Mémoire de Master Recherche

présenté par

Elisabeth Brunet

Support d'ordonnancement et d'optimisation automatisés des communications pour les réseaux hautes performances

Encadrement : Olivier Aumage et Raymond Namyst

Laboratoire Bordelais de Recherche en Informatique (LaBRI)
Université Bordeaux 1

Table des matières

Chapitre 1

Introduction

1.1 Calcul parallèle intensif et communications

Les besoins en puissance de calcul, notamment ceux des applications de simulation (mécanique, sismologie, climatologie, dynamique moléculaire, physique nucléaire, etc.) sont en perpétuelle croissance. Ces besoins proviennent bien sûr de la nécessité de contenir le temps d'exécution sous un seuil acceptable, mais plus encore de la volonté d'accroître toujours davantage la précision et la fiabilité des résultats obtenus. On sait depuis longtemps que la réponse à ces besoins ne peut pas être apportée par la seule évolution en fréquence des processeurs, et donc que le recours au *parallélisme*[1] est inévitable[2].

Jusqu'à il y a une dizaine d'années, les configurations matérielles utilisées dans les grands centres de calcul (IDRIS, CINES, pour ne citer que les établissements français), les grands organismes (Météo-France, CEA, IFP), les entreprises et les laboratoires de recherche étaient exclusivement à base de *supercalculateurs* (ex : *Cray T3D*, *IBM SP*, *CM5*, etc.) Exceptée la catégorie des architectures multiprocesseurs à mémoire commune, qui exhibaient des puissances remarquables au prix d'une évolutivité quasi inexistante, ces machines étaient les seules à pouvoir garantir des temps de communication très faibles entre les processeurs, propriété cruciale pour bon nombre d'applications dites « fortement couplées », c'est-à-dire nécessitant de fréquentes communications entre les tâches parallèles.

Depuis une dizaine d'années, les progrès réalisés dans le développement des réseaux rapides pour les stations de travail ont permis d'entrevoir une alternative à ces supercalculateurs : les grappes de PC. En effet, ces technologies réseaux à courte distance (e.g. Myrinet, Infiniband) exhibent des performances de communication remarquables, tant sur le plan de la latence (moins d'une microseconde pour certaines technologies) que sur le plan du débit (plusieurs gigabits par seconde) et les grappes peuvent ainsi être vues comme un ensemble fortement couplé de processeurs. Bien que ces caractéristiques soient encore inférieures[3] à celles des supercalculateurs, le point essentiel est que le rapport performance/prix est nettement en faveur des grappes, sans compter les avantages en terme d'évolutivité de ces dernières.

Aujourd'hui, le succès des grappes est tel que le paysage architectural est en passe d'être dominé

[1] qui est une technique consistant à répartir un traitement sur plusieurs unités de calcul fonctionnant en parallèle

[2] C'est aussi vrai à l'intérieur des processeurs, qui voient leur puissance augmenter en partie grâce à la réplication de certains de leurs composants.

[3] d'un ordre de grandeur

par les grappes (elles constituent déjà plus de 50 % des machines classées au TOP 500 [21], qui recense les 500 machines les plus puissantes au monde).

1.2 Interfaces de communication hautes performances

Vu le niveau de performance des technologies réseau utilisées au sein des grappes, le moindre surcoût logiciel ajouté par une bibliothèque de communication peut très vite devenir un facteur limitant les performances globales des applications (appel système, recopie mémoire, etc.) C'est la raison pour laquelle les interfaces et protocoles de transport « classiques » tels que TCP/IP sont inadaptés et que des travaux de recherche[4] portant sur la définition d'interfaces de communication spécialisées pour le calcul intensif ont été menés. D'abord concentrées sur des bibliothèques capables d'exploiter efficacement le réseau interne des supercalculateurs (e.g. travaux sur *Active Messages* [3]), ces recherches se sont fortement intensifiées et diversifiées lors de la percée des grappes de PC dans le domaine du calcul intensif.

Étant donnée l'hétérogénéité des solutions matérielles disponibles pour les grappes, de nombreuses interfaces de communication sont naturellement apparues, chacune dédiée à une technologie spécifique. C'est ce que l'on appelle les interfaces « orientées performance »[5]. On peut citer pour exemple l'interface MX [14] pour la technologie Myrinet, Elan [9] pour Quadrics ou encore Sisci [7] pour le réseau SCI. Si ces interfaces de communication exhibent évidemment les meilleures performances possibles sur leurs technologies respectives, elles restent uniquement destinées à servir de cible pour des bibliothèques de plus haut-niveau. Développer une application directement au-dessus d'une telle interface ruinerait tout espoir de portabilité sur d'autres types de matériel réseau.

C'est précisément pour cette raison que la disponibilité d'interfaces de communication portables est cruciale pour le développement d'applications parallèles pour des grappes de PC. En fait, de telles interfaces existent depuis longtemps (e.g. PVM [22] puis surtout MPI [12]) et des implémentations sont disponibles sur la majorité des technologies existantes (MPICH-GM [13], SCI-MPICH [20], etc.) Toutefois, l'extrême généricité des mécanismes proposés, sorte de dénominateur commun à toutes les technologies, ne permet pas toujours à de telles interface d'exploiter toutes les fonctionnalités et les subtilités du matériel sous-jacent.

Récemment, de nombreux efforts de recherche se sont donc focalisés sur la recherche d'un meilleur compromis entre portabilité et performance. Ces efforts ont abouti à la définition d'interfaces de communication de niveau intermédiaire qui, moyennant l'utilisation de paradigmes de programmation un peu plus complexes que le traditionnel échange de messages, sont capables d'effectuer des optimisations spécifiques au matériel (e.g. VIA [4], Fast Message [16], Panda [19] ou encore Nexus [5]).

1.3 Vers une meilleure optimisation des transferts de données

Parmi l'ensemble des interfaces de niveau intermédiaire existantes, Madeleine [2] est sans doute l'une des plus sophistiquées. Cette interface permet en effet de dissocier complètement la description de la structure des messages à transmettre et le transfert effectif des données sur le

[4] à la fois en contexte académique et en contexte industriel

[5] On les appelle parfois aussi « interface bas niveau ».

réseau. Ce fonctionnement est rendu possible en permettant en particulier à l'application de spécifier des contraintes et des tolérances quant à la façon dont les données doivent êtres rendues disponibles à la réception. Il s'agit d'un contrat passé avec la bibliothèque sous-jacente : l'application exige des propriétés mais s'engage également à s'interdire certaines opérations pendant certaines phases, et en contrepartie la bibliothèque peut ajuster sa stratégie d'optimisation en fonction du matériel réseau tout en respectant les limites fixées par l'application.

La définition et l'implémentation de cette interface ont fait l'objet de la thèse d'Olivier Aumage [2]. MADELEINE est utilisée par plusieurs équipes de recherche en France et à l'étranger, et sert de fondation à des environnements logiciels complexes (PM^2 [17], PADICOTM [15], HYPERION [10]). Si cette utilisation en grandeur réelle a permis de valider les concepts mentionnés précédemment, elle a aussi montré les limites[6] de la version actuelle de la bibliothèque : de nombreuses optimisations potentielles « échappent » encore à MADELEINE.

L'étude présentée dans ce document a pour objectifs de comprendre d'où proviennent ces limitations, de proposer des évolutions (éventuellement profondes) de MADELEINE permettant d'améliorer la situation, de proposer et d'implanter de nouvelles stratégies d'optimisation et de les évaluer au-dessus d'un réseau rapide représentatif.

L'organisation du document est la suivante. Le premier chapitre dresse un panorama des principales techniques utilisées au sein des interfaces de communication pour obtenir des performances élevées sur réseaux rapides et présente les principales caractéristiques de l'interface MADELEINE. Dans le chapitre suivant, nous effectuons une analyse critique de MADELEINE qui met en lumière les multiples facteurs constituant un frein à la mise en place d'optimisations plus poussées. Puis nous décrivons notre proposition, centrée sur une nouvelle architecture logicielle qui constitue une véritable plate-forme d'expérimentation de nouvelles stratégies d'optimisation. Nous montrons comment l'adoption d'un protocole de transport ajoutant un léger surcoût lors du transfert des petits messages permet la mise en œuvre de schémas d'optimisation puissants. Nous détaillons dans le chapitre 4 quelques points clés concernant l'implémentation de notre proposition au-dessus de l'interface MX/MYRINET. Cette implémentation constitue une bibliothèque complète assurant la compatibilité ascendante des applications développées au-dessus de MADELEINE. L'évaluation présentée au chapitre 5 montre que les gains obtenus grâces aux nouvelles optimisations sont parfois substantielles et que le surcoût dans les cas « non optimisables » reste très faible. Enfin, les perspectives ouvertes par ce travail sont dressées au dernier chapitre.

[6] Il est étonnant de constater combien l'utilisateur, qui s'habitue aux optimisations automatiques, devient de plus en plus exigeant.

Chapitre 2

Contexte

2.1 Technologies de communication pour réseaux rapides

Les réseaux rapides actuels sont typiquement SCI, Myrinet, InfiniBand ou encore Quadrics.

SCI (*Scalable Coherent Interface*) [6] est un réseau de communication haute performance que la société Dolphins a ouvert au public en 1992. Ce réseau a pour paradigme dominant les *lectures/écritures en mémoire distante*. Ses performances s'élève à 2 μs en latence et 300 Mo/s en débit. Sa bibliothèque de bas-niveau est SISCI.

Le réseau Myrinet [14] est un produit de la société Myricom qui est leader du marché depuis 1994. Il est orienté *échange de messages*. Il affiche des performances à 2,5 μs de latence et 500 Mo/s de débit. Plusieurs bibliothèques bas-niveau lui sont dédiées : GM [11], BIP [18] et MX [14] pour la plus récente.

InfiniBand [1] est une norme récente définie par Intel, Dell, Hewlett-Packard, IBM, Microsoft et Sun Microsystems. Initialement, ce qui devait être une révolution au niveau architecture des machines (changements de bus, de mode de stockage, de réseau, etc.) s'est restreint aux réseaux rapides. Ce matériel basé sur le RDMA (*Remote Direct Memory Access*) est très prometteur en matière de débit (jusqu'à 6 Go/s annoncés) et un peu moins en latence avec 5 μs.

Et enfin Quadrics [9] qui est une technologie développée par la société Quadrics onéreuse mais très performante : 2 μs en latence et 900 Mo/s. Elle est aussi basée sur le RDMA.

Ce sont des réseaux courte-distance ce qui induit une faible probabilité de perte de signaux lors de transmissions : ce sont de réseaux fiables par nature. Une grappe de PC câblée par un réseau rapide ne nécessite donc pas de mécanisme de tolérance aux pannes. De plus, une grappe de PC est un réseau local : les tables de routage peuvent donc être pré-calculées.

2.2 État de l'art en matière d'interfaces de communication

Nous dressons ici un état de l'art guidé par les techniques utilisées pour exploiter les différentes technologies de communication pour réseaux rapides.

2.2.1 Communications depuis l'espace utilisateur

L'accès à une carte réseau est normalement réservé au système d'exploitation[1] : 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. Cependant ces derniers sont trop coûteux pour être conservés dans une interface de communication pour réseaux rapides. Pour illustrer, avec un noyau LINUX, le coût minimal d'un appel système se mesure en milliers de cycles sur un processeur INTEL PENTIUM IV, soit environ une microseconde, ce qui est comparable à la latence du réseau QUADRICS.

La solution, déjà utilisée pour les cartes graphiques, est de procéder à un court-circuitage du système d'exploitation pour accéder directement à la carte réseau. Cette stratégie 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*.

Il faut toutefois noter que le court-circuitage du système d'exploitation pose des problèmes en matière de sécurité et d'arbitrage d'accès aux ressources. En effet, le système d'exploitation ne contrôle plus l'accès de la carte en lecture et écriture dans la mémoire : rien ne l'empêche de corrompre des données ne lui appartenant pas. Le système d'exploitation multiplexe les accès aux cartes. L'utilisation simultanée d'une carte par plusieurs applications peut donc devenir conflictuelle. Par exemple, le pilote MX/MYRINET n'autorise l'ouverture que de trois ports. En admettant qu'une application n'a besoin que d'un seul port et qu'elle sait déterminer si un port est disponible, il ne peut y avoir plus de trois applications concourantes, ce qui n'est pas arbitrable sans le système d'exploitation.

Toutes les interfaces de communication contemporaines permettent des communications en espace utilisateur. L'interface précurseur dans le domaine est U-net [23].

2.2.2 Transmissions de données locales par PIO et DMA

Il existe deux moyens de transférer des données entre une carte et la mémoire d'un PC. La technique par entrées/sorties programmées par le processeur, appelé transfert PIO (*Programmed Input/Output*) fait transiter les données dans une zone de mémoire réservée au système d'exploitation avant qu'ils n'arrivent à la carte (ou inversement, à la mémoire) (fig. 2.1). La technique par accès direct à la mémoire, appelé transfert DMA (*Direct Memory Access*), s'effectue directement sans l'aide passer par le processeur (fig.2.2). Pour faire un transfert [8], le processeur commence par initialiser les registres du contrôleur DMA avec l'adresse mémoire source, la longueur à transférer, et l'adresse de destination. Le contrôleur DMA lève une interruption une fois le transfert accompli. L'opération initiée, le processeur est alors libre d'effectuer d'autres tâches.

L'emploi de l'un ou de l'autre est généralement choisi par l'interface de communication pilotant la carte, à partir de la taille de données à traiter[2]. L'allure des courbes de coûts en temps par rapport à la taille des données transférées de ces techniques (fig. 2.3) montre qu'il est préférable d'employer le transfert PIO pour les messages courts et le transfert DMA pour les plus longs. La

[1] on passe alors en mode privilégié

[2] à l'exception de quelques interfaces, telles que SISCI, qui permettent à l'utilisateur de choisir le mode de transfert

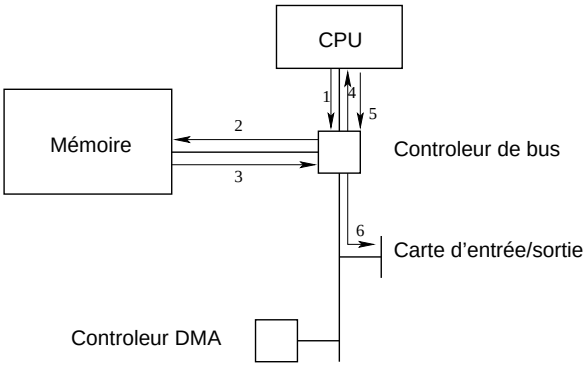

FIG. 2.1 – Transfert par entrée/sortie programmé.

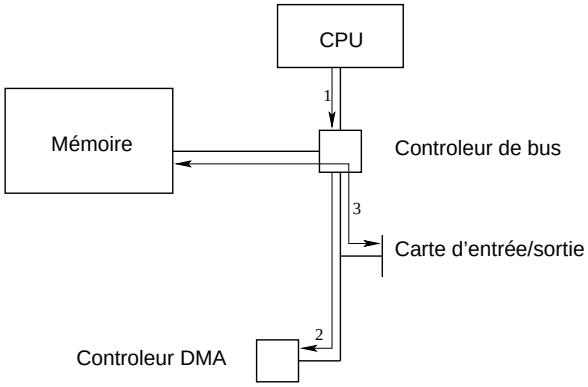

FIG. 2.2 – Transfert par accès direct à la mémoire.

courbe concernant le DMA présente un temps d'initialisation : il est dû à la synchronisation des circuits du contrôleur DMA avec la mémoire et à la réservation du bus PCI.

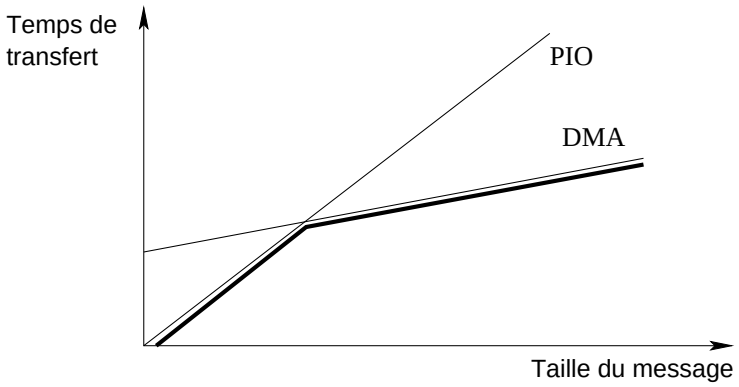

FIG. 2.3 – Allure de la courbe du coût de transfert de données.

2.2.3 Transfert de données vers une carte réseau sans copie intermédiaire

La principale cause de surcoût logiciel que les bibliothèques de communication tentent d'éliminer concerne les recopies mémoire intermédiaires des données qui sont parfois effectuées en émission ou en réception.

Les recopies en émission sont typiquement utilisées pour constituer des segments contigus à partir de données éparses en mémoire, pour ajouter des entêtes spécifiques au protocole sous-jacent, ou encore pour permettre à l'application de réutiliser des emplacements mémoire sans corrompre les données à transmettre (fig. 2.4). 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 (fig. 2.5).

Pour éliminer ces recopies « superflues » (au moins pour les messages de grande taille), il faut faire en sorte que la bibliothèque soit capable d'assurer un transfert direct des données de l'application vers la carte réseau en émission, et que la carte réseau destinataire soit capable de déposer

directement les données en mémoire à l'emplacement stipulé par l'application.

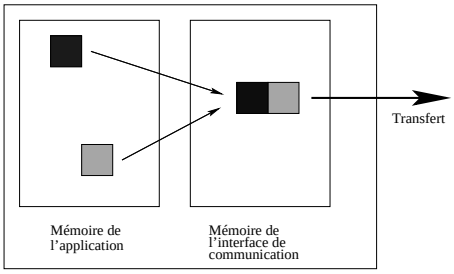

FIG. 2.4 – Copies intermédiaires avant transmission.

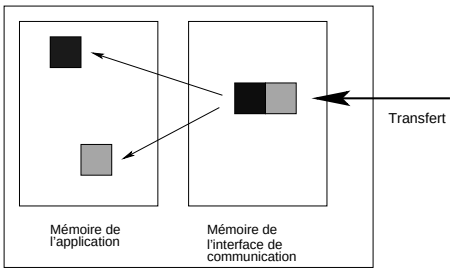

FIG. 2.5 – Copies intermédiaires à l'arrivée.

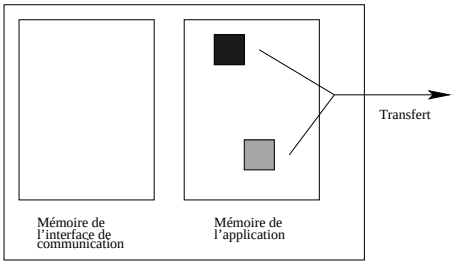

FIG. 2.6 – Transmission sans copie intermédiaire.

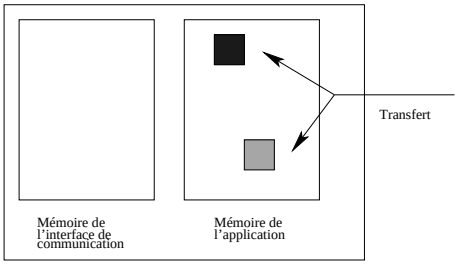

FIG. 2.7 – Transmission sans copie intermédiaire.

Èmission

Pour ce faire, les interfaces de communication obligent souvent l'application à effectuer l'opération d'émission en plusieurs étapes : d'abord l'application déclenche le début de l'opération en indiquant l'emplacement des données à émettre (fig. 2.6), ensuite elle peut effectuer d'autres tâches en veillant à ne pas modifier les données en cours de transfert, enfin elle doit interroger la bibliothèque pour savoir à quel moment le transfert est terminé.

Réception

Pour régler le problème des recopies à la réception (fig. 2.7), les bibliothèques utilisent des protocoles internes à base de *rendez-vous* qui vont permettre à un émetteur d'attendre que le récepteur soit prêt avant d'envoyer des données (fig. 2.8). Ainsi, ces dernières pourront être stockées directement au bon endroit par la carte réseau. Parfois, elles proposent une alternative à l'envoi de message traditionnel : l'écriture à distance. Dans ce cas, l'émetteur indique lui-même l'endroit auquel il faut stocker les données, le récepteur étant complètement passif. Toutefois, cette stratégie reporte souvent le problème du rendez-vous au niveau applicatif, car l'émetteur n'a pas toujours connaissance de l'emplacement à utiliser du côté récepteur.

En réalité, cette technique n'est rentable que lorsque la taille des données dépasse un certain seuil. En de ça, les données peuvent être envoyées de manière optimiste : le récepteur peut ne pas connaître leur destination en mémoire, mais tant pis ! Le cas échéant, il doit les stocker dans

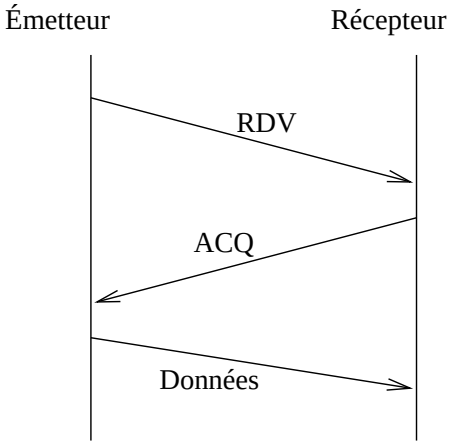

FIG. 2.8 – Scénario de rendez-vous entre l'émetteur et de récepteur.

son espace mémoire. Lorsque l'application désignera la zone de réception, les données seront alors copiées de la mémoire de l'interface de communication vers la mémoire de l'application. Le coût d'un tel envoi se solde donc à la somme du coût de l'allocation mémoire d'un tampon, du transit des données de la carte à la mémoire de l'interface de communication et de la copie des données vers la mémoire utilisateur. En se basant sur l'allure de la courbe de transfert entre une carte réseau et la mémoire (fig. 2.3), on obtient l'allure de la courbe 2.9.

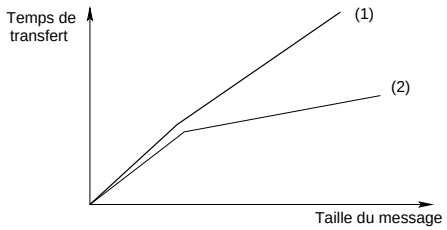

FIG. 2.9 – Allure de la courbe de transfert de données par copie en réception (1) à partir de l'allure de la courbe de transfert de données interne (2) 2.3.

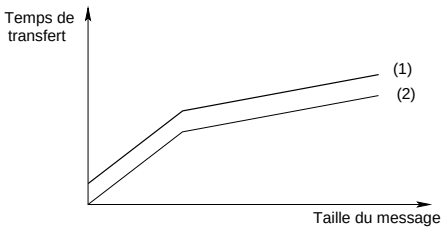

FIG. 2.10 – Allure de la courbe de transfert de données par rendez-vous en réception (1) à partir de l'allure de la courbe de transfert de données interne (2) 2.3.

La confrontation des courbes 2.9 et 2.10 amène à l'emploi de l'une ou de l'autre de ces techniques en fonction de la taille des données à envoyer (fig. 2.11).

Des interfaces de communication comme MX intègrent déjà ce mécanisme, ce qui facilite sensiblement l'implémentation des applications s'appuyant dessus.

Discussion

Remarquons pour finir que lorsqu'une bibliothèque effectue des communications en mode utilisateur, une difficulté surgit : le DMA de la carte réseau travaillant avec un adressage physique, il faut non seulement connaître la correspondance entre adresses virtuelles et adresses physiques au niveau utilisateur, mais il faut également s'assurer que celle-ci reste valide pendant toute la

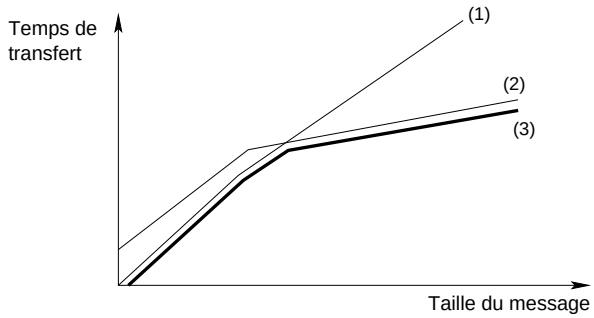

FIG. 2.11 – Juxtaposition (3) des courbes de transfert utilisant la méthode par copie (1) et la méthode par rendez-vous (2).

durée du transfert ! Pour résoudre le premier point en évitant un appel système par opération, les bibliothèques utilisent un cache d'associations mémoire virtuelle/mémoire physique. Lorsqu'une opération porte sur une région non encore référencée dans le cache, un appel système est effectué pour obtenir les adresses physiques concernées. Ce mécanisme, coûteux en régime transitoire, réussit à éviter les appels système lorsque l'application utilise des schémas de communication répétitifs, ce qui est généralement le cas. Pour résoudre le second point, les bibliothèques profitent des appels système précédemment évoqués pour interdire au système de modifier l'adresse physique des pages concernées (notamment en interdisant le swap sur ces pages) ou en modifiant le code du système de façon à tenir la bibliothèque (ou la carte réseau) informée des modifications éventuelles de la table des pages du processus.

2.2.4 Transferts *atomiques* de données non contigües en mémoire

Dans de nombreuses situations, les messages à transmettre contiennent des données éparses en mémoire telles que, par exemple, les éléments d'une structure irrégulière (e.g. matrice creuse). Pour cela, trois techniques peuvent être utilisées : envoyer les données séparément, copier les données dans un tampon intermédiaire (introduction d'une copie) et l'utilisation d'une technique appelée *gather/scatter* (grouper/éparpiller) qui consiste à transmettre directement à la carte la liste segments à envoyer. La carte rassemble ces données à la volée pour ne transmettre qu'un seul message logique sur le réseau. Au niveau matériel, l'implémentation de ce mécanisme repose la plupart du temps sur l'utilisation de « DMA chaînés ».

Notons que, malheureusement, peu d'interfaces permettent d'exploiter cette technique. VIA [4] et MX [14] font partie des interfaces qui exportent le mécanisme au niveau applicatif.

2.2.5 Le multiplexage des communications

Le multiplexage consiste à contrôler les flux de communication entre des modules applicatifs de manière à ce qu'ils ne puissent pas interférer. Pour cela, il faut pouvoir isoler leurs communications : il ne faut pas qu'une application puisse réceptionner un message qui ne lui est pas destiné et il ne faut pas que les communications de l'une bloque celles de l'autre. Une solution naïve serait d'établir une connexion entre chaque nœud pour chaque couple de modules logiciels. Cependant, les interfaces de communication ne fournissent en général pas des ressources illimitées. Par exemple avec MX [14],

on ne peut ouvrir que trois canaux de communication. Dans ces conditions, le multiplexage doit être logiciel.

Le multiplexage « logiciel » consiste à assurer les communications de plusieurs modules applicatifs sur un seul canal de communication « physique ». Pour cela, l'émetteur doit estampiller les messages selon leur source pour que le récepteur sache les rediriger vers la bonne cible. C'est typiquement ce que fait le protocole IP pour acheminer les paquets (quelque soit la couche logicielle qui les a générés) entre deux machines. Toutefois, contrairement à IP, les contraintes de performance associées aux messages ne permettent pas l'utilisation des techniques habituelles de multiplexage (ajout d'entêtes, fragmentation, etc.).

Il existe très peu d'interfaces réseau capable d'assurer le multiplexage d'un nombre élevé de canaux de communications. GM citegm est l'une des rares à fournir cette propriété. C'est aussi l'interface la moins performante pour le réseau MYRINET...

2.3 MADELEINE 3

MADELEINE 3 est une interface de communication orientée « envoi de messages » multi-protocole et multi-paradigme : multi-protocole car elle permet aux applications l'utilisation simultanément de plusieurs réseaux physiques ; et multi-paradigme car chaque protocole réseau peut exploiter plusieurs paradigmes de communication différents. Par exemple, un protocole tel que *Virtual Interface Architecture* [4] propose plusieurs techniques de transfert des données (*Remote-DMA*, envoi de messages, etc.) Au-dessus d'un tel protocole, l'organisation interne de MADELEINE 3 permet à l'implémentation de sélectionner dynamiquement le meilleur paradigme en fonction d'un certain nombre de critères (taille des données, contraintes de l'application, etc.).

2.3.1 Construction incrémentale des messages

MADELEINE 3 permet aux applications de construire incrémentalement les messages à transmettre (fig. 2.1). La construction d'un message débute par un appel à `mad_begin_packing` qui requiert en argument le processus destinataire. Ensuite, les données à transmettre peuvent être ajoutées au message en cours de construction en plusieurs étapes (`mad_pack`). La primitive `mad_pack` requiert en arguments l'adresse d'un bloc de données, sa taille et des *contraintes*[3] d'émission et de réception. La construction du message est finalisée par un appel à `mad_end_packing`. En fait, cette construction est purement *virtuelle* en ce sens que MADELEINE 3 peut décider soit d'émettre les données sur le réseau au fur et à mesure des appels à `mad_pack`, soit de les laisser *en place* en différant leur transfert, soit même de les recopier dans des tampons pré-alloués par le pilote des cartes réseaux. Une première conséquence est que les différents traitements effectués par l'application entre deux `mad_pack` peuvent éventuellement recouvrir des transferts réseaux.

Du côté récepteur, un message est reçu à l'aide d'une séquence similaire d'appels. La réception est initialisée par un appel à `mad_begin_unpacking` sur le canal spécifié en paramètre. Chaque bloc est alors extrait du message à l'aide de `mad_unpack` et la réception est finalisée par `mad_end_unpacking` qui permet de garantir que toutes les données ont effectivement été extraites du réseau. Pour des raisons d'efficacité, les messages ne sont pas auto-décrits au niveau de MADELEINE 3 : les blocs de données doivent donc précisément être extraits dans le même ordre et avec les mêmes spécifications de contraintes qu'à l'émission.

[3] Nous reviendrons sur ces *contraintes* dans la section 2.3.2.

`mad_begin_packing`	début de construction d'un nouveau message
`mad_begin_unpacking`	acceptation d'un message entrant
`mad_end_packing`	finalisation de la construction d'un message
`mad_end_unpacking`	finalisation de la réception d'un message
`mad_pack`	empaquetage d'un bloc de données
`mad_unpack`	dépaquetage d'un bloc de données

TAB. 2.1 – Interface de gestion de messages de MADELEINE 3.

2.3.2 Spécification des contraintes à l'émission et à la réception

MADELEINE 3 force l'application à spécifier des contraintes quant à l'émission et la réception des données transmises. Par exemple, lors d'une opération d'empaquetage (`mad_pack`), il est possible d'imposer que les données soient immédiatement disponibles du côté récepteur lors de l'opération `mad_unpack` correspondante. Au contraire, on peut relâcher totalement cette contrainte de disponibilité pour permettre à MADELEINE 3 d'optimiser le mode de transmission en fonction de la plate-forme sous-jacente (interface de communication, protocole, réseau). L'expression de telles contraintes par l'application constitue véritablement le point clé permettant l'obtention de bonnes performances tout en utilisant une interface *générique*.

Les contraintes d'émission disponibles sont les suivantes.

`send_SAFER` Cette contrainte indique que MADELEINE 3 doit empaqueter les données de manière à ce qu'une modification ultérieure de la zone de mémoire correspondante ne corrompe pas le message.

`send_LATER` Cette contrainte indique que MADELEINE 3 ne doit pas utiliser la valeur de la zone de donnée spécifiée tant que `mad_end_packing` n'a pas été appelée. Cela signifie que toute modification apportée à cette donnée entre l'appel à `mad_pack` et l'appel à `mad_end_packing` doit mettre à jour le contenu effectif du message.

`send_CHEAPER` C'est la contrainte par défaut. Elle permet à MADELEINE 3 d'effectuer toutes les optimisations disponibles pour transférer le bloc de données aussi efficacement que possible. En contrepartie, il ne doit être émis aucune hypothèse concernant l'accès de MADELEINE 3 à ces données. La zone de données ne doit donc pas être modifiée avant l'achèvement de l'opération d'émission. Il est à noter que la plupart des transferts de données impliqués dans des applications parallèles sont compatibles avec le mode `send_cheaper`.

Les contraintes disponibles en réception sont les suivantes.

`receive_EXPRESS` Cette contrainte oblige MADELEINE 3 à garantir que les données sont immédiatement disponibles après l'opération de dépaquetage. Elle est notamment indispensable si les données concernées sont nécessaires pour effectuer les dépaquetages suivant. Le coût de cette contrainte dépend directement des possibilités du protocole sous-jacent.

`receive_CHEAPER` Il s'agit de la contrainte par défaut en réception. Elle autorise MADELEINE 3 à *éventuellement* retarder l'extraction des données correspondantes jusqu'à l'appel de `mad_end_unpacking`. La disponibilité des données n'est pas garantie avant cet appel. En revanche, MADELEINE 3 dispose de la possibilité de réaliser des optimisations, notamment sous la forme de transferts agrégés, dans le but de maximiser l'efficacité de la communication.

Il est important d'insister sur le fait que les contraintes d'émission et de réception ne spécifient que la sémantique externe des opérations. MADELEINE 3 garde la liberté de sélectionner l'ordonnancement le plus approprié pour l'envoi des blocs de données sur le réseau. Par exemple, certains protocoles réseaux seront utilisés plus efficacement si chaque opération `pack` donne lieu à un transfert distinct. D'autres permettront la réalisation de transmissions groupées efficaces. D'autres enfin nécessiteront l'emploi de tampons pré-alloués associés à des recopies additionnelles.

Exemple

La figure 2.12 illustre la puissance de l'interface MADELEINE 3 dans une situation où le récepteur d'un message ne connaît pas a priori la nature de son contenu. Il s'agit de transférer un tableau d'octets donc la taille n'est pas connue par le processus récepteur. Dans ce cas, le récepteur doit d'abord extraire la taille du tableau (un entier) en mode `EXPRESS`, de façon à pouvoir consulter la valeur immédiatement après l'opération `mad_unpack`. Ensuite, il peut allouer dynamiquement une zone mémoire de taille suffisante puis extraire le tableau en mode `CHEAPER`. Il faudra alors appeler la primitive `end_unpacking` avant d'accéder au contenu du tableau, MADELEINE 3 ne s'engageant pas à ce que les données soient disponibles au moment du `mad_unpack`.

Du côté de l'émetteur

```
(1) connexion = mad_begin_packing(canal, dest);
(2) mad_pack(connexion, &taille, sizeof(int),
             send_CHEAPER, receive_EXPRESS);
(3) mad_pack(connexion, tableau, taille,
             send_CHEAPER, receive_CHEAPER);
(4) mad_end_packing(connexion);
```

Du côté du récepteur

```
(1) connexion = mad_begin_unpacking(canal);
(2) mad_unpack(connexion, &taille, sizeof(int),
               send_CHEAPER, receive_EXPRESS);
    tableau = malloc(taille);
(3) mad_unpack(connexion, tableau, taille,
               send_CHEAPER, receive_CHEAPER);
(4) mad_end_unpacking(connexion);
```

FIG. 2.12 – Utilisation des contraintes d'émission/réception dans le cas du transfert d'un tableau de taille variable.

Chapitre 3

L'interface de communication Madeleine 4

3.1 Analyse critique de Madeleine 3

Madeleine 3 ne permet pas de réaliser toutes les optimisations qu'un utilisateur voudraient voir s'opérer sur ses données. Pour permettre cette évolution, nous allons ici procéder à l'analyse des différents points limitant de Madeleine 3.

3.1.1 Optimisation de schémas de communication complexes

Les algorithmes d'émission et de réception d'une application sont à implémenter de manière symétrique (fig. 2.12) : à chaque appel à `mad_pack` correspond un appel à `mad_unpack` et ces appels doivent s'opérer dans le même ordre. Madeleine 3 se base sur cet état de fait pour envoyer les messages dans leur forme brute[1] ce qui a pour avantage de conserver des temps de transfert proches de ceux de la bibliothèque bas niveau. L'émetteur envoie toujours le message auquel le récepteur s'attend.

Cette entente est très limitant dans le sens où aucun facteur unilatéral ne peut être utilisé pour organiser différemment l'envoi des paquets. Le récepteur n'est pas en mesure de réordonner les paquets arrivant. Ils ne peuvent donc ni être intervertis, ni groupés de manière opportuniste. L'envoi sur plusieurs cartes (le *multi-rail*) est également à exclure : si le choix d'utilisation d'une carte réseau est fonction de facteurs inconnus du récepteur, ce dernier ne peut déterminer sur quelle carte réseau il doit attendre les données.

Pour remédier à cela, l'ajout d'un entête discriminant aux messages est nécessaire. Ces entêtes représentent une masse d'information supplémentaire à transmettre ce qui, certes, implique un surcoût évident au transfert des données par rapport à un envoi brut mais permet d'envisager des schémas de communications plus complexes que ceux supportés par Madeleine 3 actuellement. À partir des informations contenues dans cet entête, le récepteur est en mesure d'identifier le destinataire des paquets et les `mad_unpack` qui leur correspondent. La progression du transfert des paquets peut alors se faire indépendamment de l'application qui les a déposés. Ils sont intervertibles et regroupables. Le récepteur doit se référer à leur étiquetage pour les manipuler.

[1] sans entête de description

Pour profiter des transferts de données sans copie intermédiaire, certains messages doivent être directement placés à leur destination. Il faut alors procéder par la prise d'un rendez-vous préalable. Le rendez-vous est alors porteur de cette description : il annonce l'envoi de données identifiées par le récepteur. Comme celui-ci est en attente de leur réception, un entête collé n'est pas utile.

De plus, une étude menée conjointement entre ALCATEL et l'équipe RUNTIME a montré que le multiplexage est d'autant plus performant qu'il est fait au plus près possible du matériel. MADELEINE 3 est donc en bonne position pour se charger de ce multiplexage. Or, MADELEINE 3 ne le gère pas nativement. Ce mécanisme n'était pas dans ces objectifs inititiaux, un module additionnel est cependant proposé pour pallier à cela. Avec la présence d'entêtes, l'interface de communication est capable d'associer un paquet au module applicatif qui le concerne, le multiplexage logiciel se gère donc nativement. Le regroupement des paquets en émission ne se limite pas à celui des paquets d'un même module mais à l'ensemble de ceux de même destination.

3.1.2 Progression des communications en parallèle de l'application

Dans MADELEINE 3, les transactions réseau ne sont invoquées que lors d'appels aux fonctions `mad_pack`, `mad_unpack`, `mad_end_packing` et `mad_end_unpacking`[2] : rien n'est soumis à la carte réseau pendant les phases de calcul de l'application, même s'il reste encore des packs à attente.

Pour un bon recouvrement des temps de communications, la carte doit être occupée au maximum. Une carte réseau classique ne peut physiquement envoyer ou recevoir qu'un message à la fois. Du travail doit lui être fourni en fonction de son état d'occupation : une fois un envoi accompli, une autre tâche doit lui être soumise. Cela suit le principe d'un ordonnanceur de processus qui, lorsqu'il est invoqué par un processeur, déroule un algorithme permettant de choisir un nouveau processus prêt. Pour que cela soit possible, le découplage de l'activité de la carte réseau et du flot d'exécution de l'application est obligatoire. Avec l'appel à `mad_pack` et `mad_unpack`, l'application doit se limiter à « passer commande » des transactions réseau nécessaires (fig. 3.1).

3.1.3 Séparation des contraintes applicatives et des stratégies d'optimisation des communications

Les transactions de MADELEINE 3 ne sont déclenchées qu'au moment du dépôt d'un paquet ou à la terminaison d'un message. Le choix d'envoyer ou de stocker les données intervient à ce moment et est fait à partir des modes de transmission du paquet courant et de la méthode transfert spécifiée par le pilote réseau utilisé : l'opération d'optimisation est finalement dirigée par l'application.

Autoriser la progression des communications en parallèle de l'application remet en cause le mode d'application des optimisations de MADELEINE 3. L'optimisation intervient pour alimenter la carte et non pour satisfaire les contraintes données par l'application.

L'optimisation consiste en l'*observation* d'une liste recensant les paquets en attente et le *regroupement* de certains d'entre eux de façon à ce que le transfert soit le plus rentable en temps pour l'application. Le regroupement des paquets implique le respect de deux sortes de contraintes : les contraintes applicatives et les contraintes liées à la méthode de transfert du paquet spécifiée par le réseau utilisé. Les contraintes applicatives se réduisent à autoriser ou non l'inversion de deux paquets se rapportant au même message. Pour ce faire, les règles d'inversion sont consultables par toutes les stratégies d'optimisations : elles peuvent prendre une forme générique comme celle d'une matrice confrontant les caractéristiques de deux paquets. Les contraintes liées à un protocole sont

[2] exception faite sur des machines multithreadées.

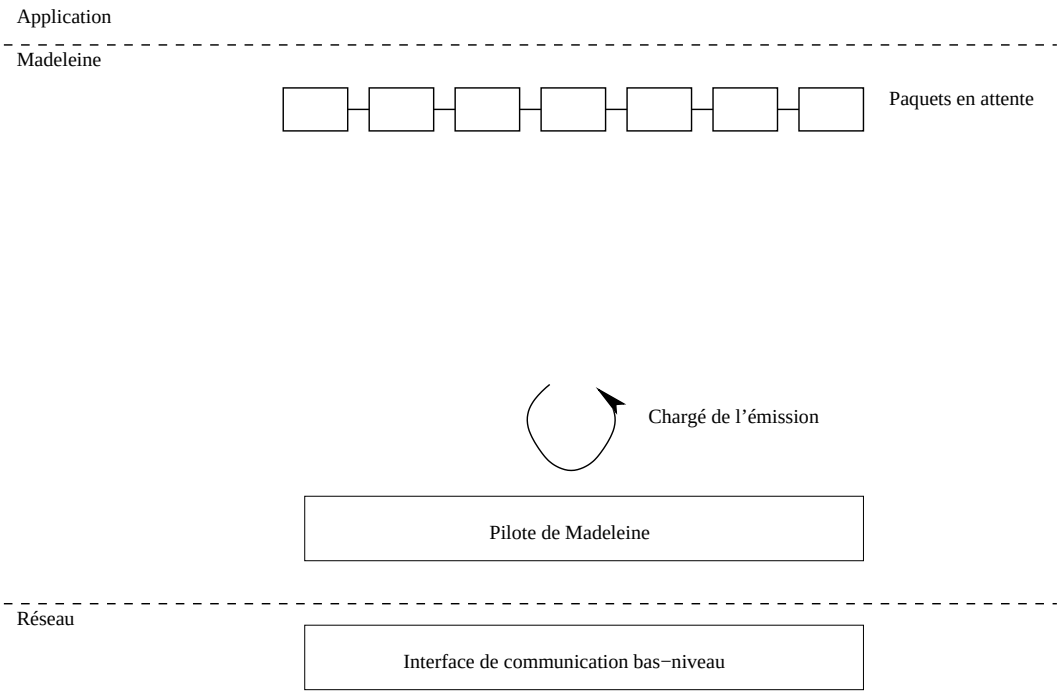

FIG. 3.1 – L'application dépose des paquets en attente dans la liste qui sont alors pris en charge par le thread chargé de l'émission.

notamment des contraintes sur la taille des messages, sur la nécessité d'un rendez-vous, etc. Ces dernières stratégies sont exprimées par le pilote réseau car il est le seul à pouvoir exprimer de telles spécificités.

Plusieurs stratégies peuvent être appliquées à une même liste pour obtenir le paquet prêt à poster selon des ordres différents. Par exemple, l'agglomération des paquets courts puis le regroupement d'un rendez-vous à un message court et inversement.

La détermination de la meilleure combinaison d'optimisations est difficile. Pour pouvoir confronter toutes les combinaisons, la détermination d'un élément de comparaison est nécessaire, il pourra s'agir du temps de transfert en fonction de la taille du paquet proposé grâce à un échantillonnage préalable des temps de transfert de chaque réseau. Ainsi, un catalogue de stratégies prédéfinies ou soumises par l'utilisateur indépendantes du matériel sous-jacent pourra être étudié et la meilleure séquence d'optimisations pourra être retenue.

3.1.4 Anticipation des communications

Certains enchaînement de paquets imposent un envoi immédiat à MADELEINE 3. Par exemple, deux paquets successifs dont le mode d'envoi est `mad_receive_EXPRESS` : le récepteur bloque son exécution tant que les données relatives à un paquet dont le mode de réception est `mad_receive_EXPRESS`, l'émetteur envoie donc dès son dépôt le paquet concerné.

L'état d'occupation de la carte peut aussi déclencher une transaction. Par exemple, la carte est libre, des paquets sont en attente mais le suivant peut se révéler intéressant pour l'optimisation. MADELEINE 3 manque de clairvoyance pour estimer s'il est préférable de retarder le transfert.

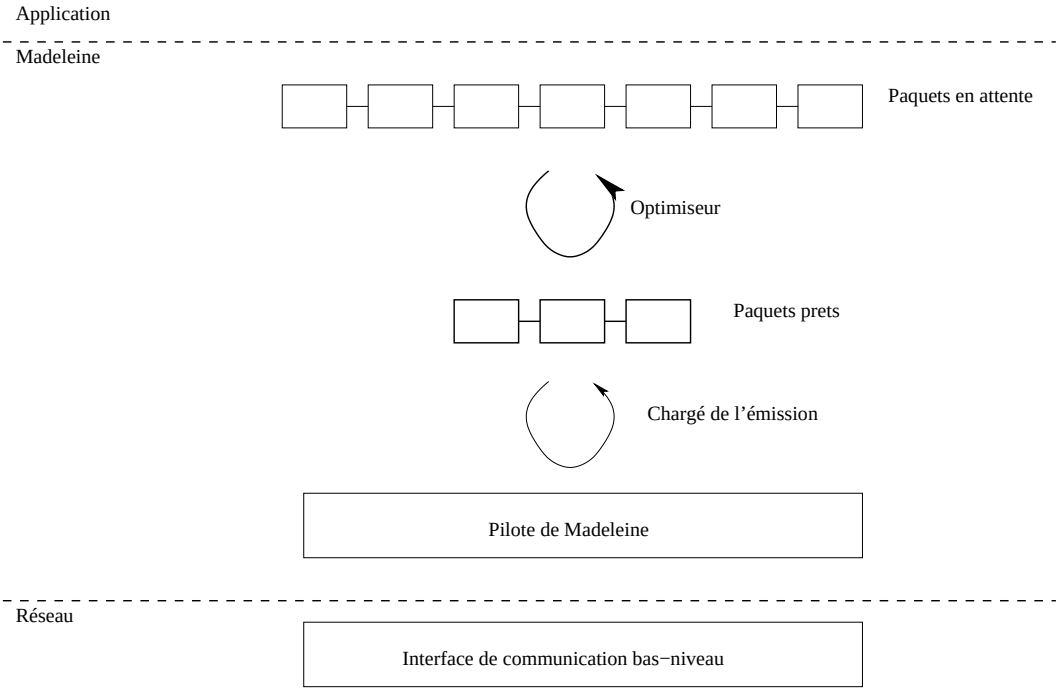

FIG. 3.2 – L'optimiseur applique des stratégies sur la liste des paquets en attente d'émission pour alimenter la liste des paquets prêts.

Une extension des fonctions `mad_pack` et `mad_unpack` est nécessaire pour l'en informer. Elles doivent indiquer si l'appel est suivi d'une période de calcul ou par un autre paquet et en indiquer les modes de réception et d'émission.

3.2 L'architecture de MADELEINE 4

MADELEINE 4 a engendrée des changements dans l'architecture de MADELEINE 3. Seules les phases d'établissement et de fermeture des connexions ont été sauvegardées à l'identique. L'architecture de MADELEINE 4 est désormais organisée en trois couches : la couche générique, chargée de la gestion des données, la couche relative à l'optimisation et la couche d'abstraction, pilotant les cartes de communication (fig. 3.3). Nous procédons ici à la présentation synthétique du rôle de chacune de ces composantes. Comme la partie liée à l'optimisation ne concerne pas le côté récepteur, nous décrivons ici une vue de la partie émettrice.

Le producteur

Avec l'appel à `mad_pack` et `mad_unpack`, l'application se limite à présent à « passer commande » des transactions réseau nécessaires à MADELEINE 4. La partie haute a pour rôle de collecter les paquets et les informations qui leur sont associées. Ces dernières doivent être discriminantes pour que le récepteur puisse identifier les données : le numéro de canal emprunté, l'identifiant de l'expéditeur et un numéro de séquence. Les paquets sont encapsulés dans une structure qui présente toutes les méta-données nécessaires à leur traitement. Ces nouveaux éléments sont placés dans une liste

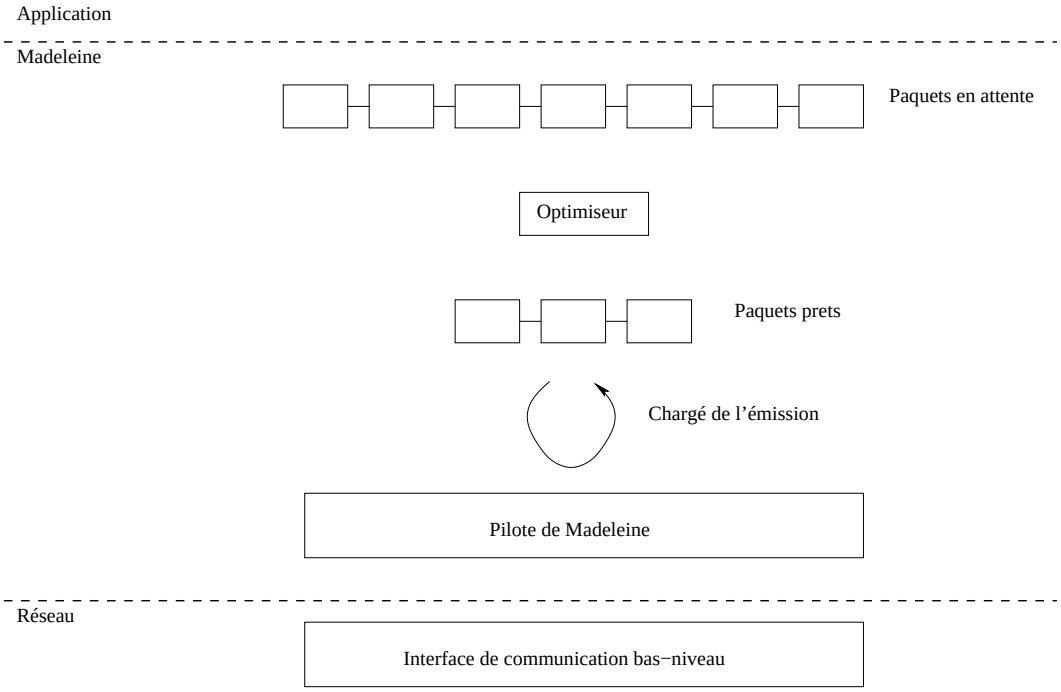

FIG. 3.3 – Architecture de MADELEINE 4.

tampon. Il existe une liste par interface de communication bas-niveau portée dans MADELEINE 4. Cette séparation permet d'estimer le travail à venir des cartes utilisées par une même interface de communication basse. En effet, si plusieurs cartes utilisent la même interface, elles sont alimentées par la même liste. Le schéma 3.3 est reproduit pour chaque interface de communication portée dans MADELEINE 4 mais n'est pas représenté pour des questions de lisibilité.

Certaines applications ont besoin de transmettre beaucoup de données. Pour améliorer leur débit, elles peuvent répartir équitablement leurs communications entre tous les réseaux à disposition. Une autre possibilité est de se servir du premier réseau disponible. Pour cela, MADELEINE 4 ajoute une liste tampon associé au pseudo-réseau ANY qui est ponctionnée lorsqu'une carte réseau a consommé tous ses paquets.

L'optimiseur

Les paquets lui sont déposés et c'est la carte qui va «tirer à elle » le travail lorsqu'elle en a besoin. L'optimisation est alors chargé de manipuler la liste pour organiser les paquets et fournir un assemblage conforme aux contraintes applicatives et liées au réseau qui soit le plus rentable en temps de transfert à la carte. Ce paquet est placé dans une liste intermédiaire à disposition de la partie la plus basse. Cette liste doit être conservée la plus réduite possible. En effet, si l'optimiseur la remplit et qu'elle n'est pas consommée, c'est autant de paquet qu'elle aurait pu conserver pour faire des optimisations plus appropriées. Cela lui laisse une marge de manœuvre plus importante qui ne peut être que bénéfique. Néanmoins, elle ne doit pas être vide : si l'optimiseur doit fournir un paquet à consommer immédiatement, le temps de formation est perdu. Comme dit plus tôt, la carte ne peut consommer qu'un paquet à la fois donc un paquet en cours et un paquet d'avance est

une mesure adaptée.

Précédemment l'optimiseur était représenté comme un thread pour améliorer la compréhension. Cependant, l'optmisation est une opération ponctuelle : elle n'a pas de raison d'être active. L'optimiseur est donc un appel de fonction.

Le consommateur

Le thread d'émission est au service de la carte réseau. Il surveille l'état d'avancement de la carte réseau. Dès qu'un paquet est parti, il le remplace par un autre qu'il prend dans la liste fournie par l'optimiseur. Il est également chargé de lancer les procédure d'exploitation des paquets pour que les données soient dirigées vers leur emplacement final, que les messages de contrôle soient pris en compte (demande de rendez-vous, acquittement, etc.)

Chapitre 4

Implémentation

L'implémentation de MADELEINE 4 est une partie majeure de l'étude menée. Ainsi, après avoir décrit la couche dédiée à l'optimisation, nous schématisons ici le fonctionnement de cette nouvelle architecture en terme d'implémentation puis nous détaillons le scénario d'envoi d'un message composé d'un paquet court et d'un paquet long.

4.1 Couche générique dédiée à l'optimisation

La tâche de l'optimiseur consiste à sélectionner, en fonction d'un jeu de stratégies prédéfinies ou non, un ensemble cohérent de paquets à envoyer sur une carte.

Pour ce faire, l'optimiseur travaille à partir d'une liste de paquets, paquets produits par les appels successifs de l'application à la fonction `mad_pack`[1], ces paquets sont enrichis de quelques méta-données (ex : le destinataire, le numéro de séquence du paquet, etc.) Pour optimiser l'accès aux éléments de cette liste, celle-ci est subdivisée en plusieurs sous-listes spécifiques, non pas directement aux cartes, mais à chacune des interfaces de communication bas niveau. De plus, une liste supplémentaire recueille les paquets dont le réseau à emprunter pour leur transmission n'a été spécifié par l'application. Les éléments récoltés dans cette dernière liste peuvent être ainsi ordonnancés sur la première carte réseau libre. Ainsi, MADELEINE 4 est bien armée pour exploiter pleinement les configurations *multi-rails*, configurations où l'on dispose de plusieurs cartes pour paralléliser les communications.

Pour sélectionner les paquets à transmettre, l'optimiseur dispose d'un éventail de stratégies qui peuvent être soit prédéfinies, soit précisées par l'application. Cependant, le choix de l'ordre d'application des optimisations sur un ensemble de paquets pour en tirer la meilleure stratégie n'est pas aisé, comme l'illustre l'exemple suivant.

Prenons deux optimisations consistant à agglomérer des messages courts : l'une groupant des demandes de rendez-vous à un message court, l'autre formant un "gros" message court à partir de "petits" messages courts. Faut-il grouper tous les messages courts au risque que la demande de rendez-vous soit envoyée seule par la suite ou doit-on réserver un message court à la demande de rendez-vous ?

Pour répondre à ce problème, nous avons prévu d'intégrer un simulateur afin d'estimer le résultat des différentes stratégies. Ce simulateur pourra être calibré à l'initialisation via échantillonnage des

[1] l'optimisation des communications ne concerne que la partie émission. Le récepteur réagit uniquement à ce que lui envoie l'émetteur

réseaux. Cependant, dans notre étude sur la bibliothèque MX, nous avons implémenté ces deux stratégies dans un ordre prédéfini.

4.2 Description du fonctionnement

Le fonctionnement de MADELEINE 4 est décrit dans la figure 4.1. Nous menons pour chaque repère une brève description dans la suite de cette section.

La figure 4.1 illustre le fonctionnement de MADELEINE 4 ; nous commentons élément par élément cette illustration :

1. **Niveau applicatif.** L'application transmet les zones de données par des appels à la fonction `mad_pack`. Ces données utilisateur sont simplement référencées et ne sont à aucun moment copiées.
2. **Fonction `mad_pack`.** L'exécution de la fonction `mad_pack` a pour effet l'empaquetage des données dans une structure appelée `mad_iovec`. Cette structure recense toutes les informations nécessaires au cheminement du paquet jusqu'à son émission effective. La structure `mad_iovec` comporte deux parties : une pour les méta-données (destinataire, numéro de séquence du paquet, taille des données, etc.) et une partie pour les données. Comme les paquets sont amenés à être regroupés par l'optimiseur, les différentes références sur les données sont organisées dans un tableau de `struct iovec` qui est une structure de la bibliothèque standard C.
3. **Liste protocolaire** Les paquets sont collectés dans la liste spécifique au protocole de communication désigné par l'application (il peut y en avoir plusieurs). À terme, celle dernière pourra néanmoins indiquer qu'elle souhaite que le paquet parte sur la première carte disponible. Pour cela, elle spécifiera l'utilisation du pseudo-réseau ANY.
4. **Fonction de progression.** La fonction de progression, appelée `mad_make_progress`, a un rôle clé : elle doit dans un premier temps consulter l'état de la carte grâce à la structure `mad_track`(5). Si la carte a consommé des `mad_iovec`[2], il faut la ré-alimenter. Pour cela, il faut initier de nouvelles transmissions puis demander un nouveau `mad_iovec` pour anticiper le prochain envoi, ensuite il s'agit de sélectionner, par ordre de priorité, soit un `mad_iovec` en cours de transmission (`MX/Myrinet` a la particularité de ne pas pouvoir envoyer des structures non contigües de plus de 32 ko), soit des messages de la liste (6) à intercaler entre les envois, soit d'appeler l'optimiseur (7) qui fournira alors un nouveau `mad_iovec`.

   Notons que dans une prochaine version de notre implémentation, un thread sera dédié à l'exécution de la fonction de progression.
5. **Ensemble des `mad_iovec` en cours d'emission.** La structure `mad_track` recense les `mad_iovec` en partance. Comme la bibliothèque MX/MYRINET admet deux méthodes de transfert (soit par copie, soit par rendez-vous), nous avons ouvert des points de communication pour chacune d'elles (représenté ici par les deux sous-structures). Elles ont pour fonction de *pipeliner* des `mad_iovec` en cours de transmission.
6. **Liste de contrôle** Cette liste recense les messages de contrôle et les messages longs venant d'être acquittés toujours en attente ; ces messages ne proviennent pas directement de la liste des paquets prêts (3).

[2] à partir de l'appel à `mad_pack`, on ne manipule que des structures `mad_iovec`.

7. **Optimiseur.** L'optimiseur applique les stratégies d'optimisations sur les éléments de même destination de la liste (3) et fournit à (4) un nouveau `mad_iovec` à émettre.
8. **Liste des paquets non acquittés.** Lors de la partie optimisation, l'optimiseur doit consulter le pilote réseau afin de connaître ses contraintes. En l'occurrence, les paquets longs sont à envoyer avec une prise de rendez-vous. Ainsi, pour un paquet long, l'optimiseur va dans un premier temps envoyer une demande de rendez-vous qu'il rattache à un message court de même destination et met de côté le message long dans la liste (8) en attendant son acquittement de bonne réception. Les messages de contrôle (demandes de rendez-vous et acquittements) se rattachent à un `mad_iovec` au même titre qu'un paquet de données. Ces messages ont chacun un format spécifique, indiqué par les premiers bits d'entêtes.
9. **Liste des paquets inattendus.** Cette liste recueille les messages reçus dont le `mad_unpack` associé n'a pas encore été exécuté par l'application. Une fonction de progression de réception a pour tâche de recoller les morceaux.

4.3 Scénario d'envoi

Nous simulons à présent l'envoi d'un message constitué de deux paquets : l'un court et l'autre long. Nous simplifions la description en nous plaçant en début de session, c'est-à-dire à l'amorçage du mécanisme.

Les appels à la fonction `mad_pack` associés à chacun des paquets provoquent leur empaquetage dans des structures `mad_iovec` qui sont placés dans la liste des éléments prêts à être envoyé (3). L'appel à la fonction `mad_end_packing` provoque l'exécution de la fonction de progression en émission. Comme les *pipelines* sont vides et la liste des messages en attente d'émission (6) aussi, l'optimiseur est sollicité. Grâce à la consultation de fonctions du pilote réseau de Madeleine 4, l'optimiseur sait comment envoyer les paquets. La première stratégie, à savoir l'agglomération des petits messages, ne provoque pas de groupement. Par contre, l'application de la seconde (agglomération des demandes de rendez-vous avec un paquet court) groupe le premier paquet et la demande de rendez-vous. Le paquet large est placé en attente dans la liste (8). Le paquet nouvellement construit est placé directement en tête du pipeline dédié à l'envoi des messages courts, son envoi est alors posté. Comme il n'y a plus de paquets disponibles dans la liste (3), la fonction de progression s'achève. Cependant, la fin de message implique l'émission de tous les paquets postés. Les fonctions de progression d'émission et de réception sont donc appelées en boucle tant que les paquets n'ont pas été marqué comme envoyés.

Lors d'un passage de la fonction de progression sur la structure (5), l'appel à des primitives du protocole sous-jacent permet de savoir si le paquet a été émis ou non. Si c'est le cas, le paquet court est marqué comme envoyé.

De son côté, le récepteur dépaquette ce qu'il vient de recevoir, il place les données à l'emplacement qu'il a associé à ses `mad_iovec` de réception courant et cherche s'il y en a un associé à la demande de rendez-vous. Si oui, il poste la réception de ce dernier et envoi un acquittement. Sinon, il envoie un refus de réception et stocke la demande de rendez-vus pour la traiter à nouveau ultérieurement.

Sur réception d'un acquittement, le message long associé est envoyé sur le pipeline dédié aux messages longs. Sur réception d'un refus de réception, l'émetteur refuse tout nouvel envoi d'un message long à destination de ce même module applicatif.

Lorsque la fonction de progression constate que les données ont finies d'être transmises, la fonction `mad_end_packing` s'achève. Le message a été envoyé.

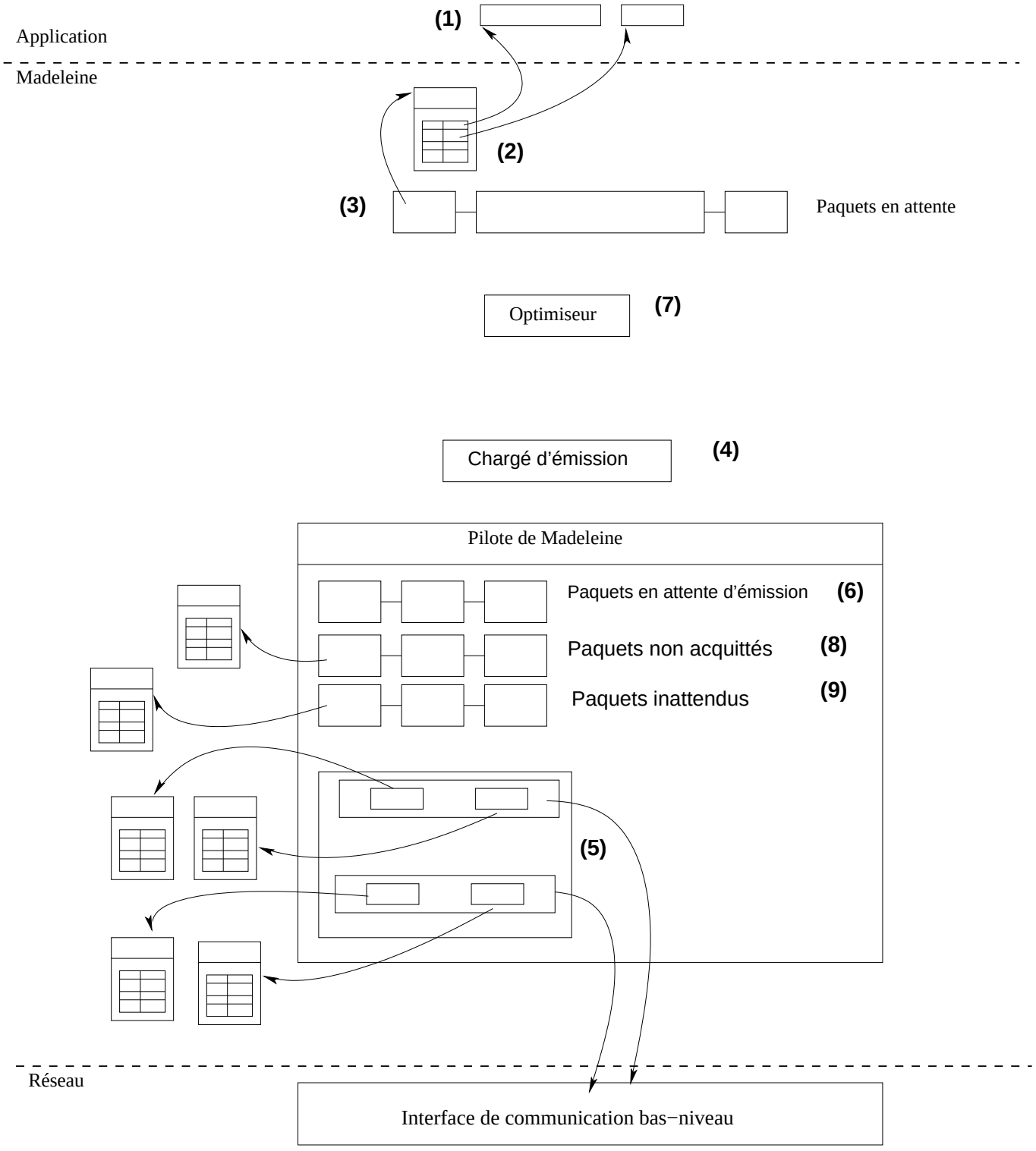

FIG. 4.1 – Vue schématique de l'implémentation de MADELEINE 4.

Chapitre 5

Évaluation

Nous confrontons Madeleine 4 à Madeleine 3 en terme de performances. Nous commençons par le test classique du *ping-pong* puis par des exemples plus ciblés afin de mesurer la portée des optimisations réalisées. Ces tests ont tous été réalisés au dessus de la bibliothèque de communication MX/Myrinet. Les tests sont réalisés sur la grappe de PC *Dalton* du projet Runtime. Chaque nœud est un bi-Xeon 2,6 GHz sous Linux 2.6 et est équipé d'une carte Myrinet 2000.

Lors de l'étude préalable de MX/Myrinet, un changement de méthode de transfert a été identifié : à hauteur de 32 ko, la méthode de transfert par envoi direct avec recopie de données à l'arrivée est remplacée par la méthode de transfert avec prise de rendez-vous. Ainsi, nous appelons *paquets courts*, les paquets de taille inférieure à 32 ko, et *paquets longs*, ceux de taille supérieure à 32 ko.

5.1 Surcoût brut par message élémentaire

Nous commençons par mesurer les performances de Madeleine 4 en les comparant à celles de la bibliothèque de communication MX et celles de Madeleine 3 (fig. 5.1 et 5.2). Le *ping-pong* consiste en une série de 10 000 allers-retours sur le réseau (lissant ainsi ses éventuelles fluctuations) avec des données de taille s'échelonnant de 4 octets à 2 ko. Pour une taille donnée, le temps moyen d'un transfert entre deux machines est obtenu à partir du temps écoulé entre le début et la fin de cette série d'échanges divisé par deux puis par le nombre de boucles effectuées.

Madeleine 4 induit un surcoût logiciel supérieur à celui de Madeleine 3. Deux facteurs sont notamment en cause. Tout d'abord, la description des entêtes implique que, à taille de données utiles égales, la masse (entêtes + données utiles) des données effectivement transmise est supérieure avec Madeleine 4. Ensuite, l'application d'optimisations sur une liste d'un élément unique ne peut être que superflue. Nous sommes donc pénalisés sur les cas où l'intervention de l'optimiseur est inutile[1].

5.2 Agglomération de paquets courts

Envoi d'un message selon un nombre variable de paquets Une des stratégies d'optimisation que nous avons implémenté consiste à agglomérer les paquets courts. Pour évaluer son impact potentiel, nous nous intéressons à l'envoi d'un message constitué d'un bloc de données unique

[1] par la suite, ce point pourra se simplifier par le court-circuitage de l'optimiseur

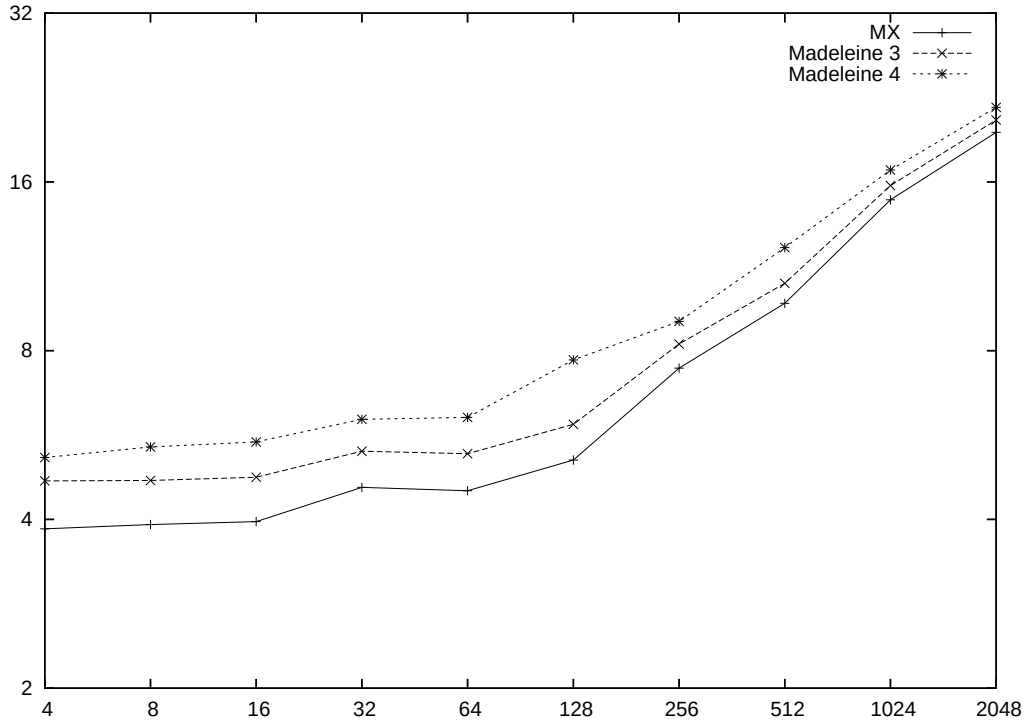

FIG. 5.1 – Ping-pong - latence - MX vs MADELEINE 3 vs MADELEINE 4.

débité en un nombre variable de paquets. Pour une taille donnée, nous subdivisons le message en N paquets de taille égale, N variant de 1 à la taille du message. Ainsi, nous pouvons observer le manque à gagner entre l'envoi d'un message composé d'un seul paquet transmis en un seul envoi par rapport à un message composé de N paquets transmis en N envois.

Les courbes sont claires (fig. 5.3 et fig. 5.4) : il est plus intéressant d'envoyer un message en un seul transfert qu'en plusieurs transferts successifs.

Évaluation de l'optimisation Pour évaluer l'impact concret de l'agglomération de messages courts réalisé par MADELEINE 4, nous comparons les performances de MADELEINE 3 et de MADELEINE 4 quant à la transmission une série de paquets courts dont le mode de réception est `mad_receive_EXPRESS`. Nous donnons le code du test dans la figure 5.5.

Le mode `mad_receive_EXPRESS` empêche MADELEINE 3 de procéder à l'agrégation des paquets : chaque paquet est envoyé séparément. De son côté, MADELEINE 4 retarde l'émission des paquets quelque soient leurs modes de transmission[2] et envoie par tranche de 32 ko[3]. Ainsi, les paquets sont envoyés en un nombre de transferts beaucoup moins élevé. Cependant, les messages ne sont pas reçus directement : comme l'appel à `mad_unpack` avec pour mode de réception `mad_receive_EXPRESS` est bloquant, les `mad_unpack` des paquets n'ont pas encore été postés, excepté pour le premier paquet. Ainsi, ils passent tous par un état d'attente avant d'être effectivement copiés à leur destination finale. Les figures 5.6 et 5.7 montrent l'impact satisfaisant de cette optimisation.

[2] pour le moment, `mad_receive_CHEAPER` ou `mad_receive_EXPRESS`

[3] c'est-à-dire tant que le groupe reste court

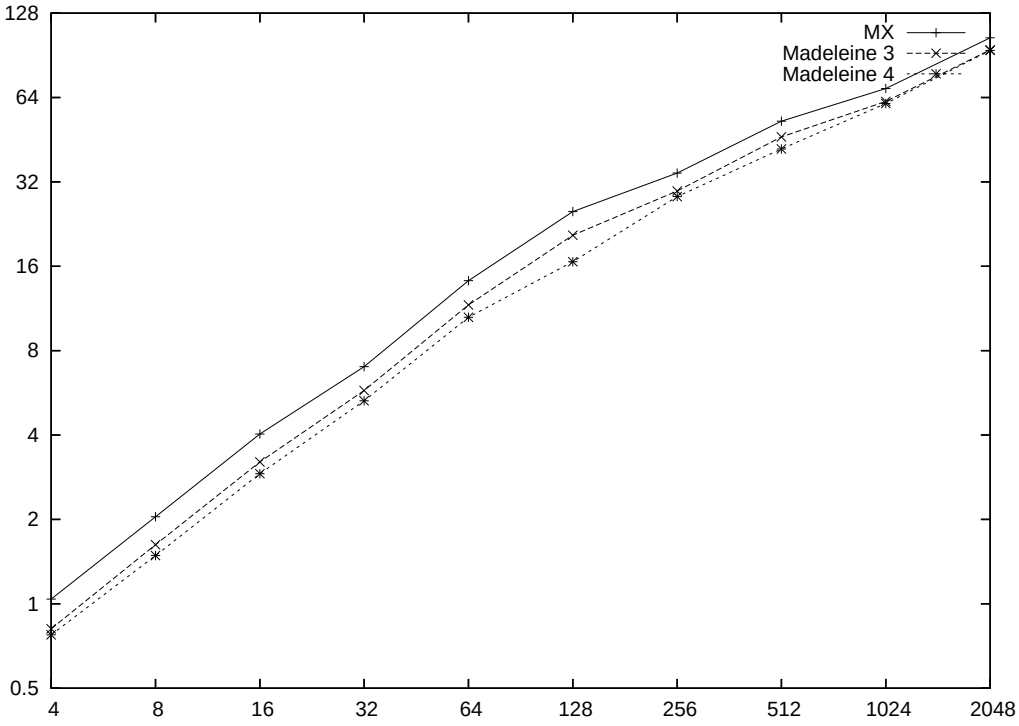

Fig. 5.2 – Ping-pong - débit - MX vs Madeleine 3 vs Madeleine 4.

5.3 Agglomération d'une demande de rendez-vous à un paquet court

L'autre stratégie d'optimisation que nous appliquons aux paquets en attente d'émission est l'agglomération d'une demande de rendez-vous avec un paquet court. Pour mettre en relief son impact sur les transfert, nous envoyons un paquet long de 65 ko puis un paquet court de 4 o. Une ébauche du code de cet exemple est donnée par la figure 5.8. Comme pour les tests précédents, la mesure est réalisée 10 000 fois pour obtenir une moyenne.

	Madeleine 3	Madeleine 4
Latence	310 μs	314 μs
Débit	201 Mo/s	200 Mo/s

Les résultats (voir le tableau de résultats) peuvent sembler décevant. En réalité, il faut savoir que la bibliothèque de communication MX/Myrinet gère implicitement le contrôle de flux. Ainsi, lorsqu'un paquet long est envoyé, la bibliothèque effectue systématiquement une prise de rendez-vous. Il faut noter que pendant le temps qui sépare l'envoi de la demande de rendez-vous et la réception de l'acquittement, la connexion est bloquée. Aucun autre message ne peut circuler. Madeleine 3 profite bien de ce mécanisme. À l'initialisation de Madeleine 3, des connexions sont ouvertes entre chaque module applicatif de chaque machine (fig. 5.9 et 5.10).

Lorsqu'un paquet doit être envoyé, Madeleine 3 commence par réserver la connexion : il ne peut y avoir qu'un paquet transmis sur une connexion. Ainsi, si un paquet long est envoyé alors que le récepteur n'est pas prêt à le recevoir, seul le module est bloqué. Dans Madeleine 4, nous faisons du

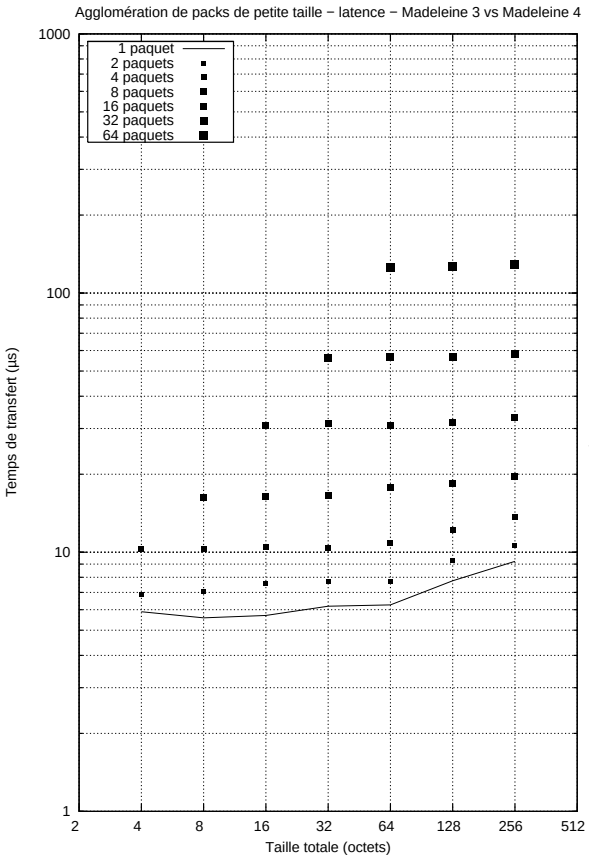

FIG. 5.3 – Subdivision d'un message en un nombre de paquets variable sur MADELEINE 4 - latence.

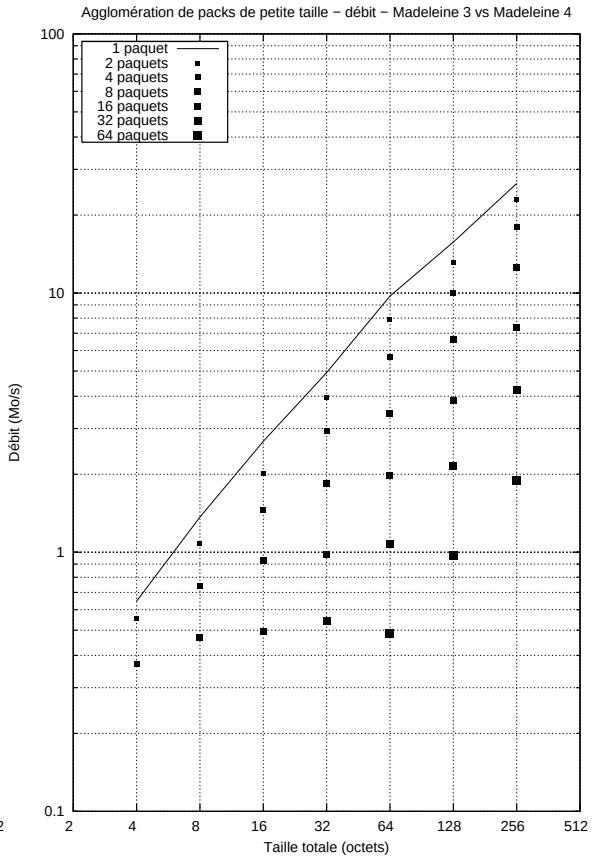

FIG. 5.4 – Subdivision d'un message en un nombre de paquets variable sur MADELEINE 4 - débit.

multiplexage : les communications de plusieurs modules applicatifs circulent en même temps sur une même connexion. Laisser MX assurer le contrôle de flux serait accepter que les modules applicatifs s'inter-bloquent : ce n'est pas la définition du multiplexage. Nous avons donc implémenté notre propre contrôle de flux qui double celui de MX/MYRINET. Malgré cela, nous pouvons constater que les performances entre MADELEINE 3 et MADELEINE 4 restent du même ordre, ce qui est totalement satisfaisant (voir le tableau de résultats). Si nous avions porté la bibliothèque de communication BIP/MYRINET, qui ne gère pas implicitement le contrôle de flux, nous aurions eu des performances beaucoup plus remarquables.

Du côté de l'émetteur

```
(1) connexion = mad_begin_packing(canal, dest);
for(i = 0; i < N; i++){
(2) mad_pack(connexion, paquet_court, taille_paquet_court,
             send_CHEAPER, receive_EXPRESS);
}
(3) mad_end_packing(connexion);

//-----------------------------------------------------------

(1) connexion = mad_begin_unpacking(canal);
for(i = 0; i < N; i++){
(2) mad_unpack(connexion, paquet_court, taille_paquet_court,
               send_CHEAPER, receive_EXPRESS);
}
(3) mad_end_unpacking(connexion);
```

Du côté du récepteur

```
(1) connexion = mad_begin_unpacking(canal);
for(i = 0; i < N; i++){
(2) mad_unpack(connexion, paquet_court, taille_paquet_court,
               send_CHEAPER, receive_EXPRESS);
}
(3) mad_end_unpacking(connexion);

//-----------------------------------------------------------

(1) connexion = mad_begin_packing(canal, dest);
for(i = 0; i < N; i++){
(2) mad_pack(connexion, paquet_court, taille_paquet_court,
             send_CHEAPER, receive_EXPRESS);
}
(3) mad_end_packing(connexion);
```

FIG. 5.5 – Envoi de N paquets consécutifs avec pour mode réception mad_receive_EXPRESS.

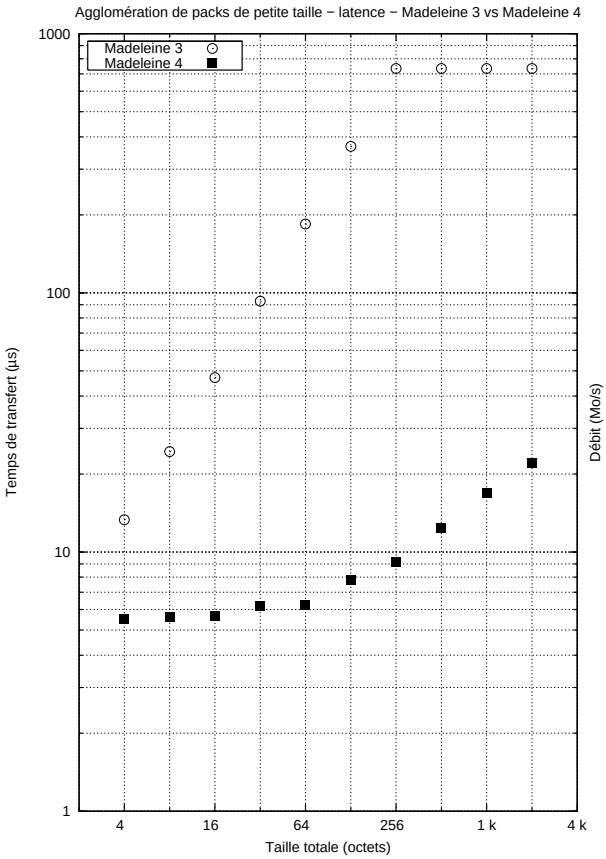

FIG. 5.6 – Envoi de N paquets consécutifs - latence.

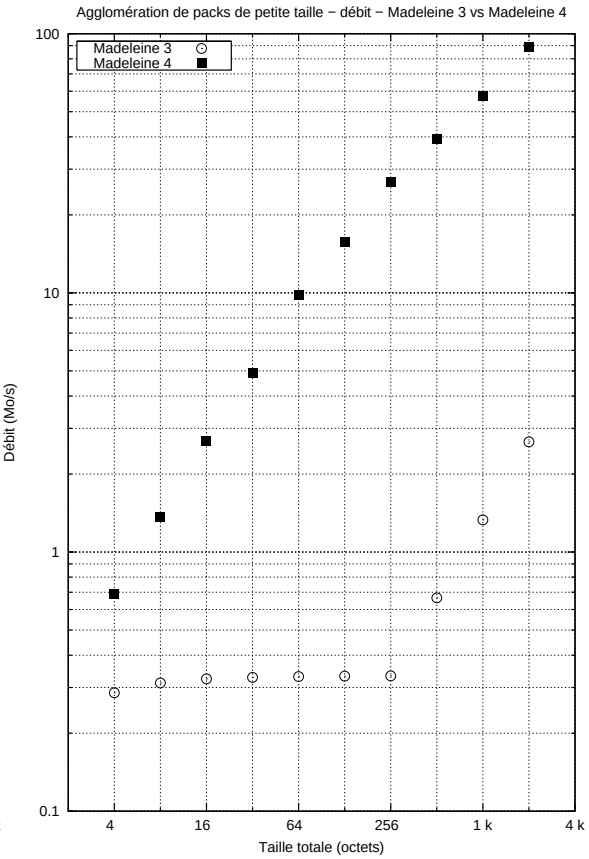

FIG. 5.7 – Envoi de N paquets consécutifs - débit.

Du côté de l'émetteur

```
(1) connexion = mad_begin_packing(canal, dest);
(2) mad_pack(connexion, paquet_court, taille_paquet_court,
             send_CHEAPER, receive_CHEAPER);

(3) mad_pack(connexion, paquet_long, taille_paquet_long,
             send_CHEAPER, receive_CHEAPER);
(4) mad_end_packing(connexion);

//-----------------------------------------------------------

(1) connexion = mad_begin_unpacking(canal);
(2) mad_unpack(connexion, paquet_court, taille_paquet_court,
               send_CHEAPER, receive_CHEAPER);

(3) mad_unpack(connexion, paquet_long, taille_paquet_long,
               send_CHEAPER, receive_CHEAPER);
(4) mad_end_unpacking(connexion);
```

Du côté du récepteur

```
(1) connexion = mad_begin_unpacking(canal);
(2) mad_unpack(connexion, paquet_court, taille_paquet_court,
               send_CHEAPER, receive_CHEAPER);

(3) mad_unpack(connexion, paquet_long, taille_paquet_long,
               send_CHEAPER, receive_CHEAPER);
(4) mad_end_unpacking(connexion);

//-----------------------------------------------------------

(1) connexion = mad_begin_packing(canal, dest);
(2) mad_pack(connexion, paquet_court, taille_paquet_court,
             send_CHEAPER, receive_CHEAPER);

(3) mad_pack(connexion, paquet_long, taille_paquet_long,
             send_CHEAPER, receive_CHEAPER);
(4) mad_end_packing(connexion);
```

FIG. 5.8 – Code du test agglomérant une demande rendez-vous pour un paquet long à un paquet court.

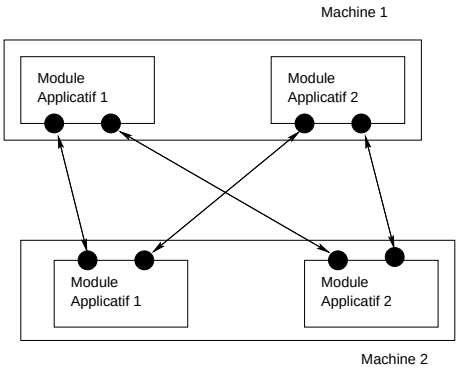

FIG. 5.9 – MADELEINE 3 ouvre des connexions entre chaque module applicatif locaux et chaque module applicatif distant.

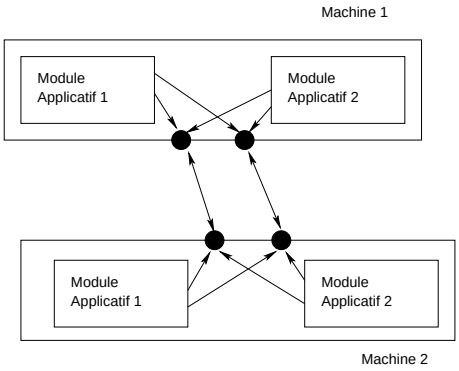

FIG. 5.10 – MADELEINE 4 ouvre des connexions entre chaque machine.

Chapitre 6

Conclusion

6.1 Bilan de notre étude

La conception de bonnes interfaces de communication pour les réseaux rapides est un enjeu important pour bénéficier concrétement, au niveau logiciel, des progrès réalisés au niveau matériel sur les architectures de type grappe. Cependant, les difficultés sont multiples car les performances des réseaux actuels sont telles que le moindre surcoût logiciel est sensible, les technologies existantes sont très différentes les unes des autres (compromis portabilité-performance difficile à trouver), et les contraintes applicatives augmentent sans cesse (les environnements contemporains pour grappes, tels que PadicoTM, ont besoin d'un nombre d'important de canaux de communication indépendants).

La bibliothèque Madeleine 3, qui introduit des mécanismes permettant de dissocier les contraintes de transport des messages de leur acheminement effectif, apporte une solution élégante et prometteuse dans ce domaine : il est possible d'optimiser le transfert des messages en fonction de la technologie réseau grâce à une certaine latitude de fonctionnement consentie par l'application. Toutefois, les stratégies implémentées dans la version courante de Madeleine etaient assez limitées dans leur efficacité, et ce pour plusieurs raisons. D'autre part, une bonne partie de l'efficacité globale de Madeleine 3 reposait sur un protocole interne simpliste, de plus en plus inadapté aux contraintes de multiplexage des environnements d'exécution actuels.

Dans ce travail, nous avons d'abord isolé les points qui empêchent la mise en œuvre d'optimisations plus poussées et nous avons montré que les limitations sont essentiellement dues à l'architecture logicielle actuelle, et aussi à quelques lacunes de l'interface de programmation. Nous avons donc complètement revu l'architecture de Madeleine en repensant le fonctionnement interne sur le principe d'une « imitation » de carte réseau : un peu à la manière d'un ordonnanceur de processus qui, lorsqu'il est invoqué par un processeur, déroule un algorithme permettant de choisir le meilleur processus prêt possible, la bibliothèque Madeleine 4, sollicitée lors des périodes d'inactivité des cartes réseaux, déroule un algorithme permettant de choisir la meilleure combinaison de paquets prêts à être transférés sur le réseau.

L'optimisation de paquets étant un problème très difficile (collisions entre plusieurs stratégie, difficulté de l'évaluation empirique), nous avons choisi de rendre la bibliothèque la plus évolutive possible, en rendant possible la définition et l'ajout dynamique de stratégies d'optimisation génériques indépendantes des technologies réseau. Il est donc possible d'écrire des fonctions d'optimisation de paquets (inversion, regroupement, contrôle de flux, etc.) en C en utilisant simplement des outils de manipulation de listes et des fonctions permettant de vérifier que les combinaisons de paquets

envisagées sont valides du point de vue de l'application.

Nous avons également enrichi l'interface, notamment pour préparer MADELEINE à pouvoir être « renseignée » par un compilateur (ou un préprocesseur) qui effectuerait une analyse permettant de déterminer, dans la plupart des cas, quel est l'agencement entre les différentes soumissions des segments d'un message. Dans bien des cas, cela permet d'améliorer grandement le comportement des heuristiques utilisées. Dans l'état actuel, ces renseignements sont fournis « à la main » par le programmeur.

Enfin, nous avons implémenté l'architecture proposée au-dessus du protocole MX/Myrinet (pas encore sorti officiellement pendant ce stage) et nous avons conduit plusieurs micro-expérimentations permettant de vérifier que l'architecture est viable et que les choix pris sont judicieux.

6.2 Perspectives de recherche

À court terme, il s'agit de consolider l'implémentation sur MX (full multithreading), de porter la bibliothèque sur d'autres réseaux (notamment Quadrics), d'améliorer la facilité d'écriture des stratégies d'optimisation et de fournir des mécanismes de debug...

À moyen terme, nous comptons expérimenter l'exploitation de grappes multi-rails en utilisant notre bibliothèque, c'est-à-dire la possibilité d'équilibrer les communications sur plusieurs réseaux physiques (différents ou pas) pour agréger de la bande passante. Notre architecture est prévue pour cela.

À plus long terme, c'est bien entendu sur les optimisations qu'il est nécessaire de conduire des travaux approfondis :

1. Le gain des stratégies doit pouvoir être calculé en même temps que l'ordonnancement des paquets. Il faut pour cela que les réseaux soit correctement échantillonnées à l'initialisation (phase de calibrage).
2. L'équipe a demontré des résultats de polynomialité sur certaines stratégies individuelles d'optimisation (groupement des petits avec tampons infinis), il faut regarder le cas des buffers finis.
3. Les stratégies peuvent interférer les unes avec les autres, c'est-à-dire qu'utiliser la stratégie 1 en première passe puis la stratégie 2 ne donne pas les mêmes gains que si l'on inverse l'ordre, ou que si l'on les fait progresser de manière entrelacée. Le problème général semble très complexe. Nous comptons utiliser des heuristiques pour trouver comment choisir les bonnes combinaisons de stratégies.

Bibliographie

[1] InfiniBand Trade Association. Infiniband technology overview, 2005. http ://www.infinibandta.org/.

[2] Olivier Aumage. *Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes.* Thèse de doctorat, spécialité informatique, École normale supérieure de Lyon, September 2002.

[3] D. Culler, K. Keeton, L.T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. The generic active message interface specification, 1994. http ://now.cs.berkeley.edu/.

[4] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The virtual interface architecture. *IEEE Micro*, 18(2) :66–76, 1998.

[5] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1) :70–82, 1996.

[6] IEEE. *Standard for Scalable Coherent Interface (SCI)*, août 1993. Standard IEEE numéro 1596.

[7] Dolphin Interconnect. Sisci documentation and library. http ://www.dolphinics.no.

[8] J.Hennessy and D. Patterson. Computer architecture : A quantitative approach - third edition. Morgan Kaufmann Publishers, 2003.

[9] Quadrics Ltd. Elan programming manual, 2003. http ://www.quadrics.com/.

[10] Mark W. MacBeth, Keith A. McGuigan, and Philip J. Hatcher. Executing java threads in parallel in a distributed-memory environment. In *CASCON '98 : Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 16. IBM Press, 1998.

[11] Glen's Message. http ://www.myricom.com/scs/.

[12] Message Passing Interface Forum. MPI : A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[13] MPICH-GM. http ://www.myricom.com/scs/.

[14] Inc. Myricom. Myrinet express (mx) : A high performance, low-level, message-passing interface for Myrinet. 2003.

[15] Padico. Padico : a software environment for computational grids. http ://www.irisa.fr/paris/Padico/.

[16] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages : Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2) :60–73, /1997.

[17] T. PM^2 Team. Getting started with PM^2. http ://www.pm2.org.

[18] L. Prylli and B. Tourancheau. BIP : A new protocol designed for high-performance networking on myrinet. 1998.

[19] T. Ruhl, H. Bal, R. Bhoudjang, K. Langendoen, and G. Benson. Experience with a portability layer for implementing parallel programming systems, 1996.

[20] SCI-MPICH. http ://www.lfbs.rwth-aachen.de/ joachim/sci-mpich/.

[21] Top500 Supercomputer Sites. 24th top500 list released, 2004. http ://www.top500.org/.

[22] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system : Evolution, experiences, and trends. *Parallel Computing*, 20(4) :531–545, 1994.

[23] Draft Comments Welcome. U-net : A user-level network interface for parallel and distributed computing.