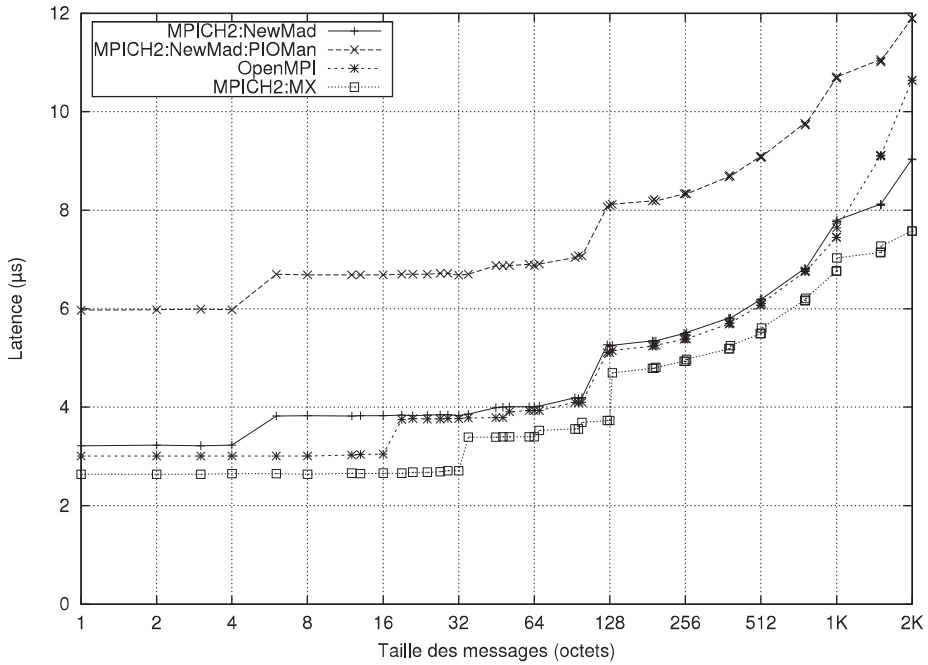

FIGURE 7.9 – *Impact de* PIOMAN *dans* MPICH2 *sur la latence du réseau* MYRINET

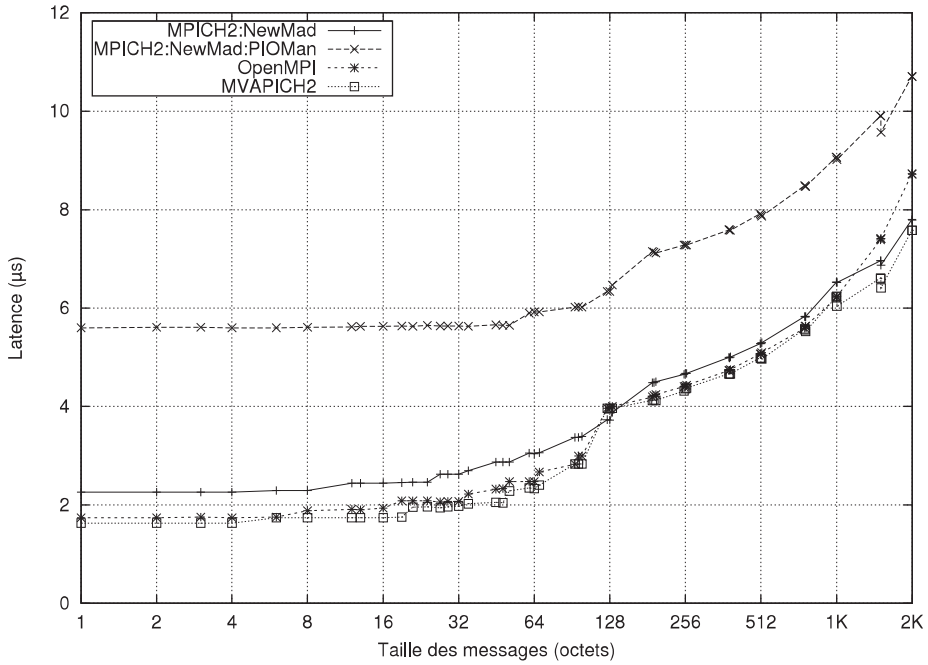

FIGURE 7.10 – *Impact de* PIOMAN *dans* MPICH2 *sur la latence du réseau* INFINIBAND

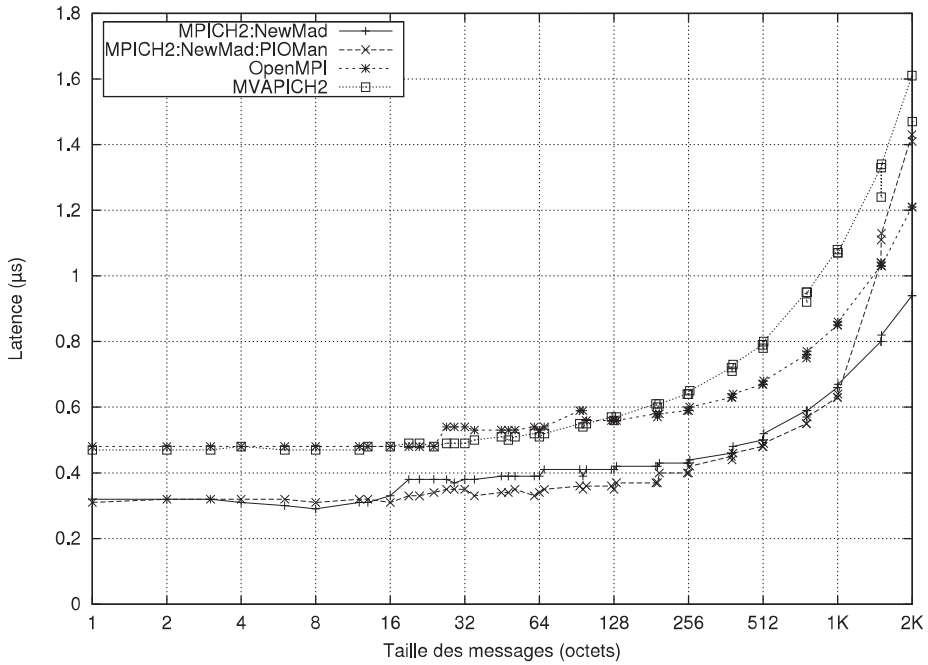

FIGURE 7.11 – *Impact de* PIOMAN *dans* MPICH2 *sur la latence en mémoire partagée*

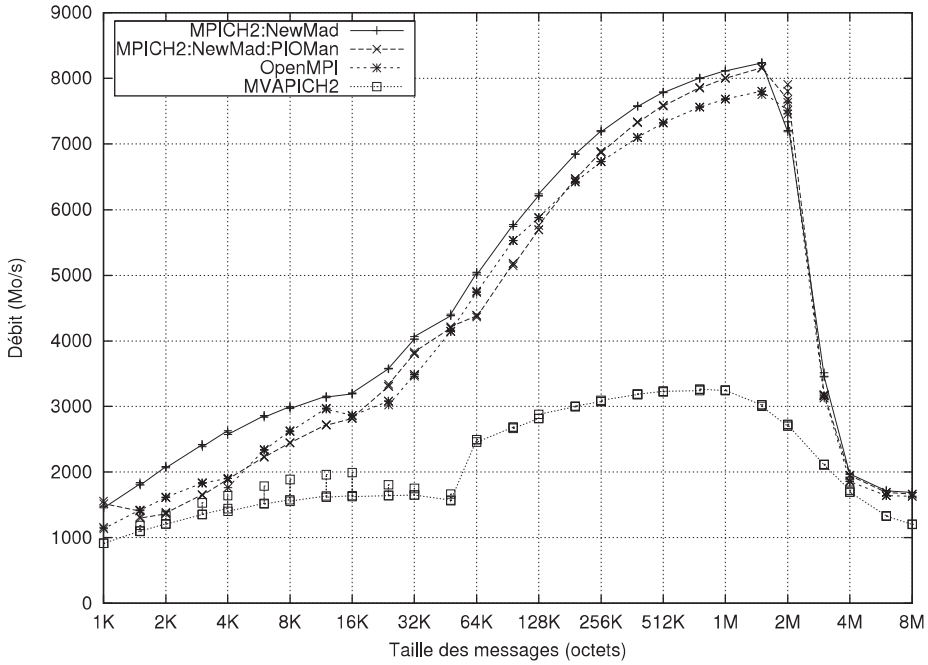

FIGURE 7.12 – *Impact de* PIOMAN *dans* MPICH2 *sur le débit en mémoire partagée*

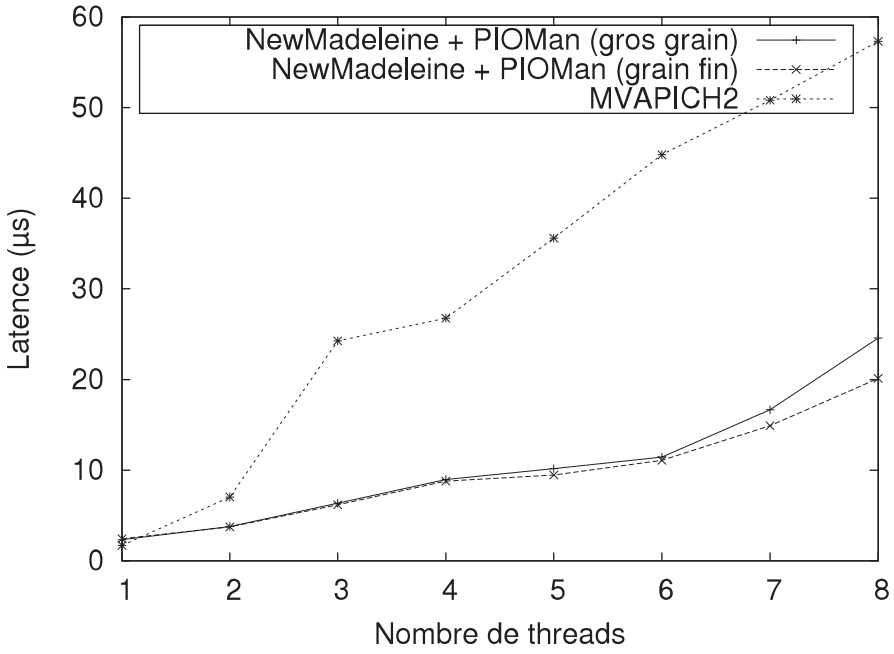

FIGURE 7.13 – *Impact des mécanismes de verrouillage sur des communications concurrentes sur le réseau* INFINIBAND

Les résultats mesurés pour les réseaux INFINIBAND et MYRINET sont reportés dans les figures 7.13 et 7.14. Les mécanismes de verrouillage à gros grain et à grain fin ont des comportements similaires : lorsque le nombre de threads augmente, la latence moyenne augmente légèrement. La contention sur les verrous ainsi que la taille des sections critiques influent sur cette augmentation. Ainsi, la latence obtenue avec le verrouillage à grain fin, dont la section critique est petite, augmente moins vite que celle obtenue avec le verrouillage à gros grain.

Les performances obtenues avec MVAPICH2 et MPICH2-NEMESIS sont beaucoup plus influencées par le nombre de threads. Ainsi, pour chaque thread ajouté, la latence mesurée augmente d'environ 7 μs pour MVAPICH2 et de 15 à 20 μs pour MPICH2. Ces performances peuvent être expliquées par la taille importante de la section critique dans ces implémentations ainsi que par la contention sur les verrous employés.

7.2.2.2 Impact de la fonction d'attente

Cette partie a pour but d'évaluer l'impact des fonctions d'attente implémentées dans NEWMADELEINE. Pour cela, nous utilisons le programme `latency_mt` inclus dans la suite de benchmarks OMB [LCW+03]. Il s'agit ici encore de mesurer la latence moyenne d'un programme multi-threadé pour des messages de 4 octets : le processus récepteur lance un certain nombre de threads qui communiquent avec un unique thread du côté émetteur. Ainsi, les threads du récepteur attendent le thread émetteur la plupart du temps.

Les résultats obtenus pour le réseau INFINIBAND sont présentés dans la figure 7.16. La latence mesurée pour MVAPICH2 augmente fortement avec le nombre de threads. Cette augmentation est due à l'implémentation des fonctions d'attente dans MVAPICH2 sous forme d'attente active. Les threads en

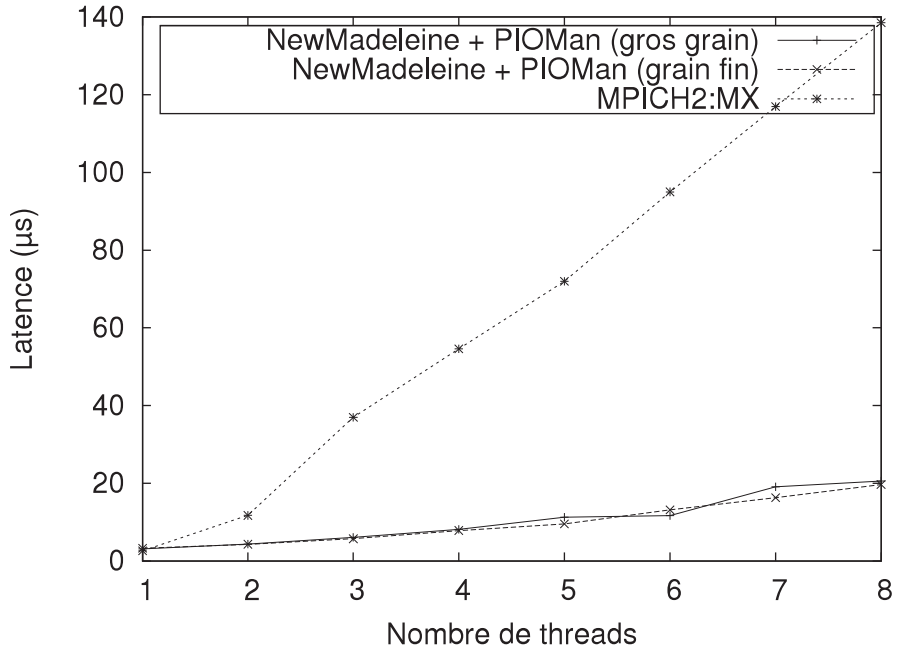

FIGURE 7.14 – *Impact des mécanismes de verrouillage sur des communications concurrentes sur le réseau* MYRINET

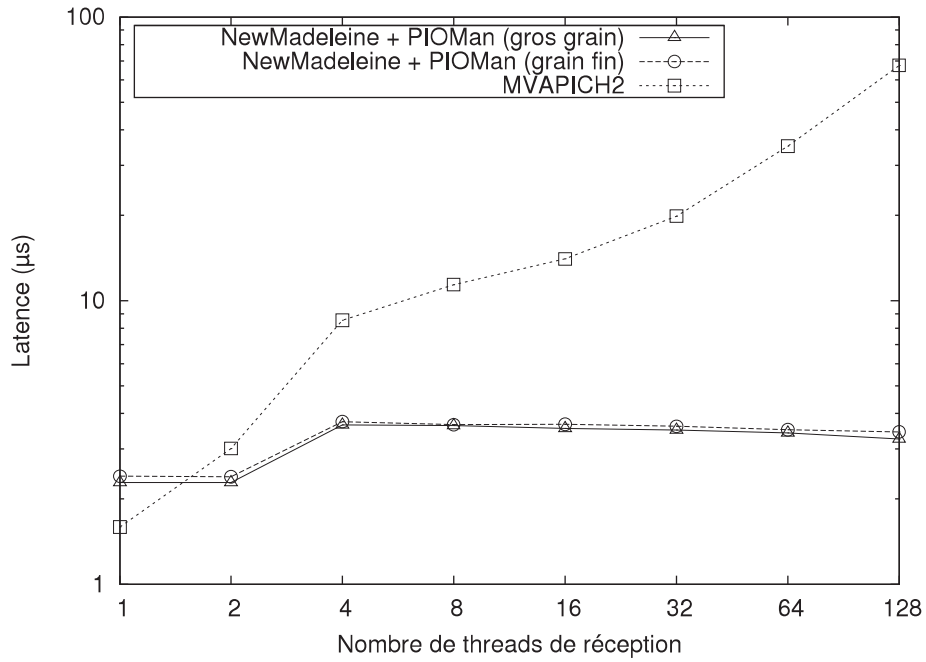

FIGURE 7.15 – *Impact des fonctions d'attente sur des communications concurrentes sur le réseau* INFINIBAND

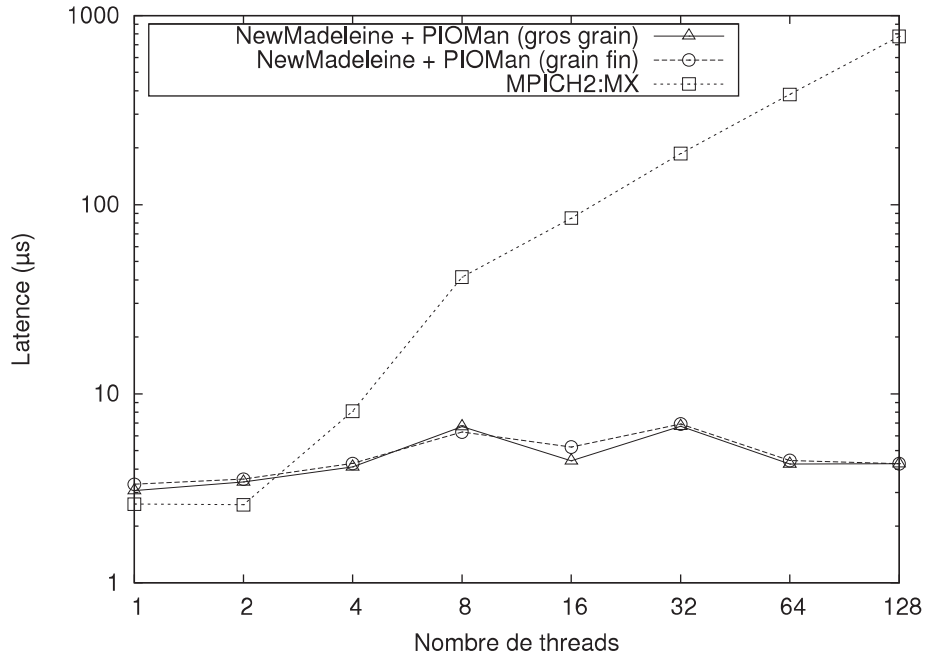

FIGURE 7.16 – *Impact des fonctions d'attente sur des communications concurrentes sur le réseau* MYRINET

attente scrutent le réseau de manière concurrente, entraînant de la contention. Les performances obtenues avec le mécanisme d'attente mixte implémenté dans PIOMAN et utilisé dans NEWMADELEINE sont, elles, fortement améliorées. Hormis un *"saut"* en passant de 2 à 4 threads, la latence mesurée reste constante à environ 3.6 μs, même lorsque la machine est surchargée (*i.e.* il y a plus de threads que de cœurs). Ce comportement s'explique par l'attente mixte utilisée par les threads. Lorsque ceux-ci attendent un message, ils se bloquent rapidement sur une condition et ne sont réveillés que lorsque le message est arrivé. Ainsi, même avec un grand nombre de threads, la machine n'est pas surchargée et la scrutation du réseau peut se faire avec une faible contention.

Les comportements observés pour le réseau MYRINET sont similaires à ceux observés pour INFINIBAND. Comme le montre la figure 7.16, la latence mesurée pour MPICH2 augmente avec le nombre de threads. Le mécanisme d'attente mixte implémenté dans PIOMAN permet à NEWMADELEINE de conserver une latence quasiment constante.

7.2.3 Progression des communications

Nous évaluons dans cette partie le comportement des mécanismes implémentés dans NEWMADELEINE et dans PIOMAN pour assurer la progression des communications. Les test synthétiques utilisés ici se rapprochent d'applications réelles dans lesquelles les threads de calcul côtoient les threads communiquants. Comme pour la partie précédente, les résultats présentés ici ont été obtenus sur la grappe JOE.

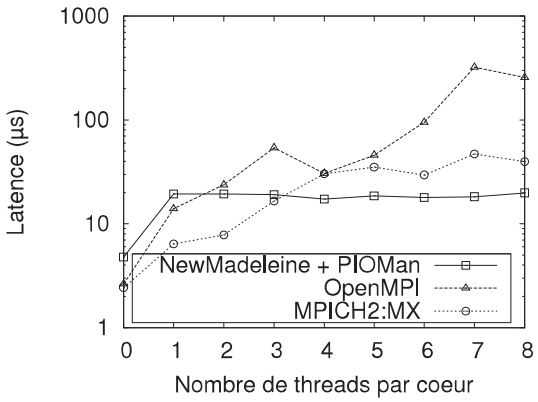

FIGURE 7.17 – *Impact de* PIOMAN *sur des machines surchargées pour le réseau* MYRINET

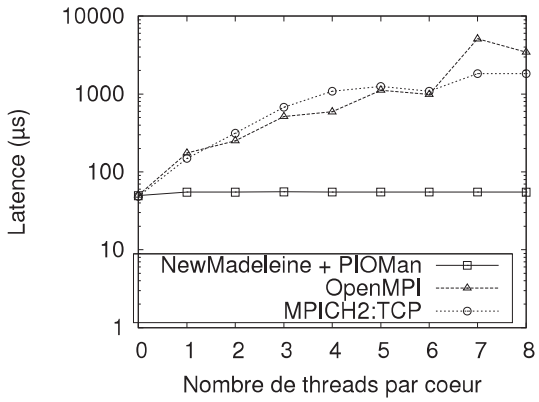

FIGURE 7.18 – *Impact de* PIOMAN *sur des machines surchargées pour le réseau TCP*

7.2.3.1 Réactivité des communications

Comme nous l'avons décrit dans le chapitre 5, la progression des communications dépends fortement de la réactivité de la bibliothèque de communication. Nous avons implémenté un mécanisme permettant de conserver des temps de réaction court, même lorsque l'application utilise un grand nombre de threads de calcul. Afin d'évaluer ce mécanisme, nous utilisons un test de réactivité qui consiste en une mesure de latence sur une machine chargée : des threads sont lancés et fixés sur les différents processeur, et le thread principal effectue un test de *"ping-pong"* avec un message de 4 octets. Le pilote INFINIBAND de NEWMADELEINE ne disposant pas d'appel bloquant, nous avons réalisé des mesures sur les réseaux MYRINET et TCP/IP.

Les résultats de ces mesures sont présentés dans les figure 7.17 et 7.18. Les comportements de MPICH2 et de OPEN MPI sont similaires pour les deux réseaux : lorsque plusieurs threads de calcul monopolisent les processeurs, la latence mesurée devient grande. De plus, les performances obtenues ne sont pas stable : d'une exécution à l'autre, de grands écarts sont constatés, malgré le grand nombre d'aller-retour constituant le test. Cela s'explique par plusieurs facteurs : tout d'abord, MPICH2 et OPEN MPI détectent la terminaison d'une requête par scrutation. Le thread principal doit donc être ordonnancé pour que les communications soient détectées, ce qui arrive plus rarement lorsque le système doit gérer un grand nombre de threads. De plus, l'attente active utilisée rend les mesures instables et gêne les threads de calcul : lorsque le thread principal est ordonnancé, il communique (avec `MPI_Send` et `MPI_Recv`) intensivement, jusqu'à ce que le système le préempte. L'application alterne donc les phases au cours desquelles les performances des communications sont bonnes avec des phases au cours desquelles aucune communication n'est détectée.

Les performances obtenues avec NEWMADELEINE et PIOMAN sont, elles, stables lorsque le nombre de threads par cœur augmente. Cela s'explique par la sélection automatique du mode de détection effectué par PIOMAN : lorsque des processeurs sont libres, ceux-ci sont utilisés pour scruter le réseau (la latence mesurée est alors faible) ; si tous les processeurs sont occupés, un appel bloquant est exporté sur un LWP supplémentaire, comme décrit dans la section 6.2.1. De plus, la latence reste stable lorsqu'un grand nombre de threads s'exécutent grâce à l'utilisation d'une bibliothèque de threads de niveau utilisateurs : l'ordonnanceur du système ne doit gérer qu'un nombre réduit de LWP (autant qu'il y a de processeurs) et le nombre de threads de niveau utilisateur n'influence pas l'ordonnancement.

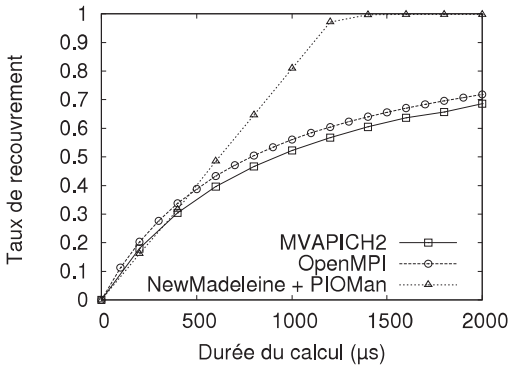

FIGURE 7.19 – *Recouvrement des communications par du calcul du côté du récepteur pour le réseau* INFINIBAND *avec des messages de 1 Mo*

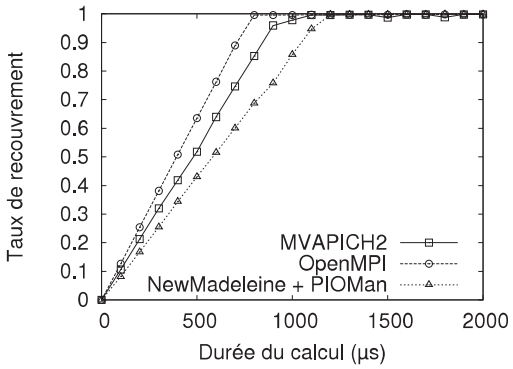

FIGURE 7.20 – *Recouvrement des communications par du calcul du côté de l'émetteur pour le réseau* INFINIBAND *avec des messages de 1 Mo*

7.2.3.2 Progression des communications

Nous évaluons ici le comportement des mécanismes de progression des communications lorsque l'application tente de recouvrir les transferts réseau par du calcul. Pour cela, nous utilisons un programme calculant le *taux de recouvrement* défini dans la section 3.1.2. Ce programme appelle une primitive non-bloquante, calcule, puis attend la fin de la communication. Après un grand nombre d'itérations, le taux de recouvrement moyen est calculé. Nous évaluons ici le taux de recouvrement en émission et en réception. En effet, comme nous l'avons vu dans la section 4.1.2, certains réseaux permettent d'optimiser le protocole du *rendez-vous* pour simplifier la progression des communications. L'utilisation de tels protocoles permet de s'affranchir de la progression du côté de l'émetteur. Il convient donc de différencier les deux types de primitives non-bloquantes. Nous n'évaluons pas ici les mécanismes de progression pour le réseau MYRINET puisque l'interface de communication MX fournit nativement un thread de progression, ce qui met les différentes implémentations de MPI sur un pied d'égalité.

Les résultats obtenus pour le réseau INFINIBAND lorsque les processus s'échangent des messages de 1 Mo sont présentés dans les figures 7.19 et 7.20. Le taux de recouvrement du côté du récepteur pour OPEN MPI et MVAPICH2 augmente lentement avec la durée du calcul. Cela est dû au coût fixe du transfert réseau qui s'ajoute à la durée du calcul. Lorsque la durée du calcul augmente, le coût des communications reste constant et le taux de recouvrement grandi jusqu'à atteindre asymptotiquement la valeur 1. Le taux de recouvrement du côté du récepteur augmente de manière linéaire lorsque PIOMAN se charge de la progression des communications. Le coût des communications est là aussi fixe, mais la progression du *rendez-vous* en arrière-plan permet de recouvrir ce coût par le calcul. Ainsi, si la réception d'un message de 1 Mo prend en moyenne 1230 μs, le taux de recouvrement atteint 1 lorsque la durée du calcul atteint 1230μs.

Lorsque l'application utilise des primitives non-bloquantes, OPEN MPI et MVAPICH2 obtiennent des résultats similaires à ceux qu'obtient PIOMAN : les communications sont totalement recouvertes dès que la durée du calcul est suffisamment grande. Toutefois, les raisons de ce recouvrement sont différentes. En effet, OPEN MPI et MVAPICH2 utilisent le protocole de *rendez-vous* à base de RDMA-Read qui est plus adapté au réseau INFINIBAND et que nous avons présenté dans la section 4.1.1. Le recouvrement obtenu par PIOMAN est lui entièrement dû à la progression des communications en arrière-plan, le

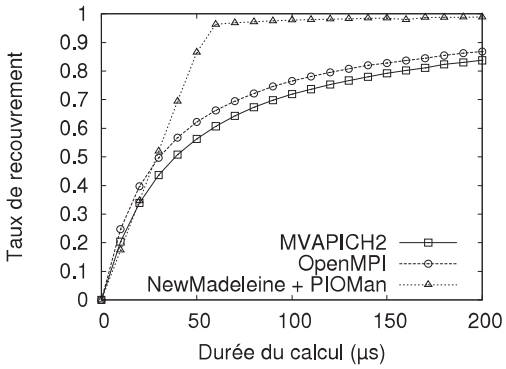

FIGURE 7.21 – *Recouvrement des communications par du calcul du côté du récepteur pour le réseau* INFINIBAND *avec des messages de 32 ko*

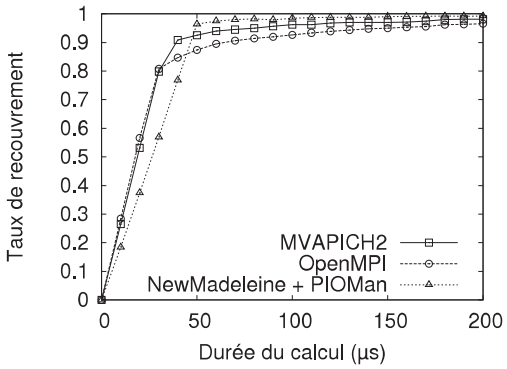

FIGURE 7.22 – *Recouvrement des communications par du calcul du côté de l'émetteur pour le réseau* INFINIBAND *avec des messages de 32 ko*

protocole de *rendez-vous* utilisé étant basé sur des RDMA-Write.

Les résultats obtenus pour le réseau INFINIBAND lorsque les processus s'échangent des messages de 32 Ko sont présentés dans les figures 7.21 et 7.22. Ces résultats sont similaires à ceux obtenus pour des messages de 1 Mo, si ce n'est que le taux de recouvrement maximum est atteint plus rapidement. En effet, la quantité de communication à recouvrir par du calcul est moins importante puisque les messages échangés sont plus petits. Il suffit alors d'une cinquantaine de microsecondes pour recouvrir complètement les communications.

7.2.4 Traitement des communications en parallèle

Nous évaluons ici le traitement des communications en parallèle dans NEWMADELEINE. Il s'agit d'évaluer l'efficacité du mécanisme de tâches et de son emploi dans NEWMADELEINE pour la parallélisation des traitements. Les résultats présentés dans cette section ont été obtenus sur la grappe JOE.

7.2.4.1 Recouvrement du calcul et des communications

Nous évaluons tout d'abord le mécanisme de décomposition des traitements des communications présenté dans la section 5.4.2. Ce mécanisme permet d'exploiter les cœurs inutilisés pour que ceux-ci se chargent de la soumission des requêtes au réseau. Pour évaluer ce mécanisme, nous utilisons un programme qui tente de recouvrir les envois de messages par de calcul. Alors que le récepteur utilise une primitive de communication bloquante (`MPI_Recv`), une phase de calcul de 20μs est insérée entre l'initialisation de l'envoi de message (*i.e.* `MPI_Isend`) et l'attente de la terminaison de l'envoi (*i.e.* `MPI_Wait`). La durée moyenne de l'ensemble – émission du message et calcul – est mesurée.

Les résultats obtenus pour le réseau INFINIBAND sont présentés dans la figure 7.23. La courbe de référence correspond au même programme sans la phase de calcul –elle représente donc la durée de l'émission– et est obtenue avec la version de NEWMADELEINE n'utilisant pas PIOMAN. MVAPICH2 et la version de NEWMADELEINE n'utilisant pas PIOMAN se comportent de la même manière : la durée d'envoi mesurée correspond à la durée du calcul à laquelle s'ajoute la durée nécessaire à l'émission du

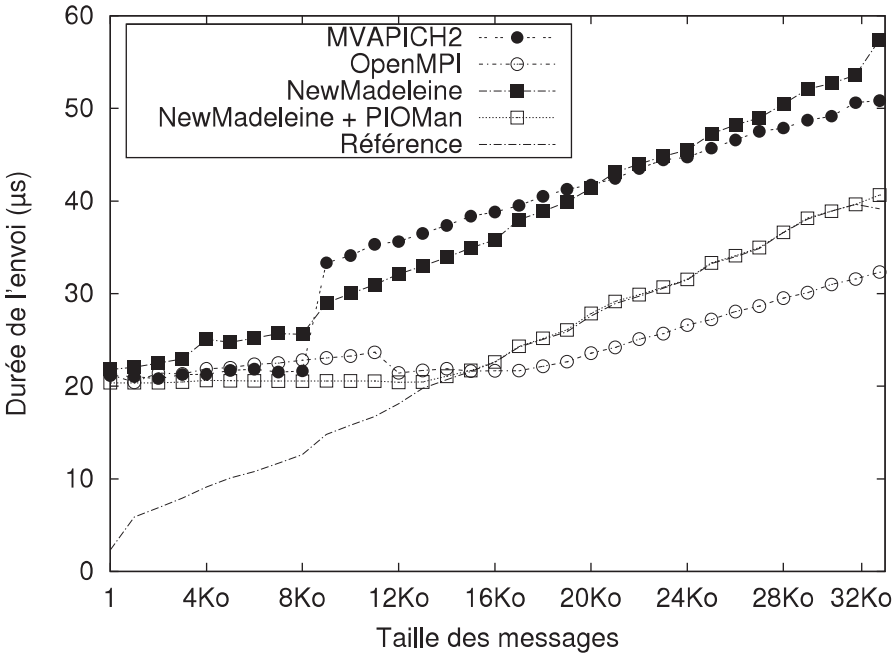

FIGURE 7.23 – *Résultats du test de recouvrement pour le réseau* INFINIBAND

message. la communication n'est donc pas recouverte par le calcul. OPEN MPI parvient à recouvrir l'envoi des messages par le calcul lorsque la taille des messages dépasse les 12 Ko. Ce cap correspond au seuil de *rendez-vous* de OPEN MPI pour le réseau INFINIBAND. Comme l'implémentation du *rendez-vous* sur INFINIBAND de OPEN MPI est basée sur le RDMA-Read, ces communications sont recouvertes par le calcul. Toutefois, OPEN MPI ne parvient pas à recouvrir l'envoi de messages plus petits par du calcul. La version multi-threadée de NEWMADELEINE parvient à recouvrir les communications par du calcul quelle que soit la taille des messages. En effet, lorsque l'application appelle la primitive d'envoi, la requête est enregistrée et un cœur inutilisé se charge de l'envoi des données. La durée mesurée correspond donc à : $max(T_{calcul}, T_{envoi})$.

Les résultats obtenus pour le réseau MYRINET sont présentés dans la figure 7.24. La courbe de référence est là aussi obtenue avec la version mono-threadée de NEWMADELEINE. Les comportements observés pour MPICH2, OPEN MPI et NEWMADELEINE sont similaires : les envois de messages ne sont pas recouverts par du calcul. La version multi-threadée de NEWMADELEINE parvient à recouvrir complètement les envois par le calcul grâce à l'utilisation de cœurs inactifs.

7.2.4.2 Gestion du multirails

Nous évaluons ici une autre mise en œuvre de la parallélisation de NEWMADELEINE qui consiste à découper un message pour le soumettre à plusieurs interfaces réseau simultanément. Afin d'évaluer ce mécanisme, nous utilisons un programme qui effectue des *"ping-pong"* et qui en mesure la latence moyenne. La latence est ici mesurée pour des tailles de message variant de 4 octets à 64 ko.

Les résultats obtenus sont présentés dans la figure 7.25. Les performances obtenues en n'utilisant qu'un des réseaux parmi MYRINET et INFINIBAND est ici comparée aux performances mesurées lorsque les

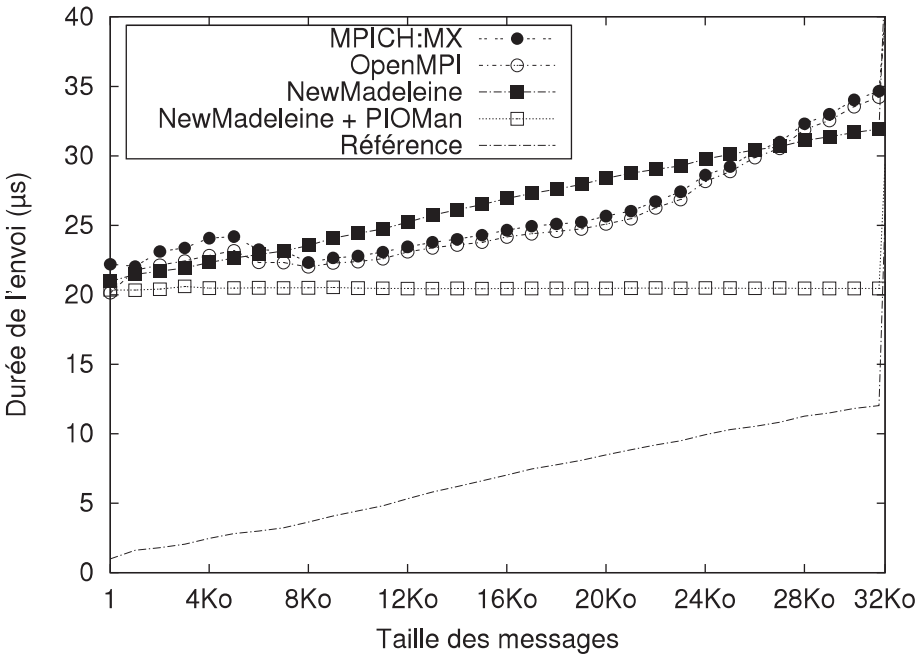

FIGURE 7.24 – *Résultats du test de recouvrement pour le réseau* MYRINET

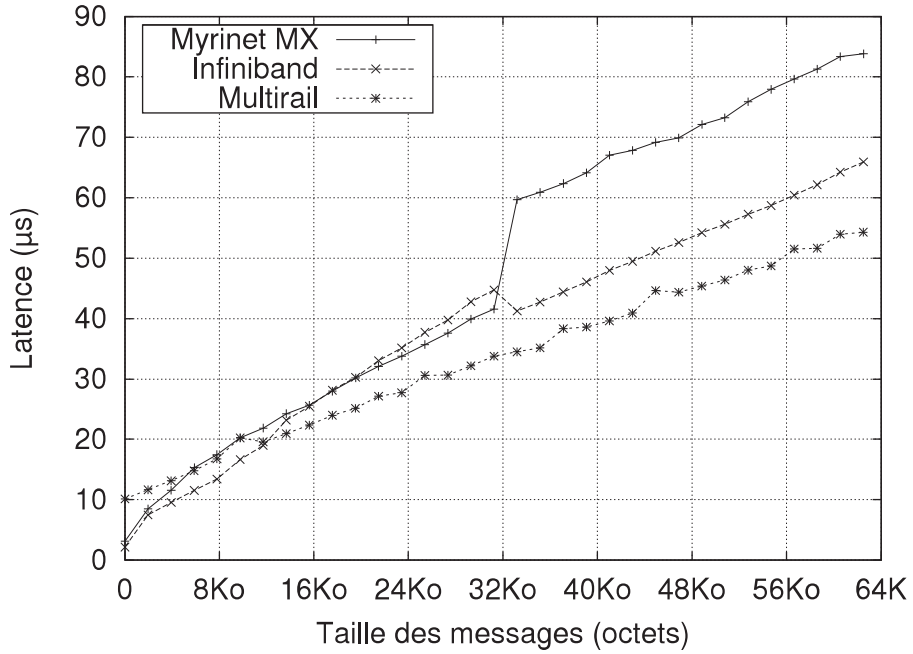

FIGURE 7.25 – *Découpage d'un message en deux morceaux et soumission aux deux réseaux simultanément*

deux réseaux sont utilisés simultanément. L'utilisation d'un seul réseau permet d'obtenir les meilleures performances pour des messages dont la taille est inférieure à 12 ko. Cela est du au surcoût lié au découpage du message et aux synchronisations entre les deux cœurs qui soumettent chacun un des morceaux de message aux réseaux. Lorsque la taille des messages est supérieure à 12 ko, l'utilisation de plusieurs réseaux simultanément permet de réduire les temps de transfert. En effet, les messages sont alors découpés en deux et deux cœurs sont utilisés pour soumettre chacun un des morceaux au réseau. Le temps de transfert mesuré est alors de la forme :

$$T(longueur) = MAX\left(T_{Réseau1}\left(\frac{longueur}{2}\right), T_{Réseau2}\left(\frac{longueur}{2}\right)\right) + Surcoût$$

Bien que le surcoût mesuré (environ 8 μs) soit important, l'utilisation de ce mécanisme permet de réduire les temps de transfert pour les messages de plus de 12 ko. La gain réalisé atteint jusqu'à 15 %.

7.3 NAS Parallel Benchmarks

Alors que le début de ce chapitre a porté sur l'évaluation des mécanismes implémentés dans PIOMAN et NEWMADELEINE à l'aide de programmes spécifiques, nous évaluons ici l'ensemble des mécanismes mis en œuvre grâce à des applications réelles. Nous utilisons pour cela la suite d'applications NAS Parallel Benchmarks (version 3.3) [BBB+91]. Cette suite a été développée par la NASA afin d'évaluer les performances à la fois matérielles et logicielles d'un calculateur. Les différents programmes qui composent cette suite ont été conçus afin d'évaluer les performances d'une plate-forme avec des applications réelles. Du fait des quelques fonctionnalités manquantes dans MPICH2/NEWMADELEINE, nous nous basons ici sur 5 des 9 applications qui composent cette suite. De plus, le programme IS n'est évalué que dans sa version utilisant 4 processus car à plus grand échelle, des types de données dérivés (*Derived Data Types*) – dont le support par MPICH2/NEWMADELEINE est encore expérimental – sont utilisés. Les résultats que nous présentons ici ont été obtenus sur la grappe BORDERLINE avec le réseau INFINIBAND.

Les résultats obtenus pour les programmes BT, CG, EP et IS sont reportés dans les figures 7.26, 7.27, 7.28 et 7.29. Les performances mesurées par les différentes implémentations MPI sont toutes relativement similaires. Toutefois, MVAPICH2 dont les performances brutes surpassent celles des autres implémentations permet d'obtenir des temps d'exécution légèrement inférieurs. L'impact de l'utilisation de PIOMAN pour la progression des communications n'est pas visible sur ces résultats : la différence de performances entre la version de MPICH2 utilisant NEWMADELEINE pour la progression des communications et celle utilisant PIOMAN reste inférieure à 2 % dans la plupart des cas. Ce résultat décevant s'explique par le manque d'optimisation à la portée de PIOMAN pour ces programmes. En effet, ces applications n'utilisent pas de primitives de communication non-bloquantes pour recouvrir les transferts réseau par du calcul. Les mécanismes de progression des communications fournis par PIOMAN ne peuvent donc pas améliorer les performances ici. Toutefois, ces résultats montrent que l'impact négatif de PIOMAN sur les performances reste très limité.

Parmi les programmes de la suite que nous utilisons, l'application SP est la seule à recouvrir une partie de ses communications par du calcul. Les résultats obtenus pour cette application sont présentés dans la figure 7.30. Dans toutes les configurations testées, les performances des différentes implémentations sont similaires. Cela peut s'expliquer par la faible part des communications dans les performances de

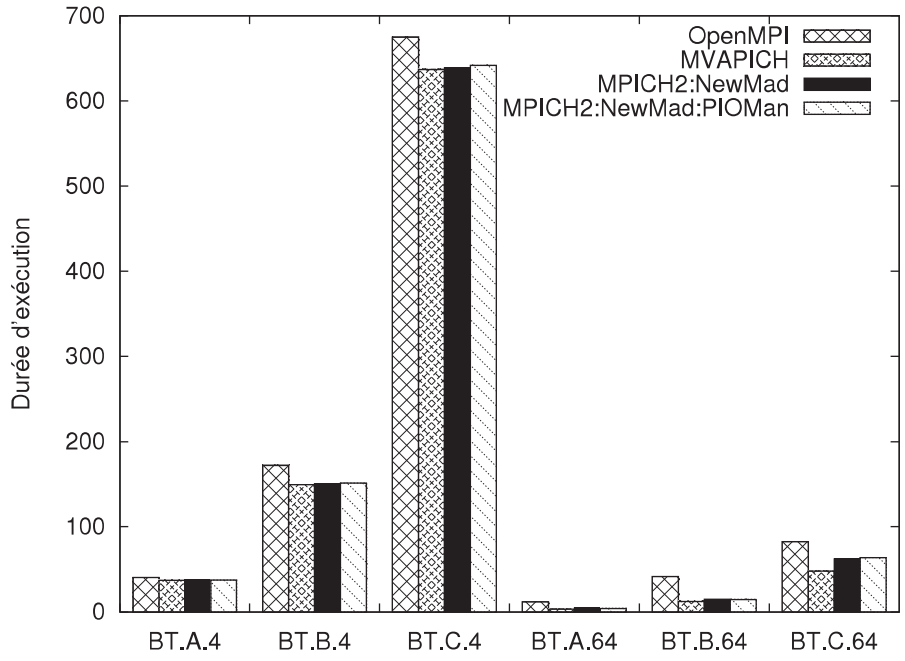

FIGURE 7.26 – *Résultats du programme BT avec 4 et 64 processus*

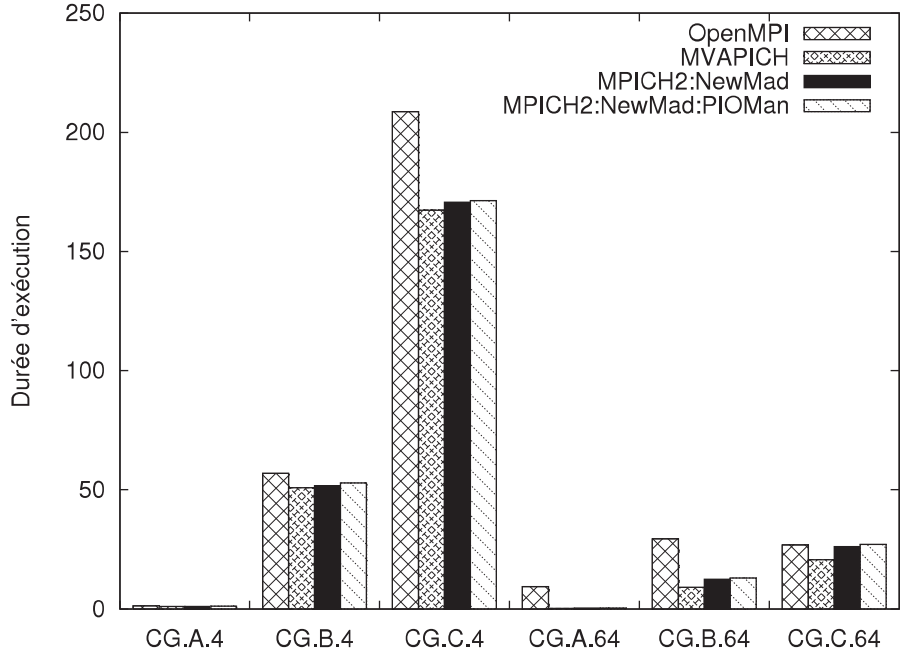

FIGURE 7.27 – *Résultats du programme CG avec 4 et 64 processus*

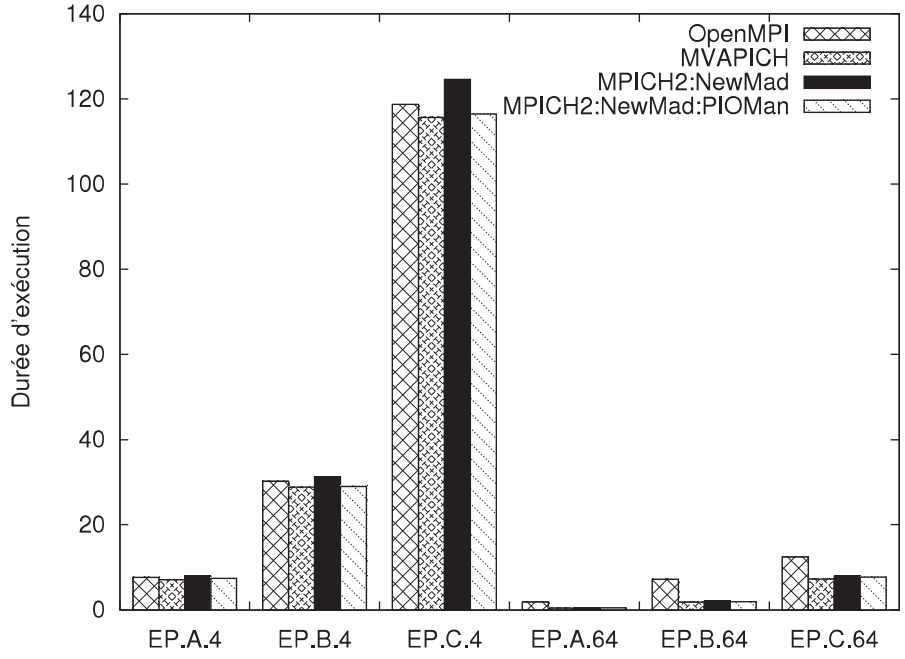

FIGURE 7.28 – *Résultats du programme EP avec 4 et 64 processus*

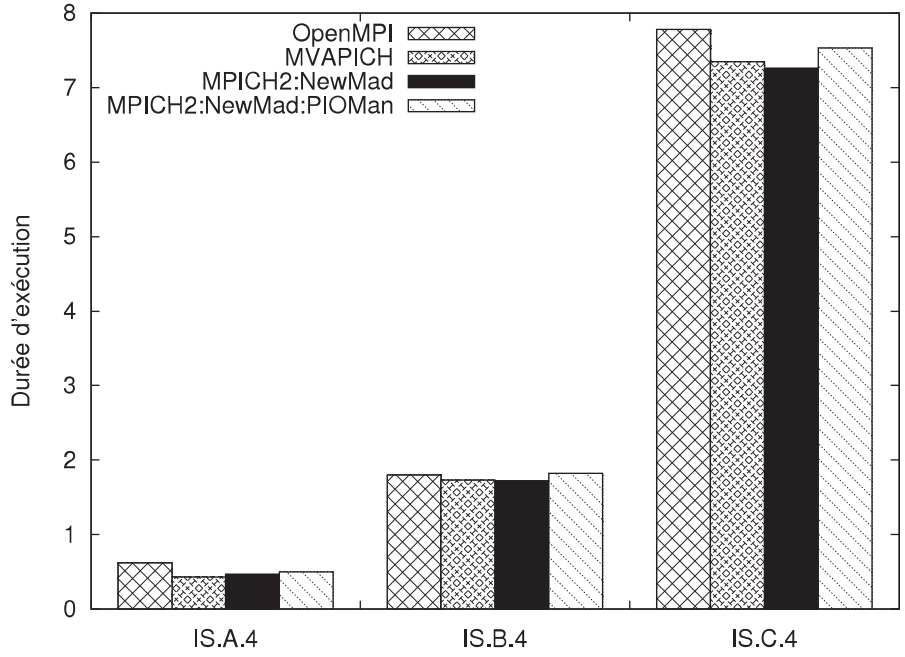

FIGURE 7.29 – *Résultats du programme IS avec 4 et 64 processus*

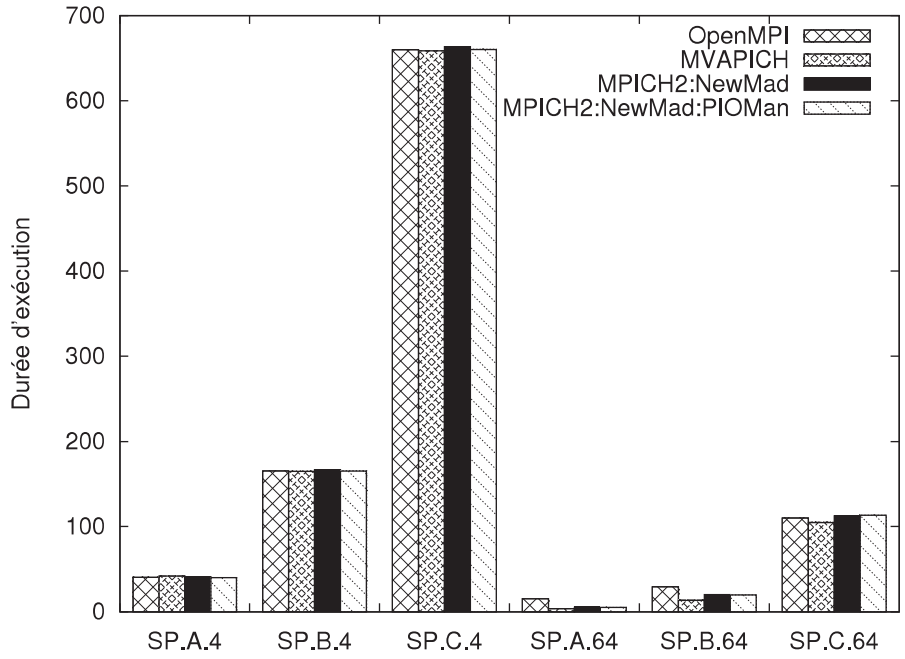

FIGURE 7.30 – *Résultats du programme SP avec 4 et 64 processus*

cette application. Ainsi, une amélioration de la quantité de communication recouverte par du calcul est très peu perceptible sur le temps d'exécution total.

Afin d'évaluer plus précisément les mécanismes de progression implémentés dans PIOMAN, nous avons donc instrumenté le code du programme afin de ne mesurer que la durée des communications. Les résultats obtenus avec ce programme pour les classes A, B et C sont présentés dans les figures 7.31, 7.32 et 7.33. Les différences de performances sont ici beaucoup plus importantes. D'une manière générale, les durées de communication mesurées pour MPICH2/NEWMADELEINE sont supérieures à celles mesurées pour OPEN MPI et MVAPICH2. MPICH2 est ici pénalisé par le léger écart des performances brutes que nous avons décrit dans la section 7.2.1.3. En effet, le but de cette évaluation n'étant pas de comparer les stratégies d'ordonnancement de NEWMADELEINE, aucune optimisation n'est effectuée ici. MPICH2 ne peut donc que souffrir du surcoût de la pile logicielle, sans bénéficier des optimisations que NEWMADELEINE pourrait apporter. Lorsque le programme est exécuté par 4 ou 16 processus, les performances obtenues avec la version de MPICH2 utilisant le moteur de progression de PIOMAN sont améliorées de 5 % à 18 %. Une partie des communications est en effet recouverte par du calcul et PIOMAN exploite les cœurs inutilisés des machines pour faire progresser ces communications. Lorsque le programme est exécuté par 64 processus les performances obtenues avec le moteur de progression de PIOMAN sont détériorées de 0.5 % à 3 %. Cela s'explique par le fait que PIOMAN ne peut pas faire progresser les communications en arrière-plan. En effet, les 64 processus utilisent tous les processeurs disponibles et PIOMAN ne peut donc pas exploiter de cœur inutilisé pour la progression des communication. Pour assurer cette progression en arrière-plan, PIOMAN devrait utiliser le mécanisme d'appel bloquant décrit dans la section 5.2.3, mais le pilote réseau permettant à NEWMADELEINE d'exploiter les réseaux INFINIBAND n'est pas capable à l'heure actuelle d'effectuer ce type d'appel système bloquant.

Les mécanismes implémentés dans PIOMAN et utilisés dans MPICH2/NEWMADELEINE permettent donc de faire progresser les communications en arrière-plan. Si les performances obtenues ne sont pas

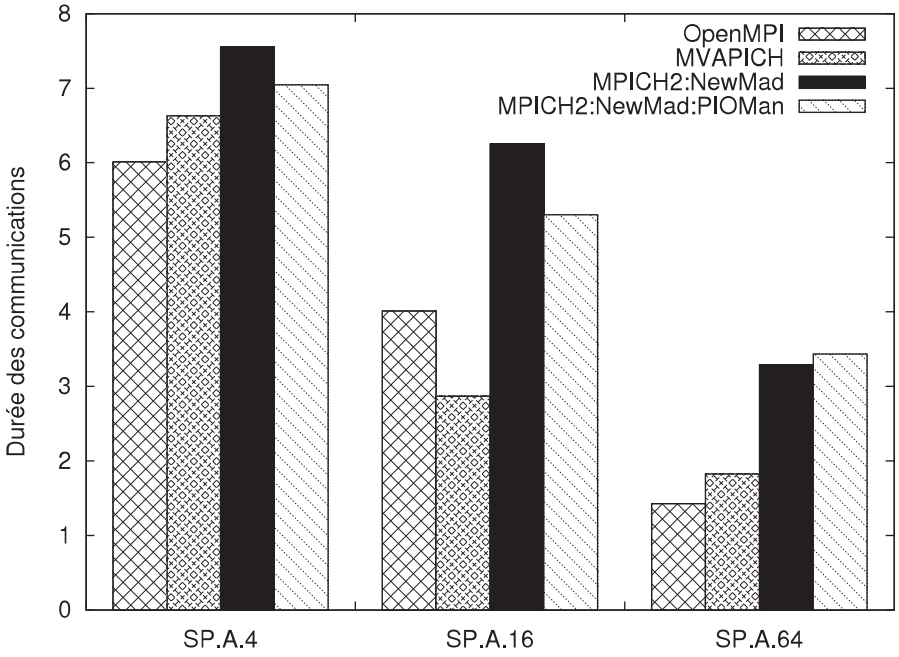

FIGURE 7.31 – *Durée des communications du programme SP, classe A.*

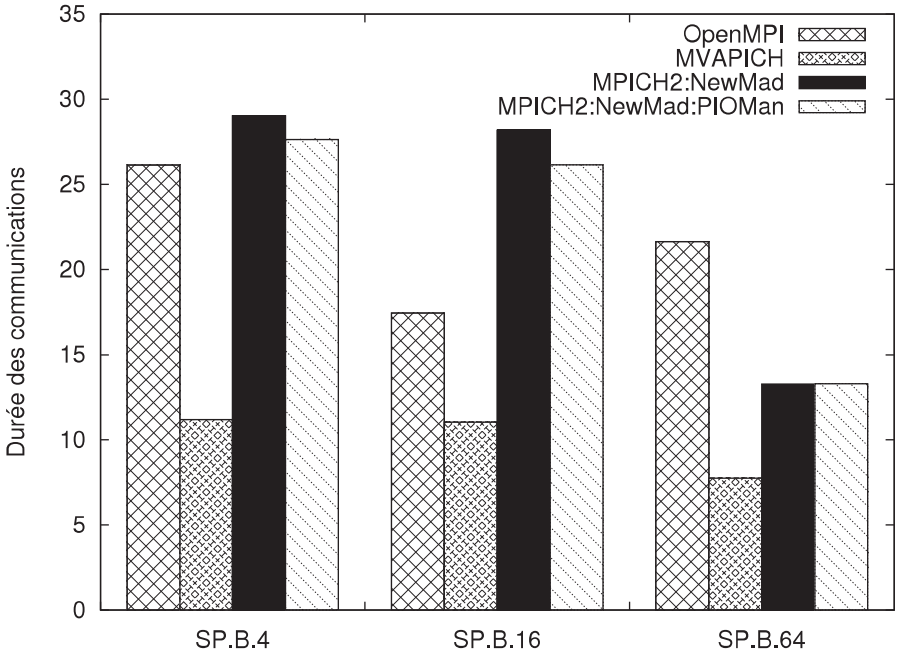

FIGURE 7.32 – *Durée des communications du programme SP, classe B.*

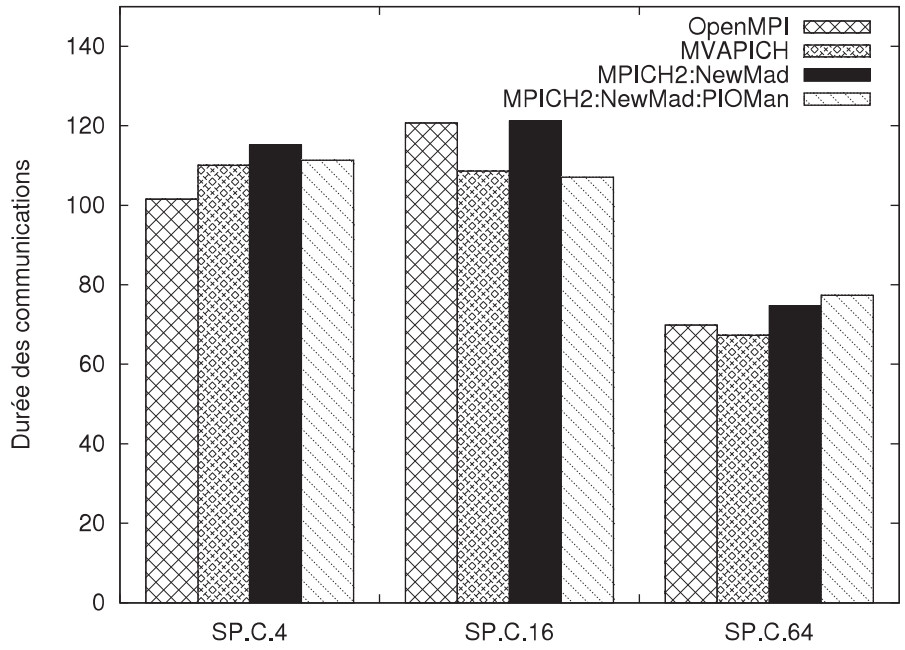

FIGURE 7.33 – *Durée des communications du programme SP, classe C.*

toujours optimales, ces mécanismes ont l'avantage d'être générique et ne dépendent pas du réseau sous-jacent. De plus, lorsque la progression des communications n'est pas possible, l'impact de PIOMAN sur les performances reste minime.

7.4 Bilan de l'évaluation

Nous avons présenté dans ce chapitre l'évaluations des différents mécanismes que nous avons proposés et implémentés. La première partie de ce chapitre a permis de montrer que ces derniers ont un impact réduit sur des schémas de communication simple de type ping-pong tout en améliorant les performances lorsque l'application se complexifie. Les accès à la bibliothèque de communication depuis plusieurs threads simultanément sont ainsi gérés efficacement et nous avons montré que NEWMADELEINE pouvait réagir rapidement à un événement réseau, même lorsque l'application surcharge la machine par des threads de calcul. L'évaluation du recouvrement des communications par du calcul a montré que PIOMAN peut exploiter les cœurs inutilisés pour faire progresser les communications en arrière-plan, ce qui permet d'atteindre des taux de recouvrement élevés.

L'évaluation des mécanismes grâce à la suite NAS Parallel Benchmarks a permis de montrer que, si les applications peuvent bénéficier des fonctionnalités proposées dans ce document, elles n'exploitent pas suffisament les primitives de communication non-bloquantes pour que le recouvrement des communications par du calcul ait un impact sur les performances.

Chapitre 8

Conclusion et Perspectives

Dans la course à la puissance de calcul, les grands constructeurs de calculateurs sont sans cesse à la recherche de solutions technologiques permettant de calculer toujours plus rapidement. Depuis les prémices du calcul parallèle, l'architecture des calculateurs a longuement évolué passant successivement du super-calculateur à la grappe de machines simples jusqu'aux grappes de machines parallèles qui envahissent aujourd'hui le paysage du calcul intensif. Les solutions matérielles évoluant par vagues, les outils permettant d'exploiter ces différents types de calculateurs doivent s'adapter rapidement.

En effet, le développement massif des processeurs multi-cœurs dans les grappes de calcul entraîne une forte augmentation du nombre d'unités de calcul par nœuds et les modèles de programmation hybrides mélangeant les threads aux communications par MPI se développent. Les affinités entre les différents cœurs d'une machine rendent très profitable l'utilisation de threads pour exploiter les unités de calcul au sein d'un nœud. Des outils tels que OpenMP ou TBB facilitent la parallélisation de codes séquentiels tout en permettant un meilleur équilibrage de charge. L'utilisation de modèles de programmation hybrides combinant le passage de messages et les threads est donc un moyen simple d'exploiter une grappe de calcul moderne.

Bien que le standard MPI ait pris en compte très tôt les changements apportés aux calculateurs en incorporant des mécanismes permettant de gérer plusieurs flots d'exécution, les implémentations du standard restent résolument tournées vers les applications mono-programmées. Le culte des performances brutes reste bien ancré et les développeurs de bibliothèques de communication ne consentent que difficilement à incorporer des mécanismes permettant de gérer le multi-threading mais qui ajoute un léger surcoût perceptible sur les performances brutes. Les problèmes de stabilité ou de performances rencontrés par les implémentations sachant gérer les accès concurrents poussent les développeurs d'applications à contourner les problèmes en arbitrant eux-même les accès aux primitives de communication.

La frilosité à l'égard des modèles de programmation hybrides mène donc à les développeurs de bibliothèque de communication à gérer les entrées/sorties malgré le multi-threading au lieu de profiter de cette technique pour exploiter efficacement les machines modernes.

Contribution

Nous avons proposé dans ce document de penser différemment la gestion des entrées/sorties en tirant parti du multi-threading au lieu de le subir. Une étroite collaboration entre l'ordonnanceur de threads et la bibliothèque de communication a pour effet une détection plus rapide des événements provenant du réseau tout en permettant une progression des calculs par les threads de l'application.

Nous avons proposé un module logiciel chargé de gérer les interactions entre la bibliothèque de communication et l'ordonnanceur de threads. Ce gestionnaire d'entrées/sorties générique prend en charge la détection des événements du réseau et exploite les multiples unités de calcul présentes sur la machine de manière transparente. Un mécanisme d'exportation de tâches permet également de paralléliser simplement une bibliothèque de communication en s'appuyant sur des événements réseau. L'utilisation de ce gestionnaire d'entrées/sorties dans une bibliothèque de communication permet donc de s'affranchir des problèmes liés au multi-threading. Ainsi, la progression des communications est entièrement déléguée à ce module et se fait en arrière-plan. La parallélisation de la bibliothèque de communication est également facilitée par le mécanisme d'exportation de tâches : les traitements sont découpés en tâches que le gestionnaire d'entrées/sorties se charge de répartir sur les différents cœurs de la machine.

Nous avons implémenté ces mécanismes au sein de la suite logicielle PM^2 sous la forme du gestionnaire d'entrées/sorties PIOMAN et nous avons modifié la bibliothèque de communication NEWMADELEINE pour qu'elle tire parti de ces outils. Grâce à une étroite collaboration avec l'ordonnanceur de threads MARCEL, PIOMAN exploite les trous laissés dans l'ordonnancement en traitant les tâches que NEWMADELEINE soumet. La progression des communications de NEWMADELEINE ainsi que la parallélisation des traitements sont ainsi déléguées à PIOMAN qui prend en charge les problématiques liées au multi-threading. Il en résulte une bibliothèque de communication adaptée aux grappes de calcul modernes et aux modèles de programmation hybrides : le multi-threading est non seulement supporté au niveau de l'application, mais également au cœur même de NEWMADELEINE. Nous avons poussé plus loin la mise en application des outils fournis par PIOMAN en intégrant les mécanismes de progression des communications dans MPICH2-NEMESIS. Outre le fait de pouvoir évaluer NEWMADELEINE et PIOMAN avec des applications réelles, cette intégration permet à MPICH2 de se doter de mécanismes adaptés aux modèles de programmation hybrides en attendant que toute la pile logicielle supporte ce type de modèles.

L'évaluation par des tests synthétiques a montré que les mécanismes implémentés fonctionnent de la manière souhaitée et que les surcoûts induits restent limités. Nous avons ainsi montré que les accès concurrents des différents threads d'une application se font efficacement. Les tests menés ont également permis de mettre en avant les bénéfices retirés par la collaboration entre l'ordonnanceur de threads et la bibliothèque de communication. Les interactions entre les modules logiciels permettent d'assurer un temps de réaction constant, même lorsque la machine est surchargée. Enfin, l'exploitation des multiples cœurs de la machine permet de recouvrir efficacement les communications par du calcul.

L'évaluation sur des applications réelles a montré que les mécanismes de progression fournis par PIOMAN permettent de réduire l'impact des communications sur le temps d'exécution global. Les applications utilisant des primitives de communication non-bloquantes pour recouvrir les transferts réseau par du calcul bénéficient donc de la progression en arrière-plan, et les performances des programmes ne recourant qu'à des primitives bloquantes ne sont que très légèrement détériorées.

Perspectives

Ces travaux, en faisant collaborer l'ordonnanceur de threads et la bibliothèque de communication pour une exploitation efficace des ressources, ouvrent de nouvelles perspectives à court, moyen et long termes.

Parallélisation avancée des traitements des communications. À court terme, il serait intéressant de poursuivre la parallélisation des bibliothèques de communication. En effet, si le mécanisme de tâches proposé et son utilisation pour exploiter différents cœurs ont permis de paralléliser certains traitements existants, d'autres opérations pourraient exploiter ce mécanisme. L'enregistrement mémoire permettant de transférer des données par DMA est une opération coûteuse et qu'il serait intéressant de traiter en arrière-plan en utilisant une tâche. L'impact des communications sur les performances des applications serait alors réduit. Certains traitements gourmands en ressources tels que la compression de données – qui peut s'avérer utile pour des réseaux longues distances, par exemple sur une grille de calcul – pourraient être effectués en arrière-plan en utilisant des tâches. Ainsi, à la manière de ADOC [Jea05] qui compresse ou non les messages en fonction de l'occupation du réseau, la compression pourrait être décidée par l'occupation des processeurs : lorsqu'un processeur est inutilisé et que le réseau n'est pas prêt, la compression des données serait effectuée sans surcoût pour l'application grâce à l'exploitation des cœurs inactifs. Les applications du système d'exportation de tâches sont nombreuses dans une bibliothèque de communication qui peut s'appuyer dessus pour implémenter des mécanismes de tolérance aux pannes – par exemple la retransmission de données, le calcul d'une somme de contrôle, etc. – ou pour gérer les transferts de données en mémoire partagée. D'une manière plus spécifique aux bibliothèques de communication capables d'optimiser les flux de communication telles que NEWMADELEINE, il serait intéressant de d'évaluer les différentes combinaisons d'optimisation afin de choisir la plus adaptée. Par exemple, NEWMADELEINE effectue ses optimisations à la volée : lorsqu'une carte réseau se libère, la stratégie d'optimisation choisi un assemblage de messages pour former un paquet de données à transmettre. Le calcul de cet assemblage doit se faire rapidement afin de réduire le coût des communications, et il n'est donc pas possible d'évaluer toutes les combinaisons possibles. Il serait donc intéressant d'exploiter les cœurs inutilisés pour pré-calculer les optimisations applicables. Ainsi, lorsqu'une carte réseau se libère, une large gamme d'optimisations est disponible et il suffit de quelques adaptations pour soumettre le paquet de données le plus adapté au réseau.

Auto-adaptation des mécanismes. Les mécanismes que nous avons proposés dans ce document voient leur efficacité varier en fonction des applications. Par exemple, le mécanisme de verrouillage à gros grain est efficace pour les applications utilisant peu les accès concurrents à la bibliothèque de communication, mais lorsque de nombreux threads communiquent, ce mécanisme engendre un surcoût important du fait de la taille de la section critique et de la forte contention. D'une manière similaire, l'efficacité du mécanisme d'attente mixte pour une application dépend de la durée pendant laquelle la bibliothèque de communication scrute le réseau. Une sélection automatique des différents mécanismes – type de verrouillage, durée de scrutation, etc. – permettrait donc de bénéficier au mieux de ces mécanismes de manière transparente. Le compilateur est, dans certains cas, une source d'informations qui permettrait d'affiner les mécanismes proposés. Ainsi, en fonction du comportement de l'application et des schémas de communication employés, la durée de scrutation pourrait être adaptée afin de réduire le surcoût des changements de contexte ou de redonner la main rapidement à un autre thread de l'application. Une autre source d'information fiable pour ce type d'adaptation est le développeur de l'application. Celui-ci pourrait donner des indications sur le schéma de communication ou sur le comportement du programme.

En s'appuyant sur des informations provenant du compilateur, de l'application ou d'une analyse de l'historique des communications, certains mécanismes pourraient être sélectionnés en fonction d'une prédiction des événements à venir.

Vers une gestion de tous les types d'entrées/sorties. Les travaux que nous avons présentés tout au long de ce document se sont focalisés sur le traitement des communications. Les mécanismes proposés n'en sont pas moins inadaptés aux autres types d'entrées/sorties employés dans les applications de calcul scientifique. Les accès aux disques – que ce soit des accès locaux ou des accès à un système de fichiers distant de type PVFS ou LUSTRE – posent les même types de problématiques que les communications réseau. Il serait donc intéressant d'exploiter les mécanismes que nous avons proposés dans une bibliothèque d'entrées/sorties sur disques. Ce type de bibliothèques bénéficierait alors naturellement de la gestion du multi-threading et pourrait exploiter simplement les multiples cœurs disponibles. Une telle intégration permettrait de plus de maîtriser l'ensemble des entrées/sorties d'un processus – communications réseau et entrées/sorties sur disque – et donc d'avoir une vision globale des interactions entre le processus local et le monde extérieur. Une telle maîtrise des entrées/sorties permettrait alors d'optimiser les différents flux sortants – requêtes d'accès au disque local, ou communication réseau provenant de l'application ou destinée à un serveur de fichiers distant – de manière à minimiser leur impact sur les performances des applications. La prise en compte indifférenciées des multiples types de flux augmenterait également le nombre d'opportunités d'optimisation.

Extension à plusieurs processus. Comme nous l'avons vu dans la section 2.4, l'exploitation des grappes de calcul a évolué au cours des dernières années en passant d'un modèle uniquement basé sur le passage de messages à un modèles utilisant conjointement le standard MPI et le multi-threading. Si les approches hybrides permettent de s'adapter aux machines hiérarchiques constituant les grappes de calcul modernes, la granularité de l'approche est problématique. En effet, l'utilisation d'un seul processus MPI par nœud n'offre pas toujours des performances optimales, notamment lorsque les nœuds sont constitués de machines NUMA. Du fait des coûts de communication entre les différents bancs mémoire d'une telle machine, il peut s'avérer plus performant de lancer plusieurs processus par machine, par exemple en exploitant chaque nœud NUMA avec un processus différent. Ce type d'approche hybride soulève alors des problèmes puisque chaque processus n'a qu'une vision partielle de l'état de la machine. Il devient donc difficile d'évaluer l'activité des cartes réseau ou d'équilibrer la charge entre les différents cœurs de la machine. Il serait donc utile de disposer d'un mécanisme offrant une vue globale de l'ensemble de la machine et permettant d'équilibrer la charge sur les différents cœurs disponibles. La création d'un module noyau capable d'optimiser l'ensemble des entrées/sorties de la machine ou la synchronisation des différents processus grâce à un segment de mémoire partagée sont des solutions qu'il serait intéressant d'étudier. La vision globale de l'état de la machine permettrait alors à la bibliothèque de communication de prendre en compte l'activité des cartes réseau ou d'optimiser les flux de communication provenant des différents processus. La répartition de la charge sur l'ensemble de la machine permettrait l'exploitation des cœurs laissés vacants par n'importe quel processus pour faire progresser les communications ou pour traiter les tâches soumises par la bibliothèque de communication.

Vers une gestion de tous les événements du système. Alors que les modèles de programmation mélangeant MPI et multi-threading se développent de plus en plus, les grappes de calcul commencent à incorporer des accélérateurs tels que des GPGPU. L'utilisation de ces technologies apporte une puissance de calcul considérable pour certaines classes de problèmes et l'usage combiné du passage de

message et de ces accélérateurs est à étudier. Outre les problèmes liés à l'utilisation conjointe du multithreading et de communications réseaux, ce type de modèle de programmation nécessite une gestion explicite des transferts de données entre la mémoire principale et la mémoire de l'accélérateur. Afin de simplifier la gestion des données ainsi que la répartition des tâches de calcul sur les unités de calcul, des bibliothèques spécialisées telles que StarPU [ATNW09] ont été développées. La programmation d'un accélérateur étant fortement similaire à l'utilisation d'une carte réseau, de nombreuses problématiques des bibliothèques de communication se retrouvent dans ces bibliothèques spécialisées. Par exemple, les transferts de données entre la mémoire et une carte graphique s'apparentent à l'envoi d'un message sur le réseau. Il serait donc intéressant de transposer les mécanismes que nous avons proposés afin de les appliquer à une bibliothèque dédiée aux architectures hétérogènes. Le modèle fortement événementiel de ce type de bibliothèque donne l'opportunité de gérer une grande partie des traitements de l'ordonnanceur tels que le déclenchement d'un transfert de données, la détection de la terminaison d'une copie ou la détection d'événements provenant de l'accélérateur. Ces opérations pourraient ainsi bénéficier des mécanismes que nous avons proposé. D'une manière plus générale, l'ensemble des événements d'un processus – que ce soit au sein d'un ordonnanceur de threads, d'une bibliothèque d'entrées/sortie, d'une interface de visualisation, etc. – pourraient s'appuyer sur une bibliothèque comme PIOMan afin de gérer de manière transparente les événements et d'exploiter l'ensemble des cœurs d'une machine.

Annexe A

Interfaces de programmation de PIOMAN

Sommaire

Cette annexe regroupe une partie de l'interface de programmation de PIOMAN.

A.1 Interface de détection des événements

Types

- typedef struct **piom_req *piom_req_t**
  Type d'une requête.
- typedef struct **piom_server *piom_server_t**
  Type d'un serveur.
- enum **piom_op_t** {
  PIOM_FUNCTYPE_POLL_POLLONE,
  PIOM_FUNCTYPE_POLL_GROUP,
  PIOM_FUNCTYPE_POLL_POLLANY,
  PIOM_FUNCTYPE_BLOCK_WAITONE,
  PIOM_FUNCTYPE_BLOCK_WAITONE_TIMEOUT,
  PIOM_FUNCTYPE_BLOCK_GROUP,
  PIOM_FUNCTYPE_BLOCK_WAITANY,
  PIOM_FUNCTYPE_BLOCK_WAITANY_TIMEOUT}
  Types d'une fonction de rappel.
- typedef int (**piom_callback_t**) (**piom_server_t** server, **piom_op_t** op, **piom_req_t** req, **int** nb_ev, **int** option)
  Prototype d'une fonction de rappel.

Fonctions

- void **piom_server_init** (**piom_server_t** server, **char** *name)
  Initilise un serveur.
- int **piom_server_add_callback** (**piom_server_t** server, **piom_op_t** op, **piom_pcallback_t** func)
  Ajoute un nouveau type de fonction de rappel à un serveur.
- int **piom_server_start** (**piom_server_t** server)
  Démarre un serveur.
- int **piom_server_stop** (**piom_server_t** server)
  Arrête un serveur.

- int **piom_req_init** (**piom_req_t** req)
  Initialise une requête.
- int **piom_req_free** (**piom_req_t** req)
  Libère une requête.
- int **piom_req_submit** (**piom_server_t** server, **piom_req_t** req)
  Soumet une requête à PIOMAN. *À partir de maintenant, la requête peut être détectée à n'importe quel moment*
- int **piom_req_cancel** (**piom_req_t** req, **int** ret_code)
  Annule une requête. Les threads en attente de la requête sont réveillés et `ret_code` *est retourné.*

- int **piom_req_wait** (**piom_req_t** req, **piom_wait_t** wait, **piom_time_t** timeout)
  Attend la fin d'une requête.
- int **piom_test** (**piom_req_t** req)
  Teste la terminaison d'une requête.
- int **piom_server_wait** (**piom_server_t** server, **piom_time_t** timeout)
  Attend la fin de n'importe quelle requête d'un serveur.
- int **piom_wait** (**piom_server_t** server, **piom_req_t** req, **piom_wait_t** wait, **piom_time_t** timeout)
  Initialise une requête, la soumet à un serveur et attend l'occurence de l'événement.

A.2 Interface d'attente d'un événement

Types

- typedef struct **piom_sem_t**
  Type d'un sémaphore.
- typedef struct **piom_cond_t**
  Type d'une condition.

Fonctions

- void **piom_sem_P** (**piom_sem_t** *sem)
  Décrémente un sémaphore.
- void **piom_sem_V** (**piom_sem_t** *sem)
  Incrémente un sémaphore.
- void **piom_sem_init** (**piom_sem_t** *sem, **int** initial)
  Initialise un sémaphore.

– void **piom_cond_wait** (**piom_cond_t** *cond, **uint8_t** mask)
Attend que la condition `cond` *vérifie un masque de bits.*
– void **piom_cond_signal** (**piom_cond_t** *cond, **uint8_t** mask)
Modifie le masque de bit d'une condition.
– int **piom_cond_testpiom_cond_t** *cond, **uint8_t** mask)
Teste la valeur d'une condition.
– void **piom_cond_initpiom_cond_t** *cond, **uint8_t** initial)
Initialise une condition.

A.3 Interface du mécanisme d'exportation de tâches

Types

– typedef **int** (**piom_ltask_func**) (**void** *arg)
Prototype d'une fonction de rappel.
– struct **piom_ltask**
Type d'une tâche.
– typedef **piom_vpset_t**
Type d'un masque de processeurs.

Fonctions

– void **piom_init_ltasks** ()
Initialise le mécanisme d'exportation de tâches.
– void **piom_exit_ltasks** ()
Termine le mécanisme d'exportation de tâches.
– void **piom_ltask_init** (**struct piom_ltask** *task)
Initialise une tâche vide.
– void **piom_ltask_set_func** (**struct piom_ltask** *task, **piom_ltask_func** * func_ptr)
Spécifie la fonction de rappel à appeler pour une tâche.
– void **piom_ltask_set_data** (**struct piom_ltask** *task, **void** *data_ptr)
Spécifie l'argument à donner lors de l'exécution d'une tâche.
– void **piom_ltask_set_options** (**struct piom_ltask** *task, **piom_ltask_option_t** option)
Spécifie les options d'une tâche.
– void **piom_ltask_set_vpmask** (**struct piom_ltask** *task, **piom_vpset_t** mask)
Spécifie le masque de processeurs à appliquer à une tâche.
– void **piom_ltask_create** (**struct piom_ltask** *task, **piom_ltask_func** * func_ptr, **void** *data_ptr, **piom_ltask_option_t** options, **piom_vpset_t** vp_mask)
Crée une tâche et spécifie ses caractéristique. Équivalent à l'initialisation de la tâche, puis la spécification des différents paramètres un par un.
– void **piom_ltask_submit** (**struct piom_ltask** *task)
Soumet une tâche.
– void **piom_ltask_wait** (**struct piom_ltask** *task)
Attend la terminaison d'une tâche.
– int **piom_ltask_test** (**struct piom_ltask** *task)
Teste la terminaison d'une tâche.

Bibliographie

[AAC+03] Almasi (G.), Archer (C.), Castanos (J.), Gupta (M.), Martorell (X.), Moreira (J.), Gropp (W.), Rus (S.) et Toonen (B.), « MPI on BlueGene/L : Designing an efficient general purpose messaging solution for a large cellular system », *Lecture Notes in Computer Science*, 2003, p. 352–361.

[ABD+02] Aumage (O.), Bougé (L.), Denis (A.), Eyraud (L.), Méhaut (J.-F.), Mercier (G.), Namyst (R.) et Prylli (L.), « A Portable and Efficient Communication Library for High-Performance Cluster Computing », *Cluster Computing*, vol. 5, n° 1, 2002, p. 43–54.

[ABLL91] Anderson (T. E.), Bershad (B. N.), Lazowska (E. D.) et Levy (H. M.), « Scheduler Activations : effective kernel support for the user-level management of parallelism », dans *SOSP '91 : Proceedings of the thirteenth ACM symposium on Operating systems principles*, p. 95–109, New York, NY, USA, 1991. ACM.

[ABMN07] Aumage (O.), Brunet (E.), Mercier (G.) et Namyst (R.), « High-Performance Multi-Rail Support with the NewMadeleine Communication Library », dans *HCW 2007 : the Sixteenth International Heterogeneity in Computing Workshop*, 2007.

[ACD+08] Ayguade (E.), Copty (N.), Duran (A.), Hoeflinger (J.), Lin (Y.), Massaioli (F.), Su (E.), Unnikrishnan (P.) et Zhang (G.), « A proposal for task parallelism in OpenMP », *Lecture Notes in Computer Science*, vol. 4935, 2008, p. 1–12.

[AMN01] Aumage (O.), Mercier (G.) et Namyst (R.), « MPICH-Madeleine : a True Multi-Protocol MPI for High-Performance Networks », dans *Proceeding of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, p. 51, San Francisco, avril 2001. IEEE.

[Ang01] Ang (B.), « An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC. HP-Labs ». Rapport technique, HPL-2001-8, 2001.

[ATNW09] Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.), « StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures », dans *Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science*, vol. 5704 (coll. *Lecture Notes in Computer Science*), p. 863–874, Delft, The Netherlands, août 2009. Springer.

[AUW08] Appavoo (J.), Uhlig (V.) et Waterland (A.), « Project Kittyhawk : building a global-scale computer : Blue Gene/P as a generic computing platform », *SIGOPS Opererating Systems Review*, vol. 42, n° 1, 2008, p. 77–84.

[BAPM] Beecroft (J.), Addison (D.), Petrini (F.) et McLaren (M.), « QsNet-II : an interconnect for supercomputing applications », dans *the Proceedings of Hot Chip' 03*.

[Bar09] Barrett (B.), « Problem with OpenMPI (MX btl and mtl) and threads ». Open MPI user's mailing list, 2009. http ://www.open-mpi.org/community/lists/users/2009/06/9601.php.

[BBB+91] Bailey (D.), Barszcz (E.), Barton (J.), Browning (D.), Carter (R.), Dagum (L.), Fatoohi (R.), Frederickson (P.), Lasinski (T.), Schreiber (R.) *et al.*, « The NAS parallel benchmarks », *International Journal of High Performance Computing Applications*, vol. 5, n° 3, 1991, p. 63.

[BBG+08] Balaji (P.), Buntinas (D.), Goodell (D.), Gropp (W.) et Thakur (R.), « Toward Efficient Support for Multithreaded MPI Communication », dans *Recent Advances in Parallel Virtual Machine and Message Passing Interface : 15th European PVM/MPI Users' Group Meeting, 2008*, 2008.

[BBH+97] Bal (H.), Bhoedjang (R.), Hofman (R.), Jacobs (C.), Langendoen (K.), Rühl (T.) et Verstoep (K.), « Performance of a high-level parallel language on a high-speed network », *Journal of Parallel and Distributed Computing*, vol. 40, n° 1, 1997, p. 49–64.

[BCF+95] Boden (N. J.), Cohen (D.), Felderman (R. E.), Kulawik (A. E.), Seitz (C. L.), Seizovic (J. N.) et Su (W.-K.), « Myrinet : A Gigabit-per-Second Local Area Network », *IEEE Micro*, vol. 15, n° 1, 1995, p. 29–36.

[BDH+08] Barker (K.), Davis (K.), Hoisie (A.), Kerbyson (D.), Lang (M.), Pakin (S.) et Sancho (J.), « Entering the petaflop era : the architecture and performance of Roadrunner », dans *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press Piscataway, NJ, USA, 2008.

[BDN02] Bougé (L.), Danjean (V.) et Namyst (R.), « Improving Reactivity to I/O Events in Multithreaded Environments Using a Uniform, Scheduler-Centric API », dans *Euro-Par '02 : Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, p. 605–614, London, UK, 2002. Springer-Verlag.

[BDT+08] Broquedis (F.), Diakhaté (F.), Thibault (S.), Aumage (O.), Namyst (R.) et Wacrenier (P.-A.), « Scheduling Dynamic OpenMP Applications over Multicore Architectures », dans *OpenMP in a New Era of Parallelism, 4th International Workshop on OpenMP, IWOMP 2008*, vol. 5004 (coll. *Lecture Notes in Computer Science*), p. 170–180, West Lafayette, IN, mai 2008. Springer.

[BGSP94] Bruening (U.), Giloi (W. K.) et Schroeder-Preikschat (W.), « Latency hiding in message-passing architectures », dans *Proceedings of the 8th International Symposium on Parallel Processing*, p. 704–709, Washington, DC, USA, 1994. IEEE Computer Society.

[BMG07] Buntinas (D.), Mercier (G.) et Gropp (W.), « Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem », *Parallel Computing, Selected Papers from EuroPVM/MPI 2006*, vol. 33, n° 9, septembre 2007, p. 634–644.

[BNM98] Bougé (L.), Namyst (R.) et Méhaut (J.-F.), « Madeleine : An efficient and portable communication interface for rpc-based multithreaded environments », dans *PACT '98 : Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, p. 240, Washington, DC, USA, 1998. IEEE Computer Society.

[BRH+93] Bhoedjang (R.), Ruhl (T.), Hofman (R.), Langendoen (K.), Bal (H.) et Kaashoek (F.), « Panda : A portable platform to support parallel programming languages », dans *In Symposium on Experiences with Distributed and Multiprocessor Systems IV*, 1993.

[BRU05] Brightwell (R.), Riesen (R.) et Underwood (K.), « Analyzing the impact of overlap, offload, and independent progress for message passing interface applications », *International Journal of High Performance Computing Applications*, vol. 19, n° 2, 2005, p. 103.

[Bru08a] Brunet (E.), « Newmadeleine : ordonnancement et optimisation de schémas de communication haute performance », *Technique et Science Informatiques*, vol. 27, n° 3-4/2008, 2008, p. 293–316.

[Bru08b] Brunet (É.), *Une approche dynamique pour l'optimisation des communications concurrentes sur réseaux haute performance*. PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, décembre 2008.

[BU04] Brightwell (R.) et Underwood (K.), « An analysis of the impact of MPI overlap and independent progress », dans *Proceedings of the 18th annual international conference on Supercomputing*, p. 298–305. ACM New York, NY, USA, 2004.

[Bul] Bull, « MPIBull2 ». `http://www.bull.com`.

[Cas05] Cashin (E. L.), « Kernel korner : ATA over ethernet : putting hard drives on the lan », *Linux Journal*, vol. 2005, n° 134, 2005, p. 10.

[CC97] Chiola (G.) et Ciaccio (G.), « GAMMA : a Low-cost Network of Workstations Based on Active Messages », dans *In Proceedings of Euromicro PDP'97*. IEEE Computer Society, 1997.

[CE00] Cappello (F.) et Etiemble (D.), « MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks », dans *Supercomputing, ACM/IEEE 2000 Conference*, p. 12–12, 2000.

[CP08] Chevalier (C.) et Pellegrini (F.), « PT-Scotch : A tool for efficient parallel graph ordering », *Parallel Computing*, vol. 34, n° 6-8, 2008, p. 318–331.

[DBL97] Dubnicki (C.), Bilas (A.) et Li (K.), « Design and Implementation of Virtual Memory-Mapped Communication on Myrinet », dans *IPPS '97 : Proceedings of the 11th International Symposium on Parallel Processing*, p. 388, Washington, DC, USA, 1997. IEEE Computer Society.

[DDW06] Dalessandro (D.), Devulapalli (A.) et Wyckoff (P.), « iWarp protocol kernel space software implementation », *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 25-29 April 2006, p. 8 pp.–.

[DNR00a] Danjean (V.), Namyst (R.) et Russel (R.), « Linux kernel activations to support multithreading », dans *18 th IASTED International Conference on Applied Informatics (AI'00)*, p. 718–723. Citeseer, 2000.

[DNR00b] Danjean (V.), Namyst (R.) et Russel (R.), « Linux kernel activations to support multithreading », dans *18 th IASTED International Conference on Applied Informatics (AI'00)*, p. 718–723, 2000.

[Dol] Dolphin Interconnect, « SISCI Documentation and Library ». `http://www.dolphinics.no`.

[EBBV95] Eicken (T. V.), Basu (A.), Buch (V.) et Vogels (W.), « U-net : a user-level network interface for parallel and distributed computing », *ACM Operating Systems Review, SIGOPS, Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, vol. 29, n° 5, 1995, p. 303–316.

[ECGS92] von Eicken (T.), Culler (D. E.), Goldstein (S. C.) et Schauser (K. E.), « Active messages : a mechanism for integrated communication and computation », dans *ISCA '92 : Proceedings*

of the 19th annual international symposium on Computer architecture, p. 256–266, New York, NY, USA, 1992. ACM.

[EE00] Evans (J.) et Elischer (J.), « Kernel-scheduled entities for FreeBSD », 2000.

[FK97] Foster (I.) et Kesselman (C.), « Globus : A metacomputing infrastructure toolkit », *International Journal of High Performance Computing Applications*, vol. 11, n° 2, 1997, p. 115.

[FKT94] Foster (I.), Kesselman (C.) et Tuecke (S.), « The Nexus task-parallel runtime system », dans *1st International Workshop on Parallel Processing (IWPP'94)*, p. 457–462, 1994.

[FKT96] Foster (I.), Kesselman (C.) et Tuecke (S.), « The Nexus approach to integrating multithreading and communication », *Journal of Parallel and Distributed Computing*, vol. 37, n° 1, 1996, p. 70–82.

[Gog08] Goglin (B.), « Design and Implementation of Open-MX : High-Performance Message Passing over generic Ethernet hardware », dans *CAC 2008 : Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, avril 2008. IEEE.

[GT07] Gropp (W.) et Thakur (R.), « Thread-safety in an MPI implementation : Requirements and analysis », *Parallel Computing*, vol. 33, n° 9, 2007, p. 595–604.

[GWS05] Graham (R. L.), Woodall (T. S.) et Squyres (J. M.), « Open MPI : A Flexible High Performance MPI », dans *The 6th Annual International Conference on Parallel Processing and Applied Mathematics*, 2005.

[HBK06] Held (J.), Bautista (J.) et Koehl (S.), « From a few cores to many : A tera-scale computing research overview », *Research at Intel white paper*, 2006.

[HL08] Hoefler (T.) et Lumsdaine (A.), « Message Progression in Parallel Computing-To Thread or not to Thread », dans *Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society*, 2008.

[HP03] Hennessy (J. L.) et Patterson (D. A.), *Computer Architecture : A Quantitative Approach*. Morgan Kaufman, 3e édition, 2003.

[HSGL08] Hoefler (T.), Schellmann (M.), Gorlatch (S.) et Lumsdaine (A.), « Communication Optimization for Medical Image Reconstruction Algorithms », dans *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, vol. LNCS 5205, p. 75–83. Springer, Sep. 2008.

[HSJ+06] Huang (W.), Santhanaraman (G.), Jin (H.-W.), Gao (Q.) et D. K. Panda (D. K. x.), « Design of High Performance MVAPICH2 : MPI2 over InfiniBand », dans *CCGRID '06 : Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, p. 43–48, Washington, DC, USA, 2006. IEEE Computer Society.

[Inf00] InfiniBand Trade Association, *InfiniBand Architecture Specification : Release 1.0*. InfiniBand Trade Association, 2000.

[JD08] Jiang (J.) et DeSanti (C.), « The role of FCoE in I/O consolidation », dans *Proceedings of the 2008 International Conference on Advanced Infocomm Technology*. ACM New York, NY, USA, 2008.

[Jea05] Jeannot (E.), « Improving Middleware Performance with AdOC : An Adaptive Online Compression Library for Data Transfer », dans *IPDPS '05 : Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[KLMO91] Karlin (A.), Li (K.), Manasse (M.) et Owicki (S.), « Empirical studies of competitve spinning for a shared-memory multiprocessor », *ACM SIGOPS Operating Systems Review*, vol. 25, n° 5, 1991, p. 41–55.

[KMAC03] Keltcher (C.), McGrath (K.), Ahmed (A.) et Conway (P.), « The AMD Opteron processor for multiprocessor servers », *IEEE Micro*, vol. 23, n° 2, 2003, p. 66–76.

[KSP09] Koop (M.), Sridhar (J.) et Panda (D. K.), « TupleQ : Fully-Asynchronous and Zero-Copy MPI over InfiniBand », dans *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009)*, May 2009.

[Lam05] Lameter (C.), « Effective Synchronization on Linux/NUMA Systems », dans *Proceedings of the May 2005 Gelato Federation Meeting, San Jose, CA*, 2005.

[LCW+03] Liu (J.), Chandrasekaran (B.), Wu (J.), Jiang (W.), Kini (S.), Yu (W.), Buntinas (D.), Wyckoff (P.) et Panda (D.), « Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics », dans *Supercomputing, 2003 ACM/IEEE Conference*, p. 58–58, 2003.

[LRBB96] Langendoen (K.), Romein (J.), Bhoedjang (R.) et Bal (H.), « Integrating polling, interrupts, and thread management », dans *FRONTIERS '96 : Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, p. 13, Washington, DC, USA, 1996. IEEE Computer Society.

[mar07] *Marcel : A POSIX-compliant thread library for hierarchical multiprocessor machines*, 2007. `http://runtime.futurs.inria.fr/marcel/`.

[MCO09] Mercier (G.) et Clet-Ortega (J.), « Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments », dans *EuroPVM/MPI*, vol. 5759 (coll. *Lecture Notes in Computer Science*), p. 104–115, Espoo, Finland, septembre 2009. Springer.

[Mes94] Message Passing Interface Forum, « MPI : a message-passing interface standard ». Technical Report n° UT-CS-94-230, 1994.

[Mes96] Message Passing Interface Forum, « MPI-2 : extensions to the message-passing interface », *University of Tennessee, Knoxville*, 1996.

[MG07] Moreaud (S.) et Goglin (B.), « Impact of NUMA Effects on High-Speed Networking with Multi-Opteron Machines », dans *The 19th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2007)*, Cambridge, Massachussetts, novembre 2007.

[mpi07] *MPICH-2 Home Page*, 2007. `http://www.mcs.anl.gov/mpi/mpich/`.

[MTBB09] Mercier (G.), Trahay (F.), Buntinas (D.) et Brunet (É.), « NewMadeleine : An Efficient Support for High-Performance Networks in MPICH2 », dans *Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, Rome, Italy, mai 2009. IEEE Computer Society Press.

[Myr] Myricom Inc., « MPICH2-MX ». `http://www.myri.com/scs/download-mpichmx.html`.

[Myr95] Myricom Inc., « `http://www.myri.com/` », 1995.

[Myr03] Myricom Inc., « Myrinet EXpress (MX) : A High Performance, Low-level, Message-Passing Interface for Myrinet », 2003. `http://www.myri.com/scs/`.

[MZOR02] MacCabe (A. B.), Zhu (W.), Otto (J.) et Riesen (R.), « Experience in Offloading Protocol Processing to a Programmable NIC », dans *CLUSTER '02 : Proceedings of the IEEE*

International Conference on Cluster Computing, p. 67, Washington, DC, USA, 2002. IEEE Computer Society.

[Nak08] Nakashima (H.), « T2k open supercomputer : Inter-university and inter-disciplinary collaboration on the new generation supercomputer », *Informatics Education and Research for Knowledge-Circulating Society, 2008. ICKS 2008. International Conference on*, Jan. 2008, p. 137–142.

[NM95a] Namyst (R.) et Méhaut (J.-F.), « PM2 : Parallel Multithreaded Machine ; A computing environment for distributed architectures », 1995.

[NM95b] Namyst (R.) et Méhaut (J.-F.), *Marcel : Une bibliothèque de processus légers*. LIFL, Univ. Sciences et Techn. Lille, 1995.

[Opea] Open Fabrics Alliance, « `http://www.openfabrics.org/` ».

[Opeb] OpenMP Forum, « OpenMP ». `http://www.openmp.org/`.

[Pfi01] Pfister (G.), « Aspects of the InfiniBand Architecture », dans *Proceeding of the IEEE International Conference on Cluster Computing, 2001*, p. 369–371, 2001.

[PKC97] Pakin (S.), Karamcheti (V.) et Chien (A. A.), « Fast Messages : Efficient, portable communication for workstation clusters and MPPs », *IEEE Concurrency*, vol. 5, n° 2, /1997, p. 60–73.

[PP02] Pakin (S.) et Pant (A.), « VMI 2.0 : A Dynamically Reconfigurable Messaging Layer for Availability, Usability, and Management », dans *The 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, 2002.

[PT98] Prylli (L.) et Tourancheau (B.), « BIP : a new protocol designed for high performance networking on Myrinet », *Lecture Notes in Computer Science*, vol. 1388, 1998, p. 472–485.

[Qua] Quadrics Ltd., « Quadrics MPI ». `http://www.quadrics.com/`.

[Qua03] Quadrics Ltd., « Elan Programming Manual », 2003. `http://www.quadrics.com/`.

[RA08] Rashti (M.) et Afsahi (A.), « Improving Communication Progress and Overlap in MPI Rendezvous Protocol over RDMA-enabled Interconnects », dans *22nd International Symposium on High Performance Computing Systems and Applications, 2008 (HPCS 2008)*, p. 95–101, 2008.

[SB01] Smith (L.) et Bull (M.), « Development of mixed mode MPI/OpenMP applications », *Scientific Programming*, vol. 9, n° 2, 2001, p. 83–98.

[SBKD06] Sancho (J.), Barker (K.), Kerbyson (D.) et Davis (K.), « Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications », dans *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM New York, NY, USA, 2006.

[Sea08] Seager (M.), « The ASC Sequoia Programming Model ». Rapport technique, LLNL-SR-406177, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2008.

[Sin08] Singhal (R.), « Inside Intel Next Generation Nehalem Microarchitecture », dans *Intel Developer Forum, Shanghai, China*, 2008.

[SJCP06] Sur (S.), Jin (H.), Chai (L.) et Panda (D.), « RDMA read based rendezvous protocol for MPI over InfiniBand : design alternatives and benefits », dans *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, p. 32–39. ACM New York, NY, USA, 2006.

[SMG96] Snell (Q. O.), Mikler (A. R.) et Gustafson (J. L.), « Netpipe : A network protocol independent performace evaluator », dans *In Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[SPH96] Skjellum (A.), Protopopov (B.) et Hebert (S.), « A Thread Taxonomy for MPI », dans *Proceedings of the Second MPI Developers Conference*. IEEE Computer Society Washington, DC, USA, 1996.

[Squ09] Squyres (J.), « Blocking communication a thread better thenasynchronous progress ? ». Open MPI user's mailing list, 2009. http ://www.open-mpi.org/community/lists/users/2009/08/10446.php.

[SSB+08] Shet (A.), Sadayappan (P.), Bernholdt (D.), Nieplocha (J.) et Tipparaju (V.), « A framework for characterizing overlap of communication and computation in parallel applications », *Cluster Computing*, vol. 11, n° 1, 2008, p. 75–90.

[SWG+06] Shipman (G. M.), Woodall (T. S.), Graham (R. L.), Maccabe (A. B.) et Bridges (P. G.), « InfiniBand Scalability in Open MPI », dans *Proceedings of the 20th IEEE Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 2006.

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

[THIS97] Tezuka (H.), Hori (A.), Ishikawa (Y.) et Sato (M.), « PM : An operating system coordinated high performance communication library », *Lecture Notes in Computer Science*, vol. 1225, 1997, p. 708–717.

[TOP] TOP500, « TOP500 Supercomputing Sites ». `http://www.top500.org/`.

[TSH+00] Takahashi (T.), Sumimoto (S.), Hori (A.), Harada (H.) et Ishikawa (Y.), « PM2 : High performance communication middleware for heterogeneous network environments », dans *Supercomputing, ACM/IEEE 2000 Conference*, p. 52–53, 2000.

[VHR+08] Vangal (S.), Howard (J.), Ruhl (G.), Dighe (S.), Wilson (H.), Tschanz (J.), Finan (D.), Singh (A.), Jacob (T.), Jain (S.) *et al.*, « An 80-tile sub-100-w teraflops processor in 65-nm cmos », *IEEE Journal of Solid-State Circuits*, vol. 43, n° 1, 2008, p. 29–41.

[VRC+03] Velusamy (V.), Rao (C.), Chakravarthi (S.), Neelamegam (J.), Chen (W.), Verma (S.) et Skjellum (A.), « Programming the Infiniband network architecture for high performance message passing systems », dans *ISCA 16th International Conference on Parallel and Distributed Computing Systems (PDCS-2003)*. Citeseer, 2003.

[WADJ+05] Worley (P.), Alam (S.), Dunigan Jr (T.), Fahey (M.) et Vetter (J.), « Comparative Analysis of Interprocess Communication on the X1, XD1, and XT3 », dans *Proceedings of the 47th Cray User Group Conference*, 2005.

[WGC+04] Woodall (T.), Graham (R.), Castain (R.), Daniel (D.), Sukalski (M.), Fagg (G.), Gabriel (E.), Bosilca (G.), Angskun (T.), Dongarra (J.), Squyres (J.), Sahay (V.), Kambadur (P.), Barrett (B.) et Lumsdaine (A.), « Open MPI's TEG Point-to-Point Communications Methodology : Comparison to Existing Implementations », dans *Proceedings, 11th European PVM/MPI Users' Group Meeting*, p. 105–111, Budapest, Hungary, September 2004.

[Wil02] Williams (N.), « An Implementation of Scheduler Activations on the NetBSD Operating System », dans *Proceedings of the FREENIX Track : 2002 USENIX Annual Technical Conference table of contents*, p. 99–108. USENIX Association Berkeley, CA, USA, 2002.

[WJPR04] Wagner (A.), Jin (H.), Panda (D.) et Riesen (R.), « NIC-based offload of dynamic user-defined modules for Myrinet clusters », dans *2004 IEEE International Conference on Cluster Computing*, p. 205–214, 2004.

[WRRF] Woodacre (M.), Robb (D.), Roe (D.) et Feind (K.), « The SGI® Altix TM 3000 Global Shared-Memory Architecture ».

[YWGP05] Yu (W.), Woodall (T.), Graham (R.) et Panda (D.), « Design and Implementation of Open MPI over Quadrics/Elan4 », *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, April 2005.