N° d'ordre : 3721

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Élisabeth Brunet**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Une approche dynamique pour l'optimisation des communications concurrentes sur réseaux haute performance

---

Soutenue le : 8 Décembre 2008

Après avis des rapporteurs :

Mme Christine Morin ...	Directeur de Recherche INRIA
M. Jean-François Méhaut	Professeur des Universités

Devant la commission d'examen composée de :

M. Franck Cappello	Directeur de Recherche INRIA	Examinateur
M. Olivier Glück	Maître de Conférences	Examinateur
M. Jean-François Méhaut	Professeur des Universités	Rapporteur
Mme Christine Morin ...	Directeur de Recherche INRIA	Rapporteur
M. Raymond Namyst ...	Professeur des Universités	Directeur de thèse
M. Jean Roman	Professeur des Universités	Président du jury

Citius, altius, fortius - Plus vite, plus haut, plus fort
Devise Olympique

Remerciements

Me voici rendue quelques semaines après cette longue, impitoyable, mais cependant trépidante aventure qu'est le doctorat. Mon sentiment de fatigue *permanente* acquis pendant le marathon de la rédaction est à présent dissout et c'est en toute sérénité que je vous présente aujourd'hui mes travaux.

Tout cela n'aurait été possible sans l'aide, le support et la confiance de bon nombre de gens. Ainsi, je tiens à remercier en premier lieu mon directeur de thèse. Raymond merci ! Merci de m'avoir fait confiance malgré mon orientation dissidente de départ, de m'avoir encadrée pour le meilleur et pour le pire ;) Ensemble, j'ai le sentiment que nous avons partagé un agréable bout de chemin qui, je l'espère, devrait se poursuivre très bientôt. Je remercie ensuite très respectueusement mon jury. Mes rapporteurs Christine Morin et Jean-François Méhaut. J'ai eu récemment l'occasion de relire leur rapport respectif durant mes préparatifs aux candidatures aux postes de maître de conférence. Avec le recul, je me rends d'autant plus compte de l'intérêt qu'ils ont porté à mes travaux et du niveau de leur expertise. Je remercie chaque membre du jury pour leur attention tout au long de la soutenance, ainsi que pour leurs questions et ce qu'elles ont ouvert dans ma réflexion. Avoir l'occasion de discuter de la vision du devenir d'un domaine n'est pas si fréquent et est un moment privilégié que j'ai particulièrement apprécié. Les questions sont le plus souvent considérées comme le moment de torture, cela restera ma partie préférée. Merci à toute l'assistance pour son assiduité.

Je remercie l'ensemble de l'équipe Runtime. Chacun aura apporté sa brique à l'*œuvre*. Par ordre d'inspiration : Gonnet mon compère de soirées labriesques, Paulette mon co-bureau hard rocker tapoteur de clavier, Cécile pour son intarrissable folie, PAW mon psy, Mateo pour les 150 milliards de tests que je lui ai fait faire *Mojito ?*, BGoglin pour les Simpsons, les Friends et les patchs de Myri-10G personnalisés, les figures qu'on a jamais envie de faire et les points mouette aussi, Sam mon second co-bureau joueurdetrombone-rollerman-fandemanga, Guigui mon sauveur du pire moment de flip de début de rédaction et *you're talking to me ?*, Alex tout premier co-bureau, tout premier à m'avoir fait douter tellement son discours me semblait obscur : la persévérance à ses vertus !, Broq, Jéjé et Steph qui étaient les étudiants indisciplinés de mon tout premier TD grrrrr une dernière fois, Ludo qui a supporté bon nombre de scènes ésotériques en tant que co-bureau du Gonnet mais pas que, Nathalie LongBeach-gâteaudeNathalie-MadMPI, Marie-Christine pour sa douceur apaisante et ses encouragements, Sylvie pour nos petites conversations, Diak parce qu'il ne faut pas oublier notre parisien ! Pour finir, quelques outsiders de chez les Scalala : Abdou, Nico, Damien, Orel, et autres joyeux lurons.

Je remercie ma famille. Mes parents et mon frère pour leur soutien tout au long de ces années. Neuf ans ! J'entends encore parler du pot de thèse préparé par leurs soins ! J'espère que vous comprenez enfin ce que je *fabrique* chaque jour. Merci à ma grand-mère adorée d'être venue immortaliser mon diplôme ! Enfin, merci à Tamy, mon cher et tendre.

Résumé : Cette thèse cherche à optimiser les communications des applications de calcul intensif s'exécutant sur des grappes de PC. En raison de l'usage massif de processeurs multi-cœurs, il est désormais impératif de gérer un grand nombre de flux de communication concurrents. Durant cette thèse, nous avons mis en évidence et analysé les performances décevantes des solutions actuelles dans un tel contexte. Nous avons ainsi proposé une architecture de communication centrée sur l'arbitrage de l'accès aux matériels. Son originalité réside dans la dissociation de l'activité de l'application de celle des cartes réseaux. Notre modèle exploite l'intervalle de temps introduit entre le dépôt des requêtes de communication et la disponibilité des cartes réseaux pour appliquer des optimisations de manière opportuniste. NewMadeleine implémente ce concept et se révèle capable d'exploiter les réseaux les plus performants du moment. Des tests synthétiques et portages d'implémentations caractéristiques de MPI ont permis de valider l'architecture proposée.

Mots clés : Calcul intensif, communications haute performance, réseaux rapides, stratégies d'ordonnancement dynamiques, MPI, multi-cœurs, multithreading

---

Abstract : The aim of this thesis is to optimize the communications of high performance applications, in the context of clusters computing. Given the massive use of multicore architectures, it is now crucial to handle a large number of concurrent communication flows. We highlighted and analyzed the shortcomings of existing solutions. We therefore designed a new way to schedule communication flows by focusing on the activity of the network cards. Its novelty consists in untying the activity of applications from the one of the network cards. Our model benefits from the delay that exists between the deposal of the communication requests and the moment when the network cards become idle in order to apply some opportunistic optimizations. NewMadeleine implements this model, thus making possible to exploit last generation high speed networks. The approach of NewMadeleine is not only validated by synthetical tests but also by real applications.

Keywords : High Performance Computing, high performance communication, high speed network, dynamic scheduling strategies, MPI, multicore, multithreading

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Le calcul intensif est à l'origine de bon nombre des avancées technologiques des dernières décennies. En effet, il est le principal support technique des simulations numériques, devenues quasi omniprésentes dans notre société. Elles peuvent tout aussi bien intervenir dans des domaines très spécifiques tels la climatologie, la finance, l'armement, la sismologie que pour des problématiques plus communes à l'attention du grand public tels la météorologie, le septième art pour les effets spéciaux ou encore le sport afin d'optimiser les mouvements des athlètes de haut niveau. Outre la modélisation de tels problèmes qui requiert des aspects scientifiques pointus, leur mise œuvre propre n'est pas sans difficulté au vue de la complexité des architectures des machines parallèles employées pour leur exécution.

Deux grands mouvements se partagent, dans des proportions fluctuant avec le temps, le paysage architectural du calcul intensif : les supercalculateurs et les grappes de PC. Tout d'abord, les supercalculateurs, machines ultra puissantes, sont apparus dans les années 70 pour connaître leur âge d'or dans les années 80. Leur force réside dans la spécificité des supports logiciels développés à l'attention de chacun de leur modèle afin d'en tirer le maximum de puissance. Cependant, leur prix élevé associé à une obsolescence rapide due à leur non-extensibilité a ouvert la voie à de nouvelles solutions. Dès le début des années 90, sont ainsi apparues les grappes de PC qui se positionnent comme une alternative avec un meilleur rapport qualité/prix. En effet, l'idée consiste à regrouper une multitude d'unités de calcul dites de technologie *courante* mais néanmoins haut de gamme en terme de puissance offerte, en les reliant par un réseau d'interconnexion de haute performance. On obtient ainsi une plate-forme parallèle certes moins puissante – car moins ajustée et requérant des communications externalisées entre les nœuds qui n'existent pas avec un supercalculateur – mais en revanche beaucoup moins onéreuse. Ces travaux de thèse se situent dans ce contexte architectural.

1.1 Évolution du paysage architectural des plate-formes de calcul

Étant donnée l'hétérogénéité des solutions matérielles disponibles pour les grappes de machines, le besoin d'outils portables d'une architecture à une autre s'est rapidement fait sentir. De nombreuses équipes académiques et industrielles ont collaboré sur ce type de problématique. Sur la question particulière des communications, un consortium a élaboré dans les années 90 l'interface désormais prépondérante MPI, définissant ainsi un ensemble de fonctionnalités nécessaires à l'implémentation d'applications parallèles sur architectures distribuées. Les remarquables performances des implémentations qui en ont été faites,

quasi identiques à celles des performances brutes des réseaux employés, ont assuré leur succès.

Cependant, la conception de ces implémentations a été réalisée à une époque où les machines constituant les grappes ne comportaient qu'un processeur et n'a pas vraiment évoluée par la suite. Depuis, ces dernières ont évolué jusqu'à devenir multi-processeurs puis aujourd'hui multicœurs. On s'attend à voir prochainement des machines à plusieurs dizaines de processeurs. Cette multiplication des unités de calcul remet inévitablement en cause la manière dont sont appréhendées les communications par MPI. En effet, l'évolution du nombre de cartes réseau équipant un nœud ne suivant pas celui des unités de calcul, la symétrie qui existait jusqu'alors ne peut plus être respectée.

1.2 Objectifs et contributions de la thèse

L'objectif majeur de cette thèse est de découvrir quels sont les impératifs des bibliothèques de communication d'aujourd'hui et de demain afin de continuer à exploiter efficacement les réseaux rapides au sein des grappes. Nous verrons cependant qu'il est nécessaire de combiner bien des techniques exploitant ce type de machines afin d'obtenir un fonctionnement satisfaisant.

Pour commencer, nous allons donc faire un état des lieux du matériel en précisant sur quels points il a évolué. Puis, nous nous attellerons à observer et identifier les lacunes des bibliothèques de communication actuelles. Nous aborderons donc la question délicate de l'obsolescence à venir de MPI si ses implémentations restent en l'état. En effet, il ne s'agit pas tant de remettre en question l'interface MPI elle-même mais plutôt l'architecture générale des implémentations existantes qui ne sont pas en l'état capables de gérer efficacement des communications concurrentes. Nous verrons que ces dernières présentent d'ores et déjà des manques que les programmeurs des couches supérieures pallient à la main afin d'améliorer l'ordonnancement de leurs communications alors que certaines fonctionnalités de MPI devraient les gérer si elles étaient implémentées correctement.

De plus, la multiplication du nombre potentiel de flux d'exécution interagissant avec les ressources de communication croissant fortement avec la parallélisation massive des applications, ce phénomène va aller grandissant. Leur accroissement n'étant pas aussi massif que ces derniers, l'arbitrage de l'accès au réseau apparaît donc nécessaire. Ainsi, à la manière d'un ordonnanceur de processus qui lorsqu'un processeur achève sa tâche en sélectionne une autre afin de la lui soumettre, nous proposons une bibliothèque de communication, nommée NEWMADELEINE, qui suit le même principe. Dès qu'une carte réseau achève un transfert, l'ordonnanceur de communication est chargé d'en sélectionner une autre parmi celles en attente afin de la lui soumettre. Au delà de cela, l'innovation de NEWMADELEINE repose sur l'avantage qu'elle tire des délais introduits entre la soumission des communications faites par l'application et celles faites au réseau lui-même. En effet, elle profite de ce laps de temps pour appliquer des optimisations variées sur l'ensemble de messages à destination d'une même machine.

Par ailleurs, NEWMADELEINE se positionne également comme une plate-forme d'expérimentation pour de nouvelles stratégies d'ordonnancement de communication aux programmeurs de couches supérieures. Grâce à NEWMADELEINE, il lui est possible de développer facilement des stratégies d'ordonnancement spécifiquement au schéma de communication suivi par son application. Ce dernier est de ce fait optimisé en accord avec les ressources disponibles de la machine d'expérimentation et cela de manière totalement transparente.

1.3 Organisation du manuscrit

Ce document s'articule autour de trois parties. La première est consacrée à l'état de l'art. Elle est divisée en deux chapitres : le premier fait l'état des lieux des plate-formes matérielles d'aujourd'hui ainsi qu'un point sur la façon dont elles ont évoluées. Le seconde chapitre remonte dans la pile logicielle pour s'intéresser aux couches supérieures. Il discute des interfaces de communication actuellement utilisées. Ce chapitre met en défaut les implémentations les plus prépondérantes et pose ainsi le contexte de recherche de ces travaux de thèse. La seconde partie introduit à la contribution. Elle s'articule autour de trois chapitres. Le premier expose notre proposition dans ses grandes lignes tandis que le second présente NEWMADELEINE, la bibliothèque de communication que nous avons développée afin de les concrétiser. Le troisième chapitre met en avant quelques points essentiels de cette implémentation. La troisième et dernière partie débute par les évaluations menées autour de NEWMADELEINE. Nous y présentons des évaluations synthétiques puis applicatives qui valident avec plus ou moins de succès notre approche. Finalement, nous achevons ce document par les conclusions et perspectives de ces travaux de thèse.

Première partie

État de l’art

Chapitre 2

Communications dans les grappes : État des lieux et perspectives

Sommaire

Avec la demande toujours plus forte de puissance de calcul pour la simulation de systèmes complexes, les recherches et développements autour du calcul haute performance se sont accrus. Sont alors apparues des plates-formes massivement parallèles très puissantes que l'on appelle les super-calculateurs. Cependant, les coûts de développement qu'impliquent de telles machines ont rapidement entraîné la retombée de l'effervescence qui leur avait été portée et la plupart des sociétés spécialisées dans les architectures parallèles ont donc fermé. Les grappes de PC se sont alors rapidement imposées grâce au rapport coût/performance offert : coût réduit par l'emploi de machines de technologies courantes dites *de bureau* et haute performance grâce aux technologies d'interconnexion spécifiquement développées. Pour en témoigner, le classement du top500 [TOP] recense les 500 meilleures machines du moment : aujourd'hui, 80% des plate-formes présentées sont des grappes.

Présentées à l'origine comme le super-calculateur du pauvre, les grappes de PC sont constituées d'un ensemble homogène de machines dites *courantes* mais néanmoins haut de gamme en terme de puissance

offerte, toutes reliées par un réseau d'interconnexion de haute performance. Ainsi, la diversité de configurations matérielles – *e.g.* du point de vue des processeurs, cartes réseau, bus mémoire, etc. – d'une grappe à une autre peut s'avérer vaste.

L'évolution des composants informatiques vers des technologies toujours plus poussées entraîne inévitablement celle de ce type d'architecture. Régulièrement, on assiste donc à une évolution non seulement des processeurs dont la puissance est toujours appelée à suivre l'évolution édictée par la loi de Moore, mais aussi des technologies réseau qui exhibent des performances en communication toujours plus extrêmes. Actuellement, les temps de transfert sont de l'ordre de la microseconde avec le réseau d'interconnexion INFINIBAND, et les taux de transfert atteignent plusieurs gigaoctets par seconde sur la technologie MYRI-10G. Pour comparaison, les performances des réseaux les plus performants au démarrage de ces travaux de thèse étaient déjà remarquables : MYRINET-2000 atteignait des latences de l'ordre de 2 μs et QSNET, des débits de 900 Mo/s.

L'objectif de ce chapitre est de dresser une vue d'ensemble de la configuration matérielle des machines composant une grappe de PC et ce, à court et à moyen terme. Pour cela, nous commençons par décrire l'état d'avancement technologique dont font preuve les réseaux haute performance contemporains, pour ensuite s'attaquer à l'évolution des architectures des processeurs. Avec la progression matérielle des grappes de PC ainsi exposée, il s'agira pour finir de tirer un bilan sur le déséquilibre que l'on peut observer entre ces deux pans de machines.

2.1 Évolution des réseaux et des protocoles

Cette section retrace l'émergence des technologies réseau qualifiées de *haute performance*. Nous y commençons par discuter des réseaux de type ETHERNET et de leurs limitations, pour ensuite introduire les différentes spécificités qui démarquent les réseaux haute performance – également nommés réseaux rapides – de ces réseaux plus usuels. Nous nous intéressons par la suite à la manière dont ces techniques avancées sont mises en pratique dans les interfaces de communication haute performance modernes.

2.1.1 Réseaux ETHERNET et la pile TCP/IP

Dans les années 70, les besoins en communication longue distance ont fait poindre des protocoles de communication dont TCP/IP, qui s'est depuis largement imposé au point de donner le nom de sa couche la plus basse au réseau le plus vaste que l'on connaisse : *Internet* (IP pour INTERNET PROTOCOL). Il s'agissait de pouvoir échanger des données de manière transparente avec un nœud distant et cela, qu'il se situe sur le même réseau local ou à l'autre bout de la planète. Ce souhait a introduit des contraintes fortes dans le modèle telles que la tolérance aux pannes avec des efforts fournis autour du routage dynamique des paquets, la tolérance aux fautes qui introduit de complexes contrôles de validité de données (*checksum*) ou encore du contrôle de flux afin d'éviter la congestion et donc la chute du réseau tout entier.

La figure 2.1 présente les différentes couches logicielles que traverse un paquet depuis l'espace utilisateur jusqu'à la carte ETHERNET. L'interface SOCKET est une interface de niveau utilisateur similaire à celle d'accès aux fichiers UNIX dont chaque primitive est un appel système. Historiquement, l'envoi d'un message nécessitait deux copies et deux parcours des données : une copie mémoire afin de faire

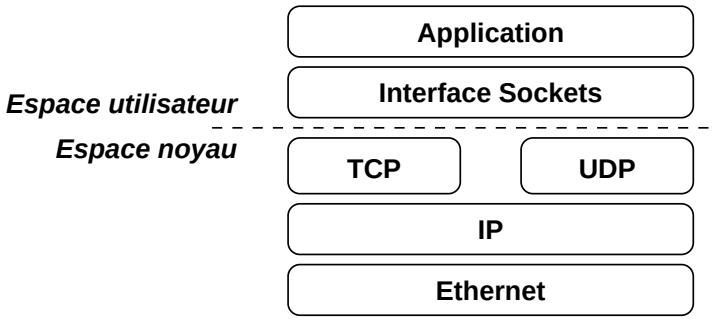


FIG. 2.1: *Accès au réseau* ETHERNET *via la pile TCP/IP.*

passer les données de l'espace utilisateur dans les tampons intermédiaires réservés en espace noyau (les *socket buffers* dans les systèmes UNIX) puis une copie de ces tampons vers les tampons embarqués dans la carte ETHERNET. Ce transfert se faisait après ajout des entêtes spécifiques des couches TCP et IP qui sont notamment constitués d'une somme de contrôle des données aidant à la détection d'erreurs dans leur transmission mais qui nécessite en contre-partie un parcours de leur ensemble. Cette pile logicielle fortement coûteuse en temps processeur et mémoire a depuis été optimisée pour ne plus nécessiter qu'une seule copie : en l'occurrence la copie mémoire faisant passer les données dans les *socket buffers* puisque le transfert des données jusqu'à la carte est à présent assuré par le processeur de la carte réseau lui-même par un procédé différent (voir section 2.1.2.1). L'ensemble des parcours effectués par le processeur de la machine hôte sont aussi supprimés : la somme de contrôle des données est à présent faite par la carte ETHERNET. De la même façon en réception, une des deux copies auparavant nécessaires (une depuis la carte jusqu'à un tampon intermédiaire puis de ce dernier au tampon utilisateur) ainsi que les contrôles de validité des données sont déportés sur la carte réseau. Par ailleurs, les interruptions nécessaires au déclenchement du traitement des données sont agrégées avant d'être remontées jusqu'au système d'exploitation de manière à réduire leur impact sur les performances. La Figure 2.2 donne une vue d'ensemble du parcours suivi par un message soumis à l'interface SOCKET. Cette dernière propose des primitives pour la plupart de type bloquant. Ainsi, les applications lui faisant appel n'ont d'autre choix que de s'endormir le temps que leurs données soient traitées. Ce dernier point constitue aussi un point coûteux en temps et en ressources matérielles.

Cependant, en dépit de ses performances moyennes en terme de latence, l'universalité d'ETHERNET a engendré un certain nombre de travaux aux niveaux matériel et logiciel. Tout d'abord, de manière stan-

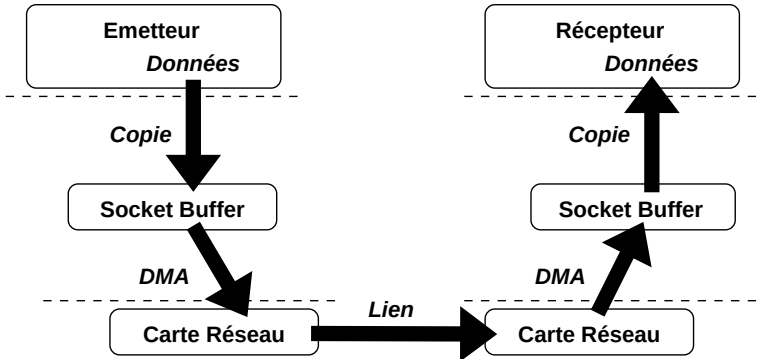


FIG. 2.2: *Transfert de données avec l'interface* SOCKET.

dard, ETHERNET initialement conçu avec un débit de 10 Mbit/s a évolué pour atteindre 100 Mbit/s avec FAST ETHERNET, 1000 Mbit/s avec le GIGABIT ETHERNET et actuellement 10 Gbit/s avec l'ETHERNET 10-GIGABIT grâce notamment à l'utilisation de câbles plus performants. De plus, les cartes ETHERNET sont généralement des cartes programmables dont les spécifications nécessaires au changement du code embarqué sur la carte (le *firmware*) sont fermées au public. Ainsi, des optimisations pouvant aller de la modification de la MTU (*Maximum Transmission Unit*) [GU01] à la mise en œuvre de mécanismes propres aux réseaux haute performance [PF01, SWP01, KRS01, DDW06] – détaillés dans la section 2.1.2 – ont été développées, mais cela toujours en collaboration avec un constructeur particulier et donc spécifiquement à un modèle de carte ETHERNET. Ces optimisations ne sont donc pas portable d'un modèle de carte ETHERNET à un autre. Ceci s'avère un point bloquant dans le cadre d'un environnement où le type de carte ETHERNET employé peut changer d'une grappe à une autre. Certaines implémentations d'interfaces de communication haute performance de niveau supérieur comme SBP [RH98], GAMMA [CC97], VIA [DRM+98], ou encore U-NET [EBBV95] se sont tout de même attachées à donner une couche d'abstraction à ETHERNET ou plus exactement à certains modèles de carte ETHERNET.

La pile TCP/IP, qui n'a jusqu'à présent jamais fondamentalement changé, reste de nos jours massivement utilisée pour accéder à une carte de type ETHERNET. Malgré des latences toujours importantes – de l'ordre de quelques dizaines de μs–, ETHERNET trouve tout de même sa place dans le paysage des réseaux haute performance à tel point que plus de la moitié des machines du top500 en sont équipées. En effet, ces performances restent suffisamment raisonnables pour les applications de calcul haute performance qui ne sont pas sensibles à la latence.

2.1.2 Réseaux haute performance et protocoles de communication dédiés

Les latences des réseaux basiques de type ETHERNET étant grandement pénalisées par la pile logicielle TCP/IP nécessaire pour y accéder de manière générique et les optimisations proposées n'étant pas génériques à toutes les cartes, l'alternative a été trouvée dès les années 80 dans le développement de nouvelles technologies réputées fiables que sont les réseaux haute performance. Leur développement a grandement été influencé par les réseaux d'interconnexion internes des supercalculateurs de l'époque. Ainsi, sur le même modèle, ils se sont directement affranchis de contraintes liées aux communications longue distance citées précédemment (tolérance aux pannes, tolérance aux fautes, contrôle de congestion, etc.) puisque destinés à des architectures fortement locales : la taille d'une grappe de PC n'excède jamais celle d'un bâtiment. De plus, grâce à l'intelligence que l'on peut désormais embarquer dans ces cartes – car programmables et dotées de mémoire plus conséquente –, les latences peuvent maintenant se réduire à presque une microseconde (performance actuelle d'INFINIBAND) et atteindre plusieurs gigaoctets par seconde de bande passante. Le gain de performance sur la latence est induit par la mise en œuvre de mécanismes très pointus développés depuis un quinzaine d'années afin de réduire en temps le chemin critique. Dans cette section, nous décrirons ces mécanismes puis leur emploi au sein des interfaces de communication les plus performantes du moment : MYRINET EXPRESS (MX) sur MYRINET, ELAN sur QSNET et VERBS sur INFINIBAND.

2.1.2.1 Clés des réseaux haute performance

La différence entre un réseau usuel et un réseau rapide réside dans l'utilisation de mécanismes clés. Nous présentons ici comment le système d'exploitation peut se retrouver exclu du mode d'accès aux

ressources de communication et comment le processeur peut être déchargé de transferts de données entre la mémoire de la machine hôte et celle de la carte réseau. Enfin, les différents paradigmes de communications sont présentés.

Communications en espace utilisateur Comme pour tous les périphériques, l'accès à une carte réseau est normalement réservé au système d'exploitation : une application ne peut dialoguer avec une carte sans son intermédiaire. Par conséquent, l'émission ou la réception d'un message depuis une application requiert un appel système, comme dans le cas de l'interface SOCKET (cf section 2.1.1). Afin d'en réduire le coût, tout comme pour les cartes graphiques, le système d'exploitation est tout bonnement court-circuité afin d'accéder directement à la carte réseau, cette technique est aussi communément appelée l'*OS-bypass*. Cette tactique nécessite l'utilisation d'une extension du noyau fournie avec l'interface de communication. À l'initialisation du programme, cette dernière établit des projections de régions mémoire qui permettent de dialoguer avec la carte dans l'espace d'adressage de l'application. Une fois ces projections établies, l'application peut alors lire et écrire dans les registres de la carte et donc la contrôler directement, c'est-à-dire sans appel système. C'est ce que l'on appelle les *communications en espace utilisateur*. Au milieu des années 90, l'interface de communication U-NET [EBBV95] est l'un des précurseurs de la mise en œuvre de ce procédé. À présent, c'est l'ensemble des interfaces de communication haute performance qui l'emploie.

Les appels système ont longtemps été considérés comme trop coûteux pour être conservés sur le chemin critique d'une interface de communication pour réseaux rapides. Cependant, alors qu'ils se mesuraient encore en milliers de cycles sous LINUX sur un processeur INTEL au début de ces travaux de thèse – soit un coût de l'ordre de la microseconde –, ils ne coûtent aujourd'hui plus qu'une centaine de nanosecondes sur les processeurs analogues d'aujourd'hui. Cette technique pourrait donc se voir devenir obsolète pour les prochaines générations de technologies haute performance.

Transfert de données vers une carte réseau sans copie intermédiaire Les recopies mémoire intermédiaires des données effectuées en émission et/ou en réception sont également partie prenante dans le surcoût logiciel observé.

Nous avons mis en évidence celles nécessaires au contexte de TCP/IP dans la section 2.1.1.

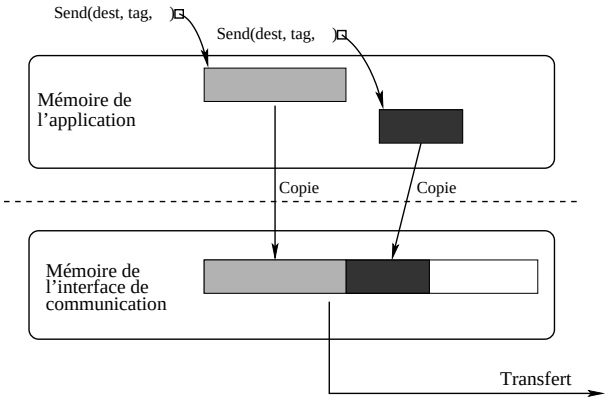


FIG. 2.3: *Copies intermédiaires avant transmission.*

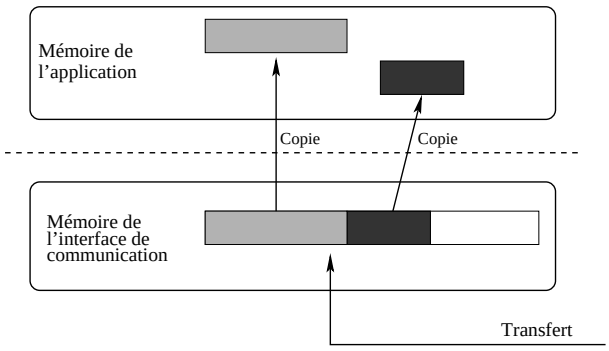


FIG. 2.4: *Copies intermédiaires à l'arrivée.*

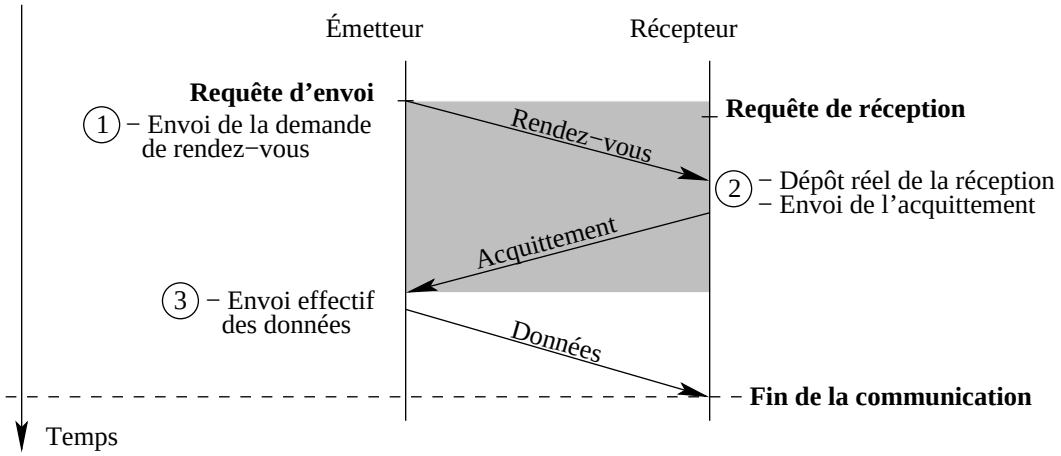


FIG. 2.5: *Scénario de rendez-vous entre l'émetteur et de récepteur. Côté récepteur, l'adresse de réception finale des données n'est fournie à la carte réseau que sur arrivée d'une demande de rendez-vous. Côté émetteur, les données ne sont effectivement transmises que sur réception d'acquittement de la prise en compte du rendez-vous préalablement envoyé.*

Des recopies peuvent être introduites par la conception d'un protocole de communication. Celles en émission sont typiquement utilisées pour ajouter des entêtes spécifiques au protocole sous-jacent, pour constituer des segments contigus à partir de données éparses en mémoire, ou encore pour permettre à l'application de réutiliser des emplacements mémoire sans corrompre les données à transmettre (voir Figure 2.3). En réception, les recopies sont surtout utilisées dans le cas où les données sont reçues par la carte réseau avant que l'application n'ait indiqué à quel endroit il fallait les placer (Figure 2.4). Pour des tailles de données importantes, une prise de rendez-vous préalable entre la source et la destination permet d'éviter cette situation car l'émetteur va attendre que le récepteur soit prêt avant d'envoyer des données (Figure 2.5). Le récepteur pourra ainsi réceptionner les données directement à leur destination finale.

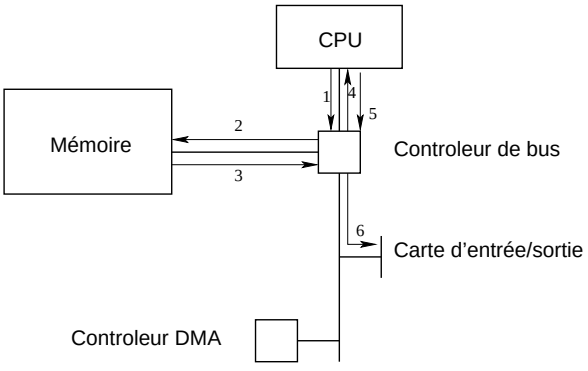


FIG. 2.6: *Transfert par entrées/sorties programmé.*

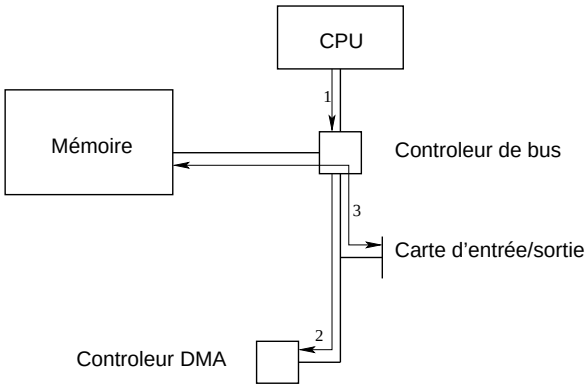


FIG. 2.7: *Transfert par accès direct à la mémoire.*

Indépendamment de ce type de copies dont le nombre est bien évidemment très limité dans un protocole de communication qui se veut efficace, il reste le transfert des données de la mémoire principale jusqu'à celle de la carte réseau elle-même. Le procédé initial appelé PIO (*Programmed Input/Output*)

fait transiter les données d'une zone de la mémoire principale à celle de la carte (ou inversement) sur les directives du processeur de la machine hôte (voir Figure 2.6). C'est sur ce mode de transfert que des progrès ont été faits. Tout d'abord conçue pour améliorer les accès aux disques durs, la technique par accès direct à la mémoire, appelée DMA (*Direct Memory Access*) permet d'initier le transfert de données sans copie supplémentaire et surtout sans l'aide du processeur de la machine hôte (Figure 2.7). Cependant, cette technique a l'inconvénient d'avoir des coûts d'initialisation et de terminaison importants mais constants. En effet, pour effectuer un transfert [HP03], le processeur commence par initialiser les registres du contrôleur DMA avec l'adresse mémoire source, la longueur à transférer, et l'adresse de destination. L'opération initiée, le processeur est alors libre d'effectuer d'autres tâches. Ce dernier ne sera sollicité par le contrôleur DMA qu'une fois le transfert accompli.

L'emploi d'un transfert par PIO ou par DMA est déterminé par l'interface de communication pilotant la carte à partir de la taille de données à traiter. Seules quelques interfaces, telles que SISCI, laisse ce choix à la charge de l'utilisateur. L'allure des courbes de coûts en temps par rapport à la taille des données transférées de ces techniques (Figure 2.8) montre qu'il est préférable d'employer le transfert PIO pour les messages courts et le transfert DMA pour les plus longs. Le modèle LOGP en propose une modélisation formelle [CKP+96].

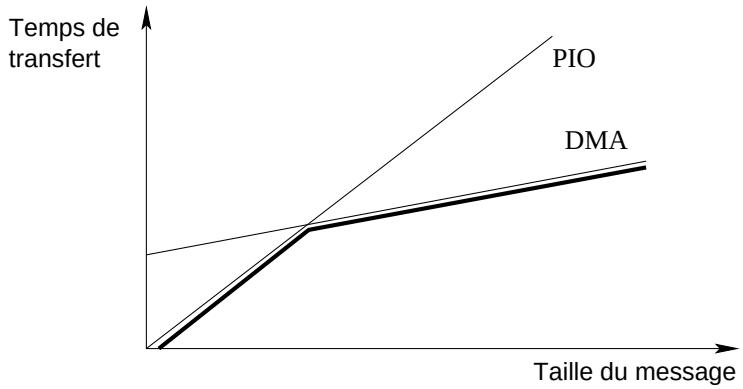


FIG. 2.8: *Allure de la courbe du coût de transfert de données.*

Toujours dans l'optique de réduire l'occupation du processeur par la copie de données, INTEL équipe depuis peu ses bus d'une nouvelle technologie nommé I/O ACCELERATOR TECHNOLOGY (I/O AT) [Int]. Ce composant est capable de procéder aux copies que le système d'exploitation ou que le *firmware* d'une carte réseau lui confient de manière asynchrone sans l'aide d'aucune autre ressource. Ainsi, le traitement des copies est totalement déporté et donc transparent en coût processeur [VP07, Gog08b].

Paradigmes de communication L'exploitation de technologies réseau plus avancées a mené à la définition de nouveaux modèles d'échange de données. Ainsi, en plus du modèle traditionnel de l'interface SOCKET qui procède par entrées/sorties sur fichiers grâce à une sémantique d'échanges de données par flux (STREAM) ou par paquets (DATAGRAM), sont venus s'ajouter des modèles de communication par passage de message et par accès direct à la mémoire d'une machine distante.

Une interface de type passage de message se caractérise par la participation de l'émetteur et du récepteur dans un échange de données via l'appel à une primitive d'envoi ou de réception. Habituellement, plusieurs variantes de ces primitives sont proposées qui se différencient par leur caractère synchrone/a-

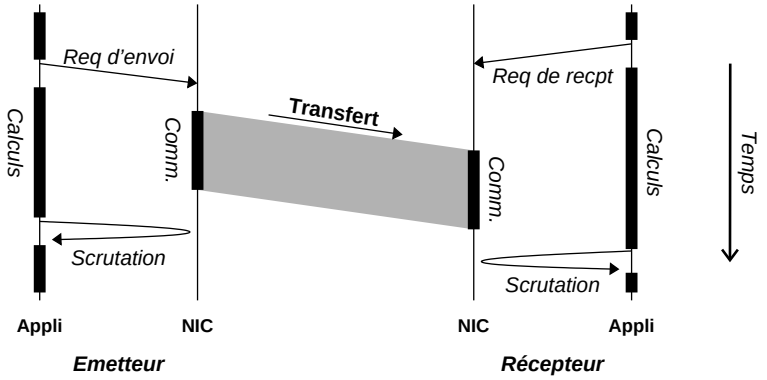


FIG. 2.9: *Paradigme du passage par message. Chaque acteur de la communication dépose une requête à la carte réseau. Le transfert est réalisé en arrière-plan des calculs si la bibliothèque de communication possède un* thread *de progression. Finalement, les cartes sont interrogées jusqu'à ce que le transfert soit achevé.*

synchrone bloquant/non bloquant. Une communication synchrone assure l'utilisateur que le récepteur a entamé le traitement des données devant être échangées lorsque la primitive rend la main à l'utilisateur tandis qu'un appel bloquant ne rend la main à l'application que lorsque la communication est totalement réalisée.

Considérons à présent les communications non bloquantes asynchrones. Ces primitives, qu'elles soient d'émission ou de réception, ne fournissent aucune garantie sur l'envoi/la réception effective des données à échanger. Peut-être vont-elles se contenter d'enregistrer la communication, de l'initier ou vont-elles aller jusqu'à leur terminaison. À part dans ce dernier cas, aucune indication sur l'état d'avancement de la communication n'est donnée. La vérification de la terminaison de telles requêtes est laissée à la charge de l'application grâce à des primitives de scrutation qui sont également disponibles sous des formes bloquante et non bloquante. En interne, il existe deux manières très distinctes d'attendre la fin d'une requête de communication. La première va procéder par attente active, c'est-à-dire qu'elle va interroger la carte tant que l'opération n'aura pas été achevée. La seconde est fondée sur les interruptions. Le pilote réseau notifie par un signal l'application lorsque sa requête prend fin. Ainsi, entre le moment où la requête de communication est déposée et le moment où elle est effectivement achevée (*i.e.* lorsque la scrutation commence), le pilote de la carte réseau a tout loisir de procéder à la communication en tâche de fond. La Figure 2.9 illustre le paradigme de passage par message.

Les accès directs à la mémoire d'une machine distante (RDMA (*Remote Direct Memory Access*)) ont un comportement assez différent. Chaque nœud met à disposition des autres des zones réservées dans son propre espace mémoire. L'adresse de chacune de ces zones est échangée pour qu'un nœud soit par la suite libre d'aller y déposer des données via une opération d'écriture ou bien d'en récupérer grâce à une opération de lecture et, cela sans même que le processeur de l'hôte ne soit sollicité. Les primitives de communication proposées pour cette classe de paradigme sont du type lecture/écriture. Il n'y a pas de notion d'asynchronisme puisque l'initiateur de la communication est l'unique intervenant dans cette dernière. On retrouve néanmoins la notion de communication bloquante/non bloquante et donc la notion de scrutation permettant de déterminer l'état d'avancement d'une communication. La Figure 2.10 illustre ce mode de fonctionnement.

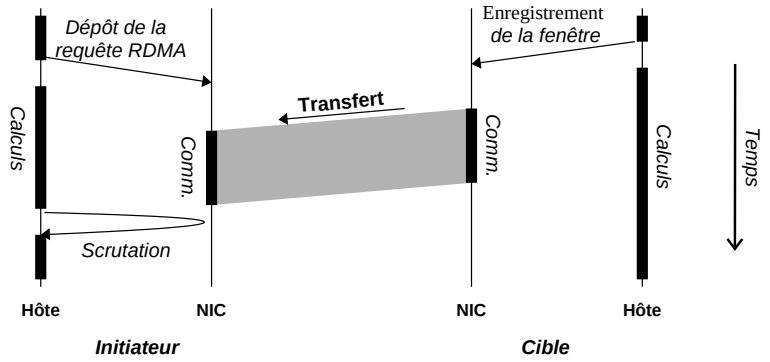


FIG. 2.10: *Paradigme par accès direct à de la mémoire distante. Le nœud destinataire enregistre une fenêtre de données. L'initiateur de la communication dépose une requête à la carte réseau. Le transfert lui-même a lieu en arrière-plan des calculs de l'initiateur et du destinataire. Finalement, l'initiateur détecte la fin de l'échange en interrogeant sa carte réseau.*

2.1.2.2 Descriptions des pilotes de réseaux haute performance actuels

Nous présentons ici l'ensemble des réseaux haute performance considérés comme les plus représentatifs durant cette thèse. En l'occurrence, MYRI-10G, QSNETII et INFINIBAND. On s'intéresse également à GIGABIT ETHERNET, qui maintient sa place dans le paysage du calcul haute performance.

MX/Myri-10G La société MYRICOM propose depuis 2006 sa dernière technologie haute performance en matière d'interconnexion nommée MYRI-10G. Cette carte est composée d'un processeur RISC, de 2 Mo de mémoire embarquée et est pilotée par une interface de communication bas niveau nommée MYRINET EXPRESS (MX), dont la sémantique proposée à l'utilisateur est très proche de celle de MPI. Même s'il s'agit d'une interface de bas niveau, MX implémente en interne un certain nombre de mécanismes qui pourraient lui conférer le statut d'interface de communication de niveau intermédiaire. Pour exemple, MX propose deux primitives d'envoi (envoi non bloquant synchrone et asynchrone) qui cachent complètement les méthodes de transfert utilisées en interne selon la taille des messages. Ces dernières diffèrent principalement par la méthode utilisée pour faire transiter les données depuis la mémoire principale vers la mémoire de la carte réseau (*e.g.* PIO, DMA, DMA pipeliné). De plus, même s'il est vrai que ce point est fréquemment assuré par l'interface de bas niveau, l'enregistrement transparent de la mémoire au niveau des tampons utilisateurs, la prise de rendez-vous si nécessaire, la progression des communications en tâche de fond par un *thread* dédié à cet usage sont autant de mécanismes que l'utilisateur n'a pas à gérer. Elle dispose également d'un puissant système d'*appariement* – également appelé *filtrage* ou encore *réception sélective* – qui permet de mettre en correspondance les données transitant et les requêtes postées par l'application et qui rend le support des messages provenant de source anonyme efficace. Cette technologie arbore 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 QUADRICS propose la technologie QSNET II. L'interface de communication associée, ELAN, procède par accès direct à la mémoire distante, mais il existe également une autre interface orientée par passage de message TPORT (pour *Tagged message Port*) construite au-dessus de ELAN (voir Figure 2.11).

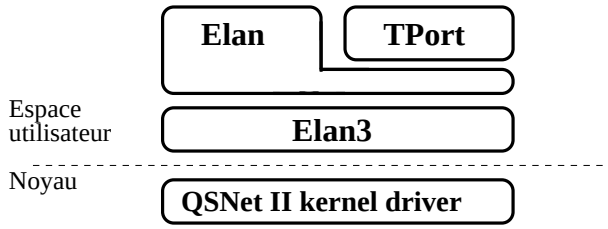


FIG. 2.11: *Pile logicielle de* QUADRICS.

Grâce à un processeur puissant qui donne l'opportunité de faire des transferts en PIO extrêmement efficaces, QSNET offre des performances en latence proches de la microseconde. Du point de vue de la bande passante, QUADRICS a innové en embarquant dans sa carte réseau un composant logiciel très particulier, à savoir une MMU (*Memory Management Unit*). Cette dernière permet la traduction directe des adresses mémoire virtuelles fournies par l'application en adresse mémoire physique. Ainsi, lorsqu'une communication est postée, comme il n'y a plus besoin de traduire les adresses, le fait de fixer la correspondance entre adresse physique et virtuelle par un punaisage de la mémoire devient obsolète. Ce mécanisme améliore grandement les performances en terme de bande passante. QSNET atteint les 900 Mo/s théoriques apportés par la fréquence du bus mémoire de la machine. Cependant, cet embarquement a un coût mémoire non négligeable. Ainsi, le nombre de processus pouvant simultanément s'exécuter sur une machine est très restreint. Une version qui court-circuite ce mécanisme et emploie donc un modèle plus classique est également disponible mais dégrade les performances en bande passante.

Malgré ces performances, différents facteurs limitent la portée de cette technologie. Tout d'abord, l'évolution des processeurs vers des architectures multicœurs a introduit de nouveaux modèles dont certains visent à fixer un processus sur chacun des cœurs disponibles de la machine. Ainsi, le nombre restreint de processus pouvant cohabiter sur une même machine est extrêmement limitant. De plus, la modification du noyau imposée par la MMU embarquée est assez lourde en terme de maintenance des machines et n'est pas toujours bien accueillie par les administrateurs des grandes plate-formes de calcul. Pour finir, malgré un prix élevé, QUADRICS n'a pas su réellement faire évoluer son produit. En effet, QSNET n'est disponible que sur les bus de type PCI et non PCI-Express, ce qui plafonne la bande passante à 900 Mo/s alors que les autres technologies affichent des débits déjà de l'ordre de 10 Gbit/s.

Verbs/INFINIBAND INFINIBAND est une technologie standard plus récente que les deux précédentes, même si les travaux de réflexion sur sa conception ont débuté avec ceux de l'interface VIA (Virtual Interface Architecture) [DRM+98] à la fin des années 90. À l'origine, cette technologie se voulait universelle en matière d'entrées/sorties : une révolution matérielle au niveau des bus internes (mémoire et d'entrées/sorties), des technologies réseau et des systèmes de stockage. Cet effort s'est finalement réduit à la partie réseau lorsque INTEL, l'un des partenaires du consortium à l'origine de ce mouvement, s'est retiré pour reporter ses efforts sur le bus d'entrées/sorties PCI-EXPRESS. Même si ce projet n'est pas allé jusqu'au bout de ses ambitions, cela a sans aucun doute été un effort de réflexion qui a orienté les architectures matérielles à venir qui tendent à uniformiser les bus mémoire et d'entrées/sorties (voir partie 2.2.1).

Les VERBS sont l'interface officielle d'INFINIBAND, ils suivent le paradigme de communication par RDMA . Ainsi, plusieurs constructeurs comme CISCO ou encore MELLANOX proposent des matériels différents avec chacun une implémentation des VERBS propriétaires. Encore plus prononcée que pour

ETHERNET (cf 2.1.1), cette diversité d'implémentations a entraîné des conceptions différentes, une exploitation donc spécifique de chaque carte et un dispersement dans l'avancement des projets. OPENFABRICS, anciennement nommé OPENIB, est un projet *open source* qui se proposait à l'origine de fournir une pile logicielle visant à exploiter INFINIBAND exclusivement. Plus récemment, s'est ajouté l'ensemble des réseaux de type RDMA/IP avec l'association de technologie comme IWARP [DDW06] qui suivent le même modèle de communication par accès direct à la mémoire de machines distantes.

Même s'il existe des interfaces de niveau supérieur comme DAPL, VERBS est la plus utilisée mais aussi la plus difficile à utiliser car de très bas niveau. Contrairement au confort rencontré avec MX, tout est laissé à la charge de l'utilisateur : enregistrement explicite des zones mémoire, choix de la méthode de transfert adaptée, optimisation des transferts des données vers la carte réseau, etc. Deux méthodes de transfert sont proposées : une qui suit le modèle classique d'accès direct à de la mémoire distante en écriture et en lecture, et une seconde, qui émule le modèle par passage de message. En effet, l'émetteur envoie les données dans une zone mémoire intermédiaire grâce à un RDMA et le récepteur, sur requête de l'application, ira ensuite récupérer les données à l'adresse convenue. Ainsi, utilisés correctement, les VERBS proposent des temps de latence de 1,5 μs et une bande passante de 1,5 Go/s sur des cartes CONNECTX. Ces performances font d'INFINIBAND l'un des réseaux les plus utilisés : un quart des grappes représentées au TOP500 en est équipé.

GIGABIT ETHERNET/Sockets L'universalité des cartes ETHERNET dans les ordinateurs basiques a indéniablement assis la place de cette technologie dans le paysage du calcul haute performance. Cela a même conduit à des travaux entrepris par les constructeurs de cartes réseau haute performance visant à offrir un accès à leur technologies via l'interface SOCKET (cf IPOMX, IP over QSNET, etc.), ce qui consiste finalement à l'encapsulation du trafic TCP/IP dans des paquets spécifiques à la technologie employée. Cependant, cela n'a pas suffi à évincer ETHERNET et différentes évolutions ont donc été apportées afin qu'il reste concurrent. En plus de modifications purement matérielles au niveau des cartes telles que les transferts DMA, des innovations des réseaux haute performance sont désormais employées couramment par ETHERNET.

La version actuelle est le GIGABIT ETHERNET qui atteint des débits de 125 Mo/s. L'ETHERNET 10GIGABIT est d'ores et déjà disponible mais le coût induit par les commutateurs nécessaires au passage à l'échelle des grappes actuelles et sûrement le manque de besoin de la part des utilisateurs freinent encore son expansion. Malgré des performances bien inférieures en terme de latence, plus de la moitié des grappes du TOP500 utilise cette technologie à moindre coût qui reste suffisante pour le cas d'applications ne nécessitant pas beaucoup de communications.

Par ailleurs, ETHERNET est de plus en plus souvent employé comme couche de transport à la place de TCP pour les échanges avec des systèmes de stockage. La tendance va vers des protocoles tels FCoE(*Fibre Channel over Ethernet*) ou encore AoE(*ATA over Ethernet*) qui utilisent ETHERNET uniquement comme couche de transport. En effet, leur force est un gain en performance significatif en terme de latence grâce au court-circuitage de la lourde pile logicielle TCP/IP. Ainsi, la remise en question de cette pile pour les communications de type passage par message dans le contexte des grappes et des réseaux locaux est fortement d'actualité. Le projet GAMMA [CC97] a ouvert la voie sur ce type d'optimisation mais n'a pas eu le succès escompté. En effet, des modifications des pilotes ETHERNET eux-même étaient nécessaires, ce qui en premier lieu était difficile au vu du nombre de pilotes différents et en second lieu, nuisait à l'interopérabilité entre des paquets provenant de la pile TCP/IP classique et ceux empruntant la pile réduite. S'est ensuite monté le projet IWARP, brique du projet OPENFABRICS, qui propose sa propre interface par accès direct à la mémoire au-dessus d'IP. Elle devrait être unifor-

misée sous peu à l'interface proposée pour la technologie INFINIBAND, *i.e.* les VERBS. Cependant, cette nouvelle pile logicielle est réservée aux cartes ayant la capacité d'accéder à de la mémoire de manière distante (dans le jargon, ce sont des *RDMA-enabled NIC* ou RNIC). Cependant, le coût d'un tel matériel enlève tout l'intérêt d'utiliser une carte ETHERNET et n'a donc pas eu le succès attendu. Plus récemment, dans un effort de standardisation de son interface, la société MYRICOM, productrice de la technologie MYRINET, a proposé OPEN-MX [Gog08a], une implémentation des couches logicielles MX pouvant s'exécuter sur n'importe quel matériel ETHERNET. Cette interface propose désormais de bonnes performances en latence et en bande passante, en exigeant ni de modifications des pilotes ETHERNET ni l'utilisation de cartes spécifiques.

Bilan Chaque technologie a une conception matérielle et une interface de communication bas niveau qui lui est propre. Même si certains mécanismes sont uniformément utilisés par chacune d'elles, le fait de chercher à optimiser au maximum l'envoi brut des données crée une diversité de programmation indiscutable. En effet, la vision précise que chaque constructeur a de sa technologie permet de produire une implémentation spécifique très optimisée qui ne serait pas réalisable par un simple utilisateur du matériel. Tout espoir de portabilité d'une application d'un réseau à un autre est donc hors de considération. De plus, dans un souci de gain de performance, ces interfaces restent assez peu fonctionnelles. Des services comme le contrôle de flux, le choix des méthodes de transferts ou encore le multiplexage de communications ne sont pas toujours offerts, ce qui exige un développement assez pointu et conséquent pour tout développeur de couches supérieures. Enfin, l'interface de communication n'a qu'une visibilité réduite de ce qui s'opère sur la machine. Autant les envois bruts peuvent être efficaces, autant des schémas de communication plus complexes dont l'exécution dépend non seulement de l'application elle-même mais aussi de l'état courant de la machine voire de son architecture ne peuvent être traités à leur niveau. Finalement, ces interfaces de communication bas niveau sont indispensables afin de pouvoir extraire tout le potentiel des différentes technologies mais ne sont pas suffisantes pour fournir un support de communication portable, toujours efficace et facile d'utilisation.

2.1.3 Abstraction des protocoles de communication dédiés

Si les interfaces de communication bas niveau présentées précédemment exhibent évidemment les meilleures performances possibles sur leurs technologies respectives, elles leur restent également spécifiques. La diversité de programmation entre chacune de ces dernières ne peut qu'amener à l'abandon de tout espoir de portabilité d'une application si la technologie réseau sous-jacente doit changer. La disparité de ces dernières dans un environnement tel que les grappes de PC a fait de ce point une réelle problématique de recherche.

Les interfaces de communication bas niveau sont prédestinées à servir de cible pour des bibliothèques de plus haut niveau, c'est-à-dire des interfaces se construisant au-dessus de ces dernières. Un grand nombre, dit de niveau intermédiaire, a ainsi été développé dans le but de fournir une abstraction du matériel sous-jacent. On peut citer des travaux tels que FAST MESSAGE [PKC97], VMI [PP02], PM [TSH+00] ou encore MADELEINE [ABD+02]. Généralement, chacune de ces implémentations propose, en plus de l'effort d'abstraction, des fonctionnalités qui leur sont originales, destinées à répondre à un besoin particulier des applications. Par exemple, alors que certaines se restreignent à fournir une interface générique de type passage par message, d'autres construisent des modèles de programmation plus complexes comme l'accès aux données de manière transparente avec un modèle de mémoire partagée distribuée, ou encore l'appel à des procédures à distance. Ces travaux résolvaient une partie du problème

de la portabilité des applications mais leur multiplicité a fait qu'aucune n'a réellement été employée. En effet, les développeurs d'applications leur ont rapidement préféré des outils peut-être moins performants car moins spécifiques mais offrant plus de garanties en termes de portabilité et de fonctionnalités. La cible des interfaces de niveau intermédiaire s'est donc plus orientée vers le support aux interfaces de communication de plus haut niveau plus universelles et plus complètes en fonctionnalités.

Après plusieurs solutions élaborées, un consortium composé des acteurs principaux de la scène du calcul haute performance (constructeurs, industriels et universitaires) s'est formé au début des années 90 dans l'optique de mettre au point une solution standard pour le développement d'applications parallèles. La première version de MPI (*Message Passing Interface*) qui définit une interface de communication de type passage par message sort au milieu des années 90. La seconde, MPI -2, complète et corrige des imperfections de la précédente spécification. Les discussions autour de la prochaine évolution sont en cours.

MPI se définit par un vaste ensemble de spécifications et de fonctionnalités qui se veulent foncièrement génériques. En aucun cas la technologie sous-jacente n'influence le modèle de programmation exposé au niveau de l'interface. Un large éventail d'implémentations est disponible : des implémentations libres et commerciales qui peuvent être le travail de constructeurs de technologies réseaux ou de machines, ou bien d'universitaires. En effet, autant l'interface standard exposée n'est pas influencée par le matériel sous-jacent, autant la réciproque n'est pas vraie. Les performances d'une implémentation de MPI peuvent se révéler dépendantes des caractéristiques du matériel. En effet, chacune d'elles est réalisée dans l'objectif d'extraire les meilleures performances possibles du matériel considéré. Ainsi, l'éventail des implémentations va de celles dédiées à un supercalculateur particulier qui ne supportent que les réseaux dont il est muni, aux génériques telles MPICH2 [GLDS96] ou OPENMPI [GWS05] qui sont disponibles sur un large spectre d'architectures processeur et de réseaux d'interconnexion. Elles sont typiquement destinées aux applications s'exécutant sur des grappes de machines. Dans une gamme intermédiaire, les constructeurs de réseaux d'interconnexion fournissent généralement une implémentation à mi-chemin. Ils choisissent une implémentation dite générique et l'adaptent spécifiquement à leur technologie (*e.g.* MPICH2-MX [mpi]). Dans tous les cas, les performances obtenues sont de très bonne qualité comme l'atteste la Figure 2.12.

2.2 Évolution de l'architecture des machines

La configuration matérielle interne à chaque nœud joue un rôle clé dans la puissance globale d'une grappe. La capacité de communiquer rapidement avec les nœuds voisins serait grandement altérée si le nœud lui-même n'était pas assez puissant pour traiter les communications qui lui arrivent et les calculs qu'exigent l'application. Ainsi, l'architecture des machines des grappes de PC a tout naturellement suivi l'évolution matérielle des composants informatiques vers des technologies toujours plus performantes. Ceci a inévitablement impacté le support nécessaire à l'exploitation de programmes parallèles. Dans cette section, nous allons dans un premier temps introduire les évolutions matérielles qu'ont subies les bus internes aux machines, pour ensuite décrire ce qui a pour ainsi dire bouleversé les modèles de programmation des applications, à savoir l'évolution qu'ont suivie les processeurs.

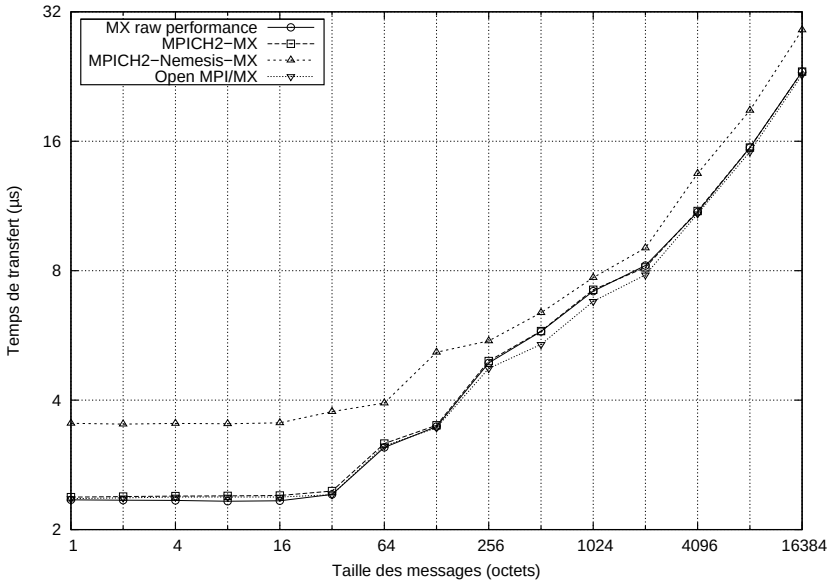


FIG. 2.12: *Temps de transfert de* MX *vs* MPICH2-MX *(version dédiée produite par* MYRICOM*) vs* MPICH2-NEMESIS-MX *(implémentation générique d'*ARGONNE *avec le channel* NEMESIS*).*

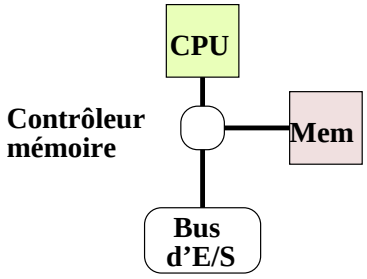


FIG. 2.13: *Architecture monoprocesseur.*

2.2.1 Connexions entre processeurs, mémoires et dispositifs d'entrées/sorties

Même si l'évolution des bus internes d'interconnexion de composants n'a pas provoqué de remise en cause des modèles de programmation des applications parallèles, ces derniers restent un élément clé des performances d'un ordinateur. Dans cette partie, nous présentons successivement les bus mémoire qui assurent les transferts de données entre les processeurs et la mémoire d'une machine, et les bus d'entrées/sorties, qui eux assurent les transferts entre les périphériques les éléments précédemment cités. Nous verrons ensuite comment la tendance actuelle mène à leur uniformisation.

2.2.1.1 Les bus mémoire

Tant que les machines étaient équipées d'un processeur unique, la conception des cartes mère étaient globalement identique d'un constructeur à l'autre. Comme le montre la Figure 2.13, le contrôleur de bus faisait le lien entre le processeur, la mémoire et le bus d'entrées/sorties sur les périphériques.

Avec l'apparition des multiprocesseurs et processeurs multicœurs que nous détaillons dans la partie 2.2.2,

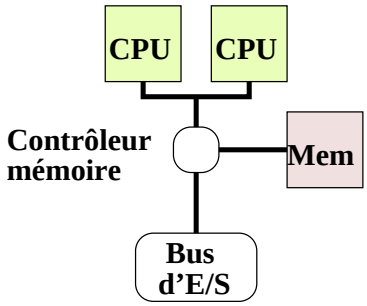


FIG. 2.14: *Architecture multiprocesseur d'*INTEL.

deux constructeurs se partagent actuellement le marché de la production de ces ressources : INTEL et AMD.

INTEL a pour le moment conservé une carte mère sur le même modèle (Figure 2.14). Tous les processeurs sont raccordés au reste de la machine au travers du même chipset mémoire. Ceci a le défaut de laisser s'établir une contention au niveau des accès à la mémoire. C'est pour cela qu'un nouveau modèle de carte mère devrait apparaître dans les prochains mois avec la toute dernière génération de processeurs tels le NEHALEM et l'ITANIUM2 TUKWILA. Le bus mémoire, nommé QUICKPATH [qui], vise à remplacer le goulot d'étranglement actuel par un modèle assez proche de celui adopté par son concurrent. En effet, AMD propose un bus mémoire totalement différent, le bus HYPERTRANSPORT. Comme le montre la Figure 2.15, le bus HYPERTRANSPORT relie les différents processeurs comme s'ils étaient en réseau. Chaque contrôleur mémoire est équipé de cinq sorties : deux servant à le relier à un processeur et en général à son banc mémoire et trois supplémentaires afin de joindre cette unité au reste de la machine. Il peut alors s'agir d'un autre processeur ou encore d'un bus d'entrées/sorties.

Contrairement au modèle suivi par les machines INTEL où l'ensemble des processeurs ne sont reliés à la mémoire et au bus d'entrées/sorties qu'en un seul point, les différents bancs mémoire et les bus d'entrées/sorties peuvent ici être raccordés à la carte mère via des processeurs différents. Les ressources sont donc partagées et ne sont pas accessibles de manière uniforme par tous les processeurs. Certains processeurs devront traverser deux liens avant d'atteindre certaines zones mémoire alors que d'autres n'en franchiront qu'un seul. Cette irrégularité dans les temps d'accès est ce qu'on appelle un effet NUMA(*Non Uniform Memory Access*).

Le nombre de processeurs par machine étant appelé à croître significativement dans les années à venir, l'orientation architecturale des cartes mère se renforce vers la hiérarchisation de l'accès à la mémoire et aux ressources d'entrées/sorties avec des liens d'interconnexion du type des bus mémoire HYPERTRANSPORT ou QUICKPATH. Ainsi, on assiste à une recrudescence des problématiques autour des effets NUMA dans les architectures actuellement disponibles.

2.2.1.2 Les bus d'entrées/sorties

Les bus d'entrées/sorties servent à connecter des cartes d'extension à la carte mère d'un ordinateur. C'est grâce à ces bus que les cartes réseau et autres cartes graphiques ou cartes son vont pouvoir être insérées dans une machine. Jusqu'au début des années 2000, les machines basiques contenaient deux bus d'entrées/sorties différents : un bus AGP(*Accelerated Graphics Port*) destiné à accueillir une carte graphique et un bus PCI(*Peripheral Component Interconnect*) [PCI] pour les autres cartes périphériques,

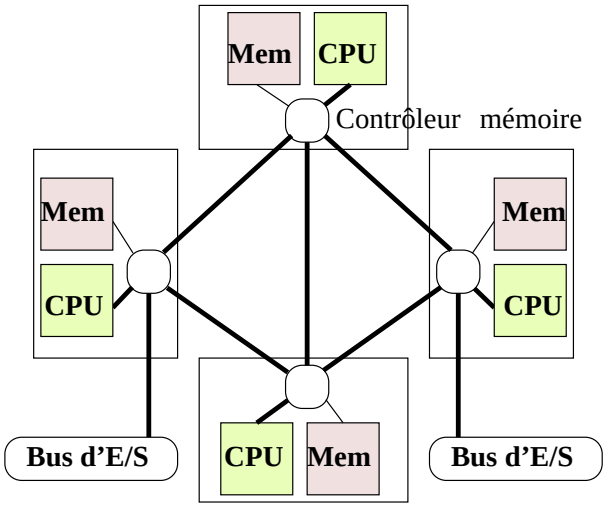


FIG. 2.15: *Architecture multiprocesseur d'*AMD.

ce qui inclut les cartes réseau. Plusieurs versions de bus PCI se sont succédées pour offrir des débits toujours plus élevés : de 133 Mo/s à 2133 Mo/s. Cependant, pour toute cette génération de bus, les échanges de données se font au travers d'un seul et unique lien : l'accès au bus est donc arbitré pour ne permettre que des dialogues sérialisés entre le processeur et les cartes périphériques. La bande passante est par conséquent partagée entre tous les éléments qui lui sont connectés.

Depuis 2004, la norme PCI EXPRESS définit un véritable réseau commuté (*i.e.* plusieurs cartes peuvent être utilisées à la fois) avec une bien meilleure bande passante. En effet, alors que le PCI n'est composé que d'un seul lien bidirectionnel pour l'ensemble des périphériques, le PCI EXPRESS dispose de plusieurs liens indépendants qu'il répartit entre toutes les cartes connectées. Une fois les liens attribués à un périphérique, ce dernier communique alors par échange de paquets comme s'il s'agissait d'un réseau local. L'arbitrage auparavant nécessaire à l'accès au bus PCI est donc remplacé par un commutateur.

2.2.1.3 Bilan

Comme nous l'avons déjà évoqué dans la section 2.1.2.2, les constructeurs ont par le passé allié leurs efforts dans la conception d'un bus unique qui unifierait le bus mémoire et le bus d'entrées/sorties (INFINIBAND). Même si cela n'a pas abouti, le besoin n'en est pas moins présent. Pour l'instant, afin de gommer le fait que deux bus distincts fédèrent l'ensemble des interconnexions d'une machine, AMD propose un connecteur à son bus mémoire HYPERTRANSPORT nommé HTX (*HyperTransport eXpansion*). Ce dernier permet d'échanger des données *directement* grâce à des transferts DMA entre les cartes périphériques et des composants traditionnellement accessibles exclusivement au travers du bus mémoire, à savoir les processeurs et la mémoire. De son côté, INTEL devrait proposer une solution équivalente avec sa prochaine génération de bus QUICKPATH.

L'intérêt de cette uniformisation est de permettre le dialogue d'un composant jusqu'à présent accessible via le bus d'entrées/sorties de la même façon que s'il était directement relié au bus mémoire, *i.e.* en s'affranchissant des traductions d'adresses. Une telle capacité ouvre la voie à la conception de matériels révolutionnaires, à des architectures hétérogènes. En effet, plutôt que de se cantonner à des unités de calculs identiques qui offrent par conséquent des capacités de calcul similaires, les nouvelles architec-

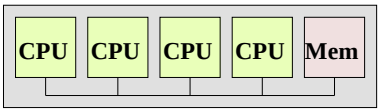


FIG. 2.16: *Architecture* SMP.

tures se parent de composants de capacités différentes. Ainsi, avec des initiatives comme GENESEO d'INTEL et TORRENZA d'AMD, au delà d'assembler des processeurs de nature différente comme dans le ROADRUNNER où CELL et OPTERON cohabitent, les GPU (les processeurs des cartes graphiques) et les cartes accélératrices (*e.g.* FPGA) ne sont dorénavant plus considérés comme des périphériques mais résolument comme des unités de calcul à part entière. Annoncé pour fin 2008, le LARRABEE [SCS+08] est la première réalisation d'une telle machine.

2.2.2 Évolution des unités de calcul

Le but de cette partie n'est pas de retracer l'évolution architecturale des processeurs de manière exhaustive, on peut pour cela se référer à des documents plus spécifiques [HP03, Thi07], mais plutôt de mettre l'accent sur le fait que de nos jours, chaque machine a pratiquement le même potentiel de calcul qu'avait une grappe de PC lorsque ses nœuds étaient encore monoprocesseurs. Il s'agit donc de montrer que cette fulgurante évolution est un nouveau paramètre à prendre en compte dans la conception des logiciels pour le calcul parallèle.

2.2.2.1 Des monoprocesseurs aux multiprocesseurs multicœurs

Alors que seuls les ordinateurs monoprocesseurs existaient, le début des années 60 a vu émerger de nouvelles configurations conjuguant plusieurs processeurs au sein d'une même machine afin d'en augmenter la puissance de calcul. Ainsi, comme le montre la Figure 2.16, ces machines nommées SMP (*Symmetric MultiProcessing*) voient leurs cartes mère s'équiper de plusieurs puces accueillant un processeur de même technologie. Cependant, rapidement, ce modèle introduit des problèmes de contention pour l'accès à la mémoire puisque tous y accèdent via le même bus mémoire. Sont donc apparues des architectures dites hiérarchiques (Figure 2.17) où la mémoire est divisée en plusieurs bancs mémoire interconnectés par le bus mémoire de la machine ou par un réseau d'interconnexion spécifique. Chaque processeur peut alors accéder à toute la mémoire mais de manière non-uniforme : ce sont les machines NUMA(*Non Uniform Memory Access*), qui vont de pair avec les effets NUMA évoqués dans la partie 2.2.1.1.

Parallèlement à cela, les constructeurs n'ont pas cessé de faire monter en puissance leurs unités de calcul en sophistiquant leurs architectures. L'espace libéré sur la puce accueillant le processeur grâce aux progrès acquis en finesse de gravure est activement employé à étendre les circuits internes. Des transistors sont ajoutés afin d'améliorer les opérations sur les flottants ou bien encore les prédictions de branchement, la taille des caches est réhaussée, ainsi que le nombre de pipelines internes, etc. Ce n'est que récemment que les architectes se sont rendus compte qu'il devenait beaucoup moins rentable de continuer à complexifier les différents organes du processeur. En effet, certains points caractéristiques atteignant leurs limites (la taille des caches se révèle à présent suffisante, la prédiction de branchement peut aller jusqu'à côtoyer des taux de réussite de l'ordre de 98%, etc.), la puissance des processeurs ne croît plus aussi spectaculairement qu'avant. Ainsi, plutôt que de persévérer dans cette voie, la finesse

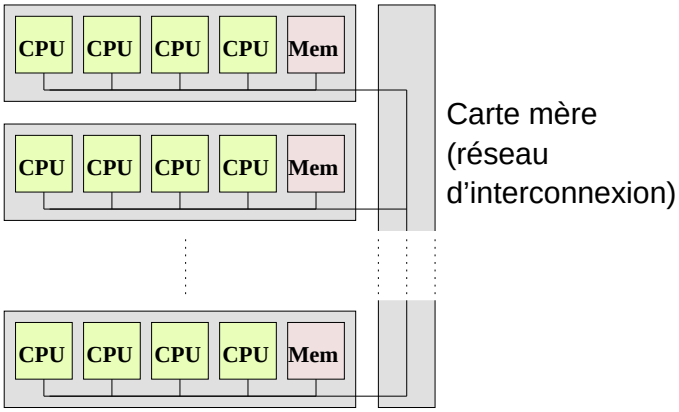


FIG. 2.17: *Architecture* NUMA.

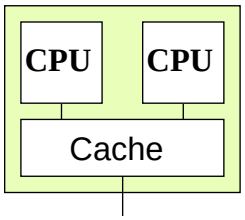


FIG. 2.18: *Puce multicœur.*

de gravure toujours plus fine aidant, l'espace libéré s'est avéré facilement exploitable par l'ajout d'un second puis de plusieurs processeurs sur une même puce, pour un gain de puissance plus significatif. Ces nouveaux modèles d'unité de calcul sont les puces multicœurs (voir Figure 2.18). De nos jours, les machines sont donc naturellement pourvues de multiprocesseurs multicœurs : les quad quad-core qui sont quatre puces équipées de quatre processeurs chacune sont à présent monnaie courante.

2.2.2.2 Conséquences de l'évolution des unités de calcul sur les grappes

Avec l'accroissement du nombre de processeurs par nœud d'une grappe, on assiste actuellement à une recrudescence du nombre de *flots* s'exécutant en parallèle sur une même machine. Cet accroissement au sein de chaque nœud passe par l'augmentation du nombre d'instances d'une application ou par la parallélisation de l'application. Pour ce dernier cas, des interfaces de programmation telles qu'OPENMP [ope] et TBB [tbb] (*Thread Building Blocks*), rendues populaires avec les multicœurs, peuvent grandement aider à la parallélisation des applications préexistantes afin de mieux s'adapter à ces nouvelles plate-formes. L'objectif est, dans les deux cas, il y ait assez de flux d'exécution pour occuper la totalité des processeurs.

L'augmentation de ces flux – que ce soit des processus ou des *threads*– engendre inévitablement une masse de communications supplémentaire qui doit transiter par le biais de mécanismes de mémoire partagée mais aussi au travers des cartes réseau. Néanmoins, alors que les interfaces réseaux étaient généralement associées à un nombre limité de flux d'exécution, leur nombre n'a pas suivi la même évolution que celle du nombre de cœurs. Cette compétition pour l'accès à une même ressource conduit à l'apparition d'un inquiétant goulet d'étranglement.

2.3 Bilan

Les progrès de ces dernières années que ce soit en matière d'unité de calcul ou en matière de réseaux d'interconnexion sont indiscutablement remarquables. En dépit de performances déjà importantes, les constructeurs de réseaux haute performance ont maintenu leurs efforts afin de concevoir des technologies toujours plus sophistiquées qui sont actuellement capables d'atteindre des temps de transfert de l'ordre de la microseconde et des bandes passantes de plusieurs dizaines de gigabits par seconde. Des bibliothèques de communication ont été conçues afin d'exploiter au mieux ces dernières dans des environnements variés, les grappes de PC n'ayant pas toute la même configuration matérielle. Parallèlement à cela, les processeurs ont évolué vers des architectures multiprocesseurs multicœurs qui engendrent une recrudescence incontestable des quantités de données devant transiter entre les nœuds par les mêmes ressources réseaux. Malgré l'ajout de cartes réseau au sein des nœuds afin de rééquilibrer un minimum le nombre d'unités de communication par rapport à celles de calcul, il parait clair que les bibliothèques de communication se sont laissées déborder par cette spectaculaire évolution.

Chapitre 3

De l'optimisation de communications dans les grappes contemporaines

Sommaire

Le succès des grappes de PC n'est pas dû uniquement aux progrès technologiques, il est également fortement lié aux avancées du support logiciel qui permet de les exploiter efficacement. La mise au point au début des années 90 du standard MPI a grandement facilité le développement de nombre d'applications parallèles. Ce succès perdure grâce à la pérennité offerte à ces dernières – beaucoup d'applications de physique telles WRF [WRF] développées en FORTRAN parallélisées grâce à leur portage au-dessus de MPI sont encore utilisées aujourd'hui – et à l'efficacité des implémentations proposées. En effet, ces dernières se révèlent très performantes dès lors que l'on s'en tient aux opérations classiques point à point, offrant ainsi une latence et un débit de premier plan. Toutefois, nous verrons dans ce chapitre que bon nombre d'aspects peuvent encore être grandement améliorés. Notamment, bien peu d'applications utilisent l'ensemble du panel de fonctionnalités offertes par MPI. La faute en revient généralement aux différentes implémentations existantes qui ne les supportent pas correctement. Mais pire encore, les implémentations actuelles se révèlent bien mal armées pour affronter la multiplication des cœurs au sein des machines parallèles contemporaines.

La première section de ce chapitre montre comment les interfaces actuellement proposées gèrent la multiplication des flux de communication. Nous verrons ensuite ce qu'a de problématique le couplage des

échanges de données avec l'activité même de l'application qui les génère sur les schémas de communication ainsi que sur leur progression. Ici, seules les principales implémentations du standard MPI-2 sont considérées.

3.1 Gestion de la concurrence

Avant de considérer l'ordonnancement des communications suivi par les implémentations de MPI actuelles, nous nous intéressons à la manière dont sont absorbées les communications concurrentes par ces interfaces. Comme nous l'avons évoqué dans la Section 2.2.2.2, l'accroissement du nombre de processeurs engendre une masse potentielle de parallélisme et donc des communications. Cela passe par deux modèles d'exploitation de machine différents : l'accroissement du nombre de processus par nœud ou celui du nombre de *threads* par processus, voire même les deux. La première sous-partie expose donc l'interaction qui existe entre les communications provenant de processus différents tandis que la seconde traite des différents niveaux de gestion de la concurrence des *threads*.

3.1.1 Concurrence de processus

Les implémentations de MPI sont des bibliothèques qui se lient aux applications. Ainsi, si plusieurs instances d'une même application sont lancées sur un même nœud, chacune soumettra ses communications à l'instance de MPI à laquelle elle aura été préalablement associée. Le problème d'un tel modèle est l'absence de coordination entre ces différents processus. En effet, comme chacun d'eux s'exécute de manière indépendante, le nombre de processus en concurrence n'influe aucunement sur la façon que ces derniers vont adopter pour traiter leurs communications. Ceci n'implique pourtant pas que les communications ne soient pas elles-mêmes perturbées par l'activité courante des cartes réseau comme le montre la Figure 3.1[1]. Les coefficients de ralentissement de l'exécution d'un processus en fonction de la charge de la machine ont par ailleurs été étudiés durant les travaux de thèse de Maxime Martinasso [Mar07]. De ce fait, la contention au niveau des ressources réseau lorsque plusieurs entités communiquent simultanément par le réseau cause la dégradation des performances que ce soit en latence ou en débit. La soumission intempestive de messages nécessitant un rendez-vous peut monopoliser la carte réseau le temps que la machine distante y réponde, pénalisant l'ensemble des autres applications.

Par ailleurs, le nombre d'unités de calcul s'accroissant et afin de ne léser aucune application au profit d'une autre, il n'est actuellement plus vraisemblable d'affilier l'instance d'une application à chacun des processeurs d'une machine, comme le font jusqu'à présent les implémentations de MPI. Il est désormais nécessaire de faire évoluer ce type de modèle d'exécution vers des solutions où l'accès aux ressources réseau est nécessairement arbitré. Pour cela, il faut soit développer des solutions où la bibliothèque de communication employée est commune à l'ensemble des applications en exécution, soit se tourner vers des solutions hybrides où un seul processus est créé par nœud mais parallélisé.

3.1.2 Support du multithreading

Du côté des applications multithreadées, on s'attend à une meilleure gestion des communications concurrentes puisque le support des *threads* fait partie de la spécification de MPI. La norme décrit plusieurs

[1] Ces mesures ont été réalisées sur des machines quad-cœurs Xeon, les processus sont donc tous ordonnancés.

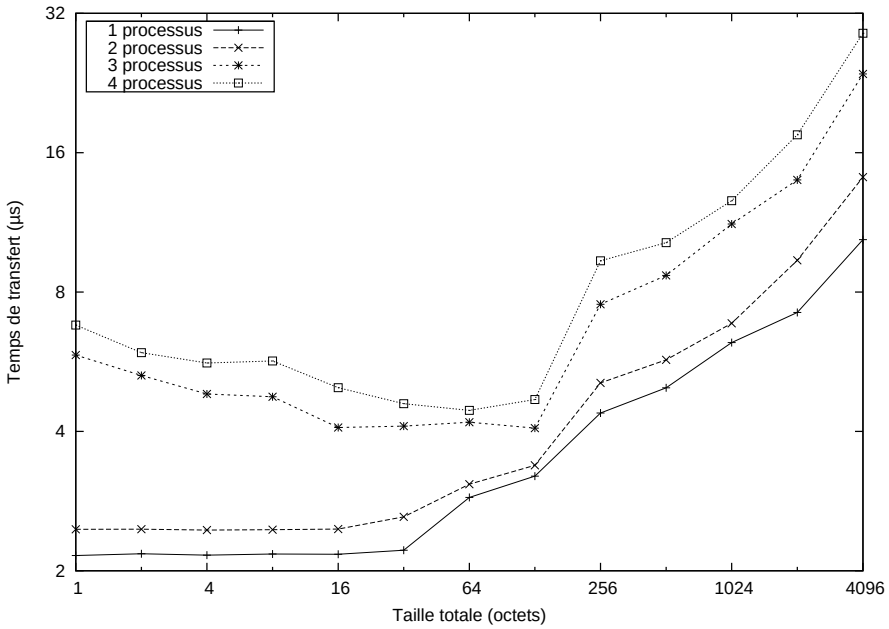


FIG. 3.1: *Mise en concurrence de plusieurs processus exécutant indépendamment un* ping-pong - *Temps de transfert moyen d'un* ping-pong.

niveaux de support :
– le mode MPI_THREAD_SINGLE correspond à une application sans *thread*,
– le mode MPI_THREAD_FUNNELED requiert que seul le *thread* principal d'une application multithreadée ne fasse appel à la bibliothèque de communication MPI,
– le mode MPI_THREAD_SERIALIZED n'autorise qu'un seul *thread* à invoquer une routine de MPI à la fois,
– et finalement le mode MPI_THREAD_MULTIPLE sans aucune contrainte qui correspond à un réel support des applications multithreadées.

Les trois premiers niveaux sont assez proches dans leur fonctionnement, un seul flux étant supposé solliciter l'implémentation de MPI à un moment donné. Cependant, les accès aux bibliothèques ayant des niveaux de gestion de concurrence de type FUNNELED et SERIALIZED sont en général protégés par des mécanismes de verrouillage à gros grain assez pénalisant. Ceci permet de s'assurer de la non-cohabitation des *threads* que pourrait provoquer le développement approximatif d'une application. Ces mécanismes bloquent tout appel à la bibliothèque de communication tant que le traitement de la requête du *thread* précédent n'est pas achevé. Ils ne sont donc pas recommandés pour un niveau de protection plus élevé tel le mode MPI_THREAD_MULTIPLE, qui exige une finesse de verrouillage plus poussée afin de ne faire patienter les différents flux que lorsque cela s'avère nécessaire. Ceci étant beaucoup plus délicat à développer [Arg, GT07, SPH96], la plupart des implémentations de MPI-2 proposées ne supportent pas ce dernier niveau de sécurité de concurrence. En effet, en plus d'exiger la protection en accès de certaines variables, il faut s'assurer que les interfaces de communication bas niveau soient ré-entrantes et arbitrer leur accès si nécessaire. Il ne s'agit donc plus de suivre les demandes de communication de l'application pour invoquer les pilotes des réseaux sous-jacents. La complexité de ces opérations a bien évidemment un coût comme on peut le voir sur la Figure 3.2. Ainsi, les développeurs d'applications préfèrent souvent utiliser une version de MPI avec des capacités de support aux *threads* moindre afin de gagner en performance, au détriment d'une utilisation plus aisée, ce qui par conséquent n'incite pas

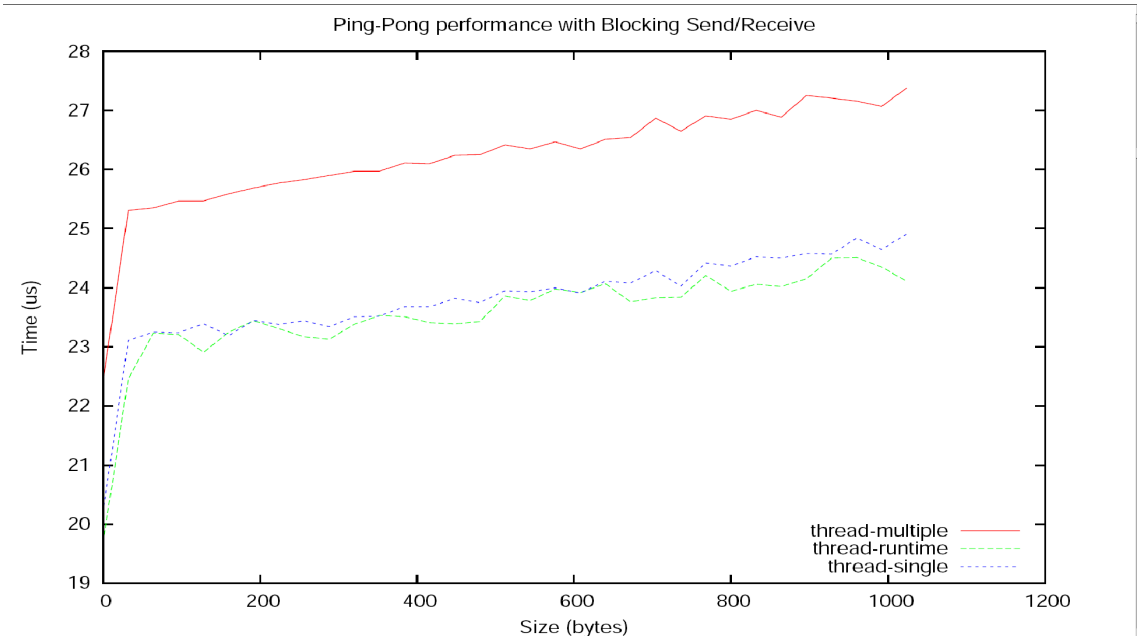


FIG. 3.2: *Surcoût induit par les différents niveaux de support à la concurrence au sein de* MPICH2. *La courbe thread-runtime correspond à une option de configuration de* MPICH2. *La version ainsi obtenue est de niveau MPI_THREAD_MULTIPLE, mais est par défaut initialisée au niveau MPI_THREAD_FUNNELED.*

les développeurs d'implémentations de MPI à offrir ce type de fonctionnalités. Actuellement, le cercle des implémentations de MPI-2 assurant le support aux *threads* de niveau MPI_THREAD_MULTIPLE se limite à OPENMPI/TCP, MPICH2/SOCK/TCP et MVAPICH sur TCP et INFINIBAND[2].

3.2 Des échanges de données assujettis par l'application

Même si les implémentations *génériques* de MPI ne font *a priori* pas de traitement particulier en fonction de la technologie réseau sur lesquelles elles s'appuient, la Figure 2.12 montre qu'elles ne sont pas pour autant inefficaces.

En premier lieu, elles savent en général tirer correctement parti des interfaces de communication bas niveau sur lesquelles elles s'appuient. En effet, si elle n'est pas déjà faite à bas niveau, la sélection de la méthode de transfert des données se fait en adéquation avec la taille du message à transférer. Comme l'exposait la Section 2.1.2.1, plusieurs méthodes peuvent être utilisées afin de faire transiter des données au sein même de la machine – les données transitent entre la mémoire principale et celle la carte réseau grâce à des transferts par PIO ou par DMA – ainsi qu'à distance en employant un protocole de communication par copie ou par rendez-vous. La Figure 3.3 donne une illustration de l'allure que chacune de ces combinaisons peut prendre. Intuitivement, l'envoi d'un message doit se faire via la méthode de transfert la moins coûteuse en temps. En réalité, l'allure de ces courbes dépend des caractéristiques du réseau sous-jacent. À tailles de données égales, certains les feront transiter par copie alors que d'autres

[2]Plusieurs implémentations du standard MPI-1 supportent également le niveau MPI_THREAD_MULTIPLE. Pour plus de détails, on peut se référer à la thèse de Guillaume Mercier [Mer04]

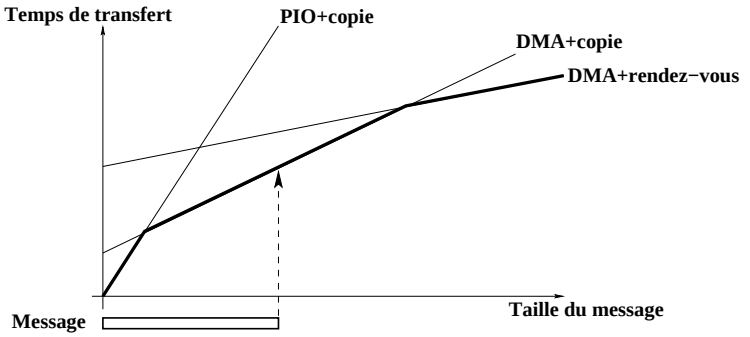


FIG. 3.3: *Allure du comportement de différentes méthodes de transfert.*

utiliseront un rendez-vous. Ainsi, le choix de la méthode de transfert à employer se fait en accord avec les caractéristiques du réseau concerné.

En second lieu, et là réside la clé des performances annoncées aux utilisateurs, une réduction à l'extrême du chemin critique permet de quasiment faire appel directement aux interfaces bas niveau. Une requête d'envoi de message faite auprès d'une implémentation de MPI se traduit pratiquement immédiatement par un appel à l'interface de communication sous-jacente. Le chemin critique se réduit à la construction des entêtes de messages nécessaires au bon fonctionnement du protocole établi par l'implémentation et à un aiguillage vers le pilote réseau sélectionné qui se chargera de faire l'appel au réseau. Ainsi, ces interfaces de communication atteignent des performances brutes en latence et bande passante quasi similaires à celles des interfaces bas niveau.

Cependant, si l'envoi brut de message se fait dans les meilleures conditions, les applications réelles ne se contentent pas de schémas de communication réguliers au point de n'avoir que des échanges totalement indépendants les uns des autres, *i.e.* qui ne sont jamais concurrents. Un problème majeur de cette approche est donc l'accès aux ressources réseaux laissé totalement libre aux différents flux de communication. Ce mode de fonctionnement est en totale opposition avec le contexte présenté dans la Section 2.2.2.2 où l'on dépeint un environnement où le nombre de communications croit avec le nombre de *threads* de l'application et le nombre de cœurs de la machine. Cette utilisation chaotique des cartes provoque inexorablement des perturbations sur les transferts proprement dits.

Le découplage des communications réseau de celles requises par les applications est donc nécessaire afin de pouvoir organiser un système d'arbitrage d'accès aux ressources réseaux. En plus de cela, le manque de recul qu'impose cette manière de faire supprime toute opportunité d'optimisation du schéma global de communication : que ce soit au sein d'un même flot ou sur leur ensemble. Les communications étant directement traitées, elles ne peuvent pas être différées afin d'être agencées avec d'autres. Pour de mettre en évidence le profit pouvant être tiré de telles opportunités, la suite de cette section est consacrée à l'étude de schémas de communication parallèles simples pour lesquelles des optimisations seraient profitables.

3.2.1 Opportunités d'ordonnancement

En considérant ne serait-ce qu'un seul flux de communication, plusieurs opérations peuvent lui être appliquées. Intuitivement, on voudrait pouvoir grouper, permuter plusieurs communications ou encore en fragmenter certaines pour en compléter d'autres, comme si l'on voulait remplir optimalement les

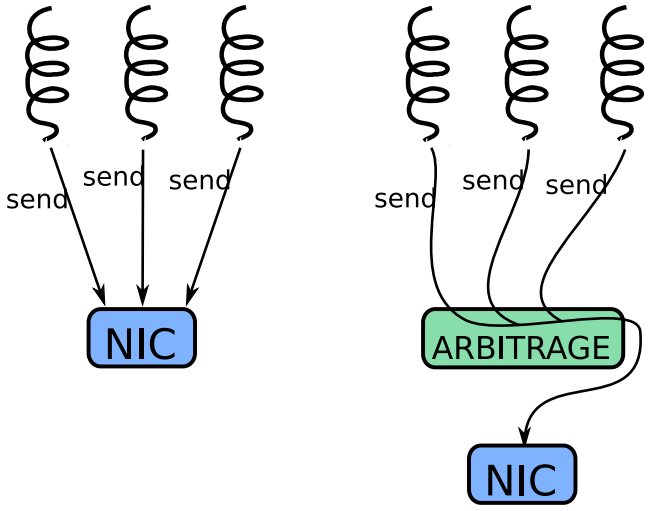


FIG. 3.4: *L'accès aux ressources de communications doit être arbitré.*

wagonnets d'un manège avec des groupes de personnes dissociés.

Mais encore au-delà, si on ouvre le spectre d'application des optimisations d'ordonnancement à l'ensemble de *toutes* les communications vers une même machine distante, les opportunités sont encore plus importantes puisque le nombre de messages à considérer à un moment donné ne peut être que plus conséquent. Toutes les opérations citées peuvent alors leur être appliquées sans aucun problème d'inter-dépendance. Il s'agit finalement de multiplexer l'ensemble des flux de communication d'une même source vers une autre. L'idée de ces mécanismes peut être modélisée par la Figure 3.4. De plus, le fait de multiplexer toutes les communications ayant la même cible permet de n'avoir qu'un nombre restreint – voire unique – de ressources d'entrées/sorties ouvertes vers cette dernière. Cette économie de ressources permet d'envisager le passage à l'échelle des interfaces de communication dans le contexte de grappes de PC ayant de plus en plus de nœuds.

La fin de cette section est consacrée à la mise en évidence des différentes opérations qui pourraient être apportées à l'ensemble des transferts d'une application.

3.2.1.1 Agrégation de messages

Agréger des messages consiste, dans notre terminologie, à fabriquer un message contigu à partir de données dont le seul point commun est d'être à destination de la même machine, et pas forcément du même processus. En pratique, cela consiste à copier les données auxquelles sont ajoutés des entêtes dans un tampon commun (ce qui revient à une opération *gather* classique), les entêtes ayant pour rôle de permettre au récepteur de distribuer les données vers leurs zones de destination (ce qui correspond à une opération *scatter*).

Comme le montre la Figure 3.5, si chaque message est traité de manière indépendante, il en résulte un envoi physique sur le réseau pour chaque message. Pourtant, si les messages étaient agrégés et envoyés en un seul bloc, le temps de transfert additionné au temps nécessaire à la copie des données dans un tampon contigu pourrait se révéler inférieur au temps de transfert précédent. La levée de cette hypothèse réside dans l'allure de la courbe de transfert du réseau sous-jacent. Les situations pouvant profiter d'une telle

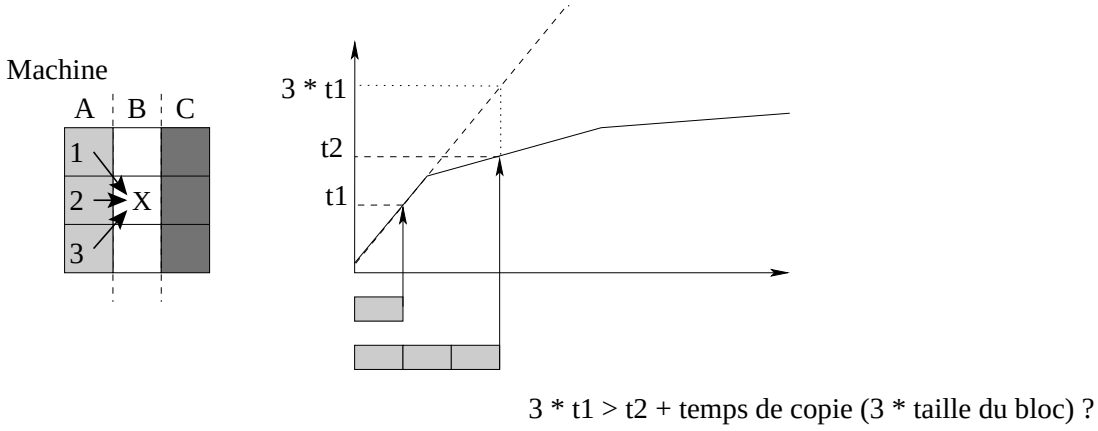


FIG. 3.5: *Les données 1, 2 et 3 de la machine A doivent être envoyées à la machine B pour contribuer au calcul de la donnée X. En sachant que ces 3 blocs doivent transiter, l'allure de la courbe de transfert du réseau sous-jacent permet de déterminer quel est l'ordonnancement le plus judicieux.*

optimisation sont courantes. Il s'agit en l'occurrence de toutes les applications ayant besoin d'échanger fréquemment des messages courts. Par exemple, l'utilisation de BLAS [bla] pour une multiplication de matrices ou bien l'application d'un filtre à une image comme celui de Perlin [PV95] nécessitant la consultation du voisinage et donc l'échange de *bandes* de données occasionnent une multitude de messages à émettre successivement.

Même si ce n'est pas le comportement par défaut des bibliothèques actuelles, ce type de schéma de communication fait toutefois l'objet d'une attention toute particulière dans la spécification même de MPI. En effet, la définition de types dérivés complexes, *i.e.* les *derived datatypes* [GLS, LS99] permet de décrire en une seule opération l'envoi d'un ensemble de messages. Ce dernier peut alors être considéré globalement, ce qui ouvre l'opportunité à l'application de stratégies d'ordonnancement de différentes natures. Elles peuvent consister en la copie des messages dans un tampon intermédiaire avant redistribution vers leur emplacement mémoire final, avec ou sans prise de rendez-vous préalable, mais aussi en des envois indépendants, ou encore par blocs de taille homogène, etc. ou tout simplement utiliser la technique de grouper/éparpiller (*gather/scatter*) si elle est proposée par le réseau sous-jacent. Cette dernière technique permet de transmettre directement à la carte réseau la liste des segments à envoyer qui vont être rassemblés à la volée grâce à un DMA chaîné pour n'être transmis qu'en un seul et unique message logique au niveau du réseau. Finalement, ces types dérivés offrent sur un ensemble de messages fini l'opportunité d'appliquer des optimisations. Cette idée a d'ailleurs été reprise pour aider l'ordonnancement des entrées/sorties avec MPI-IO, un enrichissement du standard MPI-2 [TGL98].

Même si certaines implémentations de MPI-2 comme MPICH2 [BGST03, BSTG06] apportent un soin particulier à la mise en œuvre de ce type d'opérations en allant jusqu'à prendre en compte les coûts d'accès aux données, la plupart ne les gère – ou du moins ne les gérait – pas correctement par le passé. Les développeurs des applications ont donc pris le parti de ne pas les utiliser et il est ainsi courant de voir des optimisations faites *à la main* au lieu de les employer [TG07]. Ces optimisations, en plus d'être ardues en terme de programmation, ont le désavantage de n'être absolument pas portables puisque développées spécifiquement en fonction du réseau sous-jacent.

C'est d'ailleurs ce genre de considération qui avaient déjà été à l'origine des premières versions de la bibliothèque de communication MADELEINE. En particulier, MADELEINE 3 [ABD+02] proposait

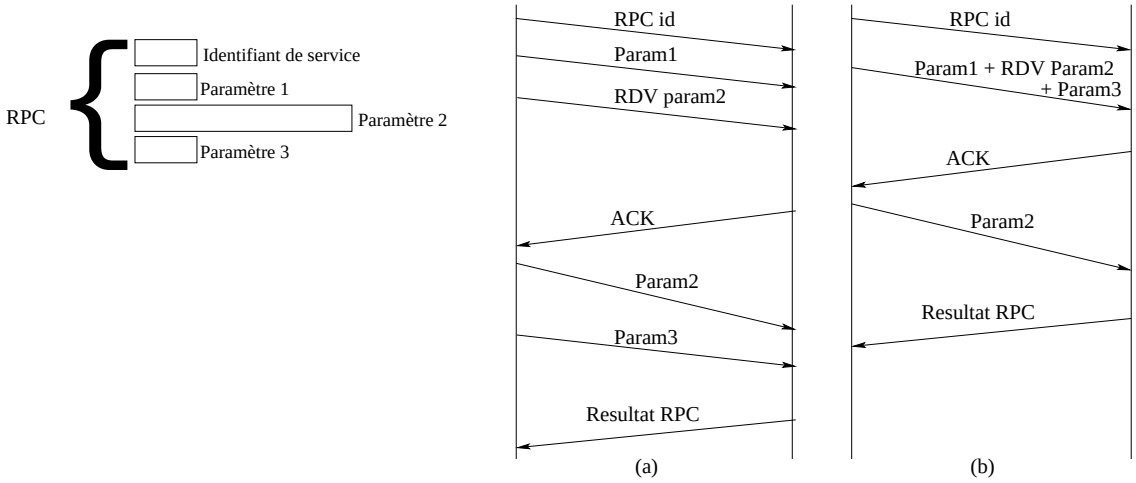


FIG. 3.6: *Une communication non nécessaire aux suivantes peut bloquer l'ensemble des transferts concurrents - Illustration du cas des RPC.*

d'accumuler et d'agréger les messages provenant d'un même flux de communication. L'envoi des paquets ainsi formés était déclenché lorsque l'application le spécifiait au niveau de l'interface utilisateur ou lorsque le tampon en charge de recueillir les données agrégées venait à être plein. Le protocole SDP (Socket Direct Protocol) [sdp] ainsi que la version utilisant l'algorithme de Nagle de TCP [Nag] utilisent également l'agrégation pour améliorer les temps de transfert des messages de petite et moyenne taille qui leur sont confiés [BBP+07].

Plus récemment, MVAPICH2 [HSJ+06, mva] a également introduit ce type de mécanisme pour améliorer ses communications par RDMA. MVAPICH2 joue sur le flou qui existe dans la spécification de MPI-2 : comme le moment et l'ordre dans lequel les transferts doivent être effectués ne sont pas précisés, MVAPICH2 profite du laps de temps entre le dépôt des communications et les barrières de synchronisation pour accumuler des messages et par la suite appliquer un nouvel ordonnancement [HSJP05]. Les messages peuvent alors être agrégés et même réordonnés.

3.2.1.2 Permutation de messages

Toujours dans l'objectif de réduire le temps de transfert global des messages, la question de l'ordre de leur envoi peut se poser. En effet, faire passer une communication avant une autre avec, par exemple, l'idée de l'agréger à la précédente peut s'avérer judicieux.

Les appels à procédure à distance (*Remote Procedure Call*, RPC) [rpc], couramment employés à travers d'intergiciels (ou *middlewares*, en anglais) tels que CORBA [Vin97], JAVA RMI [rmi], etc, peuvent être pris comme illustration. Chacun de ces appels est composé dans un premier temps de l'envoi du numéro de service à invoquer, puis de ceux des paramètres nécessaires. Ceci produit donc une série d'envois successifs indépendants les uns des autres. En admettant que l'un des paramètres de l'appel à distance soit d'une taille nécessitant une prise de rendez-vous préalable, il est possible que les communications suivantes soient bloquées par le traitement de cette dernière, comme le montre la Figure 3.6(a) : ici, l'envoi du troisième paramètre (`Param3`) l'est par le second paramètre (`Param2`). En revanche, en autorisant la permutation des messages, plusieurs schémas de communication peuvent être envisagés. La

Figure 3.6(b) propose une alternative au schéma de communication régulier : en permutant l'envoi des paramètres 2 et 3, les paramètres 1 et 3 ainsi que le message de contrôle qui initie la prise de rendez-vous du paramètre 2 peuvent alors être agrégés. Mais d'autres combinaisons peuvent être examinées. Le choix de la meilleure combinaison réside toujours dans le comportement des réseaux disponibles.

La permutation de messages est un cas plus complexe que celui de l'agrégation car il requiert de la part du récepteur la faculté de savoir réordonner les messages afin de les délivrer correctement à l'application. En effet, les communications d'émission sont postées en concordance avec celles de réception par l'application. Ainsi, les réagencements effectués par l'interface de communication doivent rester internes. Peu d'interfaces de communication proposent un tel support. Nous avons précédemment vu dans la Section 3.2.1.1 que MVAPICH2 est capable de réordonner ses transferts afin de les optimiser. Cette technique est plutôt utilisée dans des protocoles tels que TCP/IP qui nécessitent de la tolérance aux fautes et pannes [GPL04, AT04, BA02]. Cela est aussi utile au support de la qualité de service où la priorité exprimée sur certains messages doit les faire passer avant d'autres déjà postés par l'application.

3.2.1.3 Distribution de messages sur différentes cartes réseau

L'évolution technologique des nœuds composant une grappe de PC a fait croître le nombre de leurs cartes réseau. Certaines grappes, comme la grappe japonaise T2K OPEN SUPERCOMPUTER [Nak08], peuvent à présent compter jusqu'à quatre rails (*i.e.* cartes réseau) différents. Les grappes sont pour la plupart équipées de cartes de même technologie même si certaines plus expérimentales, comme la grappe BORDERLINE du projet français GRID5000, proposent des technologies réseaux hétérogènes au sein d'un même nœud. Afin de maximiser l'emploi de l'ensemble des ressources de communication et donc le débit potentiel de la machine, une distribution des messages peut être faite.

Plusieurs projets ont déjà travaillé autour de cette problématique. LA-MPI [ADD+04, CFP+01] est une implémentation de MPI capable d'envoyer des messages sur des réseaux hétérogènes. Elle est également capable de découper un message et de l'envoyer sur différentes ressources de communication mais uniquement de même technologie. Ce projet fait à présent partie du consortium OPENMPI, qui lui aussi, gère le découpage de messages et propose l'envoi des fragments sur des technologies hétérogènes. MPICH-VMI2 [PP02] est également une implémentation de MPI qui propose un support de gestion des communications sur réseaux hétérogènes. MUNICLUSTER [Moh05] permet quant à lui d'équilibrer la charge entre plusieurs cartes réseau (de même technologie ou pas) et met en pratique des stratégies de découpages de messages mais seulement au-dessus des interfaces sockets. MVAPICH [LVP04, VSH+05] a également scindé son architecture logicielle en deux branches dont l'une exploite plusieurs cartes INFINIBAND pour ses communications uni-directionnelles. Cependant, lorsqu'aucun support aux machines multirails n'est offert les processus MPI sont tout simplement affiliés à une carte réseau particulière, ce qui subdivise la contention mais ne modifie aucunement la distribution des paquets en fonction de leur engorgement. Seuls les mécanismes de tolérance aux pannes permettent à un processus de voir ses communications partir sur une carte réseau différente. Les interfaces de communication bas niveau proposent également un support basique du multirail : MX coupe les messages longs puis les envoie sur plusieurs cartes et ELAN, en plus de cela, équilibre l'envoi des messages courts sur les différentes cartes suivant l'algorithme du tourniquet.

3.2.1.4 Bilan

Bien que les interfaces de communication peaufinent particulièrement les envois bruts de données afin d'assurer des latences et bandes passantes quasi optimales, elles ne se comportent pas aussi bien dès que les schémas de communication se compliquent. Le manque de recul dû à leur conception orientée vers l'optimisation à l'extrême de leurs performances brutes ne leur permet pas de profiter des communications concurrentes pour appliquer des optimisations à leur ordonnancement. Nous avons ainsi ouvert la discussion sur les possibilités d'amélioration de cas pathologiques qui échappent totalement à ces interfaces de par cette conception et montré que les opportunités d'optimisation de ces schémas de communication sont fréquentes. Un découplage des requêtes d'échange de données faites au réseau de celles faites à l'interface de communication de niveau intermédiaire permettrait d'optimiser l'ensemble des communications d'une même application.

3.2.2 Progression des communications dirigées par l'application

De la même façon que les demandes de transfert sont directement soumises au réseau après appel à l'interface de communication, le réseau est interrogé sur la terminaison de ces dernières sur demande de l'application. Deux méthodes de détection de fin de communication sont en général proposées : l'une est fondée sur appel bloquant à l'interface de communication bas niveau tandis que la seconde procède par scrutation.

Les appels bloquants consistent à déposer une requête de complétion à la carte réseau et à attendre une interruption de sa part la signalant. Cette méthode a l'avantage de ne pas monopoliser de ressource processeur le temps que la communication considérée s'achève mais souffre cependant d'un surcoût certain induit par le traitement de l'interruption elle-même. De plus, le système ne garantit pas que le *thread* communiquant soit ordonnancé dès l'arrivée des données : s'il y a assez de *threads* actifs pour alimenter tous les processeurs, le *thread* communiquant peut ne pas reprendre son exécution immédiatement.

La fin d'un transfert peut également être détectée par scrutation, qui consiste à surveiller activement un emplacement mémoire. Cette méthode a l'avantage de n'introduire qu'un surcoût unitaire très léger (ce qui généralement revient à consulter le contenu d'une case mémoire) mais nécessite la contribution d'un processeur qui pourrait être utilisé par ailleurs pour du calcul. Ainsi, pour garantir une bonne réactivité, le *thread* effectuant la scrutation doit être ordonnancé fréquemment, ce qui ne peut être garanti si le nombre de *threads* en concurrence est plus important que celui de processeurs. Par conséquent, la multiplication des *threads* peut entraîner une surcharge de travail pour la carte réseau et ses temps de réponse peuvent donc se voir fortement allongés du fait que les requêtes soient traitées séquentiellement.

Classiquement, la méthode employée par les applications afin d'assurer la progression de leurs communications consiste à ajouter un *thread* dédié à cette tâche. Ceci a l'avantage de donner l'opportunité de détecter le plus tôt possible les évènements réseaux et donc d'accélérer leur traitement. Les communications progressent en tâche de fond des calculs opérés par les autres *threads* (voir Figure 3.7) : c'est ce qu'on appelle le recouvrement. En effet, dans un tel environnement, si la progression des communications est faite par les *threads* de calcul eux-mêmes au moment où ces derniers ont besoin des données en faisant l'objet, rien ne se sera passé entre-temps et le *thread* devra attendre à ce moment-là le temps de transfert nécessaire. Récemment, ce dernier point a été amélioré par l'introduction d'un *thread* de progression interne dans certaines implémentations de MPI (en l'occurrence OpenMPI [WGC+04, SWG+06, YWGP05] et MVAPICH2 [HSJ+06] uniquement) et même au sein

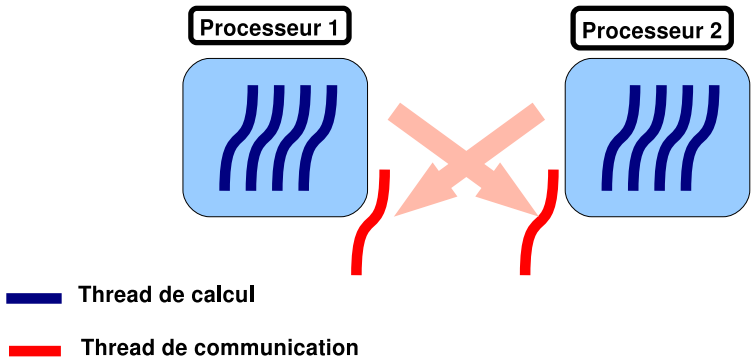


FIG. 3.7: *Un* thread *est dédié aux réceptions afin de les faire progresser en tâche de fond.*

même d'interfaces de communication bas niveau telles MX [com03][3]. Dans le cas d'implémentations de MPI ne fournissant pas de support de *thread* complet, ce même *thread* est chargé de faire les émissions comme les réceptions en plus de la progression comme le montre la Figure 3.8. Néanmoins, l'introduction de ce *thread* peut paradoxalement gêner la progression de communications en arrière-plan si la machine est chargée. En plus de ralentir les calculs, ce dernier risque de n'être alors ordonnancé que rarement, ralentissant globalement les échanges des données.

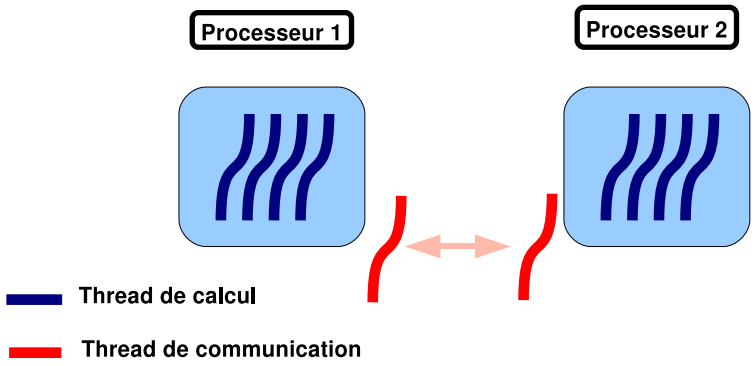


FIG. 3.8: *Un* thread *est chargé d'assurer toutes les communications.*

Finalement, une réelle gestion des flux de communication en parallèle – et non en concurrence – serait profitable dans le contexte des architectures à présent multicœurs. En effet, la simple prise en charge de communications en concurrence pour ensuite les traiter en interne de manière séquentielle n'est pas suffisante pour offrir un bon support de communication aux applications. La progression des transferts en arrière-plan des phases de calcul garantit l'amélioration de la réactivité et des zones de recouvrement pour une meilleure exécution de l'application. Les solutions actuellement proposées restent généralement insuffisantes de ce point de vue.

[3]D'autres implémentations, datant de la première version standard MPI ont également fait ce genre d'effort. Ces dernières sont recensées dans l'état de l'art de la thèse de G. Mercier [Mer04]

3.3 Pour un meilleur traitement des communications !

L'expansion des architectures multicœurs a entraîné une véritable révolution en matière de parallélisme. D'un point de vue local, des modèles de programmation différents sont apparus afin d'aider les applications à tirer parti de manière optimale de toutes les ressources de calcul proposées. Une de ces propositions consiste à paralléliser une application en utilisant de processus légers et de les distribuer au travers de l'ensemble de la machine. La multiplication des flux de communication induite par l'introduction de *threads* engendre donc une augmentation considérable des échanges de données alors que les ressources de communication n'ont de leur côté pas été grandement revalorisées. Ainsi, nous proposons ici de repenser la façon dont sont actuellement gérées les communications. Il apparaît de manière certaine que cela doit tout d'abord passer par un arbitrage de l'accès aux cartes réseau et le découplage de l'application et de l'activité réseau. L'accès sauvage de flux de communication aux ressources d'entrées/sorties offertes ne peut plus perdurer dans le sens où non seulement cela pénalise les transferts concurrents, mais également écarte toute occasion d'optimisation. Pour les mêmes raisons, la progression des communications doit faire l'objet d'une attention toute particulière. En effet, plus vite les ressources réseau sont libérées, plus vite elles peuvent être réemployées à d'autres opérations. L'objectif est ici de toujours chercher à améliorer le schéma de l'ensemble des communications de l'application courante en exploitant au maximum les ressources – d'entrées/sorties mais aussi de calcul – offertes.

Deuxième partie

Contribution

Chapitre 4

Vers une nouvelle façon de penser les communications

Sommaire

Le chapitre précédent nous a amenés à proposer une nouvelle façon d'appréhender les communications. Avant de présenter la bibliothèque que nous avons mise au point dans les chapitres suivants, nous en introduisons ici les points clés. Nous décrivons comment nous envisageons le découplage de l'activité réseau de celle de l'application en retardant les soumissions de requêtes de communication aux cartes réseau – que ce soit des envois, des réceptions ou des opérations en vue de faire progresser les communications courantes – tant que ces dernières sont déjà occupées à traiter des transferts en cours. Il s'agit finalement de ne solliciter les cartes qu'à des moments où cela ne perturbera pas les communications déjà entamées. Une fois le socle de notre bibliothèque de communication fondé, nous verrons comment profiter du délai introduit pour appliquer dynamiquement des optimisations aux différents flux de communication de l'application en se basant sur les caractéristiques des réseaux sous-jacents. Finalement, la dernière partie de ce chapitre se focalise sur l'exploitation des machines multicœurs d'aujourd'hui et de demain avec une gestion des flots de communication concurrents et le déploiement de la plate-forme de communication sur le maximum de ressources de calcul possible.

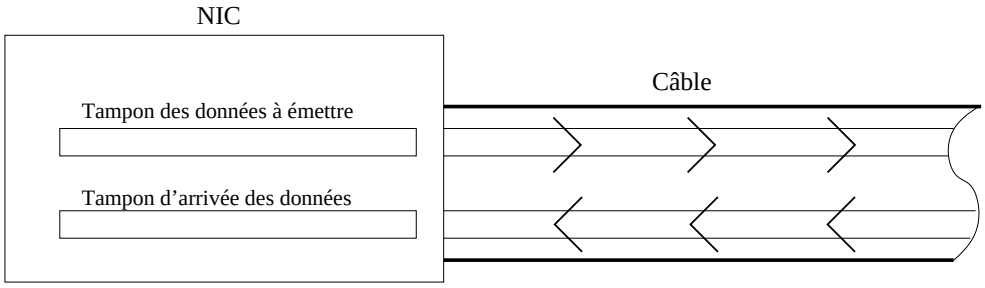


FIG. 4.1: *Carte réseau full-duplex : les données circulent dans les deux sens en même temps.*

4.1 Découpler les communications de l'application

Cette section présente le socle sur lequel toute la partie optimisation de communication va par la suite reposer. Il s'agit de concevoir un moteur de communication basé non plus sur la soumission des requêtes par l'application mais sur le rythme d'exécution des cartes réseau.

4.1.1 Suivre l'activité des cartes réseau

Afin d'améliorer les échanges entre les machines, les technologies réseau actuelles établissent des liaisons bidirectionnelles (aussi nommées *full-duplex*) entre chaque paire de machines devant échanger des données. Comme le montre schématiquement la Figure 4.1, les émissions et les réceptions ne transitent pas par les mêmes tampons de communication et ne sont pas en concurrence sur le câble. Les données pouvant ainsi circuler dans les deux sens à tout moment, les deux machines peuvent communiquer sans avoir à se partager l'accès au câble et donc à attendre que l'autre l'ait libéré.

En émission, avant d'être transmises sur le câble lui-même, les données transitent dans la carte via un tampon de transmission de taille assez réduite (*e.g.* de l'ordre de 10 Ko sur MYRI-10G). La carte peut grossièrement être considérée comme saturée lorsque ce tampon est plein, et occupée lorsqu'il n'est pas totalement vide. Même si ce n'est pas tout à fait exact, nous considérons qu'il se vide lorsqu'une lorsque la terminaison d'une communication est détectée car c'est la seule information exploitable qui remonte jusqu'à notre niveau.

De manière quelque peu différente en réception, la soumission excessive de requêtes peut également mener la carte réseau à saturation. Le problème ne va pas résider dans l'accès au réseau mais à la prise en considération des requêtes : ce qui implique leur stockage, des parcours nécessaires à la réidentification des requêtes avec les données venant d'arriver, etc. Vient s'ajouter à cela le cas particulier des messages nécessitant une prise de rendez-vous préalable. En effet, le fait d'acquitter plusieurs requêtes de ce type successivement peut provoquer un engorgement à l'arrivée des données dans la carte, le temps que ces dernières soient expédiées de la carte réseau à la zone mémoire utilisateur de destination.

De plus, les cartes réseau sont pour l'instant équipées d'un processeur unique. Celui-ci est donc seul à enregistrer les requêtes soumises, remplir le tampon, initier les transferts, gérer les retransmissions, etc. La surcharge de requêtes de communication ne peut ainsi que perturber le bon déroulement des requêtes déjà entamées. Plus de communications – qu'elles soient en émission ou en réception – sont en concurrence et plus les contentions sur la carte sont importantes.

À partir de cela, il est facile d'appréhender le fait qu'il n'est pas utile de soumettre plus de communications que la carte n'est en capacité de traiter. En émission, il s'agit de modérer la masse des données à mettre *sur le fil* à un moment donné, *i.e.* tant que le tampon de transmission de la carte est somme toute plein ; et en réception, la masse des données devant transiter par la carte avant d'atteindre leur destination finale. Nous proposons ici de réguler les demandes de transfert par carte réseau en les comptant, tout simplement. Ce nombre déterminant l'état de la carte – libre ou occupée – est fixé à l'implémentation du support du protocole bas niveau en fonction de ses caractéristiques. En effet, certaines interfaces de communication bas niveau ont la faculté de *pipeliner* leurs envois, *i.e.* le remplissage du tampon de transmission s'effectue en même temps que ce dernier se vide : le tampon n'est donc jamais vide. Ainsi, si la chaîne de ce procédé n'est pas altérée par un tarissement des données à envoyer – *i.e.* que suffisamment de travail est continuellement donnée à la carte –, on peut arriver à une activité optimale de cette dernière. Il est donc dans certaines circonstances intéressant de soumettre des requêtes d'émission à l'avance, propriété que doit prendre en compte une bibliothèque de communication. Le niveau de régulation aurait aussi pu être caractérisé par une masse de données plus que par un nombre de requêtes mais la taille du tampon de transmission ainsi que la taille des entêtes propres au protocole adopté ne sont en général pas communiquées car elles dépendent de l'implémentation de l'interface bas niveau et sont donc susceptibles de changer lors de mises à jour.

Finalement, au lancement ou lors de reprise d'échanges de l'application, le nombre de requêtes va servir à amorcer la carte. Puis, le reste du temps, c'est sur demande de la carte, *i.e.* lorsqu'elle aura achevé une requête, que la suivante lui sera donnée. Ainsi, si l'application produit régulièrement des communications, la carte atteint rapidement un régime permanent optimal en terme d'occupation. La carte réseau se place alors comme maître et non plus esclave dans le traitement des communications.

4.1.2 Constituer une fenêtre de travail

Le fait de ne pas soumettre directement les requêtes de communication à la carte ne change rien du point de vue de l'application. Si cette dernière effectue des opérations non bloquantes, les requêtes doivent être prises en charge par la bibliothèque de niveau intermédiaire en attendant d'être traitées à plus bas niveau sans la faire attendre. S'il s'agit d'opérations bloquantes, elle devra pareillement attendre la complétion de ces dernières : que le temps écoulé soit passé en attente dans la bibliothèque intermédiaire ou en attente de soumission dans celle de bas niveau ne change rien.

Ainsi, tant que la carte est occupée, les différents messages en émission peuvent facilement être accumulés, constituant une fenêtre de travail de messages qui pourront par la suite être combinés afin d'optimiser leur envoi (voir la section 4.2).

Les réceptions de messages longs sont également différées. Même si le potentiel d'optimisation est moins étendu que dans le cas de l'émission, un message qui a été annoncé par l'émetteur via un rendez-vous n'est acquitté que lorsqu'au moins une carte est considérée comme libre et qu'il est premier dans la liste des messages en attente de réception. À ce moment-là, la stratégie d'ordonnancement courante peut faire le choix de le recevoir en un ou plusieurs fragments sur des réseaux différents afin de répartir la charge sur les cartes disponibles et ainsi d'augmenter la bande passante.

4.2 Multiplexer les différents flux de communication

Une fois les messages accumulés dans la fenêtre de travail, comme l'expose la Section 4.1.2, ces derniers sont en attente de la libération d'une carte pour émission. Plutôt que de les envoyer séparément suivant un ordre FIFO, nous proposons de profiter de ce laps de temps pour leur appliquer quelques optimisations.

4.2.1 Définition du protocole de communication

Comme l'expose la Section 3.2.1, l'optimisation des schémas de communication peut passer par leur réordonnancement. Comme l'ordre d'envoi des messages ne respecte plus celui suivi par l'application lors de ses dépôts, le récepteur – qui ne connaît que celui-ci – n'est plus en mesure d'identifier par lui-même les données qui lui parviennent. Il est donc nécessaire de mettre au point un protocole de communication spécifique qui permette à l'émetteur de manipuler les données à sa guise tout en permettant au récepteur de décoder ces desseins.

Contrairement aux nombreux travaux sur les protocoles bas niveau qui s'évertuent à réduire le format de paquet à leur plus simple expression dans l'objectif d'offrir des performances brutes toujours plus proches des capacités limites de la carte, l'introduction de telles manipulations va nécessiter l'ajout d'informations aux données en transit. Ces informations sont un mal dans le sens où les performances vont être dégradées (on verra néanmoins dans la partie consacrée aux évaluations que cela reste tout à fait raisonnable) sur des tests basiques – qui ne tirent pas parti d'ordonnancements plus pointus que ceux décrits à la base – mais indéniablement pour un bien puisque cela ouvre la voie à des optimisations qui pourront bénéficier aux applications complexes. Nous acceptons ici de faire un compromis avec l'ambition d'améliorer globalement l'ensemble des schémas de communications pouvant être rencontrés.

De plus, afin de laisser libre cours à toutes les possibilités d'optimisation, il paraît important de modéliser ce nouveau protocole de manière assez souple. En effet, pour l'instant, nous n'avons parlé que d'agrégation et de permutation de messages mais on peut également considérer des opérations de subdivision ou encore des opérations permettant le rebond de messages dans le cas de passerelles afin, par exemple, d'équilibrer la charge sur des liens de communication en empruntant différentes routes. Il est par ailleurs indispensable de penser à un format de paquet complet – ou du moins flexible – qui permette l'encapsulation d'un maximum d'informations possibles relatives aux données lors de leur collecte. La section 6.1 de la partie implémentation est dédiée à ce point.

4.2.2 Stratégies, tactiques et sélection d'optimisation

La fenêtre de travail étant constituée, se pose maintenant la question du travail d'optimisation proprement dit. Considérons l'état des unités de multiplexage physique à un instant donné. Si au moins une unité de multiplexage est disponible à cet instant, il faut lui donner du travail en appliquant une *fonction d'optimisation* chargée de sélectionner (ou de générer, par exemple, par une agrégation) le prochain paquet à soumettre à chacune des unités inactives en considérant les paquets de la fenêtre de travail. En entrée de la fonction d'optimisation, nous avons une grande variété d'arguments potentiels, qui peuvent être le nombre de paquets dans la fenêtre, leurs caractéristiques respectives (destination, flux, longueur, numéro de séquence, attributs de dépendance), les caractéristiques nominales et fonctionnelles du réseau, éventuellement des indications de l'application sur la politique d'ordonnancement des paquets, ainsi

que le nombre d'unités de multiplexage physiques disponibles ou encore leur état d'activité à l'instant considéré. Et cette liste n'est pas exhaustive.

Devant un tel nombre de paramètres, plusieurs stratégies d'optimisation peuvent être payantes. Plutôt que d'imposer une improbable fonction d'optimisation construite sur une stratégie figée, nous proposons au contraire que la fonction d'optimisation soit sélectionnable parmi un ensemble de stratégies extensible et programmable : chaque *stratégie* combine l'application d'un certain nombre de *tactiques* d'optimisation. Chaque tactique est elle-même construite à partir d'opérations élémentaires de manipulation de paquets du panel d'opérations usuelles, afin de s'adapter aux besoins des différentes applications.

4.2.3 Prédiction du comportement des cartes réseau

Prédire le comportement des cartes réseau ouvre la voie à plus d'efficacité dans le traitement des communications. En effet, le fait de savoir combien de temps va coûter un envoi ou quel débit devrait être atteint offre un élément de comparaison pour aider à l'élection d'un ou plusieurs réseaux à utiliser parmi ceux disponibles. Cela permet également de déterminer quand une carte réseau est susceptible de se libérer, et donc de décider s'il vaut mieux attendre un réseau prochainement disponible plutôt que d'employer ceux couramment libres. On peut également se servir de telles informations pour comparer différentes méthodes de transfert : cela peut permettre de choisir entre un envoi contigu et celui d'un ensemble de fragments épars en mémoire (*e.g.* avec un *iovec* ou par envois successifs). Cela permet également la détermination des différents seuils de transfert comme celui du rendez-vous afin d'employer efficacement les réseaux.

Finalement, ce type d'indications aide considérablement la prise de décision d'une stratégie mais également le fonctionnement de base de la bibliothèque de communication. Leur précision permet également d'affiner leur impact, il est donc préférable d'avoir à disposition des informations cohérentes avec la plate-forme matérielle employée grâce à un échantillonnage réalisé à l'initialisation de l'application. Nous ne nous contentons plus des quelques paramètres nominaux fournis par le constructeur de cartes réseau.

4.3 Ordonnancer de multiples flots de communication sur de nombreux cœurs

L'accroissement du nombre d'unités de calcul dans les machines a renforcé l'utilisation du parallélisme (de *threads*, plus précisément) au niveau applicatif. Il est donc devenu indispensable d'apporter un support adapté à ce type de modèle de programmation. Nous présentons ici comment nous envisageons de gérer ces multiples flots concurrents dans un environnement architectural multicœur.

4.3.1 Gérer de multiples flots concurrents

Un des pré-requis d'une bibliothèque de communication moderne est le support des flots de communication concurrents. Jusqu'à présent, l'une des préoccupations de notre bibliothèque de communication consiste à décharger l'utilisateur des aspects techniques liés au matériel réseau. Dans la même optique, il est donc important de faciliter la programmation parallèle d'une application avec des *threads* en offrant le plus de flexibilité possible avec un support équivalent au niveau MPI_THREAD_MULTIPLE de

la spécification de MPI (voir la section 3.1.2). Ce support doit non seulement être capable de gérer un nombre de *threads* applicatifs important, mais sans pour autant imposer de sections critiques importantes qui grèveraient les performances : il s'agit de soigner tout particulièrement le verrouillage des variables partagées, l'accès aux interfaces bas-niveau trop peu souvent *thread safe*, etc.

Par ailleurs, la Section 3.2.2 évoquait déjà l'importance de détecter correctement les événements d'entrées-/sorties du réseau, afin de réduire les coûts de communication. En effet, plus vite un transfert va être achevé, plus vite la carte sera disponible pour traiter le suivant, que l'on soit dans le contexte d'émission ou de réception, de transferts directs ou bien avec prise de rendez-vous préalable où il faut répondre le plus rapidement possible pour débloquer la suite des évènements. Tout cela a pour but de faire progresser l'ensemble de l'application s'exécutant dans les meilleures conditions possibles.

En concordance avec le découplage de l'application, il est important de ne plus faire appel à des primitives de détection de terminaison de communication dès que l'application le suggère. Nous avons déjà bien insisté sur le fait qu'inonder la carte réseau de différentes requêtes ne peut que détériorer ses temps de traitement. Pour cela, les requêtes faites par l'application ne doivent plus se traduire par une requête sur le réseau mais se limiter à aller vérifier le statut d'une variable mise à jour en temps voulu au sein de la bibliothèque de niveau intermédiaire, le travail de progression étant fait indépendamment de ces appels, par un *thread* dédié à cette tâche par exemple. Ce dernier – ou du moins celui qui est en charge de réellement interroger le réseau – doit être invoqué à des fréquences assez délicates à déterminer : s'il est appelé trop peu, la réactivité sera dégradée tandis que trop souvent, ce sont les performances de l'application qui pourraient l'être.

4.3.2 Tirer parti des architectures multicœurs

Au delà de la mécanique nécessaire pour protéger l'accès aux sections critiques de la bibliothèque de communication, la multiplication du nombre des fils d'exécution au sein d'une même application rend leur ordonnancement plus délicat. En effet, comme nous l'avons exposé dans la Section 2.2, la complexification des machines a conduit à la hiérarchisation des unités de calcul et de la mémoire qui leur est reliée. Ainsi, l'ordonnancement de *threads* peut être à présent réalisé en prenant en compte le placement en mémoire de leurs données respectives ou encore leur affinité avec les ressources réseau. Ce type de problématique a donné lieu à de nombreux travaux tels que ceux réalisés sur BUBBLESCHED durant la thèse de Samuel Thibault [Thi07].

Le combiné de l'ordonnancement des *threads* avec la prise en compte de critères d'affinités peut amener à des situations très différentes. Par exemple, si l'on privilégie l'occupation de tous les processeurs de la machine, l'ordonnancement de *threads* sur les processeurs va conditionner le placement des données, les données subissant alors les mêmes migrations que leur *thread* de rattachement. Si au contraire, la localisation des données ou encore l'utilisation des caches est privilégiée, le placement courant des données va intervenir dans la décision de l'ordonnancement. Il peut encore être décidé de migrer les données pour pouvoir mieux répartir la charge de calcul, etc.

Au même titre qu'un sous-effectif de *threads* par rapport au nombre de processeurs, ces différentes stratégies d'ordonnancement peuvent donc, au vu de leurs choix, introduire des moments d'inoccupation de certains processeurs. Et cela va d'autant plus s'amplifier au regard des prochaines générations de machines qui vont être équipées de dizaines, voire de centaines, de cœurs. Ainsi, en la concevant correctement, la bibliothèque de communication peut s'atteler à tirer parti de ces *trous* dans l'activité des processeurs afin d'y déporter des opérations qui déchargeraient le cœur sur lequel s'exécute l'ap-

plication. On peut imaginer déporter des opérations de progression de communication afin de les faire avancer en tâche de fond, y opérer les copies nécessitant des transferts en mode PIO (consommateur de ressource processeur) ou encore utiliser un cœur pour appliquer une stratégie d'optimisation sur les paquets en attente d'émission, etc.

Le chapitre suivant présente la réalisation de l'ensemble de ces concepts dans une bibliothèque de communication que nous avons nommée NEWMADELEINE en succession à MADELEINE 3. Il est cependant utile de préciser que NEWMADELEINE n'est pas une simple mise à jour de MADELEINE 3 mais une bibliothèque de communication totalement innovante.

Chapitre 5

NewMadeleine : un moteur de communication pour les réseaux hautes performance

Sommaire

Nous présentons ici NEWMADELEINE, la concrétisation des concepts que nous avons exposés dans le chapitre précédent. Nous allons voir comment toutes les propriétés mentionnées au chapitre précédent peuvent s'articuler afin de former un moteur d'optimisation performant. Nous verrons que cesser de traduire un appel à une interface de communication par une opération au niveau du réseau lui-même permet non seulement d'éviter sa saturation, mais surtout de prendre du recul sur la masse des données en transit. Cette opportunité est alors exploitée afin de les optimiser. Ce chapitre est consacré à la présentation de l'architecture en couche pour laquelle nous avons opté. Nous commençons par en donner une vue d'ensemble puis nous détaillons chaque niveau et les interactions qui les lient.

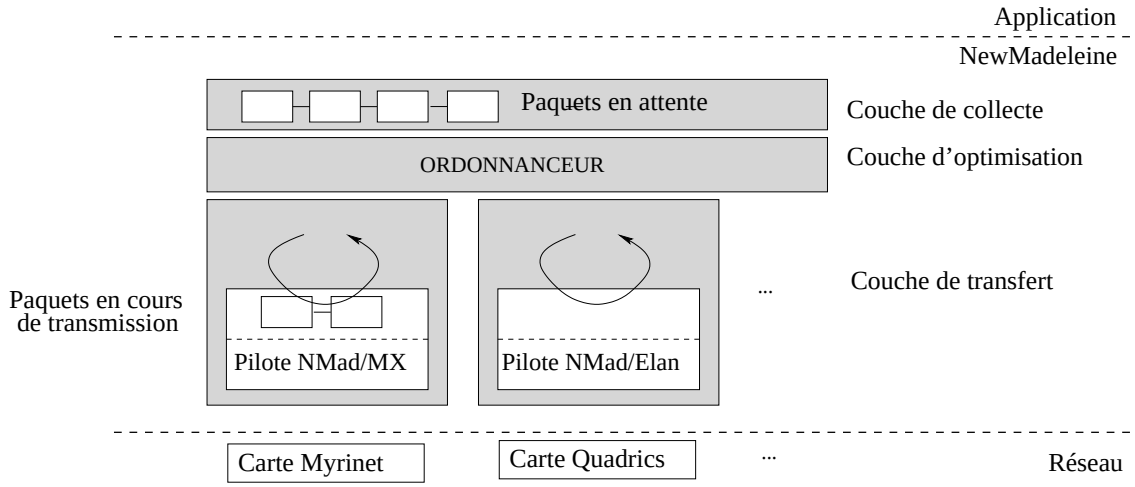


FIG. 5.1: *Architecture de NewMadeleine*

5.1 Vue d'ensemble de l'architecture de NEWMADELEINE

D'après les grandes lignes présentées dans le chapitre précédent, la partie s'occupant du transfert des données à proprement dit apparaît de manière bien dissociée de celle appliquant les optimisations, la fenêtre de travail est quant à elle complètement indépendante des deux autres. C'est donc tout naturellement que cette architecture s'est modélisée autour de trois couches interagissantes présentées à la Figure 5.1. La partie haute se charge de la collecte des messages, la partie intermédiaire correspond à la partie d'optimisation de ces derniers et enfin, la couche chargée des transferts, seule partie interagissant avec les réseaux.

La couche de collecte est la partie haute de cette architecture, c'est donc avec elle que l'application dialogue. Son action consiste à recenser les données déposées par les différents flux de communication qui feront l'objet d'optimisation par la suite quand une carte réseau viendra à se libérer.

La couche d'ordonnancement-optimisation est chargée de manipuler les messages en attente dans la fenêtre de travail pour former un nouveau paquet à fournir à la couche de transfert. Ce dernier doit consister en un assemblage à la fois conforme aux contraintes applicatives – cela ne doit pas introduire d'interblocage entre les communications – et avantageux du point de vue des performances à la couche de transfert en accord avec les critères à privilégier de l'application (*e.g.* la latence, la bande passante, etc.)

La couche de transfert est la seule couche qui interagisse avec les interfaces de communication bas niveau associées aux réseaux disponibles. Elle a pour rôle est de surveiller l'état d'avancement des cartes. Lorsque l'une d'elles se libère, elle dialogue alors avec la couche d'ordonnancement pour obtenir un nouveau paquet optimisé et le soumettre au réseau.

Nous allons maintenant analyser dans quelle mesure cette structure en trois couches permet de mettre en œuvre quasiment tous les principes énoncés au chapitre précédent. Dans la suite de ce chapitre, nous commençons à entrer dans les détails de réalisation de chaque couche. Ceci mettra en valeur ce qui a pu être fait suivant le modèle idéal et les concessions faites afin de préserver les performances ou éviter des modifications à des niveaux tel le système d'exploitation qui poseraient des problèmes de portabilité.

5.2 La couche de collecte

La couche de collecte joue véritablement un rôle de tampon entre l'application et le réseau. En effet, en recensant les messages laissés par l'application au sein de la fenêtre de travail, elle permet non seulement à l'application de retourner à son exécution sans la retarder mais aussi au réseau de s'octroyer le temps nécessaire au traitement de transactions en cours afin de ne pas être submergé. Afin d'ouvrir le spectre des optimisations possibles à l'ensemble des communications provenant de tous les flots de communication de l'application, les messages sont acheminés vers les mêmes structures de stockage. Cependant, comme les optimisations ont pour vocation de construire un nouveau paquet à émettre, ce dernier ne peut être constitué que de données ayant la même destination et les messages sont par conséquent triés suivant leur destination. Les différentes requêtes ainsi accumulées et donc mélangées – on ne respecte plus leur ordre de dépôt au sein d'un même flux – doivent néanmoins rester identifiables. Un minimum d'informations relatives (*e.g.* numéro de canal, identifiant de l'expéditeur, numéro de séquence, etc.) doivent être sauvegardées pour permettre de signaler la terminaison d'une requête à son commanditaire. Une partie est également jointe aux données sous forme d'entête afin que le récepteur soit capable d'identifier ce qu'il reçoit.

NEWMADELEINE propose trois interfaces utilisateur différentes aux applications : une interface par envoi de message, une interface par construction incrémentale de messages et un sous-ensemble de l'interface proposée dans le standard MPI-1. Les spécifications de chacune d'elles pourront être consultées en Annexe.
L'interface par passage de message (voir Annexe A.1) est pour le moins classique. Il s'agit d'échanger des messages par émissions et réceptions demandées de manière explicite par l'application et de manière symétrique, dans le sens où une émission est toujours coordonnée à une réception même si son modèle en mémoire est différent de celui décrit à l'émission.

L'interface par construction incrémentale de message (voir Annexe A.2) est l'interface historique de MADELEINE, originellement inspirée de l'interface de PVM [pvm]. Conceptuellement, cette interface qui fonctionne par passage de message est à mi-chemin entre une interface de type émission/réception symétriques comme la précédente et un modèle de flux de données tel TCP. Chaque message est délimité par des marqueurs de début et de fin qui englobent autant de bloc de données que l'application le souhaite. Dans l'idée, ceci est très similaire à l'envoi d'un tableau d'*iovec* : dans TCP, il est possible d'englober grâce aux primitives `writev` et `readv` plusieurs blocs de données non contigus en mémoire dans une seule opération de communication – ce qui signifie qu'un seul numéro de séquence est consommé mais pas qu'il y aura qu'une opération sur le réseau ou que le transfert de la mémoire à la carte se fait en une seule fois –. Cependant, le fait de soumettre les blocs de données au fur et à mesure de la construction du message à la bibliothèque au lieu que cela ne soit fait qu'en un seul appel permet d'anticiper les communications nécessaires. Par exemple, les demandes de rendez-vous nécessaires pourront être anticipées, etc.

Mad-MPI (voir Annexe A.3) est une implémentation d'un sous-ensemble de la spécification proposée de MPI-1 qui nous sert de *preuve de concept*. Le but de cette interface n'est pas de remplacer une implémentation complète mais plutôt de servir de point de référence pour les tests et applications que nous présentons dans la partie évaluation de ce document. Elle regroupe un certain nombre de primitives de communication point-à-point d'envoi (*e.g.* MPI_Send, MPI_Isend, MPI_Rsend, MPI_Ssend) et de réception (*e.g.* MPI_Recv, MPI_Irecv), accompagnées des primitives de détection de terminaison associées (*e.g.* MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Test, MPI_Testany). Des opérations de définition et d'échanges de types dérivés complexes sont également proposées. Sont ainsi acceptés les

types dérivés contigus, décrits par un tableau dont l'espacement entre les éléments et la taille de ces derniers sont homogènes ou/et hétérogènes. Les opérations collectives les plus courantes de synchronisation et d'échanges sont également implémentées.

5.3 La couche de transfert

La couche de transfert est la couche basse de NEWMADELEINE. Elle peut être assimilée à une interface de communication à elle seule puisqu'elle abstrait l'interface bas niveau du réseau sous-jacent aux couches supérieures de NEWMADELEINE.

Son rôle consiste à mimer les cartes réseau à la façon d'un ordonnanceur de processus qui, lorsqu'il est invoqué par un processeur, déroule un algorithme permettant de choisir un nouveau processus prêt. Ainsi lors de son tour de surveillance de l'état d'avancement des cartes, si l'une d'entre elles est considérée comme libre (*i.e.* lorsque son quota de communication à traiter simultanément n'est pas atteint), elle doit dialoguer avec la partie supérieure pour obtenir de nouveaux paquets optimisés à soumettre à la carte réseau.

En y regardant de plus près, cette couche se resubdivise en une partie spécifique à chaque technologie réseau supportée et une partie générique qui procède à la boucle de surveillance des cartes elles-mêmes.

5.3.1 Les pilotes réseau de NEWMADELEINE

La partie spécifique à chaque réseau – qu'on appelle communément *pilote* ou *driver* – encapsule l'ensemble de ses caractéristiques et spécificités. Afin de ne pas dupliquer de développement de portions de code à chaque nouveau pilote, l'interface qui les abstrait a été conçue de manière à être la plus minimale possible. Finalement, elle se résume à des primitives d'ouverture/fermeture de connexion, d'envoi/réception de messages et de détection de fin de transfert. À cela s'ajoute un ensemble d'informations sur les capacités du réseau ayant pour but de permettre son exploitation fine. L'ensemble des informations récoltées peut-être varié. Des indications peuvent être recensées sur la nature des échanges (en flux, par message, etc.), sur la capacité du réseau à traiter des blocs de données épars en mémoire, ou encore si ce dernier est capable de procéder à des réceptions sélectives, etc. Elles peuvent aussi être plus en rapport avec les caractéristiques de la machine employée, ce qui implique de devoir les déterminer expérimentalement. Par exemple, les performances en latence et débit mentionnées par les constructeurs sont généralement celles qui sont obtenues dans le meilleur contexte possible. Afin d'en obtenir de plus réalistes, les performances réelles de la machine utilisée doivent être recueillies. Le nombre de requêtes devant être soumises à la carte peut aussi être renseigné. Comme évoqué dans la Section 4.1.1, celui-ci permet d'assurer un remplissage constant du tampon de transmission de la carte réseau et s'avère de ce fait être un paramètre important dans l'utilisation optimale d'un réseau. Plus de détails sur la collecte et l'emploi de ces types de capacités sont donnés dans la Section 5.4.1.

Cependant, tout en devant rester concise, cette interface est appelée à être modifiée. En effet, sa forme actuelle est construite à partir des fonctionnalités que nous avons décidé d'employer. En l'état, elle ne permet notamment pas de procéder à des communications par accès direct à la mémoire (*i.e.* des communications de type RDMA) alors qu'INFINIBAND, le réseau ayant les meilleures performances du marché actuel, fonctionne suivant ce paradigme. Par ailleurs, il existe certains mécanismes développés dans certains pilotes de communication bas niveau que nous n'exploitons pas. Par exemple, nous ne

court-circuitons pas la prise de rendez-vous de NEWMADELEINE même si elle est déjà faite au niveau inférieur, la capacité du filtrage des communications n'est pas employé à l'amélioration du traitement des communications de source anonyme, etc.

5.3.2 La boucle de progression

La partie générique s'appuie sur les pilotes de communication que nous venons de présenter afin de déterminer l'état des cartes réseau sous-jacentes. Dans ce document, son comportement a jusqu'à présent été décrit comme celui d'une boucle de surveillance. Cette manière de faire est certainement facile à appréhender dans le sens où le traitement est séquentialisé : un pilote est interrogé sur l'état du réseau, s'il s'avère libre, un nouveau paquet est demandé et soumis au réseau puis on passe au suivant. Cela ne se conforme néanmoins pas aux besoins en réactivité évoqués dans le chapitre précédent. L'activité entière de NEWMADELEINE étant pilotée par l'activité de cartes réseau de la machine sur laquelle elle s'exécute, il est primordial de les surveiller le plus étroitement possible. Pour cela, synthétiquement, on admet que la surveillance de l'état d'avancement des communications est confiée à des *threads* créés spécifiquement à cette tâche qui se réveillent dès que la carte réseau à laquelle ils sont associés devient inactive pour invoquer l'optimiseur et obtenir un nouveau paquet à soumettre. Chaque carte a dorénavant une activité totalement indépendante des autres. Ceci décrit le schéma de comportement de ce qu'on souhaite obtenir, mais n'est pas exactement ce qui a été réalisé en réalité. La véritable implémentation, autrement plus fine et efficace, est faite au sein d'un gestionnaire d'entrées/sorties nommé PIOMAN que nous présentons dans la Section 6.3. Dans les grandes lignes, son travail en étroite collaboration avec un ordonnanceur de *threads* lui permet d'être ordonnancé à des moments clés de l'exécution de l'application. La détection d'événements sur le réseau est donc faite à des fréquences mieux adaptées. De plus, il a l'opportunité d'obtenir des informations concernant l'occupation des processeurs. Il profite donc de cela pour déporter certaines opérations – comme des copies, une attente active sur la fin d'une communication ou encore le traitement associé à la terminaison d'un transfert – sur ces ressources inutilisées.

De plus, une attention particulière a été donnée au placement de tels *threads* de progression au plus près de la carte réseau de rattachement dans les machines NUMA (Section 2.2). En effet, le placement des *threads* communicants et celui des données ont un impact considérable sur les performances, notamment en débit [MG07]. Sur de telles architectures, il est inévitable que les contrôleurs d'entrées/sorties (*i.e.* les cartes réseau, pour faire simple) aient une localisation plus proche de certains processeurs et bancs mémoire que d'autres. Ainsi, cette hétérogénéité dans les distances d'accès entraîne des variations de temps dans les accès. Par extension aux effets NUMA, ces variations sont nommées effets NUIOA pour *Non Uniform Input/Ouput Access*. De ce fait, à partir de la topologie de la machine, NEWMADELEINE initialise ses pilotes de communication réseau aux bons endroits. Ceci entraîne alors un placement définitif de l'ensemble des ressources spécifiques aux pilotes. Cependant, cela ne dispense pas l'application de devoir placer correctement ses données en fonction de ces placements pour être totalement efficace. Il faudra donc, par la suite, soit être capable d'exposer ces informations à l'application – ce qui implique une prise de connaissance de ce type de problème très technique au niveau de l'application et n'est pas souhaitable –, soit être capable de migrer les tâches applicatives près des cartes réseau lors de leur soumission. Ces travaux sont l'objet de la thèse de Stéphanie Moreaud.

Pour conclure, la couche de transfert est une couche très fine qui n'apporte aucune intelligence dans la constitution des paquets à soumettre au réseau puisqu'elle délègue entièrement cela à la couche d'optimisation. Cependant, elle a un potentiel important en matière d'exploitation des ressources disponibles

afin d'augmenter la réactivité, le recouvrement des communications et leur déplacement au sein même de la machine.

5.4 La couche d'ordonnancement et d'optimisation

Nous en arrivons finalement au cœur même de NEWMADELEINE avec la couche d'optimisation. Son rôle est pour le moins central : les paquets déposés par l'application sont tirés par les cartes réseau en temps voulu par son intermédiaire. C'est elle qui orchestre la modélisation et la distribution de toutes les communications sur les cartes réseau disponibles.

Comme nous l'avons signalé dans la Section 5.3.2, chaque pilote réseau renseigne le nombre de requêtes en émission et en réception qu'il peut traiter à un instant donné. Tant que ces quotas ne sont pas atteints, la couche d'ordonnancement va s'activer à soumettre des requêtes à la couche de transfert sans que cette dernière ne les sollicite. Ainsi, lorsque l'application dépose des données à émettre, la stratégie les transmet directement à la couche de transfert afin d'activer (ou compléter) le pipeline de transmission de la carte réseau sélectionnée. La stratégie peut néanmoins ne pas suivre ce mode opératoire si son mode de fonctionnement préconise l'attente de plus de données avant de libérer un paquet. De même, si c'est une réception qui est déposée, un tampon préposé à recevoir les données ou une demande de rendez-vous le cas échéant, est soumis à la couche inférieure. Une fois suffisamment remplies, il ne s'agit plus que de réalimenter les cartes sur leur demande exclusivement. Ainsi, si la couche de transfert détecte la fin d'une réception, la couche d'optimisation est invoquée afin de la remplacer par un nouveau tampon de réception qui, selon le cas, sera soit une zone de réception intermédiaire (pour les messages ne nécessitant pas de rendez-vous et tous les messages de contrôle), soit une zone de réception finale fixée au préalable par un acquittement. Sur le même modèle, toutes les émissions achevées sont remplacées par la couche d'optimisation sur invocation de la couche de transfert. La stratégie alors sélectionnée à la configuration de NEWMADELEINE est appliquée. Cette dernière considère l'ensemble des messages en attente d'émission, applique son algorithme et en donne le résultat à la couche inférieure. Afin de ne pas perdre de temps en calcul à ce moment crucial – puisque le nombre requis de requêtes correspond à celui nécessaire au bon remplissage du pipeline de la carte réseau –, la stratégie peut choisir de former un paquet à l'avance ou d'en construire un au fur et à mesure que les données sont déposées afin de n'avoir plus qu'à le finaliser, le cas échéant. Cependant, il est préférable de se limiter à un nombre restreint de tels paquets car cela laisse à l'ordonnanceur une marge de manœuvre plus importante qui ne peut être que bénéfique.

Afin de donner le plus d'informations possibles aux stratégies d'ordonnancement, un maximum d'informations doivent leur parvenir. Comme nous en avons déjà discuté précédemment, un certain nombre de paramètres pouvant aider à la prise de décision sont très dépendants de la machine sur laquelle l'application doit s'exécuter. Ainsi, afin d'offrir le maximum de précision aux stratégies, et donc obtenir le meilleur ordonnancement possible, nous avons décidé d'échantillonner la plate-forme d'exécution. La suite de la section expose comment sont réalisées et utilisées les prises de mesures avec quelques exemples de stratégies d'ores et déjà implémentées.

5.4.1 Échantillonnage de la plate-forme d'exécution

L'objectif de l'échantillonnage est de fournir les informations les plus précises possibles à l'ordonnanceur de manière à ce que ce dernier puisse prendre les meilleures décisions à un moment donné. En

complément des caractéristiques pouvant être renseignées à l'implémentation du pilote spécifique à une technologie réseau dans NEWMADELEINE, les performances réelles de la machine sur laquelle l'application va s'exécuter sont donc recensées.

Afin de ne pas pénaliser l'application pendant son exécution, l'ensemble de ces mesures est fait à l'initialisation de la plate-forme de communication. Il va principalement s'agir de tests assez classiques de type ping-pong sous différentes formes. En l'occurrence, sont effectuées des prises de performance brutes sur l'échange de données contiguës afin de pouvoir estimer le temps qu'une communication va prendre ; de même que non contiguës lorsque le pilote bas-niveau est en capacité de le faire. Ceci permettra de déterminer s'il est préférable d'utiliser ce dernier système plutôt que de copier les données en contigu avant de les transférer.

Les résultats obtenus sont stockés dans les tableaux pour des masses de données alignées sur des puissances de deux. Cela à l'avantage de permettre une évaluation rapide d'un temps de transfert ou d'un débit à partir d'une longueur de données. Il est en effet simple de déterminer l'intervalle dans lequel la taille est comprise puis d'y appliquer une interpolation linéaire afin d'obtenir une prédiction. En plus du temps que va prendre une communication, cette dernière permet de déduire également la date à laquelle va se libérer une carte réseau. Ainsi, une stratégie d'ordonnancement peut choisir d'attendre une carte réseau dont les caractéristiques répondent mieux à ses exigences plutôt que d'en utiliser une disponible dans l'immédiat.

Par ailleurs, l'échantillonnage permet de déterminer plus finement les seuils de changement de méthodes de transfert. Par exemple, INFINIBAND est muni de trois méthodes de transfert différentes pour ses transferts en RDMA. La première transfère les données dans un tampon intermédiaire pré-alloué par le nœud courant avant de les écrire dans un tampon également pré-alloué mais cette fois-ci par le récepteur. La seconde, enregistre à la volée les zones de mémoire que ce soit à l'émission ou à la réception par l'intermédiaire d'un rendez-vous. À la suite de cela, le récepteur envoie les informations sur la zone nouvellement enregistrée afin que l'émetteur puisse y placer les données. La dernière méthode suit le même procédé que la précédente avec en plus un enregistrement pipeliné des blocs de la zone à enregistrer. Cela permet de commencer la transmission effective des données avant que l'ensemble de la zone mémoire ne soit enregistré. Ainsi, si on utilise chaque méthode de transfert sur un large ensemble de tailles de données, il est aisé de définir quels sont les points d'intersection des courbes de performances obtenues, qui correspondent aux seuils de changement de méthodes de transfert.

À la suite des mesures associées, un classement des réseaux est également établi suivant des critères variés : un des plus rapides, un autre des plus performant en débit, etc. Cela à l'avantage de permettre l'accès rapide à ce type d'information et donc de prendre de meilleures décisions en un temps réduit.

Finalement, l'ensemble des informations collectées a pour objectif d'aider l'ordonnanceur à appliquer la meilleure stratégie d'ordonnancement à partir de paramètres plus fins que ceux proposés en standard par le constructeur du matériel.

5.4.2 Quelques d'exemples de stratégies d'optimisation

Selon les schémas de communication suivis par les applications (*e.g.* utilisation de RPC, d'une DSM, BLAS, etc.), les besoins d'ordonnancement vont différer : l'utilisation maximale de la bande passante va primer pour une application transférant d'importantes masses de données tandis que pour d'autres, la latence sera le paramètre primordial. Il est donc inévitable de devoir développer différentes stratégies d'optimisation en accord avec les besoins de l'application. Nous présentons ici plusieurs stratégies qui

peuvent au choix être utilisées telles quelles, être modifiées de manière mineure, ou servir de modèle à l'implémentation de nouvelles.

Cette section présente quatre stratégies d'ordonnancement implémentées dans NEWMADELEINE. Elles produisent à elles seules des schémas de communication assez variés : certaines réagencent les messages tandis que d'autres les agrègent ou les subdivisent. La dernière est la stratégie que l'on aurait voulu idéalement implémenter, nous expliquons pourquoi ce n'est pas le cas.

5.4.2.1 Strat_default, la stratégie de base

Cette stratégie est la plus basique que l'on puisse réaliser (voir Figure 5.2). En effet, elle se contente de faire suivre les requêtes de communication telles qu'elles lui ont été données. Elle ne permet donc pas d'avoir de schémas de communication plus évolués que ceux décrits par une bibliothèque au comportement plus classique, mais préserve de toute contention au niveau des cartes réseaux. Cette stratégie est celle qui nous permet dans la partie évaluation de mesurer le surcoût introduit par l'ensemble des mécanismes de NEWMADELEINE.

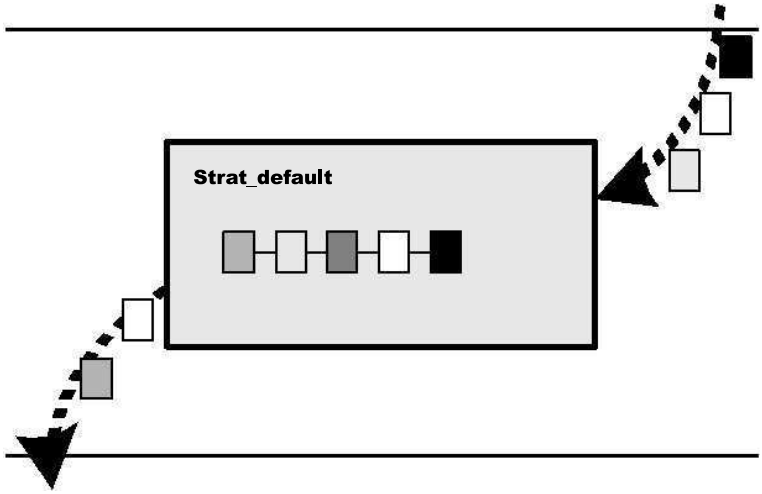


FIG. 5.2: *Strat_default : les requêtes d'émission sont transmises telles qu'elles sont soumises.*

5.4.2.2 Strat_aggreg, agrégation de messages ayant la même destination

La stratégie d'agrégation (Figure 5.3) est déjà plus évoluée que celle par défaut. Elle fait partie des stratégies que nous qualifions d'agressives, dans le sens où les paquets qui seront transmis à la couche de transfert pour leur émission y sont construits à la volée. Lorsque l'application dépose des données à émettre, la stratégie commence par considérer leur taille : si une prise de rendez-vous préalable est nécessaire, la stratégie cherche à agréger le message de contrôle adéquat à un paquet déjà en construction et sinon les données elles-mêmes. Ceci ne modifie pas la façon dont on aurait agrégé les données si nous avions attendu l'invocation de l'ordonnanceur par la couche de transfert mais permet de perdre moins de temps à ce moment-là. En effet, il n'y a alors plus qu'à transmettre le paquet déjà pré-construit pour l'alimenter. Toutes les applications échangeant fréquemment des messages de contrôle ou du moins de petite taille vont profiter d'un tel schéma de communication. Nous pensons en particulier à celles utilisant des RPC, des BLAS ou encore une DSM.

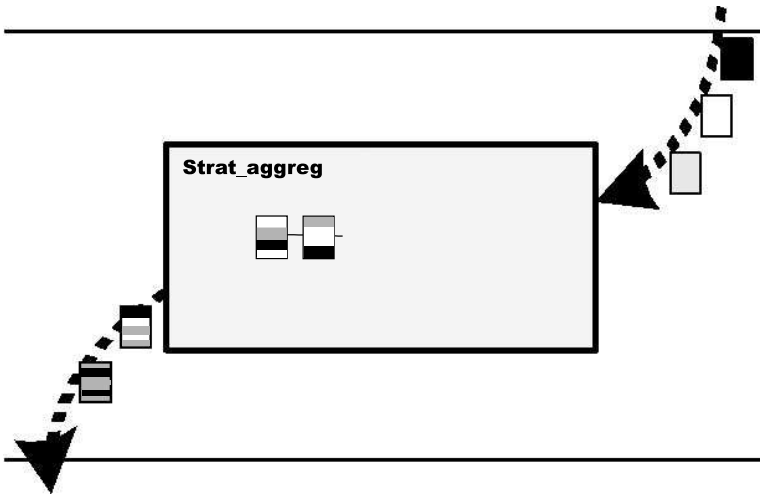


FIG. 5.3: *Strat_aggreg : les messages envoyés par copie et les messages de contrôle sont agrégés.*

5.4.2.3 Strat_multirail, distribution des messages sur plusieurs cartes réseau

Comme nous le verrons dans le chapitre 7 dédié aux évaluations, la stratégie visant à exploiter les plate-formes équipées de plusieurs cartes réseau de technologies homogènes et même hétérogènes a été construite de manière incrémentale. En est sorti que la stratégie d'agrégation reste la plus adéquate pour l'envoi des messages ne nécessitant pas de prise de rendez-vous en comparaison à une stratégie gloutonne qui distribuerait les messages sur l'ensemble des réseaux disponibles. En effet, l'envoi de tels messages est toujours assorti d'une copie bloquante de données depuis la mémoire principale vers celle des cartes réseau. Les envois de fragments obtenus si les messages étaient coupés ne peuvent donc pas être réalisés en parallèle dans le cas où un seul processeur est utilisé. Il n'y a donc pas d'utilité à répartir l'envoi d'un même message sur plusieurs réseaux. Ainsi, les messages courts et messages de contrôle vont être agrégés et expédiés sur le réseau le plus performant en terme de latence.

La spécificité de cette stratégie réside donc dans le traitement des transmissions des messages longs. Tout d'abord, cette stratégie est guidée par le côté récepteur de l'échange. Ainsi, comme le montre la Figure 5.4, sur réception d'une demande de rendez-vous, l'émetteur va consulter l'état de son pool de carte réseau et déterminer celles qui sont susceptibles de participer à l'échange. Ensuite, à partir de l'échantillonnage fait de la machine, la proportion de données à envoyer sur chaque technologie est déterminée de façon à obtenir des temps de transfert égaux et transmis à l'émetteur. Ce dernier n'a plus qu'à exécuter ce que le récepteur a décidé, les tampons de réception étant à ce stade postés dans cette configuration.

5.4.2.4 Stratos, support de la qualité de service

Le support de qualité de service nous a amené à développer une stratégie d'ordonnancement plus conséquente en terme de développement que les précédentes. En effet, par essence, offrir de la qualité de service implique d'offrir un support de communication aux flux de manière différenciée. Par exemple, certains messages seront envoyés sans délai par rapport à ceux déjà en attente car le service exigé pour leur envoi a une priorité absolue sur tous les autres. Ainsi, différents besoins doivent pouvoir

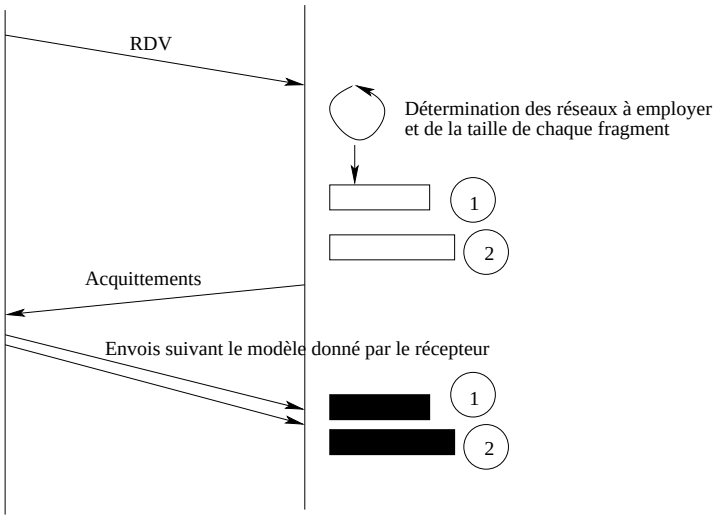


FIG. 5.4: *Strat_multirail : les messages nécessitant une demande de rendez-vous sont distribués sur plusieurs réseaux sur décision du récepteur.*

être exprimés par l'application au moment où cette dernière dépose de nouvelles requêtes de communication, notamment en terme de latence, de débit, de priorité, etc. L'objectif est de diriger ces flux vers les mécanismes qui fournissent une réponse à leurs besoins. L'interface exposée aux utilisateurs a donc été en premier lieu enrichie de façon à pouvoir exprimer le niveau de service à employer. Une première

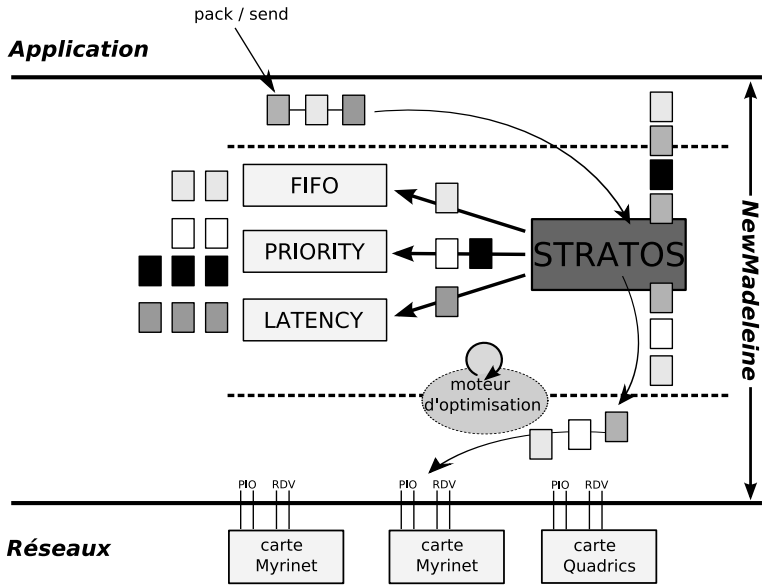
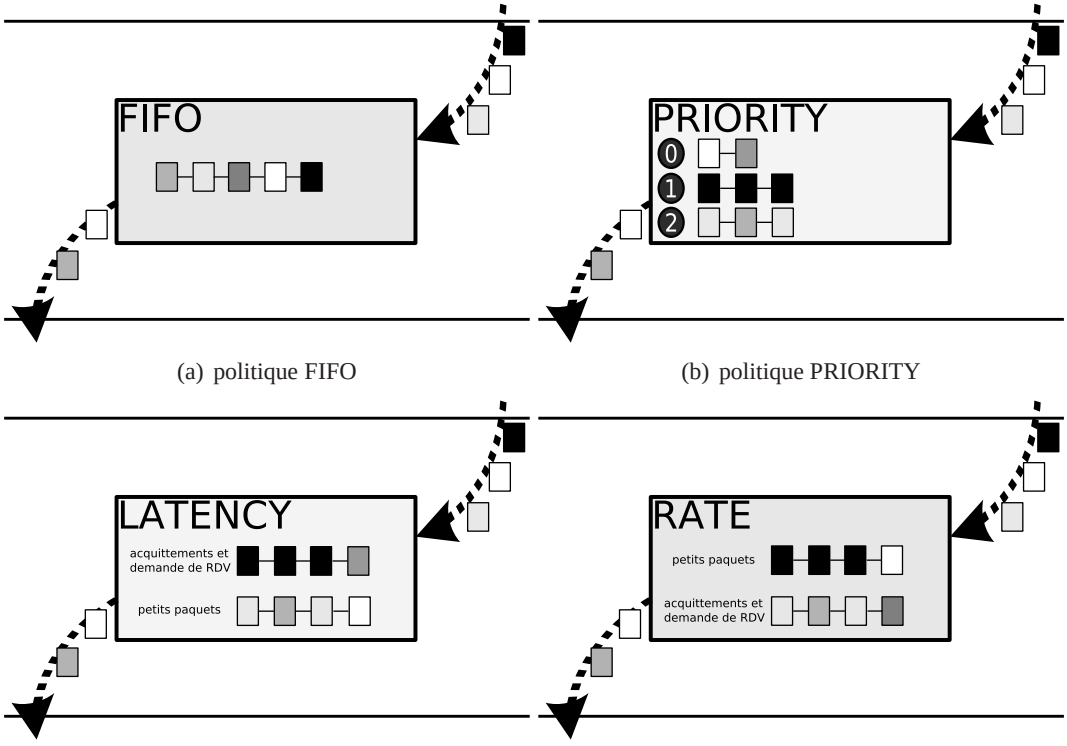


FIG. 5.5: *Stratos : support à la qualité de service.*

fonction associe un niveau de qualité de service – que l'on nomme ici politique d'ordonnancement – à un flux passé en paramètre. L'application n'a ensuite plus qu'à soumettre ses requêtes sur le bon flux pour que cette dernière soit traitée correctement. Le degré de priorité affecté aux paquets au sein même d'un flux peut également être exprimé. Actuellement, comme on peut le voir sur la Figure 5.5, stratos dispose de quatre politiques d'ordonnancement différentes que nous détaillons à partir de la Figure 5.6.



(a) politique FIFO (b) politique PRIORITY

(c) politique LATENCY (d) politique RATE

FIG. 5.6: *Politiques d'ordonnancement de la stratégie Stratos.*

FIFO Aucun traitement particulier n'est appliqué dans cette politique. Tous les paquets sont placés dans une même liste et le premier arrivé est le premier transmis. C'est le fonctionnement de base de la stratégie par défaut strat_default. Cette politique est utilisée pour tous les flux qui n'ont pas spécifié de politique particulière.

PRIORITY Dans cette politique, plusieurs files d'attente sont disponibles (dans les illustrations, trois), et correspondent chacune à un niveau de priorité. Les flux de cette politique sont alors placés dans la file de la priorité qui leur a été affectée. De ce fait, lorsque la fonction d'optimisation est appelée par la stratégie, les paquets de plus haute priorité sont traités en premier.

LATENCY Cette politique n'est pas directement issue des travaux sur la qualité de service puisque son objectif n'est pas de garantir une latence minimum, mais plutôt de privilégier les flux qui délivrent des paquets de petite taille. On retrouve une notion de priorité mais non plus entre chaque flux mais qui peut être au sein d'un même flux. En général, les applications délivrant les flux de ce type vont vouloir se transmettre des petites données en urgence.

RATE Il s'agit de la politique inverse à la précédente étant donné que ce sont les paquets nécessitant une demande de rendez-vous qui vont être privilégiés. En effet, ce type de paquet va occuper beaucoup plus de bande passante que ceux transportant moins de données. Pour élaborer cet algorithme, les de-

mandes de rendez-vous sont délivrées en priorité. Ainsi, le récepteur pourra émettre son acquittement plus rapidement que dans le cas où aucune distinction n'est faite.

Finalement, on arrive à appliquer différentes stratégies d'ordonnancement au sein d'une même stratégie à partir des indications fournies par l'application. Cependant, cette stratégie soulève de nombreuses questions. En effet, suivant la nature des politiques employées, certaines peuvent se révéler incompatibles avec d'autres. Par exemple, une politique peut vouloir utiliser un réseau ayant une bonne bande passante pour le transfert de messages longs alors qu'une autre les envoie sur plusieurs de technologies plus hétéroclites. Laquelle choisir ? sur quels critères ? La disponibilité des ressources réseaux peut également rentrer en considération dans la décision finale. La stratégie suivante est supposée démêler ce type d'hésitation.

5.4.2.5 La stratégie *ultime*

La stratégie *ultime* est dans l'idéal celle qui pourrait décider quelle stratégie est la meilleure à employer à un instant donné. Pour cela, elle est donc supposée pouvoir simuler l'application de chacune des optimisations et leur attribuer une note pour servir de point de comparaison avec les autres. Une fois la meilleure élue, cette dernière est appliquée et la meilleure combinaison de paquets pouvant être produite est soumise à la couche de transfert pour envoi immédiat.

Ce mode de fonctionnement est pour le moins délicat et difficile à mettre en place de manière à ce qu'il ait un temps d'exécution réduit. Tout d'abord, les stratégies *à la volée* ne peuvent plus être employées. En effet, tant que la stratégie devant être utilisée n'est pas déterminée, aucune autre ne peut être appliquée aux messages soumis par l'application. Toutes doivent donc être expérimentées soit au moment même où la couche de transfert invoque l'ordonnanceur ce qui peut s'avérer coûteux en temps, soit au moment de la soumission mais chacune de leur côté. Dans ce dernier cas, l'application de la stratégie pourrait être simulée au fur et à mesure que les messages sont déposés afin de n'avoir qu'à les comparer au moment où une carte réseau demande une nouvelle communication à traiter. Mais que faire si un nouveau paquet vient à remettre en cause tout ce qui a été construit auparavant ? Il faudrait alors être capable de revenir en arrière, mais jusqu'à quand ? Et comment stocker les différentes étapes nécessaires à l'utilisation réelle d'une optimisation finalement ? Chacune de ces opérations est réalisée sur le chemin critique de NEWMADELEINE et ne peuvent raisonnablement pas prendre un temps inconsidéré. Cet état de fait est accentué par le fait que plus le temps s'écoule et plus l'état de la machine change. D'autres communications s'achèvent, des prises de rendez-vous sont reçues, etc. qui remettent en question non seulement l'application de la stratégie courante mais l'ensemble de celles pratiquées précédemment.

Actuellement, NEWMADELEINE montre ses limites sur ce point. Il ne lui est pas possible d'assurer un tel mécanisme. Cependant, nous verrons par la suite (Section 6.3) que la collaboration de NEWMADELEINE avec le serveur d'évènement PIOMAN nous ouvre l'opportunité de déporter ce type d'opérations et pourrait finalement aboutir à une solution.

Chapitre 6

Élements d'implémentation

Sommaire

Ce chapitre a pour objectif de donner des points clés du fonctionnement interne de NEWMADELEINE. Dans un premier temps, nous allons mettre l'accent sur le caractère ludique de NEWMADELEINE de par sa fonction de plate-forme d'expérimentation de nouvelles stratégies d'optimisation de schémas de communication. Pour cela, nous commençons par décrire la structure qui encapsule les données dès leur dépôt dans NEWMADELEINE afin de pouvoir les manipuler correctement par la suite au sein d'une stratégie. Car, en effet, le but de la seconde partie de ce chapitre est en fin de compte de montrer au lecteur que l'implémentation d'un tel code est à la portée de tout développeur d'application qui connaît la forme de ses communications. La seconde moitié du chapitre aborde un aspect totalement différent. En effet, nous y présentons PIOMAN, un détecteur d'évènements réseau réactif qui met en œuvre l'ensemble des concepts que nous avons exposés précédemment à propos du support des accès concurrents et bien plus encore.

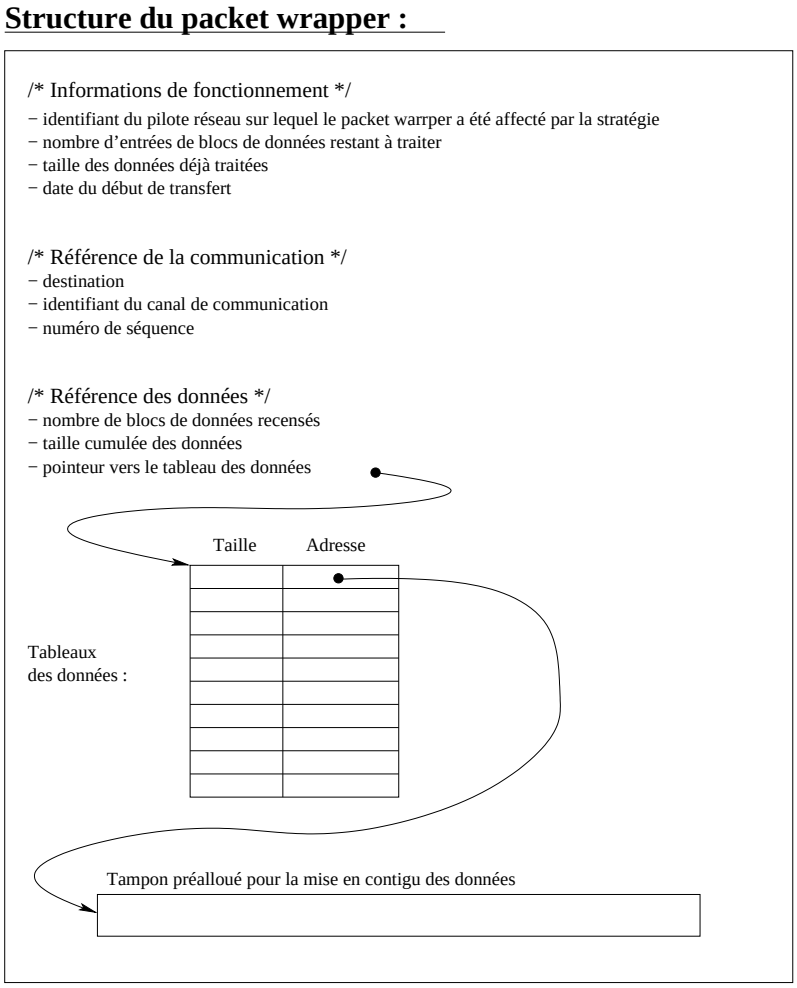


FIG. 6.1: *Encapsulation des données - Vue d'ensemble de la structure du* packet wrapper.

6.1 Le fil rouge de NEWMADELEINE : la structure d'encapsulation des données

Dès leur dépôt au sein de NEWMADELEINE, les données faisant l'objet d'une requête de communication sont étiquetées minutieusement dans une structure que nous avons nommée *packet wrapper*. Afin de pouvoir naviguer correctement entre chaque couche de la bibliothèque de communication, elles resteront ainsi encapsulées jusqu'à la terminaison de l'échange. Comme nous l'avons déjà évoqué à plusieurs reprises, le mode de recensement des messages déposés par l'application ne permet pas leur identification s'ils ne sont pas accompagnés des mentions relatives à leur transferts. En effet, les messages sont désormais collectés suivant la cible de leur transfert et non plus suivant leur canal de communication, couramment noté *tag* ou encore *channel*.

La Figure 6.1 montre l'ensemble des informations que fournit l'application au moment du dépôt recensé : la cible de l'échange, l'identifiant du canal de la communication et la description des données, c'est-à-dire l'ensemble des couples taille et adresse en mémoire. En plus d'aider au cheminement au travers des couches NEWMADELEINE, ces informations servent également à renseigner les différents

champs de l'entête qui est ajouté à tout envoi de données. Contrairement au cas courant où ce dernier est directement accolé aux données, il en est dissocié dans le cas des données nécessitant un rendez-vous préalable puisqu'il précède les données et est la prise de rendez-vous elle-même. Le rôle de ces entêtes est de permettre au récepteur de s'appuyer sur ces identifiants afin d'être en mesure de réassocier les données arrivées avec les requêtes de réception qui lui ont été soumises par l'application.

Par ailleurs, l'ajout de ces entêtes fait partie intégrante du protocole de communication que nous avons établi. Ils ne sont en aucun cas dépendant d'une stratégie d'ordonnancement pouvant être appliquée. Cette dernière peut néanmoins prétendre décrire la manière dont ils doivent être ajoutés aux données. Les deux morceaux (l'entête et les données) peuvent soit être transmis au travers d'un tableau de blocs de données non contigus, soit l'être sous une forme unifiée si l'entête et les données sont tour à tour copiés dans un même tampon intermédiaire. Le plus souvent, nous nous sommes aperçu que les données de taille ne nécessitant pas de rendez-vous préalable gagnent à être copiées en contigu de leur entête. Pour cela, dans un souci de performance, nous avons ajouté à la structure même du *packet wrapper* un tampon intermédiaire en charge de recueillir ces données.

Finalement, une des facettes de NEWMADELEINE étant le support à l'élaboration de nouvelles stratégies par des utilisateurs lambda, cette structure d'encapsulation a été conçue afin d'être la plus complète possible. La perte d'information ne peut que nuire à la prise de décision de l'optimiseur et à la perturbation des transferts eux-mêmes. Elle peut néanmoins encore évoluer avec l'ouverture à des développeurs extérieurs qui mettront sans aucune doute à jour de nouveaux besoins.

6.2 Élaboration d'une nouvelle stratégie

Comme nous l'avons déjà dit, nous avons porté une attention particulière à l'ajout de nouvelles stratégies de manière simple afin qu'un simple utilisateur de NEWMADELEINE soit en mesure d'en développer pour lui-même. Nous allons dans un premier temps détailler les fonctions à implémenter qui servent à faire descendre les requêtes de communication de la couche de collecte jusqu'à la stratégie, puis les fonctions qui permettent au socle de la couche d'ordonnancement d'interagir avec la stratégie afin d'obtenir des nouveaux paquets à soumettre au réseau ou comment traiter une demande de rendez-vous. Afin d'illustrer nos propos, les portions de code correspondant de la stratégie par défaut de NEWMADELEINE sont à chaque fois données. Pour rappel, cette dernière envoie les messages dans l'ordre de soumission de l'application. Les messages de taille supérieure à NM_SO_MAX_SMALL y sont envoyés après prise de rendez-vous. De plus les messages de taille inférieure à NM_SO_COPY_ON_SEND_THRESHOLD sont copiés dans un tampon intermédiaire afin d'être accolés à leur entête.

6.2.1 Interface pour la couche de collecte des messages : Primitives d'empaquetage

NEWMADELEINE s'appuie sur deux types de messages afin de mettre en œuvre son protocole de communication : des messages de données et des messages de contrôle. D'un côté, les fonctions `pack` et `packv` dont les prototypes respectifs sont les suivants :

```
int pack(void *_status,
         struct nm_gate *p_gate,
         uint8_t tag, uint8_t seq,
         void *data, uint32_t len);
```

```
1  static int strat_default_packv(void *_status, struct nm_gate *p_gate,
2                                 uint8_t_t tag, uint8_t seq,
3                                 struct iovec *iov, int nb_entries){
4    struct nm_so_strat_default *status = _status;
5    struct nm_so_pkt_wrap *p_so_pw = NULL;
6    int flags = NM_SO_DATA_DONT_USE_COPY;
7    int i, err = NM_ESUCCESS;
8
9    for(i = 0; i < nb_entries; i++){
10     if(iov[i].iov_len <= NM_SO_MAX_SMALL) { /* Small packet */
11       if(iov[i].iov_len <= NM_SO_COPY_ON_SEND_THRESHOLD)
12         flags = NM_SO_DATA_USE_COPY;
13
14       /* Simply form a new packet wrapper and add it to the out_list */
15       err = nm_so_pw_alloc_and_fill_with_data(tag, seq,
16                                               iov[i].iov_base, iov[i].iov_len,
17                                               flags, &p_so_pw);
18       if(err != NM_ESUCCESS) goto out;
19
20       list_add_tail(&p_so_pw->link, &status->out_list);
21     } else { /* Large packets : we have to issue a RdV request. */
22       /* First allocate a packet wrapper */
23       err = nm_so_pw_alloc_and_fill_with_data(tag, seq,
24                                               iov[i].iov_base, iov[i].iov_len,
25                                               NM_SO_DATA_DONT_USE_HEADER, &p_so_pw);
26       if(err != NM_ESUCCESS) goto out;
27
28       /* Then place it into the appropriate list of large pending
29       "sends". */
30       list_add_tail(&p_so_pw->link, &(p_gate->pending_large_send[tag]));
31
32       { /* Finally, generate a RdV request */
33         union nm_so_generic_ctrl_header ctrl;
34         nm_so_init_rdv(&ctrl, tag, seq, iov[i].iov_len);
35         err = strat_default_pack_ctrl(_status, p_gate, &ctrl);
36         if(err != NM_ESUCCESS)
37           goto out;
38       }
39
40       /* Check if we should post a new recv packet: we're waiting for an
41          ACK! */
42       nm_so_refill_regular_recv(p_gate);
43     }
44   }
45  out:
46   return err;
47 }
```

FIG. 6.2: *Code source de la fonction* `pack` *de la stratégie par défaut de* NEWMADELEINE, `strat_default`.

```
int packv(void*_status,
          struct nm_gate *p_gate,
          uint8_t tag, uint8_t seq,
          struct iovec *iov, int nb_entries);
```

permettent de prendre en compte un message de données au sein de la stratégie. Les fonctions `pack` et `packv` se différencient sur le nombre de blocs de données qu'elles sont en mesure de prendre en compte : un seul bloc est pris en compte par la fonction `pack` alors que `packv` en traite `nb_entries`. Le paramètre `_status` correspond à l'instance de la stratégie courante, le paramètre `p_gate` à la structure décrivant la destination de la communication. Ces deux paramètres sont récupérés par l'utilisateur à l'initialisation de NEWMADELEINE, il ne s'agit donc que de les retransmettre à l'appel de la fonction. Les paramètres suivants décrivent les données à envoyer : le numéro de canal `tag` et la séquence `seq` de la communication, puis le couple longueur/adresse des données pour la fonction `pack` ou le couple nombre de blocs à envoyer/tableau descriptif de ces blocs pour la fonction `packv`. La structure `struct iovec` est composée de deux champs : `iov_len` qui correspond à la longueur du bloc et `iov_base`, à son adresse.

La Figure 6.2 présente le code source de la fonction `packv` de la stratégie `strat_default`. Le comportement choisi pour cette stratégie est basique : chaque bloc de données donne lieu à une communication réseau. Ainsi, pour chaque entrée du tableau `iov`, les données vont être traitées de la même façon. Si leur taille est inférieure à la constante NM_SO_MAX_SMALL, les données sont encapsulées à la suite de leur entête (lignes 15 à 17) et placées dans la liste des paquets prêts à être soumis à la couche de transfert (ligne 20). Sinon, elles sont encapsulées sous leur forme brute (lignes 23 à 25) puis placées dans une liste en attente de leur acquittement (ligne 30) et un rendez-vous est initié (lignes 33 à 37). La fonction `nm_so_alloc_and_fill_with_data` permet de créer un *packet wrapper* et d'y ajouter des données suivant la méthode indiquée par le paramètre `flag`. Si le `flag` est positionné à NM_SO_DATA_DONT_USE_COPY, la première entrée du tableau de `struct iovec` est remplie par l'entête et la seconde par les données. Si le `flag` est positionné à NM_SO_DATA_USE_COPY, l'entête et les données sont copiées en contigu dans le tampon pré-alloué au sein même de la structure d'encapsulation. Enfin, s'il est à NM_SO_DATA_DONT_USE_HEADER, les données ne sont pas liés directement à leur entête. La description des données remplit la première entrée du tableau de `struct iovec`. La fonction `pack` a strictement le même comportement : elle correspond à un appel à `packv` sur un seul bloc de données.

De l'autre côté, comme on a pu le voir à la ligne 35 du code de la fonction `packv` (Figure 6.2), la fonction `pack_ctrl` permet d'encapsuler un message de contrôle. En l'occurrence, il s'agissait d'encapsuler une demande de rendez-vous.

```
int (*pack_ctrl)(void*_status,
                 struct nm_gate *p_gate,
 union nm_so_generic_ctrl_header *p_ctrl);
```

La Figure 6.3 montre un comportement le plus simple que puisse adopter la fonction `pack_ctrl`. Pour chaque message de contrôle – un rendez-vous, un acquittement, un message de contrôle de flux, etc. – une nouvelle structure d'encapsulation est créée et initialisée comme il convient (ligne 10). Le nouveau paquet ainsi créé est placé dans la liste des paquets prêts à être soumis à la couche de transfert (ligne 15).

```
1  static int strat_default_pack_ctrl(void *_status,
2                                     struct nm_gate *p_gate,
3                                     union nm_so_generic_ctrl_header *p_ctrl){
4
5    struct nm_so_strat_default *status = _status;
6    struct nm_so_pkt_wrap *p_so_pw = NULL;
7    int err;
8
9    /* Simply form a new packet wrapper */
10   err = nm_so_pw_alloc_and_fill_with_control(p_ctrl, &p_so_pw);
11   if(err != NM_ESUCCESS)
12     goto out;
13
14   /* Add the control packet to the out_list */
15   list_add_tail(&p_so_pw->link, &status->out_list);
16
17  out:
18   return err;
19 }
```

FIG. 6.3: *Code source de la fonction `pack_ctrl` de la stratégie par défaut de* NEWMADELEINE *`strat_default`.*

6.2.2 Interface pour le socle de la couche d'ordonnancement

Nous nous intéressons à présent aux fonctions qui vont permettre d'invoquer la stratégie afin de soit récupérer un nouveau paquet à soumettre au réseau, soit d'obtenir des indications sur la manière de répondre à une prise de rendez-vous.

6.2.2.1 Sollicitation pour un nouveau paquet optimisé

La couche de transfert notifie la couche d'ordonnancement lorsqu'une de ses communications prend fin. La terminaison des envois entraîne l'invocation de la stratégie sélectionnée afin d'en obtenir un nouveau paquet optimisé prêt à être transmis.

```
/** Compute and apply the best possible packet rearrangement, then
    return next packet to send */
int try_and_commit(void*_status,
                   struct nm_gate *p_gate);
```

La Figure 6.4 montre le fonctionnement suivi par `strat_default`. Tout d'abord, on vérifie qu'il y ait des messages en attente de transmission. Si c'est le cas, la tête de la liste des paquet prêts à être soumis est retirée (lignes 13 et 14) et est donnée à la couche de transfert via la fonction `_nm_so_post_send` (ligne 20). Cette stratégie ne fait pas partie de celles que nous avons qualifiées d'*agressives* car elle n'applique aucune optimisation sur les données soumises. Les paquets finaux sont néanmoins construits au fur et à mesure de la soumission des messages de l'utilisateur. Ainsi, il n'apparaît pas ici la partie où la stratégie applique son algorithme d'optimisation afin d'en tirer la meilleure transmission à construire à cet instant donné.

```
1  static int strat_default_try_and_commit(void *_status,
2           struct nm_gate *p_gate){
3
4    struct nm_so_strat_default *status = _status;
5    struct list_head *out_list = &(status->out_list);
6    struct nm_so_pkt_wrap *p_so_pw = NULL;
7
8    if(list_empty(out_list))
9      /* No packet to send */
10     goto out;
11
12   /* Simply take the head of the list */
13   p_so_pw = nm_l2so(out_list->next);
14   list_del(out_list->next);
15
16   /* Finalize packet wrapper */
17   nm_so_pw_finalize(p_so_pw);
18
19   /* Post packet */
20   _nm_so_post_send(p_gate, p_so_pw);
21
22  out:
23     return NM_ESUCCESS;
24 }
```

FIG. 6.4: *Code source de la fonction* `try_and_commit` *de la stratégie par défaut de* NEWMADELEINE `strat_default`.

6.2.2.2 Remontée pour le traitement des prises de rendez-vous

La dernière fonction de l'interface que doit implémenter un développeur de stratégie est la fonction `rdv_accept`. Cette fonction est appelée lorsqu'une demande de rendez-vous vient d'être reçue et qu'une requête de réception lui a été associée. Il s'agit donc de déterminer sur quel réseau la réception va avoir lieu. Cette information permet alors de construire le message d'acquittement qui décrira à l'émetteur où aiguiller ses données.

```
/** Allow (or not) the acknowledgement of a Rendez-Vous request. */
int rdv_accept(void*_status,
               struct nm_gate *p_gate,
               uint32_t len_to_send,
               int * nb_drv,
               uint8_t *drv_ids,
               uint32_t *chunk_lens);
```

L'implémentation de cette fonction dans `strat_default` est donnée Figure 6.5. Grâce à l'échantillonnage, il est possible d'obtenir un tableau des réseaux disponibles triés selon leur performance en bande passante grâce à la fonction `nm_ns_dec_bws` (ligne 10). Afin de sélectionner les réseaux les plus performants pour le transfert du message long faisant l'objet du rendez-vous, la stratégie par défaut commence à passer en revue chaque pilote réseau, déterminant ainsi quels sont ceux disponibles (lignes 12 à 16). Pour cela, elle vérifie si des réceptions ne sont pas déjà en cours sur chaque réseau (ligne 14). Si ce n'est pas le cas, elle le sélectionne et sinon passe au suivant. À la suite de cela, si aucun réseau n'est disponible, la demande de rendez-vous sera stockée et l'acquittement différé jusqu'à la libération d'un réseau (ligne 20). Si seul un réseau s'avère libre, l'acquittement ordonnera la réception de l'ensemble du message sur ce dernier (ligne 23). Et sinon, le module d'échantillonnage est consulté pour déterminer quels réseaux il est utile de faire participer à la communication et dans quelles proportions à partir de la taille des données à recevoir (lignes 26 et 27). La fonction renseigne alors les paramètres de sortie `nb_drv` qui correspond aux nombres de réseaux que la communication va utiliser, `drv_ids` qui est un tableau recensant leurs identifiants et `chunks_len`, celui de la taille respective de chaque bloc devant être posté.

Le développement de ces fonctions ne requiert pas de connaissances techniques sur le fonctionnement et structures internes de NEWMADELEINE de la part du développeur. Il ne s'agit que de manipuler les données au travers de leur structure d'encapsulation. Afin d'en faciliter encore plus le maniement, un certain nombre de primitives ont été développées à cet effet. Tout développeur de logiciels de niveau supérieur à NEWMADELEINE est ainsi invité à compléter la bibliothèque de stratégies d'optimisation afin d'en produire une en adéquation avec les schémas de communication rencontrés dans sa propre application.

6.3 PIOMAN : un détecteur d'événements réactif

Précédemment, dans les Sections 3.2.2, 4.3 et 5.3.2, nous avons discuté de l'importance de détecter correctement les évènements provenant du réseau. La question de l'introduction d'un *thread* dédié à la progression des échanges afin d'améliorer leur recouvrement et la réactivité de la bibliothèque de communication en général a été soulevée. Afin de résoudre les problèmes liés à la cohabitation des *threads*

```
1  static int
2  strat_default_rdv_accept(void *_status, struct nm_gate *p_gate,
3                           uint32_t len_to_send,
4                           int * nb_drv, uint8_t *drv_ids, uint32_t *chunk_lens){
5    int nb_drivers = p_gate->nb_drivers;
6    uint8_t *ordered_drv_id_by_bw = NULL;
7    int cur_drv_idx = 0;
8    int i, err;
9
10   nm_ns_dec_bws(p_gate, &ordered_drv_id_by_bw);
11
12   for(i = 0; i < nb_drivers; i++){
13     if(nm_so_active_recv(p_gate, ordered_drv_id_by_bw[i]) == 0) {
14       drv_ids[cur_drv_idx] = i;
15       cur_drv_idx++;
16     }
17   }
18   *nb_drv = cur_drv_idx;
19
20   if(cur_drv_idx == 0){
21     /* We're forced to postpone the acknowledgement. */
22     err = -NM_EAGAIN;
23   } else if(cur_drv_idx == 1){
24     err = NM_ESUCCESS;
25   } else {
26     nm_ns_multiple_split_ratio(len_to_send, cur_drv_idx,
27                                drv_ids, chunk_lens, nb_drv);
28     err = NM_ESUCCESS;
29   }
30
31   return err;
32 }
```

FIG. 6.5: *Code source de la fonction* `rdv_accept` *de la stratégie par défaut de* NEWMADELEINE `strat_default`.

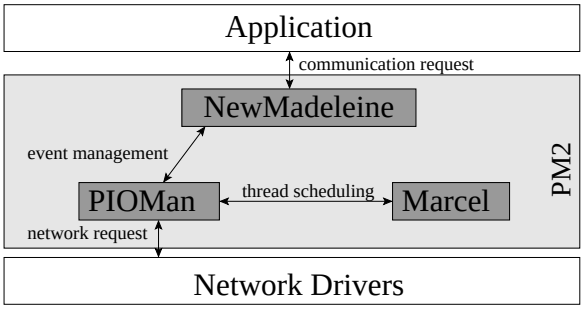


FIG. 6.6: *La suite logicielle PM2.*

et des communications, nous proposons de confier la gestion de l'interaction entre *threads* et communications à un gestionnaire d'entrées-sorties fournissant un service de détection d'événements nommé PIOMAN, développé au sein de RUNTIME par François Trahay. Son mode de fonctionnement s'organise autour de différents concepts que nous présentons dans la suite de cette section. Ainsi, PIOMAN prend la place de la boucle de progression de la couche de transfert décrite dans le chapitre 5 mais va plus loin encore. Grâce à la vision globale de l'occupation de la machine et la relation qu'il entretient avec l'ordonnanceur de *threads*, PIOMAN sait tirer parti de toutes les ressources en processeur qui pourraient être inutilisées.

6.3.1 Centralisation des requêtes de communication

En centralisant la prise en charge de l'interaction entre *threads* et communications, il est possible d'avoir une vision globale du système et des requêtes d'entrées-sorties. Les perspectives offertes sont alors multiples. Tout d'abord, les problèmes de compétition pour accéder aux cartes réseau sont clairement arbitrés. Les opérations sont faites les unes après les autres, qu'il s'agisse d'une soumission de message ou d'une détection de complétion. En effet, la soumission d'une requête réseau par l'ordonnanceur de NEWMADELEINE ne se fait que lorsque la carte est disponible. Elle est donc effectuée en un temps miminal, aucune opération ne pouvant *a priori* la perturber. De même, les requêtes à compléter entraînent à tour de rôle l'interrogation de la carte sur laquelle elles ont été postées.

Ensuite, lorsque la technologie réseau le permet, il peut être intéressant de grouper plusieurs requêtes afin de n'exécuter la méthode de détection qu'une fois pour toutes les requêtes, résumant ainsi la détection des événements réseau à un appel par carte réseau utilisée. Enfin, on peut encore imaginer bien d'autres mécanismes comme affecter des priorités aux requêtes afin de favoriser la détection de données les plus importantes pour l'application.

6.3.2 Travail en lien avec l'ordonnanceur de *threads*

PIOMAN interagit avec un ordonnanceur de *threads* à deux niveaux. Ce type d'ordonnanceur a pour caractéristique de créer un *thread* de niveau noyau par processeur afin d'y encrer plusieurs *threads* utilisateurs. L'ordonnancement de ces derniers peut alors s'effectuer entièrement en espace utilisateur, sans aucune interaction coûteuse avec le système d'exploitation. En l'occurrence, comme le montre la Figure 6.6, PIOMAN travaille de pair avec l'ordonnanceur de *threads* MARCEL, également développé

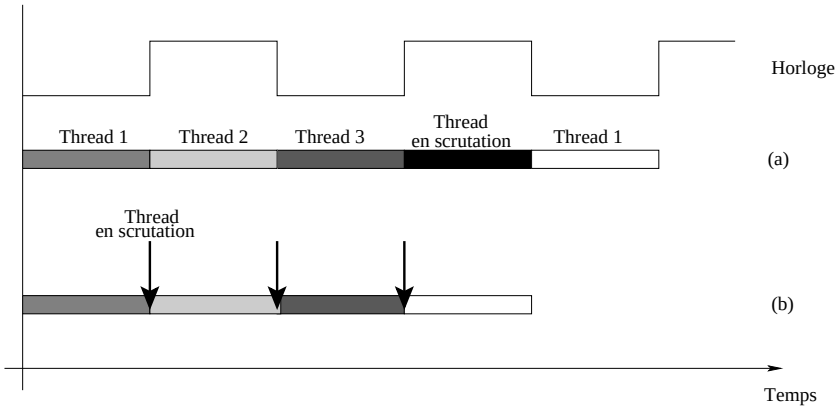


FIG. 6.7: *Ordonnancement des* threads *de calcul et de scrutation. L'ordonnancement régulier (a) traite le* thread *chargé de la scrutation au même titre que ceux de calculs. Il n'est donc pas ordonnancé fréquemment et monopolise un processeur pendant un quantum de temps sans justification. L'ordonnancement (b) intercale le* thread *de scrutation à chaque changement de contexte. Cela n'impacte pas significativement les* threads *de calcul et permet d'augmenter la fréquence de scrutation et donc de réactivité.*

dans l'équipe par Vincent Danjean [Dan04]. Cette collaboration lui permet d'obtenir des informations utiles sur l'état de la machine comme le nombre de *threads* prêts, le nombre de processeurs, etc.

Dans la suite de cette section, nous allons voir quelles perspectives ouvre la connaissance de telles informations et comment PIOMAN peut alors influencer l'ordonnancement des *threads* afin d'améliorer le déroulement des communications.

6.3.2.1 Pour améliorer la réactivité

Comme nous l'avons vu dans la Section 3.2.2, la détection des évènements par scrutation permet d'obtenir une bonne réactivité si le nombre de *threads* en concurrence est inférieur au nombre de processeurs. En effet, tant qu'un processeur est inoccupé, il est intéressant et non pénalisant de l'utiliser afin d'attendre activement un événement réseau. Cependant, quand la machine vient à se charger, il est beaucoup moins judicieux de monopoliser un processeur pour ce type de tâche alors qu'il pourrait servir à poursuivre les calculs de l'application. Par ailleurs, plus le nombre de *threads* est conséquent plus la fréquence de d'ordonnancement du *thread* de scrutation va augmenter, impliquant une baisse importante de réactivité notable.

Dans le cas de communications ayant des temps de transfert courts – et ce, malgré la surcharge de la machine –, l'attente active reste néanmoins la méthode la plus adaptée par rapport à l'utilisation d'un appel bloquant (se référer à la Section 3.2.2) ; ce dernier étant pénalisé par le surcoût imposé par le traitement de l'interruption. Afin de ne pas trop dégrader les temps de détection d'évènements réseau, PIOMAN se sert de MARCEL pour être appelé à certains moments clés de l'ordonnancement comme un changement de contexte ou un signal d'horloge. Comme le montre la Figure 6.7, l'impact sur les calculs est alors négligeable puisque les opérations de détection d'évènement réseau sont intercalées entre deux *threads* de calcul. Le temps de réaction à un événement réseau s'élève alors à un demi quantum de temps en moyenne.

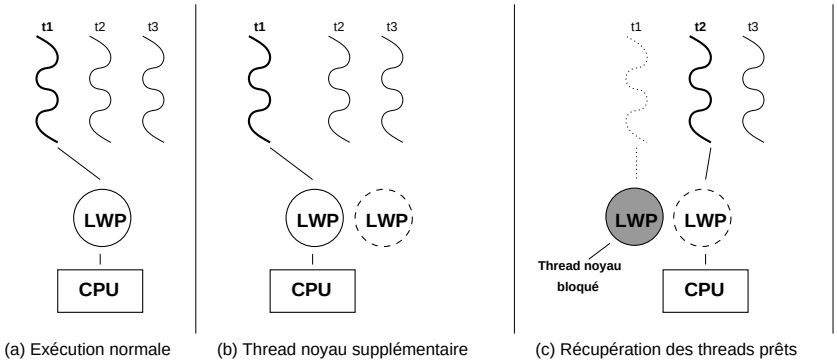


FIG. 6.8: *Utilisation d'un* thread *noyau (LWP) supplémentaire pour permettre un appel bloquant.*

Dans le cas de communications longues, l'utilisation d'appels bloquants peut être plus appropriée. En effet, le coût de traitement de l'interruption étant noyé dans le coût de la communication elle-même, autant en profiter pour libérer totalement les processeurs de la machine de toutes opérations de détection d'évènements réseaux. Toutefois, l'utilisation d'appels bloquants avec un ordonnanceur à deux niveaux est assez délicat. En effet, un *thread* effectuant un appel bloquant va bloquer son *thread* noyau de rattachement empêchant les autres *threads* s'exécutant sur ce même *thread* noyau de continuer leur progression. Les appels bloquants sont donc généralement évités sur ce genre de systèmes. Comme la Figure 6.8 le montre, afin de contourner ce problème, avant d'exécuter toute fonction potentiellement bloquante, PIOMAN réveille un *thread* noyau supplémentaire chargé de secourir les *threads* utilisateurs bloqués. Si la fonction se trouve être bloquante, les *threads* restants peuvent alors s'exécuter sur ce *thread* noyau supplémentaire jusqu'à ce que la carte réseau génère une interruption et réveille le *thread* de communication. Ce dernier peut ensuite reprendre son exécution et demander le rapatriement des autres *threads*. À l'inverse, si l'appel s'effectue sans bloquer le *thread*, ce dernier pourra continuer son exécution normalement. Le *thread* noyau supplémentaire ayant une faible priorité, celui-ci ne sera pas ordonnancé dès son réveil mais après que le *thread* communicant soit bloqué. Cette technique a l'avantage de ne nécessiter aucune modification du système d'exploitation, contrairement aux *Scheduler Activations* [ABLL91].

Outre l'état de la machine, le gestionnaire d'entrées-sorties prend en compte les informations que peut lui donner la bibliothèque de communication. En effet, celle-ci a une plus grande connaissance du temps de complétion des requêtes. Par exemple, lorsqu'une application communique avec MX/Myrinet et utilise TCP/Ethernet comme canal de contrôle, les données de contrôle sont rares et ne nécessitent pas une réactivité élevée. La bibliothèque de communication peut alors suggérer au gestionnaire d'entrées-sorties d'utiliser un appel bloquant pour les requêtes s'effectuant sur TCP, même lorsqu'un processeur est inoccupé. Ainsi, la scrutation ne concernera que les requêtes utilisant le réseau Myrinet et l'application ne sera plus perturbée par la détection d'événements sur le canal de contrôle.

Finalement, l'interaction avec l'ordonnanceur de *threads* permet également à PIOMAN de préciser à l'ordonnanceur de réveiller au plus vite les *threads* dont la requête de communication vient de s'achever.

6.3.2.2 Pour s'étendre sur toute la machine

Grâce au rapport privilégié que PIOMAN a avec l'ordonnanceur de *threads*, il est également en mesure de savoir si des processeurs sont inactifs. Cette connaissance de l'occupation de la machine lui permet ainsi de savoir où et quand déporter des opérations sur des processeurs inactifs en déchargeant par

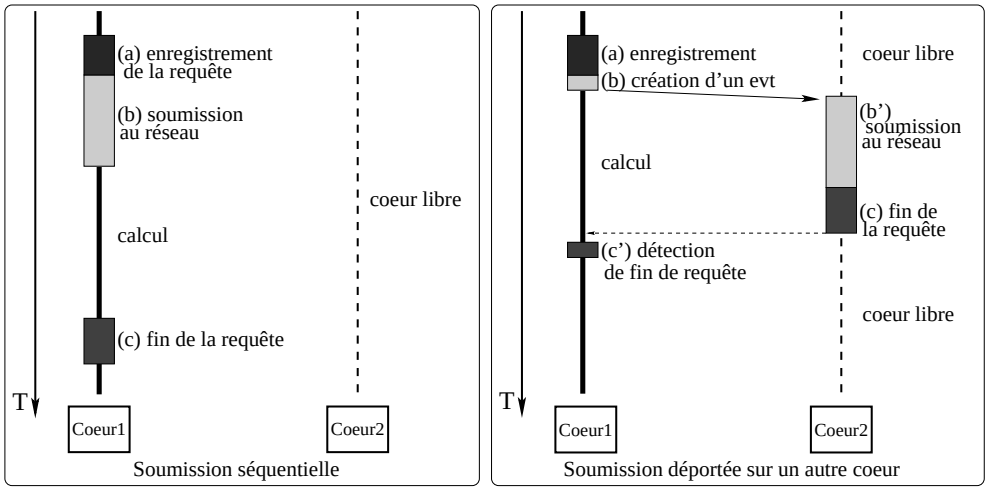


FIG. 6.9: *Progression de communications asychnrones en parallèle du calcul.*

la même occasion le processeur courant. On peut ainsi atteindre un taux d'utilisation de la machine important. Dans la suite de ce paragraphe, nous nous intéressons au cas particulier du déport de la soumission de requêtes d'émission asynchrones mais il existe un large spectre d'exploitation. On peut par exemple imaginer déporter l'application d'une stratégie d'ordonnancement de communication, ou l'élection de la plus adaptée à un moment donné, mais encore déplier un type dérivé complexe (*i.e.* un *datatype* de MPI), ou appliquer un algorithme de compression aux données, etc. Toute tâche n'ayant pas de dépendance avec le contexte courant de l'exécution de l'application est susceptible d'être déportée.

Déport de la soumission de requête d'émission asynchrone Les messages de taille réduite (typiquement de taille inférieure à 32 Ko) sont en général acheminés jusqu'à la carte réseau en utilisant copie bloquante. Les interfaces de communication bas-niveau n'assurant en général pas la progression de leurs communications en cours en interne, afin d'émettre les données au plus tôt, la soumission d'une requête d'émission se traduit par le transfert direct de ces données à la carte pour une émission immédiate, si cela s'avère possible. Dans une telle situation, qu'elle ait déposé une demande d'envoi synchrone ou asynchrone, l'application commanditaire va devoir attendre que ce transfert soit achevé avant d'être libérée et de pouvoir continuer son exécution.

Comme le montre la figure 6.9, au lieu de directement soumettre la communication sur le réseau, PIOMAN crée une tâche sur le processeur courant qui peut ensuite être exécutée sur n'importe quel cœur de la machine. Par ce procédé, l'émission est significativement recouverte par les calculs que l'application est alors en mesure de faire et ce, jusqu'à demande de complétion de la requête. Cette technique a fait l'objet d'un article auquel on peut se référencer pour plus de détails [TBDN08].

Déport de la soumission de requête d'émission sur plusieurs réseaux Sur le même principe que le déport des émissions asynchrones du paragraphe précédent, nous les déportons désormais sur plusieurs réseaux simultanément, comme l'expose la Figure 6.10.

En effet, afin de profiter à la fois des ressources de communication et des ressources de calcul, nous avons mis au point cette dernière optimisation. Contrairement au déport simple d'une communication

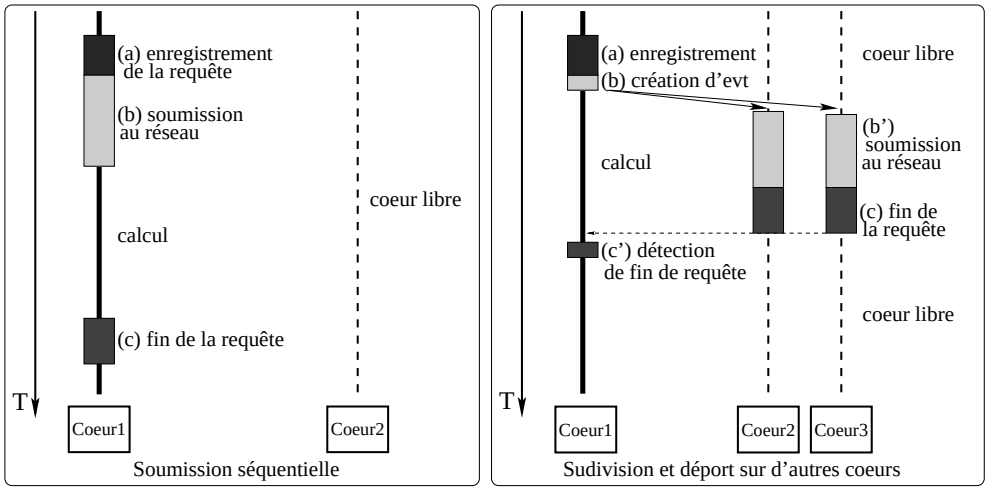


FIG. 6.10: *Envoi en parallèle de messages envoyés par copie sur plusieurs réseaux.*

qui est totalement assumé par PIOMAN, ici, NEWMADELEINE et PIOMAN travaillent de pair pour garantir la meilleure opération de communication possible.

À chaque fois que PIOMAN a une soumission de transfert à traiter et que des processeurs se révèlent être inactifs, il interroge NEWMADELEINE afin de savoir comment diviser le message courant en différents fragments ayant des temps de transfert équivalents. Pour cela, NEWMADELEINE fait bien évidemment appel à l'échantillonnage. Finalement, il s'agit d'appliquer la stratégie de multirail mais uniquement aux messages ne requérant pas de prise de rendez-vous. En effet, cette optimisation dépendant de ressources vacantes de la machine au moment même de l'émission, il est impossible d'en informer le nœud distant pour qu'il adopte le même schéma de communication en réception et donc de l'utiliser pour des messages nécessitant une prise de rendez-vous préalable. Ces travaux sont précisés dans l'article [BTD08].

Troisième partie

Validation et Conclusion

Chapitre 7

Évaluations

Sommaire

Dans ce chapitre, nous présentons les expériences que nous avons menées afin de valider NEWMADELEINE. Dans un premier temps, des tests synthétiques nous permettrons de vérifier que les surcoûts bruts introduits restent acceptables sur des schémas de communication où il n'y a pas lieu de procéder à des optimisations. Puis nous présentons des tests jouets afin de mettre en évidence le gain qu'il y a à utiliser des stratégies d'optimisation sur des schémas de communication à peine plus évolués. Dans un second temps, nous mettons NEWMADELEINE en situation réelle. En effet, les deux dernières parties montrent NEWMADELEINE servant de bibliothèque de communication à des applications qui s'exécutent usuellement au dessus d'implémentations classiques de MPI.

7.1 Tests synthétiques

Comme nous venons de l'introduire, cette partie sert de validation à NEWMADELEINE en terme de fonctionnalité et de performance. Il s'agit de montrer qu'en environnement contrôlé, notre bibliothèque

de communication se comporte comme nous l'espérons et qu'il est ainsi facile d'obtenir des gains significatifs sur des configurations d'échanges de données demeurant simples.

7.1.1 Performances brutes avec la stratégie par défaut

L'ensemble des tests qui suivent sont réalisés sur des machines quad-core Xeon cadencées à 3,16 GHz équipées de cartes réseau de type MYRI-10G. Les implémentations de MPI utilisées comme point de comparaison sont OPEN MPI 1.2.7 et MPICH2-MX-1.0.7.

7.1.1.1 Surcoût brut par message élémentaire

La latence étant en général un point critique de comparaison, nous commençons par l'évaluation du surcoût de NEWMADELEINE sur l'envoi de données brutes. Ce premier test n'apporte pas réellement de valeur ajoutée puisque nous mettons volontairement NEWMADELEINE dans une situation où elle ne peut que subir les communications qui lui sont soumises, aucune optimisation ne pouvant être appliquée. Cependant, même si son but n'est pas de privilégier ce type de communication, il s'agit néanmoins d'avoir des performances raisonnables même sur ces cas de figure défavorables.

Pour cela, nous employons un test de *ping-pong* classique qui consiste en une série d'aller-retour simples sur le réseau dont nous moyennons les résultats afin de lisser les éventuelles fluctuations.

Version sans *thread* Nous commençons par mesurer les performances de NEWMADELEINE dans sa version non multithreadée en les comparant à celles des implémentations de MPI-2 courantes, MPICH2-MX et OPEN MPI/MX. Sont également données celles de l'interface de communication bas-niveau MX à titre de référence. Les Figures 7.1 et 7.2 présentent les résultats obtenus en matière de temps de transfert et de bande passante.

Nous observons un surcoût logiciel de l'ordre de la demi-microseconde par rapport aux latences brutes du réseau. Tandis que les résultats de MPICH2-MX – l'implémentation développée spécifiquement par MYRICOM à partir de MPICH2–, sont pratiquement confondus avec ceux de MX, ceux de OPEN MPI atteignent un surcoût logiciel d'environ 300 nanosecondes. De ce fait, notre surcoût est comparable à celui d'un support de communication qu'une application utiliserait en réalité.

Cependant, gardons à l'esprit que cette perte de performance est principalement induite par deux facteurs que nous avons volontairement introduits afin d'accéder à des opportunités d'optimisation importantes dans le cas de schémas de communication plus complexes. Tout d'abord, la description des entêtes implique que, à taille de données utiles égales, la masse des données effectivement transmise est supérieure avec NEWMADELEINE. Ensuite, l'application d'optimisation sur un élément unique ne peut être que superflue. Il paraît donc naturel d'être légèrement pénalisé sur les cas où l'intervention de l'optimiseur est inutile.

Du point de vue de la bande passante, la Figure 7.2 montre que NEWMADELEINE atteint des débits du même ordre que ceux des autres implémentations considérées.

L'objectif de ce test étant de vérifier que les surcoûts induits dans ces cas de figure basiques ne sont pas aberrants, ces performances sont jugées acceptables en contrepartie des opportunités ouvertes pour les schémas de communications réalistes.

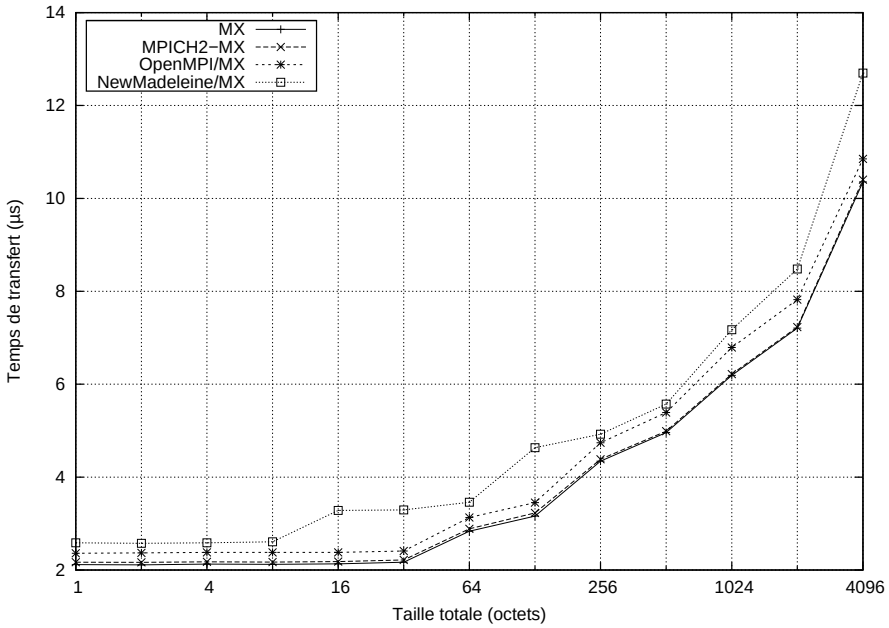


FIG. 7.1: *Ping-pong* - NEWMADELEINE *vs* MPICH2-MX *et* OPEN MPI/*MX* - *Latence.*

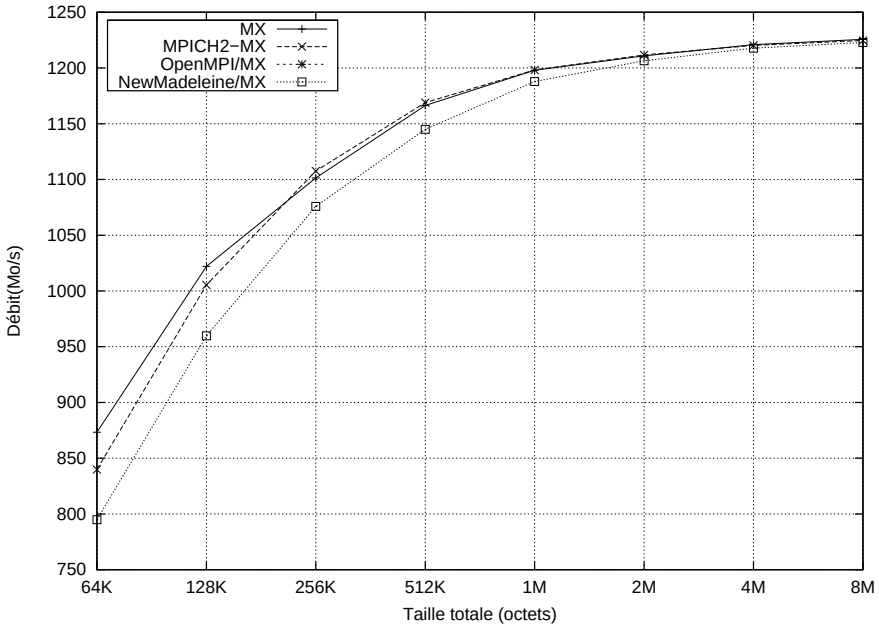


FIG. 7.2: *Ping-pong* - NEWMADELEINE *vs* MPICH2-MX *et* OPEN MPI/*MX* - *Débit.*

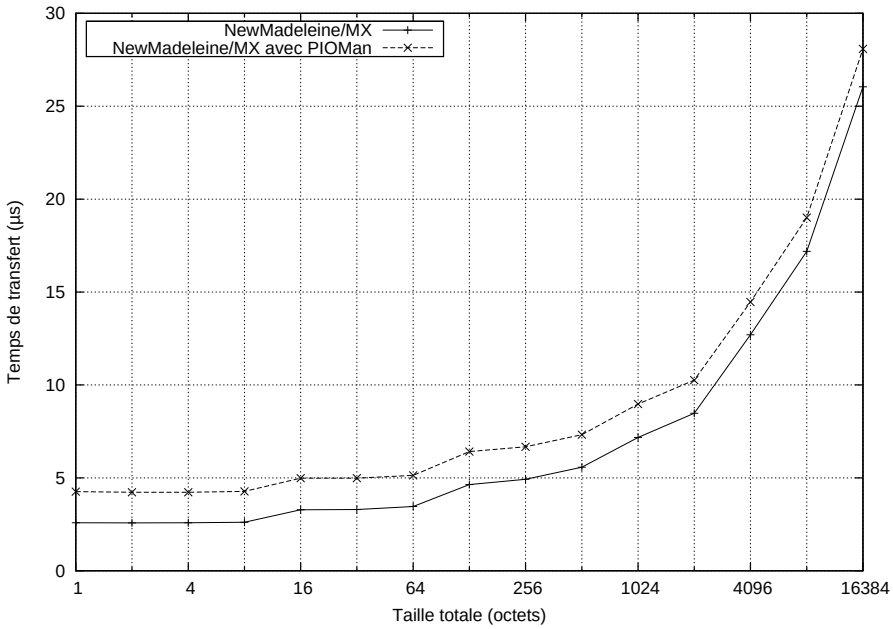


FIG. 7.3: *Ping-pong - Surcoût introduit par l'utilisation de* PIOMAN- *Latence.*

Version multithreadée Nous procédons maintenant au même type de test mais en activant le support aux *threads* offert par PIOMAN au sein de NEWMADELEINE. L'utilisation de *thread* entraîne des surcoûts autrement plus importants qui sont principalement dûs aux mécanismes de synchronisation et aux communications nécessaires entre les processeurs (remplissage du cache, etc.). Nous comparons ainsi les performances de NEWMADELEINE seule avec celles de NEWMADELEINE s'appuyant sur PIOMAN sur les Figures 7.3 et 7.4. Aucune implémentation de MPI au-dessus du pilote de communication MX/Myrinet n'offrant de support de *threads* de niveau MPI_THREAD_MULTIPLE, nous ne pouvons réaliser d'autre point de comparaison.

Ce test a l'originalité de proposer des performances qu'il n'est pas courant de rencontrer. En effet, les surcoûts introduits par l'utilisation de *threads* étant ce qu'ils sont, les performances extrêmes des réseaux haute performance employés en sont minimisées. Le rôle de cette partie est donc ici de donner un point de référence au lecteur afin qu'il ne s'alarme pas des temps de transfert que nous pourrons rencontrer par la suite. Nous avions d'ailleurs déjà pu le constater à la Figure 3.2 de la partie 3.1.2 sur le support au multithreading offert par les implémentations de MPI.

7.1.1.2 Recouvrement de communications par du calcul

Cette partie a pour but de mettre en évidence l'importance de la progression des communications en parallèle des calculs. La Figure 7.5 présente le temps nécessaire à deux *threads* pour s'échanger un message de taille réduite lorsque le nombre de *threads* de calcul **par processeur** varie. L'ordonnancement du *thread* communicant se raréfiant avec l'augmentation du nombre de *threads* sur le même processeur, nous observons ainsi le net accroissement de ses temps de réaction au fur et à mesure de l'arrivée de nouveaux *threads* si la scrutation est employée. En contrepartie d'un coût certain, les appels bloquants stabilisent et réduisent quant à eux grandement ces temps de réaction quel que soit le nombre de *threads* en concurrence.

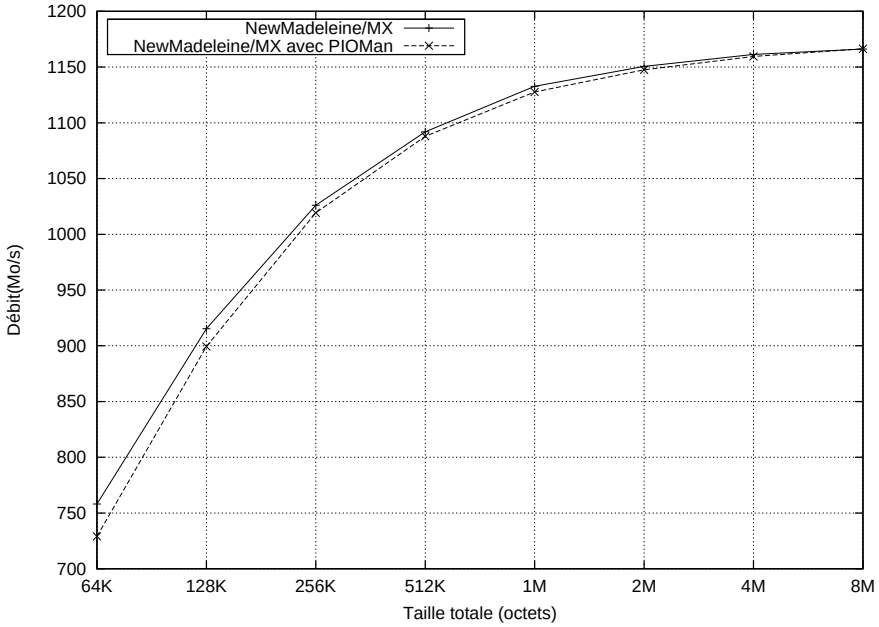


FIG. 7.4: *Ping-pong - Surcoût introduit par l'utilisation de* PIOMAN*- Débit.*

Ces premières réflexions montrent l'utilité de pouvoir adapter la méthode de détection d'évènement réseau au contexte d'exécution de la machine.

Le prochain test montre ainsi comment PIOMAN arrive à recouvrir les temps de communication par du calcul en faisant progresser les communications asynchrones en arrière plan à l'aide de ses différentes méthodes de détection. Puis nous voyons comment PIOMAN va encore plus loin en déportant les émissions des données nécessitant des copies sur un processeur inoccupé.

Progression en tâche de fond Afin d'évaluer l'efficacité de la progression des communications asynchrones grâce à PIOMAN, nous utilisons le programme décrit dans la Table 7.1. Ce programme tente de recouvrir la communication en réception par du calcul. Pour ce faire, nous nous plaçons du côté émetteur et observons les fluctuations de temps nécessaires à l'envoi de données.

La Figure 7.6 montre les résultats obtenus sur MX/Myrinet. Pour les messages de petite taille, nous observons une durée d'émission proche de la latence du réseau. En effet, l'activité du récepteur n'influence en rien le mode d'envoi des messages. Pour les messages de taille plus conséquente, au-delà du seuil de *rendez-vous*, les comportements diffèrent :

Émetteur	*Récepteur*

`get_time(t1);`
`nm_send(...);`
`get_time(t2);`
	`nm_irecv(...);`
	`compute();`
	`/* calcul pendant env. 50ms */`
	`nm_rwait(...);`

TAB. 7.1: *Programme d'évaluation de la progression de communications asynchrones.*

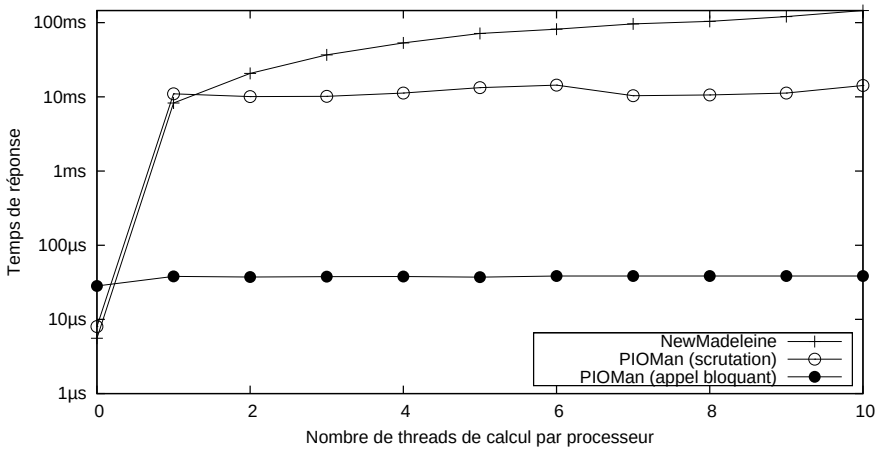


FIG. 7.5: *Temps de réponse en fonction du nombre de* threads *de calcul par processeur sur MX/Myri-10G.*

NEWMADELEINE – NEWMADELEINE seule n'est pas capable de faire progresser le *rendez-vous* en arrière-plan. L'émetteur est donc bloqué jusqu'à ce que le récepteur atteigne une instruction de terminaison de communication (*i.e.* un `wait`, ou un `test`).

PIOMAN- scrutation – La version monoprocesseur de PIOMAN fait progresser le *rendez-vous* lorsque l'ordonnanceur est exécuté. Le temps d'émission est donc borné à un quantum de temps (typiquement de l'ordre de 10 ms) et non à la durée du calcul intervenant du côté du récepteur.

PIOMAN- appel bloquant – La version multiprocesseur de PIOMAN exécute un appel bloquant pour attendre la demande de *rendez-vous* du côté du récepteur et continue le calcul sur un *thread* noyau supplémentaire. Le *rendez-vous* progresse donc sans être gêné par le calcul.

Recouvrement des temps de copie Pour encore améliorer le recouvrement des communications asynchrones mais cette fois-ci en émission, PIOMAN se sert de son rapport privilégié avec l'ordonnanceur de *threads* afin de connaître l'état d'occupation des processeurs de la machine et de tirer profit d'une unité de calcul libre, le cas échéant. En effet, comme nous l'avons expliqué dans la partie 6.3.2.2, le transfert en PIO des données jusqu'à la carte réseau – ou plus généralement la soumission des données au réseau – monopolise le processeur sur lequel le *thread* communicant vient de soumettre sa requête. Or, cette opération peut être faite de manière autonome et peut tout à fait être déportée sur une autre unité de calcul afin d'être faite en parallèle, *i.e.* recouverte.

Afin d'évaluer la capacité de PIOMAN à appliquer ce genre de technique, nous proposons d'utiliser le programme décrit dans la Table 7.2. L'émetteur y produit une requête d'envoi asynchrone, retourne à ses calculs pour une durée de 20 μs et vient par la suite en attendre la terminaison.

Les résultats obtenus sont présentés sur la Figure 7.7. La version qui ne déporte pas la soumission des requêtes d'émission sur un processeur libre atteint des temps de transfert qui concordent à la somme des

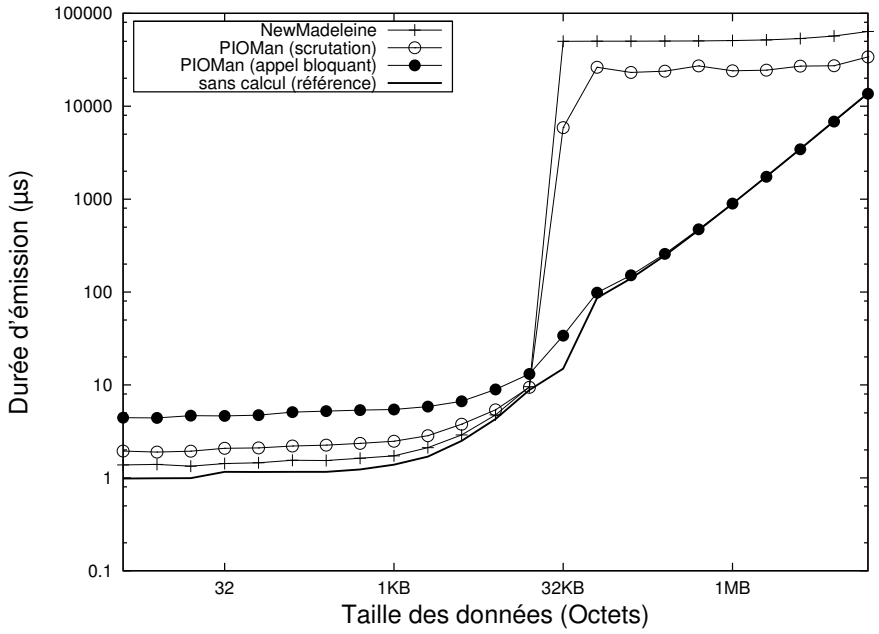


FIG. 7.6: *Temps d'émission sur MX/Myri-10G.*

Émetteur	*Récepteur*
`get_time(t1);`	
`nm_isend(len);`	
`compute(); // 20 µs`	`nm_recv(...);`
`nm_swait();`	
`get_time(t2);`	

TAB. 7.2: *Programme d'évaluation du déport de la soumission des données au réseau.*

temps de communication et de calcul. Finalement, il ne s'agit strictement que d'une translation de la courbe où seules les communications sont faites. La seconde version – qui elle les déporte – affiche des performances qui correspondent au maximum du couple temps de communication/temps de calcul. En effet, la communication est masquée par le temps de calcul tant que celui-ci lui est supérieur. Une fois ce seuil passé, l'échange s'achève lorsque la communication prend fin. Il est cependant à noter que 2 μs sont néanmoins nécessaires à l'initialisation du déport consistant en opérations inter-processeurs.

Bilan Avec l'aide de PIOMAN, les communications asynchrones de NEWMADELEINE progressent en arrière-plan, permettant ainsi le recouvrement des transferts de données pendant des phases de calcul. De plus, les unités de calcul inutilisées sont efficacement employées à décharger de toutes opérations gourmandes en ressources de calcul celles sur lesquelles les *threads* applicatifs s'exécutent.

7.1.2 Agglomération des communications

A présent, nous nous intéressons aux cas de figure où l'agrégation de messages est favorable. Nous commençons par étudier où les messages d'un même flux peuvent être regroupés avant de se pencher

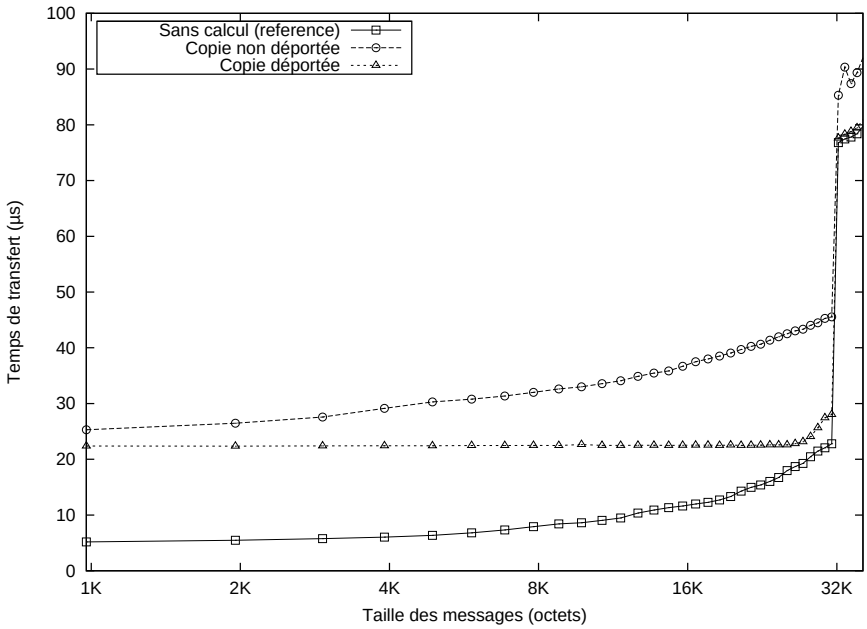


FIG. 7.7: *Temps de transfert des messages dont la soumission au réseau est déportée sur un autre cœur.*

sur le cas de ceux provenant de flux de communication différents.

7.1.2.1 Au sein d'un même flux de communication

En premier lieu, afin d'exposer l'intérêt de l'agrégation de messages, nous considérons un ping-pong où chaque séquence d'envoi est composée d'une multitude d'envois de petits segments de même taille dont nous faisons varier le nombre. Les résultats obtenus sont présentés dans les Figures 7.8 et 7.9. Pour plus de clarté, si nous considérons une taille cumulée de 4 octets, en partant du bas, le premier point correspond à un envoi de 4 octets, le second à deux envois de 2 octets et le dernier à 4 envois de 1 octet. À la vue de ces courbes, le gain à agréger apparaît comme indiscutable.

En second lieu, nous poursuivons avec un test ayant pour but de mettre à l'épreuve la stratégie d'agrégation de NEWMADELEINE. Pour cela, nous considérons un ping-pong où chaque séquence d'envoi découpe la taille à envoyer en blocs de 1 octet, ceci permettant d'accroître considérablement nos opportunités d'agrégation et donc d'optimisation. Puis, nous menons l'expérience une fois en autorisant l'agrégation de ces requêtes et une fois en l'interdisant. Les performances obtenues sont présentées dans les Figures 7.10 et 7.11.

Avec des gains pouvant atteindre les 60 %, cette stratégie d'optimisation ne peut que se révéler particulièrement adaptée aux applications qui échangent très fréquemment des messages courts comme des messages de contrôle ou de synchronisation, ou encore à celles utilisant des datatypes décrivant une multitude de petits blocs.

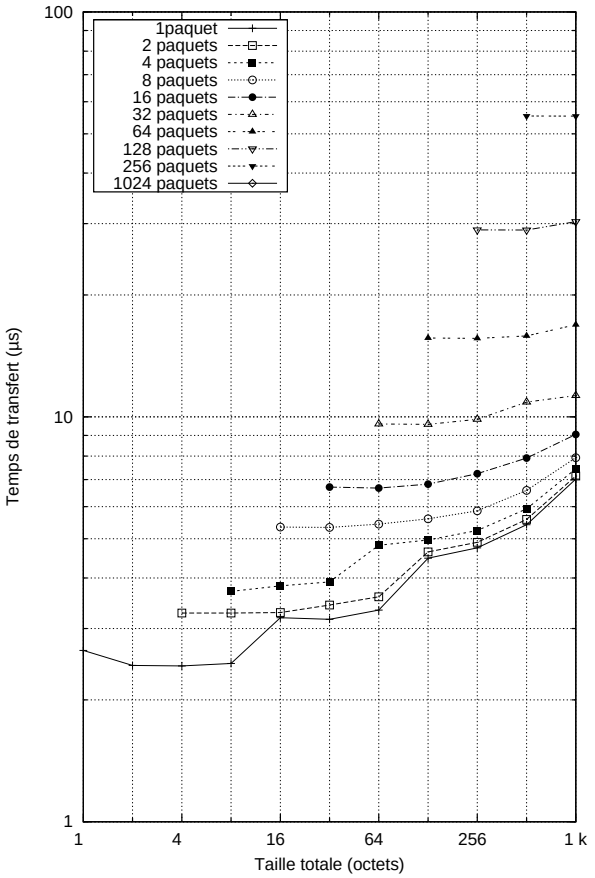


FIG. 7.8: *Subdivision d'un message en un nombre de paquets variable - Latence.*

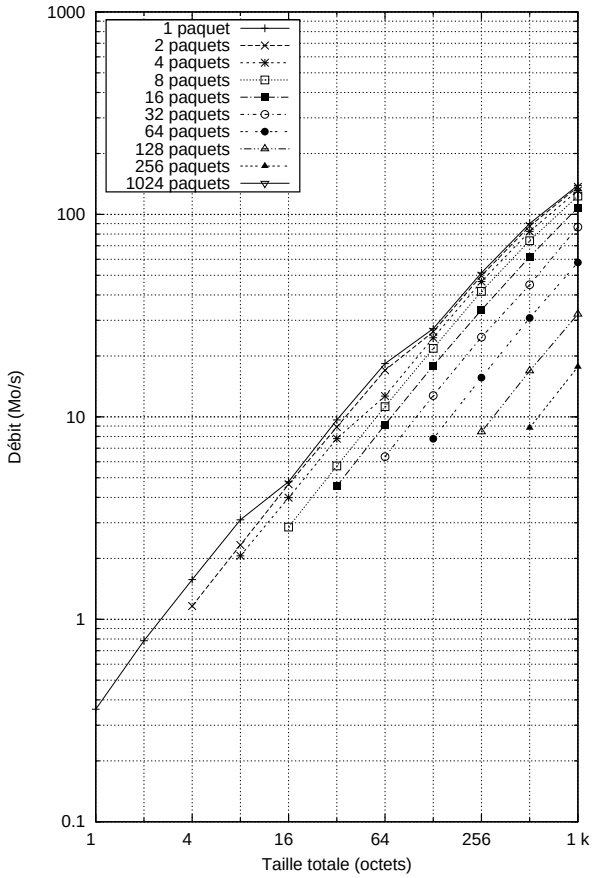


FIG. 7.9: *Subdivision d'un message en un nombre de paquets variable - Débit.*

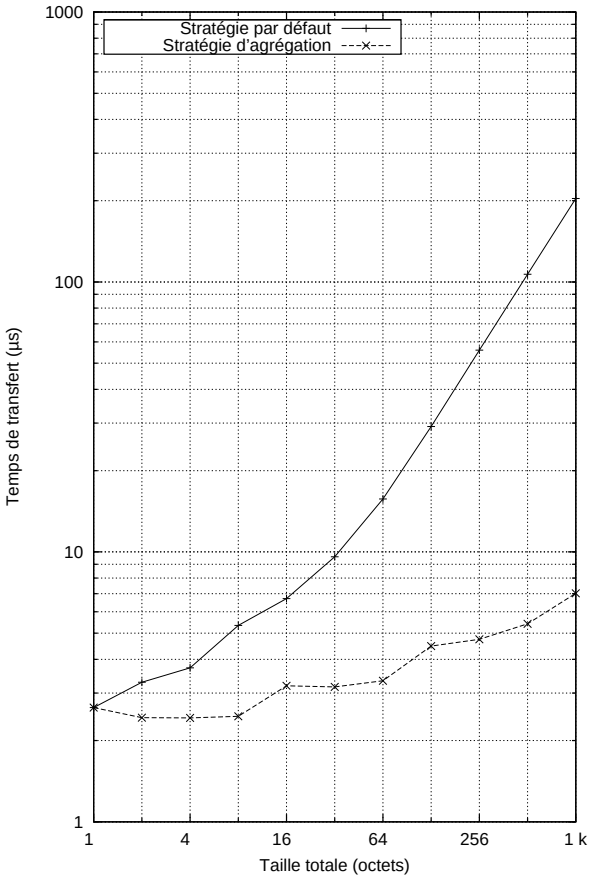


FIG. 7.10: *Envoi de N messages consécutifs (N étant la taille cumulée à transférer) de manière indépendante ou regroupée - Latence.*

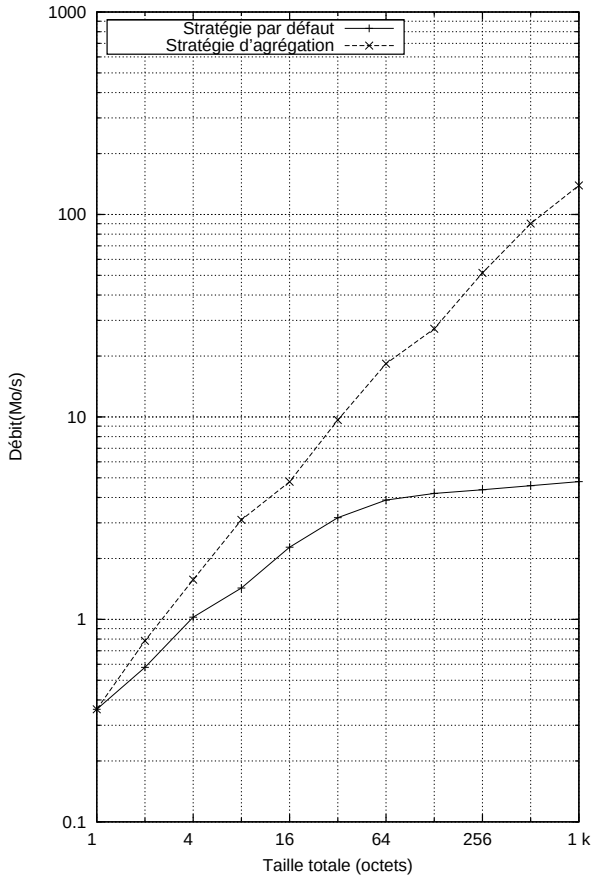


FIG. 7.11: *Envoi de N messages consécutifs (N étant la taille cumulée à transférer) de manière indépendante ou regroupée - Débit.*

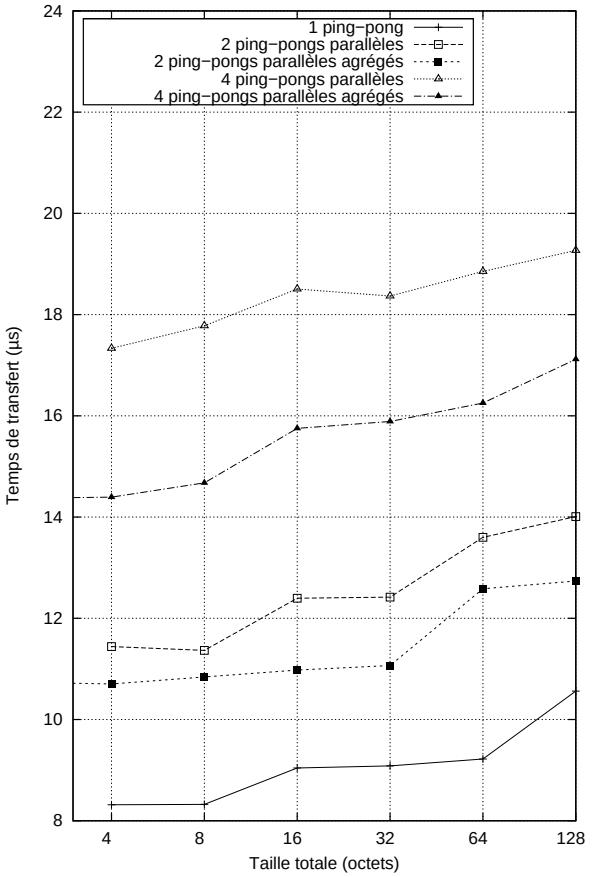


FIG. 7.12: *Mise en concurrence de plusieurs ping-pongs - Latence.*

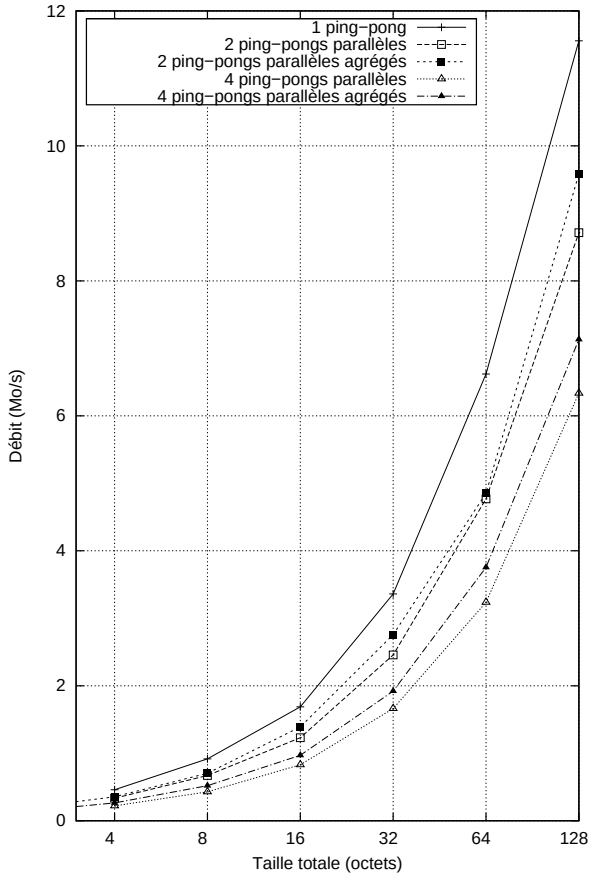


FIG. 7.13: *Mise en concurrence de plusieurs ping-pongs - Débit.*

7.1.2.2 Entre différents flux de communication

Toujours dans l'objectif de mettre en valeur l'intérêt de l'agrégation, nous procédons à présent à la mise en situation de plusieurs messages provenant de sources d'exécution différentes. Pour cela, dans un environnement multithreadé – *i.e.* avec PIOMAN activé –, plusieurs ping-pongs de facture classique sont lancés en parallèle. L'expérience est menée avec et sans emploi de l'agglomération de messages.

Les Figures 7.12 et 7.13 en exposent les comportements. Comme pour le test précédent, l'agglomération des messages ouvre la possibilité à des gains importants qui ne peuvent que profiter à des applications sensibles à la latence.

7.1.3 Distribution des messages sur plusieurs réseaux

Comme introduite dans la partie 5.4.2.3, la mise à niveau du nombre de ressources de communication par rapport à celui des unités de calcul a mené à l'ajout de cartes réseau au sein des nœuds composant les architectures en grappe actuelles. Les évaluations de cette partie ont été menées sur des machines dual dual-core Opteron équipées de deux cartes réseau de technologie différente (en l'occurrence, MYRI-10G et QSNET II) afin de mettre en avant le rôle que peut jouer le module d'échantillonnage mais

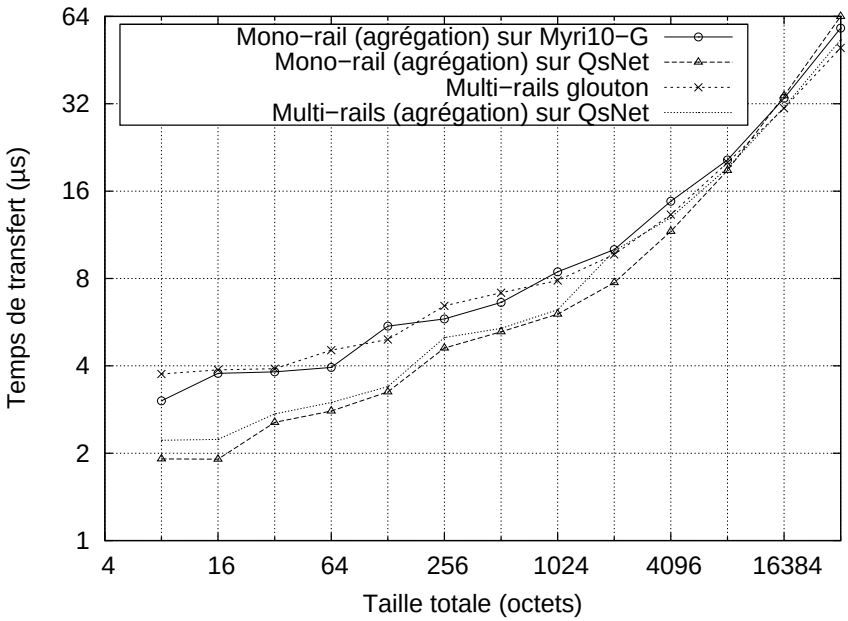


FIG. 7.14: *Envoi des agrégations de messages courts sur le réseau le plus rapide et distribution des messages longs sur l'ensemble des cartes disponibles - Latence*

peuvent tout à fait l'être au-dessus de rails homogènes.

Construction de la stratégie L'évaluation de la stratégie visant à maximiser l'emploi simultané de plusieurs cartes réseau est quelque peu différente des précédentes. En effet, nous allons ici procéder de manière incrémentale afin de montrer le cheminement qui nous a conduit à concevoir la stratégie multirail telle qu'elle est aujourd'hui.

L'ordonnanceur de NEWMADELEINE étant piloté par l'activité des cartes réseau, il a dans un premier temps été facile de développer une stratégie capable de distribuer les paquets à transmettre de manière *gloutonne* : dès qu'une carte réseau devient libre, on l'alimente tout simplement avec le prochain paquet à envoyer. Lorsque deux cartes sont disponibles et qu'un seul paquet est prêt à être envoyé, ce dernier peut être coupé en deux pour alimenter les deux cartes simultanément.

En suivant ce principe, nous avons réalisé une première implémentation naïve de cette stratégie, et l'avons évaluée au-dessus de notre plate-forme bi-rails équipée des réseaux MYRI-10G et QSNET II. Les premières évaluations de ce prototype, reportées sur les Figures 7.14 et 7.15 ont toutefois montré qu'il est nécessaire de raffiner la stratégie pour les différentes catégories de messages :

Messages de taille $\leq$ 32 Ko. La Figure 7.14 montre les performances obtenues lorsque l'on distribue de manière gloutonne deux segments de taille égale sur notre plate-forme multirail. En comparaison, nous avons reporté le temps de transfert obtenu lorsque l'on agrège ces deux segments et que l'on envoie le segment résultant en mode monorail. Les résultats montrent qu'il n'y a aucun intérêt à utiliser le multirail pour des segments dont la taille cumulée est inférieure à 32 Ko. La cause tient à la façon dont les messages sont envoyés de la mémoire centrale vers la carte réseau du côté émetteur : en effet, cela nécessite une copie qui s'avère bloquante, monopolisant donc le processeur et qui dégrade la bande

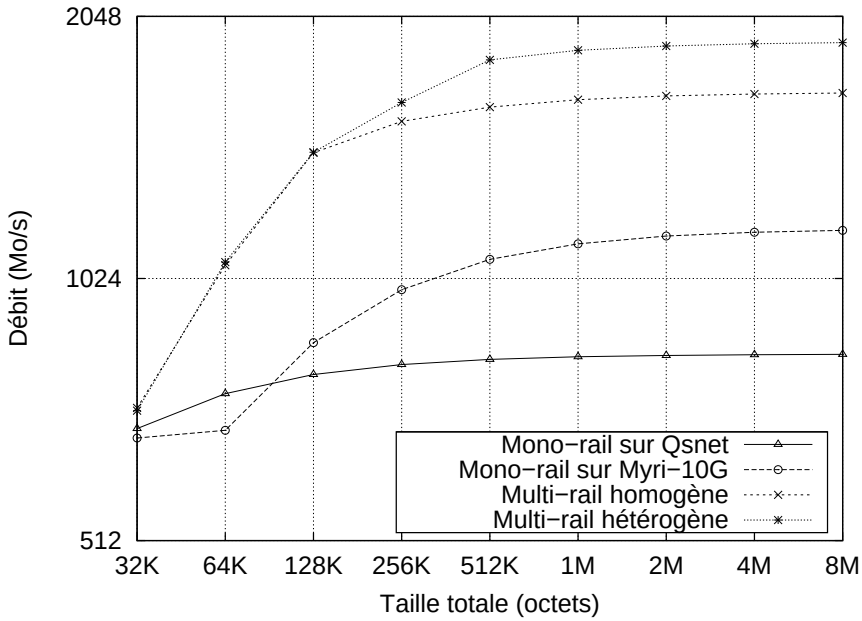


FIG. 7.15: *Messages fractionnés suivant un ratio adapté - Débit*

passante disponible sur le bus mémoire durant l'opération. Ainsi, l'émission des deux segments est quasiment sérialisée.

Messages de taille $>$ 32 Ko. La Figure 7.15 montre les performances obtenues lorsque l'on coupe un segment en deux fragments de même taille (courbe *Multirail homogène*) pour les envoyer sur deux réseaux différents. En comparaison, nous avons reporté le temps de transfert obtenu en transmettant le segment intact sur un seul des deux réseaux (courbes *Monorail*). Les résultats montrent que l'on obtient un gain important en termes de débit : pour un message de 8 Mo, le débit obtenu s'élève à 1670 Mo/s en multirail, alors qu'il ne dépasse pas 1170 Mo/s sur MYRI-10G et 850 Mo/s sur QSNET II. Toutefois, ce gain est inférieur à ce que l'on pourrait espérer en utilisant les deux réseaux au maximum de leur capacité.

Nous avons donc modifié notre stratégie afin d'agréger les segments de taille inférieure à 32 Ko pour les transmettre sur le réseau le plus rapide d'une part, et de scinder ceux de taille supérieure à 32 Ko en respectant un ratio correspondant au débit des cartes réseau pour l'intervalle de taille considéré. Ce dernier point garantit que les cartes réseau *travaillent* pendant la même durée pour envoyer le fragment qui leur a été affecté, et donc que le temps de transmission est minimal.

La Figure 7.14 montre que notre nouvelle stratégie (courbe *Multirail sur* QSNET) est bien meilleure que la version précédente. Le faible écart avec la courbe *Monorail sur* QSNET s'explique par le simple coût, dans le cas multirail, de la scrutation inutile mais systématique de tous les réseaux du côté du récepteur.

Enfin, la Figure 7.15 montre que l'utilisation d'une tactique plus élaborée pour la gestion des segments de taille importante obtient d'excellents résultats (courbe *Multirail hétérogène*) : le débit obtenu atteint 1987 Mo/s, ce qui est très proche de la limite théorique accessible au niveau matériel qui s'élève à $<$ 2 Go/s.

En plus de montrer qu'il est possible d'exploiter efficacement plusieurs cartes réseau à la fois, cette

démarche nous amène à souligner le fait que la plate-forme NEWMADELEINE facilite grandement ce genre de construction incrémentale de stratégie, sans jamais nécessiter de développer du code dépendant des technologies, tout en garantissant un surcoût logiciel moindre par rapport aux pilotes sous-jacents.

Déport sur des processeurs libres grâce à PIOMAN Au même titre que nous déportions la soumission des requêtes d'émission sur un processeur libre dans la partie 7.1.1.2, nous voudrions pouvoir employer le même type de technique pour plusieurs paquets issus de la fragmentation d'un message par la présente stratégie.

Cependant, le nombre de processeurs devant se trouver dans un état vacant à un instant donné n'est pas prédictible. La décision de pratiquer le déport de ces envois ne peut donc n'avoir lieu qu'au tout dernier moment, *i.e.* au moment même où la communication devrait être soumise au réseau. Ainsi, cette technique n'est proposée que pour des transferts de taille ne nécessitant pas de rendez-vous, puisqu'ils exigent une description du déroulement de la communication effective bien trop en avance. De ce fait, au moment de l'invocation de l'optimiseur de NEWMADELEINE en vue d'obtenir une nouvelle communication, PIOMAN consulte l'état des processeurs. Si un aucun n'est libre, la communication est soumise normalement, si un seul l'est elle est déportée sur ce dernier et finalement, si plusieurs le sont, la stratégie est invoquée afin de déterminer s'il est intéressant de subdiviser le paquet au départ. Les proportions de chaque fragment sont également indiquées après avoir été définies grâce à l'échantillonnage. Les potentiels fragments ainsi formés sont alors chacun déportés sur des processeurs libres différents pour émission immédiate.

Pour des raisons de synchronisations entre les processeurs encore mal maîtrisées, cette technique n'a pas encore montré ses capacités réelles : elle reste pour l'instant trop coûteuse. Elle a néanmoins fait l'objet d'un papier [BTD08] dans une session *travail en cours*, ce qui nous pousse à tout de même la présenter dans ce document.

7.2 MPICH2/NEWMADELEINE

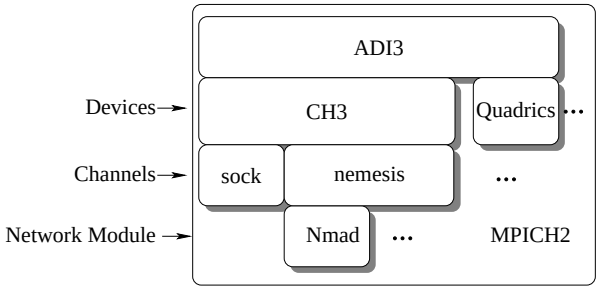


FIG. 7.16: *Pile logicielle de MPICH2.*

Une des implémentations libres de MPI les plus utilisées actuellement est MPICH [GLDS96], réalisée au sein de l'Argonne National Laboratory. Tandis que MPICH suit la première version du standard MPI, sa seconde version MPICH2 apporte la compatibilité avec la norme MPI-2.

L'architecture de MPICH2 s'organise en différentes couches comme le montre la Figure 7.16. ADI3 (*Abstract Device Interface*) et CH3 (*CHannel*) sont deux briques qui implémentent un certain nombre

de fonctionnalités de MPI telles que les algorithmes nécessaires aux opérations collectives. Typiquement, les constructeurs de technologies réseaux se placent au niveau de la couche *device* ce qui leur permet de câbler le plus étroitement possible leurs appels au matériel avec les primitives de MPI. De son côté, CH3 compte plusieurs *channels* comme *shm* (*shared memory channel*), *ssm* (*socket and shared memory channel*), *sshm* (*scalable shared memory channel*) qui se focalisent sur les communications en mémoire partagée ainsi que *Nemesis*. NEMESIS est le *channel* officiel de la distribution actuelle de MPICH2. Il est avant tout destiné aux communications en mémoire partagée pour lesquelles il offre des performances de tout premier plan mais propose également un support aux communications inter-nœud sur réseaux rapides. Il adopte le même mode de fonctionnement que les communications soient locales ou distantes. Ce choix est certes très abouti et élégant au niveau architectural mais n'atteint cependant pas les performances escomptées pour les communications externes. NEWMADELEINE se positionne donc naturellement comme complémentaire de NEMESIS afin de lui fournir un support pour les communications inter-nœud plus efficace. Pour cela, un pilote de MPICH2-Nemesis s'appuyant sur NEWMADELEINE a été développé.

Cette intégration a cependant montré que certains aspects de l'architecture de NEMESIS dupliquaient ou entravaient le mode de fonctionnement de NEWMADELEINE. De plus, il a été proposé d'utiliser également PIOMAN comme gestionnaire d'entrées/sorties pour les communications en mémoire partagée afin d'augmenter leur réactivité et d'assurer leur progression en tâche de fond. Ainsi, l'ensemble de ces travaux ont donné lieu à une collaboration entre Runtime et Argonne National Laboratory qui a débutée au début de l'année 2008.

Nous présentons par la suite l'ensemble des résultats que nous avons réalisé afin de montrer l'apport de NEWMADELEINE à MPICH2-NEMESIS.

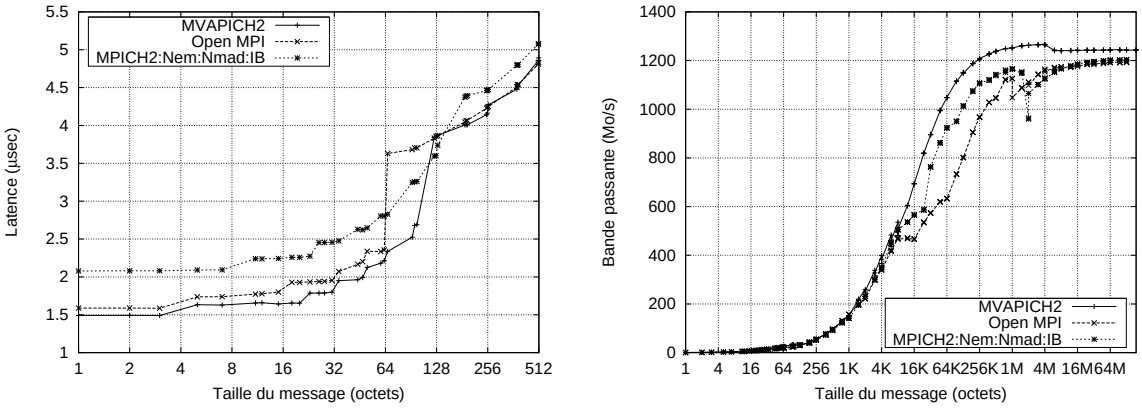
7.2.1 Performances brutes

Pour commencer, nous nous intéressons aux performances brutes obtenues à l'aide du programme de test NETPIPE [SMG96]. Nous comparons les résultats de MPICH2/NEWMADELEINE avec ceux des implémentations les plus répandues de MPI. Ces tests ont été réalisés sur une grappe de QuadCore INTEL XEON cadencés à 3,16GHz équipés d'une carte MYRI-10G ainsi que d'une carte INFINIBAND de type CONNECTX.

La Figure 7.17 présente les performances de différentes implémentations de MPI sur le réseau INFINIBAND. Nous comparons MPICH2/NEWMADELEINE avec MVAPICH2 1.0.3 et OPEN MPI 1.2.7 utilisant le BTL *openib*. MVAPICH2 et OPEN MPI atteignent des temps de transfert similaires, très proches de ceux du pilote de communication bas niveau. MPICH2/NEWMADELEINE affiche quant à elle un surcoût de l'ordre de 300 nanosecondes ce qui correspond à ce que nous avions annoncé lors des premières évaluations de NEWMADELEINE à la Section 7.1.1.1.

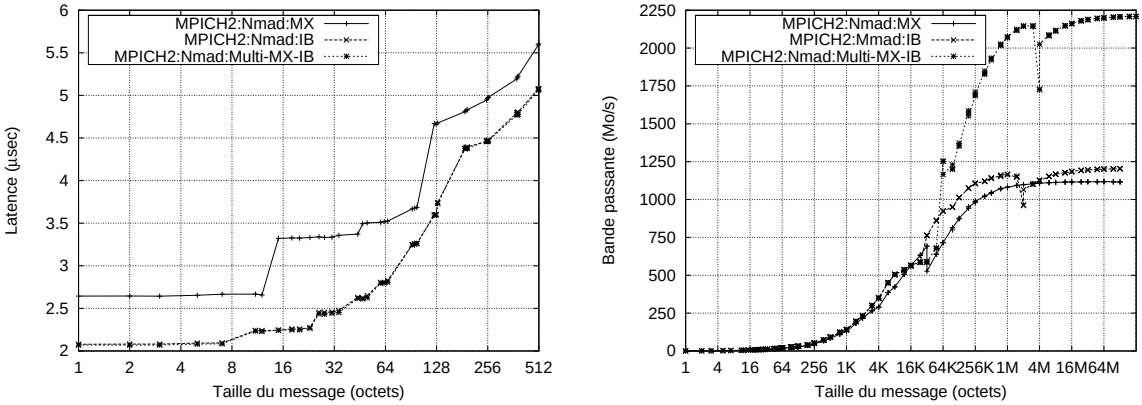
Du point de vue de la bande passante, MVAPICH2 est incontestablement meilleur que toutes les autres implémentations. Tandis qu'OPEN MPI atteint de bonnes performances en terme de latence, nous pouvons noter que cela est moins le cas en ce qui concerne le débit. MPICH2/NEWMADELEINE s'avère même meilleur pour le traitement de messages de taille intermédiaire.

Nous avons également validé le support multirail de NEWMADELEINE au sein de MPICH2, fonctionnalité nouvelle pour ce dernier. Les résultats de la Figure 7.18 confirment ceux que nous avions observés dans la Section 7.1.3.



(a) Latence (b) Bande passante

FIG. 7.17: *Performance de MPICH2-Nemesis/*NEWMADELEINE *au-dessus d'*INFINIBAND.



(a) Latence (b) Bande passante

FIG. 7.18: *Performance de MPICH2/*NEWMADELEINE *en multirail au-dessus d'*INFINIBAND *et de* MYRI-10G.

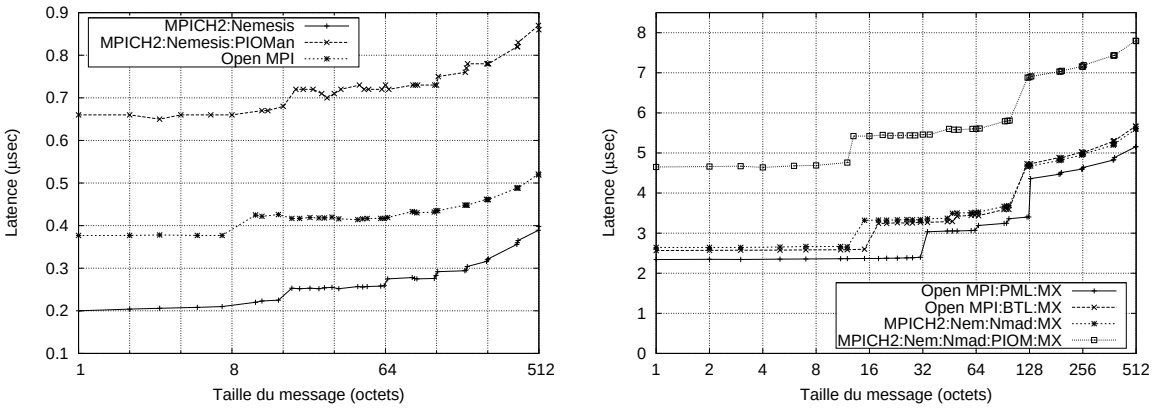
7.2.2 Progression des communications dans MPICH2/NEWMADELEINE

Dans sa version actuelle, MPICH2-NEMESIS ne dispose pas d'un mécanisme faisant progresser ses communications en tâche de fond. Ainsi, l'emploi de PIOMAN en vue de faire progresser les communications vers des nœuds distants dans NEWMADELEINE aussi bien qu'au sein même de NEMESIS pour les communications en mémoire partagée constitue une amélioration substantielle.

Son utilisation pour les communications devant transiter sur le réseau a été implantée de manière quasiment transparente via l'intégration de NEWMADELEINE. Seule la boucle de progression séquentielle de NEMESIS a été changée dans le cas de la détection d'une terminaison de communication bloquante (`MPI_Wait`). En effet, à l'origine, cette dernière s'active à interroger successivement le réseau puis la mémoire partagée tant qu'aucun évènement ne satisfait la requête de communication considérée. Dans le cas où NEWMADELEINE est utilisée, pour la partie s'intéressant aux communications externes, interroger NEWMADELEINE revient à consulter un champ indéfiniment et non à faire progresser à proprement

parler les communications. Ainsi, nous avons modifié la boucle de progression de NEMESIS afin que la détection d'évènement sur le réseau ne soit plus faite via une scrutation, mais par une attente sur un sémaphore afin de libérer un maximum de ressources en temps processeur et de permettre à PIOMAN de procéder effectivement à la progression de échanges de données en cours. Ceci entraîne également la modification du mode de détection d'évènement se produisant en mémoire partagée. En effet, autant l'endormissement sur un sémaphore attendant un évènement peut s'avérer convenable lorsque la provenance de la communication est connue et distante, autant cela ne peut pas fonctionner dans le cas d'une communication de source anonyme (*i.e.* lorsqu'il s'agit d'une communication `any_src`). La progression des communications provenant de la mémoire partagée a de ce fait été complètement déléguée à PIOMAN. Cette étape a demandé des modifications au niveau de l'implémentation de NEMESIS. Ainsi, lorsque NEMESIS attend un message provenant d'un processus local, il notifie PIOMAN et lui fournit une adresse contenant un champ qui est incrémenté lorsque la communication a effectivement eu lieu. Avec ce système de boîte aux lettres, PIOMAN n'a plus qu'à aller vérifier le statut de ce champ pour vérifier l'état d'avancement de la communication courante. Si celle-ci est achevée, le sémaphore introduit dans la boucle de progression est relâché et NEMESIS reprend son activité normale.

Les premiers résultats présentés à la Figure 7.19 montrent des surcoûts importants, que ce soit pour des communications distantes ou locales sur un simple test de type ping-pong. Ces derniers sont principale-



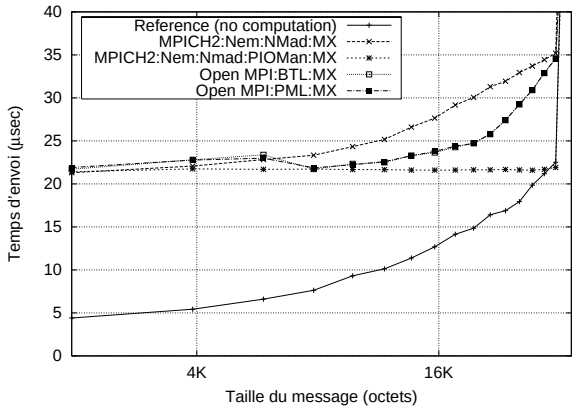
(a) Latence en mémoire partagée

(b) Latence sur MX/Myri-10G

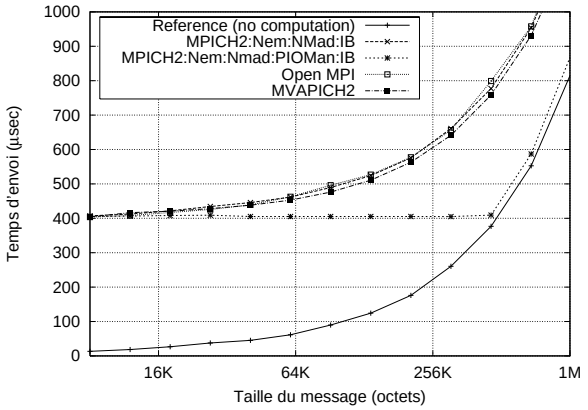
FIG. 7.19: *Performance en latence de MPICH2/*NEWMADELEINE *avec* PIOMAN *faisant progresser l'ensemble des communications.*

ment dûs à la nécessité de protéger l'accès à certaines structures (*i.e.* listes des requêtes en cours, etc.) qui n'était pas nécessaire à NEMESIS dont le comportement était séquentiel jusqu'à présent. Nous notons cependant que ces surcoûts se révèlent être constants quelque soit la taille des messages traités et la réactivité dans le cas de messages longs est donc d'ores et déjà améliorée.

Ce premier test nous permet de valider ce nouveau mode de fonctionnement mais ne met pas réellement en avant l'avancée apportée à MPICH2-NEMESIS. En effet, le but de ce mécanisme est de faire progresser les communications en tâche de fond. Les Figures 7.20(a) et 7.20(b) montrent les résultats obtenus sur le même type de tests que nous avions menés dans la Section 7.1.1.2. Avec l'aide de PIOMAN, MPICH2-NEMESIS se révèle à présent capable de recouvrir ses communications par du calcul.



(a) Recouvrement des communications échangées par copie sur MX/Myri-10G.



(b) Progression en tâche de fond des échanges via une prise de rendez-vous préalable sur INFINIBAND

7.2.3 Bilan

L'association de MPICH2-NEMESIS avec NEWMADELEINE et PIOMAN est encore un travail en cours de réalisation. Il a déjà montré de bonnes avancées puisque MPICH2-NEMESIS profite déjà d'un support aux communications multirails et à la progression des communication en tâche de fond, propriétés dont il n'était pas muni jusqu'à présent et qu'il se montrait délicat à faire directement au sein de l'architecture originale.

Par ailleurs, plusieurs tests applicatifs ont été menés afin de mettre à l'épreuve la robustesse de cette nouvelle association. L'ensemble de la suite des NAS et d'IMB s'exécutent avec des performances similaires à la version initiale tant que les types dérivés ne sont pas utilisés. En effet, cette partie n'est pas encore traitée, elle requiert encore des travaux de *branchement* entre NEMESIS et NEWMADELEINE afin d'être opérationnelle. Plus conséquent, nous avons aussi employé des applications telles SWEEP3D [swe] et WRF [WRF] qui respectivement modélisent le transport de neutrons et des données météorologiques, s'exécutent sur des temps plus longs. Malheureusement, ces applications ne génèrent pas suffisamment de communication pour que l'on observe des gains significatifs. Des performances équivalentes à celles obtenues avec des implémentations de MPI usuelles en sont ressorties.

7.3 PASTIX

La résolution de très grands systèmes linéaires creux [HNP91, Saa96] est une brique de base algorithmique fondamentale pour les applications scientifiques de calcul intensif. Cette étape est souvent, dans un environnement de simulation numérique, la plus consommatrice aussi bien en temps CPU qu'en espace mémoire. La taille des systèmes utilisés pour les grandes applications entraîne que le calcul haute performance est de fait incontournable. En effet, les solveurs – *i.e.* les outils qui servent à la résolution – sont amenés à traiter des problèmes qui atteignent typiquement plusieurs dizaines voire plusieurs centaines de millions d'inconnues.

Devant cette masse de données, il est nécessaire d'utiliser des outils parallèles qui permettent à la fois de distribuer les données sur différentes machines et de disposer d'une puissance de calcul plus importante. Pour cela, sur des plates-formes à base de nœuds à mémoire partagée, il est important d'employer des

solutions multithreadées de manière à réduire le volume des données échangées.

Dans cette partie, nous nous intéressons au solveur PASTIX [HRR02] développé au sein de l'équipe SCALAPPLIX. Celui-ci permet la résolution parallèle de systèmes linéaires creux via une factorisation de Cholesky au travers de solutions hybrides basées sur le passage de messages via MPI et l'exploitation de la mémoire partagée à l'aide de *threads*.

7.3.1 Adaptation du modèle de communication au support de concurrence offert

Afin d'exploiter au mieux les différents niveaux de support du *multithreading* dans les implémentations de MPI, PASTIX propose plusieurs versions de son solveur qui se différencient par leur mode de communication. Le premier s'adapte à un environnement supportant la concurrence de fil d'exécution de niveau MPI_THREAD_FUNNELED. Ce mode requiert que seul le *thread* principal d'une application multithreadée ne fasse appel à la bibliothèque de communication MPI (se référer à la partie 3.1.2 pour plus de détails). Comme le montre la Figure 7.20(a), cette version dédie donc un *thread* aux échanges de données, que ce soit en émission ou en réception. Les deux autres implémentations nécessitent le niveau de *multithreading* le plus élevé, *i.e.* le niveau MPI_THREAD_MULTIPLE. Elles se distinguent l'une de l'autre par l'ajout d'un *thread* chargé de la progression en réception des données au sein même du solveur dans le cas où l'implémentation de MPI utilisée ne le ferait pas en interne. Ainsi, comme on peut le voir dans la Figure 7.20(b), le modèle nommé THCOMM utilise un *thread* afin d'assurer l'ensemble des réceptions de données de toute l'application. Ce dernier s'occupe à la fois de leur dépôt et de leur progression en arrière plan. La seconde quant à elle laisse aux *threads* de calcul la charge de l'ensemble de leurs communications *i.e.* en émission et en réception, comme nous pouvons le voir sur la Figure 7.20(c).

Comme nous l'avons déjà exposé dans les parties 3.2.2, 4.3 et 5.3.2, il ne peut être que profitable à PASTIX de pouvoir déléguer à MPI la régulation des accès concurrents et la progression des transferts en tâche de fond. Il est donc tout naturel que nous ayons entrepris de porter PASTIX au dessus de NEWMADELEINE.

7.3.2 Délégation des communications à NEWMADELEINE

La factorisation de systèmes matriciels entraîne de nombreuses contributions d'un bloc colonne à un autre. Ces blocs colonnes étant distribués sur l'ensemble de nœuds utilisés, un grand nombre de communications est généré. Pour réduire ce nombre, PASTIX utilise deux techniques. Afin de les illustrer, nous nous reportons à la Figure 7.21. La première, mathématique, consiste à sommer l'ensemble de contributions pour un même bloc dans un même tampon. Ces opérations sont symbolisées par les fines flèches pleines. La seconde, algorithmique, rassemble l'ensemble des blocs de contribution à destination d'un même bloc colonne afin de les transmettre en une seule communication. Ainsi, le nombre d'échanges se voit réduit et la bande passante mieux exploitée. Ceci correspond aux blocs hachurés dont l'envoi est modélisé par les flèches pleines plus épaisses.

NEWMADELEINE intervient uniquement sur la technique algorithmique. En effet, il ne s'agit pas de modifier le comportement de l'application, mais seulement de libérer PASTIX des optimisations qui ont été réalisées dans l'optique d'atteindre de meilleures performances à partir des caractéristiques du réseau sous-jacent. Ainsi, nous passons du schéma de communication décrit par la Figure 7.21 à celui décrit à la Figure 7.22. Dans cette nouvelle version, les contributions pour un même bloc sont donc

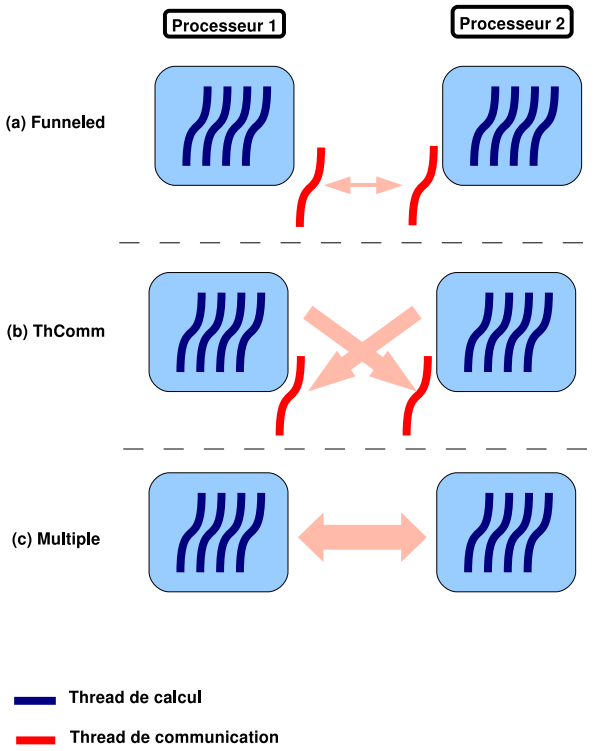


FIG. 7.20: *Différents mode de communication de* PASTIX.

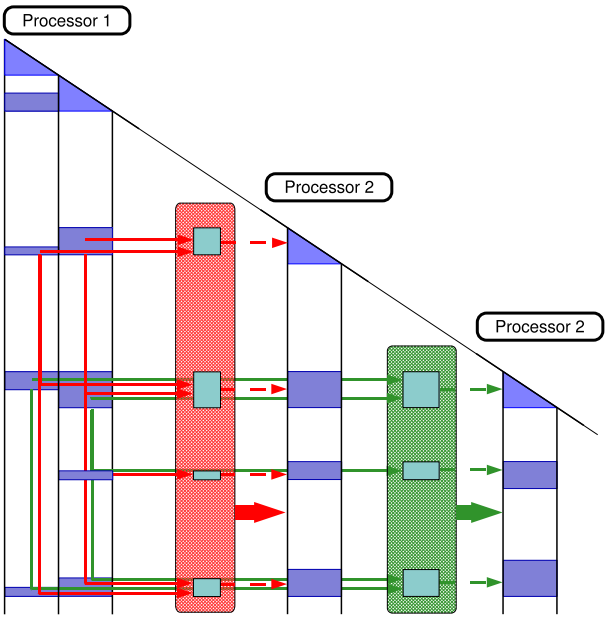


FIG. 7.21: *Schéma de communication de* PASTIX.

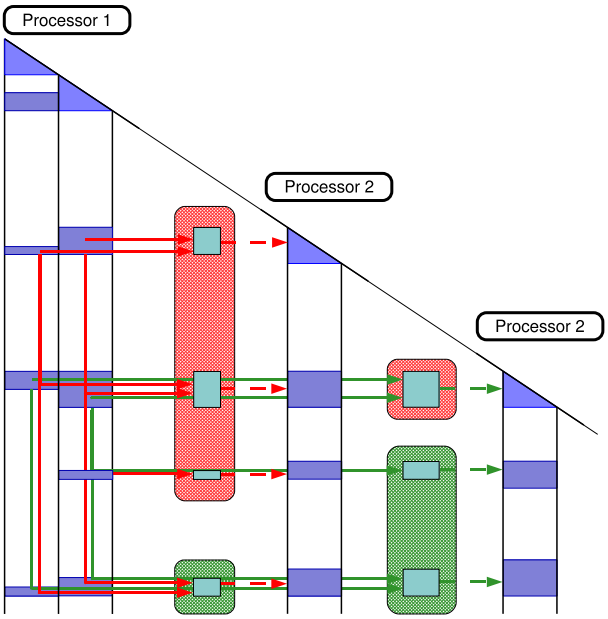


FIG. 7.22: *Schéma de communication de* PASTIX *par* NEWMADELEINE.

toujours rassemblées dans un même tampon mais l'ensemble des blocs ayant pour cible le même bloc colonne n'est plus regroupé en une seule communication. À présent, chaque bloc de contributions est soumis tel quel à NEWMADELEINE au travers de l'interface MAD-MPI et va se charger de faire les optimisations adéquates. En l'occurrence, nous utilisons ici la stratégie consistant à agréger au maximum les données. Ainsi, les blocs de données ne sont plus obligatoirement groupés selon le bloc colonne cible mais seulement selon la machine sur laquelle le bloc colonne cible se situe.

Nous présentons ici les résultats obtenus avec la version `funneled` de PASTIX. Un seul *thread* est chargé de procéder à l'ensemble des opérations de communication, qu'elles soient d'émission ou de réception. La version `multiple` où chaque *thread* de calcul se charge de ses propres communications devrait être opérationnelle d'ici peu.

7.3.3 De 500.000 à 1.000.000 inconnues

Deux matrices différentes ont été employées afin de mener les expériences de cette partie. La première, la matrice MHD, correspond à un problème 3D non-symétrique de Magneto-Hydro-Dynamique. Elle comporte 485 597 inconnues, 24 233 141 termes non-nuls dans la matrice non factorisée et 1.629e+09, dans la matrice factorisée. La seconde, la matrice AUDI, décrit quant à elle un problème symétrique de mécanique des structures issue de la collection PARASOL [par]. Elle compte 943 695 inconnues, 39 297 771 termes non-nuls dans la matrice non factorisée et 1.214e+09, dans la matrice factorisée.

Ces tests ont été réalisés sur la plate-forme d'évaluation BORDERLINE de GRID5000. Chaque nœud de cette grappe comporte quatre processeurs dual-core AMD Opteron cadencé à 2.6 GHz et est munie d'une carte de technologie MYRI-10G.

Dans les deux cas présentés dans les Tables 7.3 et 7.4, nous observons des performances équivalentes, ce qui valide nos attentes. En effet, nous permettons au développeur de PASTIX de s'affranchir des op-

nombre de threads par nœud	agrégation faite par	2 nœuds	4 nœuds	8 nœuds
1	PASTIX	1520.00s	758.00s	404.00s
	NEWMADELEINE	1530.00s	760.00s	411.00s
2	PASTIX	803.00s	413.00s	208.00s
	NEWMADELEINE	797.00s	409.00s	207.00s
4	PASTIX	422.00s	212.00s	119.00s
	NEWMADELEINE	423.00s	214.00s	120.00s
8	PASTIX	234.00s	128.00s	77.30s
	NEWMADELEINE	232.00s	127.00s	76.60s

TAB. 7.3: *Matrice MHD*

nombre de threads par nœud	agrégation faite par	2 nœuds	4 nœuds	8 nœuds
1	PASTIX	880.00s	418.00s	212.00s
	NEWMADELEINE	880.00s	417.00s	212.00s
2	PASTIX	450.00s	224.00s	111.00s
	NEWMADELEINE	448.00s	225.00s	112.00s
4	PASTIX	230.00s	114.00s	64.20s
	NEWMADELEINE	230.00s	115.00s	62.70s
8	PASTIX	127.00s	66.50s	-
	NEWMADELEINE	127.00s	65.40s	-

TAB. 7.4: *Matrice Audi*

timisations qu'il avait auparavant effectuées *à la main* afin d'améliorer son schéma de communication. Avec NEWMADELEINE, il accède d'ores et déjà de manière **transparente** à une stratégie d'ordonnancement qui atteint le même niveau de résultat. Nous lui ouvrons donc l'opportunité d'employer d'autres sortes d'optimisation qui pourraient se révéler encore plus performantes et plus en adéquation avec la configuration de la plate-forme d'expérimentation.

Chapitre 8

Conclusion et Perspectives

Sommaire

Les constructeurs de composants informatiques ne semblent jamais à bout d'argument dans leur quête de technologies toujours plus avancées. Ainsi, les supports d'aide à l'exécution d'applications sont sans cesse amenés à se mouvoir vers des solutions qui sachent tirer parti de ces évolutions afin de les exploiter au mieux. Dans le cas plus précis des grappes de machines, une attention particulière doit bien sûr être portée aux aspects communications, paramètre nécessaire à la bonne marche des applications de calcul intensif dans un tel environnement, si tant est que leurs performances soient dictées par celles du réseau sous-jacent.

Actuellement, la tendance est de construire des architectures dont le nombre d'unités de calcul croît massivement. Pour l'instant, les machines les plus imposantes sont munies de quelques dizaines de processeurs mais les constructeurs annoncent déjà des machines dotées de plusieurs centaines, voire milliers de cœurs d'ici peu. Le déséquilibre qui existe entre le nombre de ressources de calcul et celles de communication accentue notablement les défauts des bibliothèques de communication utilisées à ce jour. En effet, le modèle courant où un processus est assigné à un processeur et surtout, partageant avec un nombre très restreint l'accès aux ressources réseau du nœud crée une forte contention au niveau des cartes réseau, en plus de ne pas les exploiter efficacement. Par ailleurs, des outils aidant à la parallélisation des programmes pré-existants comme OpenMP ou TBB étendent l'exécution d'un processus en mode séquentiel sur un seul processeur à une exécution sur l'ensemble de la machine grâce à l'emploi de processus légers. Chaque processeur devient alors une source potentielle de communications simultanées.

Les implémentations de MPI actuelles ne tiennent pas compte de cet état de fait. Elles persistent vers des architectures qui soumettent directement les requêtes de transferts aux cartes réseau afin de maintenir des performances exceptionnelles dans le cadre de tests basiques tels des ping-pongs. L'augmentation de la masse de données devant transiter par un nombre restreint de ressources de communication doit néanmoins être canalisée afin de ne plus les saturer et d'obtenir une amélioration de l'ensemble du schéma de communication d'une application.

8.1 Contribution

Les travaux de cette thèse proposent de penser différemment la manière d'effectuer les transferts de données à un niveau bas. Nous faisons le compromis de sacrifier légèrement leurs performances brutes au profit de l'amélioration des communications dans leur globalité. Pour cela, contrairement aux architectures des interfaces de communications couramment rencontrées, nous découplons la soumission de requêtes de transferts faites par l'application de celles faites au réseau lui-même. Ceci, en plus de désengorger les cartes réseau, a pour effet de nous permettre de prendre du recul sur l'ensemble des communications à traiter à un instant donné et ainsi d'avoir l'opportunité de leur appliquer des stratégies d'ordonnancement.

Ce concept a été implémenté au sein de la bibliothèque de communication pour réseaux haute performance NEWMADELEINE. Celle-ci expose une interface utilisateur assez basique par échange de messages destinée aux interfaces de communications de niveau supérieur telles les implémentations de MPI. En effet, un certain nombre de fonctionnalités couramment employées par les applications de calcul intensif ne sont pas supportées comme les opérations effectuées en mémoire partagée ou encore les opérations collectives ou de synchronisation. Ce type de fonctionnalités sont pour la plupart extrêmement bien réalisées par les implémentations les plus performantes de MPI. Ainsi, la cible de NEWMADELEINE est de fournir un support aux communications externes aux interfaces de communication de niveau supérieur. Le développeur de niveau applicatif peut néanmoins interagir avec NEWMADELEINE en implémentant une stratégie d'ordonnancement de communication en adéquation avec le comportement de son application. En effet, NEWMADELEINE a été conçue de manière à pouvoir abstraire et interchanger facilement la stratégie à appliquer aux communications. Cette dernière ne requiert aucune connaissance technique liée à un réseau sous-jacent, elle se résume à de la manipulation de paquets et à la consultation des propriétés générales des cartes de communication disponibles afin de produire, de son point de vue, le meilleur agencement possible. Plusieurs stratégies ont déjà été implémentées qui proposent notamment d'agréger le maximum de données en vue de réduire le nombre d'échanges ou encore d'exploiter simultanément différentes cartes réseau qui ne sont pas obligatoirement de même technologie.

Par ailleurs, NEWMADELEINE a été conçue en collaboration avec PIOMAN, un gestionnaire d'entrées/sorties qui lui fournit un support efficace à l'ensemble de ses attentes en matière de réactivité et de recouvrement de communication. Celui-ci, en plus de s'occuper de la progression des communications, sait tirer parti des nouvelles architectures multicœurs en se déployant sur le maximum de processeurs disponibles. Un processeur libre est rapidement réquisitionné afin de déporter une opération indépendante de l'application comme la soumission de communication aux cartes réseau ou encore la détection de leur terminaison. L'implémentation actuelle sur architectures multicœurs nécessite encore quelques optimisations, mais a d'ores et déjà montré un fort intérêt dans un contexte multithreadé.

Finalement, des tests synthétiques permettent de vérifier que les mécanismes introduits par le mode de fonctionnement propre à NEWMADELEINE n'induisent pas des surcoûts logiciels excessifs. De plus, ils mettent en avant les gains de stratégies d'ordonnancement dans le cas de schémas de communication basiques, ce qui laisse à penser qu'elles ne peuvent qu'être bénéfiques à toute application ayant un besoin en communication. D'autre part, des mesures de performance ont été faites au niveau d'applications réelles. Ceci a été rendu possible par le portage de MPICH2 au-dessus de NEWMADELEINE en tant que pilote réseau et à l'implémentation d'une couche d'interfaçage de NEWMADELEINE appelée MADMPI qui fournit un sous-ensemble de l'interface MPI elle-même. Dans certain cas, nous avons pu ainsi montrer que NEWMADELEINE permettait d'affranchir le développeur applicatif de l'optimisation de

ses communications manuellement. L'application se contentant de déléguer ses échanges de données à NEWMADELEINE a obtenu des performances similaires à une version optimisée *à la main* pour un réseau cible particulier. Cela constitue une contribution importante aussi bien du point de vue de la portabilité de programmation, que de celui de la portabilité des performances.

8.2 Perspectives

Ces travaux ouvrent de nombreuses perspectives à court, moyen et long terme qui ont pour ambition d'aider à ordonnancer de manière toujours plus pointue les communications.

Plus de stratégies d'ordonnancement Pour commencer, une étude d'applications complexes mêlant de nombreux flux de communication doit être entamée afin de découvrir s'il est nécessaire de développer de nouvelles stratégies. Pour cela, la collaboration avec des équipes développant des applications cibles doit être privilégiée. Nous avons constaté dans le Chapitre 7 qu'il n'est pas aisé de trouver des applications qui ne soient trop limitées en communication, rendant inutile l'utilisation de NEWMADELEINE. En effet, NEWMADELEINE requiert des communications fréquentes et le maximum d'asynchronisme possible afin de montrer toute son efficacité. Une collaboration pourrait aider à construire de nouvelles stratégies mais également à sensibiliser les développeurs à ce type de paradigme de programmation.

Même si nous avons vu que l'implémentation de la stratégie *ultime* s'annonce ardue, elle pourrait faire l'objet de travaux communs avec des équipes de théoriciens afin de modéliser une façon de découvrir quelles stratégies d'ordonnancement peuvent être appliquées sur les communications et appliquer le meilleur compromis efficacement.

Les travaux autour de la qualité de service sont également à investiguer. En effet, les environnements à plus grande échelle que sont les grilles de calcul sont particulièrement concernés par cette problématique. Il serait donc intéressant de continuer dans cette voie afin d'appliquer les politiques d'ordonnancement qu'ils emploient de manière dynamique à l'échelle d'une grappe.

Poursuite des travaux sur le recouvrement des communications Même si ces travaux sont d'ores et déjà bien entamés, il reste encore du travail d'implémentation afin d'obtenir de meilleures performances. L'emploi de plusieurs cartes réseau depuis différents cœurs de la machine pour une même communication, l'intégration encore plus fine de ces mécanismes au sein de MPICH2-NEMESIS sont des points sur lesquels nous devons encore nous pencher. Cependant, au delà de cela, il s'agit de convaincre les différents acteurs de la sphère décidant de l'évolution MPI qu'il est important de continuer dans cette voie. Le nombre de cœurs par machine s'accroissant spectaculairement, il va donc être de plus en plus attractif de savoir déporter une partie des opérations vers une autre unité de calcul, qu'il s'agisse, par exemple, d'opération sur le réseau, de copies mémoires ou du calcul du meilleur ordonnancement des communications courantes.

Les communications par accès direct à la mémoire La prise en compte des communications par accès direct à la mémoire d'une machine distante est un aspect qui n'est pas encore pris en compte dans NEWMADELEINE. L'intégration de tels transferts, la découverte de leurs caractéristiques pour déterminer quand les employer doivent encore être réalisées. Des travaux préliminaires [Aug06] ont été

fait qui montrent que l'utilisation de transferts en RDMA sont particulièrement adaptés aux messages longs. En effet, leur transit et les temps nécessaires au punaisage de l'espace mémoire cible peuvent alors être complètement recouverts.

Extraire de l'information grâce à un pré-traitement Toute l'originalité de NEWMADELEINE réside dans l'application de stratégies d'ordonnancement qui s'adaptent à l'état des communications et du matériel à un instant donné. Contrairement au nombre de ressources de calcul et de communications qui n'est pas supposé fluctuer au cours d'une exécution, celui des communications n'est pas fixe. Leur masse dépend de l'application elle-même. Or, les opportunités d'optimisation de ces communications croit avec leur nombre. En effet, plus il y en a et plus il y a de combinaisons et de possibilités d'ordonnancement à réaliser. Ainsi, la récolte de toute information donnant des indications sur des communications à venir ou encore l'échéance à laquelle une communication doit être réalisée est importante. L'analyse du code de l'application au cours d'un pré-traitement ouvre cette perspective. Plus d'informations sont fournies à NEWMADELEINE et plus cette dernière est susceptible d'appliquer des méthodes d'optimisation efficaces.

Connaissance du comportement matériel plus fine La prise de décision de l'optimiseur s'appuie grandement sur les informations récoltées à l'initialisation de la plate-forme. Cela lui permet de prédire le comportement des cartes réseau face à la charge qu'il leur soumet et donc d'adapter la suite de l'exécution en conséquent. Ainsi, plus la collecte de ces données est fine et plus sa prédiction est juste. Un grand nombre de paramètres devrait être pris en compte en plus de ceux qui le sont actuellement. À partir de la topologie de la machine, il faudrait étudier les effets de la contention sur le bus mémoire des transferts de données jusqu'à la carte. Des travaux se sont déjà orientés sur ce type de thématique comme ceux de Maxime Martinasso [Mar07] et peuvent donc nous orienter vers le choix des paramètres à prendre en compte. Ceci viendrait s'ajouter aux travaux concernant le placement des *threads* de communication au plus proche des cartes réseau. Par ailleurs, les effets de la contention provoquée par des communications excessives au niveau des routeurs devraient également être considérés. Des travaux [TW03, HSL08] ont déjà mis en avant l'impact que peuvent avoir ces problèmes sur les échanges.

Extension de la portée des optimisations à plusieurs processus Comme nous le disions à la Section 2.2.2.2, l'exploitation des machines multicœurs se fait suivant deux modèles différents. L'un consiste à paralléliser les applications afin de générer assez de processus légers et de les déployer sur l'ensemble de la machine. Nous en avons particulièrement parlé étant donné que des travaux autour d'OPENMP sont en cours dans l'équipe. En collaboration avec l'ordonnanceur de *threads* BUBBLESCHED, des travaux [TBG+08] s'attachent à ordonnancer les processus légers générés grâce à une extension du compilateur GNU OPENMP de façon à préserver l'affinité qui les lie à leurs données. De plus, l'intégration de MPI avec OPENMP fait également partie des travaux de recherche menés. Ils cherchent à ajuster la granularité des modèles hybrides en adaptant le nombre de processus MPI à employer par nœud suivant sa topologie.

Cependant, la complexité de l'équilibrage de charge d'une application utilisant un modèle de programmation hybride et de la collaboration des *threads* et des communications fait que la majeure partie de la littérature lui préfère pour l'instant l'autre alternative qui consiste à associer un processus à une unité de calcul. En plus de problème de consommation mémoire, cette approche a le désavantage de multiplier les instances du support de communication. En effet, ces derniers sont en général liés de manière

statique aux programmes et chacun en démarre un exemplaire. Le mode de fonctionnement de NEWMADELEINE se base sur l'activité même des cartes réseau. Si plusieurs de ses instances venaient à se mettre en concurrence, cela provoquerait une incohérence dans leur façon de traiter les communications. Une perspective est donc de faire en sorte de mutualiser une seule instance de NEWMADELEINE pour l'ensemble de processus lancés. Plusieurs solutions peuvent être envisagées comme la création d'un module noyau ou le lancement d'une instance par la première application activée qui synchronisera ensuite les suivantes.

Influence sur la programmation parallèle Avec notre approche, nous offrons la possibilité de s'adapter aux architectures de demain que ce soit du point de vue des nouvelles technologies matérielles (processeurs, bus, etc.) ou encore de celui des paradigmes de programmation parallèles émergents (modèles hybrides). Au delà de cela, la diffusion de NEWMADELEINE via son intégration au sein des implémentations de MPI les plus répandues (MPICH2 et OPEN MPI) ouvre la perspective d'influencer l'évolution de MPI et plus généralement de tous les standards de la programmation parallèle.

Annexe A

Interfaces utilisateur de NEWMADELEINE

Sommaire

Cette annexe regroupe un extrait de la documentation des interfaces utilisateur offerte par NEWMADELEINE, extraite automatiquement du code source. La dernière version est disponible sur `http://pm2.gforge.inria.fr/newmadeleine/doc/html/`.

A.1 Interface par passage de message

A.1.1 Primitives relatives à l'émission de messages

– int **nm_sr_isend** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, const void *data, uint32_t len, nm_sr_request_t *p_request)
Post a non blocking send request.
– int **nm_sr_rsend** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, const void *data, uint32_t len, nm_sr_request_t *p_request)
Post a ready send request.
– int **nm_sr_isend_iov** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, const struct iovec *iov, int nb_entries, nm_sr_request_t *p_request)
Post a non blocking send request for non contiguous data set.

– int **nm_sr_stest** (nm_core_t p_core, nm_sr_request_t request) *Test for the completion of a non blocking send request.*
– int **nm_sr_stest_range** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, unsigned long seq_inf, unsigned long nb)
Test for the completion of a continuous series of non blocking send requests.
– int **nm_sr_swait** (nm_core_t p_core, nm_sr_request_t request)
Wait for the completion of a non blocking send request.
– int **nm_sr_swait_range** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, unsigned long seq_inf, unsigned long nb)
Wait for the completion of a continuous series of non blocking send requests.

A.1.2 Primitives relatives à la réception de messages

– int **nm_sr_irecv** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, void *data, uint32_t len, nm_sr_request_t *p_request)
Post a non blocking receive request.
– int **nm_sr_irecv_with_ref** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, void *data, uint32_t len, nm_sr_request_t *p_request, void *ref)
Post a non blocking receive request with a reference which will be given at the request completion.
– int **nm_sr_irecv_iov** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, struct iovec *iov, int nb_entries, nm_sr_request_t *p_request) *Post a non blocking receive request for non contiguous data set.*
– int **nm_sr_irecv_iov_with_ref** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, struct iovec *iov, int nb_entries, nm_sr_request_t *p_request, void *ref)
Post a non blocking receive request for non contiguous data set with a reference which will be given at the request completion.
– int **nm_sr_rtest** (nm_core_t p_core, nm_sr_request_t request)
Test for the completion of a non blocking receive request.
– int **nm_sr_rtest_range** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, unsigned long seq_inf, unsigned long nb)
Test for the completion of a continuous series of non blocking receive requests.
– int **nm_sr_rwait** (nm_core_t p_core, nm_sr_request_t request)
Wait for the completion of a non blocking receive request.
– int **nm_sr_rwait_range** (nm_core_t p_core, nm_gate_t p_gate, nm_tag_t tag, unsigned long seq_inf, unsigned long nb)
Wait for the completion of a continuous series of non blocking receive requests.
– int **nm_sr_probe** (nm_core_t p_core, nm_gate_t p_gate, nm_gate_t *p_out_gate, nm_tag_t tag)
Unblockingly check if a packet is available for extraction on the (gate,tag) pair.
– int **nm_sr_get_size** (nm_core_t p_core, nm_sr_request_t *request, nm_tag_t tag, int *size)
Returns the received size of the message with the specified request.

A.2 Interface par construction incrémentale de message

A.2.1 Primitives relatives à l'émission de messages

– int **nm_begin_packing** (nm_core_t p_core, nm_gate_t gate, nm_tag_t tag, nm_pack_cnx_t *cnx)
Start building and sending a new message.

– int **nm_pack** (nm_pack_cnx_t *cnx, const void *data, uint32_t len)
Append a data fragment to the current message.
– int **nm_end_packing** (nm_pack_cnx_t *cnx)
End building and flush the current message.
– int **nm_flush_packs** (nm_pack_cnx_t *cnx)
Wait for ongoing send requests to complete.
– int **nm_test_end_packing** (nm_pack_cnx_t *cnx)
Check ongoing send requests for completion.

A.2.2 Primitives relatives à la réception de messages

– int **nm_begin_unpacking** (nm_core_t p_core, nm_gate_t gate, nm_tag_t tag, nm_pack_cnx_t *cnx)
Start receiving and extracting a new message.
– int **nm_unpack** (nm_pack_cnx_t *cnx, void *data, uint32_t len)
Extract a data fragment from the current message.
– int **nm_end_unpacking** (nm_pack_cnx_t *cnx)
End receiving and flush extraction of the current message.
– int **nm_flush_unpacks** (nm_pack_cnx_t *cnx)
Wait for ongoing receive requests to complete.
– int **nm_test_end_unpacking** (nm_pack_cnx_t *cnx)
Check ongoing receive requests for completion.

A.3 Mad-MPI, un sous-ensemble de MPI

A.3.1 Communicateurs pré-définis et fonctions associées

Typedef et Defines
– typedef int **MPI_Comm**
Communicator handle.
– #define **MPI_COMM_WORLD** ((MPI_Comm)91)
Default communicator that includes all processes.
– #define **MPI_COMM_SELF** ((MPI_Comm)92)
Communicator that includes only the process itself.
– #define **MPI_COMM_NULL** ((MPI_Comm)0)
Invalide request handle.

Fonctions
– int **MPI_Comm_group** (MPI_Comm comm, MPI_Group *group)
Returns a handle to the group of the given communicator.
– int **MPI_Comm_split** (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
Partitions the group associated to the communicator into disjoint subgroups, one for each value of color.
– int **MPI_Comm_dup** (MPI_Comm comm, MPI_Comm *newcomm)
Creates a new intracommunicator with the same fixed attributes as the input intracommunicator.

– int **MPI_Comm_free** (MPI_Comm *comm)
Marks the communication object for deallocation.
– int **MPI_Group_translate_ranks** (MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)
Maps the rank of a set of processes in group1 to their rank in group2.

A.3.2 Communication point-à-point

– int **MPI_Send** (void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Performs a standard-mode, blocking send.
– int **MPI_Isend** (void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Posts a standard-mode, non blocking send.
– int **MPI_Rsend** (void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Performs a ready-mode, blocking send.
– int **MPI_Ssend** (void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Performs a synchronous-mode, blocking send.
– int **MPI_Esend** (void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Communication_Mode is_completed, MPI_Comm comm, MPI_Request *request)
This function does not belong to the MPI standard : Performs a extended send.
– int **MPI_Pack** (void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)
Packs a message specified by inbuf, incount, datatype, comm into the buffer space specified by outbuf and outsize.
– int **MPI_Recv** (void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
Performs a standard-mode, blocking receive.
– int **MPI_Irecv** (void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
Posts a nonblocking receive.
– int **MPI_Unpack** (void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize.
– int **MPI_Sendrecv** (void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
Executes a blocking send and receive operation.
– int **MPI_Wait** (MPI_Request *request, MPI_Status *status)
Returns when the operation identified by request is complete.
– int **MPI_Waitall** (int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)
Returns when all the operations identified by requests are complete.
– int **MPI_Waitany** (int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)
Blocks until one of the operations associated with the active requests in the array has completed.
– int **MPI_Test** (MPI_Request *request, int *flag, MPI_Status *status)
Returns flag = true if the operation identified by request is complete.
– int **MPI_Testany** (int count, MPI_Request *array_of_requests, int *index, int *flag, MPI_Status *sta-

tus)
Tests for completion of the communication operations associated with requests in the array.
- int **MPI_Iprobe** (int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
Nonblocking operation that returns flag = true if there is a message that can be received and that matches the message envelope specified by source, tag and comm.
- int **MPI_Probe** (int source, int tag, MPI_Comm comm, MPI_Status *status)
Blocks and returns only after a message that matches the message envelope specified by source, tag and comm can be received.
- int **MPI_Cancel** (MPI_Request *request)
Marks for cancellation a pending, nonblocking communication operation (send or receive).
- int **MPI_Request_free** (MPI_Request *request)
Marks the request object for deallocation and set request to MPI_REQUEST_NULL.
- int **MPI_Get_count** (MPI_Status *status, MPI_Datatype datatype, int *count)
Computes the number of entries received.
- int **MPI_Request_is_equal** (MPI_Request request1, MPI_Request request2)
This function does not belong to the MPI standard : compares two request handles.

A.3.3 Communications sur des datatypes

- int **MPI_Get_address** (void *location, MPI_Aint *address)
Returns the byte address of location.
- int **MPI_Address** (void *location, MPI_Aint *address)
Returns the byte address of location.
- int **MPI_Type_size** (MPI_Datatype datatype, int *size)
Returns the total size, in bytes, of the entries in the type signature associated with datatype.
- int **MPI_Type_get_extent** (MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)
Returns the lower bound and the extent of datatype.
- int **MPI_Type_extent** (MPI_Datatype datatype, MPI_Aint *extent)
Returns the extent of the datatype.
- int **MPI_Type_lb** (MPI_Datatype datatype, MPI_Aint *lb)
Returns the lower bound of the datatype.
- int **MPI_Type_create_resized** (MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype) *Returns in newtype a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to lb + extent.*
- int **MPI_Type_commit** (MPI_Datatype *datatype)
Commits the datatype.
- int **MPI_Type_free** (MPI_Datatype *datatype)
Marks the datatype object associated with datatype for deallocation.
- int **MPI_Type_optimized** (MPI_Datatype *datatype, int optimized)
This function does not belong to the MPI standard : Marks the datatype object associated with datatype as being optimized, i.e the pack interface can be used for communications requests using that type, instead of copying the data into a contiguous buffer.
- int **MPI_Type_contiguous** (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
Constructs a typemap consisting of the replication of a datatype into contiguous locations.
- int **MPI_Type_vector** (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
Constructs a typemap consisting of the replication of a datatype into location that consist of equally

spaced blocks.
- int **MPI_Type_hvector** (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
  Constructs a typemap consisting of the replication of a datatype into location that consist of equally spaced blocks, assumes that the stride between successive blocks is a multiple of the oldtype extent.
- int **MPI_Type_indexed** (int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)
  Constructs a typemap consisting of the replication of a datatype into a sequence of blocks, each block is a concatenation of the old datatype.
- int **MPI_Type_hindexed** (int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)
  Constructs a typemap consisting of the replication of a datatype into a sequence of blocks, each block is a concatenation of the old datatype ; block displacements are specified in bytes, rather than in multiples of the old datatype extent.
- int **MPI_Type_struct** (int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)
  Constructs a typemap consisting of the replication of different datatypes, with different block sizes.

A.3.4 Communications collectives

- **typedef** void **MPI_User_function** (void *, void *, int *, MPI_Datatype *)
  User combination function.
- int **MPI_Barrier** (MPI_Comm comm)
  Blocks the caller until all group members have called the routine.
- int **MPI_Bcast** (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
  Broadcasts a message from the process with rank root to all processes of the group, itself included.
- int **MPI_Gather** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
  Each process sends the contents of its send buffer to the root process.
- int **MPI_Gatherv** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)
  Extends the functionality of MPI_Gather() by allowing a varying count of data.
- int **MPI_Allgather** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
  Extends the functionality of MPI_Gather(), except all processes receive the result.
- int **MPI_Allgatherv** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)
  Extends the functionality of MPI_Gatherv(), except all processes receive the result.
- int **MPI_Scatter** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
  Inverse operation of MPI_Gather().
- int **MPI_Alltoall** (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvType, MPI_Comm comm)
  Extension of MPI_Allgather() to the case where each process sends distinct data to each of the receivers.
- int **MPI_Alltoallv** (void *sendbuf, int *sendcount, int *sdispls, MPI_Datatype sendtype, void *recvbuf, int *recvcount, int *recvdispls, MPI_Datatype recvType, MPI_Comm comm)

Adds flexibility to MPI_Alltoall() in that the location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls.

– int **MPI_Op_create** (MPI_User_function *function, int commute, MPI_Op *op)
Binds a user-defined global operation to an op handle that can subsequently used in a global reduction operation.
– int **MPI_Op_free** (MPI_Op *op)
Marks a user-defined reduction operation for deallocation.
– int **MPI_Reduce** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
Combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root.
– int **MPI_Allreduce** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
Combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of each process in the group.
– int **MPI_Reduce_scatter** (void *sendbuf, void *recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
First does an element-wise reduction on vector of elements in the send buffer defined by sendbuf, count and datatype.

A.3.5 Communications persistantes

– int **MPI_Send_init** (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.
– int **MPI_Recv_init** (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
Creates a persistent communication request for a receive operation.
– int **MPI_Start** (MPI_Request *request)
Initiates a persistent request.
– int **MPI_Startall** (int count, MPI_Request *array_of_requests)
Start all communications associated with requests in array_of_requests.

Bibliographie

[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.

[ADD+04] Aulwes (R. T.), Daniel (D. J.), Desai (N. N.), Graham (R. L.), Risinger (L. D.), Taylor (M. A.), Woodall (T. S.) et Sukalski (M. W.), « Architecture of LA-MPI, A Network-Fault-Tolerant MPI. », dans *Proc. 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, April 2004.

[Arg] Argonne National Laboratory, *Analysis of thread safety needs of MPI routines*. `http://www.mcs.anl.gov/mpi/mpich2/developer/design/threadlist.htm`.

[AT04] Arjuna (S.) et Tomasz (R.), « Improving the performance of tcp in the case of packet reordering », *HSNMC 2004 : High Speed Networks and Multimedia Communications*, 2004.

[Aug06] Augonnet (C.), « Optimisation des communications sur réseaux rapides par utilisation de RDMA », 2006.

[BA02] Blanton (E.) et Allman (M.), « On making tcp more robust to packet reordering », *SIGCOMM Comput. Commun. Rev.*, vol. 32, n° 1, 2002, p. 20–30.

[BBP+07] Balaji (P.), Bhagvat (S.), Panda (D. K.), Thakur (R.) et Gropp (W.), « Advanced flow-control mechanisms for the sockets direct protocol over infiniband », dans *ICPP '07 : Proceedings of the 2007 International Conference on Parallel Processing*, p. 73, Washington, DC, USA, 2007. IEEE Computer Society.

[BGST03] Byna (S.), Gropp (W.), Sun (X.-H.) et Thakur (R.), « Improving the performance of mpi derived datatypes by optimizing memory-access cost », *cluster*, vol. 00, 2003, p. 412.

[bla] *Basic Linear Algebra Subprograms*. `http://www.netlib.org/blas/`.

[BSTG06] Byna (S.), Sun (X.-H.), Thakur (R.) et Gropp (W.), « Automatic Memory Optimizations for Improving MPI Derived Datatype Performance », dans *EuroPVM/MPI*. Springer, 2006.

[BTD08] Brunet (E.), Trahay (F.) et Denis (A.), « A multicore-enabled multirail communication engine », dans *Cluster 2008*, Tsukuba, Japan, September 2008. IEEE.

[CC97] Chiola (G.) et Ciaccio (G.), « Gamma : a low cost network of workstations based on active messages », 1997.

[CFP+01] Coll (S.), Frachtenberg (E.), Petrini (F.), Hoisie (A.) et Gurvits (L.), « Using Multirail Networks in High-Performance Clusters », dans *Proceedings of the 3rd IEEE International Conference on Cluster Computing*, p. 15–24. IEEE Computer Society, 2001.

[CKP+96] Culler (D. E.), Karp (R. M.), Patterson (D.), Sahay (A.), Santos (E. E.), Schauser (K. E.), Subramonian (R.) et von Eicken (T.), « LogP : a practical model of parallel computation », *Commun. ACM*, vol. 39, n° 11, 1996, p. 78–85.

[com03] *Myrinet EXpress (MX) : A High Performance, Low-level, Message-Passing Interface for Myrinet*, 2003. `http://www.myri.com/scs/`.

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

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

[DRM+98] Dunning (D.), Regnier (G.), McAlpine (G.), Cameron (D.), Shubert (B.), Berry (F.), Merritt (A.), Gronke (E.) et Dodd (C.), « The virtual interface architecture », *Micro, IEEE*, vol. 18, n° 2, Mar/Apr 1998, p. 66–76.

[EBBV95] von Eicken (T.), 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.

[GLDS96] Gropp (W.), Lusk (E.), Doss (N.) et Skellum (A.), « A high performance, portable implementation of the MPI message passing interface standard ». Rapport technique n° ANL/MCS-TM-213, Argonne Nat. Lab., 1996.

[GLS] Gropp (W.), Lusk (E.) et Swider (D.), « Toward Faster Packing and Unpacking of MPI Datatypes ». `http://www.mcs.anl.gov/mpi/mpich1/papers/index.html`.

[Gog08a] 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.

[Gog08b] Goglin (B.), « Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX », dans *Proceedings of the IEEE International Conference on Cluster Computing*, Tsukuba, Japan, septembre 2008. IEEE Computer Society Press. To appear.

[GPL04] Gharai (L.), Perkins (C.) et Lehman (T.), « Packet reordering, high speed networks and transport protocol performance », *Computer Communications and Networks, 2004. ICCCN 2004. Proceedings. 13th International Conference on*, Oct. 2004, p. 73–78.

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

[GU01] Gilfeather (P.) et Underwood (T.), « Fragmentation and high performance ip », dans *IPDPS '01 : Proceedings of the 15th International Parallel & Distributed Processing Symposium*, p. 165, Washington, DC, USA, 2001. IEEE Computer Society.

[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.

[HNP91] Heath (M. T.), Ng (E. G.-Y.) et Peyton (B. W.), « Parallel algorithms for sparse linear systems », *SIAM Review*, vol. 33, 1991, p. 420–460.

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

[HRR02] Hénon (P.), Ramet (P.) et Roman (J.), « PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems », *Parallel Computing*, vol. 28, no 2, janvier 2002, p. 301–321.

[HSJ+06] Huang (W.), Santhanaraman (G.), Jin (H.-W.), Gao (Q.) et Panda (D. K.), « Design and Implementation of High Performance MVAPICH2 : MPI2 over InfiniBand », dans *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, Singapore, May 2006.

[HSJP05] Huang (W.), Santhanaraman (G.), Jin (H.-W.) et Panda (D. K.), « Scheduling of mpi-2 one sided operations over infiniband », *ipdps*, vol. 10, 2005, p. 215a.

[HSL08] Hoefler (T.), Schneider (T.) et Lumsdaine (A.), « Multistage Switches are not Crossbars : Effects of Static Routing in High-Performance Networks », dans *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, 10 2008.

[Int] Intel, *Accelerating High-Speed Networking with Intel I/O Acceleration Technology*.

[KRS01] Kurmann (C.), Rauch (F.) et Stricker (T. M.), « Speculative defragmentation – leading gigabit ethernet to true zero-copy communication », *Cluster Computing*, vol. 4, no 1, 2001, p. 7–18.

[LS99] Lusk (E.) et Swider (D.), « Improving the performance of mpi derived datatypes », dans *In MPIDC*, p. 25–30. MPI Software Technology Press, 1999.

[LVP04] Liu (J.), Vishnu (A.) et Panda (D. K.), « Building multirail infiniband clusters : Mpi-level design and performance evaluation », dans *SC '04 : Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 33, Washington, DC, USA, 2004. IEEE Computer Society.

[Mar07] Martinasso (M.), *Analyse et modélisation des communications concurrentes dans les réseaux haute performance*. PhD thesis, Université Joseph Fourier, BP 53 - 38041 Grenoble Cedex 9, France, mai 2007. 195 pages.

[Mer04] Mercier (G.), *Communications à hautes performances portables en environnements hiérarchiques, hétérogènes et dynamiques*. Thèse de doctorat, Université de Bordeaux 1, Labri, Bordeaux, France, décembre 2004.

[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.

[Moh05] Mohamed (N.), « Self-Configuring Communication Middleware Model for Multiple Network Interfaces », dans *The 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, 2005.

[mpi] *MPICH2-MX*. `http://www.myri.com/scs/download-mpichmx.html`.

[mva] *MVAPICH2 specification*. `http://mvapich.cse.ohio-state.edu/overview/mvapich2/`.

[Nag] *Algorithme de Nagle*. RFC 896.

[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.

[ope] *OpenMP*. `http://www.openmp.org/`.

[par] *Parasol Project*.

[PCI] *PCI Express*. `http://www.pcisig.com/`.

[PF01] Pratt (I.) et Fraser (K.), « Arsenic : A user-accessible gigabit ethernet interface », dans *INFOCOM*, p. 67–76, 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.

[PV95] Perlin (K.) et Velho (L.), « Live paint : painting with procedural multiscale textures », dans *SIGGRAPH*, p. 153–160, 1995.

[pvm] *Parallel Virtual Machine*. `http://www.csm.ornl.gov/pvm/`.

[qui] *The common system interface : Intel future interconnect*. `http://www.realworldtech.com/page.cfm`.

[RH98] Russell (R. D.) et Hatcher (P. J.), « Efficient kernel support for reliable communication », dans *SAC '98 : Proceedings of the 1998 ACM symposium on Applied Computing*, p. 541–550, New York, NY, USA, 1998. ACM.

[rmi] *Remote Method Invocation*. `http://java.sun.com/javase/technologies/core/basic/rmi/`.

[rpc] *Remote Procedure Call*. RFC 1831.

[Saa96] Saad (Y.), *Iterative Methods For Sparse Linear Systems*. Ed. PWS publishing Compagny, 1996.

[SCS+08] Seiler (L.), Carmean (D.), Sprangle (E.), Forsyth (T.), Abrash (M.), Dubey (P.), Junkins (S.), Lake (A.), Sugerman (J.), Cavin (R.), Espasa (R.), Grochowski (E.), Juan (T.) et Hanrahan (P.), « Larrabee : A many-core x86 architecture for visual computing », dans *Siggraph 2008*, 2008.

[sdp] *SDP specification*. `http://www.rdmaconsortium.org`.

[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 *In Second MPI Developer's Conference, Los Alamos*, p. 50–57. IEEE Press, 1996.

[swe] *Sweep3D : 3D Discrete Ordinates Neutron Transport*. `http://www.c3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html`.

[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 IEEE Parallel and Distributed Processing Symposium*, April 2006.

[SWP01] Shivam (P.), Wyckoff (P.) et Panda (D.), « EMP : Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing », dans *ACM/IEEE Conference on Super Computing*, p. ??–??, 2001.

[tbb] *Thread Building Blocks*. `http://www.intel.com/software/products/tbb/`.

[TBDN08] Trahay (F.), Brunet (E.), Denis (A.) et Namyst (R.), « A multithreaded communication engine for multicore architectures », dans *CAC 2008 : Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, avril 2008. IEEE.

[TBG+08] Thibault (S.), Broquedis (F.), Goglin (B.), Namyst (R.) et Wacrenier (P.-A.), « An Efficient OpenMP Runtime System for Hierarchical Architectures », dans Chapman (B. M.), Zheng (W.), Gao (G. R.), Sato (M.), Ayguadé (E.) et Wang (D.), éditeurs, *A Practical Programming Model for the Multi-Core Era, 3rd International Workshop on OpenMP, IWOMP 2007, Beijing, China, June 3-7, 2007, Proceedings*, vol. 4935 (coll. *Lecture Notes in Computer Science*), p. 161–172. Springer, 2008.

[TG07] Thakur (R.) et Gropp (W.), « Open issues in mpi implementation », dans *12th Asia-Pacific Computer Systems Architecture Conference (ACSAC 2007)*. Springer, 2007.

[TGL98] Thakur (R.), Gropp (W.) et Lusk (E.), « A case for using mpi's derived datatypes to improve i/o performance », *Supercomputing, 1998. SC98. IEEE/ACM Conference on*, Nov. 1998, p. 1–1.

[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.

[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 *SC'00*, p. 52–53, 2000.

[TW03] Tam (A. T. C.) et Wang (C.-L.), « Contention-Aware Communication Schedule for High-Speed Communication », *Cluster Computing*, vol. 6, n° 4, 2003, p. 339–353.

[Vin97] Vinoski (S.), « CORBA : integrating diverse applications within distributed heterogeneous environments », *IEEE Communications Magazine*, vol. 14, n° 2, 1997.

[VP07] Vaidyanathan (K.) et Panda (D.), « Benefits of I/O Acceleration Technology (I/OAT) in Clusters », *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, April 2007, p. 220–229.

[VSH+05] Vishnu (A.), Santhanaraman (G.), Huang (W.), wook Jin (H.) et Panda (D. K.), « Supporting mpi-2 one sided communication on multi-rail infiniband clusters : Design challenges and performance benefits », dans *High Performance Computing (HiPC)*. Springer, 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.

[WRF] *The Weather Research & Forecasting Model*. `http://www.wrf-model.org/`.

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