Université Paris VI – Pierre & Marie Curie

THESE DE DOCTORAT

Spécialité : **Informatique**

Présentée par **Olivier GLÜCK**

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE Paris VI

Optimisations de la bibliothèque de communication MPI pour machines parallèles de type « grappe de PCs » sur une primitive d'écriture distante

---

Soutenue publiquement le 12 juillet 2002,
devant le jury composé de :

M. Bernard LECUSSAN	**Rapporteur**
M. Loïc PRYLLI	**Rapporteur**
M. Jean-Marie CHESNEAUX	**Examinateur**
M. Paul FEAUTRIER	**Examinateur**
M. Claude GIRAULT	**Examinateur**
M. Daniel MILLOT	**Examinateur**
M. Alain GREINER	**Directeur de thèse**

A Marc-André GLÜCK

Olivier.Gluck@lip6.fr
http://mpc.lip6.fr

Avant-propos et remerciements

Le travail présenté dans cette thèse a permis des collaborations avec différents laboratoires de recherche et différentes équipes : le PRiSM de l'Université de Versailles, le LIP de l'ENS Lyon, le Laria de l'Université de Picardie, le département informatique de l'INT Evry, le CERT de l'Office National d'Etudes et de Recherches Aérospatiales de Toulouse et l'équipe ANP du LIP6.

Je tiens, en premier lieu, à remercier tout particulièrement Alain Greiner, Professeur de l'Université Paris VI et directeur du département ASIM du LIP6, qui a dirigé mes recherches. J'ai à cœur de lui témoigner toute ma gratitude, non seulement car il a toujours été disponible pour m'accompagner dans mes travaux, mais aussi pour sa relecture très attentive de ce manuscrit. Ses conseils avisés, sa clairvoyance, son suivi constant de mes recherches, ses qualités d'encadrement et son soutien m'ont été d'une aide plus que précieuse.

J'adresse mes remerciements à Bernard Lecussan et Loïc Prylli qui m'ont fait l'honneur d'être les rapporteurs de ma thèse, ainsi qu'à l'ensemble des membres du jury.

Je remercie l'ensemble des membres du laboratoire ASIM qui m'ont très bien accueilli et avec qui j'ai eu des échanges fructueux. J'ai eu un grand plaisir à travailler avec l'ensemble des membres de l'équipe MPC qui m'ont tous aidé à un certain moment dans l'avancement de mes travaux. Je remercie en particulier Cyril Spasevski, Amal Zerrouki, Fabricio Silva et surtout, Franck Wajsbürt, Jean-Lou Desbarbieux et Alexandre Fenÿo qui ont toujours été disponibles et qui m'ont été d'une aide précieuse. J'ai d'ailleurs une pensée toute particulière pour Alexandre qui a réalisé les couches logicielles bas-niveau de la machine MPC, et sans qui cette thèse n'aurait sans doute pas pu se faire. Il a toujours su me conseiller dans les différents choix que j'ai eu à faire. Ses compétences techniques en matière de programmation système et sa connaissance des systèmes d'exploitation UNIX m'ont été d'une grande utilité.

J'ai encadré plusieurs étudiants de DEA ou de l'INT dans le cadre de leur stage de fin d'études d'ingénieur. Ils m'ont permis d'avancer plus rapidement dans mes travaux. Je remercie donc Emmanuel Dreyfus qui a travaillé sur le portage sous LINUX de la couche PUT, Laurent Valeyre qui a travaillé sur la redistribution de la mémoire d'un processus, Djilali Abdellou qui a travaillé sur la couche PUT en mode utilisateur, et Laurent Lechevallier qui a exécuté des applications MPI sur la machine MPC.

Mes travaux n'auraient pas pu voir le jour en dehors du projet MPC. Je remercie pour cela les membres des différentes équipes de recherche qui ont suivi les évolutions de mes travaux. En particulier, j'exprime ma gratitude à Philippe Lalevée et Daniel Millot qui ont permis mon intégration au sein du projet MPC. Je remercie très sincèrement Jean-Luc Lamotte de l'équipe ANP du LIP6 avec qui j'ai eu un grand plaisir à travailler.

Je remercie également Loïc Prylli, Roland Westrelein, Bernard Tourancheau, et Raymond Namyst qui m'ont accueilli à l'ENS Lyon au début de mes travaux et qui m'ont permis de démarrer mes travaux dans la bonne direction.

Enfin, j'ai une pensée particulière pour Marie-Martine Glück qui a relu attentivement ce manuscrit et pour Elodie Sanchez qui m'a soutenu pendant ces trois années.

Résumé

Le travail présenté dans cette thèse s'inscrit dans le cadre du projet de recherche MPC (Multi-PC) démarré en 1995 à l'Université Pierre et Marie Curie. Le but de ce projet est la réalisation d'une machine parallèle à faible coût. Les nœuds de calcul sont des PC standard achetés dans le commerce auxquels s'ajoutent des composantes aussi bien matérielles que logicielles réalisées dans le cadre du projet.

Cette thèse présente des optimisations de la bibliothèque de communication MPI pour machines parallèles de type « grappe de PCs », disposant d'un réseau de communication qui fournit une primitive d'écriture en mémoire distante (*Remote DMA*). Ce mécanisme de communication est implanté de manière très efficace au niveau matériel. Notre objectif est de faire bénéficier les applications de la très faible latence matérielle de ce réseau spécifique, en minimisant le temps de traversée des couches logicielles qui sépare l'appel à une primitive de communication au niveau applicatif, de la prise en compte du transfert par le matériel réseau. Ce manuscrit de thèse présente, dans ce cadre, une implémentation optimisée de MPI au-dessus d'une primitive d'écriture distante. La machine MPC du Laboratoire d'Informatique de Paris VI constitue notre plate-forme expérimentale mais nous avons construit nos couches de communication au-dessus d'une API (*Applications Programming Interface*) générique d'écriture en mémoire distante permettant de porter facilement notre implémentation de MPI sur n'importe quelle plate-forme matérielle disposant d'une primitive d'écriture en mémoire distante.

Nous étudions l'impact de divers facteurs sur les performances obtenues au niveau applicatif et nous proposons des solutions optimisées pour implanter l'environnement de programmation parallèle MPI sur la primitive d'écriture distante. Plus particulièrement, nous décrivons des mécanismes pour réaliser les communications en mode utilisateur en éliminant les appels système et la signalisation par interruption matérielle. Un inconvénient de la primitive d'écriture en mémoire distante est qu'elle utilise des adresses physiques pour réaliser ses transferts : le contrôleur réseau accède directement à la mémoire physique (DMA) sur le nœud émetteur et sur le nœud récepteur pour transférer les données. Les principales difficultés concernent le partage des ressources réseau entre plusieurs processus utilisateur et le problème des conversions d'adresses. Nous proposons une solution pour réduire au maximum le coût de traduction des adresses virtuelles fournies par l'application en adresses physiques utilisables par le contrôleur réseau.

Mots clés : **machine parallèle, grappes de PCs, bibliothèque de communication, passage de messages, MPI, écriture distante, DMA, communication en mode utilisateur, gestion mémoire, adresse virtuelle/physique, traduction d'adresse**.

Optimizations of the Message Passing Interface communication library for PCs clusters using a remote write communication primitive

Abstract

This Ph.D Thesis is a part of the MPC (Multi-PC) research project started in 1995 at Pierre et Marie Curie University, Paris. The goal was to design a low cost and high performance parallel computer. The MPC parallel computer consists of several processing nodes interconnected by a gigabit High Speed Link network.

This work presents how the Message Passing Interface (MPI) communication library can be optimized for a parallel computer made of clusters of workstations, providing a remote-write communication primitive. From the hardware point of view, this communication mechanism is very efficient. Our goal is to minimize the overhead of communication software layers used by applications for accessing the high speed network. This thesis focuses on an efficient and optimized implementation of MPI built on a simple Remote Direct Memory Access hardware primitive. For experimental purposes, the MPC parallel computer of LIP6 laboratory was used. However, our communication software layers were built over a generic remote-write API in order to port easily our MPI implementation on every hardware platform using a remote write primitive.

We study the impact of several factors on application performances and we propose efficient mechanisms to implement the Message Passing Interface on a remote DMA communication primitive. Precisely, we describe solutions to eliminate system calls and interrupts during communications. A drawback of the remote-write primitive is that it uses physical memory addresses for sending data: the network controller accesses directly the host memory on the sender node and the receiver node. The major difficulty of this work deals with the user-level accesses to the network interface by several processes and the address translations. We propose a mechanism to significantly reduce the overhead due to the translations of virtual addresses supplied by the applications in physical addresses used by the network controller.

Keywords : **parallel computer, PCs clusters, communication library, message passing, MPI, remote write, Direct Memory Access, user-level communication, memory management, virtual/physical address, address translation.**

Table des matières

Table des figures

Table des tableaux

Chapitre I : Introduction

Sommaire

Le travail présenté dans cette thèse s'inscrit dans le cadre du projet de recherche MPC (Multi-PC) [MPC] démarré en 1995 à l'Université Pierre et Marie Curie. Le but de ce projet est la réalisation d'une machine parallèle à faible coût. Les nœuds de calcul sont des PCs standard achetés dans le commerce auxquels s'ajoutent des composantes aussi bien matérielles que logicielles réalisées dans le cadre du projet. De nombreuses équipes participent au groupe de recherche MPC. On peut citer en particulier des équipes issues du Laboratoire d'Informatique de Paris 6, de l'Ecole Nationale Supérieure des Télécommunications, de l'Institut National des Télécommunications d'Evry, du PRISM de l'Université de Versailles, du LAboratoire de Recherche en Informatique d'Amiens de l'université de Picardie Jules Verne et des équipes de l'Université de Toulouse (CERT/IRIT).

I.1. Un parallélisme nécessaire

Depuis les débuts de l'informatique, les besoins en puissance de calcul ne cessent d'augmenter. Beaucoup d'applications (par exemple dans les domaines des prévisions météorologiques, de la circulation des océans, de la dynamique des fluides, de la modélisation des semi-conducteurs et supraconducteurs, du traitement d'images) ont besoin de plus de puissance de calcul qu'une machine séquentielle ne peut fournir.

Par ailleurs, les applications ont besoin d'être exécutées en un temps toujours plus court. C'est pourquoi la nécessité d'accroître les performances matérielles et logicielles constitue un problème primordial. C'est même l'un des enjeux les plus importants de l'informatique du futur.

Une solution à ces deux problèmes est d'améliorer les performances des processeurs et d'autres composants matériels comme par exemple les temps d'accès à la mémoire. En 1965, Gordon Moore [Moore,1965] constata en traçant la courbe de croissance des performances des microprocesseurs, que chaque génération de puce était deux fois plus puissante pour un délai de développement variant entre 18 et 24 mois. Aujourd'hui, les performances des microprocesseurs ont un accroissement de 55% par an en terme de puissance de calcul, de 25% en terme de fréquence de fonctionnement et de 35% en terme de densité d'intégration.

Cependant, cela ne suffit pas. Une solution alternative, pour réduire les temps de calcul est de faire travailler plusieurs processeurs ensemble et de coordonner leurs efforts de calcul pour résoudre un même problème. C'est ainsi que sont apparues les machines parallèles. G. Pfister [Pfister,1998] fait remarquer qu'il y a trois façons d'améliorer les performances de travail d'un homme :

- ✓ Travailler plus dur (« work harder »)
- ✓ Travailler plus efficacement (« work smarter »)
- ✓ Demander de l'aide à autrui (« get help »)

En terme de calcul scientifique, l'analogie est la suivante :

- ✓ « travailler plus dur » signifie : utiliser du matériel qui va plus vite (par exemple en réduisant le temps de cycle).
- ✓ « travailler plus efficacement » signifie : optimiser les algorithmes.
- ✓ « Demander de l'aide » signifie : utiliser plusieurs machines pour résoudre un même problème.

Les techniques de calcul parallèle arrivent à maturité et commencent à être exploitées commercialement grâce au développement d'outils et d'environnements de programmation parallèle et surtout, du fait des progrès stupéfiants réalisés dans les réseaux haut débit. La Figure I-1 montre l'évolution dans le temps du calcul séquentiel et parallèle.

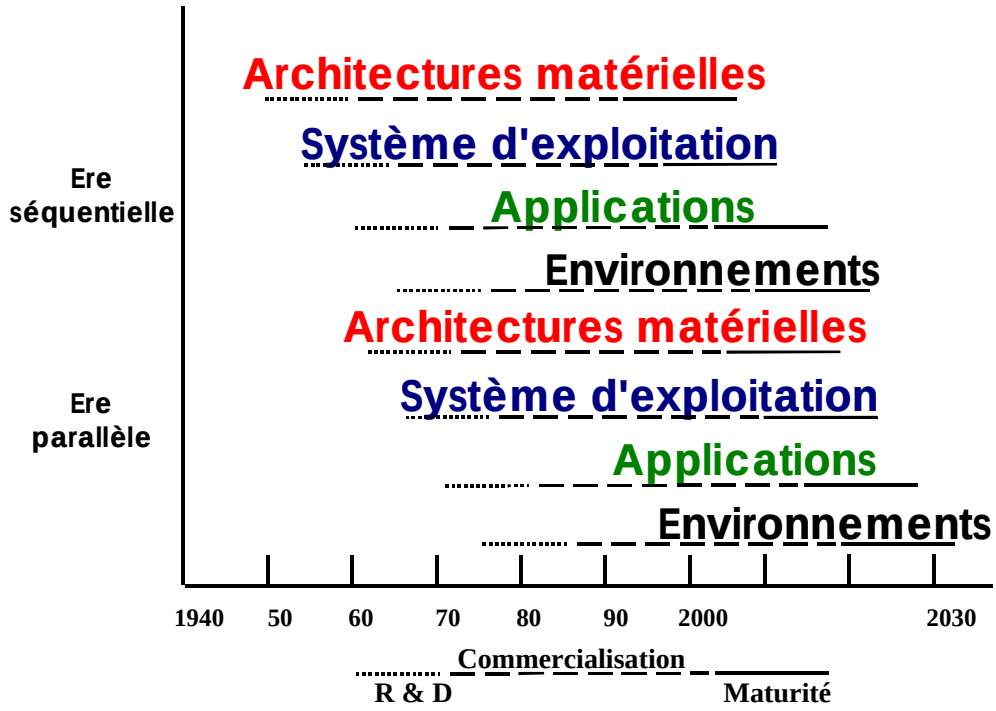


Figure I-1 : Evolution des machines séquentielles et parallèles

Les technologies du calcul parallèle ne sont pas encore arrivées à une maturité suffisante pour être pleinement exploitées. De gros progrès doivent encore être faits, en particulier dans le domaine des environnements de programmation.

I.2. Des architectures parallèles variées

Il existe différentes classifications des machines parallèles. La première a été faite par Flynn en 1972 [Flynn,1972]. Elle se base sur le flux de données et le flux d'instructions de la machine :

- ✓ **SISD**: *Single Instruction Single Data*
  Il s'agit d'une machine traditionnelle qui exécute une seule instruction à la fois sur un seul ensemble de valeurs.
- ✓ **SIMD**: *Single Instruction Multiple Data*
  Il s'agit des machines vectorielles qui peuvent par exemple, additionner en parallèle des éléments de 2 vecteurs différents. Nous retrouvons dans cette catégorie les machines vectorielles de CRAY Systems [CRAY] ou les machines CM de Thinking Machines [CM].
- ✓ **MISD**: *Multiple Instruction Single Data*
  Plusieurs processeurs exécutent des instructions sur un même flux de données. Cette configuration existe très peu en pratique.
- ✓ **MIMD**: *Multiple Instruction Multiple Data*
  Plusieurs processeurs exécutent simultanément des instructions sur des données qui leur sont propres. Cette architecture est la plus couramment utilisée. Elle regroupe en particulier les machines à mémoire partagée et les machines à mémoire distribuée.

Dans cette dernière catégorie, nous nous intéressons aux *clusters* ou grappes de stations de travail *(Network Of Workstations)*. Il s'agit d'un ensemble de PCs ou de stations de travail individuelles interconnectés par un réseau standard (Ethernet, ATM) ou par un réseau rapide dédié (SCI, Myrinet).

[Pfister,1998] et [Hwang,1998] donnent une description détaillée des architectures existantes. Dans ce manuscrit, nous nous focalisons sur les grappes de PCs utilisant un réseau dédié.

I.3. Enjeux et objectifs

Les travaux présentés dans ce manuscrit s'inscrivent dans le cadre des machines parallèles de type « grappe de PCs », disposant d'un réseau de communication optimisé qui fournit une primitive d'écriture en mémoire distante (*Remote DMA*).

Ce type de réseaux fournit un mécanisme de communication extrêmement efficace au niveau matériel : la latence matérielle d'un transfert est généralement de quelques micro-secondes et la bande passante du réseau atteint bien souvent (ou même dépasse) le Gigabits par seconde. L'inconvénient de la primitive d'écriture en mémoire distante est qu'elle utilise des adresses physiques pour réaliser ses transferts : le contrôleur réseau accède directement à la mémoire physique (DMA) sur le nœud émetteur et sur le nœud récepteur pour transférer les données. Au niveau applicatif, le programmeur manipule des adresses virtuelles, et préfère utiliser des communications canalisées.

La difficulté est donc de faire bénéficier les applications des très bonnes performances du réseau en minimisant le temps de traversée des couches logicielles qui sépare l'appel à une primitive de communication au niveau applicatif, de la prise en compte du transfert par le matériel réseau.

La bibliothèque de communication MPI s'est imposée ces dernières années comme le standard de programmation parallèle à passage de messages utilisé par les applications. Notre objectif est d'optimiser la bibliothèque MPI pour les machines parallèles de type « grappe de PCs » disposant d'une primitive d'écriture en mémoire distante. Nous étudions dans ce manuscrit l'impact de divers facteurs sur les performances obtenues au niveau applicatif et nous proposons des solutions optimisées pour implanter MPI sur la primitive d'écriture distante.

Pour cela, nous avons défini une API (*Applications Programming Interface*) générique d'écriture en mémoire distante permettant de porter facilement notre implémentation de MPI sur n'importe quelle plate-forme matérielle disposant d'une primitive d'écriture en mémoire distante. Il s'agit ainsi de formaliser les services fournis par la primitive d'écriture distante pour masquer les particularités des réseaux utilisant cette technique.

Notre plate-forme expérimentale est la machine MPC [Potter,1996] [Greiner,1998] [Zerrouki,2000] [Glück,2001] du LIP6. Il s'agit d'une grappe de PCs équipée d'un réseau haut débit fournissant une primitive d'écriture en mémoire distante. L'environnement de calcul matériel, le réseau de contrôle et le système d'exploitation sont des composants standard. Le réseau Gigabit et les couches de communications rapides ont été développées par le groupe de recherche MPC. L'architecture de la machine MPC est présentée à la section III.1.4.

L'objectif général de nos travaux peut se résumer de la façon suivante :

Des machines parallèles de type « grappe de PCs » fournissent un mécanisme de communication extrêmement efficace d'écriture en mémoire distante qui peut être assimilé à un *Remote DMA*. Faire bénéficier les applications de la très faible latence matérielle de ce réseau spécifique en réduisant au maximum le coût de traversée des couches logicielles de communication constitue le centre de notre propos. Ce manuscrit de thèse présente, dans ce cadre, une implémentation optimisée de l'environnement de programmation parallèle MPI au-dessus d'une primitive d'écriture distante, telle que celle de la machine MPC qui constitue notre plate-forme expérimentale.

I.4. Organisation du manuscrit

Les travaux présentés dans ce manuscrit s'organisent de la façon suivante :

- ✓ **Chapitre II, Problématique**. Le matériel réseau fournit une primitive d'écriture en mémoire distante. Nous étudions les caractéristiques d'une écriture distante et nous analysons les différents problèmes auxquels nous sommes confrontés pour implanter efficacement la bibliothèque de communication MPI sur cette primitive de communication spécifique.

- ✓ **Chapitre III, Etat de l'art**. Des problèmes analogues à ceux que nous avons rencontrés se sont présentés lors de la conception de certaines couches de communication bas niveau et haut niveau. Nous étudions dans ce chapitre les méthodes de résolution proposées.
- ✓ **Chapitre IV, Architecture de MPI sur une primitive d'écriture distante**. Nous présentons dans ce chapitre une première implémentation de MPI sur la primitive d'écriture distante : MPI-MPC1. Nous définissons une API générique d'écriture en mémoire distante, appelée RDMA, qui sert de brique de base sur laquelle notre implémentation de MPI s'appuie et, qui fournit une abstraction suffisante des caractéristiques spécifiques du réseau réalisant la primitive d'écriture distante. Après l'étude des services que nous devons fournir au niveau applicatif, nous proposons deux modes de transfert des données de l'application. Nous décrivons comment faire remonter les informations de signalisation des événements réseau au niveau MPI. Nous évaluons les performances en termes de débit et de latence à l'aide d'un « *ping-pong* » MPI, sur notre plate-forme expérimentale : la machine MPC. L'analyse de ces performances montre que les limites de cette première implémentation sont liées au fait que la primitive d'écriture distante se trouve dans le système d'exploitation et que la signalisation est réalisée par interruptions matérielles provenant du contrôleur réseau.
- ✓ **Chapitre V, Optimisations des couches basses de communication**. Ce chapitre propose une deuxième implémentation de MPI (MPI-MPC2) qui utilise une primitive d'écriture distante en mode utilisateur et une signalisation par scrutation des ressources réseau. Nous proposons des mécanismes génériques permettant le partage des ressources réseau entre plusieurs processus de l'application et nous décrivons comment nous avons appliqué ces solutions à la primitive d'écriture distante de la machine MPC. Nous analysons les problèmes liés à la scrutation des ressources réseau. Nous réalisons de nouvelles mesures de performance avec MPI-MPC2. En comparant celles-ci à celles obtenues avec MPI-MPC1, nous constatons que la discontinuité des tampons de l'application en mémoire physique est pénalisante pour le transfert des messages de grande taille.
- ✓ **Chapitre VI, Optimisation des traductions d'adresse par redistribution de la mémoire**. Nous proposons dans ce chapitre une méthode originale permettant de se ramener à une situation dans laquelle les régions mémoire de chaque processus de l'application correspondent à des zones de mémoire physique contiguës. L'intérêt de la solution proposée est qu'elle n'entraîne aucune modification non seulement, du système d'exploitation et de la librairie C, mais surtout de l'application. Nous réalisons des mesures de performances avec cette troisième implémentation de MPI (MPI-MPC3), analogues à celles des chapitres IV et V, pour étudier l'impact des mécanismes proposés sur les performances.
- ✓ **Chapitre VII, Résultats expérimentaux sur des applications réelles**. Nous réalisons dans ce chapitre des mesures de performances sur des applications réelles (avec les trois implémentations de MPI décrites aux chapitres IV, V, et VI). On cherche ici à évaluer si les conclusions générales qui ont pu être tirées de la comparaison des trois implémentations de MPI dans le cas du ping-pong restent valables dans le cas d'une application réelle.

- ✓ **Chapitre VIII, Conclusions et perspectives**. Nous résumons dans ce chapitre les résultats obtenus dans ce manuscrit et les limites des solutions que nous avons proposées. Enfin, nous discutons de quelques perspectives ouvrant de nouveaux horizons de recherche.

Chapitre II : Problématique

Sommaire

La primitive d'écriture distante étant l'interface de communication dont nous disposons au niveau matériel, nous étudions dans ce chapitre les difficultés rencontrées pour faire bénéficier le niveau applicatif des très bonnes performances de ce réseau rapide.

II.1. Présentation du problème

Nous pouvons distinguer deux principales raisons qui expliquent l'émergence ces dernières années des réseaux de stations ou clusters dans le monde des architectures parallèles. La première est économique. Le volume de production influençant directement le coût final, les éléments qui composent les stations de travail sont beaucoup moins coûteux que ceux qui équipent les supercalculateurs. La deuxième raison est l'apparition récente de matériel standard dans le domaine des réseaux rapides ayant des performances très satisfaisantes comparativement aux réseaux spécifiques des supercalculateurs.

II.1.1. Un besoin des applications

La disponibilité de réseaux de communication rapides pour interconnecter les différents nœuds de calcul d'un cluster a élargi sensiblement leurs domaines d'utilisation. Le matériel des réseaux de stations et celui des architectures concurrentes (calculateur ou serveur) convergent. Toutefois, les clusters doivent répondre à plusieurs attentes. Tout d'abord, il faut pouvoir disposer de logiciels, tels que environnements de programmation et systèmes d'exploitation, d'un niveau de performance et de fiabilité comparable à celui des architectures concurrentes. Ensuite, les applications nécessitent, d'une part de bonnes performances en terme de puissance de calcul (calcul flottant par exemple) et, d'autre part de communications très performantes entre les différents nœuds de calcul composant le réseau de stations. C'est ce deuxième point qui va désormais nous intéresser.

En effet, les communications ont une influence significative sur les performances globales dans les applications parallèles. La Figure II-1 représente deux processus d'une même application parallèle qui ont besoin de s'échanger des données. Les processus P1 et P2 s'exécutent dans l'espace utilisateur. P1 se trouve sur un nœud de calcul et P2 sur un autre nœud. P1 envoie des données à P2. Pour ce faire, il est nécessaire de traverser les couches de communication pour accéder aux composants matériels du réseau sous-jacent, du côté récepteur comme du côté émetteur. Ces couches de communication peuvent se trouver en partie dans le système d'exploitation et en partie dans l'espace utilisateur. Le matériel est schématisé par deux contrôleurs réseaux (*Network Interface Controler*) NIC1 et NIC2 par souci de simplicité.

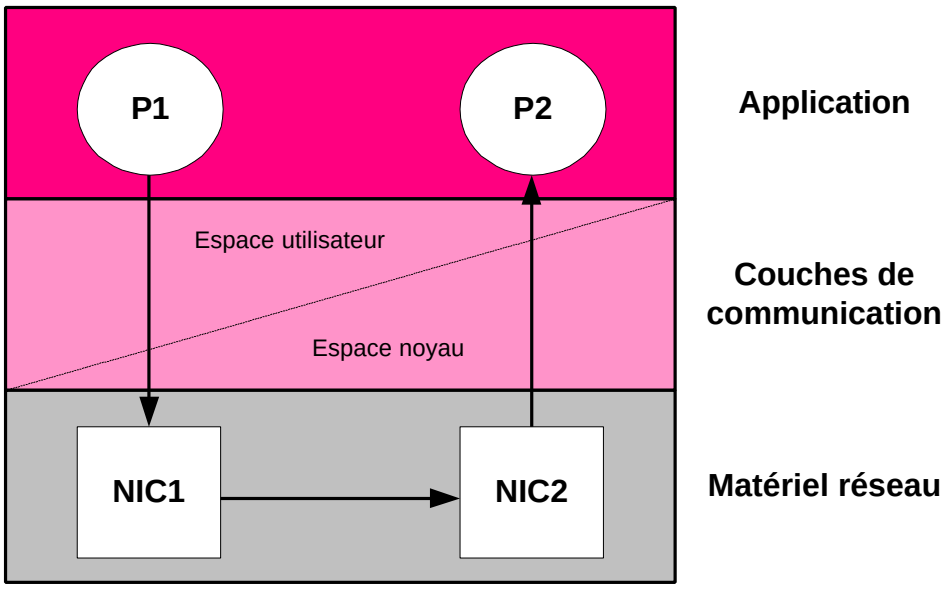


Figure II-1 : Les éléments intervenant dans une communication

La nécessité de communications performantes se traduit non seulement par des réseaux rapides d'un point de vue matériel mais aussi par des couches de communication les plus efficaces possibles, c'est-à-dire introduisant un surcoût minimum dans le temps de transfert d'un message entre les différents processus d'une application.

II.1.2. Des réseaux de plus en plus rapides

Des progrès considérables dans les réseaux rapides ont été faits ces dernières années. Nous pouvons citer à titre d'exemples ATM [ATM,1995], SCI [SCI] [Gustavson,1992], Myrinet [Boden,1995], HSL [Reibaldi,1997] [HSL,1994], etc. Ils permettent des communications point à point jusqu'à 3Gbit/s avec une latence matérielle de l'ordre de la micro-seconde. Le Tableau II-1 présente un ordre de grandeur des performances « brutes » des réseaux SCI, Myrinet et HSL en termes de latence et de débit.

	SCI	Myrinet	HSL
Latence	2µs	3µs	2µs
Débit max.	3 Gbits/s	2 Gbits/s	1 Gbits/s

Tableau II-1 : Performances de réseaux rapides

De ce fait, le temps d'accès à de la mémoire distante est de l'ordre de la micro-seconde alors que celui relatif à la mémoire cache est de l'ordre de la nano-seconde, celui de la mémoire locale de la cinquantaine de nano-secondes, celui d'un disque local étant de l'ordre de la milli-seconde. Ce type de réseaux modifie la hiérarchie des performances d'accès aux données en introduisant potentiellement les mémoires distantes entre la

mémoire locale et le disque. Dans le même sens, les disques distants deviennent accessibles avec un coût voisin de celui du disque local.

Ces performances sont du même ordre que celles des réseaux de calculateurs parallèles (l'IBM SP2 [Agerwala,1995] par exemple et son réseau HPS [Stunkel,1995]). Dès lors, avec des composants aux performances similaires, ce sont essentiellement les couches de communications logicielles qui différencieront les machines parallèles classiques des clusters.

II.1.3. Les couches logicielles de communication

Les performances de communication des architectures parallèles dépendent des différents niveaux de la hiérarchie qui séparent un processus utilisateur d'une application parallèle du matériel de communication (Figure II-1). Une architecture parallèle comporte quatre ressources principales :

- ✓ Les processeurs.
- ✓ La mémoire.
- ✓ Les entrées/sorties.
- ✓ Le réseau de communication interne.

L'évolution des trois premières est essentiellement guidée par la technologie qui permet à la fois une augmentation de la vitesse, des fonctionnalités et de la capacité (largeur des mots, tailles des mémoires intermédiaires, etc.) grâce à l'augmentation de la densité d'intégration. Ce n'est pas encore le cas des réseaux locaux rapides ou des réseaux internes des architectures parallèles classiques pour lesquels le temps passé en dehors du matériel du réseau pour communiquer domine largement. Comme à la fois le processeur et le matériel réseau sont fixés, **la problématique des communications concerne surtout l'interface logicielle entre les applications et le réseau**. L'introduction de réseaux rapides ne peut suffire à réduire la latence de communication et le débit atteint est bien souvent très inférieur aux possibilités du matériel.

Pour les applications numériques parallèles, les besoins en communication ont été étudiés dans [Cypher,1993]. Les conclusions sont que 48% des messages ont une taille inférieure à 16 octets et que 80% des données sont transférées dans des messages de plus de 8Ko. Ces valeurs indiquent clairement que les mécanismes de communication doivent être améliorés non seulement en terme de latence mais aussi en terme de débit. [Martin,1997] a apporté des précisions supplémentaires en observant sur une dizaine d'applications parallèles la contribution des paramètres du surcoût logiciel, de la latence matérielle et de délai inter-émission sur la performance globale. Les résultats montrent que toutes les applications étudiées présentent une dégradation de performance linéaire suivant le surcoût logiciel.

Une nécessité est donc de réduire autant que possible le chemin critique logiciel que constitue la traversée de la pile de protocoles qui sépare l'appel à une fonction de communication par un processus utilisateur de la prise en compte du message par le matériel du réseau. La problématique générale de nos travaux est de réduire ce chemin critique logiciel dans un cadre bien précis qui est défini dans le paragraphe suivant.

II.1.4. Cadre de nos travaux et hypothèses générales

Nous nous plaçons dans le cadre d'une architecture parallèle de type cluster et plus précisément de type « grappe de PCs » possédant des nœuds de calcul homogènes tant d'un point de vue matériel que d'un point de vue logiciel. Chaque nœud de calcul possède un système d'exploitation de type UNIX standard : LINUX ou FreeBSD. Les machines individuelles de la grappe de PCs peuvent être monoprocesseur ou SMP.

Nous supposons que la grappe est interconnectée par un réseau rapide fournissant une primitive d'écriture en mémoire distante (cf. II.2). La grappe est par ailleurs interconnectée par un réseau de contrôle classique de type Ethernet.

Nous faisons les hypothèses suivantes sur le réseau rapide :

- ✓ Le réseau rapide est fiable (pas de gestion des erreurs au niveau logiciel).
- ✓ Les communications sont point à point et bidirectionnelles.
- ✓ Le réseau est FIFO : les messages ne peuvent pas se doubler ; ils arrivent chez le récepteur dans l'ordre où ils ont été émis.
- ✓ La topologie du réseau est quelconque ; chaque nœud doit pouvoir communiquer avec tous les nœuds distants ; on ne s'intéresse pas aux problèmes de contention sur le réseau.
- ✓ Le contrôleur réseau accède directement à la mémoire physique du nœud hôte par accès DMA (*Direct Memory Access*).
- ✓ Le contrôleur réseau fournit une primitive d'écriture en mémoire distante sans intervention du processeur de calcul.

Enfin, nous supposons que les applications sont lancées dans un mode de type « *batch* », c'est-à-dire qu'une seule application parallèle à la fois peut s'exécuter sur la machine. Ce choix repose sur l'idée que pour donner les meilleures performances possibles aux applications, il faut éviter de partager la ressource de calcul à savoir le processeur hôte. Cela permet en particulier d'éviter des changements de contexte trop coûteux qui se produiraient si plusieurs applications étaient exécutées en même temps sur la machine parallèle.

Nous limiterons notre point de vue aux communications point à point bidirectionnelles et par passage de messages entre processus communicants. Par ailleurs, par souci de simplicité, nous nous focaliserons sur les communications entre processus distants (sur des nœuds de calcul distincts) via le réseau haut débit. Nous n'étudierons pas les communications « intra-node ».

La machine MPC présentée à la section **III.1.4** sera notre plate-forme expérimentale mais nous veillerons à ce que les techniques utilisées dans notre étude soient suffisamment générales pour être applicables sur n'importe quelle plate-forme matérielle fournissant une primitive d'écriture distante.

Il nous reste à définir le niveau applicatif sur lequel notre travail est basé. Cette question est discutée dans le paragraphe suivant.

II.1.5. Les applications visées : notre choix

Comme nous l'avons déjà remarqué, une des forces des clusters est de bénéficier de modèles de programmation construits non seulement par dessus des langages de programmation séquentiels standard (C, Fortran, etc.) mais aussi avec des bibliothèques de fonctions de communications standard présentes dans la plupart des systèmes. L'utilisation de standards permet la portabilité des applications parallèles et leur diffusion rapide. Un de nos objectifs est de viser le plus grand nombre d'applications possibles. C'est pourquoi nous souhaitons fournir un environnement de programmation parallèle standard.

Les deux principaux modèles de programmation parallèle sont les modèles à passage de messages et les modèles à mémoire partagée. Ces derniers sont moins diffusés et parfois encore à l'état de prototype dans les réseaux de stations de travail, surtout du fait que les mécanismes de communication par mémoire partagée sont difficiles à implanter efficacement sur les clusters. Nous avons donc choisi d'utiliser un modèle à passage de messages.

Il existe principalement deux bibliothèques de passages de messages qui sont très largement utilisées dans les clusters : *Parallel Virtual Machine* (PVM [PVM,1994] [PVM]) et *Message Passing Interface* (MPI [MPI]).

PVM, le plus ancien, s'est longtemps imposé comme un standard de fait. Le principal avantage de PVM est sa portabilité et la possibilité de lancer des applications sur des architectures hétérogènes. Les performances sont la principale faiblesse de PVM, surtout sur les clusters. La construction de PVM par dessus les sockets UNIX et la pile de protocoles TCP/IP implique que les performances de communication de PVM sont plus faibles que celles des sockets UNIX. L'organisation de PVM implique un surcoût non négligeable. Les auteurs de [Blum,1996] montrent que PVM représente une part minoritaire du temps de communication lorsqu'il est utilisé par dessus des sockets, un système d'exploitation et du matériel standard. En revanche, par dessus des sockets optimisées sans intervention du système et sur du matériel performant, la part de PVM dans le temps de communication peut dépasser 80%. La construction de PVM par dessus les standards de communication a favorisé sa diffusion mais pénalisé sévèrement ses performances. Une implémentation de PVM a été réalisée sur la machine MPC [Silva,1998] mais les résultats ne sont pas très probants pour les mêmes raisons.

MPI, l'autre standard de programmation parallèle à passage de messages, a pris le dessus sur PVM ces dernières années. D'une part, il offre un ensemble de primitives de communication beaucoup plus complet que celui de PVM. D'autre part, les implémentations existantes de MPI sont bien plus efficaces que PVM et permettent des portages plus ou moins faciles sur diverses architectures.

Nous avons donc choisi de réaliser une implémentation optimisée de MPI au-dessus d'un réseau rapide utilisant une primitive d'écriture en mémoire distante.

II.1.6. Objectif général

Nous disposons d'une machine parallèle de type « grappe de PCs » interconnectée par un réseau rapide utilisant une primitive d'écriture en mémoire distante telle que celle de la machine MPC. Notre objectif est de fournir l'environnement de programmation parallèle à passage de messages *standard* MPI aux applications parallèles destinées à s'exécuter sur ce type d'architecture. La problématique générale de nos travaux est de faire bénéficier ces applications des très bonnes performances du réseau matériel sous-jacent en réduisant autant que possible le chemin critique logiciel qui sépare l'appel à une fonction de communication par un processus utilisateur de la prise en compte du message par le matériel du réseau.

Les piles de protocoles classiques (TCP/IP) ne sont pas assez performantes et ne sont pas adaptées aux réseaux rapides actuels. [Barak,1999] constate que les performances de TCP/IP et UDP/IP au-dessus d'un réseau rapide tel que Myrinet sont loin des limites du matériel en terme de bande passante et de latence. Elles sont parfois remplacées par des protocoles de communications plus légers comme les messages actifs [Eicken,1992] ou les *Fast-Messages* [Pakin,1995] [Pakin,1997]. Ces protocoles limitent le nombre de copies intermédiaires des informations à communiquer, évitent les appels systèmes, etc.

Il existe différentes techniques générales d'optimisation pour réduire le temps de traversée des couches de communications. [Cappello,1999] présente certaines d'entre elles. Nous pouvons citer quelques points clés qui seront étudiés plus en détail dans les paragraphes II.3, II.4, II.5, et II.6 :

- ✓ Eviter la traversée d'un grand nombre de couches de communication et simplifier les opérations réalisées en utilisant des protocoles légers.
- ✓ Réduire le nombre de copies des données.
- ✓ Réduire le nombre d'appels systèmes en réalisant des communications en espace utilisateur et éviter l'utilisation d'interruptions pour la signalisation des communications.
- ✓ Réduire les opérations de traduction d'adresses virtuelles/physiques.

II.2. La primitive d'écriture distante

Nous présentons dans cette section la primitive d'écriture en mémoire distante sur laquelle notre travail va s'appuyer.

II.2.1. Synopsis d'un échange standard

La Figure II-2 présente les différentes phases successives qui constituent une opération d'écriture distante. Le nœud A envoie des données se trouvant en mémoire physique au nœud B.

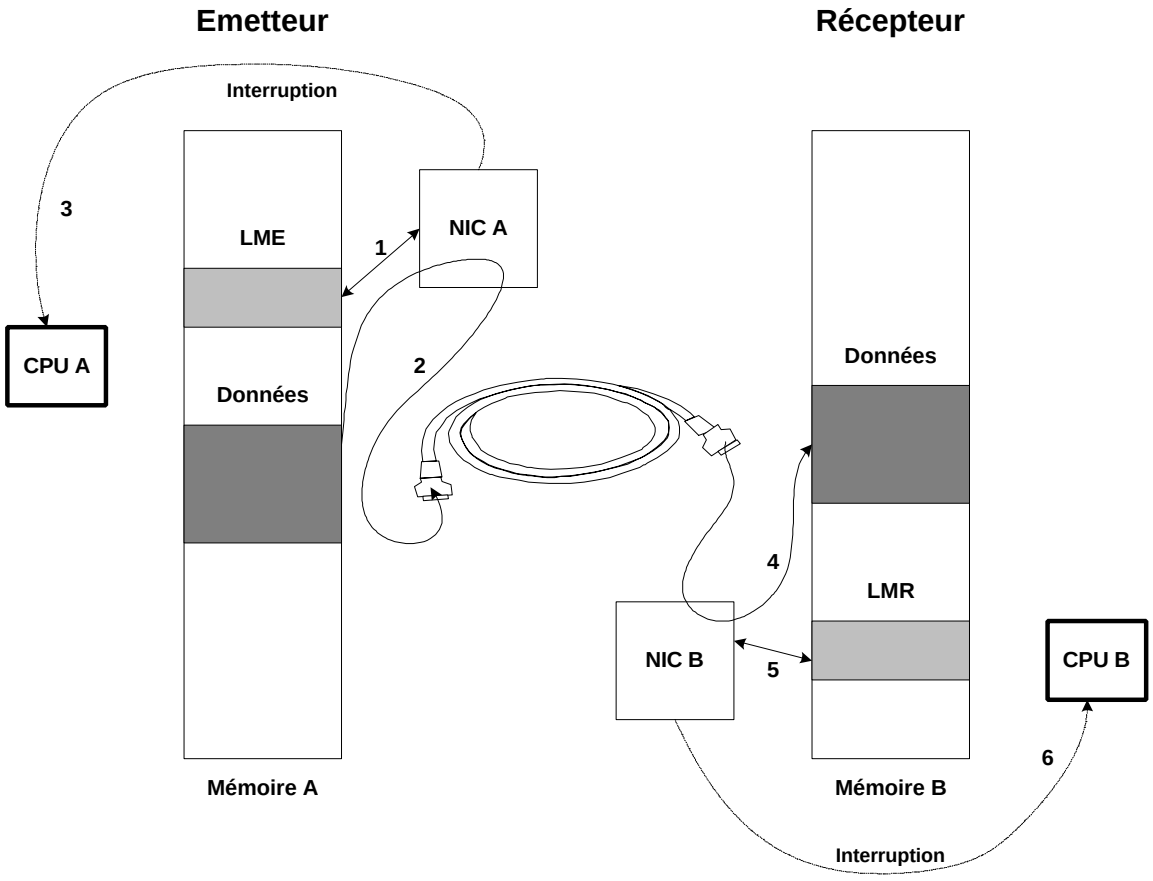


Figure II-2 : La primitive d'écriture en mémoire distante

Une entrée de la Liste des Messages à Emettre (LME) contient les champs suivants :

- ✓ un identifiant de message,
- ✓ l'adresse physique locale des données à émettre,
- ✓ la taille des données,
- ✓ l'adresse physique distante où les données doivent être écrites sur le nœud récepteur,
- ✓ un registre contenant un certain nombre d'indicateurs (pour la signalisation par exemple).

A une entrée de LME correspond un couple (tampon d'émission / tampon de réception). Ces tampons doivent être de même taille et surtout contigus en mémoire physique.

En réception, la Liste des Messages Reçus (LMR) contient les identifiants des messages qui ont été déposés.

Les différentes phases constituant l'écriture distante sont :

❶ Le système d'exploitation rajoute dans la LME les entrées correspondant au message à émettre.
❷ Le contrôleur réseau émetteur (NIC A) encapsule les données dans des paquets réseau et les transmet au contrôleur réseau récepteur (NIC B).
❸ Le contrôleur réseau émetteur signale la fin de l'émission des données au processeur hôte (CPU A) par une interruption.
❹ Le contrôleur réseau récepteur (NIC B) dépose les données en mémoire.
❺ Une fois que toutes les données sont déposées, le contrôleur réseau récepteur inscrit dans la LMR l'identifiant du message reçu.
❻ Enfin, il signale la fin de réception au processeur hôte (CPU B) par une interruption.

II.2.2. Caractérisation

Nous listons ici les différents points qui caractérisent la primitive d'écriture distante, en différenciant les caractéristiques essentielles de celles qui ne le sont pas. Les caractéristiques nécessaires sont les suivantes :

- ✓ Nous rappelons que nous supposons que le réseau est fiable. Les pertes ou corruptions de données sont détectées et signalées par le contrôleur réseau mais sont supposées rares. Elles ne sont donc pas corrigées par le logiciel et entraînent un arrêt brutal de l'application.
- ✓ Nous supposons également que le réseau est FIFO. C'est le cas de la machine MPC qui est présentée au chapitre III.
- ✓ Le contrôleur réseau utilise des accès DMA en lecture et en écriture pour accéder à la mémoire du nœud hôte.
- ✓ La LME et la LMR sont uniques sur chacun des nœuds. Le système d'exploitation doit donc gérer les accès multiples provenant de différents processus à ces structures de données partagées.
- ✓ Lors d'un transfert, le tampon d'émission et le tampon de réception associé doivent être contigus en mémoire physique.
- ✓ Les données sont déposées en mémoire sur le nœud récepteur sans aucun contrôle possible de la part du processeur local. Pour réaliser un transfert, le nœud émetteur doit savoir « à l'avance » où déposer les données sur le nœud récepteur.

- ✓ Lors de la signalisation d'une fin d'émission ou d'une fin de réception, le processeur local doit disposer de l'identifiant du message qui vient d'être déposé ou émis. La gestion des identifiants de messages est réalisée par les couches de communication utilisant la primitive d'écriture distante.

La primitive d'écriture distante de la machine MPC, constituant notre plate-forme expérimentale, satisfait les conditions énoncées ci-dessus. Elle est présentée au chapitre III et est également caractérisée par les points suivants qui sont eux non nécessaires dans notre étude mais qu'il convient de signaler :

- ✓ Lorsqu'une erreur se produit, le contrôleur réseau de la carte Fast-HSL émet une interruption matérielle pour prévenir le processeur du nœud hôte.
- ✓ Dans le cas de la machine MPC, la LME et la LMR se trouvent dans la mémoire physique du nœud hôte mais ces structures de données pourraient être sur le NIC.
- ✓ La transmission est de type *zéro-copie*. Les données ne sont pas *bufferisées* sur la carte réseau Fast-HSL.
- ✓ La taille maximale d'un message correspondant à une entrée de la LME est de 64Ko. Cette limite nous est imposée par le matériel de la machine MPC.
- ✓ Lors de la signalisation d'une fin d'émission ou d'une fin de réception, les informations dont le processeur local de la machine MPC dispose sont :
  - en réception : l'identifiant du message qui vient d'être déposé.
  - en émission : l'adresse physique locale des données émises, l'adresse physique distante où les données sont déposées, la taille des données et l'identifiant du message.
- ✓ Dans le réseau HSL, l'identifiant d'un message est codé sur 24 bits.
- ✓ La signalisation se fait par interruption matérielle sur la machine MPC mais elle pourrait tout aussi bien se faire par une scrutation de la LME et de la LMR comme nous le verrons au chapitre V.

II.3. Les empilements de couches de communication

Lorsqu'un processus utilisateur d'une application fait un appel à une fonction de communication, les traversées successives de différentes couches de communication en émission comme en réception sont pénalisantes. Elles introduisent bien souvent un *overhead* non négligeable.

Par ailleurs, ces couches de communication doivent réaliser des opérations les moins complexes possibles. Les différentes fonctionnalités assurées par les piles de protocoles classiques sont la copie depuis et vers l'espace utilisateur, la copie pour une retransmission éventuelle, le multiplexage, le formatage, la détection d'erreurs, le contrôle de flux, etc. Ces opérations sont coûteuses.

Les couches logicielles de la machine MPC sont regroupées dans un noyau de communication appelé MPC-OS qui a été développé par Alexandre Fenÿo [Fenÿo,2001]. La Figure II-3 présente de façon simplifiée les différentes couches de communication de MPC-OS.

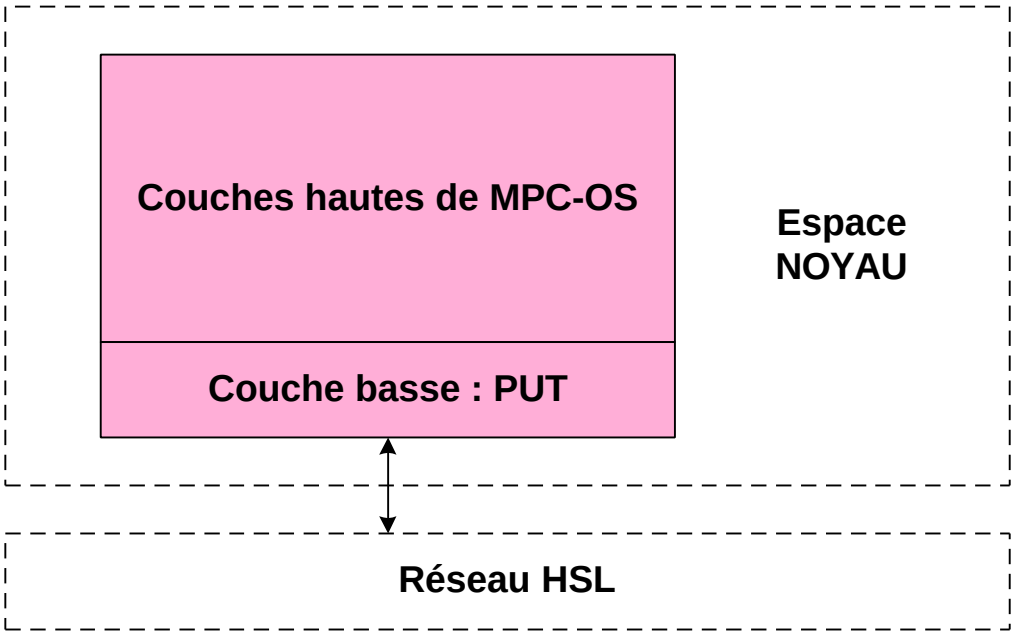


Figure II-3 : Les couches logicielles de la machine MPC

La couche **PUT**, présentée au chapitre III, est l'interface bas niveau avec le matériel du réseau HSL. Elle réalise la primitive d'écriture en mémoire distante présentée à la section II.2. Les couches hautes de MPC-OS s'appuient sur PUT pour implémenter des protocoles proposant des services à forte valeur ajoutée. La couche de plus haut niveau permet en particulier des échanges sécurisés (avec correction des erreurs) sur des canaux virtuels, avec un adressage en mémoire virtuelle, sans recopie des tampons de données. Toutes ces couches de communication sont dans l'espace noyau et font donc partie intégrante du système d'exploitation de la machine MPC.

Le portage de PVM sur la machine MPC [Silva,1998] a été réalisé au-dessus des couches hautes de communication de MPC-OS. Les résultats n'étaient pas très probants non seulement à cause des mauvaises performances intrinsèques à PVM mais aussi du fait du surcoût introduit par la traversée des couches de communication de MPC-OS. Cependant, le choix de réaliser le portage au-dessus des couches hautes de MPC-OS se justifiait par le fait qu'il aurait été beaucoup plus laborieux d'interfacer PVM directement sur la couche de bas niveau PUT.

Le délai d'invocation de la fonction d'émission au niveau des couches hautes de MPC-OS est de l'ordre de plusieurs dizaines de micro-secondes sur un processeur Intel Pentium cadencé à 200MHz [Fenÿo,2001]. Ces délais ne sont pas négligeables au vu des performances du matériel réseau de la machine MPC. La gestion des canaux virtuels et la reprise sur erreur sont des mécanismes relativement coûteux. Alexandre Fenÿo conclut ses travaux par le fait que l'on paie au prix fort les services de plus haut niveau fournis par les différentes couches de communication qui viennent s'empiler au-dessus de la couche PUT, et qu'une participation du matériel au support de ces

services à valeur ajoutée est nécessaire si on veut conserver des performances analogues à celles de PUT.

Pour notre part, nous avons choisi de supposer le réseau fiable pour les deux raisons suivantes :

- ✓ Le taux d'erreur actuel sur le lien HSL est très faible (de l'ordre de 10^{-12}).
- ✓ La plupart des projets de réseaux hautes performances prévoient d'intégrer la reprise sur erreur dans le matériel de la carte d'interface réseau pour des raisons de performances évidentes. Les équipes du projet MPC travaillent sur la mise en place au niveau matériel d'un protocole réalisant une primitive d'écriture en mémoire distante sécurisé : *Zero Copy Secured Protocol* (ZCSP).

Pour obtenir au niveau applicatif les meilleures performances possibles, l'idéal serait de baser notre travail directement sur la couche de communication bas niveau PUT. La première question qui se pose alors est la suivante : comment réaliser une implémentation efficace de MPI s'interfaçant directement au-dessus d'une couche de communication fournissant une simple primitive d'écriture en mémoire distante ?

II.4. Une stratégie « zéro copie »

Les recopies intermédiaires de données dans le chemin critique de communication prennent du temps CPU et pénalisent les processus utilisateur de l'application. Les performances se trouvent dégradées principalement en terme de débit puisque le coût d'une recopie augmente suivant la taille des données. Le problème des copies multiples a été soulevé bien avant la disponibilité de matériels de communication rapides. Dans [Clark,1989], les auteurs l'identifient comme l'un des principaux responsables de la latence importante de TCP.

La Figure II-4 présente un certain nombre de recopies pouvant avoir lieu dans les piles de protocoles classiques comme TCP/IP par exemple. Le changement d'espace utilisateur/système en émission et système/utilisateur en réception se traduit généralement par une copie des données. Celles-ci peuvent également être recopiées depuis l'espace noyau vers les tampons de la carte réseau avant d'être transmises, le même phénomène se produisant côté récepteur dans le sens inverse. Par ailleurs, d'autres copies intermédiaires peuvent avoir lieu au sein même de l'espace système et de l'espace utilisateur. Par exemple, dans l'environnement de programmation parallèle PVM, les données utilisateur sont empaquetées puis découpées en fragments avant d'être recopiées dans les tampons du système. L'empaquetage et le découpage génèrent chacun une recopie supplémentaire.

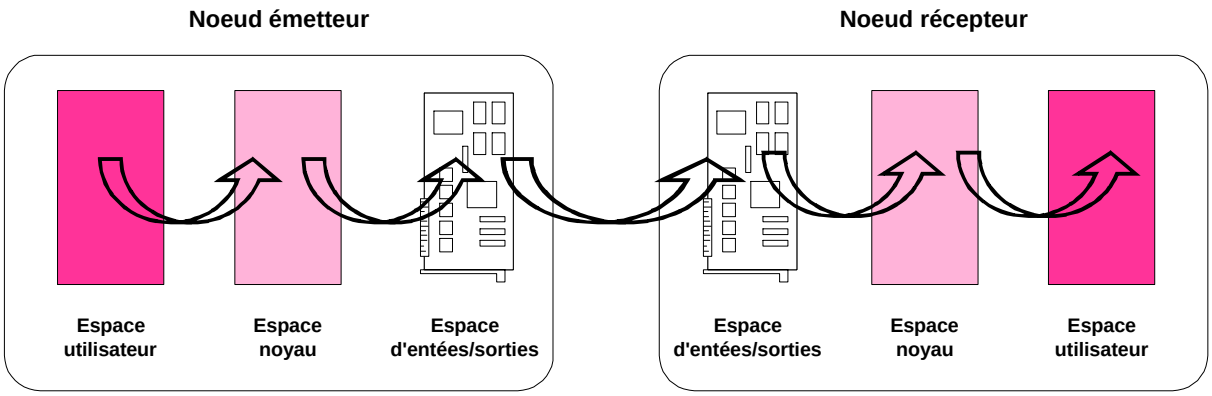


Figure II-4 : Les recopies de données dans les protocoles classiques

C'est pourquoi, des protocoles dit « zéro-copie » sont apparus ces dernières années. Ce type de protocoles rend généralement accessibles les tampons utilisateurs directement par la carte d'interface réseau par accès DMA par exemple. Cela suppose en particulier que les données aient été verrouillées en mémoire physique au préalable. La Figure II-5 représente le mode de transfert utilisé par les protocoles zéro-copie. Les données utilisateur sont transmises directement depuis un processus utilisateur local à un processus utilisateur distant sans aucune copie intermédiaire et sans *bufferisation* sur la carte d'interface réseau.

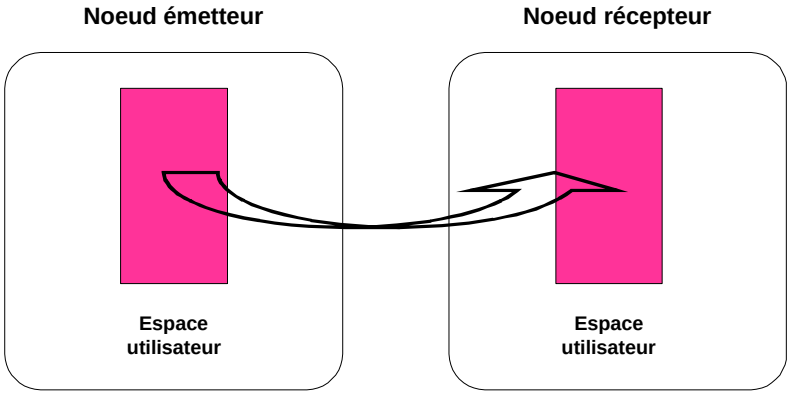


Figure II-5 : Les protocoles *zéro-copie*

Le matériel de la machine MPC et la couche de communication de plus bas niveau (PUT) permettent de réaliser des transmissions zéro-copie : les données utiles ne sont recopiées ni en émission, ni en réception, ni sur la carte réseau. Les données doivent à cet effet avoir été verrouillées en mémoire physique et une traduction d'adresse virtuelle/physique est nécessaire en émission comme en réception. La question suivante se pose alors : est-ce qu'il sera possible de conserver ce caractère zéro-copie des transmissions au niveau applicatif, c'est-à-dire lors de l'appel par un processus utilisateur d'une primitive de communication de l'environnement de programmation parallèle MPI, et à quel prix ?

II.5. Les appels système et les interruptions

L'utilisation d'appels système dans le chemin critique des communications et la façon de faire remonter la signalisation des événements réseau sont également deux points clé pour les performances des applications.

II.5.1. Les appels système

Le fait de faire un appel système lors de chaque émission et réception peut s'avérer être la cause d'une perte de performances significative pour les applications parallèles s'exécutant sur des réseaux rapides. Les appels systèmes provoquent des changements de contexte qui sont coûteux. Le problème se situe principalement au niveau de la façon d'accéder à l'interface réseau. La Figure II-6 représente les deux possibilités : soit le processus utilisateur peut accéder directement aux registres et tampons de la carte d'interface réseau (à droite), soit il est obligé de passer par un pilote d'accès au matériel se trouvant nécessairement dans le système d'exploitation (à gauche). Dans le cas d'accès direct, toutes les couches de communications se trouvent dans l'espace utilisateur ; les tampons et registres de la carte réseau sont projetés depuis l'espace noyau vers l'espace utilisateur. Ainsi, lors d'appels aux primitives d'émission ou de réception, il n'est plus nécessaire de faire un appel système ; le système n'est utilisé que dans les phases d'initialisation et de terminaison.

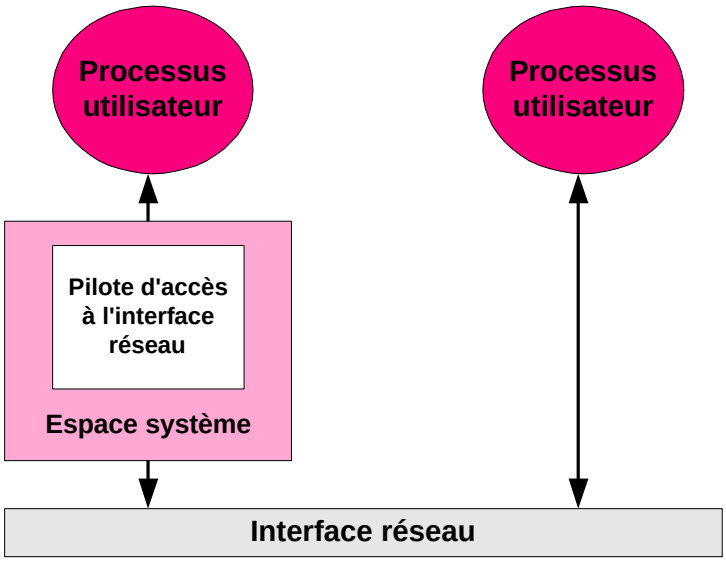


Figure II-6 : L'accès à l'interface réseau

Cependant, pour ces protocoles se trouvant entièrement dans l'espace utilisateur, la question du partage des ressources de la carte réseau entre différents processus utilisateur se pose. Ce problème peut se résoudre au niveau de l'interface réseau directement lorsque celle-ci comporte suffisamment d'intelligence. Certains de ces

protocoles sont présentés au chapitre III. Nous pouvons citer par exemple BIP (*Basic Interface for Parallelism*) [Prylli,1998] [BIP] et FM (*Fast Messages*) [Pakin,1995] [Pakin,1997] qui sont implantés au-dessus d'un réseau Myrinet [Boden,1995].

II.5.2. Interruption versus scrutation

Un autre point crucial dans les performances des couches de communication est la façon de faire remonter au niveau applicatif la signalisation des événements réseau comme la fin d'émission ou de réception d'un message. Il existe deux principaux mécanismes : la signalisation par interruption matérielle et la signalisation par scrutation (ou *polling*).

Dans le premier cas, le contrôleur réseau provoque une interruption matérielle pour signaler un événement. Un gestionnaire d'interruption se trouvant nécessairement dans le système d'exploitation est appelé. Il lui revient alors la charge de faire remonter l'information au niveau applicatif par émission d'un signal vers le processus utilisateur concerné ou bien par modification d'un drapeau se trouvant dans la mémoire virtuelle du noyau et éventuellement projeté dans l'espace utilisateur. La Figure II-7 illustre ce mécanisme. L'interface réseau provoque une interruption matérielle pour signaler la fin de réception d'un message. La fonction *Interrupt_handler()* se trouvant dans le noyau traite toutes les interruptions provenant de la carte réseau. Elle peut par exemple envoyer le signal adéquat au processus utilisateur concerné par cette réception. Celui-ci va alors pouvoir traiter le message reçu en appelant la fonction *Message_received()* se trouvant dans son espace utilisateur.

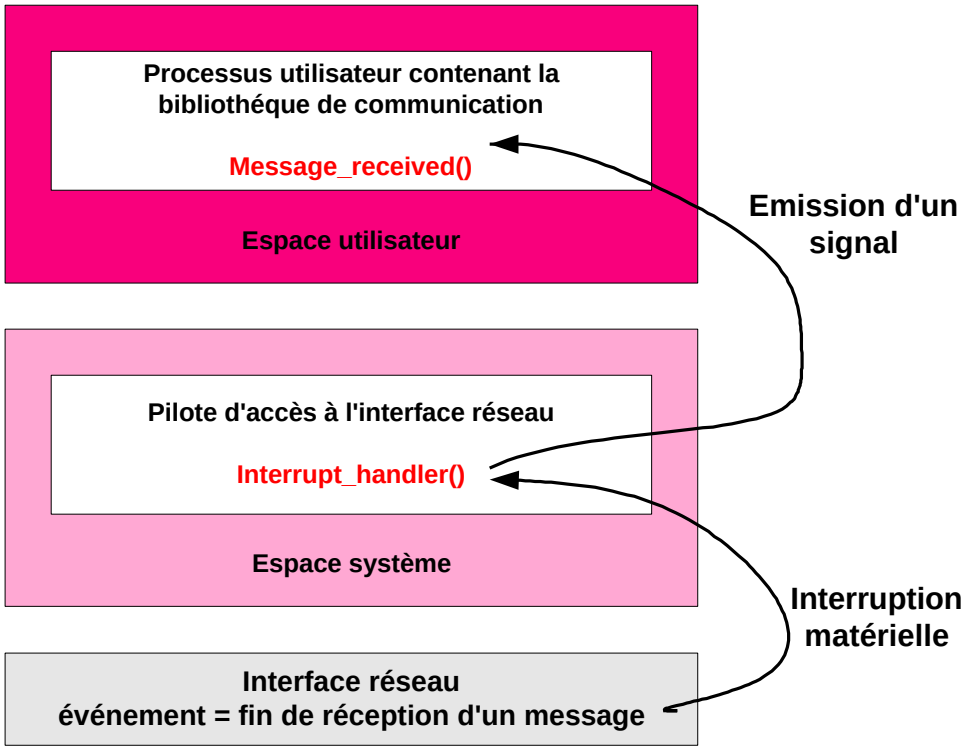


Figure II-7 : Signalisation par interruption

Dans le deuxième cas, les processus utilisateur de l'application scrutent directement les registres de statuts de l'interface réseau lorsqu'ils sont en attente d'un événement réseau. Ces registres contiennent en particulier des drapeaux indiquant qu'une émission ou une réception est terminée. Ils sont accessibles soit par accès mémoire soit par accès I/O. Les accès mémoire sont plus rapides mais pas toujours possibles, ce facteur dépendant du contrôleur réseau cible. Il est alors possible de projeter les registres de la carte d'interface réseau non seulement dans la mémoire virtuelle noyau mais aussi dans l'espace mémoire du processus.

Le principal inconvénient de la signalisation par interruption matérielle est le surcoût qu'elle engendre, surtout pour le transfert de petits messages. Une interruption déclenche un passage obligé par le système d'exploitation et provoque donc un changement de contexte systématique. Par ailleurs, les interruptions (on parle aussi d'exception matérielle) ne sont pas toujours implémentées efficacement aussi bien au niveau matériel qu'au niveau du système d'exploitation. Il est donc préférable de procéder par scrutation plutôt que par interruption lorsque cela est possible.

II.5.3. Le cas de la machine MPC

La couche de communication bas niveau PUT de la machine MPC se trouve dans le noyau du système d'exploitation (cf. Figure II-3). Ainsi, lors de chaque émission et lors de chaque réception, les couches de communication MPI devront faire au minimum un appel système. Par ailleurs, la signalisation des événements réseau est réalisée par interruption matérielle provenant du contrôleur réseau PCI-DDC. La question qui se pose alors est la suivante : est-il possible d'optimiser, d'une part les couches de communication en évitant de faire appel au système d'exploitation lors de chaque communication et, d'autre part, la signalisation d'une fin d'émission ou de réception d'un message en utilisant un mécanisme de signalisation par scrutation ?

II.6. Les opérations de traduction d'adresses

L'utilisation d'accès directs à la mémoire (DMA) par le contrôleur réseau pour transférer les données est très efficace car elle ne nécessite pas l'intervention du processeur hôte durant les transferts mais elle complique la tâche des couches de communications. Les mécanismes de DMA utilisés par des périphériques au travers du bus d'entrée/sortie ne sont pas intégrés au système de gestion de la mémoire virtuelle. En effet, les DMA se programment à l'aide des adresses physiques de la mémoire principale. Cela pose deux problèmes majeurs. D'une part, le système d'exploitation doit être au courant des zones de mémoire virtuelle potentiellement accessibles par DMA pour qu'elles soient gardées en mémoire physique et non *swappées* sur le disque. Les pages correspondantes seront alors verrouillés en mémoire physique. D'autre part, les couches de communication vont devoir effectuer une traduction d'adresse virtuelle en adresse(s) physique(s) du côté émetteur et inversement,

d'adresse(s) physique(s) en adresse virtuelle du côté récepteur. Les tampons contigus en mémoire virtuelle ne le sont pas forcément en mémoire physique.

La Figure II-8 présente un exemple dans lequel un processus émetteur (à gauche) veut transférer un tampon utilisateur vers un processus récepteur (à droite). Les tampons d'émission et de réception sont contigus en mémoire virtuelle mais ne le sont pas en mémoire physique ; ils sont chacun décomposés en deux morceaux distincts en mémoire physique. Dans le cadre de nos travaux, le contrôleur réseau ne sait transférer que des blocs contigus en mémoire physique, non seulement sur le nœud émetteur, mais aussi sur le nœud récepteur. Avant une émission, les couches de communication du processus émetteur doivent connaître la description du tampon de réception dans la mémoire physique distante et faire une association de celle-ci avec la description du tampon d'émission dans la mémoire physique locale. Sur la Figure II-8, trois appels à la primitive d'écriture distante seront nécessaires pour réaliser le transfert.

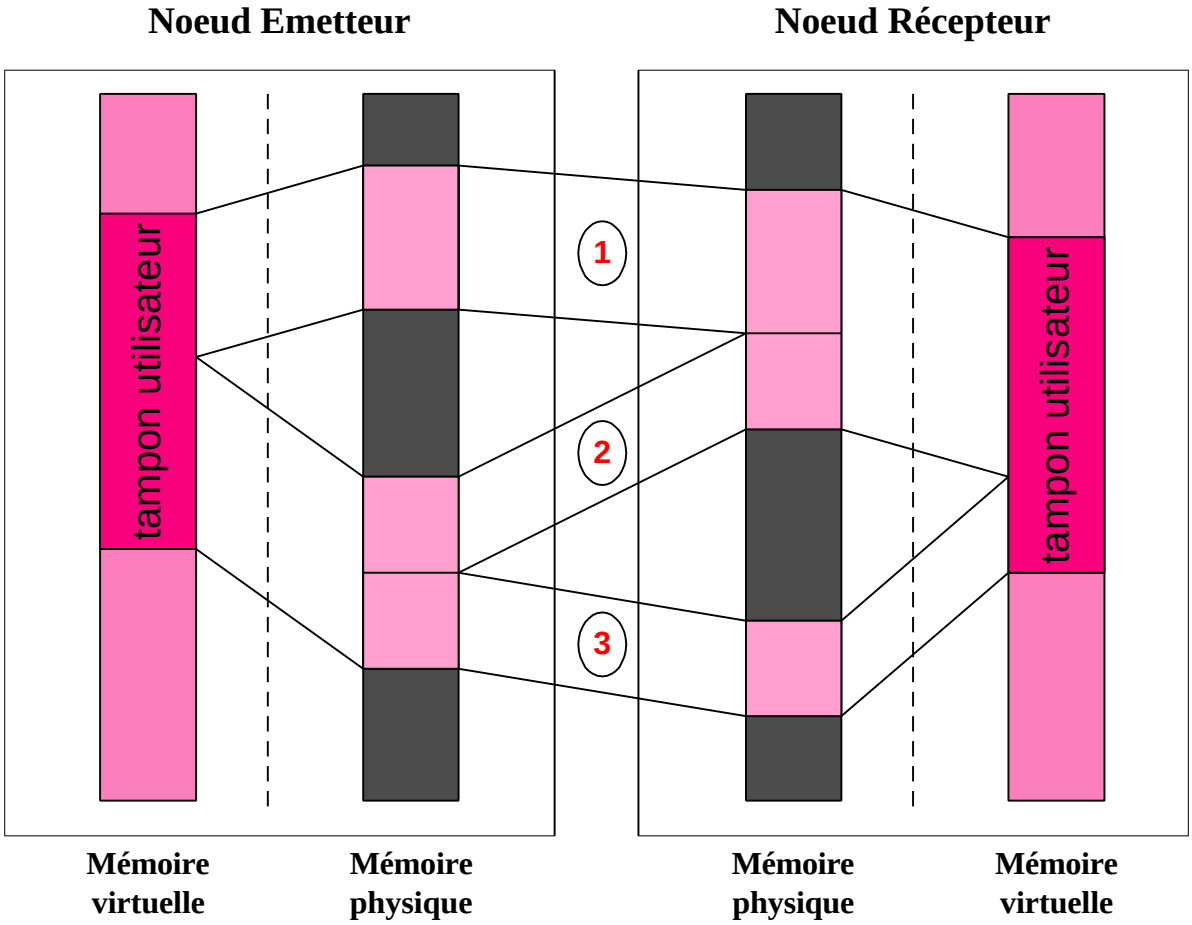


Figure II-8 : Les traductions d'adresse virtuelle/physique

Les opérations de verrouillage des zones d'émission et de réception ainsi que les opérations de traduction d'adresses virtuelles/physiques sont coûteuses car elles font nécessairement intervenir le système d'exploitation. Ces opérations pouvant avoir lieu lors de chaque communication, comment réduire ou même éliminer totalement leur impact sur les performances ?

II.7. Evaluation des performances

Nous discutons dans cette section de la façon dont nous allons évaluer les performances de nos couches de communication et des questions que nous devrons nous poser lors de ces mesures. Nous pouvons distinguer deux types de mesure des performances. Le premier consiste à analyser les performances en termes de latence et de débit. Le deuxième consiste à mesurer les performances sur des applications réelles et des *benchmarks* classiques servant à évaluer les performances des machines parallèles.

II.7.1. Evaluation en termes de latence et de débit

Les performances des communications peuvent être caractérisées par deux paramètres : **le débit** entre deux processus communicants (en octets ou bits par seconde) et **la latence**. La latence est le temps de l'acheminement d'une donnée de taille minimale entre le processus émetteur et le processus récepteur (en µs). Elle peut être décomposée en deux parties : le surcoût logiciel et la latence matérielle [Culler,1993]. Un paramètre découlant du débit et de la latence est le demi débit : c'est la taille du message en octets permettant d'atteindre la moitié du débit asymptotique.

Pour mesurer ces paramètres, nous procéderons comme suit. Nous considérons un processus émetteur et un processus récepteur se trouvant sur des nœuds de calcul distincts. Le processus émetteur envoie successivement 1000 messages de 1 octet, 1000 messages de 2 octets et ainsi de suite pour toutes les tailles de message correspondant à une puissance de 2 et ce, jusqu'à une taille maximale permettant d'atteindre la saturation du débit, c'est-à-dire le débit maximal. Le processus récepteur renvoie à l'émetteur tous les messages qu'il reçoit. Les réceptions sont bloquantes. Le pseudo-code correspondant est présenté dans le Tableau II-2. Il s'agit d'un *ping-pong* entre deux processus.

Processus émetteur	**Processus récepteur**

Pour taille de 1 à 2^n
Faire
  Pour i de 1 à 1000
  Faire
  timer_start()
  envoie(taille)
  reçoit(taille)
  timer_stop()
  Finpour i
Finpour taille
	Pour taille de 1 à 2^n
	Faire
	  Pour i de 1 à 1000
	  Faire
	
	  reçoit(taille)
	  envoie(taille)
	
	  Finpour i
	Finpour taille

Tableau II-2 : Pseudo-code d'un *ping-pong*

Le temps de chaque aller-retour est mesuré par déclenchement d'un *timer*. Nous réalisons ensuite, pour chaque taille de message, une moyenne sur les 1000 allers retours que nous divisons par 2 afin d'obtenir le temps moyen de transfert d'un message de taille 2^n octets. La latence est alors le temps moyen de transfert d'un message de 1 octet. Le débit pour une taille donnée s'obtient par la formule suivante :

$$Débit(taille) = \frac{8 * taille}{temps\_de\_transfert}$$

La taille est exprimée en octet, le temps de transfert en micro-secondes (µs) et le débit est obtenu en Mbits/s. Cela permet de tracer la courbe du débit en fonction de la taille du message transféré. Le demi débit s'obtient alors facilement par lecture sur la courbe.

Nous proposerons une première implémentation de MPI sur la primitive d'écriture en mémoire distante au chapitre IV puis deux variantes aux chapitres V et VI. Les questions qui se poseront alors seront les suivantes :

- ✓ Quelles sont les valeurs de la latence, du débit et du demi débit au niveau de deux processus communicants qui utilisent les primitives de communication classiques de MPI, à savoir `MPI_Send()` et `MPI_Recv()` ?
- ✓ Quel est le coût de traversée de nos couches de communication ?

II.7.2. Evaluation avec des applications réelles

Nous testerons et nous évaluerons au chapitre VII les performances de nos couches de communication sur des applications réelles. Nous nous poserons la question suivante :

- ✓ Qu'en est-il de notre implémentation sur des applications réelles ?

II.8. Synthèse

Nous avons présenté au début de ce chapitre l'objectif global que l'on cherche à atteindre : fournir l'environnement de programmation parallèle à passage de messages MPI au-dessus d'un matériel proposant une primitive d'écriture en mémoire distante en réduisant autant que possible le chemin critique logiciel qui sépare l'appel à une fonction de communication par un processus utilisateur de la prise en charge du message par le matériel réseau. Nous avons décrit la primitive de dépôt direct en mémoire distante ainsi que ses principales caractéristiques.

On s'est rendu compte au fur et à mesure de la réflexion des différents facteurs qui auraient une influence significative sur les performances des couches de communication :

- ✓ les empilements de couches de communication,
- ✓ le fait de conserver ou non la stratégie zéro-copie proposée par le matériel,
- ✓ le fait de faire appel ou non au système d'exploitation lors des communications,
- ✓ le mécanisme utilisé pour réaliser la signalisation des événements réseau,
- ✓ les opérations de traduction d'adresses (virtuelles/physiques).

Nous avons été amenés à nous poser, dans ce cadre, un certain nombre de questions :

- ✓ Il faut tout d'abord faire le lien entre les primitives de communications MPI et la primitive d'écriture en mémoire distante.
  - Comment réaliser une implémentation efficace de MPI s'interfaçant directement au-dessus d'une couche de communication fournissant une simple primitive d'écriture en mémoire distante ?
  - Est-ce qu'il sera possible de conserver le caractère zéro-copie des transmissions au niveau applicatif, c'est-à-dire lors de l'appel par un processus utilisateur d'une primitive de communication de l'environnement de programmation parallèle MPI ?
  - Quel sera le coût de la traversée de nos couches de communications en terme de latence ? Le débit maximal fourni par le matériel sera-t-il atteint ?

- ✓ Les questions concernant l'utilisation systématique d'appels système et la signalisation des événements réseau se posent alors :
  - Est-il possible d'optimiser les couches de communications en évitant de faire appel au système d'exploitation lors de chaque communication ?
  - Comment réaliser une signalisation performante des fins d'émission ou de réception d'un message en utilisant un mécanisme de signalisation par scrutation ?
  - Le fait d'éviter les appels systèmes et d'utiliser un mécanisme de signalisation par scrutation apporte-t-il un gain significatif en terme de performances ?

- ✓ Comment réduire ou même éliminer totalement l'impact des opérations de traduction d'adresses sur les performances des communications ? Le gain obtenu sera-t-il à la hauteur de celui espéré ?

- ✓ Ce dernier point concerne l'évaluation des performances de nos couches de communication.
  - Quelles sont les valeurs de la latence, du débit et du demi débit au niveau de deux processus communicants qui utilisent les primitives de communication classiques de MPI, à savoir `MPI_Send()` et `MPI_Recv()` ?
  - Quel est le coût de traversée de nos couches de communication ?
  - Quelle est l'influence sur les performances des différents facteurs cités précédemment ?
  - Qu'en est-il de notre implémentation sur des applications réelles ?

Chapitre III : Etat de l'art

Sommaire

Les machines parallèles de type « grappe de PCs » sont le plus souvent composées d'un réseau physique et de cartes d'interface réseau propriétaires interconnectant des nœuds de calcul standard. On trouve ensuite des couches de communication bas niveau, permettant d'accéder aux fonctionnalités de base fournies par le matériel réseau. On trouve enfin des bibliothèques de communication haut niveau fournissant un environnement de programmation parallèle aux applications. Ces bibliothèques proposent généralement de nombreuses primitives de communication point à point ou collectives.

III.1. Les machines parallèles de type « grappe de PCs »

Nous avons présenté une classification des architectures parallèles dans la section I.2. Nous décrivons dans cette section les machines parallèles de type « grappe de PCs » qui sont au centre des travaux présentés dans ce manuscrit.

III.1.1. Les *clusters* ou les réseaux de stations de travail

Un cluster est un ensemble de machines interconnectées qui travaillent ensemble et qui sont vues comme un seul système.

[Buyya,1999] [Pfister,1998] [Hwang,1998] [Spector,2000] [Sterling,1999] [Wilkinson,1999] traitent des architectures, de la programmation et des applications des *clusters*.

III.1.1.1. Pourquoi les *clusters* ?

C'est en fait dans les années 1990 que les *clusters* (ou réseaux de stations de travail) sont vraiment devenus populaires du fait de leur très faible coût comparativement aux supercalculateurs parallèles propriétaires (Cray/SGI T3E) qui restent très chers, pour une puissance de calcul quasiment équivalente. Le *benchmark* LINPACK [LINPACK] [Dongarra,1994] est très souvent utilisé pour comparer des machines parallèles entre elles. Il sert en particulier de référence pour le classement TOP500 [TOP500]. Il consiste en la résolution d'un système dense d'équations linéaires par la méthode du pivot de Gauss partiel. Cette transition a pu se faire du fait des rapides progrès accomplis dans les composants haute performance qui constituent les *clusters* : processeur, mémoire, réseaux d'interconnexion haut-débit, etc.

Un autre avantage des clusters est leur grande extensibilité. Ils sont construits à partir de PCs standard ou de stations de travail. Ainsi, à partir d'un budget donné, il est possible de construire une plate-forme qui peut convenir à une grande classe d'applications.

Un des facteurs importants qui a fait que les clusters sont devenus une alternative sérieuse aux supercalculateurs est la standardisation de nombreux outils logiciels utilisés par les applications parallèles. Nous pouvons citer l'environnement de programmation parallèle MPI (*Message Passing Interface*) [MPI] et les langages de programmation parallèle HPF [Koelbel,1994] et OpenMP [Chandra,2000].

La liste suivante résume les différentes raisons qui font que les clusters sont souvent préférés aux machines parallèles spécialisées [Anderson,1995] [Baker,1996]:

- ✓ Les stations de travail individuelles et les PCs standard sont de plus en plus puissants [Moore,1965].
- ✓ Les réseaux d'interconnexion des stations de travail (en particulier les réseaux haut débit) sont de plus en plus performants : la latence ne cesse de diminuer et le débit ne cesse d'augmenter.
- ✓ Les clusters sont plus faciles à intégrer dans les réseaux existants que les supercalculateurs.
- ✓ La standardisation des outils de développement pour les clusters est un avantage comparativement aux solutions propriétaires non standardisées.
- ✓ Les clusters, à puissance de calcul égale, sont beaucoup moins chers que les machines parallèles spécialisées.
- ✓ Un cluster peut évoluer très facilement : ajout de nœuds de calcul, changement ou addition de composants dans les nœuds de calcul (processeur, mémoire).
- ✓ Le coût de maintenance d'un cluster est très inférieur à celui d'un supercalculateur.

III.1.1.2. Qu'est-ce qu'un *cluster* ?

Nous allons dans ce paragraphe voir quelle est l'architecture d'un cluster ainsi que les composants qui le constituent.

III.1.1.2.1. Définition

Voici une définition possible d'un cluster [Buyya,1999] : un cluster est un système réalisant du calcul parallèle ou distribué, qui consiste en un ensemble de machines individuelles interconnectées entre elles, et travaillant ensemble comme une unique ressource de calcul. Un nœud de calcul (machine individuelle) peut être un système monoprocesseur ou multiprocesseur (PC, station de travail, ou machine SMP) comportant de la mémoire, des accès entrées/sorties, et un système d'exploitation. Les nœuds de calcul peuvent se trouver dans des lieux géographiques très éloignés du moment qu'ils sont reliés par un réseau LAN (*Local Area Network*). L'ensemble des nœuds de calcul doit apparaître comme un système unique du point de vue des utilisateurs et des applications.

III.1.1.2.2. Architecture et composants d'un *cluster*

Une architecture typique d'un cluster ressemble à celle de la Figure III-1 [Buyya,1999].

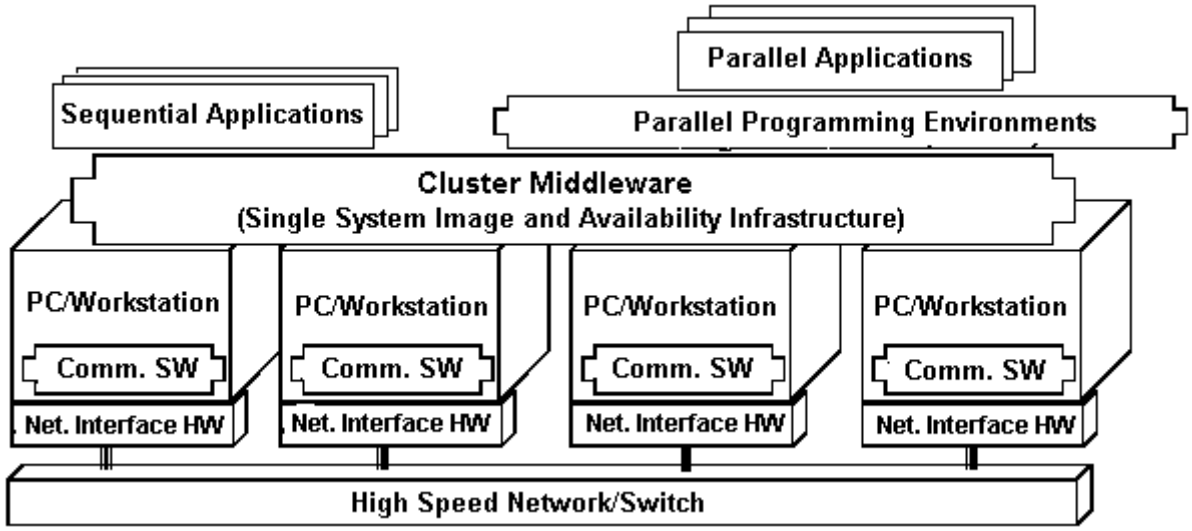


Figure III-1 : Architecture d'un cluster

Un cluster est constitué des éléments suivants :

- ✓ plusieurs machines individuelles (PC, station de travail ou machine SMP),
- ✓ un système d'exploitation sur chaque machine individuelle (le plus couramment utilisé est LINUX),
- ✓ un réseau d'interconnexion entre les différentes machines constituant le cluster (un réseau standard comme GigaEthernet ou un réseau spécialisé comme Myrinet [Boden,1995], SCI [SCI] [Gustavson,1992]),
- ✓ au moins une carte réseau par machine (NIC) sur laquelle se trouve bien souvent un processeur dédié qui s'occupe de la transmission et de la réception des données entre les différents nœuds du cluster,
- ✓ des couches de communications bas niveau (par exemple Active Messages [Eicken,1992], Fast Messages [Pakin,1995] [Pakin,1997] ou BIP [Prylli,1998] [BIP]),
- ✓ un *Middleware* : un ensemble de couches logicielles qui donnent l'illusion que le cluster est un système à image unique ; par exemple, le nombre de nœuds de calcul est transparent pour les utilisateurs et les applications,
- ✓ un environnement de programmation parallèle qui peut inclure des librairies à passage de messages (par exemple MPI [MPI] ou PVM [PVM,1994] [PVM]), des compilateurs, etc.
- ✓ des applications qui peuvent être parallèles ou séquentielles.

Certains environnements de programmation parallèle pour les clusters permettent de communiquer avec différents réseaux d'interconnexion simultanément. Cela constitue

un des objectifs du projet LANDA [Monteil,1995] (*Local Area Network for Distributed Applications*). Dans le cadre de ce projet, les auteurs de [Gauchard,2000] proposent un modèle de communication efficace entre différents réseaux, appliqué aux réseaux Myrinet et HSL. Madeleine-II [Aumage_1,2001] est une librairie de communication pour des applications *multi-threads*, capable de contrôler plusieurs protocoles de communication (BIP, SISCI, VIA) sur différents réseaux (Ethernet, Myrinet, SCI) au sein d'une même application. Elle est capable de sélectionner dynamiquement, pour un réseau donné, la meilleure méthode de transfert suivant plusieurs paramètres tels que la taille des données ou les besoins de l'utilisateur. [Petri,1999] décrit l'API HPCC (*High Performance Cluster Communication*) qui fournit au niveau applicatif un accès direct et efficace à différents réseaux haut débit. Des performances comparatives sur les réseaux SCI, Myrinet, Gigabit Ethernet et HIC sont présentées. VMI (*Virtual Machine Interface*) [Pant,2000], réalisée par le *National Center for Supercomputing Applications* (Univ. of Illinois), est une autre bibliothèque de communication haut niveau qui permet de choisir dynamiquement le meilleur réseau pour communiquer. VMI supporte actuellement VIA/Giganet, la mémoire partagée (*shmem*) et les *sockets* standard (Ethernet) sous UNIX et Windows NT. Une implémentation de MPI sur VMI a été réalisée.

III.1.2. Le projet NOW

Le projet NOW (*Networks Of Workstations*) [Anderson,1995] [NOW] de l'Université de Berkeley, consiste à utiliser un grand nombre de stations de travail individuelles (quelques centaines), à les raccorder par un réseau rapide, et à y installer une bibliothèque de communication performante, afin d'obtenir des performances analogues à celles d'une machine massivement parallèle, à bas coût.

L'expérimentation NOW à Berkeley a rassemblé 100 Ultra Sparc et 40 Sparc Stations Sun sous Solaris, 35 PC sous NT et Unix, 300 stations de travail HP, et entre 500 et 1000 disques, le tout connecté par un réseau de commutateurs Myrinet.

III.1.3. Le projet *Beowulf*

Le projet Beowulf [Sterling,1995] [Sterling,1999] [Beowulf] consiste en la réalisation d'une grappe de PCs sous LINUX interconnectés par un réseau Ethernet. Les protocoles de communication standard du monde UNIX sont utilisés. Pour accroître les performances, plusieurs interfaces réseau sont utilisées simultanément. Une technique, appelée « *channel bonding* », permettant d'exploiter plusieurs réseaux locaux en parallèle est utilisée. Ce système est commercialisé par Red Hat sous le nom « *Extreme Linux* ». Une topologie classique avec Beowulf consiste à mettre en place une grille à deux dimensions : chaque nœud est connecté à deux réseaux Ethernet.

La machine « theHIVE » (*the Highly-parallel Integrated Virtual Environment*) [theHIVE], mise en place par la NASA, est un *cluster* Beowulf équipé de 332

processeurs répartis sur quatre sous *clusters*. Les nœuds de calcul sont interconnectés par trois types de réseau : Fast-Ethernet, Gigabit-Ethernet, et Myrinet.

III.1.4. Notre plate-forme expérimentale : la machine MPC

La philosophie du projet MPC, initié en 1995 à l'Université Pierre et Marie Curie, est la même que celle de l'expérimentation NOW de Berkeley : rassembler un ensemble de stations de travail individuelles interconnectées par un réseau rapide et munies d'un ensemble de couches de communications performantes afin d'obtenir une puissance de calcul similaire à celle des supercalculateurs mais à un coût beaucoup plus faible. Sa particularité est d'utiliser un réseau d'interconnexion spécifique conçu au LIP6.

En ce sens, la machine MPC [Potter,1996] [Greiner,1998] [Zerrouki,2000] [Glück,2001] est un cluster dont les composants sont les suivants :

- ✓ Les machines individuelles constituant le cluster sont des **PCs standard**. Chaque nœud de calcul est équipé d'une carte mère standard possédant un *chipset* et un bus mémoire quelconque, d'une mémoire locale, d'un périphérique de stockage de masse local de quelques Go et d'un bus d'interface PCI 32bits/33Mhz. Un nœud de calcul peut éventuellement être SMP.
- ✓ Sur chacun des nœuds de calcul est installé un système **d'exploitation Unix standard** (FreeBSD ou LINUX).
- ✓ Un réseau d'interconnexion rapide entre les différentes machines constituant le cluster : **le réseau Gigabit HSL** (*High Speed Link*) [Reibaldi,1997] [HSL,1994].
- ✓ Un réseau de contrôle bas débit **Ethernet 100Mbits/s** raccordant l'ensemble des nœuds de calcul ainsi qu'une console.
- ✓ Une **carte réseau FastHSL** par nœud de calcul, sur laquelle se trouvent un routeur (Rcube) et un contrôleur réseau (PCI-DDC) spécifiques au réseau HSL.
- ✓ **Des couches de communications bas niveau** permettant l'accès au réseau HSL [Fenÿo,2001].

Le contrôleur réseau implémente un protocole fournissant une primitive d'écriture en mémoire distante.

La Figure III-2 présente la machine MPC du Laboratoire d'Informatique de Paris VI. Elle est constituée de 8 nœuds de calcul possédant chacun deux processeurs PIII-1GHz et 1Go de mémoire (soit 16 processeurs PIII-1GHz et 8 Go de mémoire) ainsi que d'une console servant de serveur de fichiers et permettant aux utilisateurs de lancer leurs applications. Sur la partie droite, on peut distinguer les connecteurs de la carte FastHSL ainsi que les liens HSL.

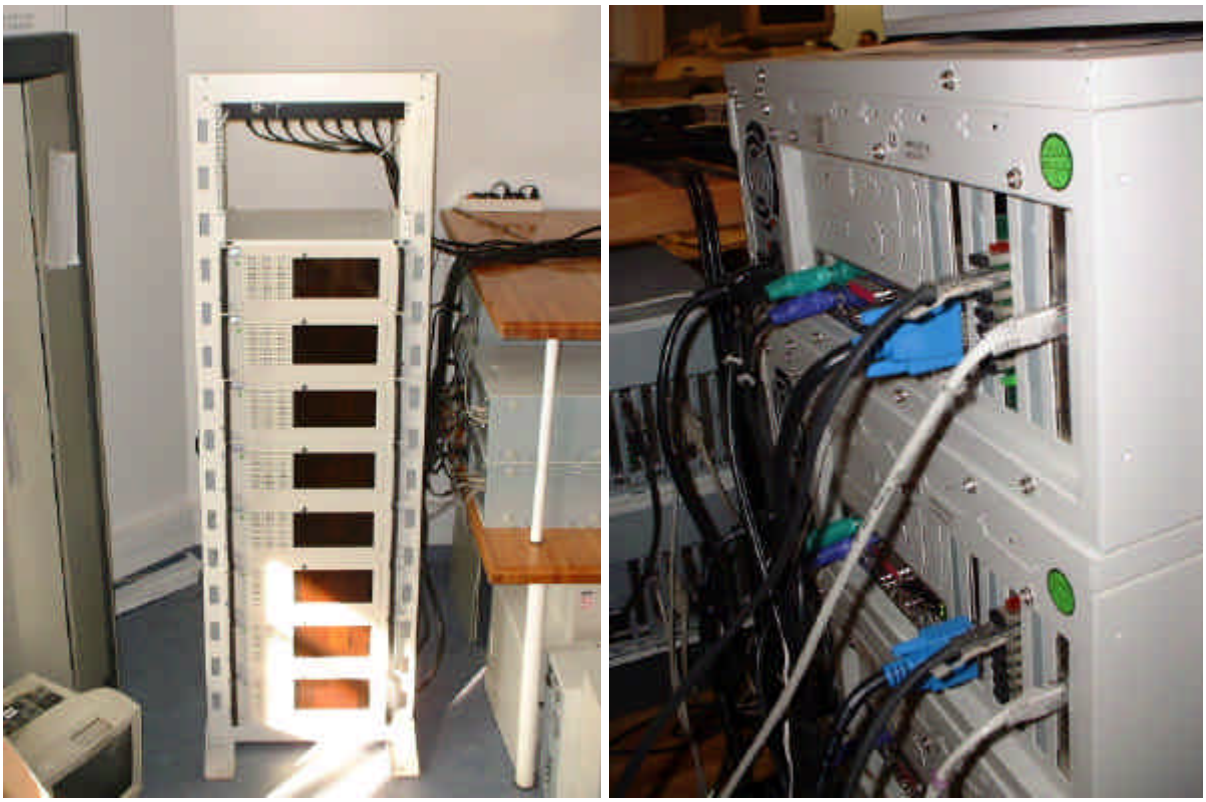


Figure III-2 : La machine MPC du LIP6

La Figure III-3 présente l’architecture d’une machine MPC possédant quatre nœuds de calcul. Les liaisons HSL entre les différentes machines individuelles ne sont pas représentées (seuls les connecteurs HSL sont indiqués sur la figure) : grâce au routeur dynamique présent sur la carte FastHSL, la fonction de routage est décentralisée et on peut construire la topologie de son choix en raccordant les connecteurs des différentes cartes FastHSL entre eux.

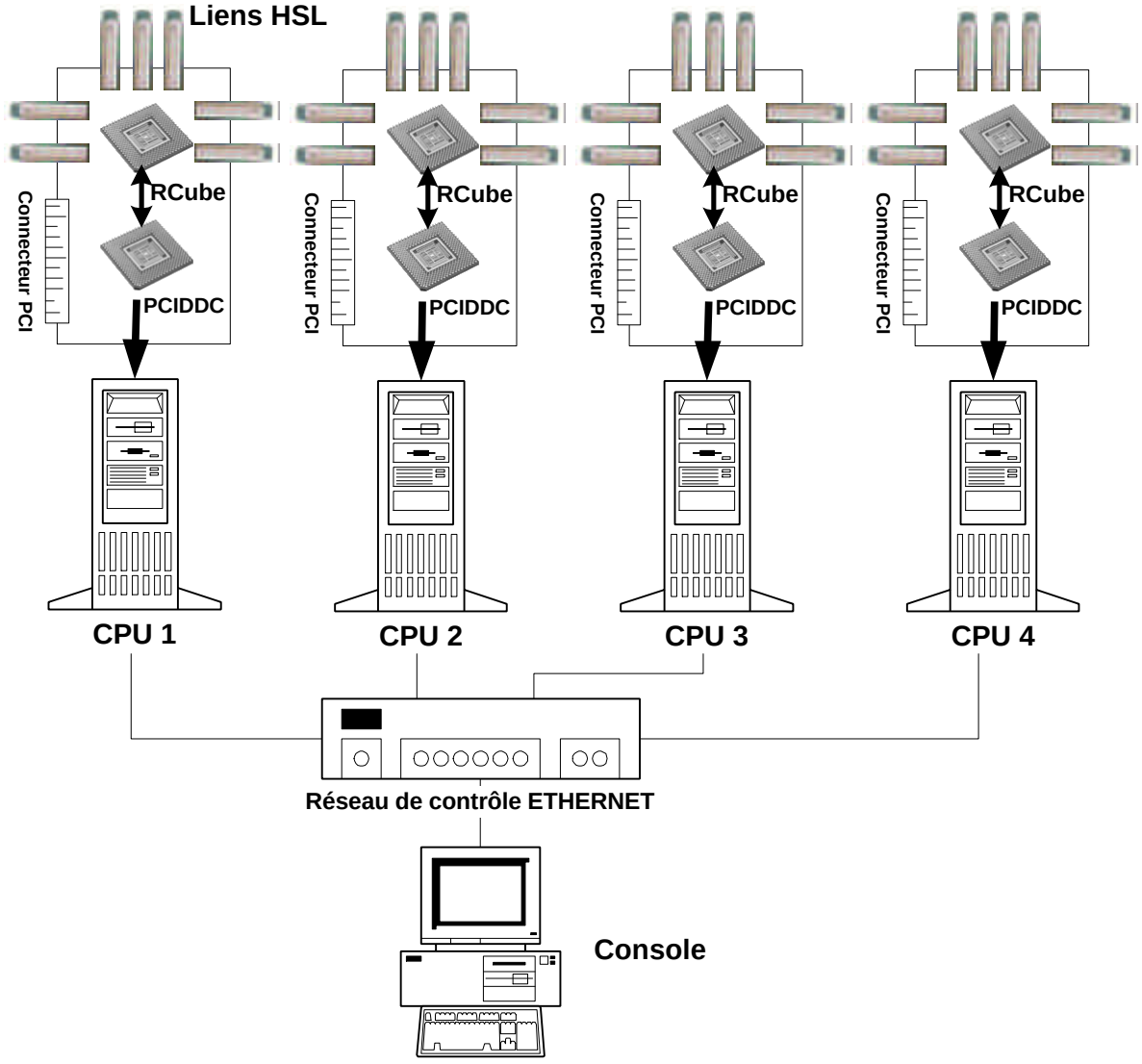


Figure III-3 : Architecture d'une machine MPC à quatre nœuds

La machine MPC est une machine parallèle de type « grappe de PCs » fournissant une primitive d'écriture en mémoire distante. Elle sera notre plate-forme expérimentale pour les travaux présentés dans ce manuscrit.

III.2. Les réseaux haut débit

Nous présentons dans cette section trois architectures matérielles : SCI, Myrinet et HSL.

III.2.1. SCI : Scalable Coherent Interface

SCI [Gustavson,1992] [SCI] correspond au standard IEEE 1596-1992. Son but est de fournir une mémoire partagée globale à faible latence aux différents processeurs constituant le *cluster*. Un système de cohérence de cache à répertoire est utilisé [Chung,2000]. La sémantique des communications repose sur l'établissement de zones de mémoire partagée.

La société Dolphin [Dolphin] est le principal fournisseur de matériel SCI. Les interfaces réseau peuvent être interconnectées en anneaux et deux anneaux peuvent être interconnectés par un commutateur. Pour les applications qui ne nécessitent pas un gros débit, les protocoles standard supportent la topologie en anneau, ce qui revient à partager la bande passante disponible sur les liens entre plusieurs nœuds mais évite le coût d'un commutateur.

SCI peut s'adapter à plusieurs types de bus : le bus PCI [Cyliax,2000], le bus SBus [Sun Microsystems] des SPARC de *Sun Microsystems*, le bus VME [VMEbus,1996], etc.

Une communication SCI est initiée par un simple accès mémoire, en lecture ou écriture, effectué par un processus, vers un espace d'adressage global unique. L'accès mémoire génère typiquement un défaut de cache, ce qui conduit le contrôleur de cache de l'interface SCI à accéder à la mémoire distante à travers le réseau pour récupérer la donnée. Un module noyau fournit à l'application la possibilité d'exporter et d'importer des régions mémoire. Une fois qu'une correspondance est effectuée entre un segment exporté par un nœud distant et importé par un nœud local, de simples écritures et lectures processeur permettent d'accéder à la mémoire distante de façon transparente. La Figure III-4 illustre ce mécanisme : le nœud j peut lire/écrire dans la région mémoire exportée par le nœud i. L'interface réseau accède à la mémoire locale par des transferts DMA. La mémoire exportée doit être verrouillée en mémoire physique. Les conversions entre les adresses virtuelles locales et les couples (nœud distant, identificateur logique du segment correspondant qui contient l'adresse physique) sont stockées dans une table de l'interface réseau. Il est possible de déclencher une interruption sur un nœud distant.

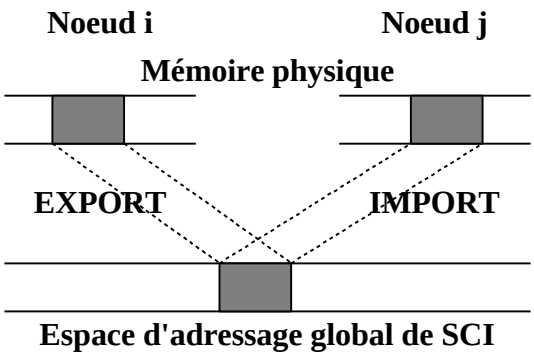


Figure III-4 : Le mécanisme de mémoire partagée de SCI

Plusieurs limitations ont ralenti le développement de SCI :

- ✓ le coût important du matériel,
- ✓ les difficultés pour étendre le réseau SCI à un grand nombre de nœuds,
- ✓ les performances sont relativement inégales : les écritures dans la mémoire distante sont très efficaces mais le coût des lectures sur un nœud distant est tellement élevé que cette fonctionnalité est peu utilisée,
- ✓ la quantité de mémoire qu'il est possible d'importer ou d'exporter est limitée : une application qui manipule un large espace mémoire ou qui souhaite utiliser une partie quelconque de son espace d'adressage pour communiquer devra, soit passer par des phases de négociation fréquentes et très coûteuses pour importer/exporter de nouvelles zones, soit utiliser des copies mémoire intermédiaires et systématiques des données depuis le tampon de l'application vers une zone importée/exportée,
- ✓ implanter efficacement des sémantiques de communication variées en se reposant sur des transferts de données par mémoire partagée n'est pas simple.

Autant SCI est très attrayant pour des applications qui peuvent exploiter directement le paradigme de mémoire partagée, autant la pertinence de son utilisation pour la réalisation d'un système à passage de messages reste à montrer.

III.2.2. Le réseau Myrinet

La technologie Myrinet [Boden,1995] est issue de deux projets de recherche plus anciens : MOSAIC [Seitz,1993] et ATOMIC [Felderman,1994]. Actuellement, la technologie Myrinet est commercialisée par la société Myricom [Myricom] qui est l'aboutissement de ces deux projets.

Contrairement à SCI, Myrinet est un réseau à passage de messages. La technologie Myrinet est très répandue dans le monde entier comme réseau d'interconnexion pour les clusters. Elle se compose d'interfaces réseau connectées sur le bus d'entrée/sortie de la machine hôte et de commutateurs. Différents liens physiques sont disponibles : SAN (nappe, jusqu'à 3 mètres), série (connecteur standard HSSDC, jusqu'à 10 mètres), et optique (fibre multimode, jusqu'à 200 mètres). Un contrôle de flux et un mécanisme de détection d'erreur sont assurés sur le lien physique. Les commutateurs sont des *crossbars* et assurent une commutation de type *wormhole*. Ils disposent chacun de 16 ports *full duplex*.

Il est possible de réaliser un réseau de topologie quelconque en reliant plusieurs commutateurs (cf. Figure III-5).

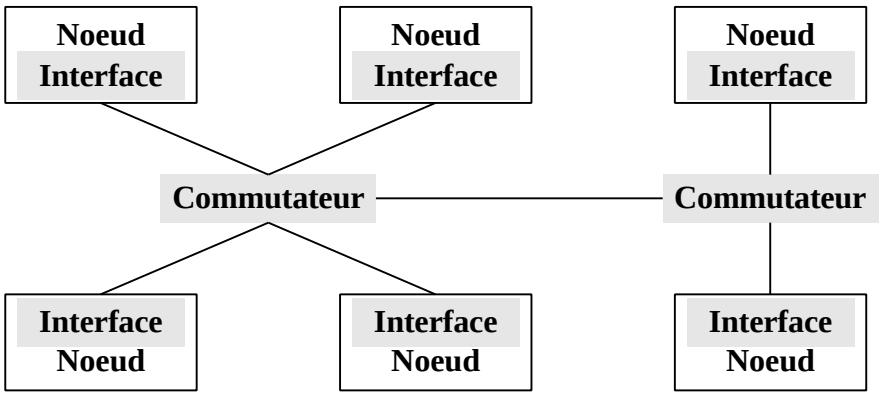


Figure III-5 : Topologie du réseau Myrinet

L'interface réseau inclut un processeur RISC (le LANai) et de la mémoire SRAM. Le LANai réalise le transfert des données entre le nœud hôte et le réseau, par des opérateurs de DMA. Il contient un micro-contrôleur programmable qui exécute le MCP (*Myrinet Control Program*). Les données transférées transitent obligatoirement par la mémoire SRAM embarquée. Cette mémoire stocke, outre les données, le MCP et les différentes structures de données qui permettent aux bibliothèques de communication résidant sur le nœud hôte de poster les ordres d'émission. La mémoire de la carte est accessible par l'hôte à travers le bus d'entrée/sortie. Les communications *full-duplex* sont supportées par le lien physique : un DMA dans chaque sens est possible simultanément. Les cartes réseau actuelles supportent le bus PCI 64bits/66MHz et contiennent de 2 à 8 Mo de mémoire. Le processeur est cadencé à 133 ou 200Mhz.

Les spécifications matérielles et logicielles de Myrinet, des documentations, et un environnement de développement pour le processeur embarqué sont disponibles sur le site web de Myricom. Cette ouverture, rare pour des produits industriels, a contribué au succès de cette technologie et a permis beaucoup d'expérimentations et d'innovations autour des systèmes de communication et des protocoles réseaux. Par ailleurs, le peu de contraintes imposées par le matériel (le format des paquets n'est pas rigide) et le caractère programmable du LANai ont contribué à la popularité de Myrinet dans les laboratoires de recherche. En revanche, les commutateurs restent encore assez chers et le passage intermédiaire des données par la mémoire embarquée empêche la construction de protocoles de type zéro-copie.

III.2.3. Le réseau HSL de la machine MPC

Le réseau HSL [HSL,1994] et les couches de communications bas niveau ont été développées par le groupe de recherche MPC [MPC].

Le réseau Gigabit de la machine MPC est composé de cartes à interface PCI présentes dans chaque nœud. La carte réseau FastHSL réalise deux fonctions principales : elle contient un processeur câblé qui exécute le protocole de communication, ainsi qu'un

routeur intégré qui permet de construire des réseaux de taille et de topologie quelconques. Elle est donc équipée de deux composants VLSI développés au Laboratoire d'Informatique de Paris VI :

- ✓ Le routeur rapide **Rcube** [Reibaldi,1997] : il s'agit d'un circuit CMOS d'environ 380000 transistors, qui offre une communication de paquets de type *wormhole* entre huit ports Gigabit à la norme HSL IEEE-1355. Les paquets du réseau HSL sont constitués d'un en-tête de deux octets qui spécifie entre autres la destination des paquets. Un paquet peut avoir une taille quelconque.

- ✓ Le contrôleur réseau **PCI-DDC** [Wajsbürt,1997] : un circuit CMOS d'environ 200000 transistors qui implémente une primitive d'écriture distante en mémoire à travers le réseau HSL. Il assure l'interface entre le bus PCI et le routeur Rcube. Il accède directement à la mémoire du nœud local par accès DMA (*Direct Memory Access*) en lecture comme en écriture, ce qui décharge le processeur local pendant les transferts de données.

Le lien HSL est un lien série, haut débit, point à point et bidirectionnel qui permet d'avoir un débit matériel de 1 Gbit/s dans chacun des deux sens de communication simultanément. Il est constitué de deux gaines coaxiales.

La Figure III-6 représente la carte FastHSL. On peut distinguer sur la photographie les deux composants VLSI développés par le groupe de recherche MPC. A droite se trouve le composant PCIDDC et à gauche Rcube.

La technologie HSL a des performances matérielles comparables aux réseaux SCI et Myrinet. Son principal avantage est de disposer d'un routeur rapide sur chaque carte réseau, permettant d'interconnecter plusieurs centaines de nœuds sans pénaliser les performances. En ce qui concerne le débit matériel du réseau HSL, le lien entre le contrôleur réseau (PCI-DDC) et le routeur (Rcube) constitue le goulot d'étranglement. Il s'agit d'un lien parallèle 8 bits. Comme la fréquence des composants matériels sur la carte FastHSL actuelle est de 66MHz, le débit matériel maximum théorique est de 66Mo/s, soit 528Mbits/s dans chaque sens de communication, ce qui est largement inférieur au débit du lien série. Par ailleurs, les caractères de contrôle et l'en-tête des paquets sur le réseau HSL consomment au minimum 3% de la bande passante. Ainsi, le débit utile matériel théorique maximum est de 512Mbits/s.

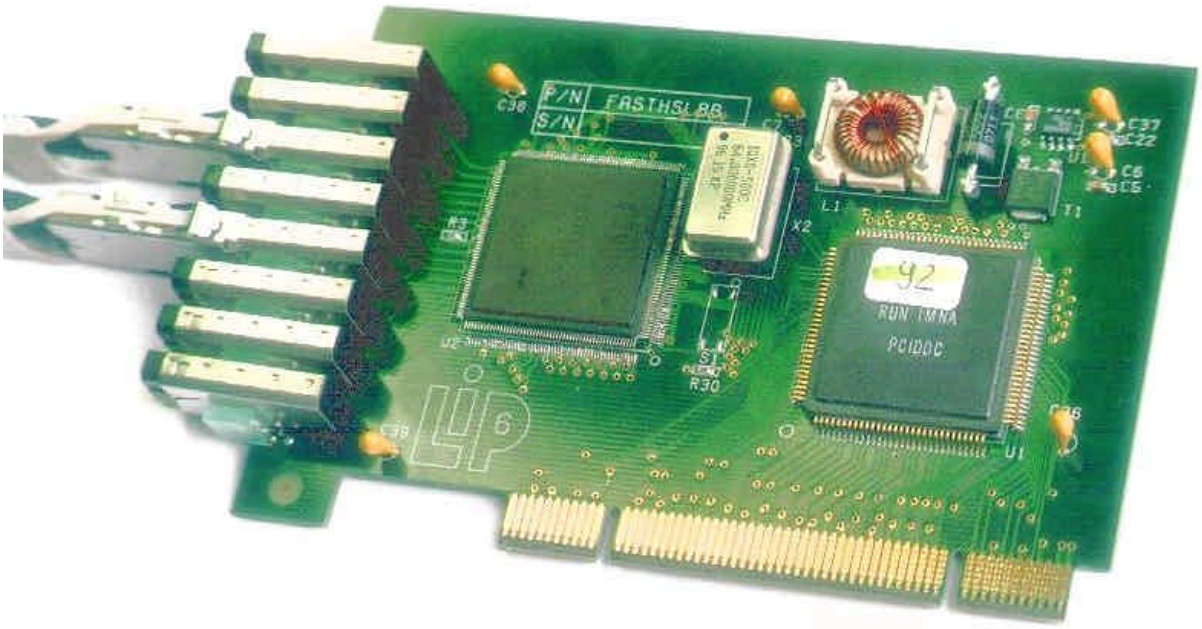


Figure III-6 : La carte FastHSL

III.3. Les bibliothèques de communication bas niveau

III.3.1. Les techniques d'optimisation

On trouve de très nombreuses bibliothèques de communication bas niveau. On peut les classer en deux catégories : les bibliothèques en mode utilisateur qui ne font pas appel au système d'exploitation pour communiquer avec le matériel réseau, et les bibliothèques qui utilisent des services noyau.

Les bibliothèques en mode utilisateur bénéficient d'une très faible latence par l'absence d'appel système mais elles sont confrontées au problème de la sécurité/intégrité du système et du partage des ressources réseau entre les différents processus utilisateur.

Différentes techniques sont mises en œuvre dans ces bibliothèques bas niveau pour optimiser les communications :

- ✓ Utiliser plusieurs réseaux simultanément : cette solution est au cœur du projet Beowulf ; elle permet d'augmenter la bande passante globale de la grappe mais ne permet pas de réduire la latence d'un transfert.
- ✓ Simplifier le protocole de communication : les protocoles standard comme TCP/IP intègrent de nombreuses fonctionnalités comme le contrôle de flux, la détection d'erreur ou même la retransmission de paquets erronés. Les protocoles de

communications « légers » laissent au matériel le soin de s'en charger ou s'appuient sur la grande fiabilité de certains réseaux rapides.

- ✓ Eviter les recopies dans des tampons intermédiaires : les copies sont pénalisantes pour les performances sauf pour les paquets de petites tailles. Les protocoles de type « zéro-copie » utilisent les fonctionnalités particulières du matériel sous-jacent (par exemple les transferts DMA).
- ✓ « Pipeliner » les phases de communication : le contrôleur réseau peut être programmé pour transmettre des données sur le lien physique tout en réalisant des transferts DMA simultanément.

III.3.2. AM : Active Messages

Les Messages Actifs (AM) [Eicken,1992] se différencient du modèle traditionnel de communication *send*/*receive*. Il s'agit d'un mode de communication unilatéral : quand un émetteur initie un échange, celui-ci est effectué indépendamment de l'activité courante du processus récepteur. Il n'y a donc pas d'opération de type *receive*.

Cette sémantique permet d'éviter des recopies intermédiaires dans des tampons temporaires. En effet, dans les systèmes de communication standard par passage de messages, on utilise généralement un tampon de stockage temporaire chez le destinataire en attendant que celui consomme les données. Avec les Messages Actifs, l'arrivée d'un message chez le récepteur provoque l'invocation d'une fonction dans le processus récepteur. Cette fonction, nommée *receiver handler*, est exécutée dans le contexte d'un processus léger (*thread*) dont le rôle est de consommer les données. La gestion de la réception est découplée du traitement effectué par le processus récepteur. Le *receive handler* se charge de positionner des structures de données pour la gestion du message suivant ou pour donner une information à l'activité principale.

Un message est constitué de deux parties : le corps du message et un pointeur explicite vers le *receive handler* destinataire chargé de consommer les données.

Les Messages Actifs ne prévoient pas de retransmission des données en cas d'erreur. Un contrôle de flux est utilisé pour éviter des débordements : chaque émetteur possède un certain nombre de jetons pour faire des émissions vers un récepteur particulier. Les messages sont délivrés sur le récepteur dans l'ordre d'émission. La signalisation est réalisée sur le récepteur par interruption ou par scrutation. Il ne peut y avoir qu'un seul processus communicant par nœud de calcul. L'émetteur peut déclencher l'exécution de n'importe quelle fonction sur le nœud récepteur, ce qui peut poser un problème de sécurité.

Une nouvelle version des Messages Actifs (AM-II) est présentée dans [Mainwaring,1995]. Elle permet de s'affranchir de certaines limitations de la première implémentation. Elle offre en particulier la possibilité d'avoir plusieurs processus communicants par nœud et prévoit une retransmission des données erronées.

[Ibel,1997] propose une implémentation des Messages Actifs sur une plate-forme SCI. [Chun,1998] implémente AM-II au-dessus du réseau Myrinet.

III.3.3. FM : Fast Messages

Les Fast Messages [Pakin,1997] de l'Université d'Illinois constituent une bibliothèque de communication de type Messages Actifs implémentée sur une plate-forme Myrinet. L'idée est de fournir un ensemble restreint de primitives de communication et d'obtenir un recouvrement calcul/communication optimal : comme pour les Messages Actifs, chaque message contient dans son en-tête l'adresse d'une fonction (*handler*) qui doit être exécutée sur le récepteur pour consommer les données. Il s'agit d'une extension des RPC (*Remote Procedure Call*).

FM fournit une garantie d'acheminement ordonné des données, un système de contrôle de flux, et suppose le réseau fiable.

Il existe deux générations de Fast Messages. La première (FM 1.x) était destinée uniquement aux grappes de stations SPARC. Elle proposait uniquement trois primitives : une fonction pour l'émission des messages longs, une pour celle des messages courts (quatre mots), et une pour vider les files de réception en appelant le *handler* associé. Comme pour les Messages Actifs, il ne peut y avoir qu'un seul processus communicant par nœud. Un portage de MPI sur FM est présenté dans [Lauria,1997] mais les performances ne sont pas très satisfaisantes à cause des recopies intermédiaires des données liées à l'interface de FM 1.x .

La deuxième génération (FM 2.x) [Lauria,1998] permet d'utiliser le bus PCI et des nœuds de calcul de type *Pentium*. L'API a été modifiée pour fournir de meilleures performances aux bibliothèques haut niveau utilisant FM. Un message est un ensemble de « *streams* » de taille quelconque. L'émetteur initie une communication en précisant le destinataire, la taille du message et la fonction qui doit être exécutée en réception. Ensuite, il envoie le message en plusieurs fragments de taille quelconque. Plusieurs *handlers* peuvent s'exécuter simultanément sur le récepteur, ce qui permet d'améliorer les performances de la première version des FM. Il est possible de *pipeliner* les communications au niveau utilisateur puisque l'émetteur peut envoyer des fragments d'un message pendant que le récepteur traite la réception d'autres fragments. Les opérations collectives de type *scatter*/*gather* sont supportées efficacement, ce qui est utile pour certaines bibliothèques haut niveau comme MPI. Un deuxième portage de MPI sur FM 2.x est présenté dans [Lauria,1999]. [Giannini,1998] présente une implémentation de primitives de type Put/Get sur FM, ce qui est intéressant dans la perspective du standard MPI-2 qui prévoit des communications de ce type (« *one-sided communications* »).

III.3.4. GM

GM [GM,2000] est la couche de communication bas niveau fournie par la société Myricom pour accéder au réseau Myrinet. Il s'agit d'une bibliothèque en mode utilisateur : les applications peuvent poster des ordres d'émission/réception sans faire appel au système d'exploitation. Un module noyau permet le chargement du MCP (*Myrinet Control Program*) dans le contrôleur réseau avant le démarrage de l'application. Il fournit des « ports » aux applications pour leurs communications et permet de verrouiller les tampons de communication en mémoire physique. Un « port » est une structure de données à travers laquelle l'application communique avec l'interface réseau. Une fois qu'un port est ouvert, les communications se font en mode utilisateur. La Figure III-7 montre l'architecture de GM. Dans cet exemple, le processus a deux ports ouverts qui lui permettent de poster des émissions/réceptions dans le MCP, à travers la bibliothèque GM, sans passer par le système d'exploitation. La figure représente les opérations de DMA qui transfèrent les données directement entre les tampons du processus et le réseau.

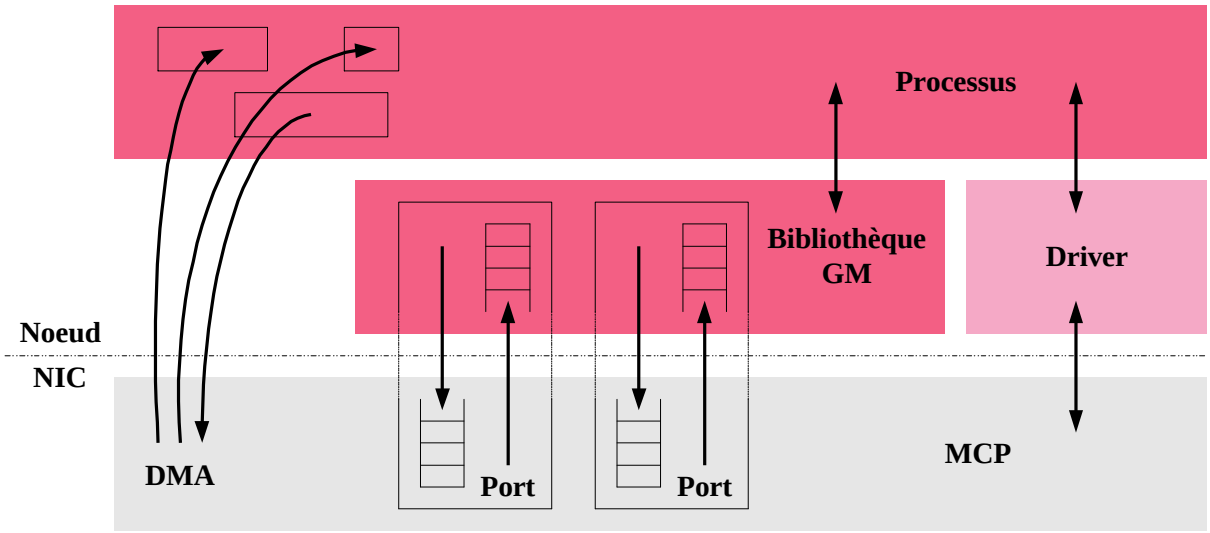


Figure III-7 : Architecture de GM

GM numérote les paquets circulant sur le réseau pour assurer l'ordre d'arrivée des messages sur un port donné. Un mécanisme d'acquittement, implanté au niveau matériel, permet à GM de retransmettre un paquet si une erreur survient sur le réseau. Un contrôle de flux permet d'éviter les débordements dans une file associée à un port : à chaque port est associé un nombre limité de jetons (*send/receive tokens*) pour les émissions et les réceptions. Un jeton d'émission est utilisé pour l'envoi d'un message. Lorsque celui-ci est terminé, le contrôleur réseau appelle une fonction de rappel du processus en lui fournissant le jeton associé. Un mécanisme similaire est utilisé en réception. GM fournit un ensemble de primitives pour verrouiller en mémoire physique les tampons d'émission/réception et informer le contrôleur réseau de la correspondance entre une adresse virtuelle et les adresses physiques associées. Les zones d'émission/réception doivent donc être enregistrées auprès du NIC avant

d'initier une communication. Ce mécanisme est appelé « *registering memory* ». GM fournit une primitive d'écriture en mémoire distante (`gm_directed_send`) depuis une zone enregistrée au préalable sur le nœud émetteur vers une zone enregistrée sur le nœud récepteur mais, par défaut, elle n'est pas autorisée pour des raisons de sécurité.

III.3.5. BIP : Basic Interface for Parallelism

BIP [Prylli,1998] [BIP] est, comme GM, une bibliothèque de communication en mode utilisateur pour le réseau Myrinet. Elle a été réalisée par une équipe mixte de l'Université de Lyon 1 et de l'Ecole Normale Supérieure de Lyon.

BIP fournit un ensemble de primitives de communication de type *send/receive* bloquantes ou non bloquantes. Les communications utilisent des « *tags* » : à chaque *tag* est associée une file de réception qui contient des tampons intermédiaires alloués lors du démarrage de l'application.

Il existe deux types de messages dans BIP : les messages courts et les messages longs. Les messages courts transitent via les tampons intermédiaires associés au *tag* utilisé, ce qui permet de faire une émission même si le récepteur n'est pas prêt à recevoir le message. Les messages longs sont transmis suivant un protocole de rendez-vous qui suppose que la réception a été postée avant l'émission. Ils sont découpés en fragments de taille fixe. Les transmissions des fragments d'un même message sont *pipelinées* de la façon suivante : (1) un transfert DMA recopie un fragment depuis la mémoire centrale vers la mémoire SRAM de la carte réseau ; (2) le fragment précédent est transmis depuis la mémoire SRAM vers le réseau. Les étapes (1) et (2) sont réalisées en parallèle. Sur le nœud récepteur, un procédé identique est utilisé. Si le message est décomposé en plus de quatre fragments, cette parallélisation des transmissions permet d'améliorer considérablement les performances. Lors du transfert d'un message long, BIP fait appel au système d'exploitation pour verrouiller les tampons d'émission/réception en mémoire physique et obtenir les conversions en adresses physiques. Le protocole de rendez-vous permet au récepteur d'envoyer dans l'acquittement le *tag* associé au tampon de réception. Quand le contrôleur réseau reçoit les données, le *tag* lui permet de retrouver les adresses physiques du tampon de réception. Ainsi, BIP peut faire des transmissions de type zéro-copie.

La transmission d'un message court ne nécessite pas la mise en place d'un DMA : les données sont recopiées dans un tampon pré-alloué, enregistré auprès du contrôleur réseau lors de l'initialisation de l'application.

Les paramètres d'une émission sont le numéro du nœud destinataire, un *tag*, et l'adresse du tampon d'émission et sa taille. Les paramètres d'une réception sont un *tag*, l'adresse du tampon de réception, et sa taille. Les primitives de réception peuvent retrouver le numéro du nœud émetteur et le type du message (court ou long).

BIP garantit l'arrivée ordonnée des messages sur le récepteur. Contrairement à GM ou FM, les utilisateurs de BIP doivent fournir un mécanisme de contrôle de flux pour éviter un débordement dans les files de réception. BIP détecte les erreurs réseau mais ne permet pas de retransmettre des paquets erronés alors que GM fournit cette fonctionnalité. Une autre contrainte imposée par BIP est qu'il ne peut pas y avoir, pour un *tag* donné, plusieurs émissions/réceptions simultanées. Enfin, BIP monopolise les ressources réseau et ne permet pas leur partage entre différentes applications.

Du fait de ces caractéristiques, les performances de BIP sont meilleures que celles de GM ou FM : il permet d'obtenir 96% du débit maximum du réseau et une latence très faible (4,3µs).

III.3.6. VMMC : Virtual Memory-Mapped Communication

VMMC [Dubnicki,1997] est une bibliothèque de communication issue du projet SHRIMP (*Scalable High-performance Really Inexpensive Multi-Processor*) de l'Université de Princeton. L'idée de VMMC est de faire réaliser par le matériel réseau des fonctionnalités traditionnellement fournies par le système d'exploitation. Les applications accèdent directement à la carte réseau pour communiquer en mode utilisateur. Cette bibliothèque a été conçue dans un premier temps pour la machine parallèle SHRIMP [Blumrich,1994] mais elle a été adaptée au réseau Myrinet. Une implémentation sous Linux et une sous Windows NT ont été réalisées.

Le modèle de communication dans VMMC impose d'importer/exporter le tampon d'émission/réception avant de commencer le transfert : le processus récepteur exporte le tampon de réception des données se trouvant dans son espace d'adressage. L'émetteur doit importer le tampon destinataire avant de commencer la transmission des données. L'importation/exportation de régions mémoire se fait à travers le système d'exploitation. Une fois cette phase initiale terminée, les communications se font directement depuis l'espace d'adressage du processus émetteur vers celui du processus récepteur, sans faire appel au système d'exploitation.

VMMC propose deux modes de transfert : « *deliberate update* » et « *automatic update* ». Le premier correspond à une écriture en mémoire distante (*Remote DMA*) : l'émetteur écrit directement dans la mémoire du processus récepteur. Le deuxième repose sur l'établissement d'une liaison (*binding*) entre le tampon d'émission et le tampon de réception. Une fois ce lien effectué, les écritures du processus émetteur dans le tampon d'émission se traduisent automatiquement par une écriture distante dans le tampon de réception associé, sans faire un appel explicite à une primitive d'émission. La signalisation repose sur des envois de signaux aux processus, ce qui n'est pas très performant, mais les auteurs prévoient d'implanter un mécanisme analogue aux Messages Actifs reposant sur l'exécution d'une fonction du processus récepteur lors d'une fin de réception.

Contrairement à BIP, VMMC fournit un mécanisme de protection des communications en mode utilisateur en vérifiant les adresses des tampons d'émission/réception et, supporte l'exécution de plusieurs applications simultanément. VMMC fournit également un mécanisme de reprise sur erreur en cas de transmission erronée et garantit l'ordre d'arrivée des messages sur le nœud récepteur dans l'ordre où ils ont été émis. Aucun mécanisme de contrôle de flux n'est nécessaire puisqu'il s'agit d'un dépôt direct en mémoire distante. Les performances de VMMC sont proches des performances « brutes » du matériel sur la machine SHRIMP. Elles sont moins satisfaisantes sur une grappe Myrinet : la latence est de 10µs et le débit maximum de 100Mo/s [VMMC,1999], ce qui est assez éloigné des performances de BIP.

III.3.7. PM

PM [Tezuka,1997] est une bibliothèque de communication pour le réseau Myrinet. Elle fait partie d'un environnement de programmation parallèle qui comprend, outre PM, un langage de programmation parallèle appelé MPC++, et un système d'exploitation appelé Score. Score est implémenté sous la forme d'un démon s'exécutant au-dessus d'un système d'exploitation UNIX qui se charge de la répartition des processus de l'application sur les différents nœuds de calcul et du multiplexage de l'interface réseau. La technique du *gang scheduling* est utilisée pour partager dans le temps les ressources réseau entre les différents processus de l'application.

PM garantit l'ordre d'arrivée des messages sur le récepteur dans l'ordre où ils ont été émis et fournit un mécanisme de contrôle de flux (appelé « ACK/NACK ») entre chaque couple émetteur/récepteur. Les erreurs réseau ne sont pas corrigées. Deux modèles de communication sont disponibles : le modèle *send/receive* classique et une primitive d'écriture directe en mémoire distante.

Les processus communiquent sur des canaux de communication qui sont partagés entre les différents processus par la technique du *gang scheduling*. A chaque canal de communication est associé un tampon intermédiaire dans la mémoire de la carte réseau. La signalisation est réalisée par une scrutation sur le nœud récepteur à l'aide d'un processus léger. Une technique, appelée « *Immediate Sending* », permet de pipeliner les communications sur le nœud émetteur, comme dans BIP. En revanche, cette technique n'est pas utilisée sur le nœud récepteur car elle est incompatible avec la détection des erreurs réseau (une erreur de CRC ne peut être détectée qu'une fois que tout le message a été reçu).

III.3.8. U-Net

Deux versions majeures de U-Net [UNET] [Eicken,1995] ont vu le jour : U-Net pour ATM qui nécessite un support matériel spécifique, et U-Net pour Ethernet qui fonctionne avec n'importe quel type d'adaptateur Ethernet.

L'intérêt majeur de U-Net, vis à vis de la plupart des autres bibliothèques en mode utilisateur, est la protection et la garantie d'intégrité du système, malgré le choix du mode utilisateur. U-Net pour ATM assure le multiplexage de l'interface réseau par un processeur Intel i960 localisé sur l'adaptateur réseau. L'absence de ce processeur sur les adaptateurs Ethernet empêche U-Net pour Ethernet de communiquer en mode utilisateur : l'API est la même que sur ATM mais un support noyau spécifique réalise le partage des accès au réseau.

Le contrôleur réseau est partagé par des terminaisons (« *end-point* »), distribuées aux différents processus. Chaque terminaison est constituée d'un tampon de mémoire noyau, d'une file d'émission et d'une file de réception, permettant la synchronisation processus/adaptateur. Les tampons de terminaison constituent un stockage intermédiaire pour les opérations d'émission/réception. Le rôle du système d'exploitation se limite à établir des projections mémoire des *end-points* dans l'espace d'adressage des processus constituant l'application.

Les performances de U-Net sont très proches du matériel ATM à 155Mb/s : 44,5µs de latence et 120Mb/s de débit [Welsh,1996].

[Welsh,1997] présente U-Net/MM, une extension de U-Net qui permet de s'affranchir des tampons intermédiaires de stockage des données, en transférant les données directement entre les espaces d'adressage des processus émetteur et récepteur. La version 2.1 de U-Net supporte le réseau Myrinet [UNET].

III.3.9. VIA : Virtual Interface Architecture

VIA correspond à un effort de standardisation des systèmes de communication de niveau utilisateur, proposée par Compaq, Intel et Microsoft [Dunning,1998] [VIA] dont l'objectif est double : fournir une interface de programmation stable et définir un cadre pour les interfaces réseau afin de guider les fabriquants dans la conception de matériels réseau efficaces à moindre coût.

VIA a fortement été influencé par U-Net : le but est de fournir un accès sécurisé de niveau utilisateur aux ressources réseau. Son fonctionnement correspond à des « super sockets ». Un processus communique au travers de VI. Pour pouvoir envoyer/recevoir des messages, une phase de connexion entre deux VI est obligatoire. La première VI attend des demandes de connexion et peut les accepter ou les rejeter. La deuxième émet des demandes de connexion. Une VI est désignée par une adresse et un « *discriminator* », ce qui correspond à une adresse IP et un numéro de port avec les *sockets* UNIX standard. Une fois qu'une connexion est établie, le processus communique directement avec l'interface réseau à travers deux files d'attente, appelées « *work queues* » : une pour l'émission, l'autre pour la réception. Il poste des descripteurs dans ces files. La notification des fins d'émission/réception peut se faire suivant trois mécanismes : en scrutant le descripteur correspondant à la communication attendue, en faisant un appel bloquant qui attend d'être réveillé par le

système d'exploitation, ou en fournissant un *handler* à appeler lors de la complétion de l'opération.

La spécification définit deux types d'opérations de communication : les opérations de type *send/receive* avec l'obligation de poster la réception avant l'arrivée du message sur le récepteur et des opérations de lecture/écriture distante (RDMA).
VIA supporte les communications sans recopie mémoire d'espace utilisateur à espace utilisateur et supporte l'enregistrement dynamique des zones mémoire. Toutes les communications doivent se faire depuis ou vers des zones mémoire enregistrées.

Pour assurer les services définis dans la spécification, une implémentation de VIA doit être composée d'une bibliothèque et d'un module noyau (pour la création et la connexion des VI, la gestion de la mémoire, la réception des interruptions). Il existe différentes implémentations de VIA avec ou sans support matériel. M-VIA [M-VIA] est une implémentation du standard sur des cartes Fast-Ethernet ou Giga-Ethernet sous Linux. Une deuxième version est prévue pour supporter le matériel Giganet, Servernet II et Myrinet. Une implémentation de VIA au-dessus de GM [VI-GM] a été réalisée par la société Myricom. [Chelius,2001] est une autre implémentation du standard au-dessus de GM, initialement réalisée chez Myricom. [Buonadonna,1998] est une implémentation de VIA pour Myrinet avec un support spécifique au niveau du matériel.

III.3.10. PUT

PUT [Fenÿo,2001] est la bibliothèque de communication bas niveau de la machine MPC. Elle fournit une API minimale permettant de faire des écritures distantes depuis un nœud émetteur vers un nœud récepteur via le réseau HSL.

Cette couche de communication se trouve entièrement dans le système d'exploitation. Les applications désirant utiliser PUT doivent faire des appels système pour poster des ordres d'émission. Un ordre d'émission consiste à donner, l'adresse physique et la taille d'un tampon contigu en mémoire physique contenant les données, l'adresse du tampon de réception (lui aussi contigu en mémoire physique), le numéro du nœud destinataire, un identificateur de message appelé MI, et des drapeaux utilisés pour la signalisation. Il n'y a pas de primitive de réception.

La signalisation est réalisée par une interruption matérielle provenant du contrôleur réseau, en émission et en réception. Deux fonctions de rappel se trouvant dans le noyau sont appelées par le gestionnaire d'interruptions de PUT : une associée à la fin d'émission et une associée à la fin de réception.

Les transmissions sont de type zéro-copie : aucun tampon intermédiaire n'est utilisé pendant les phases de communication. Les données sont transmises directement depuis le tampon d'émission du nœud émetteur vers le tampon de réception du nœud récepteur. Le contrôleur réseau accède directement à la mémoire centrale du nœud

hôte par des transferts DMA. Une limitation du matériel fait que la taille maximale des données transportées lors d'une écriture distante est de 64Ko. Si l'application veut transmettre des données d'une taille plus grande, plusieurs écritures distantes sont nécessaires.

Comme les transmissions sont de type zéro-copie, aucun mécanisme de contrôle de flux n'est implanté dans PUT. Aucun mécanisme de retransmission en cas d'erreurs n'est utilisé. Les erreurs sont détectées et provoquent l'arrêt de l'application.

Les accès aux registres internes du contrôleur réseau se font par des accès en configuration à travers le bus PCI. PUT se trouvant dans l'espace système, il réalise le multiplexage des ressources réseau entre les différents processus de l'application. PUT permet l'exécution de plusieurs applications simultanément en leur fournissant un « SAP » (*Service Access Point*).

Les performances intrinsèques de PUT sont très satisfaisantes avec un ping-pong réalisé dans le noyau qui échange des données contiguës en mémoire physique. Cependant, les applications de niveau utilisateur doivent passer par le système d'exploitation lors de chaque communication.

Une implémentation de PUT en mode utilisateur a été réalisée dans le cadre de nos travaux. Elle est présentée au chapitre V.

III.3.11. PAPI : Pci-ddc Application Programming Interface

Une interface en mode utilisateur, appelée PAPI (*Pci-ddc Application Programming Interface*), de la primitive d'écriture distante de la machine MPC a été réalisée par Eric Renault [Renault_3,2000], au laboratoire PRiSM de l'Université de Versailles.

Le but de cette implémentation est l'étude du problème de la sécurité et de l'intégrité du système dans les bibliothèques en mode utilisateur, et leur impact sur les performances [Renault_1,2000]. L'ensemble est composé de différents modules réalisant l'interface entre le système d'exploitation, le matériel spécifique de la machine MPC et l'utilisateur. PAPI intègre la mise en place de différentes formes de protection permettant à l'administrateur de la machine de choisir le meilleur compromis entre rapidité et sécurité. PAPI permet de choisir, lors de la phase de compilation, le niveau de sécurité souhaité, et propose le cas échéant des communications en mode utilisateur ou en mode noyau. [Renault_2,2000] propose deux méthodes de protection des adresses, basées sur une organisation précise des informations relatives aux intervalles de mémoire alloués, et ne nécessitant pas d'échange d'information entre les nœuds lors de la vérification et autorisant l'arithmétique sur les adresses. Les adresses vérifiées peuvent être des adresses virtuelles et physiques.

Comme PUT, le modèle de programmation est le *remote write*. Cependant, PAPI fournit également des primitives de réception : elles se contentent de faire une scrutation pour détecter l'arrivée des messages. Les primitives d'émission travaillent en adresses physiques. Elles transmettent des données se trouvant dans une zone contiguë en mémoire physique vers une zone identique sur le nœud récepteur. PAPI fournit une primitive d'allocation d'un tampon contigu en mémoire physique. Un mécanisme de notification par interruption matérielle est également disponible.

Aucun mécanisme de contrôle de flux n'est nécessaire. Il n'y a pas de retransmission en cas d'erreurs sur le réseau. Les transmissions sont de type zéro-copie. PAPI ne permet pas le partage des ressources de la carte réseau entre plusieurs processus s'exécutant en mode utilisateur. PAPI ne permet pas l'exécution simultanée de plusieurs applications.

Une architecture logicielle générique, appelée GRWA (*Global Remote-Write Architecture*), est en cours de développement et fournira les mêmes services et performances que PAPI dans un environnement pouvant être hétérogène : un portage sur Myrinet est en cours de développement, Ethernet et SCI sont aussi prévus.

III.3.12. Performances des bibliothèques bas niveau

Le Tableau III-1 présente les performances des différentes bibliothèques présentées dans cette section. Les comparaisons sont difficiles car les plates-formes matérielles sont différentes. Par exemple, les performances sur le réseau Myrinet dépendent de la version du matériel utilisé. Cependant, on peut remarquer que les latences de BIP, PUT et PAPI sont très faibles.

Bibliothèque	Plate-forme	Latence (µs)	Débit (Mbits/s)	Référence
AM-II	UltraSPARC/Myrinet	21	248	[Chun,1998]
FM 2.x	PII-300/Myrinet	9	736	[Lauria,1999]
FM/VIA	PII-450 SMP/GigaNet	~10	~640	[Liu,1999]
GM	PIII-666/Myrinet	7	2000	[GM,2001]
BIP	PPro200/Myrinet	4,3	1008	[Prylli,1998]
VMMC	Pentium/Myrinet	10	800	[VMMC,1999]
PM	PPro200/Myrinet	7,5	908	[Buyya,1999] [1]
U-Net	FORE PCA-200 ATM	44,5	120	[Welsh,1996]
	FastEthernet	30	97	
M-VIA	PIII-450/Gigabit Ethernet	16	448 [2]	[Ong,2000]
VIA Berkeley	PII-300/Myrinet	24	425	[Buonadonna,1998]
PUT	PII-350/HSL	5,1	494	[Fenÿo,2001]
PAPI	PII-233/HSL	4,4	492	[Renault,2001]

[1] (Vol. 1, p656) [2] (pour un message de 32Ko)

Tableau III-1 : Performances des bibliothèques bas niveau

Le débit maximum de M-VIA est encore inconnu car la taille maximum des tampons supportés par M-VIA est de 32Ko [Ong,2000].

Les auteurs de [Araki,1998] ont comparé les performances de AM, BIP, FM, PM et VMMC sur la même plate-forme expérimentale, constituée de deux processeurs Pentium Pro cadencés à 200MHz, interconnectés par un réseau Myrinet première génération. La latence varie entre 5µs (BIP) et 17µs (AM) alors que le débit maximum est compris entre 400Mbits/s (AM) et 1000Mbits/s (BIP).

III.4. The Message Passing Interface (MPI)

III.4.1. Le standard

MPI [MPI] offre la possibilité aux programmeurs d'applications parallèles de développer des applications portables dans un environnement facile à utiliser tout en exploitant au mieux les performances des machines parallèles.

La première version du standard (MPI-1) [MPI,1994] définit un grand nombre de primitives aussi bien pour les communications point à point que pour les opérations collectives. Les appels point à point peuvent être bloquants ou non bloquants (pour permettre le recouvrement calcul/communication). Il existe quatre modes différents de communication : *standard, synchrone, bufferisé, ready* (nous reviendrons sur leur sémantique au chapitre IV). MPI offre la possibilité au programmeur de définir ses propres types complexes et d'envoyer des messages depuis des tampons situés dans la mémoire du processus.

MPI permet de définir des sous-groupes de processus au sein d'une même application qui communique à travers un « *communicator* ». Les informations identifiant une requête d'émission ou de réception sont le *communicator*, le *tag* MPI et le numéro de la tâche source ou destinataire. La correspondance entre requêtes d'émission et requêtes de réception se fait suivant quatre règles :

- ✓ Les opérations collectives et point à point sont indépendantes même si elles opèrent dans le même *communicator*.
- ✓ Une réception correspond à une émission si elle s'effectue dans le même *communicator*.
- ✓ Le récepteur précise la source du message qu'il désire recevoir. Il peut décider de recevoir un message depuis un émetteur quelconque au sein de son *communicator*.
- ✓ Les requêtes sont marquées avec un *tag* MPI. Une communication se fait entre les requêtes d'émission et de réception portant sur le même *tag*.

Toutes les communications portant sur le même *communicator*, le même *tag* et le même numéro source/destinataire se font dans un ordre FIFO.

Une deuxième version du standard (MPI-2) [MPI,1997] a été définie pour étendre les fonctionnalités de MPI : création dynamique de nouveaux processus, primitives de communication unidirectionnelles (type *Get/Put*) appelées « *one-sided communications* », primitives d'accès à un système de fichiers parallèle (MPI-IO), etc.

Il existe actuellement différentes implémentations de MPI. Certaines sont dédiées à des architectures particulières alors que d'autres supportent plusieurs plates-formes matérielles. Deux implémentations *open source* majeures, d'origine académique, existent : LAM-MPI [Burns,1994] [LAM] issu de l'*Ohio Supercomputer Center* et MPICH [Gropp,1996] issu de l'*Argonne National Laboratory*. Ces deux implémentations visent principalement les grappes de stations de travail. MPICH, qui est probablement l'implémentation la plus répandue, fonctionne aussi sur de nombreuses machines parallèles. Une comparaison des performances de LAM et de MPICH, sur un *cluster* composé de huit stations DEC 3000/300 interconnectées par un réseau FDDI, est présentée dans [Nevin,1996].

III.4.2. MPICH : une implémentation générique

MPICH [Gropp,1996] est une implémentation organisée en couches pour faciliter le développement de portages sur différentes architectures. Dans l'esprit des auteurs, pour adapter MPICH à une nouvelle architecture logicielle ou matérielle, il faut développer une surcouche au système de communication visé fournissant les fonctionnalités de l'ADI (*Abstract Device Interface*) [Gropp,1994]. L'ADI correspond à un ensemble de primitives de communication point à point pour chacun des modes de communication de MPI.

MPICH fournit une implémentation modèle de l'ADI qui supporte plusieurs canaux de communication. Elle utilise deux types de messages : les messages de contrôle véhiculant des informations relatives aux requêtes postées et les messages de données. Elle définit plusieurs protocoles internes suivant la taille des données : *short* (les données sont incluses dans un message de contrôle), *eager* (les données sont envoyées immédiatement à la suite d'un message de contrôle) et *rendez-vous* (les données ne sont envoyées qu'après que l'émetteur ait reçu un acquittement de la requête d'émission provenant du récepteur). Les données peuvent être sauvegardées dans un tampon intermédiaire sur le nœud récepteur en attendant que la réception correspondante soit postée.

L'architecture de MPICH est en cours de redéfinition [Gropp,2002] pour tenir compte des travaux récents sur les systèmes de communication de niveau utilisateur, ainsi que de l'évolution du matériel réseau.

La majorité des bibliothèques de communication bas niveau présentées dans la section précédente disposent d'un portage de MPI souvent basé sur MPICH. Nous les présentons dans la suite. Ces ports reposent sur les protocoles définis dans l'implémentation de MPI qu'ils utilisent (tels que les protocoles *short*, *eager* et *rendez-*

vous de MPICH) et utilisent des appels à la bibliothèque bas niveau pour transférer les messages MPI. Suivant les particularités du système de communication utilisé, les problèmes à résoudre sont les suivants :

- ✓ découper les messages MPI en fragments si les transferts sur le système de communication sous-jacent imposent une taille maximale,
- ✓ fournir un contrôle de flux pour les messages de contrôle,
- ✓ passer d'une sémantique de communication à passage de messages (celle de MPI) sur canal de communication, à une sémantique différente (accès mémoire distant, Messages Actifs).

III.4.3. MPI/SCI

La première implémentation de MPI sur un réseau SCI est décrite dans [Worringen,1999]. La librairie SMI (*Shared Memory Interface*) [Dormanns,1997] réalise l'interface entre l'ADI de MPICH et l'API SISCI [Giacomini,1999] qui permet de communiquer avec le pilote IRM de l'adaptateur PCI-SCI de la société Dolphin. Elle a consisté en l'écriture d'un nouveau « *device* », appelé « *ch_smi* » pour l'ADI, à partir du *device* « *ch_shmem* » supporté dans MPICH. Dans un premier temps, les auteurs ont remplacé les fonctions d'allocation de mémoire partagée de *ch_shmem* par celle de la librairie SMI. Dans un deuxième temps, il a été nécessaire de réécrire une partie de l'ADI pour obtenir de meilleures performances en considérant la particularité suivante du réseau SCI : les écritures distantes permettent d'atteindre un débit dix fois supérieur aux lectures distantes.

La phase d'initialisation de SCI-MPICH consiste à établir des segments de mémoire partagée projetés dans l'espace d'adressage des processus. Pendant l'exécution de l'application MPI, *ch_smi* utilise les services d'allocation dynamique de mémoire et les barrières de synchronisation fournis par la librairie SMI. Ces services et la phase d'initialisation nécessitent des appels au système d'exploitation. En ce qui concerne les phases de communication, l'adaptateur SCI se contente d'accéder directement à l'espace d'adressage des processus pour transférer les données. Le protocole *short* de MPICH est utilisé pour les messages MPI d'une taille inférieure à 64 octets car ils peuvent être transmis en une seule écriture distante SCI. La transmission de ces messages consiste à recopier les données dans un tampon intermédiaire situé dans un segment de mémoire partagée. A chaque paire émetteur/récepteur est associé un tel tampon. La notification de réception se fait par la scrutation du dernier octet du tampon intermédiaire. Le protocole *eager* de MPICH permet le transfert des messages MPI d'une taille inférieure à quelques Kilo-octets. Il utilise des segments de mémoire partagée SCI réservés pour le protocole *eager*. La transmission consiste à recopier les données dans la zone partagée avec le destinataire sur le nœud émetteur puis à envoyer un message court pour indiquer au récepteur dans quel tampon les données ont été déposées. La réception consiste à retrouver ce tampon et à recopier les données dans le tampon de l'application. Les messages transmis par le protocole *eager* ont une taille maximale car les tampons associés ont une taille fixe : ils ont été alloués lors de la

phase d'initialisation. Le protocole de rendez-vous permet de transmettre les messages d'une taille supérieure en allouant dynamiquement les tampons de mémoire partagée lors de chaque communication. Le principe est le suivant : (1) l'émetteur envoie la taille du message à transmettre au récepteur grâce à un message court ; (2) le récepteur alloue une zone de mémoire partagée et renvoie l'adresse et la taille de la zone à l'émetteur (message court) ; (3) L'émetteur copie le message fragment par fragment dans la zone de mémoire partagée et (4) envoie, après chaque copie d'un fragment, un message court « BLOCK_READY » au récepteur pour lui indiquer qu'il peut recopier le fragment dans le tampon de réception. Le fait de découper le message en fragments permet de faire les recopies sur le nœud émetteur et sur le nœud récepteur simultanément.

[Worringen,2000] présente la deuxième génération de SCI-MPICH qui corrige des problèmes et des limitations de la première implémentation. Par exemple, les émissions asynchrones deviennent plus efficaces grâce à l'utilisation de transferts DMA (la première implémentation utilise des transferts PIO) et des interruptions matérielles. La taille des messages courts n'est plus fixée à 64 octets mais peut être un multiple de cette taille grâce à une modification de leur format. Le portage des MPI-IO du standard MPI-2 sur SCI est également traité.

[Worringen,2001] aborde le problème des transmissions zéro-copie dans SCI-MPICH. Pour faire des transmissions zéro-copie sur un réseau SCI, des tampons contigus en mémoire physique alloués « à l'avance » par le *driver* SCI doivent être utilisés pour les communications. Le problème est que les tampons de communication des applications MPI ne sont pas alloués par le *driver* SCI. Le principe consiste à enregistrer les tampons d'émission et de réception de l'application auprès du pilote SCI avant d'envoyer les données. Les transmissions zéro-copie ne sont utilisées que pour les transferts suivant le protocole de rendez-vous car l'enregistrement des tampons est trop coûteux pour être avantageux avec les protocoles *short* et *eager*. Les appels aux primitives `malloc` et `free` de la librairie C sont interceptés pour les remplacer par une allocation d'un segment de mémoire partagée SCI. Un cache des zones enregistrées est utilisé pour ne faire l'enregistrement qu'une seule fois, pour toutes les communications impliquant les mêmes tampons d'émission et de réception.

Les auteurs de [Worringen,2002] proposent deux optimisations pour MPI/SCI. La première permet des communications efficaces pour les types de données dérivés de MPI qui se trouvent dans des tampons non contigus au niveau de l'application. L'idée est d'assembler les différentes données directement dans le segment de mémoire partagé SCI plutôt que d'utiliser une recopie pour faire l'assemblage. La deuxième optimisation concerne les communications « *one-sided* » définies par le standard MPI-2.

Il existe par ailleurs une implémentation commerciale de MPI pour SCI, réalisée par la société Scali [SCAMPI].

III.4.4. MPI/AM

Une implémentation de MPI sur AM-II est décrite dans [Wong,1999]. Le portage de l'ADI de MPICH a été réalisé au-dessus des Messages Actifs. Un message de contrôle MPI a une taille maximale de 8Ko. Il est transféré directement à l'aide de la requête d'émission des Messages Actifs. Les messages de contrôle sont systématiquement stockés dans un tampon intermédiaire sur le récepteur pour éviter une synchronisation entre l'émetteur et le récepteur. Un algorithme de contrôle de flux à crédits est utilisé pour éviter les débordements.

Les messages d'une taille supérieure à 8Ko sont fragmentés et transférés à l'aide de plusieurs messages actifs. L'émetteur envoie tous les fragments au processus destinataire qui remet en ordre et dépose dynamiquement les fragments dans le tampon de réception. Si la réception correspondante n'a pas été postée, une copie intermédiaire dans un tampon temporaire est réalisée. La notification de réception d'un message MPI a lieu lorsque tous les messages actifs associés ont été traités sur le nœud récepteur.

Les performances de MPI/AM-II [MPI-AM] sur la machine NOW de Berkeley, composée de 105 stations UltraSPARC I model 170 interconnectées par un réseau Myrinet, sont une latence de 42,5µs et un débit maximum de 24,6Mo/s.

III.4.5. MPI/FM

Une première implémentation de MPI/FM est présentée dans [Lauria,1997]. Elle repose sur FM 1.x, la première génération des *Fast Messages*. L'implémentation a consisté à porter certaines primitives de l'ADI de MPICH sur FM 1.x. En émission, la primitive utilisée est `FM_send` qui prend comme paramètres le numéro du destinataire, le *handler* qui doit être exécuté sur le récepteur, l'adresse virtuelle et la taille du tampon d'émission. En réception, la primitive `FM_Extract` est appelée pour vider les files de réception de FM et exécuter le *handler* associé. Dans FM 1.x, lorsque les données arrivent sur le récepteur, le contrôleur réseau les dépose dans des zones intermédiaires (« *DMA buffers* ») verrouillées en mémoire physique à l'avance. Lorsque le *handler* est exécuté, les différentes régions DMA doivent être rassemblées dans le tampon de réception en utilisant une recopie. Par ailleurs, si la réception MPI correspondante n'a pas été postée quand le *handler* est exécuté, il y a une recopie supplémentaire des données dans un tampon intermédiaire. Un des problèmes de la première implémentation de MPI/FM est que la phase d'assemblage se fait systématiquement dans un tampon intermédiaire au niveau de FM même si la réception MPI a été postée, et cela pour deux raisons : (1) FM n'a pas la possibilité de connaître l'adresse du tampon de réception MPI même quand la requête a été postée ; (2) des en-têtes spécifiques à FM ne doivent pas être déposés dans le tampon de réception de l'application. Une recopie sur le nœud émetteur est également systématique dans MPI/FM : une phase d'assemblage a lieu pour construire le message MPI. Il est constitué d'un en-tête MPI et des données de l'application. FM 1.x n'étant

pas capable de transmettre en une seule fois des zones discontinues en mémoire virtuelle, cette recopie est nécessaire.

Les performances de la première version de MPI/FM sont médiocres : le débit atteint au mieux 35% du débit de FM. [Lauria,1997] explique cette perte de performances par le fait que les services fournis par FM 1.x étant insuffisants, de nombreuses recopies intermédiaires des données sont nécessaires. Une étude a donc été réalisée pour fournir l'API FM 2.x en vue d'un portage optimisé de MPI.

[Lauria,1999] présente les performances de la deuxième implémentation de MPI sur cette nouvelle API qui permet l'élimination de nombreuses recopies intermédiaires, grâce, en particulier, à la décomposition d'un message FM en fragments (« *streams* »). La latence obtenue est de 13µs (9 pour FM) et le débit maximum de 91Mo/s (98% du débit maximum de FM), sur une plate-forme constituée de processeurs Pentium-II cadencés à 300MHz et interconnectés par un réseau Myrinet.

III.4.6. MPI/BIP et MPI/GM

Le portage de MPI/BIP, basé sur MPICH, a été réalisé par Loic Prylli [Prylli,1999]. BIP a été conçu en vue du portage de MPI : sa sémantique est très proche de celle de l'ADI de MPICH. Le transfert des messages de contrôle de MPICH utilise les messages courts de BIP. Ils sont transférés en utilisant une recopie intermédiaire sur le nœud émetteur et le nœud récepteur dans des tampons intermédiaires alloués lors du démarrage de l'application. BIP ne fournissant pas de contrôle de flux pour ces tampons, MPI/BIP implémente un mécanisme de contrôle de flux à crédits.

Les messages MPI d'une taille supérieure à la taille maximale des messages de contrôle sont transférés à l'aide des messages longs de BIP, suivant un protocole de rendez-vous. Il se déroule en trois étapes : (i) l'émetteur envoie une requête d'émission à l'aide d'un message court, (ii) le récepteur reçoit la requête, réserve un *tag* BIP pour la transmission des données, réalise la traduction d'adresse du tampon de réception si elle n'a pas déjà été faite lors d'une communication précédente, lui associe le *tag*, et envoie un acquittement à l'émetteur contenant le *tag* à l'aide d'un message court, (iii) l'émetteur envoie les données avec le *tag* choisi par le récepteur. Si la réception MPI n'a pas été postée lorsque le récepteur reçoit une requête d'émission, MPI/BIP alloue un tampon intermédiaire pour recevoir les données. MPI/BIP fragmente le message MPI en plusieurs paquets et BIP réalise un pipeline des communications correspondant à l'émission de ces paquets.

La version de BIP utilisée pour MPI a une latence de 6,5µs et un débit maximum de 126Mo/s [Prylli,1999], sur des processeurs PentiumPro cadencés à 200MHz interconnectés par un réseau Myrinet. Sur la même plate-forme, les performances de MPI/BIP sont une latence de 9µs et un débit de 125Mo/s. Le demi débit est atteint pour un message d'une taille de 8Ko.

Le portage de MPI sur GM est tout à fait similaire à celui de MPI sur BIP : la première version a été réalisée par Loic Prylli en 1999.

III.4.7. MPI/PM

Le portage de MPI sur PM est présenté dans [Ocarroll_1,1998] [Ocarroll_2,1998]. Il s'agit aussi d'une implémentation basée sur MPICH. PM permet deux modes de transfert : un mode classique de type *send/receive* qui utilise des tampons intermédiaires embarqués sur la carte réseau Myrinet et une primitive d'écriture en mémoire distante qui réalise des transferts de type zéro-copie.

Les messages de contrôle dans MPI/PM sont envoyés suivant le mode de transmission classique de PM. Ils ont une taille maximale qui est paramétrable lors du lancement de l'application Cela permet à l'utilisateur de choisir la valeur optimale, celle-ci dépendant en particulier de la plate-forme utilisée.

Les messages d'une taille supérieure sont transmis à l'aide de la primitive d'écriture distante zéro-copie de PM. Pour l'utiliser, l'émetteur doit connaître l'adresse du tampon de réception. Cela implique que la réception correspondante ait été postée et que le récepteur ait informé l'émetteur de l'adresse du tampon destinataire, avant le début du transfert des données. Cela est réalisé grâce à un protocole de rendez-vous qui utilise les primitives de type *send/receive* de PM pour transférer la requête et l'acquittement. Ce dernier transporte l'adresse virtuelle du tampon de réception du récepteur vers l'émetteur. Lorsqu'une requête arrive, le récepteur doit verrouiller en mémoire physique le tampon destinataire. Quand l'émetteur reçoit l'acquittement, il verrouille le tampon d'émission en mémoire physique et réalise le transfert des données en utilisant la primitive d'écriture distante de PM autant de fois que nécessaire (la taille des données transférées par une écriture distante est limitée). Une TLB (*Translation Lookaside Buffer*) logicielle présente dans la mémoire embarquée de la carte réseau contient les conversions d'adresses relatives aux communications.

Pour combler une lacune éventuelle de mémoire physique, deux mécanismes sont implantés dans l'ADI de MPI/PM [Ocarroll_2,1998] :

- ✓ Une taille maximum de mémoire verrouillée est attribuée, d'une part pour les émissions et, d'autre part pour les réceptions. Le fait de distinguer la quantité de mémoire pour les zones de réception et les zones d'émission permet d'être certain de pouvoir faire au moins une émission et une réception simultanément et donc d'éviter des situations de blocages.
- ✓ Quand la quantité maximum de mémoire verrouillée pour les réceptions est atteinte, une file d'attente permet de retarder les communications concernées en attendant que des zones de mémoire se libèrent. Cette file est régulièrement consultée pour garantir l'avancement des réceptions. A l'opposé, les émissions ne sont jamais retardées : MPI/PM fait en sorte qu'au moins une page de mémoire

peut être verrouillée. S'il n'y a pas assez de mémoire, les émissions sont réalisées en plusieurs fois grâce à la page disponible.

La version de PM utilisée pour MPI a une latence de 7,5µs et un débit maximum de 113,5Mo/s [Buyya,1999 (Vol. 1, p656)], sur des processeurs Pentium Pro cadencés à 200MHz interconnectés par un réseau Myrinet. Sur la même plate-forme, les performances de MPI/PM sont une latence de 13,2µs et un débit de 104Mo/s. Ces performances correspondent au meilleur des cas, c'est-à-dire quand le manque de mémoire physique n'intervient pas.

III.4.8. MPI/VIA

[Dimitrov,1998] présente un implémentation de MPI sur VIA, appelée MPI/pro, pour une ferme de stations Windows NT. MPI/Pro est une implémentation *multi-threads* qui permet de réaliser simultanément des communications sur des VI et des communications SMP. Elle a été testée sur une ferme de 72 stations SMP sous NT interconnectées par un réseau ServerNet, au Sandia National Laboratory. L'implémentation utilise des processus légers pour réaliser la signalisation, permettant ainsi un meilleur recouvrement calcul/communication. Un processus léger est en charge des communications sur les VI et un autre s'occupe des communications SMP. Contrairement à MPICH qui n'utilise que deux files d'attente en réception (une pour les messages attendus et une autre pour les messages inattendus), MPI/Pro implante plusieurs files de réception : quatre par processus (deux pour les communications sur les VI et deux pour les communications SMP). Cela permet de réduire le temps pour associer l'arrivée d'un message à la requête de réception correspondante, et d'éliminer des points de synchronisation. MPI/Pro permet de choisir le protocole de communication optimisé suivant la taille du message et les caractéristiques du *device* utilisé (VI ou SMP). Les messages courts sont transmis en utilisant des recopies intermédiaires. Quand les recopies deviennent trop pénalisantes, la primitive d'écriture distante de VIA (VI RDMA) est utilisée suivant un protocole de rendez-vous qui permet au récepteur de transmettre l'adresse du tampon de réception. Enfin, MPI/Pro optimise les transmissions des types de données dérivés qui correspondent à des zones discontinues en mémoire virtuelle.

[Liu,1999] présente un portage des *Fast Messages* sur VIA au-dessus du réseau GigaNet. Cela permet de bénéficier du port de MPI sur FM. L'implémentation de FM sur VIA consiste à établir les connexions entre VI lors de l'initialisation de l'application : une connexion est créée entre chaque paire de processus. Pour identifier un processus, le couple (adresse réseau, *discriminator*) associé à la connexion est utilisé. A chaque connexion d'un processus, FM/VIA associe une file de descripteurs. A chacun de ces descripteurs correspond un tampon pré-alloué de taille fixe. Des recopies intermédiaires dans ces tampons sont faites lors de chaque émission et lors de chaque réception. La primitive d'écriture distante de VIA n'est pas utilisée.

[Bertozzi,2001] [Bertozzi,1999] présentent une implémentation de LAM/MPI sur M-VIA dans le cadre du projet PARMA^2 [PARMA^2] de l'Université de Parme. Afin de permettre les émissions asynchrones définies par le standard MPI, des descripteurs VIA sont postés « à l'avance » dans les files de réception avant l'établissement d'une connexion. Ces descripteurs déjà consommés sont réutilisés pour des communications ultérieures afin de permettre un nombre d'émission/réception supérieur au nombre de descripteurs « pré-postés ». Les tampons intermédiaires (« registered memory ») de VIA sont également réservés lors de l'initialisation et sont réutilisés pour plusieurs communications. Les tampons de l'application sont recopiés dans ces zones intermédiaires lors de chaque communication, en émission comme en réception. Lors de la phase d'initialisation, les adresses des tampons intermédiaires distants sont échangées. Pour les communications, la primitive RDMA de VIA est utilisée selon le protocole suivant : (1) les données à émettre sont recopiées dans un tampon intermédiaire sur l'émetteur ; (2) un descripteur d'émission est initialisé puis posté dans la file d'émission associée à la connexion VI correspondant au processus récepteur ; (3) l'écriture distante est réalisée dans le tampon RDMA de réception ; (4) le récepteur scrute l'ensemble de ses files de réception ; si un descripteur est trouvé, alors les données ont été écrites dans le tampon RDMA associé, sinon le récepteur peut choisir d'attendre l'arrivée d'un message ou de sortir de la fonction ; (5) les données sont recopiées depuis le tampon RDMA vers le tampon de l'application ; (6) le descripteur associé est libéré puis replacé dans la file pour une autre réception. Un mécanisme de contrôle de flux est implanté au niveau de MPI pour prévenir du manque de ressources (tampons RDMA et descripteurs pré-postés). En particulier, quand un émetteur envoie un message, il doit s'assurer de disposer d'un tampon de réception. Les performances obtenues sur un réseau Fast Ethernet montrent que le portage de LAM/MPI sur M-VIA permet de réduire la latence de 45% par rapport à celle de LAM/MPI sur TCP/IP. Cependant, en terme de débit, il n'y a aucune amélioration significative du fait des recopies intermédiaires. Les auteurs prévoient des solutions pour les éviter mais elles supposent une réécriture de presque tout le code de LAM/MPI.

Les auteurs de [Ong,2000] font une comparaison entre LAM/MPI [BURNS,1994] [LAM], MPICH [Gropp,1996] et MVICH [MVICH] sur un *cluster* LINUX constitué de processeurs Pentium III cadencés à 450MHz, interconnectés par un réseau Gigabit Ethernet. LAM/MPI et MPICH utilisent les sockets UNIX standard alors que MVICH est une implémentation spécifique de MPI pour VIA qui utilise M-VIA [M-VIA] pour accéder au réseau Gigabit Ethernet. La latence de MVICH est de 26µs alors qu'elle est de 16µs pour M-VIA. Les conclusions sont que les performances de LAM/MPI et MPICH sont médiocres du fait de la traversée de la pile de protocoles standard TCP/IP alors que les performances de MVICH sont encourageantes bien que l'implémentation ne soit pas encore optimisée. Les performances de MVICH présentées sur le site internet [MVICH] du *National Energy Research Scientific Computing Center* sont une latence de 13,5µs et un débit maximum de 776Mbits/s sur un réseau GigaNet (la cadence des processeurs n'est pas précisée).

[Brightwell,2000] discute l'utilisation de VIA pour le développement d'une implémentation de MPI pour une ferme de stations de travail de plusieurs milliers de processeurs (dans le cadre du projet CPLANT de Sandia National Labs). Les auteurs soulignent plusieurs limites de la spécification principalement liées à la nature orientée connexion de VIA : le nombre de VI nécessaires est égal au carré du nombre de processus (pour que tous les processus puissent s'échanger des messages) et des zones mémoires doivent être associées à chacune des VI. [Liu,1999] arrive aux mêmes conclusions.

III.4.9. Performances des portages de MPI

Le Tableau III-2 présente les performances trouvées dans la littérature concernant les différents portages de MPI que nous avons présentés dans cette section. Les performances de MPI/GM sont en cours de préparation et ne sont pas encore disponibles sur [GM,2001].

Bibliothèque	Plate-forme	Latence (µs)	Débit (Mbits/s)	Référence
MPI/AM-II	UltraSPARC/Myrinet	42.5	197	[MPI-AM]
MPI/AM	UltraSPARC/Myrinet	16,7	309	[MPI-AM]
MPI/FM 2.x	PII-300/Myrinet	13	728	[Lauria,1999]
MPI/BIP	PPro200/Myrinet	9	1000	[Prylli,1999]
MPI/PM	PPro200/Myrinet	13,2	832	[Buyya,1999] [1]
MVICH/MVIA	PIII-450/Gigabit Ethernet	26	280 [2]	[Ong,2000]
MPIPro/VIA	PII-400/Giganet	21	~1500	[Dimitrov,1998]
MPI/FM/VIA	PII-450 SMP/Giganet	~15	~600	[Liu,1999]
LAM/MVIA	PII-450 SMP/FastEthernet	63	80	[Bertozzi,2001]
SCI-MPICH	PII-450/SCI	6,6	590	[Worringen,1999]
ScaMPI	PII-400/SCI	16,2	525	[Worringen,1999]

[1] (Vol. 1, p656) [2] (pour un message de 32Ko)

Tableau III-2 : Performances des ports de MPI

Une comparaison des performances de MPI/FM/VIA/GigaNet, MPI/FM/Myrinet et MPI/Pro/GigaNet est réalisée dans [Liu,1999]. Les performances de MPI/Pro sont légèrement inférieures à celles de MPI/FM/VIA. En revanche, les performances de MPI/FM sur Myrinet sont meilleures que celles de MPI/FM sur GigaNet.

[Prylli,2000] présente une comparaison des implémentations et des performances de MPI/BIP, MPI/GM, MPI/PM et MPI/p4 (MPICH sur FastEthernet). [Aumage_2,2001] présente une implémentation de MPICH sur Madeleine-II. Le but est de permettre l'accès à plusieurs réseaux simultanément à travers le même *device* au niveau de MPICH, appelé « ch_mad », sans pénaliser pour autant les performances de l'application. Cela est possible grâce à Madeleine-II [Aumage_1,2001], une librairie de communication « multi-protocoles » pour des applications *multi-threads*, capable

de contrôler plusieurs protocoles de communication (BIP, SISCI, VIA) sur différents réseaux (Ethernet, Myrinet, SCI) au sein d'une même application. Elle est capable de sélectionner dynamiquement, pour un réseau donné, la meilleure méthode de transfert suivant plusieurs paramètres tels que la taille des données ou les besoins de l'utilisateur. Les performances de MPICH/Madeleine sont comparées à ScaMPI, SCI-MPICH, MPI/GM et MPI/PM.

III.5. Synthèse

III.5.1. Le problème des traductions d'adresse

Dans les réseaux rapides comme Myrinet, HSL ou SCI, le contrôleur réseau utilise des accès directs en mémoire physique (DMA) pour transférer les données. Cela permet des transmissions de type zéro-copie depuis la mémoire virtuelle du processus émetteur vers celle du processus récepteur à condition de verrouiller les tampons de communication en mémoire physique et d'établir la correspondance entre l'adresse virtuelle du tampon et les adresses physiques correspondantes. Cependant, le coût de la mise en place du DMA (verrouillage et conversion d'adresse) est important. Cette opération est souvent appelée « enregistrement d'un *buffer* ». Différentes stratégies permettent d'en minimiser le coût. Une technique très répandue consiste à utiliser une zone de transit permanente : une zone verrouillée une fois pour toute et contiguë en mémoire physique est allouée lors du démarrage de l'application. Tous les DMA se font depuis/vers cette zone. Une copie mémoire est alors nécessaire pour passer les données depuis/vers le tampon de l'application. Une autre technique consiste à enregistrer dynamiquement les tampons de l'application concernés par la communication afin de faire des transferts réellement zéro-copie. Enregistrer un tampon de l'application consiste à verrouiller les pages qui le composent en mémoire physique et obtenir leurs adresses physiques. Cela nécessite un appel au système d'exploitation. Une troisième technique parfois utilisée est de ne pas « désenregistrer » les zones lorsque la communication se termine pour ne pas avoir à refaire l'enregistrement lors d'une communication ultérieure. Un cache logiciel permet de conserver les zones déjà enregistrées. Certains systèmes prévoient un mécanisme permettant de désenregistrer des pages quand il n'y a plus de mémoire physique disponible. Enfin, des systèmes s'arrangent pour que ce cache soit accessible directement par le contrôleur réseau lors des transferts DMA. D'une part, cela permet à l'application de poster des émissions/réceptions en communiquant uniquement l'adresse virtuelle du tampon de communication. C'est le contrôleur réseau qui se charge d'établir la correspondance avec les adresses physiques associées. D'autre part, cela permet au récepteur d'associer un identificateur logique à un tampon de réception. Cet identificateur est utilisé par l'émetteur pour faire les écritures distantes. Il n'a donc pas besoin de connaître les adresses physiques du tampon destinataire. Dans la suite, nous analysons les différentes techniques utilisées par les bibliothèques de communication présentées dans ce chapitre.

Dans FM, les données sont systématiquement déposées dans des zones intermédiaires, appelées « *DMA buffers* », par le contrôleur réseau. Ces régions DMA sont allouées et verrouillées en mémoire physique à l'avance par FM. Il y a donc une recopie systématique sur le nœud récepteur depuis les régions DMA vers le tampon de réception de l'application. De la même manière, AM-II utilise une copie en émission et en réception pour les longs messages (« *bulk messages* »).

GM ne peut transférer des données que si les tampons d'émission/réception ont été « enregistrés » au préalable, c'est-à-dire qu'ils sont verrouillés en mémoire physique et que le contrôleur réseau est informé de la correspondance entre les adresses virtuelles/physiques. Cela est pénalisant pour la transmission de messages courts car la préparation des tampons est beaucoup plus coûteuse qu'une simple recopie par exemple. GM donne la possibilité à l'application de s'allouer un tampon contigu en mémoire physique dans la limite de ce que le système d'exploitation peut fournir.

BIP utilise des recopies intermédiaires pour les messages courts pour s'affranchir du coût de la préparation du DMA. Pour les messages longs, BIP utilise un module noyau pour verrouiller les données en mémoire et réaliser la traduction d'adresse de façon transparente, sans appel à une fonction particulière. La liste des pages verrouillées en mémoire est gardée dans un cache logiciel ainsi que les correspondances entre les adresses virtuelles/physiques. Plusieurs communications consécutives sur un même tampon ne déclencheront généralement qu'une seule opération d'enregistrement du DMA. Pour enlever du cache les pages d'un tampon qui n'appartient plus au processus, BIP intercepte les appels aux primitives de gestion de la mémoire de la bibliothèque C (`malloc`, `free`, `mmap` ou des primitives de plus bas-niveau).

VMMC [Dubnicki,1997] nécessite d'importer/exporter les zones de communication avant de réaliser une écriture distante. A chaque processus est associée une table appelée « *User-managed TLB* (UTLB) » qui contient la correspondance entre les adresses virtuelles/physiques des zones de communication. Cette table est stockée dans la mémoire du noyau (cf. Figure III-8). Pour des raisons de sécurité, la bibliothèque VMMC ne peut pas modifier les entrées de la table en mode utilisateur. Elle garde un cache des entrées stockées dans le noyau pour savoir quelles zones mémoire sont présentes dans la UTLB. Une scrutation de ce cache est effectuée lors de chaque émission/réception. Si le tampon associé n'est pas présent dans le cache, un appel système est effectué pour l'ajouter dans la UTLB et faire les opérations de verrouillage/ traduction de la zone. Un cache de la UTLB est également stocké dans la mémoire de la carte réseau qui est accessible plus rapidement par le contrôleur réseau lors des opérations de DMA.

Lors d'une communication, si le tampon d'émission/réception est présent dans la UTLB, le surcoût est évalué à 1,5µs. S'il n'est pas présent dans la UTLB, le coût des opérations de verrouillage/traduction est évalué à plus de 50µs (il dépend du nombre de pages mémoire correspondant au tampon). Enfin, s'il n'est pas présent dans le cache du contrôleur réseau, la mise à jour du cache est évaluée à 3 ou 4µs.

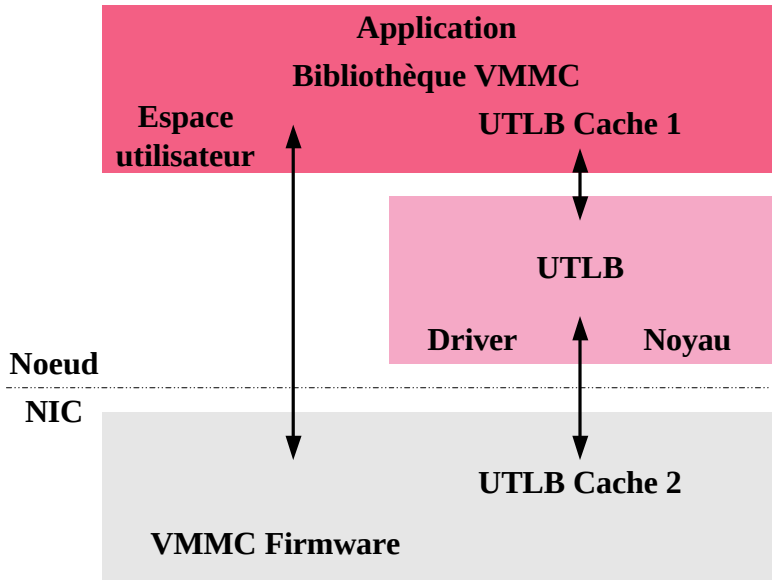


Figure III-8 : Les traductions d'adresse dans VMMC-2

PM permet deux modes de transfert : un mode classique de type *send/receive* qui utilise des tampons intermédiaires embarqués sur la carte réseau Myrinet et une primitive d'écriture en mémoire distante qui réalise des transferts de type zéro-copie. Les tampons intermédiaires, appelés « *pinned-down areas* », ne peuvent pas être évincés de la mémoire physique. Pour le transfert de gros messages, plutôt que de faire des recopies intermédiaires dans les tampons dédiés, [Tezuka,1998] propose une technique appelée « *pin-down cache* » qui permet de faire des transferts de type zéro-copie. Le principe consiste à conserver dans un cache logiciel les zones déjà verrouillées lors des transferts précédents afin de ne pas refaire l'opération coûteuse de verrouillage/conversion si les mêmes tampons sont transférés plusieurs fois. L'API de PM fournit des primitives permettant de verrouiller/déverrouiller des zones de mémoire virtuelle. Si l'application demande la libération d'un zone, celle-ci n'est pas déverrouillée tant que la taille maximum des zones verrouillées n'est pas dépassée. Dans le cas contraire, des zones sont libérées suivant la méthode LRU (*Least Recently Used*) de gestion des caches. Ce cache des zones verrouillées est conservé dans une table dans le système d'exploitation. Elle est accessible directement par le contrôleur réseau de Myrinet lors des transferts DMA pour faire la correspondance entre les adresses virtuelles et les adresses physiques. Cette table contient, dans la version courante de PM, 1024 entrées ce qui permet dans le pire des cas de « cacher » au plus 4Mo de mémoire physique. Le coût de verrouillage, traduction, insertion dans le cache, déverrouillage, suppression du cache est évalué à 40+7*(le nombre de pages)µs dans [Tezuka,1998], sur un processeur cadencé à 200MHz.

U-Net/MM [Welsh,1997] utilise une approche similaire mais moins performante : une TLB (*Translation Lookaside Buffer*) globale est stockée dans la mémoire de la carte réseau ; quand une entrée n'est pas présente dans la TLB, le contrôleur réseau interrompt le processeur hôte qui verrouille la zone en mémoire et lui retourne les adresses physiques correspondantes. Ce mécanisme d'interruption est coûteux lorsque

les tampons de communication ne se trouvent pas dans le cache. Or, une étude autour de U-Net/MM a montré que dans les applications réelles, un défaut de cache a lieu pour 35% des communications.

VIA impose que tous les tampons de communication soient verrouillés en mémoire physique. L'application peut, soit utiliser des tampons de communication pré-enregistrés (« *registered memory* »), alloués par un appel à `VipRegisterMemory`, soit réaliser le verrouillage et la traduction d'adresse à la volée. Les performances des deux stratégies sont évaluées dans [Liu,1999] pour différentes tailles de message. Dans les deux cas, les adresses physiques des tampons de communication sont stockées dans une table appelée « *Translation and Protection Table (TPT)* » stockée dans la mémoire de la carte réseau. Un gestionnaire de mémoire verrouillée, appelé « *Locked Memory Manager* (LMM) » [Seifert_2,2001], a été développé à l'Université Technologique de Chemnitz dans le cadre d'une implémentation de VIA. Il s'agit d'un module noyau LINUX qui permet de verrouiller/déverrouiller les tampons de l'application et de retrouver efficacement leurs adresses physiques à l'aide d'un mécanisme appelé « *kiobufs* ». Un *kiobuf* décrit un ensemble de pages physiques verrouillées appartenant à une zone de mémoire virtuelle contiguë d'un processus. [Seifert_1,2001] explique comment le pilote de l'adaptateur PCI-SCI a été modifié pour permettre d'enregistrer dynamiquement des tampons de l'application dans l'espace mémoire partagée de SCI afin de permettre des transmissions de type zéro-copie. L'enregistrement des zones mémoire utilise le gestionnaire de mémoire LMM. Les auteurs soulignent que l'opération consistant à préparer et réaliser tous les transferts DMA relatifs à la description en mémoire physique des tampons d'émission et de réception induit une chute du débit maximum de 20% par rapport au cas où chaque tampon de communication correspond à une zone contiguë en mémoire physique. Cependant, cette solution évite deux recopies mémoire.

PUT et PAPI ne fournissent aucun service de traduction d'adresse : les primitives d'émission utilisent des adresses physiques.

A moins d'utiliser des recopies intermédiaires, la plupart des bibliothèques de communication en mode utilisateur, basées sur des transferts DMA, nécessitent un mécanisme coûteux d'enregistrement dynamique des tampons d'émission/réception lors de chaque communication. Des bibliothèques comme GM, PM, VMMC ou U-Net utilisent un cache accessible aussi bien par l'application que par le contrôleur réseau pour réduire ce coût en évitant de répéter les opérations de verrouillage/conversion lorsque le même tampon est utilisé plusieurs fois lors des communications.

III.5.2. Communications en mode utilisateur

[Oliveira,2000] fait une étude comparative des APIs bas niveau pour les réseaux Myrinet (BIP, GM, FM) et SCI (SISCI, SMI et YASMIN). [Araki,1998] compare AM, BIP, FM, PM, VMMC. Enfin, [Bhoedjang,1998] propose une comparaison détaillée des implémentations des bibliothèques pour le réseau Myrinet : AM-II, FM,

PM, VMMC, BIP et U-Net. Nous récapitulons ici, quelques caractéristiques des bibliothèques de communication que nous avons présentées dans ce chapitre.

Modèle de communication :
La spécification de VIA définit deux types d'opérations de communication : les opérations de type *send/receive* avec l'obligation de poster la réception avant l'arrivée du message sur le récepteur et des opérations de lecture/écriture distantes (RDMA). Toutes les primitives d'émission et de réception sont non bloquantes pour permettre un recouvrement calcul/communication. GM, PM, VMMC, PUT et PAPI fournissent une primitive d'écriture en mémoire distante. Avec GM, cette fonctionnalité n'est pas disponible par défaut pour des raisons de sécurité : l'application doit appeler une fonction particulière avant de faire des écritures distantes. GM et PM fournissent également un modèle de communication de type *send/receive*. AM et FM reposent sur des messages actifs qui permettent de déclencher l'exécution d'un *handler* sur le récepteur, une fois la réception terminée.
Partage des ressources réseau :
L'intérêt majeur de U-Net, vis à vis de la plupart des autres bibliothèques en mode utilisateur, est la protection et la garantie d'intégrité du système, malgré le choix du mode utilisateur. Cependant, cela se traduit par un stockage intermédiaire des données dans les tampons associés aux « *end-points* ». Les *end-points* sont projetés dans l'espace d'adressage des processus constituant l'application. U-Net pour ATM assure le multiplexage de l'interface réseau par un processeur Intel i960 localisé sur l'adaptateur réseau. L'absence de ce processeur sur les adaptateurs Ethernet empêche U-Net pour Ethernet de communiquer en mode utilisateur : l'API est la même que sur ATM mais un support noyau spécifique réalise le partage des accès au réseau. Avec BIP, la mémoire de la carte réseau est projetée dans la mémoire de chaque processus constituant l'application. Il est impossible d'exécuter plusieurs applications simultanément. Pour permettre le partage de l'interface réseau entre plusieurs processus d'une même application, un mécanisme d'exclusion mutuelle est utilisé au niveau de l'hôte pour multiplexer les ressources. AM, FM, GM et VMMC projettent dans l'espace mémoire de chaque processus les ressources de la carte réseau afin de réaliser des communications en mode utilisateur et d'éviter des copies intermédiaires des données. Cependant, un mécanisme d'exclusion mutuelle fait qu'un seul processus à la fois peut communiquer avec la carte réseau à un instant donné. PM est inclus dans un environnement de programmation parallèle qui supporte plusieurs applications simultanément. Un démon est en charge du multiplexage des accès réseau. Il utilise la technique du *gang scheduling* pour permettre le partage dans le temps des ressources réseau entre les différents processus de l'application : à un instant donné, un seul processus peut accéder à l'interface réseau, ce qui élimine les problèmes de protection des ressources. PAPI est mono-utilisateur : il n'y a pas de problème de partage de ressources. PUT étant une couche de communication en mode noyau, celui-ci assure le partage des ressources entre les différentes applications.
Transmissions de type zéro-copie :
Une des caractéristiques des bibliothèques fournissant une primitive d'écriture en mémoire distante (GM, PM, VMMC, PUT et PAPI) est la nécessité de connaître l'adresse du tampon de réception sur l'émetteur avant de commencer un transfert. Cela

n'est pas adapté au modèle de communication classique de type *send/receive*. Deux stratégies sont possibles : utiliser un tampon intermédiaire de stockage des données sur le nœud récepteur ou faire des transmissions de type zéro-copie basées sur un protocole de rendez-vous qui permet au récepteur de transmettre une information à l'émetteur servant à identifier le tampon de réception. BIP utilise la première stratégie pour les messages courts et la deuxième pour les messages longs. GM associe un tampon de réception à un tampon d'émission à l'aide de la notion de « port ». Les auteurs de [Dubnicki,1997], qui ont implanté VMMC sur le réseau Myrinet, proposent un mécanisme, appelé « *transfer redirection* », qui permet de faire des transmissions zéro-copie sans utiliser de message supplémentaire du récepteur vers l'émetteur. L'idée est de faire systématiquement une émission vers un tampon intermédiaire appelé « *default buffer* ». Lorsque le contrôleur réseau récepteur reçoit un paquet du message, il regarde si la réception correspondante a été postée. Si tel est le cas, il réoriente les données vers le tampon de réception de l'application, sinon il dépose le paquet dans le *defaut buffer*. Une recopie sera alors nécessaire quand la réception aura été postée. Les paquets arrivant après que la réception ait été postée ne transitent pas par le tampon intermédiaire. Le coût pour poster la réception est évalué à 6µs, le temps supplémentaire induit au niveau matériel est de 2.5µs par paquet traité. Le standard VIA supporte les communications sans recopie mémoire d'espace utilisateur à espace utilisateur et supporte l'enregistrement dynamique des zones mémoire. Toutes les communications doivent se faire depuis ou vers des zones mémoire enregistrées au préalable. PUT et PAPI fournissent une primitive d'écriture distante extrêmement simplifiée : ils ne savent transmettre des données que d'une zone contiguë en mémoire physique (sur le nœud émetteur) vers une zone contiguë en mémoire physique sur le nœud récepteur. L'émetteur doit connaître l'adresse physique du tampon de réception avant de démarrer une transmission.

Signalisation :

La notification des fins d'émission/réception dans VIA peut se faire suivant trois mécanismes : en scrutant le descripteur correspondant à la communication attendue, en faisant un appel bloquant qui attend d'être réveillé par le système d'exploitation, ou en fournissant un *handler* à appeler lors de la complétion de l'opération. Dans BIP et GM, la signalisation est réalisée par une scrutation des *tags* (BIP) ou des *ports* (GM). Dans PM, un processus léger prend en charge la scrutation des canaux de communication. AM-II, VMMC et U-Net utilisent, selon les cas, une signalisation par interruption ou par scrutation. PUT utilise les interruptions matérielles en émission et en réception. PAPI fournit des primitives de réception permettant une signalisation par scrutation mais permet aussi la signalisation par interruption.

Contrôle de flux :

Dans U-Net, aucun mécanisme de contrôle de flux n'est implémenté et des données peuvent être perdues en cas de débordement dans les tampons de réception. Dans VMMC-2, aucun mécanisme de contrôle de flux n'est nécessaire car la réception doit en principe être postée avant que l'émetteur envoie les données. Si tel n'est pas le cas, les données sont stockées dans un tampon intermédiaire sur le récepteur. S'il y a un débordement dans ce tampon, le message est ignoré en réception et l'émetteur doit le renvoyer. BIP ne fournit aucun mécanisme de contrôle de flux pour les messages courts : le problème est reporté au niveau de l'application ou des couches de plus haut

niveau. PUT et PAPI n'ont pas besoin de mécanisme de contrôle de flux : le contrôle de flux assuré sur le lien physique par le matériel suffit. AM-II, FM et PM fournissent un mécanisme de contrôle de flux.

Retransmission en cas d'erreurs réseau :

GM et VMMC utilisent un mécanisme de retransmission des paquets erronés au niveau du matériel réseau, basé sur une numérotation et une mémorisation des paquets d'un même message sur le nœud récepteur et un acquittement des paquets correctement reçus. Les auteurs de VMMC [Dubnicki,1997] évaluent le coût de la retransmission à une augmentation de la latence de 2µs et perte de débit entre 2% et 10%. BIP, PM, FM, PUT et PAPI ne fournissent aucun mécanisme de reprise sur erreur.

Sécurité des communications :

L'intérêt majeur de U-Net, vis à vis de la plupart des autres bibliothèques en mode utilisateur, est la protection et la garantie d'intégrité du système, malgré le choix du mode utilisateur. Dans BIP et PUT, l'application passe les adresses physiques au contrôleur réseau sans aucune vérification de leur validité. Contrairement à BIP ou PUT, GM, VMMC et PAPI permettent des communications en mode utilisateur sécurisées : un processus de l'application ne peut pas écrire n'importe où dans la mémoire physique du nœud récepteur. Le but de PAPI est l'étude du problème de la sécurité et de l'intégrité du système dans les bibliothèques en mode utilisateur, et leur impact sur les performances. PAPI permet de choisir, lors de la phase de compilation, le niveau de sécurité souhaité, et propose le cas échéant des communications en mode utilisateur ou en mode noyau. [Renault_2,2000] propose deux méthodes de protection et vérification des adresses utilisées lors des communications. Des méthodes statiques et dynamiques permettent, d'une part de produire une adresse sécurisée lors de l'allocation du bloc de mémoire contiguë et, d'autre part de vérifier que cette adresse est correcte lors de son utilisation. L'administrateur de la machine peut choisir parmi différentes méthodes celle qu'il désire pour réaliser l'authentification de l'adresse. Ensuite, il peut aussi choisir parmi différentes méthodes pour encrypter ces informations tout en garantissant l'arithmétique sur les adresses. Par ailleurs, VMMC, AM-II, GM et FM2.x assurent un partage sécurisé des ressources réseau.

Contrairement à beaucoup de bibliothèques de communication utilisant un réseau Myrinet, la primitive d'écriture distante de la machine MPC est simplifiée à l'extrême et fournit un service minimum : elle ne dispose pas de primitive de réception et travaille exclusivement en adresses physiques avec un contrôleur réseau non programmable qui ne dispose d'aucune intelligence particulière pour traiter les traductions d'adresse par exemple. Par opposition, le contrôleur réseau de Myrinet étant programmable, des bibliothèques de communication comme U-Net, VMMC, PM, GM ou BIP peuvent y stocker la correspondance entre les adresses virtuelles et les adresses physiques. Ces protocoles (à l'exception de BIP) utilisent même un cache logiciel accessible directement par le contrôleur réseau. Cela permet non seulement d'accélérer la traduction d'adresse mais aussi de réaliser des communications sur des canaux virtuels (un *tag* dans BIP, un *port* dans GM ou l'adresse virtuelle du tampon de réception dans PM). L'émetteur n'a donc pas besoin de connaître les adresses physiques du tampon de réception de l'application. Par ailleurs, BIP, GM et PM fournissent des primitives de réception qui permettent une signalisation par scrutation.

Chapitre IV : Architecture de MPI sur une primitive d'écriture distante

Sommaire

L'objectif général de ce chapitre est de montrer comment implanter la bibliothèque de programmation parallèle MPI au-dessus d'une primitive d'écriture en mémoire distante.

IV.1. Introduction

MPI (*Message Passing Interface*) s'est imposé ces dernières années comme le standard de programmation parallèle à passage de messages. Le MPI Forum [MPI] s'est appuyé sur l'expérience de chercheurs académiques et industriels afin de définir la syntaxe et la sémantique d'un ensemble de fonctions de communication regroupées dans le standard MPI. MPI ouvre l'opportunité pour le programmeur d'applications parallèles de développer des applications portables tout en exploitant au mieux les performances des machines parallèles.

L'objectif de ce chapitre est de réaliser une première implémentation du standard MPI au-dessus d'une couche basse de communication basée sur une primitive d'écriture en mémoire distante telle que celle décrite à la section II.2. La difficulté est de faire le lien entre les primitives de communication haut niveau définies dans le standard MPI et la primitive d'écriture distante, en réduisant le coût de traversée des différentes couches logicielles. Par exemple, on cherche à conserver le caractère zéro-copie des transmissions des longs messages. Un autre objectif de ce chapitre est de définir une API générique, appelée RDMA (*Remote DMA*), caractérisant n'importe quel réseau physique fournissant une primitive d'écriture en mémoire distante.

Le cadre des travaux présentés dans ce chapitre est le suivant :

- ✓ La couche basse de communication réalisant la primitive d'écriture distante se trouve dans le noyau du système d'exploitation.
- ✓ Le système d'exploitation utilisé est un système UNIX standard : Linux ou FreeBSD.
- ✓ La signalisation des événements réseau se fait par interruption matérielle provenant du contrôleur réseau.
- ✓ Le réseau est supposé fiable. Nos couches de communication ne doivent pas corriger les erreurs ou pertes de données éventuelles.
- ✓ Le réseau est FIFO : entre deux nœuds i et j, les messages arrivent sur le nœud récepteur j dans l'ordre où ils ont été émis par le nœud émetteur i.
- ✓ Les données sont déposées en mémoire sur le nœud récepteur sans aucun contrôle possible de la part du processeur local. Pour réaliser un transfert, le nœud émetteur doit savoir à l'avance où déposer les données sur le nœud récepteur.
- ✓ Le contrôleur réseau utilise des accès DMA en lecture et écriture pour accéder à la mémoire du nœud hôte. Lors d'une écriture distante, le tampon d'émission et le tampon de réception associé doivent donc être contigus en mémoire physique.
- ✓ Une application MPI est constituée de plusieurs tâches (ou processus) MPI communiquant entre elles. Nous supposons qu'il ne peut s'exécuter qu'une seule

application MPI à la fois sur la machine parallèle et que les tâches s'exécutent sur des nœuds distincts. Nous n'aborderons donc pas dans ce manuscrit le problème des communications *intra-node*.

Ce chapitre est composé de six parties. Il existe plusieurs implémentations du standard MPI. Nous expliquons pourquoi nous avons choisi MPICH [Gropp,1996] comme point de départ et présentons ses caractéristiques. Nous étudions les problèmes spécifiques posés par l'utilisation de la primitive d'écriture en mémoire distante. Nous définissons une API générique, appelée RDMA, qui fournit une abstraction suffisante des caractéristiques spécifiques du réseau physique réalisant la primitive d'écriture distante. Nous décrivons notre propre implémentation de MPICH au-dessus de notre API RDMA. Enfin, nous analysons les performances de cette première implémentation sur la machine MPC du LIP6 qui constitue notre plate-forme expérimentale.

IV.2. Notre choix d'implémentation de MPI : MPICH

Une application parallèle MPI est composée d'un ensemble de tâches pouvant communiquer entre elles par l'utilisation de primitives de communication. Le standard définit la syntaxe et la sémantique de ces primitives. La première version du standard (MPI-1) est décrite dans [MPI,1994]. En 1997, la norme MPI-2 [MPI,1997] proposant une extension de MPI-1 est apparue. Les principales modifications apportées par MPI-2 concernent la création dynamique de tâches.

IV.2.1. Pourquoi le choix de MPICH ?

MPICH est sans doute l'implémentation la plus utilisée de MPI ce qui lui confère une certaine pérennité. Il a été spécialement conçu pour allier performance et portabilité : le « CH » de MPICH est issu de « Chameleon », symbole d'adaptabilité à plusieurs environnements. Cette portabilité est atteinte en particulier, grâce au découpage en différentes couches. MPICH est très utilisé pour des portages spécifiques sur des architectures variées. On peut citer par exemple : HP/Convex, SGI, Compaq (sur Alphaserver SC), NEC, ParaStation, T3E, Myrinet (MPICH-GM, MPI-BIP, MPICH-FM et MPICH-PM), SCI-MPICH pour les clusters utilisant un réseau SCI.

Nous avons donc choisi de partir d'une implémentation de MPICH, cela nous permettant en particulier de pouvoir nous inspirer du portage de MPI sur le réseau Myrinet qui utilise une primitive d'écriture en mémoire distante, réalisé par Loic Prylli [Prylli,1999].

Les travaux que nous présentons dans ce manuscrit sont basés sur une implémentation du standard MPI-1 mais les solutions que nous présentons pourraient s'appliquer au standard MPI-2. Nous avons choisi de partir de l'implémentation MPICH [Gropp,1996] du standard MPI-1.

IV.2.2. Les différentes couches de MPICH

La Figure IV-1 présente l'architecture de MPICH et ses différentes couches. MPICH fournit aux applications l'API définie par le standard MPI. Les deux principales interfaces internes définies par MPICH sont l'*ADI* (*Abstract Device Interface*) [Gropp,1994] et le *channel interface* [Gropp,1995].

L 'ADI est une interface permettant d'émettre et recevoir des données contiguës en mémoire virtuelle. Elle correspond à un jeu de primitives de communication point à point, bloquantes ou non bloquantes, pour chacun des modes de communication MPI (*standard*, *bufferisé*, *synchrone*, *ready*). Les couches supérieures à l'ADI sont en charge des opérations collectives, de la gestion des communicateurs, et des types complexes. Les couches en-dessous de l'ADI sont en charge de l'émission et de la réception de messages, des copies éventuelles des données entre le processus utilisateur et la carte d'interface réseau, de la gestion des messages en attente, de la gestion du type des données, du découpage éventuel des messages en paquets avec ajout d'un en-tête, d'assurer le cas échéant des communications hétérogènes, de fournir des informations sur l'environnement d'exécution (par exemple, le nombre de tâches MPI composant l'application), etc.

La couche située en-dessous de l'API *channel interface* est uniquement en charge du transfert des données. Elle fait le lien entre l'ADI et le matériel.

MPICH permet donc d'effectuer le portage sur une nouvelle architecture à différents niveaux, cela dépendant du niveau de performance exigé et des services fournis par la couche de communication bas niveau cible.

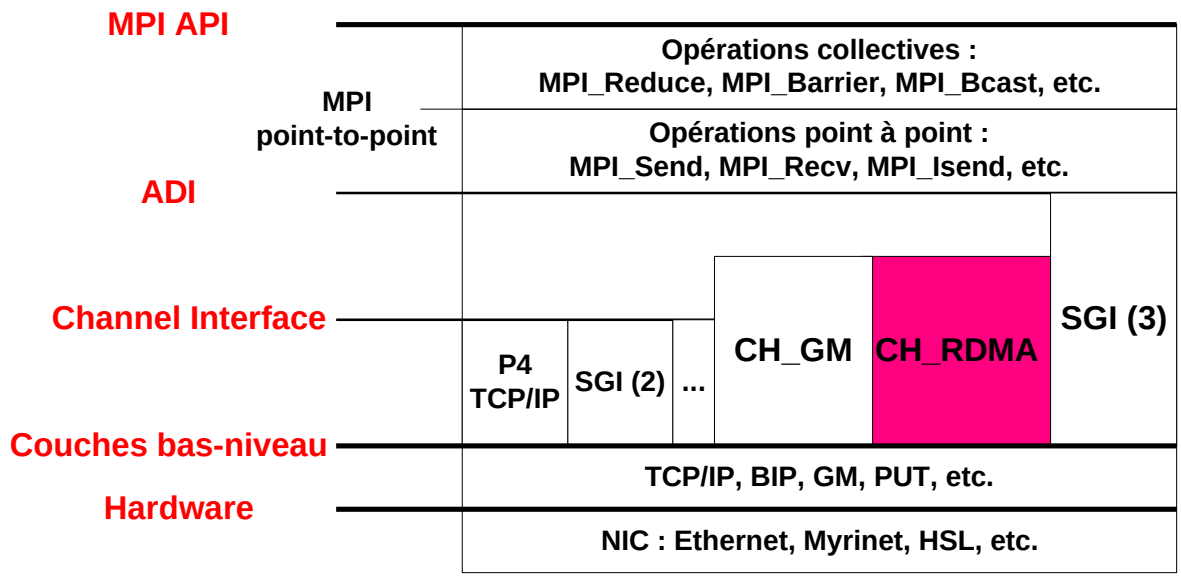


Figure IV-1 : Architecture de MPICH

Par exemple, la deuxième implémentation de MPICH sur les machines SGI (SGI 2) a consisté en la réécriture complète de la couche *channel interface* alors que la troisième implémentation plus performante (SGI 3) a nécessité la réécriture complète de la couche ADI. Dans notre cas, nous avons pris comme point de départ le portage de MPICH sur GM [GM,2000] fourni par la société Myricom qui commercialise les cartes réseau Myrinet. Celui-ci a nécessité la réécriture d'une partie de l 'ADI et de toute la *channel interface*. La couche CH_GM est représentée sur la Figure IV-1. Nous avons décidé de partir de cette implémentation car certaines problématiques étaient communes aux réseaux Myrinet et HSL : ils utilisent tous deux une primitive d'écriture en mémoire distante.

IV.2.3. Messages et protocoles de MPICH

Il existe deux types de message dans MPICH : **les messages de contrôle** (appelés messages ***CTRL***) et **les messages de données** (appelés messages ***DATA***),

- ✓ les messages *CTRL* sont utilisés pour transférer soit des informations de contrôle MPI, soit des données utilisateur de taille limitée. Les messages *CTRL* sont transférés grâce à l'utilisation de tampons intermédiaires alloués par MPI au démarrage de l'application. Ces messages ont donc une taille limitée.

- ✓ les messages *DATA* contiennent uniquement des données utilisateur. Ils sont envoyés après l'envoi d'un ou plusieurs messages *CTRL* suivant le protocole utilisé. Contrairement aux messages *CTRL*, les messages *DATA* n'ont à priori pas de taille maximale.

Les informations identifiant une requête d'émission ou de réception sont le *communicator*, le *tag* MPI et le numéro de la tâche source ou destinataire. Comme expliqué dans la section III.4.1, la correspondance entre requêtes d'émission et requêtes de réception se fait suivant quatre règles :

- ✓ les opérations collectives et point à point sont indépendantes même si elles opèrent dans le même *communicator*,
- ✓ une réception correspond à une émission si elle s'effectue dans le même *communicator*,
- ✓ le récepteur précise la source du message qu'il désire recevoir. Il peut décider de recevoir un message depuis un émetteur quelconque au sein de son *communicator*,
- ✓ les requêtes sont marquées avec un *tag* MPI. Une communication se fait entre les requêtes d'émission et de réception portant sur le même *tag*.

Suivant la taille des données à envoyer, trois protocoles sont disponibles dans MPICH :

- ✓ ***short*** : les données sont incluses dans un message *CTRL*,

- ✓ ***eager*** : les données sont envoyées dans un message *DATA* immédiatement à la suite d'un message *CTRL* véhiculant les informations identifiant la requête d'émission et doivent donc être stockées dans un tampon intermédiaire,
- ✓ ***rendez-vous*** : les données ne sont pas envoyées tant que le récepteur ne les demande pas explicitement. L'émetteur envoie une requête de transmission à l'aide d'un message *CTRL*. Le récepteur renvoie une réponse indiquant à l'émetteur qu'il peut envoyer les données par un autre message *CTRL*. L'émetteur peut alors transmettre les données dans un message *DATA*.

La plupart des ports de MPICH utilise les protocoles *short* et *rendez-vous*. C'est le cas en particulier, des portages de MPICH sur le réseau Myrinet.

IV.3. Les problèmes à résoudre

Nous analysons dans cette section les problèmes liés à l'utilisation d'une primitive d'écriture distante dans le cadre de MPICH.

IV.3.1. Les problèmes liés à la primitive d'écriture distante

Les caractéristiques de l'écriture distante présentées à la section II.2 soulèvent deux principaux problèmes.

Le premier problème est lié au dépôt direct : les données sont déposées en mémoire par le contrôleur réseau du nœud récepteur sans aucune intervention du processeur récepteur. Pour réaliser un transfert, le nœud émetteur doit donc connaître les adresses des tampons où déposer les données dans la mémoire physique du nœud récepteur, de façon à mettre cette information dans les messages envoyés.

Le deuxième problème concerne l'utilisation d'adresses physiques : nous rappelons que les principaux paramètres d'une écriture distante sont :

- ✓ l'adresse **physique** locale des données à émettre,
- ✓ la taille des données,
- ✓ l'adresse **physique** distante où les données doivent être écrites sur le nœud récepteur.

Lors d'un transfert, le tampon d'émission et le tampon de réception associé doivent être contigus en mémoire physique. Les tampons d'émission et de réception au niveau MPI se trouvent en mémoire virtuelle. Il faut donc s'assurer que les données sont verrouillées en mémoire physique, réaliser la traduction d'adresse virtuelle/physique aussi bien sur l'émetteur que sur le récepteur puis faire la correspondance entre les zones contiguës en mémoire physique sur l'émetteur et sur le récepteur comme cela est illustré sur la Figure II-8. L'envoi d'un message MPI nécessitera donc généralement plusieurs appels à la primitive d'écriture distante.

IV.3.2. Les services à fournir à MPICH

Afin de réaliser une implémentation de MPICH au-dessus d'une primitive d'écriture en mémoire distante, nous avons un certain nombre de services à réaliser :

- ✓ émission et réception des messages *CTRL*, à l'aide du protocole « *short* », en utilisant des tampons intermédiaires alloués lors du démarrage de l'application,
- ✓ émission des messages *DATA* à l'aide d'un protocole de rendez-vous avec éventuellement la gestion du découpage des données en plusieurs messages *DATA*, si cela est nécessaire, à cause du problème posé par la discontinuité des données en mémoire physique,
- ✓ mécanisme de contrôle de flux pour les messages *CTRL* : le nombre de tampons intermédiaires est fixe et il y a donc un risque de débordement,
- ✓ signalisation des événements réseau (fin d'émission, fin de réception d'un message MPI) aux couches MPI,
- ✓ attribution lors du démarrage de l'application des numéros de tâches MPI (cette allocation est statique dans le standard MPI-1 qui ne permet pas la création dynamique de nouvelles tâches).

Pour réaliser ces services, nous devrons résoudre les problèmes suivants :

- ✓ passer d'une sémantique à passage de messages (au niveau de MPICH) à la sémantique de l'écriture distante qui ne dispose pas de primitive de réception,
- ✓ transformer une requête d'émission ou de réception MPI correspondant à un tampon en mémoire virtuelle en plusieurs écritures distantes se faisant en adresse physique,
- ✓ fixer le seuil optimal entre les messages *CTRL* et les messages *DATA*.

IV.4. Définition d'une API générique d'écriture en mémoire distante : l'API RDMA

Nous souhaitons formaliser les services fournis par la primitive d'écriture distante en définissant une API générique appelée RDMA (*Remote DMA*). Cette API RDMA fournit une abstraction pour n'importe quel réseau disposant d'une primitive d'écriture en mémoire distante. L'objectif est double :

- ✓ définir la brique de base sur laquelle notre implémentation de MPI s'appuie,
- ✓ masquer les particularités des réseaux fournissant une primitive d'écriture en mémoire distante pour rendre cette implémentation la plus générique possible.

Les paramètres d'un transfert DMA (*Direct Memory Access*) local sont les suivants :

- ✓ `plad` : adresse physique à laquelle les données sont lues ou écrites,
- ✓ `len` : taille des données,

- ✓ `sid` : identificateur de la requête de DMA,
- ✓ `ns` : booléen indiquant si la terminaison du transfert doit être signalée.

L'identificateur de la requête (`sid`) permet de savoir quel transfert DMA est signalé lorsqu'un accès se termine.

Un « *Remote DMA* » fait intervenir un nœud source (émetteur) et un nœud destinataire (récepteur). Il se déroule en trois phases : lecture des données dans la mémoire physique de l'émetteur par un accès DMA, transmission des données sur le réseau, et écriture des données dans la mémoire physique du récepteur par un accès DMA. Un DMA distant fait intervenir les paramètres suivants :

- ✓ `plad` : adresse physique locale à laquelle les données sont lues sur le nœud émetteur,
- ✓ `len` : taille des données,
- ✓ `sid` : identificateur de la requête de DMA sur l'émetteur,
- ✓ `ns` : booléen indiquant si la terminaison du DMA sur le nœud émetteur doit être signalée,
- ✓ `nsrc` : numéro du nœud source,
- ✓ `ndst` : numéro du nœud destinataire,
- ✓ `prad` : adresse physique distante à laquelle les données sont écrites sur le nœud récepteur,
- ✓ `rid` : identificateur de la requête de DMA sur le récepteur,
- ✓ `nr` : booléen indiquant si la terminaison du DMA sur le nœud récepteur doit être signalée.

Les quatre premiers paramètres sont ceux d'un DMA local. Les paramètres `len`, `nsrc`, `ndst`, `prad`, `rid` et `nr` doivent être transmis au réseau et au nœud récepteur pour diverses raisons : effectuer le routage des données à travers le réseau (`ndst`), fournir les informations nécessaires au contrôleur réseau récepteur pour qu'il réalise son DMA en écriture (`len`, `prad`, `rid`, `nr`), fournir les informations nécessaires à la signalisation sur le nœud récepteur (`nsrc`, `rid`).

Nous sommes maintenant en mesure de définir les primitives constituant l'API RDMA : une primitive qui réalise l'écriture en mémoire distante, une primitive pour la notification de la terminaison du transfert DMA sur le nœud émetteur et une primitive pour la notification sur le nœud récepteur.

En ce qui concerne la signalisation, deux politiques différentes sont possibles suivant les caractéristiques intrinsèques du réseau physique utilisé. Ou bien la signalisation s'effectue par une interruption matérielle du contrôleur réseau, ou bien elle repose sur une scrutation des ressources réseau. Cette scrutation pourrait par exemple être faite sur la Liste des Messages à Emettre (LME) et la Liste des Messages Reçus (LMR) définies à la section II.2, à condition que ces listes existent et que leur scrutation soit

possible sur le réseau physique cible. Dans le cas de la deuxième hypothèse, une quatrième primitive est utilisée pour réaliser la scrutation.

L'API RDMA n'impose aucune contrainte au réseau physique concernant les mécanismes de signalisation : elle peut être réalisée aussi bien par interruption matérielle que par scrutation.

Le Tableau IV-1 récapitule les primitives de notre API RDMA ainsi que leurs paramètres. Nous avons séparé ces primitives en trois catégories :

- ✓ la primitive réalisant l'écriture distante : à partir des paramètres fournis, elle traduit l'écriture distante générique que nous avons définie en une écriture distante sur le réseau physique utilisé,
- ✓ les fonctions de notification ou de rappel : elles sont appelées lorsqu'un événement réseau s'est produit et font remonter les informations relatives à cet événement ; les couches utilisatrices de l'API RDMA (MPI dans notre cas) décident du traitement qu'elles souhaitent voir réalisé par ces primitives,
- ✓ une primitive de signalisation par scrutation : elle réalise la scrutation des événements réseau, si le réseau physique le permet, et appelle les fonctions de notification quand une fin d'émission ou une fin de réception doit être signalée.

a) La primitive d'écriture distante

❶ `RDMA_SEND(nsrc, ndst, plad, prad, len, ctrl, sid, rid, ns, nr)`

Rôle : écriture en mémoire distante d'un tampon contigu en mémoire physique
`nsrc` : numéro du nœud local sur lequel les données sont lues
`ndst` : numéro du nœud distant sur lequel les données sont écrites
`plad` : adresse physique locale des données à transmettre (*physical local address*)
`prad` : adresse physique distante du tampon de réception (*physical remote address*)
`len` : taille des données à émettre
`ctrl` : booléen indiquant s'il s'agit d'un message *CTRL* ou d'un message *DATA*
`sid` : identificateur de la requête d'émission
`rid` : identificateur de la requête de réception
`ns` : booléen indiquant si la fin d'émission doit être signalée sur l'émetteur (*notify sent*)
`nr` : booléen indiquant si la fin de réception doit être signalée sur le récepteur (*notify recv*)

b) Les fonctions de notification

❷ `RDMA_SENT_NOTIFY(ctrl, sid)`

Rôle : signaler la fin d'émission d'un message (contrôle ou données)
`ctrl` : booléen indiquant s'il s'agit d'un message *CTRL* ou d'un message *DATA*
`sid` : identificateur de la requête d'émission

❸ `RDMA_RECV_NOTIFY(nsrc, ctrl, rid)`

Rôle : signaler la fin de réception d'un message (contrôle ou données)
`nsrc` : numéro du nœud distant duquel proviennent les données
`ctrl` : booléen indiquant s'il s'agit d'un message *CTRL* ou d'un message *DATA*
`rid` : identificateur de la requête de réception

c) La signalisation par scrutation (optionnelle)
❹ `RDMA_NET_LOOKUP(blocking)` Rôle : scrutation des événements réseau et appel des primitives RDMA_SENT_NOTIFY ou RDMA_RECV_NOTIFY le cas échéant `blocking` : booléen indiquant si la scrutation doit être bloquante

Tableau IV-1 : Les primitives de l'API RDMA

Nous utilisons le paramètre `ctrl` pour typer les messages de la couche qui utilise RDMA. Il est utilisé uniquement pour la signalisation et doit être transmis de l'émetteur vers le récepteur. La Figure IV-2 présente le format des messages de l'API RDMA.

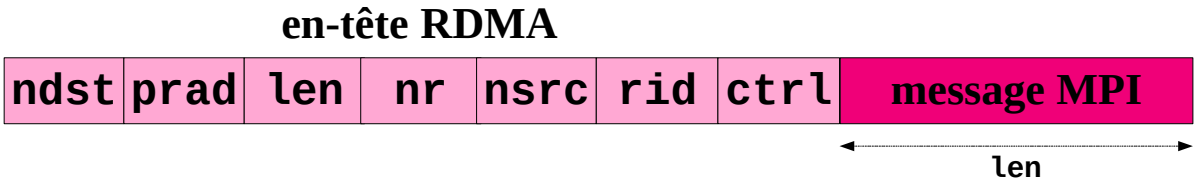


Figure IV-2 : Format des messages de l'API RDMA

Le paramètre `ndst` doit être dans l'en-tête RDMA pour que le message soit acheminé vers le contrôleur réseau destinataire. `prad` indique au contrôleur réseau récepteur où déposer les données dans la mémoire physique distante et `len` lui précise la taille des données. `nr` permet au contrôleur réseau récepteur de savoir si la terminaison du DMA doit être signalée : il se peut que la couche qui utilise l'API RDMA ait besoin de faire plusieurs écritures distantes pour transférer un message ; dans ce cas, seule la fin de réception du dernier message RDMA doit être signalée sur le nœud récepteur. La décision de signalisation doit être prise par l'émetteur, pour chaque message transmis. `nsrc`, `rid` et `ctrl` sont les informations que le récepteur doit transmettre à la couche RDMA pour la notification d'une fin de réception.

Dans le cas où la signalisation est réalisée par interruption, le gestionnaire d'interruptions (cf. Figure II-7) doit alors appeler la fonction de rappel `RDMA_SENT_NOTIFY` s'il s'agit d'une fin d'émission, ou `RDMA_RECV_NOTIFY` s'il s'agit d'une fin de réception.

Dans le cas où la signalisation est réalisée par scrutation, c'est la primitive `RDMA_NET_LOOKUP` qui provoque l'appel de l'une ou l'autre des deux fonctions de rappel si une fin d'émission/réception est détectée.

La contrainte imposée par l'API RDMA au réseau physique est que les informations `ctrl`, `rid` et `nsrc` doivent être véhiculées de l'émetteur vers le récepteur par la primitive d'écriture distante, pour les besoins de la signalisation en réception.

IV.5. Notre implémentation : MPICH au-dessus de RDMA

Nous décrivons dans cette section comment nous avons fait le lien entre les primitives de communication MPI point à point et les primitives RDMA. Il s'agit en particulier de voir comment est réalisée la transmission des messages *CTRL*, la transmission des messages *DATA* et la notification d'une fin d'émission ou d'une fin de réception d'un message MPI.

IV.5.1. Points de départ

L'appel de la primitive `RDMA_SEND` se fait à travers le système d'exploitation qui dans notre cas est un système UNIX standard tel que LINUX ou FreeBSD. Nous avons donc écrit un module noyau à chargement dynamique pour MPI (MPIDRIVER) sur chacun des deux systèmes d'exploitation cible. Celui-ci fournit des points d'entrée vers le noyau pour les opérations suivantes :

- ✓ récupération du numéro de nœud de la machine locale,
- ✓ enregistrement de MPI comme utilisateur de la primitive d'écriture distante,
- ✓ réalisation du verrouillage en mémoire physique et de la traduction d'un tampon contigu en mémoire virtuelle,
- ✓ appel de la primitive d'écriture distante : cela correspond à l'ajout d'une (ou de plusieurs) entrée dans la Liste des Messages à Emettre (LME, cf. section II.2).

La Figure IV-3 représente l'architecture de notre première implémentation de MPI sur une primitive d'écriture distante : MPI-MPC1.

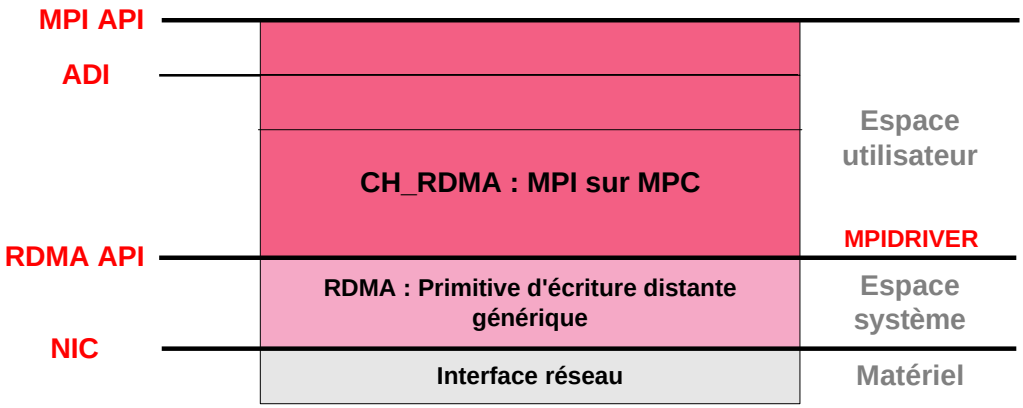


Figure IV-3 : Architecture de MPI-MPC1

Dans cette première implémentation, l'API RDMA se trouve dans l'espace système et la signalisation des événements réseau se fait par interruption matérielle. Lors de l'enregistrement de MPI comme utilisateur de la primitive d'écriture distante, les deux fonctions de rappel se trouvant dans l'espace système (dans la couche RDMA) sont enregistrées (une pour les fins d'émission et une pour les fins de réception).

Nous disposons enfin d'un allocateur de mémoire physique contiguë appelé CMEM et réalisé par Alexandre Fenÿo [Fenÿo,2001]. Son fonctionnement est simple : il s'alloue de façon statique au démarrage de la machine et verrouille une zone de mémoire physique contiguë et en redistribue par la suite des morceaux aux applications. De telles zones de mémoire peuvent être projetées dans la mémoire virtuelle du processus en en faisant la demande à l'aide de la fonction `mmap` de CMEM. Nous utiliserons cet allocateur en particulier pour les tampons intermédiaires nécessaires aux messages *CTRL*.

IV.5.2. Transmission des messages de contrôle

Les messages *CTRL* permettent de transférer, soit des informations de contrôle, soit des données de l'application d'une taille limitée. Ils sont envoyés directement de l'émetteur vers le récepteur. Ils sont transmis à l'aide d'une recopie systématique en émission comme en réception dans des tampons intermédiaires alloués lors du démarrage de l'application.

Dans notre cas, ces tampons intermédiaires sont demandés à l'allocateur de mémoire physique contiguë CMEM (cf. section précédente). Lors de l'initialisation, chaque tâche MPI constituant l'application s'attribue une zone de mémoire physique contiguë. Celle-ci est projetée dans la mémoire virtuelle de la tâche MPI. Elle est découpée en NNODES files circulaires où NNODES représente le nombre de nœuds de la machine. Ensuite, chaque file est elle-même divisée en NBUF tampons d'une taille de L octets où L est la taille maximale d'un message *CTRL*.

La Figure IV-4 illustre ce découpage pour une machine constituée de 3 nœuds. Seuls les tampons intermédiaires des nœuds 1 et 3 sont représentés. Chaque file circulaire est constituée de quatre tampons dans l'exemple ci-dessous.

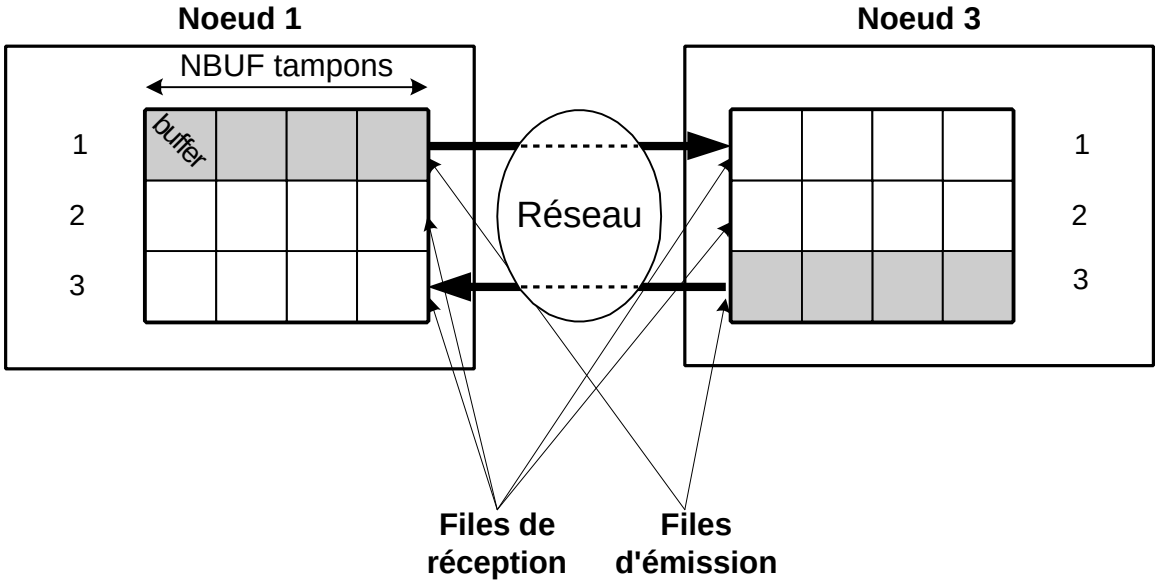


Figure IV-4 : Les tampons intermédiaires pour les messages de contrôle

Sur chacun des nœuds, il y a une file d'émission et NNODES-1 files de réception (2 dans le cas de la figure). Les files d'émission sont grisées sur la Figure IV-4 : il s'agit de la file 1 sur le nœud 1 et de la file 3 sur le nœud 3. Lorsque le nœud 1 veut envoyer un message *CTRL* au nœud 3, il y a une recopie en émission dans un des tampons disponibles de la file 1. Le message est alors écrit dans la file 1 du nœud 3 récepteur. Si des données utilisateur sont encapsulées dans le message *CTRL*, celles-ci seront recopiées dans le tampon utilisateur destinataire lorsque la réception correspondante sera postée.

Ce découpage de la zone de mémoire physique contiguë nous permet de répondre au premier problème posé à la section IV.3.1 (où écrire sur le récepteur ?). Lors du démarrage de l'application, chaque tâche MPI communique à toutes les autres tâches constituant l'application, via le réseau de contrôle Ethernet, l'adresse physique du début de sa propre zone de mémoire contiguë. Les files de réception étant associées à chacune des tâches distantes potentiellement émettrice, il y a un unique émetteur qui peut écrire dans une file réceptrice. Par exemple, seul le nœud 1 peut écrire dans les quatre tampons de la file 1 du nœud 3. Il suffit alors de comptabiliser sur chaque tâche les émissions en cours dans toutes les files de réception distantes : le nœud 1 sait à chaque instant combien de messages il a écrit dans la file 1 du nœud 2 et dans la file 1 du nœud 3. Il sait donc où écrire le message suivant dans chacune des files de réception distantes qui lui correspondent. Comme nous supposons le réseau FIFO, les messages arrivent sur le récepteur dans l'ordre où ils ont été émis.

Concernant le deuxième problème soulevé dans la section IV.3.1, les tampons intermédiaires étant contigus en mémoire physique, il n'a pas de traduction d'adresse à faire avant l'appel de la primitive d'écriture distante.

En émission, les tampons intermédiaires sont libérés lorsque le contrôleur réseau notifie la fin d'émission du message *CTRL*. Le mécanisme de signalisation des messages *CTRL* est détaillé dans la section IV.5.8. Comme il peut y avoir une saturation des tampons de réception, il est nécessaire d'assurer un contrôle de flux entre chaque couple émetteur/récepteur.

Nous utilisons un contrôle de flux à crédits qui est déjà présent dans le portage de MPICH sur GM. Le principe est le suivant : chaque émetteur maintient le compte des messages qu'il peut envoyer à chaque récepteur potentiel. Lors de chaque nouvel envoi, il le décrémente. Quand il est nul, il arrête d'émettre vers ce récepteur en attendant que celui-ci lui redonne des crédits. Chaque récepteur garde pour chaque émetteur potentiel le compte des messages qu'il a reçus. Chaque fois que les données sont consommées en réception, il l'incrémente. Pour éviter de faire fonctionner l'émetteur par à-coups et pour ne pas générer de trafic réseau supplémentaire, nous utilisons tous les messages *CTRL* en transit pour rendre des crédits à un émetteur (*piggybacking*). Nous insérons pour cela dans chaque message *CTRL*, le nombre de crédits à rendre au destinataire du message. Le format des messages *CTRL* est détaillé dans la section IV.5.4. Un récepteur ne devra envoyer un message *CTRL* spécifique de type *CREDIT* à un émetteur qu'en cas de famine : par exemple, dans le cas de la

Figure IV-4, si le nœud 1 envoie successivement quatre messages au nœud 3 sans recevoir de message de celui-ci entre temps, alors le nœud 3 devra envoyer au nœud 1 un message de type *CREDIT* pour lui rendre quatre crédits.

IV.5.3. Transmission des messages de données

La transmission des messages *DATA* utilise un protocole de rendez-vous entre l'émetteur et le récepteur comme l'illustre la Figure IV-5. L'émetteur envoie une requête (message *CTRL* de type *REQ*) au récepteur contenant la taille des données à émettre et attend la réponse de celui-ci (message *CTRL* de type *RSP*) avant d'envoyer les données sans aucune encapsulation (message *DATA*). Contrairement aux messages *CTRL*, les messages *DATA* ne comportent pas d'en-tête. L'avantage de ce protocole est de transmettre les données directement depuis le tampon utilisateur source vers le tampon utilisateur destinataire. Ce protocole est donc zéro-copie. Cependant, cela suppose que la requête de réception ait été postée avant l'envoi du message *RSP*. Dans le contraire, pour ne pas bloquer l'émetteur, le récepteur alloue un tampon intermédiaire avant d'envoyer le message *RSP*. Une copie sera alors nécessaire depuis ce tampon intermédiaire vers le tampon de l'application lorsque la requête de réception sera postée.

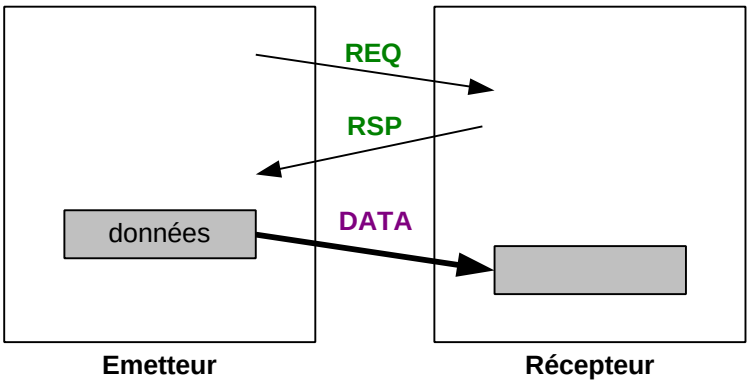


Figure IV-5 : Le protocole de rendez-vous

Il est nécessaire de verrouiller en mémoire physique le tampon de l'application aussi bien sur l'émetteur que sur le récepteur. Il faut également faire les traductions d'adresses pour le tampon d'émission et le tampon de réception afin d'obtenir la projection (*mapping*) de chacun des deux tampons en mémoire physique. Ces deux opérations sont réalisées, d'une part sur le récepteur avant d'envoyer le message *RSP*, et d'autre part sur l'émetteur avant d'envoyer les données (message *DATA*). Pour que l'émetteur sache où écrire les données dans la mémoire physique du récepteur, la correspondance en mémoire physique du tampon de réception lui est envoyée dans le message *RSP*. C'est ce qui nous permet de faire une transmission de type zéro-copie. Le message *RSP* contient donc une liste de descripteurs de tampons de réception. Un descripteur décrit une zone contiguë en mémoire physique : il contient l'adresse physique du début de la zone ainsi que sa taille. Avant d'envoyer le message *DATA*, l'émetteur devra faire la correspondance entre les zones contiguës en mémoire

physique du tampon d'émission et celles du tampon de réception qu'il aura reçues via le message *RSP*. Il pourra alors faire le nombre d'appels nécessaires à la primitive d'écriture distante pour transmettre les données. Dans le cas de la Figure II-8, le message *RSP* contient deux descripteurs de DMA (le tampon de réception correspond à deux zones contiguës en mémoire physique) et l'émetteur doit faire trois appels à la primitive d'écriture distante pour transmettre les données.

Ce protocole permet l'envoi de très longs messages *DATA* en mode zéro-copie mais il suppose que le message *RSP* puisse contenir la description en mémoire physique du tampon de réception : nous rappelons que la taille des messages *CTRL* est limitée. Les constantes suivantes sont définies dans notre implémentation :

- ✓ *NB_DMA_MAX* : nombre maximum de descripteurs DMA pouvant être transmis dans un message de type *RSP*,
- ✓ *MAX_CTL_MSG_LEN* : taille maximale d'un message *CTRL*,
- ✓ *GMPI_MAX_DATA_FRAG* : taille maximale d'un message de données.

Sachant que la taille d'un descripteur de DMA est de 8 octets et qu'un message *RSP* contient un en-tête de 20 octets (cf. section IV.5.4), nous pouvons facilement en déduire la formule suivante :

$$NB\_DMA\_MAX = (MAX\_CTL\_MSG\_LEN - 20)/8 \qquad \textbf{(1)}$$

Les messages *DATA* ont donc une taille maximale. Cela nous oblige à étendre le protocole de rendez-vous présenté dans la Figure IV-5. La Figure IV-6 illustre ce protocole étendu.

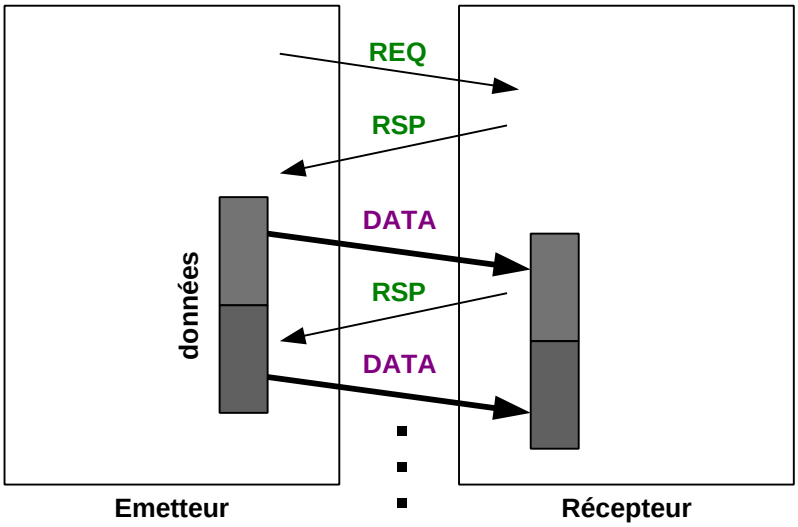


Figure IV-6 : Le protocole de rendez-vous étendu

Il s'agit d'un protocole de rendez-vous en un nombre de phases qui peut être supérieur à trois. Le message de contrôle *REQ* contient toujours la taille des données que l'émetteur veut envoyer mais celles-ci sont fragmentées en plusieurs messages *DATA*, si la taille des données à émettre est supérieure à la taille maximale d'un message *DATA*. Après chaque réception d'un message *DATA*, le récepteur regarde s'il a reçu toutes les données. Si tel n'est pas le cas, il renvoie un message *RSP* décrivant les zones de réception en mémoire physique du tampon de réception du prochain fragment. Ce protocole est toujours un protocole zéro-copie et nous permet de transmettre des données de n'importe quelle taille.

Dans le cas où la requête de réception n'était pas présente avant d'envoyer le premier message *RSP*, l'avantage de cette fragmentation en plusieurs messages *DATA* est, d'une part de limiter la taille du tampon intermédiaire en réception et, d'autre part de laisser une chance à la requête de réception d'arriver pendant la transmission du premier fragment et donc d'éviter des recopies intermédiaires inutiles pour les fragments suivants.

L'Annexe A explique comment les traductions d'adresse virtuelle/physique sont réalisées.

IV.5.4. Les différents types de message de contrôle

Nous pouvons désormais décrire les différents sous-types de message *CTRL* que nous utilisons. Il en existe quatre :

- ✓ Le sous-type *SHORT* : il s'agit d'un message *CTRL* dans lequel des données de l'application sont encapsulées pour être transmises directement (sans utiliser le protocole de rendez-vous) mais avec une recopie systématique des données en émission et en réception.
- ✓ Le sous-type *REQ* : c'est le message *CTRL* qui permet d'initier un protocole de rendez-vous. Il est envoyé par l'émetteur.
- ✓ Le sous-type *RSP* : c'est la réponse à un message de type *REQ* qui contient la description des tampons de réception en mémoire physique. Il est envoyé par le récepteur.
- ✓ Le sous-type *CREDIT* : c'est un message *CTRL* spécifique pour le contrôle de flux à crédits décrits à la fin de la section IV.5.2.

La Figure IV-7 représente le format de chacun des quatre types de messages *CTRL*. Chacun de ces messages MPI doit ensuite être encapsulé dans un message RDMA.

Le champ ***crdt*** se trouve dans l'en-tête de chaque message. Il permet de renvoyer des crédits au nœud destinataire sans avoir à utiliser un message spécifique de type *CREDIT*. Le champ ***type*** contient le type du message *CTRL*. Le champ ***tsrc*** est le numéro de la tâche MPI qui a envoyé le message *CTRL*. Le champ ***tag*** est le *tag* MPI choisi par l'application pour la communication en cours. Le champ ***dtlen*** est la taille

des données de l'application qu'il faut transmettre. Dans un message de type *SHORT*, le champ `dtlen` indique au récepteur la taille des données encapsulées dans le message. Dans un message de type *REQ*, cela indique au récepteur la taille du message *DATA* qui sera envoyé ensuite et lui permet éventuellement d'allouer un tampon intermédiaire si la requête de réception correspondante n'a pas été postée. Le champ ***recv_map*** est la description en mémoire physique du tampon de réception des données de l'application. ***NB_DMA*** est le nombre de descripteurs de DMA nécessaires pour décrire cette zone. Le champ ***sid*** (***send_id***) est un numéro qui permet d'identifier la requête d'émission qui est en cours de traitement. Il est envoyé dans le message *REQ* afin d'être encapsulé dans la réponse du récepteur pour que l'émetteur sache, quand il reçoit un message de type *RSP*, de quelle requête d'émission il s'agit. Enfin, le champ ***rid*** (***recv_id***) est l'identificateur de la requête de réception. Il est utilisé pour la signalisation des messages *DATA* sur le récepteur. La section IV.5.6 explique comment les identificateurs de requêtes (***sid*** et ***rid***) sont utilisés.

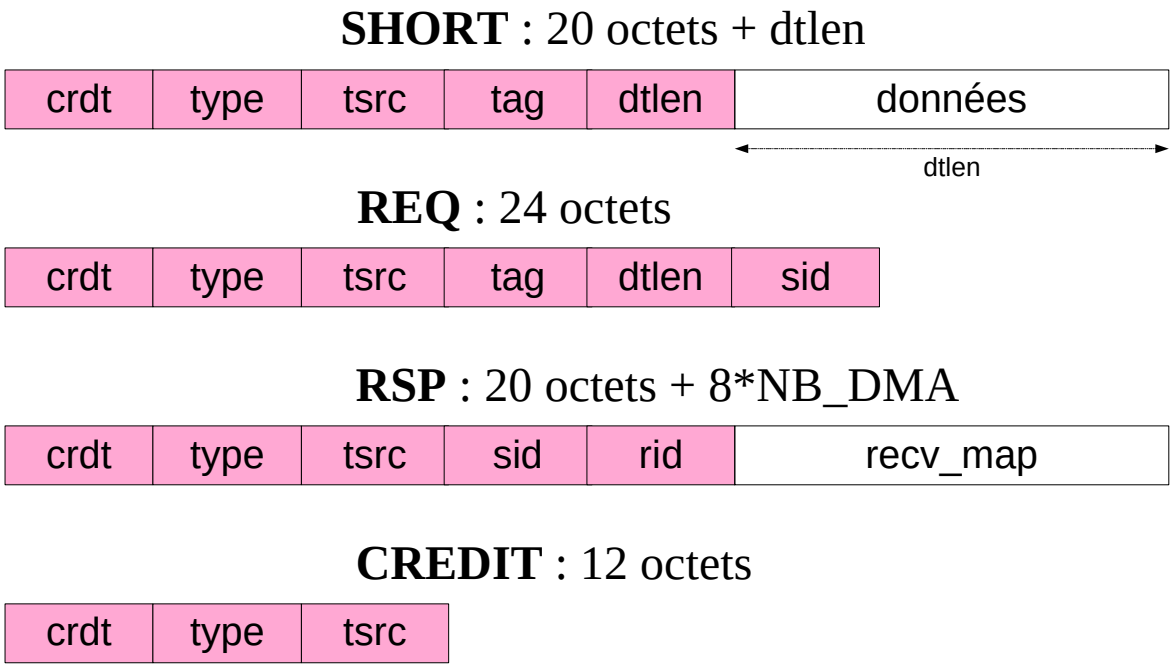


Figure IV-7 : Le format des messages de contrôle

IV.5.5. Les primitives de communication MPI

Nous décrivons dans cette section comment nous faisons le lien entre les primitives de communication point à point définies par le standard MPI et les différents types de messages décrits ci-dessus. Le standard MPI définit un grand nombre de primitives pour les communications point à point qui peuvent être bloquantes ou non bloquantes (pour permettre du recouvrement calcul/communication). Au delà du caractère bloquant ou non bloquant des primitives, il existe quatre modes de communication :

- ✓ *standard* : le mode asynchrone de communication ; la terminaison d'une requête d'émission ne dépend pas nécessairement de la présence d'une requête de réception sur le récepteur,
- ✓ *synchrone* : l'envoi ne se termine que lorsque la requête de réception correspondante a été postée (il y a une synchronisation entre l'émetteur et le récepteur),
- ✓ *ready* : une émission en mode *ready* ne doit être faite que si l'application est certaine que la réception correspondante a été postée,
- ✓ *bufferisé* : l'émetteur demande explicitement une sauvegarde intermédiaire des données dans un tampon réservé aux émissions *bufferisées* ; l'envoi se termine dès que les données ont été recopiées dans ce tampon.

C'est l'émetteur qui choisit le mode de la communication. Le Tableau IV-2 récapitule les différentes primitives d'émission point à point. En réception, les deux primitives de communication point à point sont `MPI_Recv` (bloquante) et `MPI_Irecv` (non bloquante).

Mode	Bloquant	Non bloquant
standard	`MPI_Send`	`MPI_Isend`
synchrone	`MPI_Ssend`	`MPI_Issend`
ready	`MPI_Rsend`	`MPI_Irsend`
bufferisé	`MPI_Bsend`	`MPI_Ibsend`

Tableau IV-2 : Les primitives d'émission point à point dans MPI

Le « I » dans le nom des primitives de communication MPI signifie *Immediate*. Le principe des primitives non bloquantes est de rendre la main aussitôt (dès que la requête de transmission ou de réception est enregistrée). L'application peut savoir par la suite si la communication est terminée en appelant la primitive `MPI_Wait`. Plus de détails concernant la sémantique de ces primitives sont disponibles dans le standard MPI [MPI,1994]. L'Annexe C présente les paramètres de ces primitives de communication.

IV.5.5.1. Le mode standard

Dans le mode standard, l'émetteur fait un `MPI_Send` ou un `MPI_Isend` et le récepteur un `MPI_Recv` ou un `MPI_Irecv`. La Figure IV-8 montre comment cela se traduit dans notre implémentation. Deux cas peuvent se produire. Si les données de l'application peuvent être encapsulées dans un message *CTRL* de type *SHORT*, la transmission nécessite deux recopies (dans les tampons intermédiaires décrits sur la Figure IV-4) mais elle se fait en utilisant un seul message. Si la taille du message est trop grande, l'émetteur initie un protocole de rendez-vous en au moins trois phases mais aucune recopie des données n'est nécessaire. Si la taille du message est supérieure à la taille maximale d'un message *DATA*, les données sont envoyées en utilisant le protocole de rendez-vous étendu de la Figure IV-6.

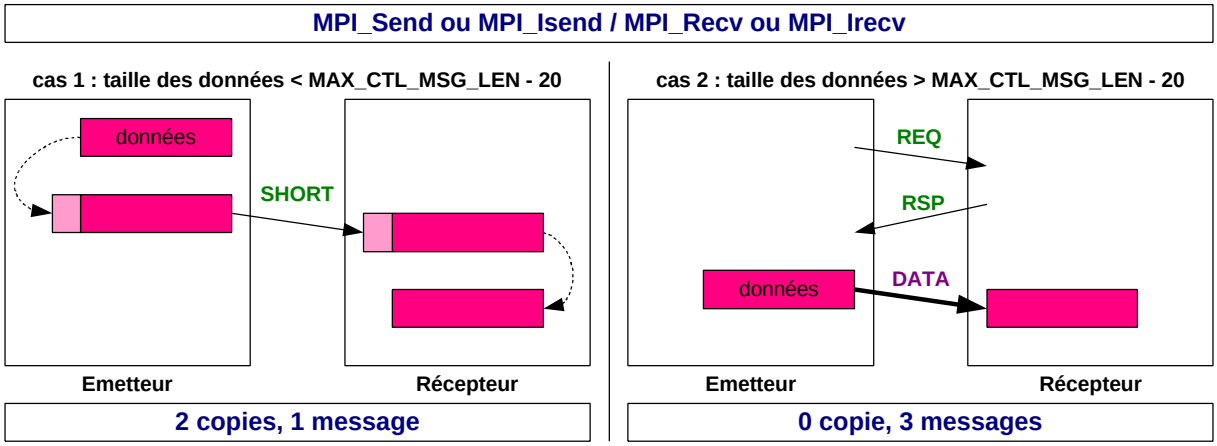


Figure IV-8 : Le mode standard

Les primitives `MPI_Isend` et `MPI_Irecv` rendent la main dès que la requête d'émission ou de réception a été enregistrée.

Dans le cas 1 de la Figure IV-8, `MPI_Send` rend la main quand les données ont été encapsulées dans le message *CTRL* de type *SHORT* et que celui-ci a été recopié dans le tampon intermédiaire d'émission. Dans le cas 2, elle rend la main lorsque la fin d'émission du dernier message *DATA* est signalée. L'application peut alors modifier le contenu de son tampon d'émission.

Dans le cas 1, `MPI_Recv` rend la main quand les données ont été extraites du message *CTRL* et qu'elles ont été recopiées dans le tampon de réception de l'application. Dans le cas 2, elle rend la main lorsque la fin de réception du dernier message *DATA* a été signalée.

IV.5.5.2. Le mode synchrone

Dans le mode synchrone, l'émetteur fait un `MPI_Ssend` ou un `MPI_Issend` et le récepteur un `MPI_Recv` ou un `MPI_Irecv`. La Figure IV-9 montre comment cela se traduit dans notre implémentation. Contrairement au mode standard, quelque soit la taille des données, un protocole de rendez-vous est nécessaire pour réaliser la synchronisation entre l'émetteur et le récepteur. Le premier message *RSP* allant du récepteur vers l'émetteur n'est envoyé que lorsque la requête de réception associée a été postée. De ce fait, dans le mode synchronisé, le caractère zéro-copie du protocole de rendez-vous est garanti : il n'est pas gênant de faire attendre l'émetteur si la réception correspondante n'est pas arrivée puisque cela fait partie intégrante de la sémantique du mode synchronisé.

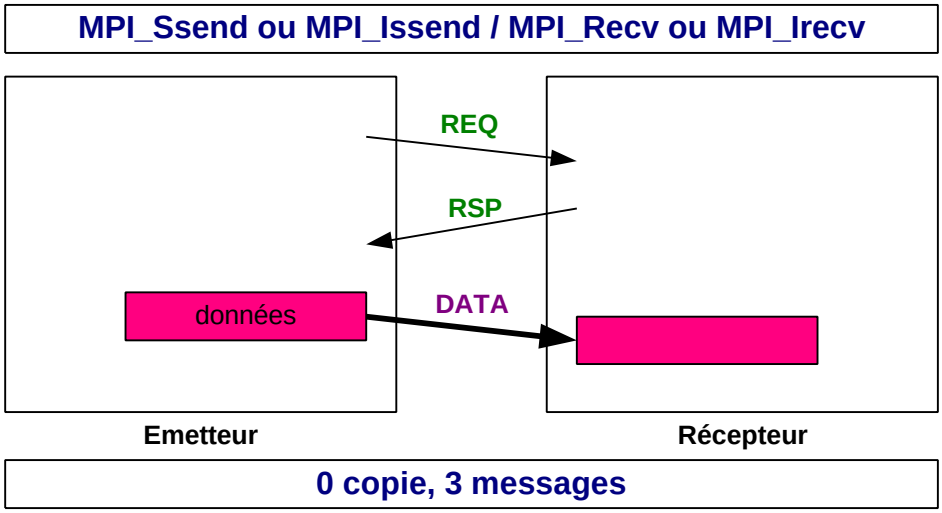


Figure IV-9 : Le mode synchrone

`MPI_Ssend` rend la main lorsque la fin de transmission du dernier message *DATA* est signalée. `MPI_Issend` rend la main lorsque le premier message *RSP* est reçu. `MPI_Recv` rend la main lorsque la fin de réception du dernier message *DATA* est signalée. `MPI_Irecv` rend la main dès que la requête de réception est enregistrée.

IV.5.5.3. Le mode ready

Dans le mode *ready*, l'émetteur fait un `MPI_Rsend` ou un `MPI_Irsend` et le récepteur un `MPI_Recv` ou un `MPI_Irecv`. Dans notre implémentation, le mode *ready* est identique au mode synchrone. Le standard MPI a prévu ce mode uniquement pour permettre à certaines implémentations de mettre à profit le fait de savoir sur l'émetteur que la réception correspondante a été postée. Ce n'est pas notre cas.

IV.5.5.4. Le mode bufferisé

Dans le mode *bufferisé*, l'émetteur fait un `MPI_Bsend` ou un `MPI_Ibsend` et le récepteur un `MPI_Recv` ou un `MPI_Irecv`. La terminaison de l'émission est indépendante de l'état du récepteur : la primitive `MPI_Bsend` rend la main dès que les données ont été recopiées dans un tampon intermédiaire. Ce tampon est unique pour toutes les émissions en mode *bufferisé* : l'application est chargée de s'assurer qu'elle peut réutiliser ce tampon avant de faire une nouvelle émission *bufferisée*. Une fois les données recopiées, celles-ci sont envoyées selon une émission non bloquante utilisant le protocole du mode *standard*.

IV.5.6. Les émissions/réceptions simultanées utilisant le protocole de rendez-vous

Comment réaliser plusieurs émissions/réceptions simultanément lorsque le transfert se fait suivant le protocole de rendez-vous ?

Lorsqu'une émission ou une réception est postée par une tâche de l'application MPI, on crée une structure de données qui conservera l'état de la communication durant toute sa durée de vie. Cette structure s'appelle *SHANDLE* pour une requête d'émission et *RHANDLE* pour une requête de réception. Elle contient par exemple, le *tag* MPI du message, sa taille, l'adresse virtuelle du tampon d'émission/réception, le numéro de la tâche source/destinataire, le nombre d'octets déjà envoyés/reçus, etc. On souhaite avoir plusieurs requêtes d'émission/réception en cours simultanément.

Concernant les messages *CTRL*, il peut y avoir autant d'émissions/réceptions simultanées qu'il y a de tampons intermédiaires : dans le cas de la Figure IV-4, chaque tâche MPI peut faire quatre émissions simultanées d'un message *CTRL* et quatre réceptions simultanées par tâche distante.

Concernant les messages *DATA*, si l'on souhaite pouvoir traiter plusieurs requêtes d'émission ou de réception simultanément, il faut être capable, lors de la signalisation d'une fin de réception ou d'une fin d'émission d'un message *DATA*, de retrouver la requête (*RHANDLE* ou *SHANDLE*) associée à la communication en cours. Nous conservons donc pour chacune des tâches, deux tables associatives : une pour les émissions des messages *DATA* et l'autre pour les réceptions des messages *DATA*. La table des émissions en cours associe un identificateur de la requête d'émission (*sid*) à un *SHANDLE*, et celles des réceptions un *rid* à un *RHANDLE*. Ces tables permettent également de conserver le *sid* ou le *rid* distant : le *rid* sert à l'émetteur pour envoyer les messages *DATA* et le *sid* sert au récepteur pour envoyer les messages *RSP*.

La Figure IV-10 présente un exemple dans lequel trois tâches MPI ont chacune deux communications simultanément (2 émissions pour la tâche x, 2 réceptions pour la tâche y, une émission et une réception pour la tâche z). Dans l'exemple présenté, chacune des tâches peut avoir quatre émissions et quatre réceptions, utilisant le protocole de rendez-vous, simultanément. Le champ SHANDLE ou RHANDLE d'une table contient l'adresse de la structure de données qui conserve toutes les informations relatives à la requête en cours. Le couple (*sid*, *rid*) constitue en quelque sorte un canal virtuel entre deux tâches MPI s'échangeant des données suivant le protocole de rendez-vous.

L'identificateur de réception (rid) est choisi par le récepteur avant d'envoyer le message *RSP*. Celui d'émission (sid) est choisi par l'émetteur avant d'envoyer le message *REQ*.

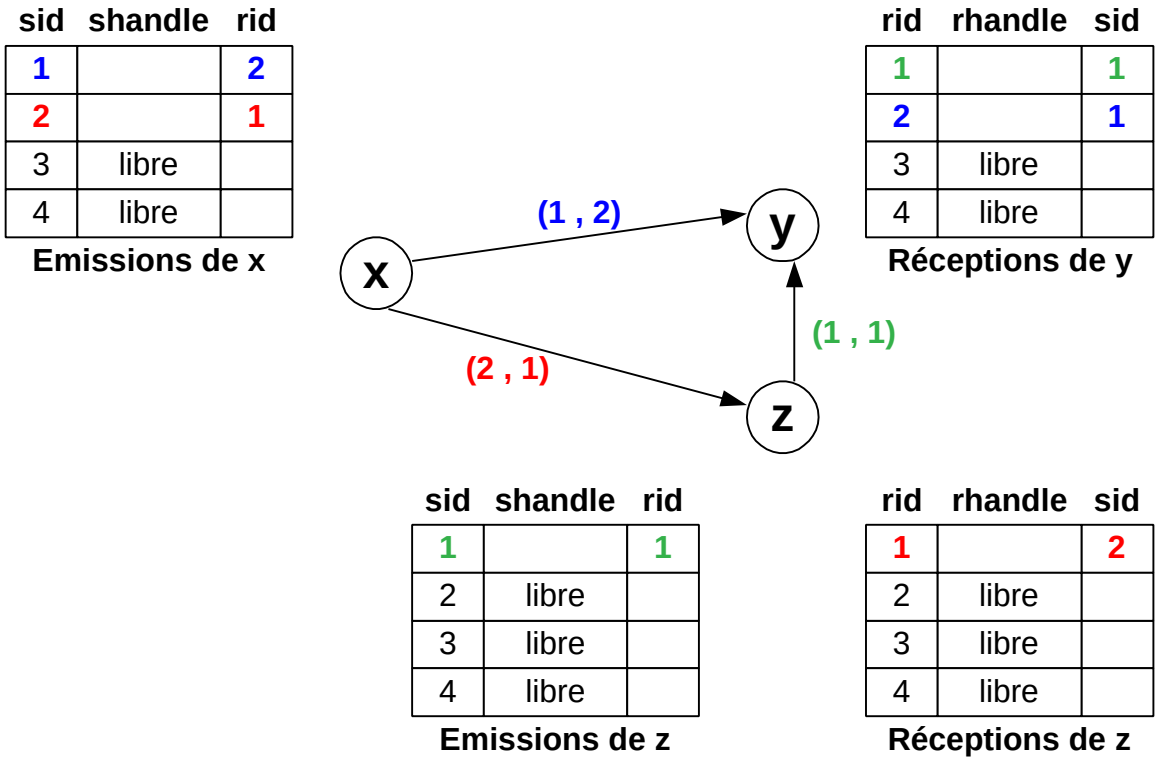


Figure IV-10 : Les identificateurs de requêtes pour les messages *DATA*

IV.5.7. Liens entre notre implémentation de MPI et l'API RDMA

Les fonctions présentées dans le Tableau IV-3 correspondent aux cinq types de messages définies en IV.5.3 et IV.5.4.

Rappelons les paramètres de la fonction `RDMA_SEND` :

❶ `RDMA_SEND(ndst, plad, prad, len, ctrl, sid, rid, ns, nr)`

Rôle : écriture en mémoire distante d'un tampon contigu en mémoire physique

Le paramètre `ctrl` permet de distinguer les messages *CTRL* des messages *DATA* de la couche CH_RDMA.

Les paramètres `sid` et `rid` sont utilisés pour la signalisation d'une fin d'émission ou d'une fin de réception d'un message *DATA*. Par conséquent, ils n'ont aucune signification dans le cas où le paramètre `ctrl` indique qu'il s'agit d'un message *CTRL* (`ctrl=1`).

❶	`CHRDMA_SEND_SHORT(vad, len, tid_dst, tag)` Rôle : encapsulation et émission d'un message *CTRL* de type SHORT `vad` : adresse virtuelle du tampon de l'application où se trouvent les données `len` : taille des données à émettre `tid_dst` : numéro de la tâche MPI destinataire `tag` : tag MPI choisi par l'application pour cette communication
❷	`CHRDMA_SEND_REQ(len, tid_dst, tag, sid)` Rôle : encapsulation et émission d'un message *CTRL* de type REQ `len` : taille des données à émettre `tid_dst` : numéro de la tâche MPI destinataire `tag` : tag MPI choisi par l'application pour cette communication `sid` : identificateur de la requête d'émission
❸	`CHRDMA_SEND_RSP(tid_dst, sid, rid, recv_map)` Rôle : encapsulation et émission d'un message *CTRL* de type RSP `tid_dst` : numéro de la tâche MPI destinataire `sid` : identificateur de la requête d'émission `rid` : identificateur de la requête de réception `recv_map` : description en mémoire physique du tampon de réception des données
❹	`CHRDMA_SEND_CREDIT(tid_dst)` Rôle : encapsulation et émission d'un message *CTRL* de type CREDIT `tid_dst` : numéro de la tâche MPI destinataire
❺	`CHRDMA_SEND_DATA(vad, len, tid_dst, recv_map, sid, rid)` Rôle : traduction en adresses physiques de (`vad`, `len`) et émission d'un message *DATA* `vad` : adresse virtuelle du tampon de l'application où se trouvent les données `len` : taille des données à émettre `tid_dst` : numéro de la tâche MPI destinataire `recv_map` : description en mémoire physique du tampon de réception des données `sid` : identificateur de la requête d'émission `rid` : identificateur de la requête de réception
❻	`CHRDMA_NET_LOOKUP(blocking)` Rôle : scrutation des événements réseau (cf. section IV.5.8) `blocking` : booléen indiquant si la scrutation doit être bloquante

Tableau IV-3 : Les primitives de CH_RDMA liées à l'API RDMA

Le fonctionnement de la primitive `CHRDMA_NET_LOOKUP` sera détaillé dans la section IV.5.8.

Les primitives `CHRDMA_SEND_SHORT`, `CHRDMA_SEND_REQ`, `CHRDMA_SEND_RSP`, `CHRDMA_SEND_CREDIT` permettent l'envoi d'un message *CTRL*, et doivent réaliser les opérations suivantes :

- ✓ construction du message *CTRL* tel qu'il est défini dans la Figure IV-7 et calcul de sa taille (`len`),
- ✓ attribution d'un tampon intermédiaire dans la file d'émission des messages *CTRL*,
- ✓ recopie du message *CTRL* dans le tampon intermédiaire,
- ✓ transformation du paramètre `tid_dst` en un numéro de nœud destinataire (`ndst`): lors de la phase d'initialisation, chacune des tâches MPI

communique à toutes les autres le numéro du nœud sur lequel elle se trouve via le réseau de contrôle,

- ✓ transformation de l'adresse virtuelle du tampon intermédiaire d'émission en une adresse physique (`plad`) : il s'agit d'un simple décalage puisque les tampons intermédiaires sont contigus en mémoire physique et qu'ils sont projetés dans la mémoire virtuelle de la tâche MPI lors de l'initialisation,
- ✓ incrémentation du compteur des messages *CTRL* déjà écrits dans la file de réception de la tâche destinataire et déduction de l'adresse physique distante (`prad`) du tampon intermédiaire de réception du message *CTRL*,
- ✓ appel de la primitive `RDMA_SEND` avec :
  - `len` = taille du message *CTRL*
  - `ctrl` = 1 (pour dire qu'il s'agit d'un message *CTRL*)
  - `sid` = 0 (ce paramètre n'a de sens que pour les messages *DATA*)
  - `rid` = 0 (idem)
  - `ns` = 1 (notification de la fin d'émission du message *CTRL*)
  - `nr` = 1 (il faudra signaler la fin de réception du message *CTRL* à la tâche réceptrice)

La primitive CH_RDMA_SEND_DATA doit réaliser les opérations suivantes :

- ✓ transformation de `tid_dst` en un numéro de nœud destinataire (`ndst`),
- ✓ verrouiller en mémoire physique le tampon d'émission (`vad`, `len`) où se trouvent les données à émettre,
- ✓ faire la traduction d'adresse du tampon d'émission afin d'obtenir sa description en mémoire physique (`send_map`),
- ✓ établir les correspondances entre les zones contiguës en mémoire physique du tampon d'émission et celles du tampon de réception (`recv_map`),
- ✓ faire les appels nécessaires à la primitive `RDMA_SEND` avec ctrl = 0 pour indiquer qu'il s'agit d'un message *DATA*.

La Figure IV-11 illustre un exemple d'utilisation de cette primitive. Dans cette figure, la tâche 0 présente sur le nœud 0 initie un protocole de rendez-vous pour envoyer des données vers la tâche 1 du nœud 1.

La tâche émettrice choisit 2 comme identificateur de requête d'émission alors que la tâche réceptrice choisit l'identificateur 3 de requête de réception. La taille du message *RSP* est de 36 octets (20+2*8) car le tampon de réception correspond à deux zones contiguës en mémoire physique. Les descriptions en mémoire physique des tampons d'émission et de réception nécessitent trois écritures en mémoire distante du nœud 0 vers le nœud 1 pour transmettre le message *DATA*. Seule la dernière écriture distante doit générer la signalisation de fin d'émission sur le nœud émetteur et de fin de réception sur le nœud récepteur.

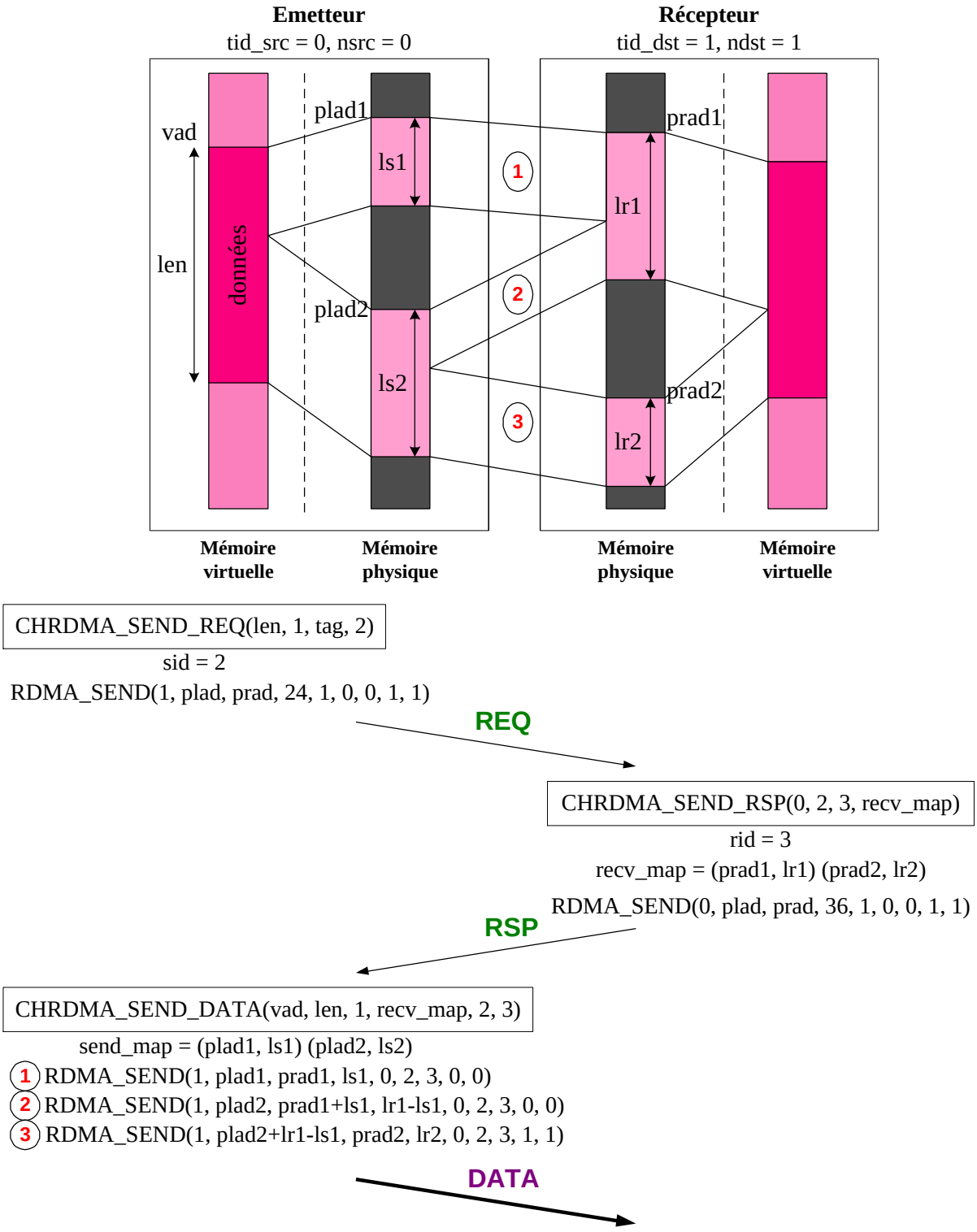


Figure IV-11 : Un exemple d'utilisation de CHRDMA_SEND_DATA

IV.5.8. La signalisation des événements réseau

Nous discutons ici de la façon de faire remonter les événements réseau (fin d'émission ou fin de réception d'un message de la couche CH_RDMA). Nous avons besoin de signaler les événements suivants :

- ✓ fin d'émission d'un message *CTRL*
- ✓ fin de réception d'un message *CTRL*
- ✓ fin d'émission d'un message *DATA*
- ✓ fin de réception d'un message *DATA*.

A chacun de ces quatre événements correspond un certain traitement à effectuer qui nécessite la connaissance d'informations récapitulées dans Tableau IV-4.

Traitement	Information nécessaire
`CHRDMA_CTL_SENT_EVENT`	Le tampon intermédiaire utilisé pour cette émission
`CHRDMA_CTL_RECV_EVENT`	Le tampon intermédiaire où le message a été déposé
`CHRDMA_DATA_SENT_EVENT`	La requête d'émission associée au message *DATA* (SHANDLE)
`CHRDMA_DATA_RECV_EVENT`	La requête de réception associée au message *DATA* (RHANDLE)

Tableau IV-4 : La signalisation

La primitive `CHRDMA_NET_LOOKUP` doit détecter la fin d'émission/réception d'un message. Elle peut être appelée pour plusieurs raisons :

- ✓ l'application a appelé une primitive de communication MPI bloquante,
- ✓ l'application souhaite s'informer de la complétion d'une communication,
- ✓ des ressources doivent être libérées (par exemple, s'il y a une famine de tampons intermédiaires pour les messages *CTRL*).

Pour faire remonter les informations de signalisation de la couche RDMA vers la couche CH_RDMA, nous utilisons un ensemble de drapeaux qui peuvent prendre la valeur 0 ou 1. On distingue quatre types de drapeaux : ceux associés à une fin d'émission d'un message *CTRL*, ceux associés à une fin de réception d'un message *CTRL*, ceux associés à une fin d'émission d'un message *DATA*, et ceux associés à une fin de réception d'un message *DATA*. Lorsqu'un drapeau vaut 1, cela signifie que la transmission/réception associée est terminée. La fonction de la primitive `CHRDMA_NET_LOOKUP` est de faire une boucle de scrutation sur l'ensemble des drapeaux. Lorsqu'elle trouve un drapeau à 1, elle le remet à 0 et appelle la primitive du tableau ci-dessus (celle qui correspond au type du drapeau) en lui fournissant les informations dont elle a besoin. Par conséquent, dans notre implémentation, les deux fonctions de rappel `RDMA_SENT_NOTIFY` et `RDMA_RECV_NOTIFY` ont pour

unique tâche de mettre à 1 le bon drapeau à partir des informations qui lui sont transmises. Les drapeaux doivent être accessibles en écriture par les couches CH_RDMA et RDMA.

Deux problèmes se posent : pour la fonction `CHRDMA_NET_LOOKUP`, la difficulté consiste à retrouver les informations du Tableau IV-4 correspondant au drapeau que l'on trouve à 1 ; pour les primitives `RDMA_SENT_NOTIFY` et `RDMA_RECV_NOTIFY`, la difficulté est de trouver le drapeau qu'il faut mettre à 1 à partir des informations fournies par le matériel. Nous rappelons que `RDMA_SENT_NOTIFY` doit disposer des paramètres `ctrl` (message *CTRL* ou message *DATA*) et `sid` (identificateur de la requête d'émission) et que `RDMA_SENT_NOTIFY` doit disposer des paramètres `ctrl`, `rid` (identificateur de la requête de réception) et `nsrc` (numéro du nœud qui a transmis le message).

Pour nos expérimentations, nous avons utilisé la primitive d'écriture distante de la machine MPC (appelée PUT). Elle est présentée au chapitre III. L'Annexe B explique comment nous avons fait le lien entre l'API RDMA et l'API PUT.

Le Tableau IV-5 présente le nombre de drapeaux nécessaires pour chacun des quatre types d'événements et pour chaque nœud de la machine. Soient les constantes suivantes :

- ✓ `NBUF` : le nombre de tampons intermédiaires pour les messages *CTRL* dans une file circulaire d'émission ou de réception (`NBUF` = 4 dans la Figure IV-4),
- ✓ `NNODES` : le nombre de nœuds constituant la grappe de PCs,
- ✓ `NSID` : le nombre maximum d'émissions simultanées utilisant le protocole de rendez-vous sur une tâche MPI,
- ✓ `NRID` : le nombre maximum de réceptions simultanées utilisant le protocole de rendez-vous sur une tâche MPI.

Type des drapeaux	Nombre nécessaire
`SCTL_FLAG` (émission des messages *CTRL*)	`NBUF`
`RCTL_FLAG` (réception des messages *CTRL*)	`NBUF*(NNODES-1)`
`SDATA_FLAG` (émission des messages *DATA*)	`NSID`
`RDATA_FLAG` (réception des messages *DATA*)	`NRID`

Tableau IV-5 : Le nombre de drapeaux de signalisation

Pour les messages *CTRL*, il y a autant de drapeaux que de tampons intermédiaires. Pour les messages *DATA*, il y a autant de drapeaux que d'émissions/réceptions simultanément possibles.

La Figure IV-12 présente le mécanisme de signalisation utilisé dans MPI-MPC1. Dans cette première implémentation, la couche RDMA se trouve dans l'espace système et le contrôleur réseau utilise une interruption matérielle pour signaler une fin d'émission

ou de réception. Tous les drapeaux se trouvent dans l'espace système et sont projetés dans la mémoire virtuelle de chacune des tâches MPI lors de leur initialisation. Ils sont alors accessibles en lecture/écriture par les couches RDMA et CH_RDMA.

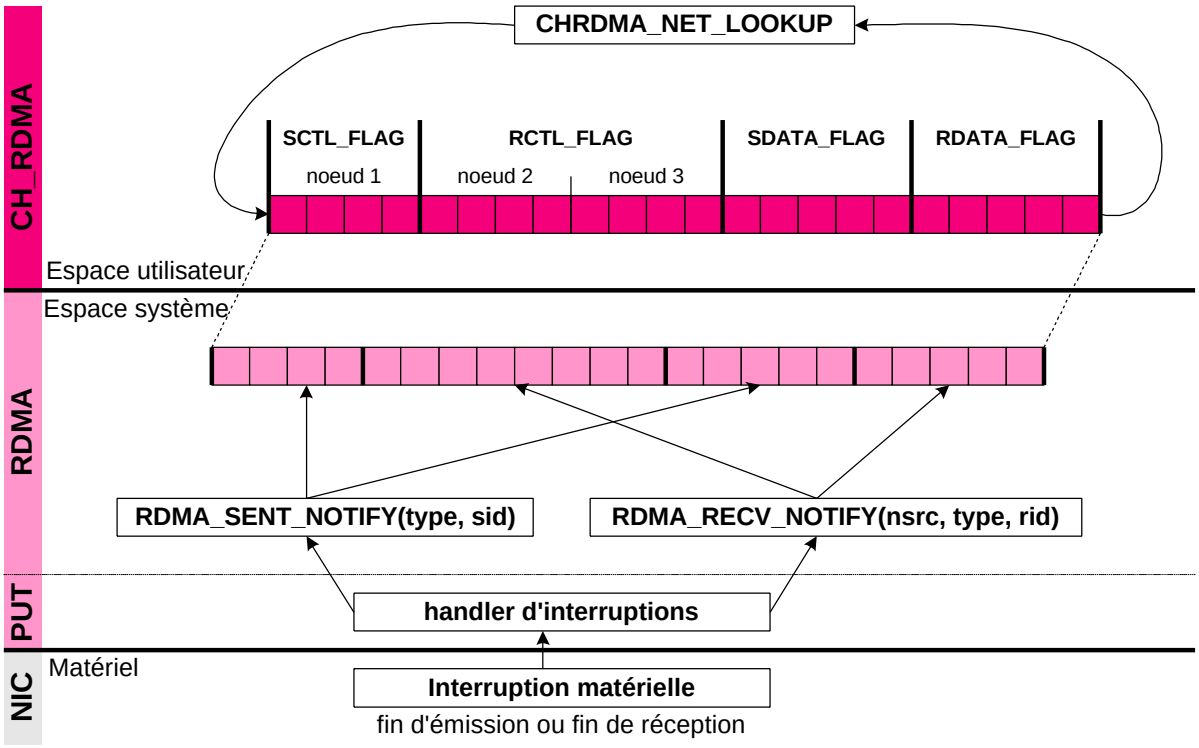


Figure IV-12 : La signalisation dans MPI-MPC1

L'interruption matérielle déclenche l'exécution du gestionnaire d'interruptions. Celui-ci retrouve dans ses tables les informations concernant la primitive d'écriture qui a provoqué l'interruption, en déduit les champs (`ctrl`, `sid`) ou (`nsrc`, `ctrl`, `rid`) et appelle la primitive `RDMA_SENT_NOTIFY` ou `RDMA_RECV_NOTIFY` si la notification avait été demandée lors du `RDMA_SEND` (`ns`=1 ou `nr`=1). `RDMA_SENT_NOTIFY` met à jour un des drapeaux `SCTL_FLAG` (si `ctrl`=1) ou `SDATA_FLAG` (si `ctrl`=0). `RDMA_RECV_NOTIFY` agit sur un des drapeaux `RCTL_FLAG` (si `ctrl`=1) ou `RDATA_FLAG` (si `ctrl`=0). La primitive `CHRDMA_NET_LOOKUP` scrute l'ensemble des drapeaux. Lorsqu'elle trouve un drapeau à 1, elle le remet à 0 et appelle la fonction qui doit traiter l'événement associé. Il reste à préciser comment la fonction `CHRDMA_NET_LOOKUP` associe un tampon à un drapeau (cf. Tableau IV-4) et comment la primitive `RDMA_SENT_NOTIFY` ou `RDMA_RECV_NOTIFY` trouve le drapeau qu'elle doit mettre à 1.

a) Concernant les messages *DATA* (`ctrl`=0), le fonctionnement est le suivant :

- ✓ `RDMA_SENT_NOTIFY` met à 1 le drapeau numéro `sid` de `SDATA_FLAG`.
- ✓ `RDMA_RECV_NOTIFY` met à 1 le drapeau numéro `rid` de `RDATA_FLAG`.
- ✓ `CHRDMA_NET_LOOKUP` trouve à 1 un drapeau de `SDATA_FLAG` / `RDATA_FLAG` et en déduit le `sid/rid` correspondant. Pour retrouver la requête d'émission/réception associée (`SHANDLE/RHANDLE`), elle cherche dans la table des émissions/réceptions en cours l'entrée numéro `sid/rid` (cf. Figure IV-10).

Dans le cas où le protocole de rendez-vous nécessite l'envoi de plusieurs messages *DATA*, il n'est pas gênant de conserver le même `sid/rid` durant toute la transaction : il n'est pas possible d'envoyer/recevoir un message *DATA* tant que la fin d'émission/réception du précédent n'a pas été signalée (à cause du message *RSP* qui s'intercale entre deux messages *DATA* successifs).

b) Concernant les messages *CTRL* (`ctrl`=1), il faut mémoriser sur chaque nœud dans la couche RDMA quel est le prochain tampon intermédiaire à signaler pour chacune des files d'émission/réception. La Figure IV-13 illustre ce mécanisme sur le nœud 1 avec une machine composée de trois nœuds. Dans la file d'émission des messages *CTRL*, la prochaine fin d'émission concernera le tampon numéro 2. La prochaine réception d'un message *CTRL* provenant du nœud 3 concernera le tampon numéro 3.

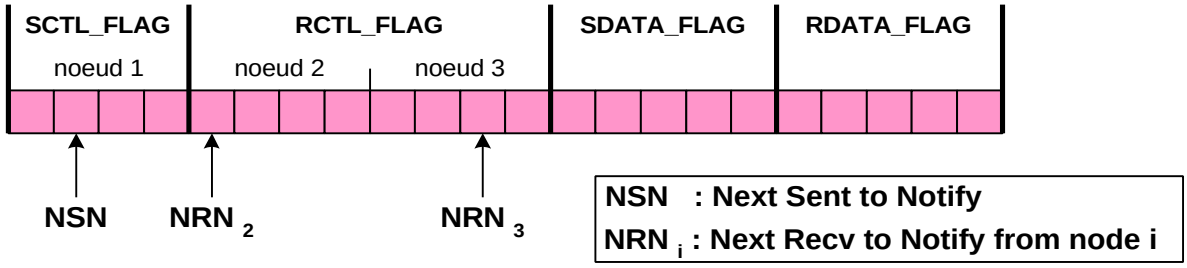


Figure IV-13 : La signalisation des messages de contrôle

- ✓ `RDMA_SENT_NOTIFY` met à 1 le drapeau numéro `NSN` de `SCTL_FLAG` et incrémente `NSN` (modulo le nombre de tampons présents dans la file).
- ✓ `RDMA_RECV_NOTIFY` est appelée avec le paramètre `nsrc` (numéro du nœud émetteur). Elle met à 1 le drapeau numéro NRN_{nsrc} de la file de réception du nœud `nsrc` et incrémente NRN_{nsrc}.
- ✓ `CHRDMA_NET_LOOKUP` trouve à 1 un drapeau de `SCTL_FLAG` / `RCTL_FLAG` et déduit à partir du numéro du drapeau le tampon intermédiaire associé.

Ce mécanisme fonctionne car nous supposons que le réseau a un comportement FIFO et que les messages sont signalés par le contrôleur réseau dans l'ordre où ils ont été postés.

IV.6. Mesures de performances avec un « *ping-pong* » MPI

Pour mesurer les performances de notre première implémentation de MPI au-dessus d'une primitive d'écriture distante, nous utilisons un « ping-pong » tel qu'il a été décrit dans la section II.7.1. Les primitives de communication MPI utilisées sont `MPI_Send` et `MPI_Recv` qui correspondent au mode standard de communication dans MPI (cf. section IV.5.5).

IV.6.1. La plate-forme expérimentale

La primitive d'écriture distante que nous utilisons pour nos expérimentations est celle de la machine MPC. Du point de vue matériel, cela repose sur le contrôleur réseau de la carte *FastHSL* et du point de vue logiciel sur la couche bas niveau d'écriture distante de la machine MPC, appelée PUT. Cette couche de communication est très proche du matériel. Elle se situe dans le système d'exploitation et utilise un mécanisme de signalisation par interruption. L'Annexe B explique comment nous avons interfacé la couche RDMA avec la couche PUT. La Figure IV-14 précise l'architecture de MPI-MPC1 utilisée pour nos expérimentations.

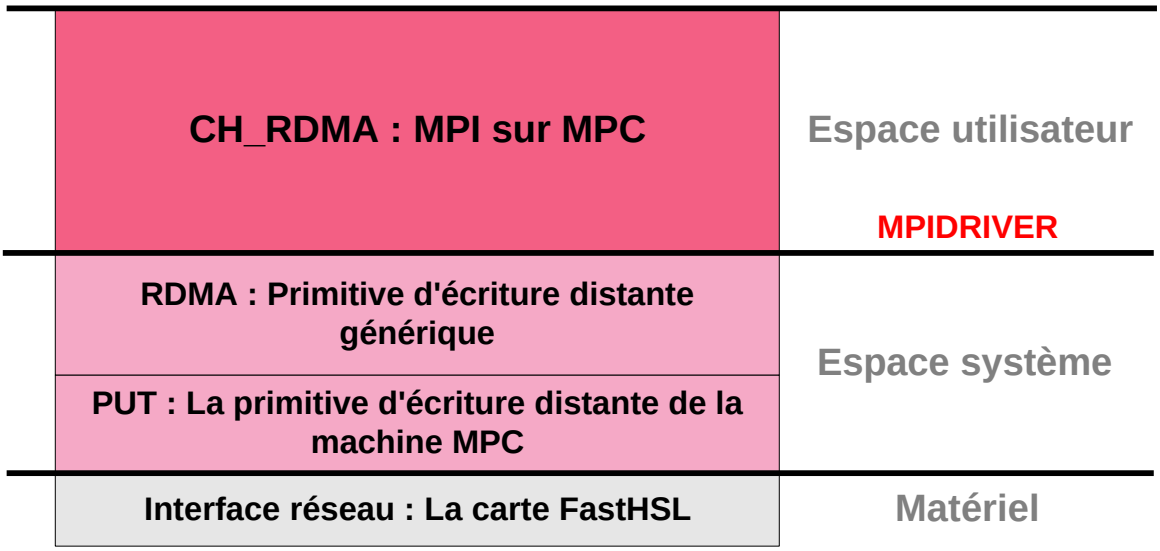


Figure IV-14 : Architecture bas niveau de MPI-MPC1

Le banc de test utilisé est une machine MPC constituée de deux nœuds. Chaque nœud est équipé d'un processeur PentiumII-350MHz, de 256Mo de mémoire physique et d'un bus PCI à 33Mhz. Le système d'exploitation utilisé est un noyau LINUX 2.2.18.

IV.6.2. Le seuil optimal pour les messages de contrôle

La première expérimentation consiste à mesurer le seuil optimal pour les messages *CTRL*, c'est-à-dire à déterminer la taille maximale des messages *CTRL*. En utilisant la

primitive de communication `MPI_Send`, les données peuvent être transférées suivant deux modes (cf. Figure IV-8) :

- ✓ si les données peuvent être encapsulées dans un message *CTRL*, le transfert se fait en un seul message mais deux recopies intermédiaires sont nécessaires (une en émission et une en réception),
- ✓ si la taille des données est trop grande, le transfert utilise le protocole de rendez-vous mais il n'y a pas de recopie intermédiaire des données.

Déterminer la taille maximale des messages *CTRL* consiste à mesurer à partir de quelle taille l'utilisation du protocole de rendez-vous devient plus performante que le transfert direct avec recopies intermédiaires. L'expérimentation consiste à mesurer les temps de transferts pour différentes tailles de messages, dans les deux modes de transmission. Pour forcer l'encapsulation des données dans un message *CTRL*, il suffit de fixer une très grande taille maximale pour les messages *CTRL*. Pour forcer le protocole de rendez-vous, même pour les messages de très petite taille, il suffit d'utiliser le mode synchrone de MPI et donc de remplacer dans le ping-pong `MPI_Send` par `MPI_Ssend`. Ces mesures permettent de tracer deux courbes de débit dont le point d'intersection est le seuil optimal recherché. La Figure IV-15 présente les résultats obtenus sur notre banc de test.

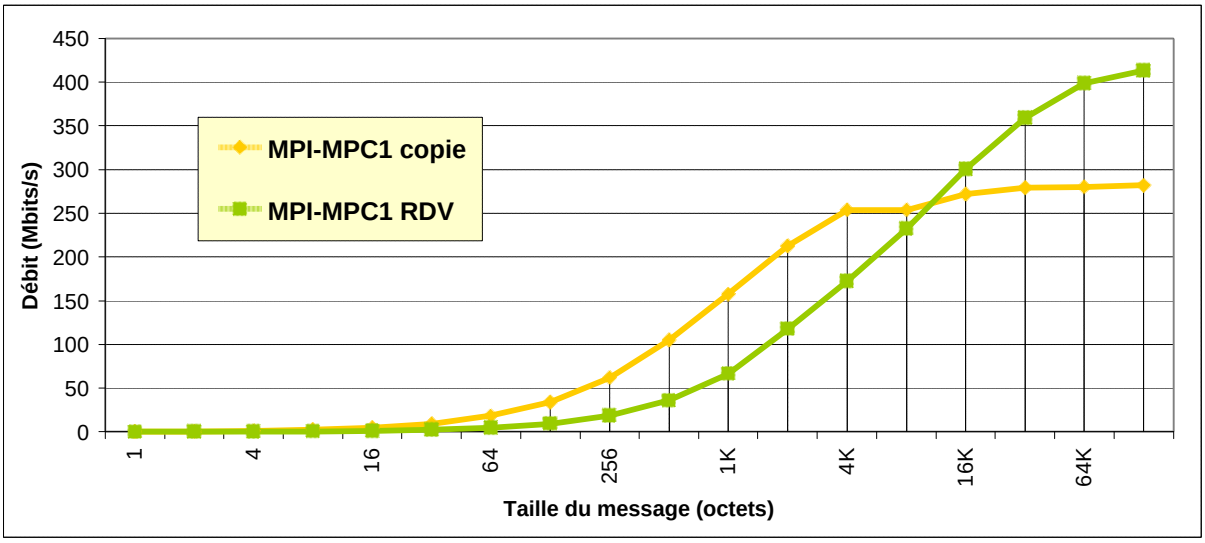


Figure IV-15 : Le seuil optimal des messages de contrôle dans MPI-MPC1

La courbe « MPI-MPC1 copie » correspond au mode de transfert utilisant des recopies intermédiaires et la courbe « MPI-MPC1 RDV » correspond au mode de transfert par le protocole de rendez-vous. Pour des messages de petite taille, le protocole de rendez-vous est beaucoup moins performant car il nécessite un échange en trois phases avec une traduction d'adresse des tampons d'émission et de réception. En revanche, à partir de 16Ko, les recopies intermédiaires deviennent pénalisantes.

Sur notre implémentation MPI-MPC1, le protocole de rendez-vous devient plus performant que l'encapsulation des données dans un message *CTRL* à partir de 16Ko.

IV.6.3. Courbe de débit

Maintenant que la taille maximale optimale des messages *CTRL* a été déterminée, nous pouvons tracer la courbe de débit de MPI-MPC1. Les résultats obtenus sont représentés sur la Figure IV-16.

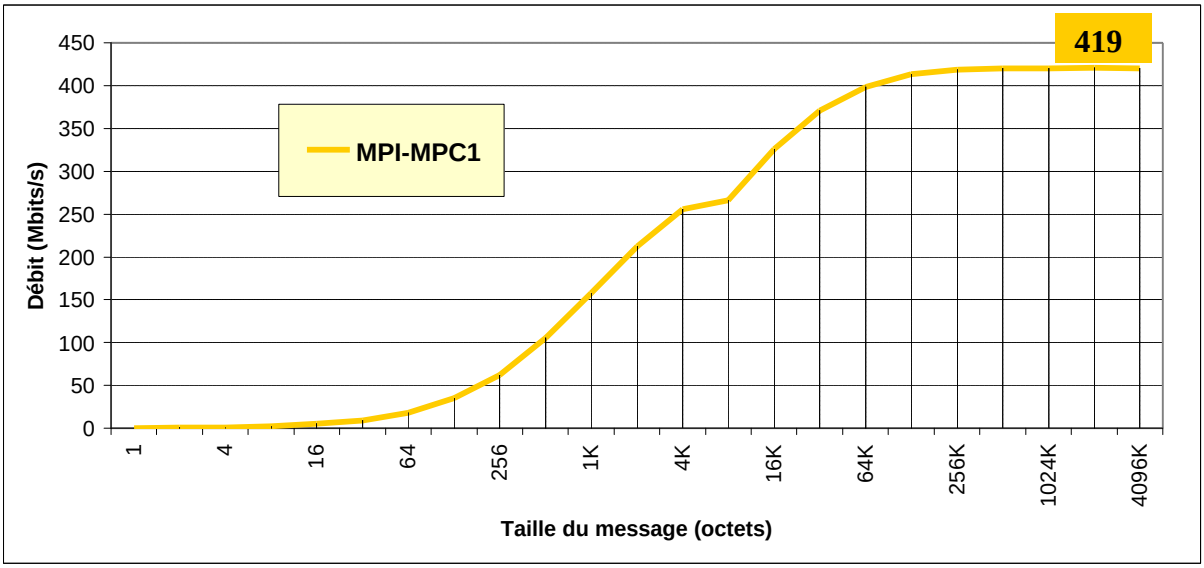


Figure IV-16 : Courbe de débit pour MPI-MPC1

Pour réaliser cette courbe, nous avons fixé la taille maximale des messages *CTRL* à 9020 octets et la taille maximale d'un message *DATA* à un million d'octets. Le point d'inflexion correspond au changement du mode de transfert des données. A partir de 1024Ko, les données sont fragmentées en plusieurs messages *DATA*. Le débit utile maximum au niveau de MPI-MPC1 est de **419Mbits/s**.

IV.6.4. Le demi débit

Le demi débit correspond à la taille de message en octets pour lequel le débit est égal à la moitié du débit utile maximal. Plus la valeur du demi débit est faible, meilleures sont les performances.

La Figure IV-17 montre que le demi débit de MPI-MPC1 est de 2Ko ce qui est un assez bon résultat.

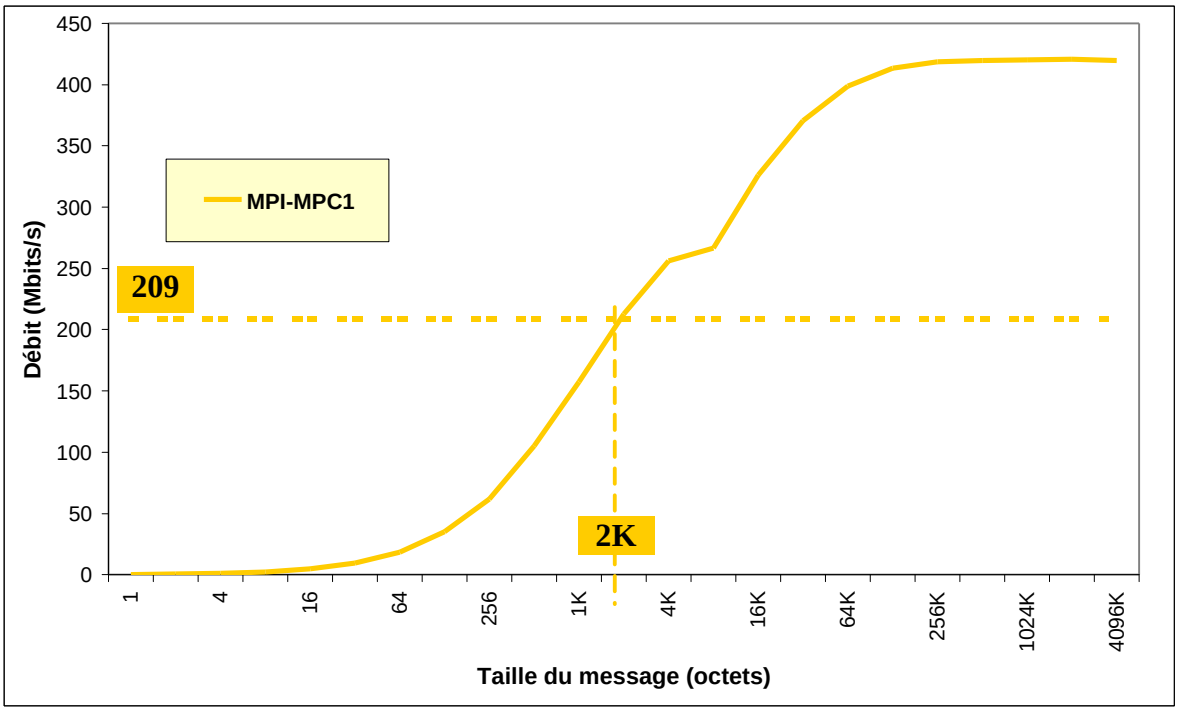


Figure IV-17 : Demi débit de MPI-MPC1

IV.6.5. Latence matérielle et latence logicielle

Nous avons mesuré sur notre plate-forme expérimentale, constituée de processeurs cadencés à 350MHz, le temps nécessaire pour envoyer 1 octet de données avec MPI-MPC1 du nœud émetteur vers le nœud récepteur. Ce temps, appelé latence, correspond à un aller (ou *ping*) en utilisant le « ping-pong » MPI.

La latence (L) peut se décomposer en deux parties :

$$L = S + H \quad \text{avec}$$

- ✓ S : latence logicielle strictement proportionnelle à la vitesse du processeur,
- ✓ H : latence matérielle indépendante de la vitesse du processeur.

La primitive d'écriture distante est réalisée à travers le réseau HSL sur notre plate-forme expérimentale. Dans notre implémentation MPI-MPC1, la latence matérielle représente essentiellement le temps nécessaire au matériel (contrôleurs réseau émetteur/récepteur) pour transférer un message de 21 octets (le message *CTRL* dans lequel les données sont encapsulées contient une entête de 20 octets et 1 octet de données), le temps de prise en compte de l'interruption matérielle et le temps pour lire ou modifier les registres internes du contrôleur réseau. Les registres de la carte FastHSL sont accessibles uniquement par des accès en configuration à travers le bus PCI qui sont relativement coûteux : nous avons évalué le coût d'un tel accès à 1,5µs. [Renault,2001] l'évalue à 1850ns.

Plus précisément, la latence de MPI-MPC1 se décompose en différentes phases :

- ✓ appel de la primitive `MPI_Send` sur le nœud émetteur pour transférer un octet de données,
- ✓ encapsulation de l'octet de données dans un message *CTRL* de type *SHORT* (20 octets d'en-tête + 1 octet de données),
- ✓ recopie du message *CTRL* vers le tampon intermédiaire d'émission contigu en mémoire physique,
- ✓ appel système pour exécuter la primitive `RDMA_SEND` (ajout d'une entrée dans la Liste des Messages à Emettre),
- ✓ accès en configuration pour indiquer au contrôleur réseau qu'il y a une nouvelle entrée à émettre (modification d'un pointeur sur la Liste des Messages à Emettre),
- ✓ transfert des 21 octets constituant le message *CTRL* sur le réseau HSL,
- ✓ interruption matérielle sur le nœud récepteur et deux accès en configuration pour prendre en compte la fin de réception des données (il s'agit de lire puis de mettre à jour les pointeurs sur la Liste des Messages Reçus),
- ✓ appel de la primitive `RDMA_RECV_NOTIFY`
- ✓ traitement de l'en-tête du message *CTRL* reçu et recopie de l'octet de données dans le tampon de réception de l'application.

L'interruption matérielle en émission n'est pas comptabilisée dans la latence car elle se produit en parallèle de la réception. Nous avons évalué le temps de l'appel système vide (temps pour rentrer dans le noyau et en ressortir aussitôt) à 1,5µs sur notre plate-forme expérimentale. Le temps de transfert d'un octet sur le réseau HSL est compris entre 1,7µs et 2,1µs [Fenÿo,2001]. Nous pouvons donc considérer que le temps de transfert des 21 octets du message *CTRL* coûte environ 2µs. La Figure IV-18 schématise les différentes phases intervenant dans la latence de MPI-MPC1.

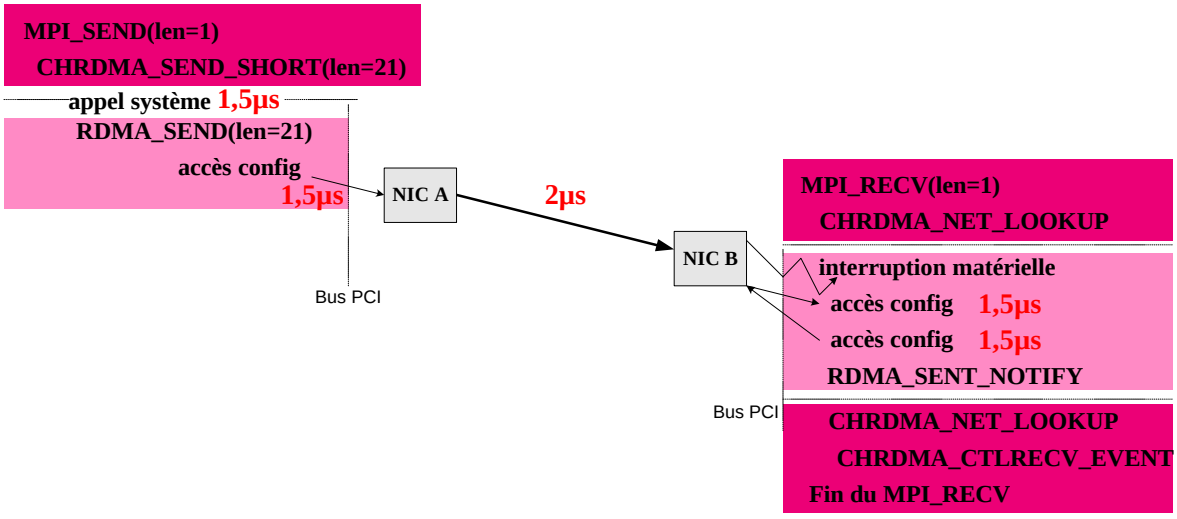


Figure IV-18 : Décomposition de la latence

La latence totale (logicielle + matérielle) pour transférer un message d'un octet est de 26µs sur notre plate-forme expérimentale constituée de processeurs cadencés à 350MHz. La latence matérielle liée à la plate-forme choisie (incluant la traversée du réseau HSL et la modification des registres internes du contrôleur réseau) a été évaluée à 6,5µs.

IV.6.6. Tableau récapitulatif

Le Tableau IV-6 récapitule les performances de MPI-MPC1. La latence (correspondant au temps de transfert de 1 octet) mesurée sur notre plate-forme expérimentale est de 26µs. Ce temps inclut la traversée des couches MPI, l'encapsulation des données dans un message *CTRL*, la recopie dans le tampon intermédiaire d'émission, l'appel de la primitive d'écriture distante, la latence matérielle du transfert, la signalisation sur le récepteur, l'extraction de l'octet de donnée et la recopie en réception.

	Latence (µs)	Débit Max. (Mb/s)	Seuil (Ko)	Demi débit (Ko)	Latence RDV (µs)
MPI-MPC1	**26**	**419**	**16**	**2**	**105**

Tableau IV-6 : Performances de MPI-MPC1

La dernière colonne du tableau correspond à la latence mesurée avec le protocole de rendez-vous. Cela correspond au temps de transfert de trois messages *CTRL* auquel s'ajoute le verrouillage et la traduction d'adresse du tampon de réception et du tampon d'émission.

IV.6.7. Analyse des performances de MPI-MPC1

Au delà des caractéristiques du matériel utilisé pour réaliser la primitive d'écriture distante, les principaux facteurs influençant les performances sont :

- ✓ le nombre d'appels système,
- ✓ le nombre d'interruptions matérielles,
- ✓ le nombre de recopies des données,
- ✓ le nombre d'écritures distantes nécessaires pour transmettre un message.

Le Tableau IV-7 comptabilise le nombre de chacun de ces facteurs pour un transfert haut niveau (un *ping* MPI). On distingue le cas où les données sont transmises directement encapsulées dans un message *CTRL* (protocole *short*), du cas où les données sont transmises suivant le protocole de rendez-vous. La colonne RDMA_SEND correspond au nombre d'écritures distantes nécessaires pour transmettre les données de l'application (c'est-à-dire le nombre d'appels à la primitive `RDMA_SEND`).

	Appels systèmes		Interruptions		Recopies		RDMA_SEND	
Protocole	short	rdv	short	rdv	short	rdv	short	rdv
MPI-MPC1	**1**	**1+6F**	**2**	**2+4F**	**2**	**0**	**1**	**1+F+NB_DMA*F**

Tableau IV-7 : Analyse des performances de MPI-MPC1

Dans ce tableau, F est le nombre de messages *DATA* nécessaires pour transmettre les données (F dépend de la taille des données à transférer). NB_DMA est le nombre d'écritures distantes nécessaires pour transmettre un message *DATA*. Il dépend de la description en mémoire physique des tampons d'émission et de réception.

Lorsque le mode de transfert utilisé est l'encapsulation des données dans un message *CTRL*, il y a un appel système qui correspond à l'appel de la primitive d'écriture distante. Un appel à RDMA_SEND est suffisant car les tampons intermédiaires d'émission/réception correspondent tous les deux à une zone contiguë en mémoire physique. Deux recopies des données de l'application sont nécessaires : une en émission et une en réception. Enfin, deux interruptions matérielles interviennent : une pour signaler la fin de transmission du message sur l'émetteur et une pour signaler la terminaison de la réception sur le récepteur.

Lorsque le transfert se fait suivant le protocole de rendez-vous, 1+F messages *CTRL* (*REQ* + F messages *RSP*) et F messages *DATA* sont échangés (soit un total de 1+2F messages). Chaque message implique un appel système pour appeler la primitive d'écriture distante et deux interruptions matérielles. Cela correspond à 2+4F interruptions matérielles et 1+2F appels système auxquels il faut ajouter les appels système causés par le verrouillage et la traduction d'adresse des tampons d'émission/réception. Pour chaque message *DATA*, il y a donc 4 appels système supplémentaires. D'où un total de 1+6F appels système. Enfin, pour réaliser un transfert des données de l'application, il faut 1+F écritures distantes pour envoyer les messages *CTRL* et NB_DMA*F écritures distantes pour transmettre les données.

Sur notre plate-forme expérimentale, nous avons évalué le coût d'un appel système vide (rentrer dans le noyau et en ressortir). Cela peut se mesurer en utilisant l'appel système standard `get_pid()` qui consiste à récupérer le numéro du processus en cours d'exécution. Ce coût est de 1,5µs pour un processeur PII-350MHz. Par ailleurs, une interruption matérielle est très coûteuse car elle provoque un changement de contexte du processus en cours d'exécution. Le coût des traductions d'adresse dépend de la taille du tampon dont on souhaite obtenir la description en mémoire physique. Nous expliquons dans l'Annexe A en quoi consiste une telle opération. Cependant, nous pouvons évaluer le coût des verrouillage/traduction des tampons d'émission et de réception : le temps de transfert d'un octet en utilisant le protocole de rendez-vous, mesuré sur notre plate-forme, est de 105µs (cf. Tableau IV-6). La latence de transfert d'un message *CTRL* étant de 26µs, en supposant que le coût de transfert d'un message *DATA* d'un octet est proche de 26µs également, il en résulte que les coûts des deux verrouillage/traduction (nécessitant quatre appels système) est de l'ordre d'une vingtaine de micro-secondes. Il faudrait donc environ 10µs pour verrouiller puis

traduire un tampon d'une taille d'un octet. Nous aurons l'occasion de confirmer cette affirmation au chapitre VI qui traite du problème des traductions d'adresse.

Nous avons mesuré une latence de 26µs et un débit utile de 419Mbits/s. On peut comparer ces performances, mesurées au niveau applicatif MPI, avec les performances « brutes » de la plate-forme matérielle MPC. La latence d'un appel à la primitive d'écriture distante de la machine MPC a été mesurée par Alexandre Fenÿo dans [Fenÿo,2001] : elle est proche de 5µs sur le même banc de test que nous avons utilisé pour réaliser nos expérimentations. Le débit utile maximum pour transmettre des tampons contigus en mémoire physique en utilisant la primitive d'écriture distante de la machine MPC est de 494Mbits/s [Fenÿo,2001]. On voit donc que dans cette première implémentation, le coût de traversée des couches logicielles reste relativement élevé au regard des performances matérielles du réseau HSL.

IV.7. Conclusion

Nous avons décrit dans ce chapitre une première implémentation (MPI-MPC1) du standard de communication MPI au-dessus d'une primitive d'écriture en mémoire distante.

Les deux principaux problèmes intrinsèques à la sémantique de la primitive d'écriture distante sont :

- ✓ le fait qu'elle travaille en adresse physique alors que les tampons de l'application sont décrits en adresse virtuelle,
- ✓ la réception des données se fait sans aucun contrôle du processeur local : l'émetteur doit savoir avant d'émettre les données où les écrire dans la mémoire physique du récepteur.

Les deux solutions proposées sont :

- ✓ soit d'utiliser des tampons intermédiaires contigus en mémoire physique, ce qui permet à l'émetteur de savoir où écrire sur le nœud récepteur mais nécessite deux recopies des données (une en réception et une en émission),
- ✓ soit d'utiliser un protocole de rendez-vous qui permet de faire des transmissions de type « zéro-copie » dans lequel le récepteur envoie à l'émetteur la description en mémoire physique du tampon de réception ; ce protocole nécessite un verrouillage en mémoire physique et une traduction d'adresse des tampons d'émission/réception de l'application.

La première solution est avantageuse pour les transmissions de message de petite taille alors que la deuxième s'avère performante pour les messages de grande taille.

Nous avons également montré que ces deux modes de transfert permettent de respecter la sémantique de toutes les primitives de communication point à point définies par le standard MPI [MPI,1994].

Nous avons défini une API générique d'écriture en mémoire distante, appelée RDMA, permettant de faire le lien entre notre implémentation de MPI et n'importe quelle plate-forme utilisant une primitive d'écriture en mémoire distante. Nous avons expliqué comment réaliser la signalisation de fin d'émission/réception des messages internes à MPI (messages *CTRL* et messages *DATA*).

Nous avons mesuré, au niveau applicatif MPI, une latence de 26µs et un débit utile de 419Mbits/s. Cette première implémentation a prouvé qu'il est possible d'obtenir des performances assez proches des performances obtenues sur le réseau Myrinet (MPI/BIP, MPI/GM, MPI/PM), alors que les services fournis par le réseau sont assez différents : notre implémentation MPI n'utilise qu'une primitive d'écriture distante en mémoire physique, et ne suppose pas l'existence d'une primitive de réception. Par opposition, le contrôleur réseau de Myrinet étant programmable, des bibliothèques de communication PM, GM ou BIP peuvent y stocker, dans une table, la correspondance entre les adresses virtuelles et les adresses physiques. Ces protocoles utilisent même un cache logiciel, accessible directement par le contrôleur réseau, sauf dans le cas de BIP. Cela permet non seulement d'accélérer la traduction d'adresse mais aussi de réaliser des communications sur des canaux virtuels (un *tag* dans BIP, un *port* dans GM ou l'adresse virtuelle du tampon de réception dans PM). L'émetteur n'a donc pas besoin de connaître les adresses physiques du tampon de réception de l'application, ce qui n'est pas le cas avec l'API RDMA. Par ailleurs, BIP, GM et PM fournissent des primitives de réception qui facilitent une signalisation par scrutation.

Les limites de cette première implémentation (MPI-MPC1) proviennent de l'utilisation systématique des appels système et des interruptions matérielles. L'implémentation MPI-MPC2, décrite au chapitre V, a pour objectif l'élimination des interruptions matérielles et des appels système, hormis ceux résultant des traductions d'adresse. L'implémentation MPI-MPC3, présentée au chapitre VI, proposera une méthode pour s'affranchir des opérations de traduction d'adresse.

Chapitre V : Optimisations des couches basses de communication

Sommaire

L'objectif général de ce chapitre est d'éliminer les interruptions matérielles et les appels système du chemin critique logiciel séparant l'appel à une primitive de communication point à point de MPI de la prise en compte du message par le matériel réseau.

V.1. Introduction

Nous avons décrit au Chapitre IV une première implémentation de MPI au-dessus d'une primitive d'écriture distante : MPI-MPC1. Celle-ci reposait sur une primitive d'écriture distante accessible au travers d'appels système et utilisant une signalisation par interruption matérielle. La primitive d'écriture distante de notre plate-forme expérimentale se trouvait dans l'espace système, essentiellement pour des raisons de partage des ressources réseau et d'accès au contrôleur réseau se trouvant sur le bus PCI.

L'objectif de ce chapitre est triple :

- ✓ montrer qu'il est possible de partager les ressources réseau tout en accédant aux composants matériel en mode utilisateur, c'est-à-dire en éliminant les appels système des phases de communication (à l'exception des appels système intrinsèques aux traductions d'adresse qui seront traités au Chapitre VI),
- ✓ remplacer les interruptions matérielles, qui entraînent des changements de contexte et nécessitent un traitement dans l'espace système, par une signalisation reposant sur une scrutation de la Liste des Messages Reçus et de la Liste des Messages à Emettre en mode utilisateur (sans faire intervenir de changement de contexte),
- ✓ montrer que l'élimination des appels système et des interruptions a une influence importante sur les performances : nous avons obtenu au Chapitre IV pour MPI-MPC1 une latence de 26µs et un débit maximum de 419Mbits/s, ce qui semble encore assez éloigné des performances intrinsèques de notre plate-forme expérimentale.

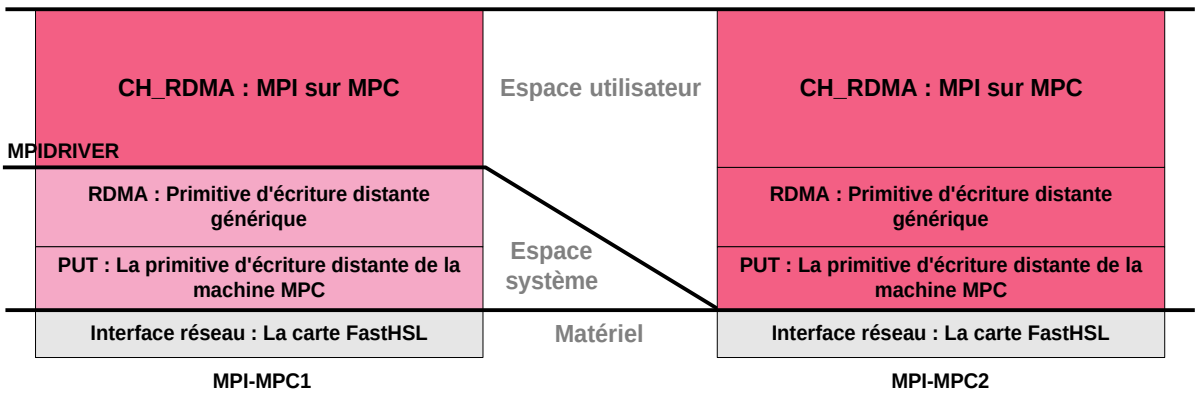


Figure V-1 : Architectures de MPI-MPC1 et MPI-MPC2

Nous appellerons MPI-MPC2 cette deuxième implémentation de MPI, qui reposera sur une primitive d'écriture distante en mode utilisateur. La Figure V-1 rappelle l'architecture bas niveau de MPI-MPC1 que nous avons définie au Chapitre IV et présente l'architecture de MPI-MPC2.

Bien que l'implémentation MPI-MPC1, présentée au Chapitre IV, suppose qu'il y a au maximum une tâche MPI par nœud de calcul, nous souhaitons que la primitive d'écriture en mémoire distante décrite dans ce chapitre permette des accès concurrents aux ressources réseau. Nous considérons que cette limitation était une simplification temporaire qui n'a aucune justification de principe.

Une interface en mode utilisateur, appelée PAPI (*Pci-ddc Application Programming Interface*), de la primitive d'écriture distante de la machine MPC qui constitue notre plate-forme expérimentale a été proposée par Eric Renault dans [Renault_1,2000] [Renault_3,2000]. Elle est présentée au chapitre III. Cependant, nous avons préféré développer notre propre interface pour les deux raisons suivantes : d'une part, PAPI ne permet pas le partage des ressources de la carte réseau entre plusieurs processus s'exécutant en mode utilisateur ; d'autre part, nous souhaitons avoir la même interface pour la primitive d'écriture distante en mode utilisateur et en mode noyau.

V.2. La primitive d'écriture distante de la machine MPC

Nous décrivons dans cette section la couche bas niveau réalisant la primitive d'écriture distante que nous avons utilisée pour nos expérimentations dans le Chapitre IV. Cette couche de communication, appelée PUT, se trouve dans le système d'exploitation. Elle peut être considérée comme une implémentation de l'API générique RDMA définie au chapitre précédent. La Figure V-2 présente les différents modules de la couche PUT.

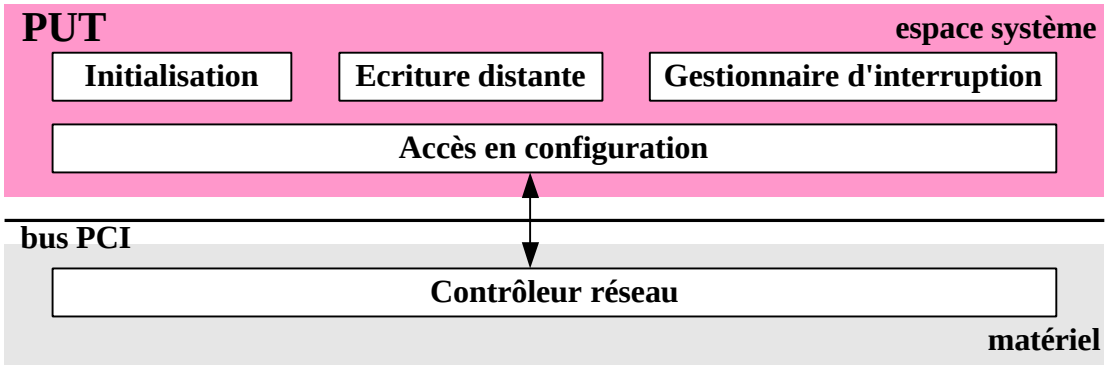


Figure V-2 : Structure de la couche bas niveau de la machine MPC

La phase d'initialisation permet de configurer le contrôleur réseau et le routeur de la carte FastHSL. Le module d'écriture distante transmet les ordres d'écriture distante au contrôleur réseau. Cela consiste en l'ajout d'une entrée dans la Liste des Messages à Emettre (cf. section II.2) se trouvant en mémoire physique et en un accès en configuration pour indiquer au contrôleur réseau qu'une entrée a été ajoutée. Le

module de gestion des interruptions reçoit et traite les interruptions provenant du contrôleur réseau et fait remonter les informations de signalisation à l'application qui fournit deux fonctions de rappel (une pour la fin d'émission et une pour la fin de réception) lors de la phase d'initialisation. Enfin, le module « Accès en configuration » permet d'aller lire/écrire dans les registres internes du contrôleur réseau par des accès en configuration à travers le bus PCI.

Les ressources partagées du contrôleur réseau sont :

- ✓ la Liste des Messages à Emettre (LME),
- ✓ la Liste des Messages Reçus (LMR),
- ✓ les registres internes du contrôleur réseau.

Nous avons décrit dans la section II.2 le rôle de la LME et de la LMR. Dans le cas de la machine MPC, ces listes se trouvent en mémoire centrale et non pas dans la mémoire interne du contrôleur réseau. Celui-ci ne fait que conserver deux pointeurs sur chacune des listes et va en modifier le contenu par accès DMA. La Figure V-3 représente la LME, la LMR et les pointeurs associés d'un nœud de calcul.

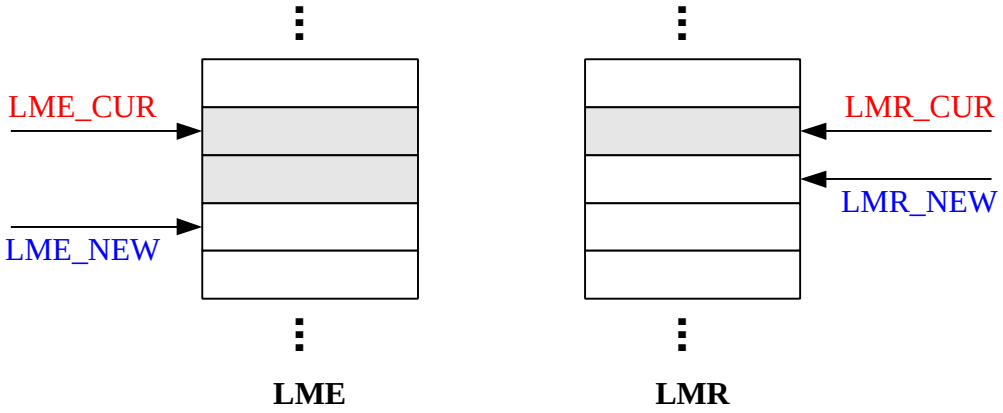


Figure V-3 : Les pointeurs sur la LME et la LMR

Ces pointeurs se trouvent dans les registres internes du contrôleur réseau mais une copie est conservée en mémoire centrale. Les pointeurs LME_CUR et LMR_NEW sont incrémentés par le contrôleur réseau lors d'une fin d'émission ou d'une fin de réception. Inversement, les pointeurs LME_NEW et LMR_CUR sont modifiés par le logiciel, c'est-à-dire par la couche PUT, lorsqu'une entrée a été ajoutée dans la LME en émission ou lorsqu'une fin de réception a été prise en compte. Dans le cas de la Figure V-3, deux entrées de la LME n'ont pas encore été traitées par le contrôleur réseau et une entrée de la LMR n'a pas encore été prise en compte par la couche PUT.

Une contrainte supplémentaire est imposée au logiciel : toutes les entrées de la LME correspondant à un même message au niveau applicatif doivent être consécutives dans la LME. Pour l'application, un message est une zone contiguë en mémoire virtuelle qui se trouve être une succession de tampons éparpillés en mémoire physique.

V.3. Vers une primitive d'écriture distante en mode utilisateur

L'objectif de cette section est d'identifier les problèmes à résoudre pour aboutir à une primitive d'écriture distante en mode utilisateur utilisant une signalisation par scrutation.

V.3.1. Hypothèses

Comme nous l'avons signalé précédemment, nous supposons que plusieurs tâches MPI locales à un même nœud sont susceptibles de se partager les ressources réseau. Nous devons donc protéger les accès multiples à ces ressources par l'utilisation de verrous.

En revanche, nous supposons toujours qu'il n'y a qu'une seule application MPI qui s'exécute à un instant donné sur la machine.

V.3.2. Structure de PUT en mode utilisateur

La Figure V-4 représente la structure de la couche PUT à laquelle nous souhaitons aboutir.

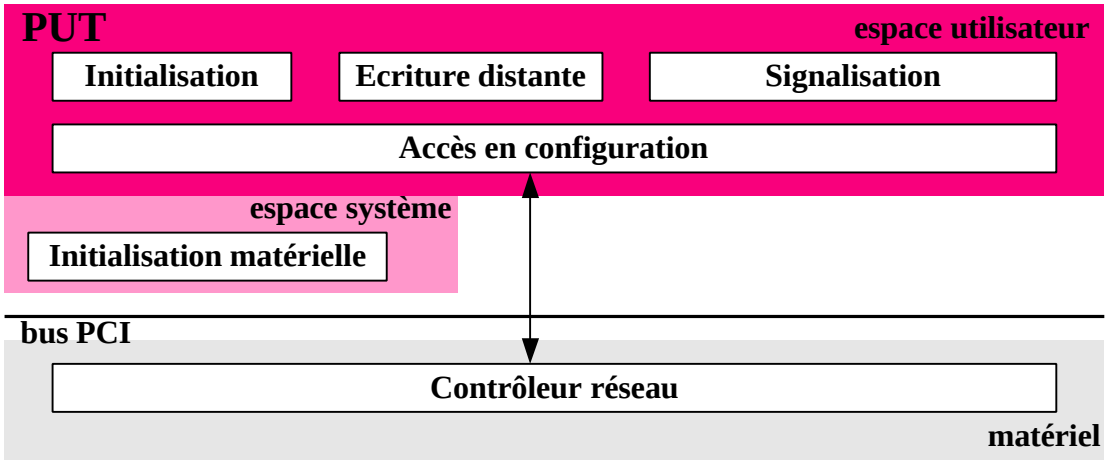


Figure V-4 : Structure de PUT en mode utilisateur

Le module « Gestionnaire d'interruption » de la Figure V-2 est remplacé par le module « Signalisation » dont le rôle est de faire une scrutation sur la LME et la LMR dont le principe sera exposé dans la suite de ce chapitre. Une partie de la phase d'initialisation (initialisation des composants matériels) reste dans l'espace système car elle est faite une fois pour toute lors du démarrage de la machine et n'influe absolument pas sur les performances de l'application. Le module « Initialisation » se trouvant dans l'espace utilisateur permet à chaque processus constituant l'application, lors de sa phase d'initialisation, d'allouer des structures de données, de projeter dans son espace

mémoire certaines ressources partagées (comme la LME et la LMR), d'indiquer au module « Signalisation » les fonctions de rappel à appeler lors d'une fin d'émission/réception, etc. Les modules « Ecriture distante » et « Accès en configuration » ne doivent pas faire appel au système d'exploitation.

En pratique, la couche PUT devient une librairie statique qui est liée à chaque processus constituant l'application lors de la compilation. Elle travaille donc dans la mémoire virtuelle du processus qui s'exécute.

V.3.3. Identification des problèmes à résoudre

Dans un système de communication au niveau utilisateur, des problèmes de protection ou d'isolation des processus les uns par rapport aux autres doivent être résolus. Chaque processus possède un espace d'adressage qui lui est propre mais les ressources réseau doivent être partagées entre les différents processus utilisateur qui y accèdent.

Les problèmes à résoudre sont les suivants :

- ✓ Comment réaliser le partage des listes LME et LMR. Lorsque la couche PUT se trouvait dans le système d'exploitation, chacune de ces listes était unique et accessible uniquement en mode noyau. Comment rendre les listes LME et LMR accessibles en lecture/écriture, non seulement par les processus utilisateur constituant l'application, mais aussi par le contrôleur réseau local ?
- ✓ Comment réaliser des accès concurrents aux registres internes du contrôleur réseau en mode utilisateur ? En effet, le module « Accès en configuration » se trouvant maintenant dans l'espace utilisateur, il faut que chaque processus puisse lire/écrire dans un registre se trouvant sur la carte réseau sans passer par le système d'exploitation et en garantissant la cohérence des lectures/écritures.
- ✓ Comment réaliser la signalisation des événements réseau de manière efficace, sans utiliser les interruptions matérielles ? Il existe différentes politiques de scrutation, entre l'attente active où le processeur boucle jusqu'à ce que l'événement attendu se produise et l'attente passive qui consiste à aller consulter de temps en temps les événements survenus.
- ✓ Comment garantir que toutes les entrées d'un même message soient consécutives dans la LME ?

V.4. Implémentation de PUT en mode utilisateur

Nous décrivons dans cette section comment nous sommes passés d'une primitive d'écriture distante en mode noyau avec une signalisation par interruption à une primitive d'écriture distante en mode utilisateur avec une signalisation par scrutation.

V.4.1. Principe de la méthode

Le principe est de conserver une LME et une LMR uniques dans l'espace système et de les projeter dans l'espace mémoire de chaque processus de l'application lors de la phase d'initialisation. Un processus pourra alors lire ou modifier une entrée de ces listes sans utiliser d'appel système. La même stratégie est adoptée pour les quatre pointeurs décrits dans la Figure V-3. Par ailleurs, ces listes doivent être accessibles également par le contrôleur réseau qui utilise des accès DMA. Elles doivent donc être verrouillées et se trouver dans une zone contiguë en mémoire physique. Pour cela, nous utilisons l'allocateur de mémoire physique que nous avons mentionné dans la section IV.5.1. Lors de l'initialisation des composants matériels, les adresses en mémoire physique de ces listes sont transmises au contrôleur réseau.

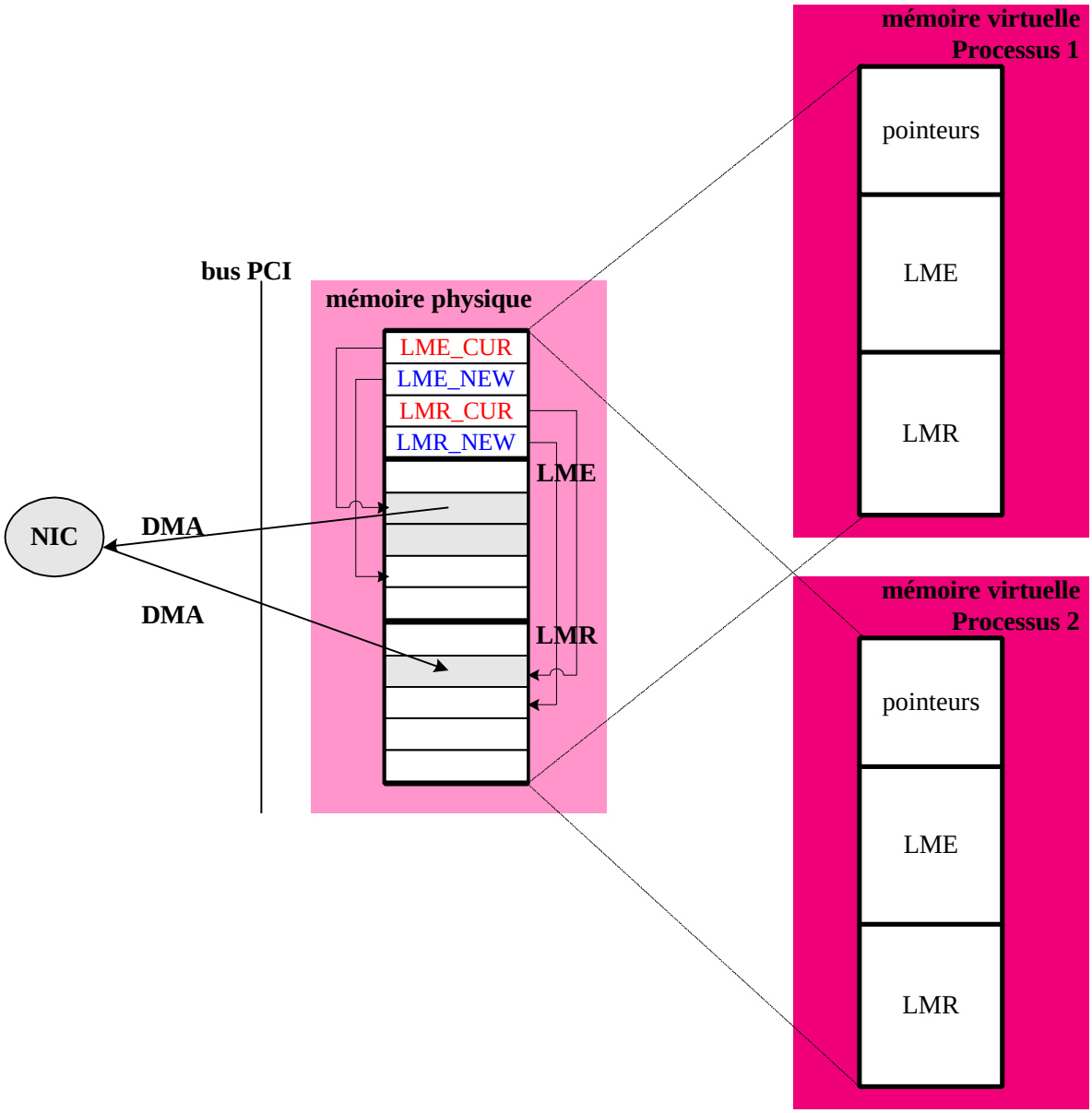


Figure V-5 : Les accès à la LME/LMR en mode utilisateur

La Figure V-5 schématise les emplacements de la LME et de la LMR dans le cas où deux processus utilisateur partagent les ressources de la carte réseau. Nous n'avons pas représenté la projection dans la mémoire virtuelle du noyau. Le contrôleur réseau lit la LME et écrit dans la LMR, par accès DMA. Les quatre pointeurs représentés sur la figure sont des copies de ceux qui se trouvent dans les registres internes du contrôleur réseau.

Puisque ces ressources sont partagées par plusieurs processus, il faut garantir le caractère atomique des accès à ces ressources par un verrou spécifique dans l'espace utilisateur. Avant tout accès à une ressource, un processus doit acquérir le verrou pour obtenir un accès exclusif, et relâcher le verrou lorsqu'il a terminé l'opération. Cette question est traitée dans la section V.4.5.

V.4.2. Emission d'un message

Une écriture distante en mode utilisateur réalise toujours les deux étapes suivantes :

- ✓ ajout d'une entrée dans la LME décrivant l'écriture distante que le contrôleur réseau doit réaliser,
- ✓ incrémentation du pointeur LME_NEW se trouvant dans un registre interne du contrôleur réseau pour lui indiquer qu'une nouvelle entrée doit être traitée.

Lorsqu'un processus veut envoyer un message (nous appelons message un tampon contigu en mémoire virtuelle dont la fin d'émission/réception doit être signalée, il se peut que son émission nécessite plusieurs écritures distantes et donc l'ajout de plusieurs entrées dans la LME (cf. Figure IV-11). Ces entrées doivent impérativement être consécutives dans la LME. Lorsque la primitive d'écriture distante se trouvait dans l'espace système, il suffisait pour émettre un message de faire un seul appel système et de faire tous les appels à la primitive `RDMA_SEND` dans une section critique empêchant les interruptions pendant l'ajout des entrées dans la LME. En mode utilisateur, un problème peut se produire si un processus commence à ajouter dans la LME les entrées correspondant à un message et qu'un deuxième processus prend la main pour lui aussi ajouter une nouvelle entrée. Il est donc nécessaire de définir une nouvelle fonction pour l'ajout d'un message dans la LME et de protéger son exécution par un verrou atomique.

V.4.3. Les accès en configuration

Les registres du contrôleur réseau de la machine MPC sont accessibles uniquement par des accès en configuration. La Figure V-6 représente les différents composants qui entrent en jeu lors d'un accès en configuration.

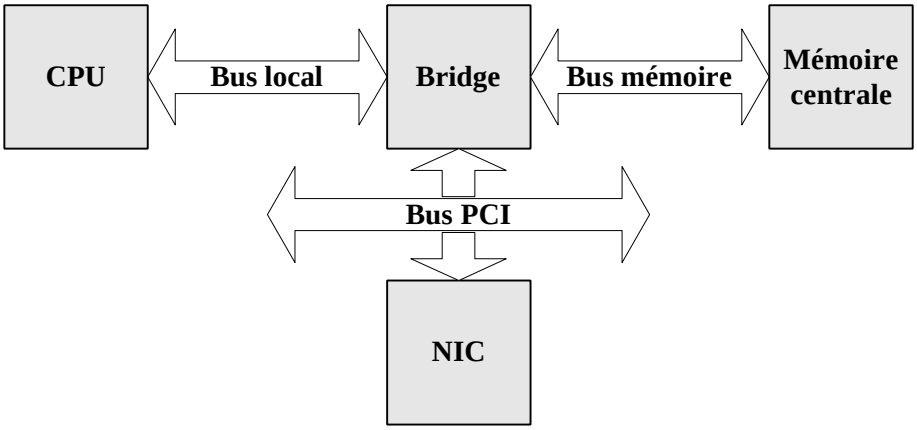


Figure V-6 : Les accès en configuration

Un accès en configuration se déroule en deux phases :

- ✓ le processeur écrit dans un registre du *bridge Host/PCI* l'adresse du registre qu'il veut lire ou modifier sur la carte FastHSL (cette adresse contient non seulement l'adresse du registre sur la carte mais aussi l'adresse PCI de la carte elle-même) et le *bridge* lui retourne un port sur lequel lire ou écrire,
- ✓ le processeur va lire/écrire la valeur dans le registre de la carte sur le port spécifié.

Il faut donc faire deux accès d'entrée/sortie pour réaliser un accès en configuration.

Nous souhaitons réaliser ces deux phases depuis un processus utilisateur. Les systèmes UNIX que nous utilisons permettent de faire des accès d'entrée/sortie depuis un processus utilisateur à condition de lui en donner le privilège. Cela se fait par l'intermédiaire de la fonction `iopl` (*change I/O privilege level*) de la librairie C que l'on exécute lors de la phase d'initialisation de l'application. Cependant, l'opération qui consiste à faire deux accès I/O successifs n'est pas atomique. Nous sommes capables d'empêcher deux processus de notre application de faire des accès en configuration simultanément en utilisant un verrou spécifique connu de tous les processus de l'application MPI, mais nous ne pouvons pas éviter qu'un autre processus, extérieur à l'application MPI, ou même le noyau fasse un accès vers une autre carte PCI simultanément. Fort heureusement, les accès en configuration sont en principe réservés à la phase d'initialisation des cartes PCI. Nous avons pu constater sur notre plate-forme expérimentale que de tels accès ne se produisaient que lors du démarrage de la machine.

V.4.4. La signalisation par scrutation

Afin d'éviter les changements de contexte causés par les interruptions matérielles, nous souhaitons réaliser une signalisation par scrutation. Nous avons ajouté deux primitives à la couche PUT réalisant la primitive d'écriture distante sur notre plate-forme expérimentale :

- ✓ la primitive `PUT_FLUSH_LME` réalisant la signalisation des fins d'émission,
- ✓ la primitive `PUT_FLUSH_LMR` réalisant la signalisation des fins de réception.

En ce qui concerne les fins d'émission, cette scrutation est réalisée sur les entrées non vides de la LME, se trouvant au-dessus de l'entrée pointée par LME_CUR : il s'agit des entrées dont l'émission est terminée mais dont la signalisation éventuelle doit être traitée. `PUT_FLUSH_LME` réalise les opérations suivantes :

- ✓ sauvegarde du pointeur LME_CUR dans LME_CUR_OLD,
- ✓ mise à jour de LME_CUR par une lecture en configuration sur le contrôleur réseau,
- ✓ si LME_CUR est différent de LME_CUR_OLD, faire pour chaque entrée de la LME située entre ces deux pointeurs :
  - regarder si l'application a demandé que cette entrée soit signalée,
  - si c'est le cas, appeler la fonction de rappel associée aux fins d'émission (`RDMA_SENT_NOTIFY`),
  - passer à l'entrée suivante.

Pour les fins de réception, toutes les entrées déposées dans la LMR par le contrôleur réseau doivent être signalées. Comme le contrôleur réseau écrit dans la LMR, nous pouvons éviter un accès en configuration coûteux pour lire le pointeur LMR_NEW. Les différentes étapes de la scrutation sont :

- ✓ tant que l'entrée pointée par LMR_CUR n'est pas vide, faire les opérations suivantes :
  - appeler la fonction de rappel associée aux fins de réception (`RDMA_RECV_NOTIFY`),
  - marquer l'entrée vide,
  - passer à l'entrée suivante (incrémentation de LMR_CUR)
- ✓ mettre à jour le pointeur LMR_CUR sur le contrôleur réseau par un accès en configuration.

Ces deux primitives réalisant la scrutation doivent être appelées par l'application : c'est le rôle de la primitive `RDMA_NET_LOOKUP` que nous avons introduite dans la section IV.4. La Figure V-7 montre les modifications apportées à la signalisation dans MPI-MPC2 par rapport à celle de MPI-MPC1 présentée sur la Figure IV-12.

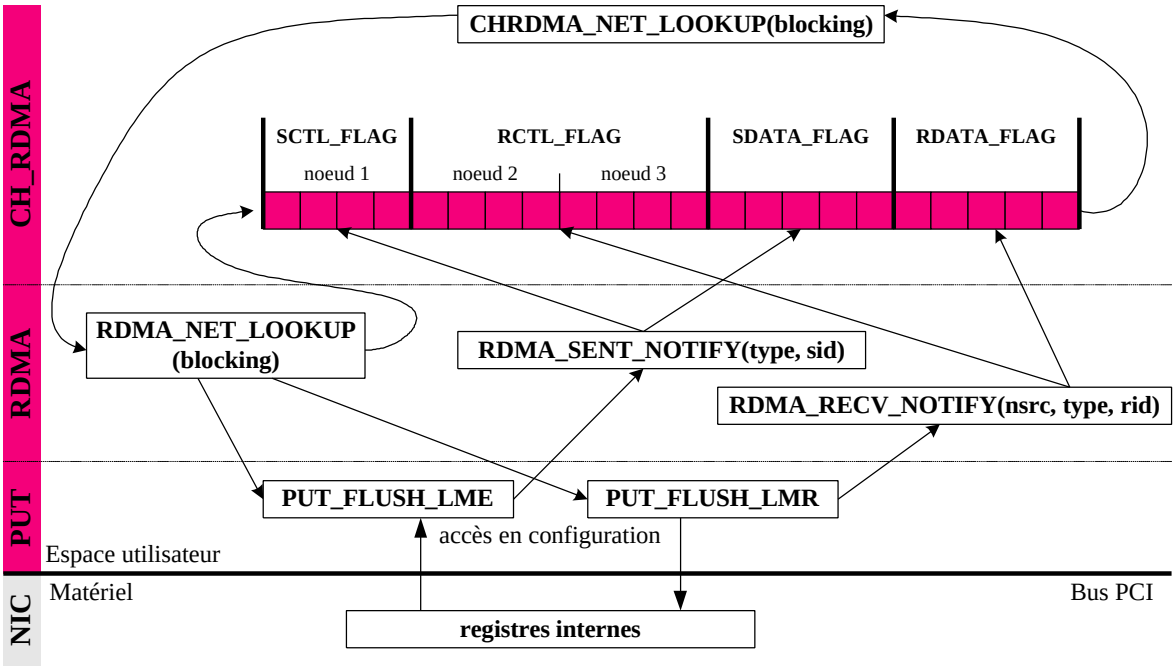


Figure V-7 : La signalisation dans MPI-MPC2

Désormais, il n'y a plus besoin de faire appel au système d'exploitation pour réaliser les opérations de signalisation. La primitive `CHRDMA_NET_LOOKUP` est appelée par l'application MPI lorsque celle-ci est en attente d'une fin d'émission ou d'une fin de réception d'un message. Celle-ci appelle la primitive `RDMA_NET_LOOKUP` qui exécute `PUT_FLUSH_LME` et `PUT_FLUSH_LMR`. Ces deux primitives vont lire les entrées de la LME/LMR et font remonter les informations de signalisation via les primitives `RDMA_SENT_NOTIFY` et `RDMA_RECV_NOTIFY` qui mettent à jour les drapeaux de l'application en conséquence.

Le fait que la scrutation soit active ou passive est laissé au libre choix de l'application. Si la scrutation est bloquante (blocking=1), les fonctions `PUT_FLUSH_LME` et `PUT_FLUSH_LMR` seront appelées jusqu'à ce qu'une fin d'émission/réception soit détectée. Pour ce faire, ces deux primitives retournent le nombre d'événements signalés.

Dans le cas où deux processus (P1 et P2) faisant partie de l'application MPI s'exécutent sur un même nœud, nous sommes confrontés au problème suivant : lorsque les primitives `PUT_FLUSH_LME` et `PUT_FLUSH_LMR` sont exécutées par le processus P1, il se peut qu'elles détectent des fins d'émission/réception de messages concernant le processus P2. Si P1 tombe sur une entrée qui doit être signalée à P2, il arrête le parcours de la table correspondante et relâche le verrou en attendant que P2 prenne la main pour acquitter les entrées qui le concernent.

Comme nous le verrons dans la section V.5, la signalisation par scrutation est nettement plus efficace que la signalisation par interruption. Cependant, nous nous sommes rendu compte lors de nos expérimentations qu'une attente active pouvait être pénalisante dans le cas du transfert de gros messages. En effet, lorsqu'une tâche MPI attend une signalisation, elle tourne simplement dans la boucle de scrutation. Cela n'est pas grave car elle ne peut exécuter aucun travail utile tant qu'elle n'a pas reçu la signalisation attendue. Il faut cependant éviter que l'attente soit « trop active », c'est-à-dire que la boucle de scrutation ne sature le bus PCI par des accès en configuration qui peuvent consommer une grosse partie de la bande passante du bus et ralentir le transfert des données. La solution que nous avons adoptée est de ralentir la boucle de scrutation dans la primitive `RDMA_NET_LOOKUP` en introduisant une centaine de cycles entre deux accès au bus. Un compromis a dû être trouvé pour que ce ralentissement ne pénalise pas la latence tout en permettant un débit élevé.

Dans le cadre de notre plate-forme expérimentale, pour ne pas générer un trafic trop important sur le bus PCI et pour que la scrutation ne pénalise pas les performances de l'application, il est nécessaire de ralentir légèrement la boucle de scrutation lorsque l'application a choisi de faire une attente active.

V.4.5. Les différents verrous

Les ressources partagées concernées par le problème d'accès multiples sont la LME, la LMR, les pointeurs sur ces deux listes et les registres internes du contrôleur réseau. Les opérations qui accèdent à ces ressources partagées sont :

- ✓ `PUT_ADD_SEVERAL_ENTRIES` : ajout d'un nouveau message dans la LME,
- ✓ `PUT_FLUSH_LME` : signalisation des fins d'émission,
- ✓ `PUT_FLUSH_LMR` : signalisation des fins de réception,
- ✓ accès en configuration à un registre du contrôleur réseau.

Ces opérations doivent être exécutées de façon atomique. La solution consiste à associer à chacune de ces opérations un verrou. Un processus prend le verrou lorsqu'il débute l'opération et le libère quand l'opération est terminée. Si le verrou est déjà pris par un autre processus, il attend que celui-ci se libère. Il s'agit en fait du principe des sémaphores mais nous n'utilisons pas ces derniers car cela serait trop coûteux en terme de performance. Les verrous sont des variables mémoire partagées entre les différents processus. L'opération qui consiste à prendre ou à libérer un verrou doit être atomique au niveau du processeur. Le système LINUX que nous utilisons permet de faire un « *test & set* » en mode utilisateur et nous garantit donc l'atomicité de la prise et de la libération d'un verrou. Le Tableau V-1 récapitule les verrous que nous avons définis dans notre implémentation MPI-MPC2.

Nom du verrou	Opération associée	Ressources concernées
LME_LOCK	PUT_ADD_SEVERAL_ENTRIES	LME et LME_NEW
LME_FLUSH_LOCK	PUT_FLUSH_LME	LME_CUR
LMR_FLUSH_LOCK	PUT_FLUSH_LMR	LMR et LMR_CUR
CONFIG_LOCK	Accès en configuration	Registres internes

Tableau V-1 : Les verrous dans MPI-MPC2

On remarque, par exemple, qu'il n'est pas gênant qu'un processus commence un PUT_ADD_SEVERAL_ENTRIES et qu'entre temps un autre processus vienne faire un PUT_FLUSH_LME : ces deux opérations ne concernent pas les mêmes ressources.

V.5. Mesures de performances avec un « *ping-pong* » MPI

Cette section décrit les performances obtenues avec notre deuxième implémentation de MPI : MPI-MPC2. La différence entre MPI-MPC1 et MPI-MPC2 est l'utilisation d'une primitive d'écriture distante en mode utilisateur et le remplacement des interruptions par un mécanisme de scrutation. Les expérimentations présentées et les mesures effectuées sont analogues à celles présentées au chapitre IV. Nous utilisons toujours un « ping-pong » MPI pour déterminer les facteurs suivants : le seuil optimal pour les messages *CTRL*, le débit maximum, le demi débit, et la latence. Nous comparons les résultats obtenus à ceux que nous avions obtenus avec MPI-MPC1.

V.5.1. La plate-forme expérimentale

La plate-forme expérimentale utilisée est identique à celle décrite au chapitre IV : la primitive d'écriture distante est réalisée par le matériel de la machine MPC et les processeurs des nœuds de calcul sont cadencés à 350MHz.

La Figure V-8 présente l'architecture bas niveau de MPI-MPC2.

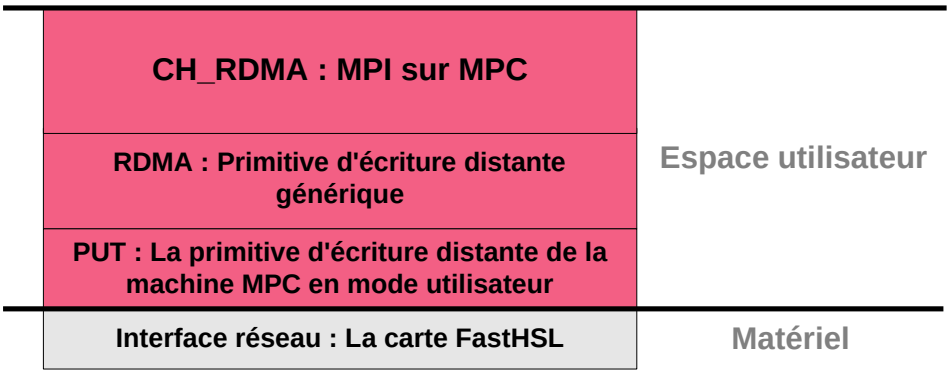


Figure V-8 : Architecture bas niveau de MPI-MPC2

V.5.2. Le seuil optimal pour les messages de contrôle

La première expérimentation consiste à mesurer le seuil optimal pour les messages *CTRL*, c'est-à-dire à déterminer à partir de quelle taille de message le protocole de rendez-vous devient plus efficace que l'encapsulation des données dans un message *CTRL* avec deux recopies intermédiaires des données de l'application. La Figure V-9 présente la courbe de débit en utilisant le protocole de rendez-vous et celle utilisant les recopies intermédiaires.

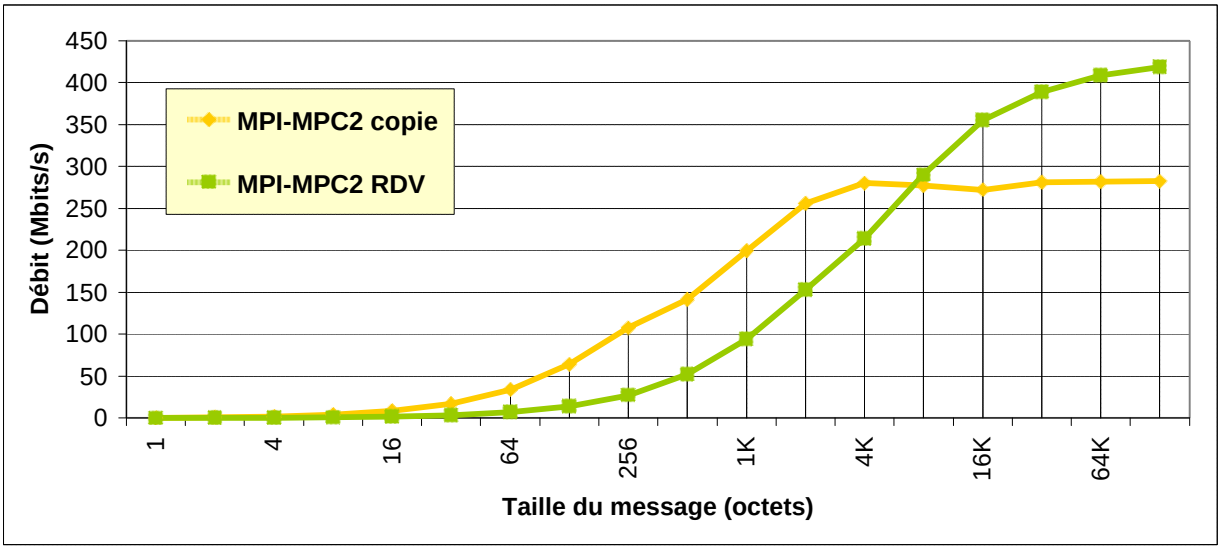


Figure V-9 : Le seuil optimal des messages de contrôle dans MPI-MPC2

Dans MPI-MPC2, le protocole de rendez-vous devient plus performant que l'encapsulation des données dans un message *CTRL* à partir de 8Ko.

V.5.3. Courbe de débit

Maintenant que nous avons déterminé la taille maximale des messages *CTRL*, nous pouvons tracer la courbe de débit de MPI-MPC2. La Figure V-10 représente les résultats obtenus en rappelant la courbe que nous avons obtenue pour MPI-MPC1.

Nous remarquons que l'utilisation de la primitive d'écriture distante en mode utilisateur et le remplacement d'une signalisation par interruption par une signalisation par scrutation permettent une nette amélioration du débit dans le cas des messages courts. En revanche, le débit maximum de MPI-MPC2 reste très proche de celui de MPI-MPC1.

L'augmentation du débit atteint presque un facteur 2 pour les petits messages (de longueur inférieure à 256 octets).

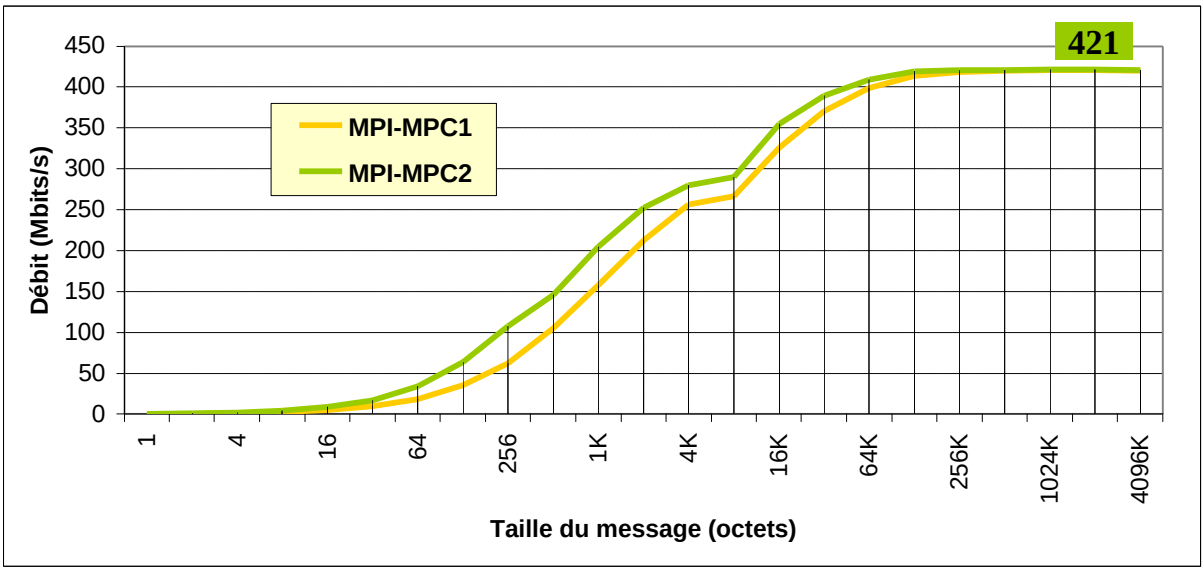


Figure V-10 : Courbe de débit de MPI-MPC2

V.5.4. Le demi débit

La Figure V-11 montre que la valeur du demi débit de MPI-MPC2 est de 1Ko ce qui est un meilleur résultat que ce que nous avions obtenu pour MPI-MPC1 (2Ko).

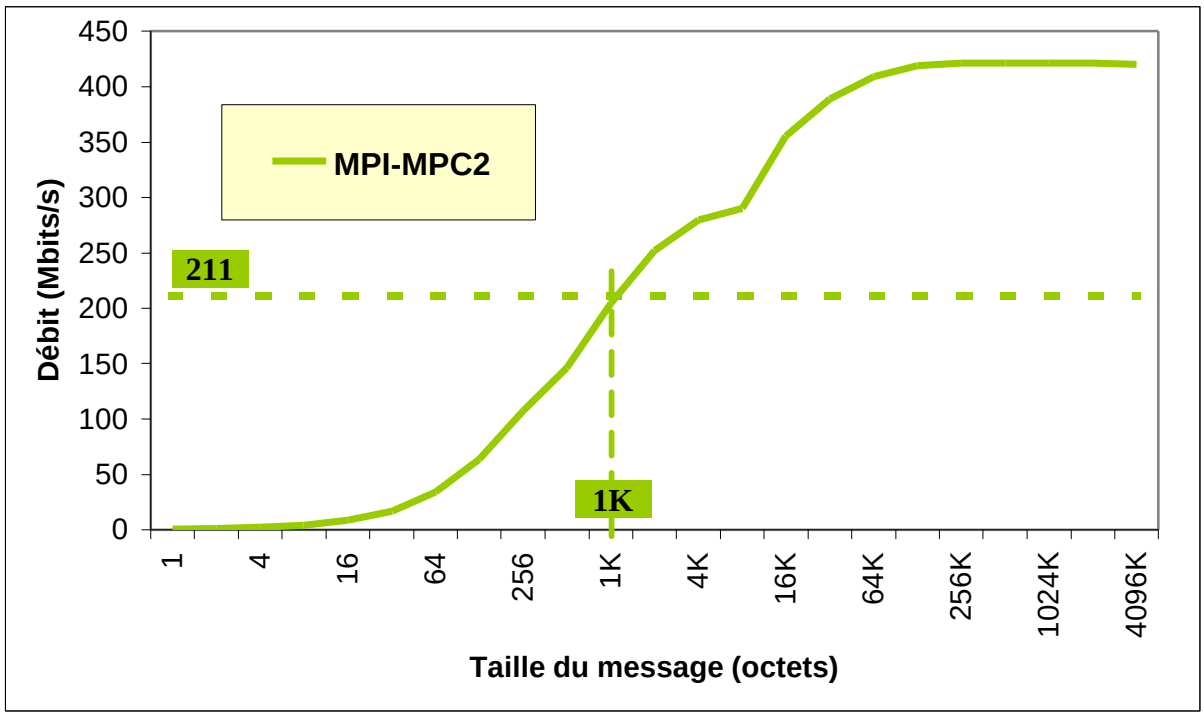


Figure V-11 : Demi débit de MPI-MPC2

V.5.5. Latence matérielle et latence logicielle

Les différentes phases intervenant dans la latence de MPI-MPC2 sont légèrement différentes de celles présentées au chapitre IV. Du côté de l'émetteur, la seule différence est l'élimination de l'appel système pour exécuter la primitive `RDMA_SEND`. Du côté récepteur, l'utilisation d'une signalisation par scrutation apporte quelques modifications.

- ✓ dès que la requête de réception est postée, la primitive `CHRDMA_NET_LOOKUP` est appelée avec le paramètre `blocking`=1,
- ✓ la primitive `RDMA_NET_LOOKUP` est également appelée avec `blocking`=1,
- ✓ quand le message arrive, un accès en configuration est nécessaire pour prendre en compte la réception (mise à jour du pointeur LMR_CUR),
- ✓ la primitive `RDMA_RECV_NOTIFY` est appelée,
- ✓ traitement de l'en-tête du message *CTRL* reçu et recopie de l'octet de données dans le tampon de réception de l'application.

La Figure V-12 récapitule les différentes phases intervenant dans la latence de MPI-MPC2. Comparativement à MPI-MPC1, il y a un appel système de moins sur l'émetteur, le traitement de l'interruption matérielle et un accès en configuration de moins sur le récepteur (cf. Figure IV-18).

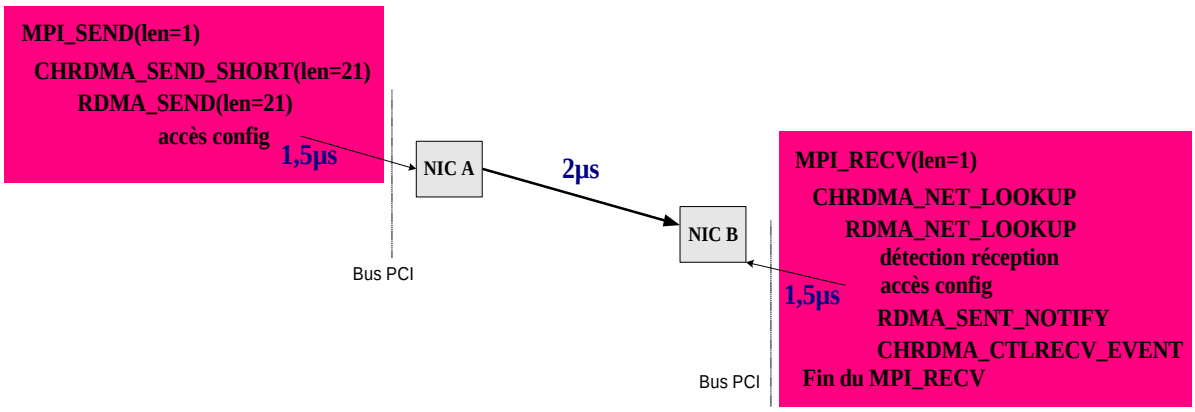


Figure V-12 : Décomposition de la latence dans MPI-MPC2

Dans MPI-MPC2, la latence matérielle se limite à deux accès en configuration et le transfert de 21 octets sur le réseau HSL de notre plate-forme expérimentale.

La latence totale que nous avons mesurée est de **15µs** pour MPI-MPC2. La latence matérielle du transfert est environ de 5µs (nous avons supprimé un accès en configuration sur le récepteur par rapport à MPI-MPC1). Le temps de traversée de toutes nos couches de communication est donc de 10µs sur des processeurs cadencés à 350MHz (soit 3500 cycles), ce qui constitue un excellent résultat. La latence logicielle étant proportionnelle à la fréquence du processeur, celle-ci serait inférieure à la latence matérielle sur un processeur cadencé à 1GHz. Nous avons atteint un de nos objectifs consistant à restituer à l'application les très bonnes performances du matériel réseau utilisé.

V.5.6. Tableau récapitulatif

Le Tableau V-2 récapitule les performances de MPI-MPC2. Les performances que nous avons obtenues pour MPI-MPC1 sont rappelées. La latence (correspondant au temps de transfert de 1 octet) mesurée sur notre plate-forme expérimentale est de 15µs avec MPI-MPC2 alors qu'elle était de 26µs avec MPI-MPC1.

	Latence (µs)	Débit Max. (Mb/s)	Seuil (Ko)	Demi débit (Ko)	Latence RDV (µs)
MPI-MPC1	26	419	16	2	105
MPI-MPC2	**15**	**421**	**8**	**1**	**72**

Tableau V-2 : Performances de MPI-MPC2

La latence mesurée avec le protocole de rendez-vous diminue elle aussi fortement comparativement à MPI-MPC1 car le temps pour envoyer un message est beaucoup plus faible avec MPI-MPC2. Les mesures semblent d'ailleurs cohérentes : 105-72=33µs ce qui correspond bien à 3*11µs (11µs étant l'écart entre la latence de MPI-MPC1 et celle de MPI-MPC2).

V.5.7. Analyse des performances

Le Tableau V-3 récapitule, pour MPI-MPC2, le nombre de chacun des facteurs que nous avons définis à la section IV.6.7 : appels système, interruptions, recopies des données de l'application, écritures distantes.

	Appels systèmes		Interruptions		Recopies		RDMA_SEND	
Protocole	short	rdv	short	rdv	short	rdv	short	rdv
MPI-MPC1	1	1+6F	2	2+4F	2	0	1	1+F+NB_DMA*F
MPI-MPC2	**0**	**4F**	**0**	**0**	**2**	**0**	**1**	**1+F+NB_DMA*F**

Tableau V-3 : Analyse des performances de MPI-MPC2

Les seuls changements entre MPI-MPC1 et MPI-MPC2 concernent le nombre d'appels système et d'interruptions. Les appels système restant dans MPI-MPC2 sont inhérents aux traductions d'adresse. Les interruptions matérielles ne sont plus utilisées dans MPI-MPC2.

En analysant plus précisément les latences respectives de MPI-MPC1 (26µs) et MPI-MPC2 (15µs), nous pouvons évaluer le coût inhérent à l'utilisation des interruptions. La différence entre les deux latences est de 11µs. En retranchant le coût de l'appel système (1,5µs) et le coût d'un accès en configuration (1,5µs), il reste 8µs d'écart correspondant au traitement de l'interruption matérielle en réception de MPI-MPC1.

Le gain de MPI-MPC2 par rapport à MPI-MPC1 concerne essentiellement la latence. L'utilisation des interruptions matérielles dans MPI-MPC1 est le facteur le plus pénalisant pour les performances.

La Figure V-13 représente le gain sur les temps de transfert avec MPI-MPC2 par rapport à ceux de MPI-MPC1 pour différentes tailles de message. La différence entre les temps de transfert de MPI-MPC1 et de MPI-MPC2 est indépendante de la taille des messages envoyés : elle ne dépend que du nombre de messages *CTRL* et de messages *DATA* échangés.

Le gain est quasiment constant jusqu'à une taille de message de 256 octets : il est légèrement supérieur à 40% ce qui correspond aux 11µs séparant la latence de MPI-MPC1 de celle de MPI-MPC2.

Au-dessus de 256 octets, le gain diminue assez nettement car le temps nécessaire aux recopies devient de plus en plus important et le temps nécessaire au matériel pour transférer les données augmente également.

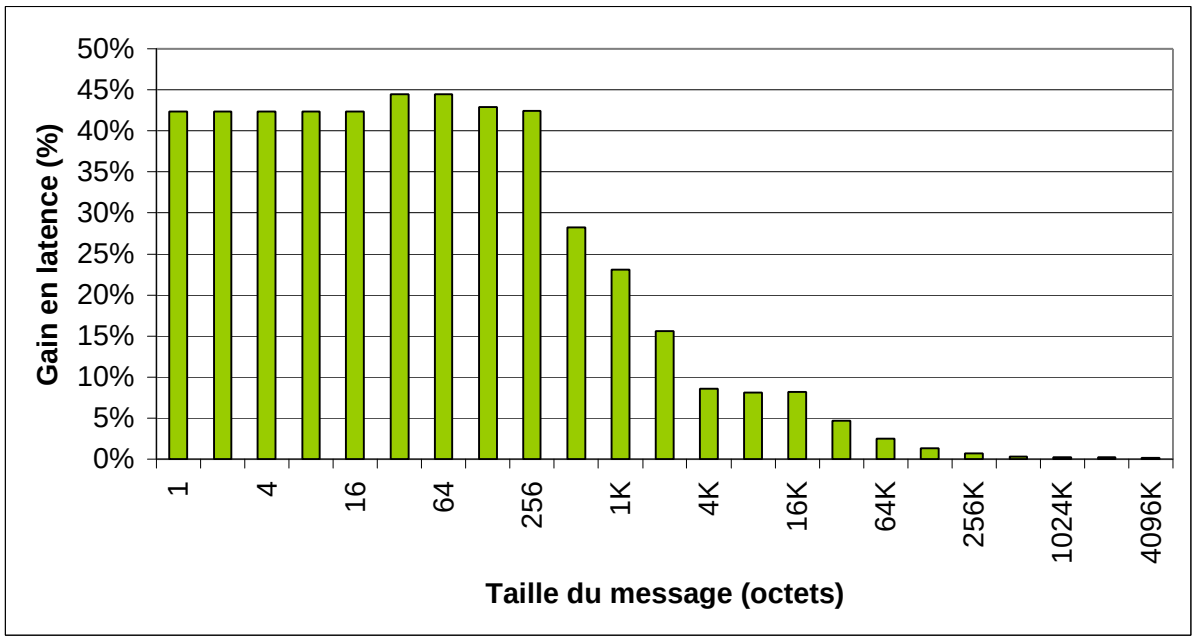


Figure V-13 : Gain de MPI-MPC2 / MPI-MPC1

L'utilisation d'une primitive d'écriture distante en mode utilisateur et sans interruption est très efficace pour les petits messages (gain > 40%). En revanche, le gain mesuré pour les gros messages est faible car la différence entre les temps de transfert de MPI-MPC1 et MPI-MPC2 est négligeable comparativement au temps de transfert total.

V.6. Conclusion

Nous avons décrit dans ce chapitre notre deuxième implémentation (MPI-MPC2) du standard MPI au-dessus d'une primitive d'écriture en mémoire distante. Nous avons montré comment passer d'une primitive d'écriture distante se trouvant dans le noyau et utilisant une signalisation par interruption à une primitive d'écriture distante en mode utilisateur reposant sur une signalisation par scrutation. Cela nous a permis d'éliminer les appels systèmes des phases de communication, à l'exception des appels système inhérents aux traductions d'adresse pour les gros messages.

Nous avons montré comment réaliser un partage efficace des ressources réseau entre différents processus. Nous avons par ailleurs décrit un mécanisme de signalisation par scrutation qui s'est avéré très efficace sur notre plate-forme expérimentale à condition de ralentir la boucle de scrutation (lorsque celle-ci est active) pour ne pas surcharger le bus PCI et risquer de ralentir les transferts DMA du contrôleur réseau.

Un autre objectif atteint dans ce chapitre a été de montrer que l'API RDMA que nous avions définie au chapitre IV s'adapte très bien aux caractéristiques de cette nouvelle primitive d'écriture distante et ainsi, de mettre en valeur le caractère générique de cette interface entre notre implémentation de MPI et n'importe quelle primitive d'écriture en mémoire distante.

Les résultats que nous avons obtenus montrent l'importance de l'élimination des appels système et des interruptions dans les phases de communication pour les performances de l'application. L'amélioration des performances de MPI-MPC2 par rapport à MPI-MPC1 est significative, en particulier pour la latence qui passe de 26µs à 15µs. [Glück,2002] présente une comparaison entre MPI-MPC1 et MPI-MPC2.

Le dernier objectif que nous avons atteint est un gain en portabilité avec MPI-MPC2 : la seule partie de nos couches de communication restant dépendante du système d'exploitation concerne la phase d'initialisation des composants matériels.

Enfin, nous avons validé de manière expérimentale sur notre plate-forme le partage des ressources réseau dans le cas où plusieurs processus communicants s'exécutent sur un même nœud de calcul. La méthode a consisté à lancer simultanément quatre processus (deux par nœud de calcul) réalisant deux ping-pong en parallèle, et utilisant la primitive d'écriture distante en mode utilisateur.

Chapitre VI : Optimisation des traductions d'adresses par redistribution de la mémoire

Sommaire

L'objectif général de ce chapitre est de proposer une méthode permettant d'éviter les opérations de traduction d'adresses pendant les phases de communication.

VI.1. Introduction

En utilisant une primitive d'écriture distante en mode utilisateur et en éliminant les interruptions matérielles dans MPI-MPC2, nous avons obtenu un gain significatif en terme de latence. Cependant, le débit de MPI-MPC2 (421Mbits/s) reste encore inférieur au débit utile maximum pour transmettre des tampons contigus en mémoire physique en utilisant la primitive d'écriture distante de notre plate-forme matérielle. Ce dernier a été mesuré à 494Mbits/s par Alexandre Fenÿo [Fenÿo,2001]. Nous expliquons cette différence par les raisons suivantes :

- ✓ le transfert des gros messages utilise le protocole de rendez-vous et nécessite pour chaque message *DATA* transféré quatre appels système coûteux : deux pour le verrouillage des tampons d'émission/réception de l'application en mémoire physique et deux pour obtenir la projection en mémoire physique de ces tampons,
- ✓ la taille du message *RSP* du protocole de rendez-vous, contenant la description en mémoire physique du tampon de réception, est de plus en plus importante lorsque la taille des données envoyées augmente, ce qui est pénalisant en terme de débit,
- ✓ le nombre d'écritures distantes pour transmettre un message *DATA* peut s'avérer élevé pour des gros messages.

Le problème n'est pas tant le coût des verrouillages/traductions des tampons d'émission/réception mais plutôt la discontinuité de ces tampons en mémoire physique qui pénalise les performances.

Les objectifs de ce chapitre sont :

- ✓ éliminer les appels système inhérents aux traductions d'adresse en proposant une méthode pour qu'un tampon contigu en mémoire virtuelle reste contigu en mémoire physique,
- ✓ faire que cette méthode soit transparente pour les applications : cela signifie qu'aucune modification ne doit être apportée au code des applications, et qu'on s'interdit l'ajout de primitives qui seraient spécifiques à notre architecture afin de rester compatible avec le standard MPI,
- ✓ n'apporter aucune modification, ni au système d'exploitation, ni à la librairie C, pour des raisons de portabilité.

Ce chapitre est composé de quatre parties. Nous analysons les différentes solutions envisageables pour minimiser l'impact des traductions d'adresse sur les performances des applications. Nous expliquons le principe de notre solution à ce problème. Nous décrivons notre implémentation. Enfin, nous analysons les performances obtenues sur notre plate-forme expérimentale.

VI.2. Les diverses solutions envisagées

Le problème de la discontinuité en mémoire physique des tampons de l'application est récurrent dans les réseaux utilisant une primitive d'écriture distante (*Remote DMA*).

Une première solution est d'utiliser une recopie intermédiaire des données, en émission et en réception, dans des tampons contigus en mémoire physique. C'est la méthode que nous utilisons pour la transmission des messages *CTRL* mais les recopies intermédiaires deviennent beaucoup trop coûteuses dès que la taille des données est supérieure à quelques kilo-octets. Il n'est donc pas envisageable d'utiliser les recopies pour les gros messages.

Une deuxième solution serait d'utiliser l'allocateur de mémoire physique contiguë dont nous avons fait mention à la section IV.5.1. Elle consiste à mettre à la disposition des applications une nouvelle primitive ajoutée à celles du standard MPI. Elle permettrait aux applications de s'allouer un tampon contigu en mémoire physique et de le projeter dans la mémoire virtuelle de l'application. Il s'agirait d'un `malloc` en mémoire physique. Le problème de cette solution est qu'elle ne remplit pas l'objectif que nous nous sommes fixés de n'apporter aucune modification à l'application et de rester dans le standard MPI.

Une solution évitant de modifier les applications existantes consiste à intercepter la fonction `malloc` de la librairie C pour la remplacer par une primitive allouant un tampon contigu en mémoire physique. Le problème de cette solution est que les données échangées par les applications MPI ne se trouvent pas forcément dans le tas, c'est-à-dire dans un tampon alloué par un `malloc`. Elles peuvent se trouver dans la pile ou dans le segment de données du processus. Cette solution ne répond donc que partiellement au problème posé.

Une quatrième solution est d'utiliser un cache des traductions d'adresse. Le principe est de conserver dans une table, pour chaque tâche de l'application, la correspondance entre une zone contiguë en mémoire virtuelle et sa description en mémoire physique. Le verrouillage et la traduction de la zone sont faits lors du premier transfert des données. Cette solution s'avère assez efficace dans la mesure où ce sont très souvent les données d'un même tampon qui sont échangées dans une application parallèle. Cette méthode suppose d'intercepter la fonction `free` de la librairie C : si un processus fait un `malloc`, un `free`, puis de nouveau un `malloc`, il est possible que l'adresse virtuelle retournée par le deuxième `malloc` soit identique à celle du premier alors que les deux tampons ne correspondent pas à la même description en mémoire physique, et il faut donc invalider l'entrée correspondante du cache. Le protocole bas niveau BIP utilise cette solution. Les protocoles PM, VMMC et U-Net pour le réseau Myrinet utilisent également un cache, qui plus est accessible par le contrôleur réseau, mais l'interception des fonctions de gestion de la mémoire de la librairie C n'est pas mentionnée par les auteurs. Les auteurs de [Seifert_2,2001] présentent, dans le cadre d'une implémentation de VIA, une technique similaire basée sur leur gestionnaire de

mémoire verrouillée : « *Locked Memory Manager* (LMM) ». Des informations supplémentaires sur ces différents protocoles sont présentées au chapitre III. Nous avons implanté cette méthode dans notre portage de MPI sur l'API RDMA mais les performances ne sont pas très satisfaisantes pour plusieurs raisons : (1) la gestion d'un cache est coûteuse ; (2) cette solution n'empêche pas que la taille du message *RSP* dans le protocole de rendez-vous reste élevée et que plusieurs écritures en mémoire distante sont nécessaires pour transférer un message *DATA* ; (3) contrairement au LANai, le contrôleur réseau de la machine MPC n'est pas capable d'aller lire le cache en mémoire centrale pour établir la correspondance entre les adresses virtuelles et les adresses physiques ; (4) une étude autour de U-Net/MM a montré que, lors de l'utilisation d'un tel cache avec des applications réelles, beaucoup de communications génèrent un défaut de cache (plus de 40% pour certaines applications) [Welsh,1997].

VI.3. Principe de la solution retenue

La solution que nous proposons est d'allouer, statiquement, la mémoire physique disponible sur chaque nœud de calcul aux processus constituant l'application, lors de son démarrage. Cette idée part d'un double constat : d'une part, les nœuds de calcul d'une machine parallèle de type « grappe de PCs » disposent actuellement de beaucoup de mémoire physique (de l'ordre d'1 Go) à un faible coût et, d'autre part notre machine parallèle est entièrement dédiée à l'exécution d'une seule application MPI à laquelle nous fournissons l'ensemble des ressources disponibles. Notre but est de nous ramener à une situation où toutes les données de l'application se retrouvent dans une zone contiguë en mémoire physique.

La mémoire virtuelle de n'importe quel processus s'exécutant sur un système UNIX standard tel que LINUX se présente toujours sous la même forme. La Figure VI-1 représente l'espace d'adressage d'un processus, c'est-à-dire l'ensemble des zones de mémoire virtuelle allouées à celui-ci.

Sur l'architecture x86 que nous utilisons pour nos expérimentations, l'espace adressable est de quatre giga-octets. Sur les systèmes LINUX, un giga-octet est réservé à la mémoire utilisée par le système d'exploitation. Les trois giga-octets restants sont décomposés en régions mémoire utilisables par le processus :

- ✓ le code du processus : ce segment contient le code exécutable (qui inclut le code de toutes les librairies statiques utilisées par le processus),
- ✓ les données du processus : elles se décomposent en deux parties, d'une part *data* qui contient les variables initialisées, et d'autre part *bss* qui contient les variables non initialisées,
- ✓ le tas : il permet à un processus de s'allouer dynamiquement, lors de son exécution, de nouvelles zones de mémoire virtuelle,
- ✓ la pile utilisée par le processus.

La mémoire d'un processus est décomposée en trois segments : le segment de code, le segment de données (*data*+*bss*+tas), et le segment de pile. Lorsqu'un processus commence son exécution, ses segments possèdent une taille fixe. Il existe toutefois des fonctions d'allocation/désallocation de mémoire, qui permettent à un processus de manipuler des variables dont le nombre ou la taille ne sont pas connus au moment de la compilation. La taille du tas et de la pile évolue dynamiquement lors de l'exécution du processus, selon ses propres besoins. Le tas évolue suivant les adresses croissantes et la pile suivant les adresses décroissantes.

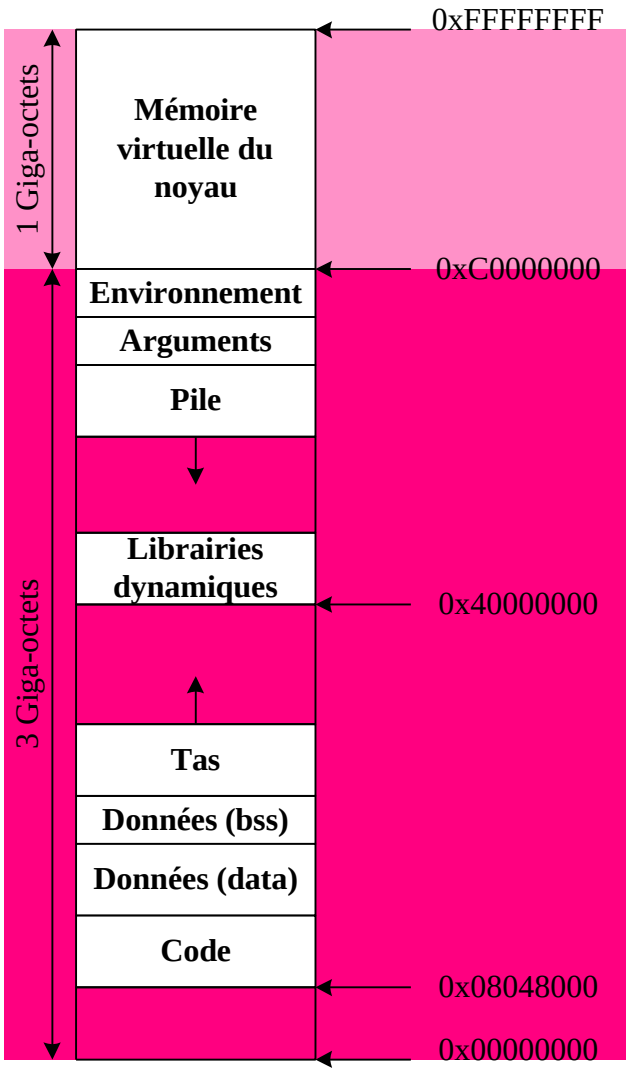


Figure VI-1 : Espace d'adressage d'un processus

La zone « Environnement » contient les variables d'environnement du processus et la zone « Arguments » les arguments de la ligne de commande qui a provoqué l'exécution du processus. Ces deux zones se trouvent dans le segment de pile.

Les librairies dynamiques liées au processus sont chargées en mémoire entre le tas et la pile. En particulier, une librairie dynamique, s'exécutant avant le processus, s'occupe du chargement de celui-ci en mémoire et de la création de toutes les zones

mémoire présentées sur la Figure VI-1. La valeur des adresses est spécifique au système LINUX.

Nous proposons d'intervenir entre le chargement du processus en mémoire et le début d'exécution de celui-ci, c'est-à-dire avant l'appel de la fonction *main* du langage C. Les données échangées dans une application MPI peuvent se trouver dans le segment de données, dans le segment de pile et plus rarement dans le segment de code. Nous souhaitons que ces trois segments correspondent à des zones contiguës en mémoire physique. Il s'agit de faire une « redistribution de la mémoire » du processus en mémoire physique avant le début de son exécution.

La méthode se décompose en quatre phases (représentées sur la Figure VI-2) :

- ✓ sauvegarde des trois segments du processus sur le disque local du nœud de calcul,
- ✓ attribution de deux zones de mémoire physique contiguë, une pour le segment de pile et une pour les segments de code et de données,
- ✓ projection de ces zones dans la mémoire virtuelle du processus.
- ✓ recopie des trois segments dans les deux zones de mémoire physique contiguë,

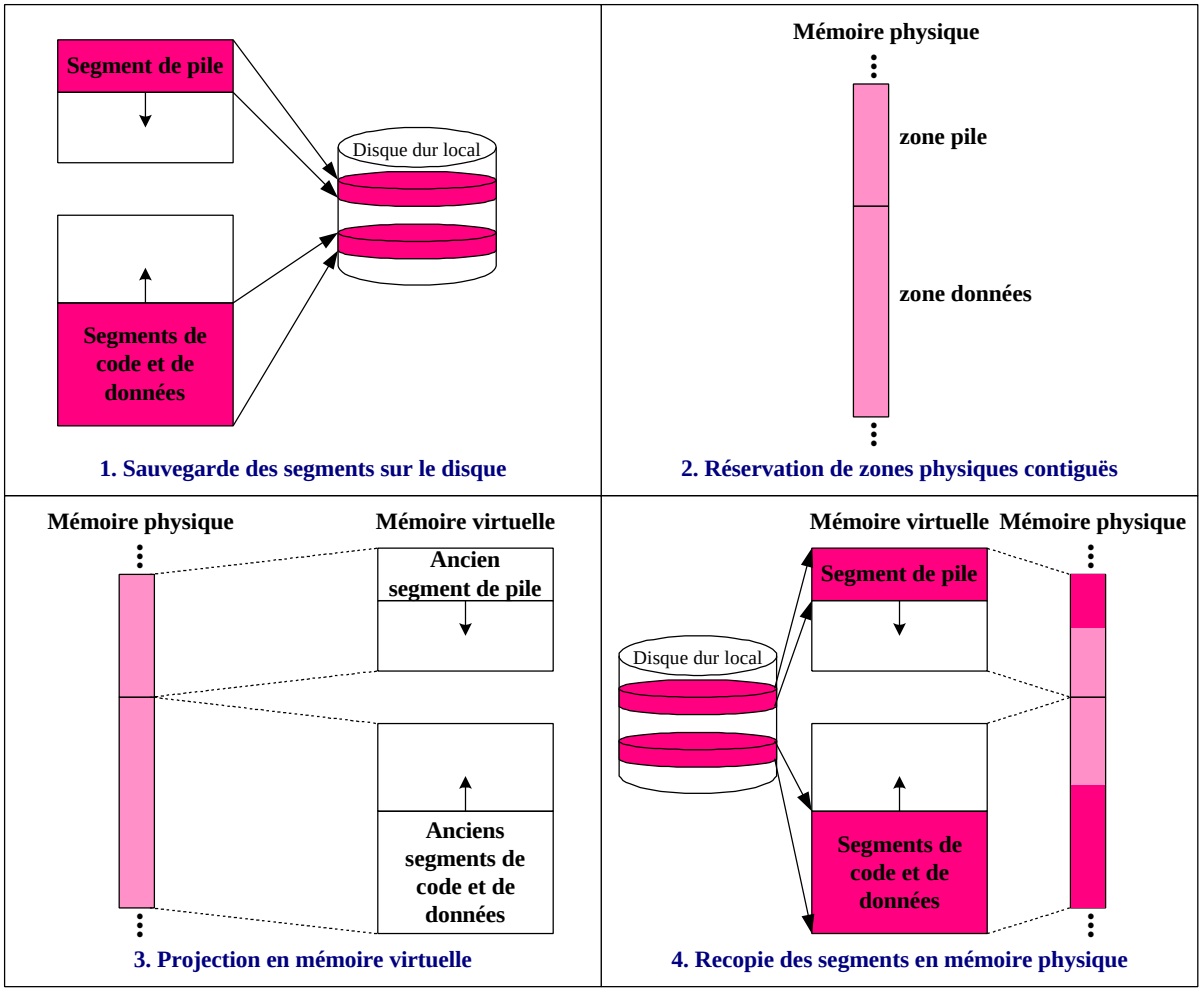


Figure VI-2 : Les quatre étapes de la redistribution de la mémoire

La taille des segments représentés sur la figure est celle fixée juste après le chargement du processus en mémoire. Elle sera amenée à augmenter pendant la durée de vie du processus. C'est pourquoi la taille des zones de mémoire physique contiguë réservées est plus grande que la taille des segments à l'origine.

L'allocateur de mémoire physique se réserve, lors du démarrage du nœud de calcul, la plus grande zone de mémoire physique contiguë possible en laissant le strict nécessaire au système d'exploitation et aux démons s'exécutant sur la machine. Notre politique est de partager, de façon statique, toute la mémoire physique restante entre les processus de l'application s'exécutant sur le nœud de calcul. On réalise ce partage de manière équitable dans la mesure où tous les processus d'une application MPI sont identiques. Par exemple, s'il y a 100Mo de mémoire physique disponible sur un nœud de calcul sur lequel deux processus de l'application vont s'exécuter, chaque processus disposera de 50Mo de mémoire physique contiguë. L'utilisateur décide lors du lancement de l'application comment les 50Mo doivent être répartis entre la zone « pile » et la zone « données », suivant que son application a besoin de beaucoup de mémoire dans le tas ou dans la pile. Il s'agit donc d'une allocation statique de la mémoire physique aux processus de l'application lors de son démarrage.

La principale difficulté concerne la phase de projection en mémoire virtuelle. La zone « données » doit être projetée à l'adresse 0x08048000 en mémoire virtuelle (cf. Figure VI-1) : c'est l'adresse de début du segment de code sous LINUX (cette valeur est la même pour tous les processus). La fonction `mmap` de la librairie C permet une projection à une adresse fixe. Dans le cas présent, le problème est que le processus doit réécrire par dessus son segment de code. Pour employer une métaphore, cela revient à couper la branche sur laquelle on est assis. Pour cette raison, le code qui réalise les opérations de la Figure VI-2 ne peut se trouver dans le segment de code du processus. Une solution à ce problème est de réaliser ces opérations dans une librairie dynamique qui aura ses propres segment de code et segment de données dans la zone « Librairies dynamiques » de la Figure VI-1. En revanche, cette librairie dynamique utilise la pile du processus pour ses propres besoins. Il n'est donc pas possible de projeter la zone « pile » qui contiendra la nouvelle pile du processus par dessus l'ancienne pile. Notre solution est de projeter la nouvelle pile quelque part entre le tas et l'ancienne pile et de déplacer le pointeur de pile de l'ancienne vers la nouvelle. Cette opération ne peut être réalisée qu'en langage assembleur.

Comme nous venons de le voir, toutes ces opérations doivent être réalisées dans une librairie dynamique à laquelle le processus devra se lier lors de la compilation. Il nous reste à voir comment faire exécuter le code de cette librairie avant l'exécution du processus. L'idée est très simple et repose sur les propriétés du langage C++ qui permet l'appel d'une fonction (ou méthode) appelée « constructeur » dès lors qu'une instance d'une classe objet est déclarée.

Avec la redistribution de la mémoire, il n'y a plus besoin de verrouiller les tampons de communication en mémoire physique : les zones « données » et « pile » ne peuvent pas être évincées de la mémoire physique par le système d'exploitation. Les traductions d'adresse sont simplifiées à l'extrême : il suffit de conserver au niveau de chaque processus de l'application et pour chacune des deux zones de mémoire physique contiguë un triplet (adresse virtuelle, adresse physique, longueur). Lorsqu'un processus veut obtenir l'adresse physique correspondant à une adresse virtuelle, il lui suffit de trouver la zone correspondante, de récupérer l'adresse physique du début de la zone et de soustraire le décalage à l'adresse virtuelle considérée. Les appels système liés aux opérations DMA sont supprimés.

La section suivante présente de manière plus détaillée notre implémentation de la redistribution de la mémoire.

VI.4. Implémentation de la redistribution de la mémoire

Nous expliquons dans cette section comment nous avons implanté la redistribution de la mémoire des processus d'une application MPI.

VI.4.1. La librairie dynamique

Toutes les opérations de la redistribution de la mémoire que nous détaillons dans la suite de cette section sont exécutées dans une librairie dynamique afin de travailler dans un segment de code distinct de celui du processus. Cette libraire dynamique est écrite dans le langage C++ pour utiliser une propriété très forte de ce langage : l'appel d'un constructeur de manière transparente avant l'exécution de la fonction `main` du processus. La Figure VI-3 présente un exemple simplifié d'utilisation de la librairie dynamique.

Redistrib.hh

```
class Redistrib
{
  Redistrib();
  ~Redistrib();
  ...
}
```

Redistrib.cc

```
#include "Redistrib.hh"
// constructeur
Redistrib::Redistrib()
{
  //opérations à réaliser
  //avant le main
  ...
}
// destructeur
Redistrib::~Redistrib()
{
  ...
}
```

Librairie dynamique (libRedistrib.so)

Monappli.c ou **Monappli.cc**

```
main()
{

  ...

}
```

mpc.cc

```
#include "Redistrib.hh"

Redistrib MPC;
```

Application MPI

Figure VI-3 : Un exemple d'utilisation de la librairie dynamique

La librairie dynamique est ici constituée de deux fichiers (*Redistrib.hh* et *Redistrib.cc*). Nous définissons en C++ une classe objet appelée `Redistrib` ainsi que ses deux méthodes : le constructeur `Redistrib()` et le destructeur `~Redistrib()`. Il suffit, lors de la compilation, de lier l'application MPI à la librairie dynamique (*libRedistrib.so*) et au fichier objet *mpc.o* obtenu par la compilation du fichier *mpc.cc* pour que le constructeur de la classe `Redistrib` soit appelé avant l'exécution de la fonction `main` de l'application. Le fait de déclarer une instance de la classe `Redistrib` (appelée `MPC`) dans le fichier *mpc.cc* provoquera l'appel du constructeur de cette classe lors du chargement du processus en mémoire. Le code des opérations que nous voulons réaliser avant l'exécution de l'application se trouve dans le constructeur de la classe. Le destructeur sera appelé après la terminaison de la fonction `main`.

Nous utilisons une librairie dynamique écrite en C++ pour réaliser les opérations nécessaires à la redistribution de la mémoire avant le commencement de l'exécution des processus de l'application MPI. Cela nous permet non seulement d'exécuter du code avant l'appel de la fonction `main` du processus mais aussi d'exécuter ce code en dehors du segment de code du processus. Il suffit pour cela de modifier l'édition des liens lors de la compilation de l'application MPI.

VI.4.2. Le partage de la mémoire physique

Nous partageons toute la mémoire physique contiguë disponible sur un nœud de calcul entre tous les processus de l'application qui s'exécutent sur ce nœud. Le standard MPI-1 [MPI,1994] ne permettant pas la création dynamique de nouvelle tâche MPI, nous pouvons réaliser ce partage de la mémoire physique de façon statique lors du démarrage de l'application. Par ailleurs, le standard MPI-1 ne donne aucune spécification concernant le lancement des applications. En ce qui concerne l'implémentation de MPICH que nous utilisons, le lancement de l'application se fait en utilisant la commande `mpirun`. Dans notre implémentation, il s'agit d'un script écrit en *shell* UNIX qui lance le nombre de tâches demandées par l'utilisateur de manière équitable sur les différents nœuds de calcul de la machine parallèle. Nous avons rajouté un paramètre (`stack_size`) au script `mpirun` donnant la possibilité à l'utilisateur de spécifier le pourcentage de mémoire physique attribué à la zone « pile », le reste étant réservé à la zone « données ».

Voici un exemple de lancement de l'application MPI `Monappli` :

```
$ mpirun –np 10 –stack_size 50 Monappli
```

Cette commande lance une application MPI composée de 10 tâches identiques `Monappli`. On commence par calculer le nombre de tâches par nœud en fonction du nombre de nœuds disponibles. De plus, la commande indique que, pour chaque tâche, la taille de la zone « pile » doit être égale à celle de la zone « données ». Notre librairie

dynamique récupère ces informations pour en déduire le nombre de tâches lancées sur le nœud local et la taille des zones « pile » et « données » attribuées à chaque processus.

La taille maximale du segment de pile et du segment de données de chaque processus est fixée lors du démarrage de l'application. Elle dépend de la mémoire physique disponible sur la machine et du nombre de processus choisi par l'utilisateur. La répartition de la mémoire physique entre la zone « pile » et la zone « données » est choisi de façon statique par l'utilisateur lors du lancement de l'application.

VI.4.3. Le tas

La taille du segment de données d'un processus est amenée à augmenter ou diminuer durant son exécution. Celui-ci peut utiliser des fonctions d'allocation/désallocation de zones de mémoire virtuelle pour stocker ses données. Les fonctions `malloc` et `free` de la librairie C en sont le meilleur exemple. La Figure VI-4 représente le segment de code et le segment de données du processus. Le tas d'un processus a une limite supérieure qui est fixée par un pointeur appelé *brk*. Le pointeur *start_brk* est l'adresse de début du tas. Le tas est décomposé en deux parties sur cette figure : une zone occupée qui contient des données utilisées par le processus et une zone libre.

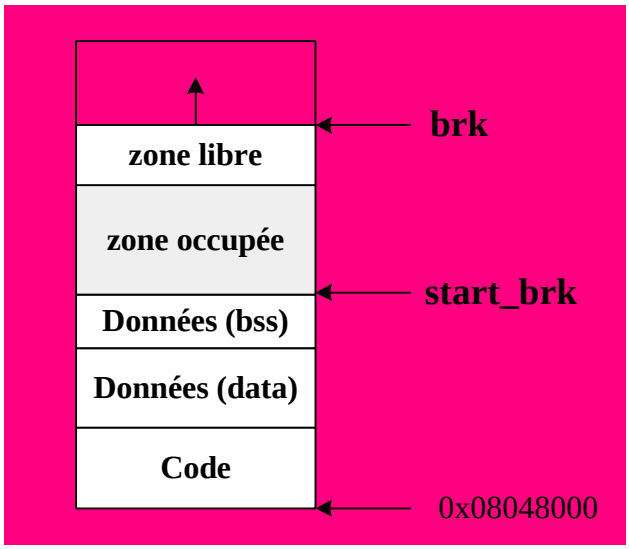


Figure VI-4 : Le segment de données d'un processus

Lorsqu'un processus fait appel à la fonction `malloc` de la librairie C, il fournit en paramètre la taille du tampon qu'il souhaite obtenir et l'adresse de début de ce tampon lui est retournée. Il y a alors deux possibilités : soit il y a suffisamment d'espace dans la zone libre pour lui fournir un tampon de la taille demandée, soit de nouvelles pages de mémoire virtuelle doivent être allouées au processus et le pointeur *brk* est augmenté en conséquence. L'allocation de nouvelles pages est réalisée en appelant la fonction

`do_mmap` du noyau qui est elle-même appelée lorsqu'un processus fait appel à la primitive `mmap` de la librairie C. Des informations plus précises concernant la gestion du tas sous LINUX peuvent être trouvées dans [Bovet,2001].

Dans notre cas, il ne faut pas que de nouvelles pages soient allouées de cette façon durant l'exécution de l'application car cette allocation entrerait en conflit avec la projection dans la mémoire virtuelle du processus de la zone « données » se trouvant en mémoire physique (cf. Figure VI-2). Notre solution est d'augmenter la taille du tas jusqu'à ce qu'il atteigne la taille maximale que nous lui avons réservée en mémoire physique. Cette opération est réalisée avant la phase 1 de la Figure VI-2. Pour monter *brk* à sa valeur maximum, il suffit de faire un « gros » `malloc` suivi d'un « petit » `malloc` puis un `free` du « gros » `malloc` pour libérer toute la zone de mémoire virtuelle se trouvant entre le début du tas et la zone réservée pour le « petit » `malloc`. Ainsi, lorsque l'application demandera dynamiquement de la mémoire virtuelle par des appels à la primitive `malloc`, celle-ci sera prise dans la zone libérée. Le fait de ne pas libérer le « petit » `malloc` nous assure que le pointeur *brk* restera à son maximum. En revanche, si l'application demande plus de mémoire que celle disponible dans la zone libre, *brk* sera augmenté et le tas du processus sortira de la zone de mémoire physique que nous lui avons attribué. Si un tel cas de figure se produit, nous le détectons au moment de la traduction d'adresse et nous arrêtons l'application. C'est le prix à payer pour le contrôle de la mémoire physique. Cela peut paraître brutal mais dans cette situation, toute la mémoire physique disponible sur le nœud de calcul serait saturée et les performances de l'application deviendraient très médiocres. La Figure VI-5 représente l'état dans lequel se trouve le tas après la redistribution de la mémoire, c'est-à-dire avant le début de l'exécution du processus.

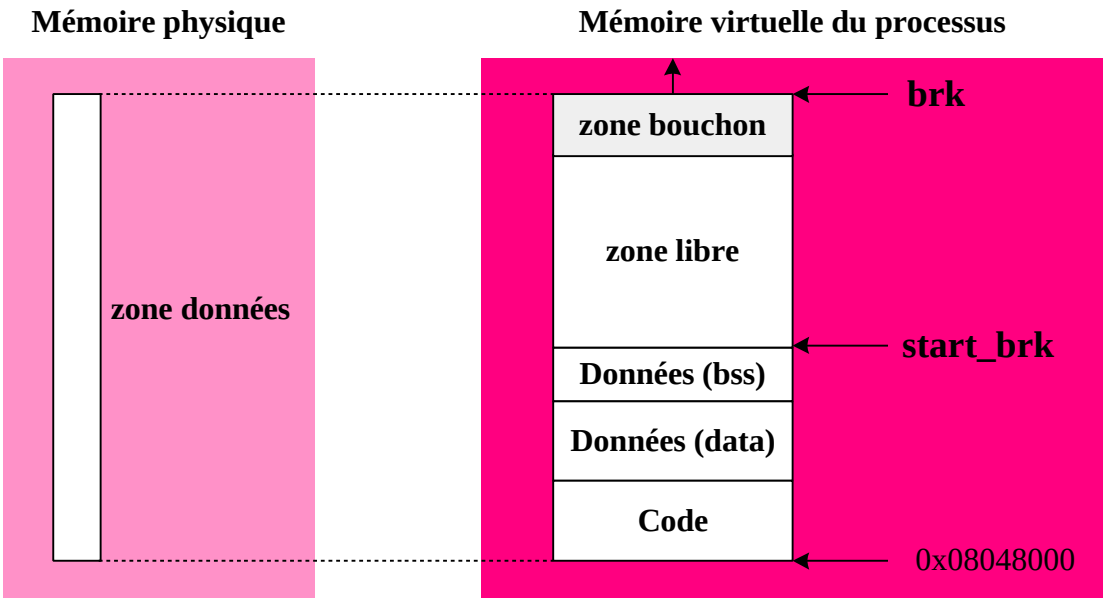


Figure VI-5 : L'état du tas après la redistribution de la mémoire

La zone « données » est la zone de mémoire physique réservée pour les segments de code et de données du processus. La zone « bouchon » sur la figure correspond à l'espace mémoire réservé attribué au « petit » `malloc` afin d'empêcher la taille du tas de diminuer. La zone « libre » est l'espace disponible pour le processus pour ses allocations dynamiques de mémoire virtuelle. Le dessin ci-dessus n'est pas du tout à l'échelle dans la mesure où sur une machine disposant de beaucoup de mémoire physique, la zone « données » peut faire plusieurs centaines de Mega-octets.

VI.4.4. La pile

Le cas de la pile est tout aussi délicat que celui du tas. Comme nous l'avons mentionné à la section VI.3, la zone de mémoire physique réservée pour la pile du processus ne peut pas être projetée dans la mémoire du processus à l'emplacement de l'ancienne pile car celle-ci est utilisée par notre librairie dynamique. Nous sommes obligés de projeter la nouvelle pile à un autre endroit dans la mémoire virtuelle, comme cela est représenté sur la Figure VI-6.

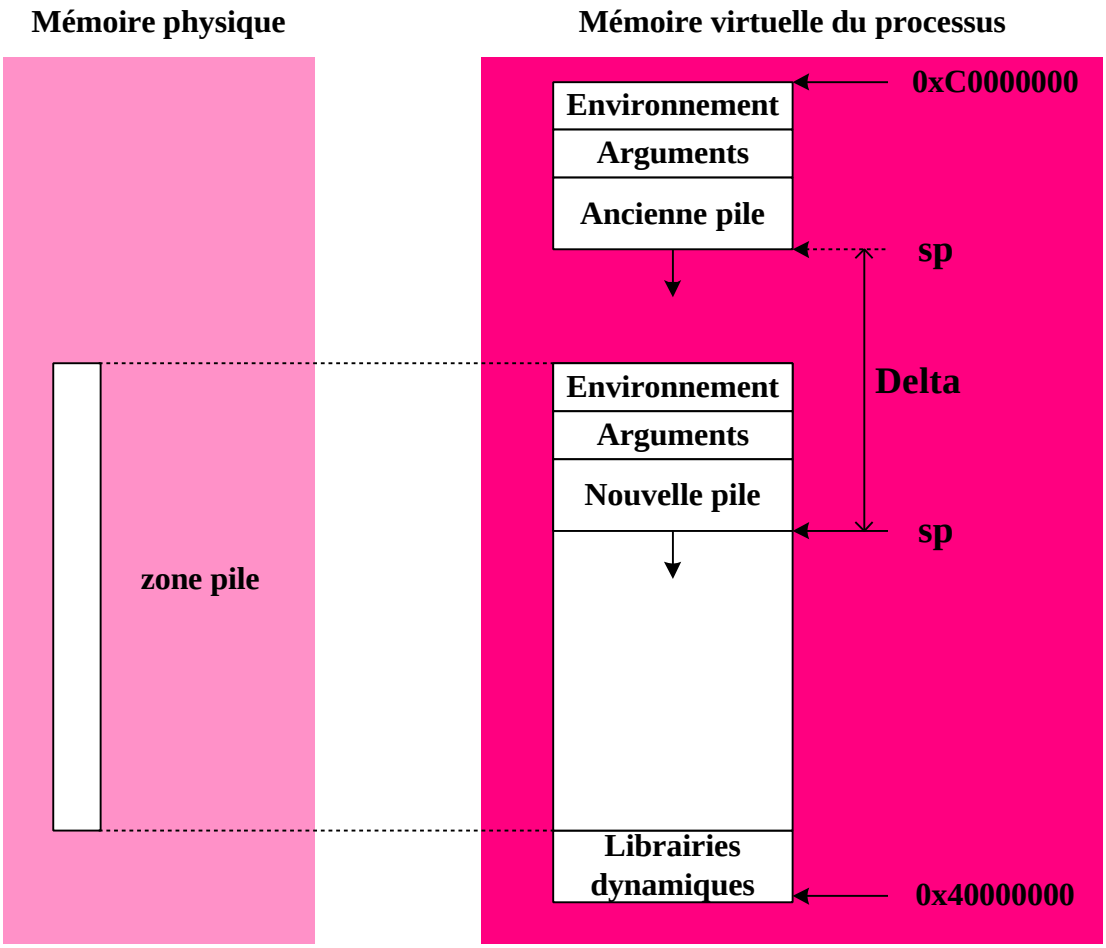


Figure VI-6 : L'ancienne et la nouvelle pile

L'ancienne pile est celle utilisée par notre librairie dynamique. Le contenu de cette ancienne pile est recopié dans la zone de mémoire physique que nous avons réservée pour la nouvelle pile. Avant de quitter le constructeur de la librairie dynamique pour aller exécuter le `main` du processus, nous déplaçons le pointeur de pile *sp* pour qu'il pointe sur la nouvelle pile. La pile est translatée d'une valeur *Delta* comme cela est représenté sur la figure ci-dessus.

Lorsque le constructeur de notre librairie dynamique est appelé par le chargeur du processus en mémoire, une sauvegarde du contexte est faite et une adresse de retour est empilée, dans l'ancienne pile. La phase de sauvegarde du contexte permet en particulier de sauvegarder dans la pile la valeur du pointeur *bp* (*Base Pointer*). Ce pointeur permet d'accéder aux paramètres et aux variables locales de la fonction appelée qui ont été empilés. Notre problème est que, si nous laissons le constructeur se terminer normalement, l'exécution du `main` va se faire dans l'ancienne pile et donc la nouvelle pile contiguë en mémoire physique ne sera pas utilisée. En effet, si le constructeur se termine normalement, le contexte avant son appel est restauré et *sp* est ramené dans l'ancienne pile. Notre solution à ce problème est d'appeler nous-mêmes la fonction `main` du processus à la fin de l'exécution du constructeur. Ainsi, le `main` sera exécuté dans la nouvelle pile. Quand le `main` se termine, l'exécution du constructeur reprend son cours. Nous restaurons alors l'ancien contexte : le pointeur *sp* est ramené dans l'ancienne pile et l'instruction assembleur `ret` nous permet de sortir proprement du constructeur. Ensuite, le destructeur est appelé normalement et l'application se termine correctement.

L'appel de la fonction `main` est fait en utilisant l'instruction assembleur `call`. Nous devons empiler, dans la nouvelle pile, les paramètres du `main` avant d'exécuter cette instruction. L'instruction `call` empile l'adresse de retour qui sera utilisée lors de la terminaison du main afin de revenir dans le constructeur. Nous rappelons que le `main` a deux paramètres : `argc`, le nombre d'arguments de la ligne de commande et `argv`, l'adresse du tableau de pointeurs sur les chaînes de caractères représentant les arguments. La zone « Arguments » de la Figure VI-6 contient les arguments de la ligne de commande qui sont des chaînes de caractères. Les adresses de chacun des arguments sont empilées dans la pile par le chargeur du processus comme cela est représenté sur la Figure VI-7.

Dans cet exemple, la ligne de commande a deux arguments qui sont stockés dans la zone « Arguments ». L'adresse du premier argument (`argv[0]`) et celle du deuxième (`argv[1]`) ont été empilées lors du chargement du processus en mémoire. Comme nous avons translaté la pile, nous avons également translaté la zone des arguments. Il faut donc modifier les valeurs de `argv[0]` et `argv[1]` en conséquence dans la nouvelle pile afin qu'ils pointent sur la zone « Arguments » de la nouvelle pile. L'opération consiste à leur retrancher la valeur de *Delta*.

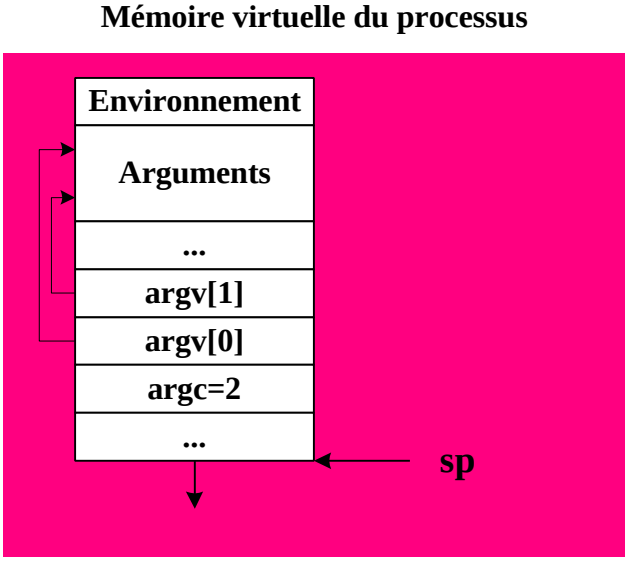


Figure VI-7 : Les arguments de la ligne de commande et la pile

Avant d'exécuter l'instruction `call` pour exécuter le `main`, nous devons empiler dans la nouvelle pile, `argc` et `argv` qui est l'adresse de `argv[0]`. La Figure VI-8 représente la structure de la nouvelle pile juste avant l'exécution du `call`. L'adresse de `argv[0]` est notée `&argv[0]`.

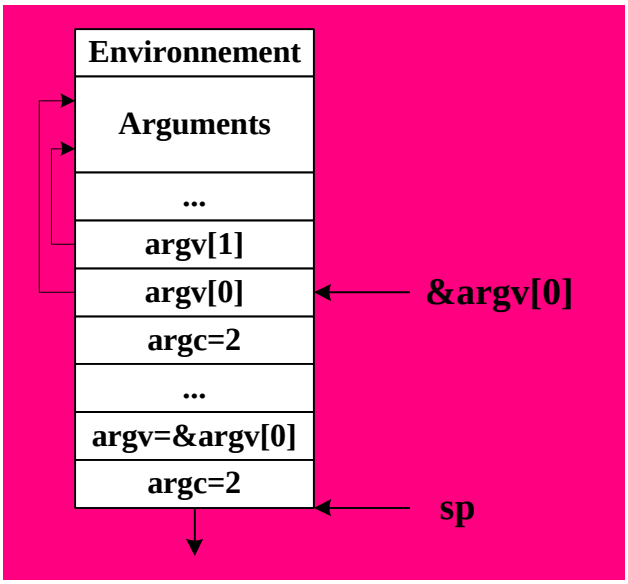


Figure VI-8 : La nouvelle pile avant l'exécution du main

Lors de l'exécution de l'application, les paramètres et les variables locales des fonctions appelées seront empilés dans la nouvelle pile. Le système n'aura pas besoin d'allouer de nouvelles pages de mémoire pour la pile tant que le pointeur *sp* reste dans la zone « pile » que nous avons allouée. S'il sort de cette zone, cela provoque une exception de type « *Segmentation Fault* » et l'application se termine.

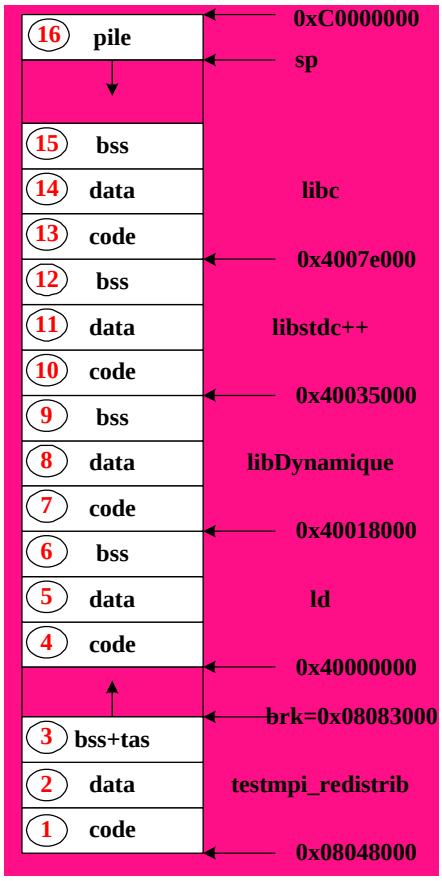
VI.4.5. Récapitulation des différentes étapes

Dans cette section, nous récapitulons les différentes étapes de la redistribution de la mémoire du processus en mémoire physique qui sont réalisées dans le constructeur de la librairie dynamique :

- ✓ récupération des informations suivantes : quantité de mémoire physique contiguë disponible, nombre de tâches locales de l'application, pourcentage de cette mémoire allouée pour la pile,
- ✓ calcul de la taille de la zone « pile » et de la zone « données »,
- ✓ augmentation de la taille du tas pour que le cumul des tailles du segment de données et du segment de code soit égal à la taille de la zone « données » (cf. Figure VI-5),
- ✓ sauvegarde des segments du processus sur disque,
- ✓ réservation de la zone « pile » et de la zone « données » en mémoire physique,
- ✓ projection des zones contiguës de mémoire physique en mémoire virtuelle (la zone « données » est projetée à l'adresse 0x08048000 à la place des segments de code et de données du processus),
- ✓ recopie des segments sauvegardés dans les zones de mémoire physique projetées dans la mémoire du processus,
- ✓ translation du pointeur de pile de l'ancienne pile vers la zone « pile » qui devient la nouvelle pile,
- ✓ modification dans la nouvelle pile des adresses des arguments de la ligne de commande,
- ✓ empilement dans la nouvelle pile des paramètres de la fonction `main` du processus,
- ✓ appel du `main` par l'instruction assembleur `call`,
- ✓ restauration du contexte avant l'appel du constructeur et appel de l'instruction assembleur `ret` pour terminer l'exécution du constructeur,
- ✓ le destructeur est appelé et l'application se termine.

VI.4.6. Un exemple d'utilisation de la redistribution de la mémoire

Nous présentons un exemple de processus MPI utilisant la redistribution de la mémoire. Une commande du système LINUX permet de visualiser les différentes zones de mémoire virtuelle d'un processus en cours d'exécution.



1 08048000-08072000 /home/olivier/testmpi_redistrib
2 08072000-08074000 /home/olivier/testmpi_redistrib
3 08074000-08083000
4 40000000-40016000 /lib/ld-2.2.so
5 40016000-40017000 /lib/ld-2.2.so
6 40017000-40018000
7 40018000-4001e000 /home/olivier/libDynamique.so
8 4001e000-40020000 /home/olivier/libDynamique.so
9 40020000-40030000
10 40035000-4006f000 /usr/lib/libstdc++-3-libc6.2-2-2.10.0.so
11 4006f000-4007b000 /usr/lib/libstdc++-3-libc6.2-2-2.10.0.so
12 4007b000-4007e000
13 4007e000-40187000 /lib/libc.so.6
14 40187000-4018d000 /lib/libc.so.6
15 4018d000-40191000
16 bffdc000-c0000000

Figure VI-9 : Avant la redistribution de la mémoire

La Figure VI-9 présente les régions mémoire du processus `testmpi_redistrib` avant de lui appliquer la redistribution de la mémoire. Les adresses sont en hexadécimal. Quatre librairies dynamiques ont été chargées : `ld`, `libstdc++`, `libc`, et `libDynamique` (notre librairie dynamique). A chaque librairie correspondent trois régions mémoire : son segment de code, son segment de données initialisées (*data*) et son segment de données non initialisées (*bss*). La troisième région mémoire du processus contient son segment de données non initialisées et le tas du processus.

La Figure VI-10 montre les régions mémoire du même processus après la redistribution de la mémoire. Les deux figures présentées dans cette section ne sont pas à l'échelle. Les zones correspondant aux librairies dynamiques n'ont pas changé.

En revanche, la pile a été translatée et la taille du tas du processus a augmenté pour atteindre la taille de la zone « données ». Les régions mémoire 1 et 14 correspondent à une zone contiguë en mémoire physique. Elles ont été attribuées par notre allocateur de mémoire physique contiguë (le pilote `/dev/cmem`). Les régions 1, 2 et 3 de la Figure VI-9 ont été remplacées par la région 1 de la Figure VI-10.

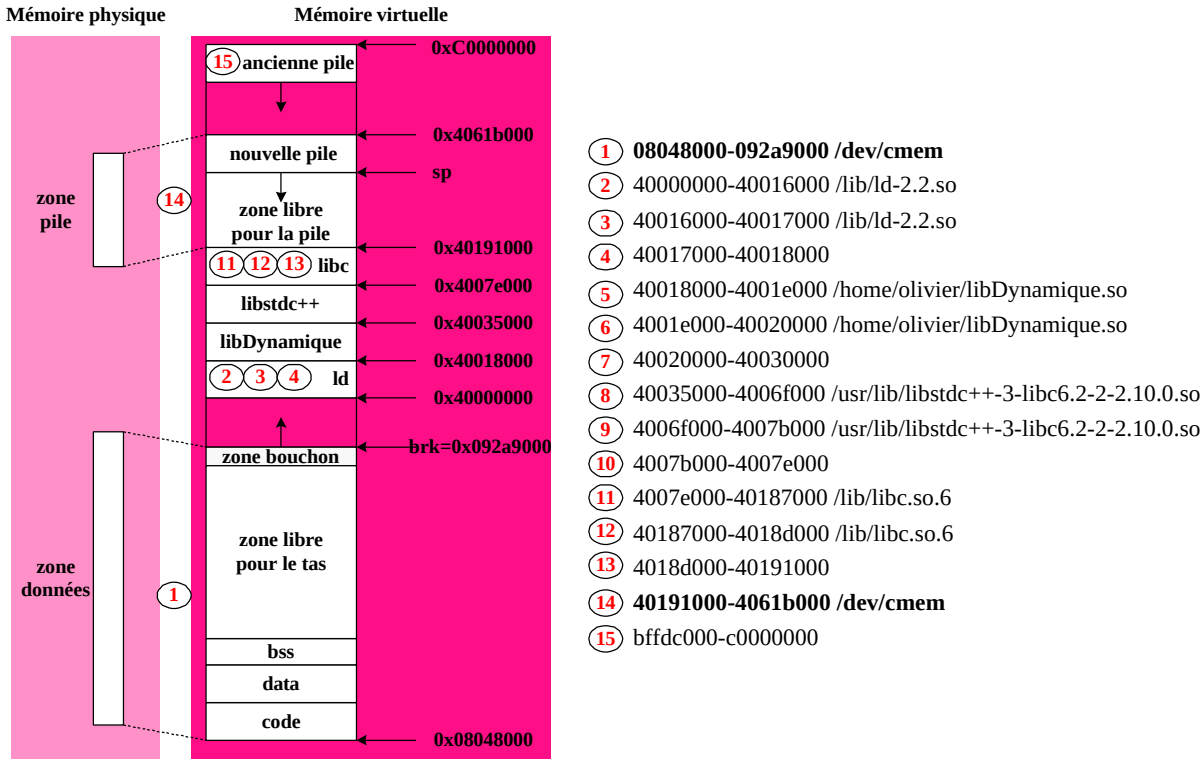


Figure VI-10 : Après la redistribution de la mémoire

Toutes les données du processus, susceptibles d'être transmises par notre librairie MPI, correspondent à un tampon contigu en mémoire physique, dès lors qu'elles sont dans un tampon contigu en mémoire virtuelle.

VI.5. Mesures de performances avec un « *ping-pong* » MPI

Cette section décrit les performances obtenues avec notre troisième implémentation de MPI : MPI-MPC3. L'unique différence entre MPI-MPC2 et MPI-MPC3 est la suppression des opérations de traduction d'adresse. Les expérimentations présentées sont analogues à celles présentées dans les chapitres IV et V.

VI.5.1. La plate-forme expérimentale

Nous utilisons la même plate-forme expérimentale que dans les chapitres IV et V. La primitive d'écriture distante est réalisée par le matériel de la machine MPC. La couche RDMA se trouve dans l'espace utilisateur et nous utilisons la signalisation par scrutation décrite au chapitre V. L'architecture bas niveau de MPI-MPC3 est identique à celle de MPI-MPC2 (cf. Figure V-8). L'unique différence provient du fait que dans MPI-MPC3, l'application MPI est liée à notre librairie dynamique implantant la redistribution de la mémoire.

VI.5.2. Le seuil optimal pour les messages de contrôle

La première expérimentation consiste à mesurer le seuil optimal pour les messages *CTRL*, c'est-à-dire déterminer à partir de quelle taille de message le protocole de rendez-vous devient plus efficace que l'encapsulation des données dans un message *CTRL*. La Figure VI-11 présente la courbe de débit en utilisant le protocole de rendez-vous et celle utilisant les recopies intermédiaires.

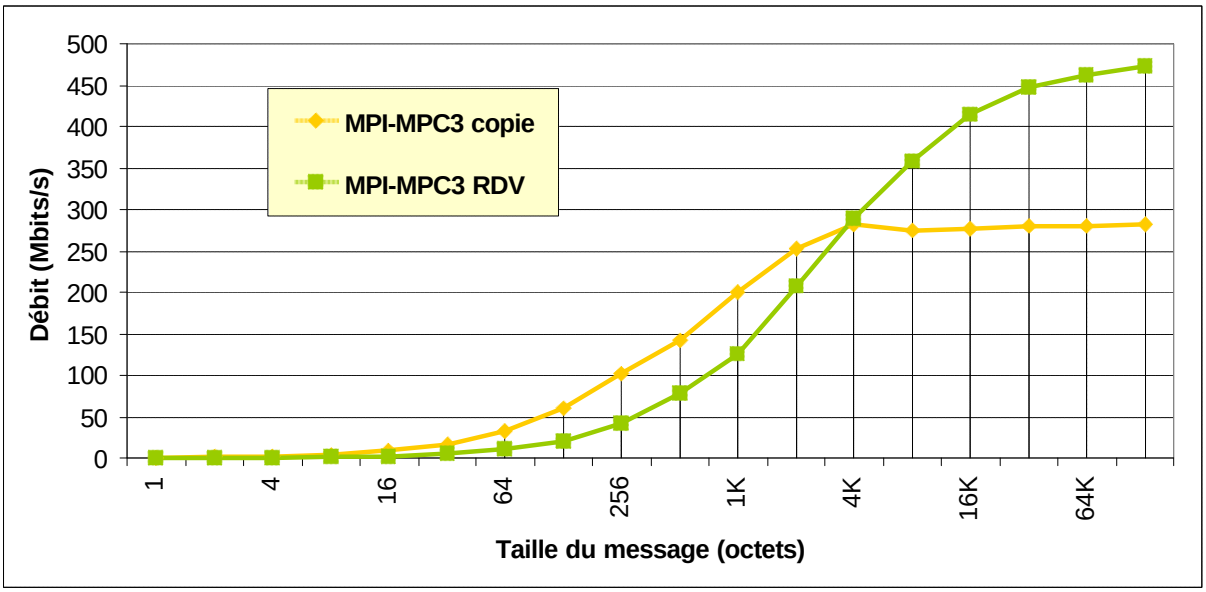


Figure VI-11 : Le seuil optimal des messages de contrôle dans MPI-MPC3

Sur notre implémentation MPI-MPC3, le protocole de rendez-vous devient plus performant que l'encapsulation des données dans un message *CTRL* à partir de 4Ko.

VI.5.3. Courbe de débit

Maintenant que nous avons déterminé la taille maximale des messages *CTRL*, nous pouvons tracer la courbe de débit de MPI-MPC3. La Figure VI-12 représente les résultats obtenus en rappelant les courbes que nous avons obtenues pour MPI-MPC1 et MPI-MPC2.

La redistribution de la mémoire n'apporte aucune amélioration pour les messages d'une taille inférieure à 4Ko : les traductions d'adresse n'interviennent pas tant que les données sont encapsulées dans un message *CTRL*. En revanche, dès que le protocole de rendez-vous est utilisé, le débit de MPI-MPC3 devient supérieur à celui de MPI-MPC2. Le gain de MPI-MPC3 par rapport à MPI-MPC2 sera analysé dans la section VI.5.7.

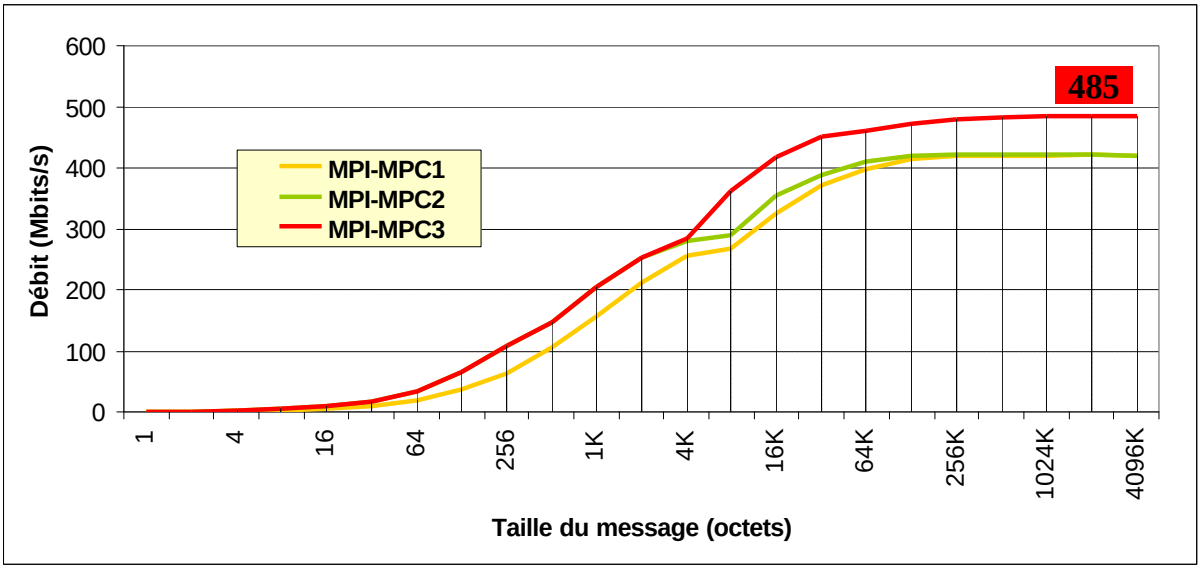


Figure VI-12 : Courbe de débit de MPI-MPC3

Le débit maximum de MPI-MPC3 est de **485Mbits/s** alors que celui de MPI-MPC2 était de 421Mbits/s. Il est proche du débit utile maximum pour transmettre des tampons contigus en mémoire physique avec la primitive d'écriture distante de notre plate-forme expérimentale : 494Mbits/s [Fenÿo,2001].

VI.5.4. Le demi débit

La Figure VI-13 montre que le demi débit de MPI-MPC3 est de 1,5Ko. Il était de 1Ko pour MPI-MPC2 et de 2Ko pour MPI-MPC1. On ne constate pas d'amélioration sensible car le débit maximum de MPI-MPC3 est supérieur à celui de MPI-MPC2 et MPI-MPC1.

VI.5.5. Latence

La redistribution n'apporte aucun changement par rapport à MPI-MPC2 en ce qui concerne la latence. Ce que nous avons décrit à la section V.5.5 reste valable pour MPI-MPC3.

La latence totale que nous avons mesurée est de **15µs** pour MPI-MPC3. Elle est égale à celle mesurée pour MPI-MPC2.

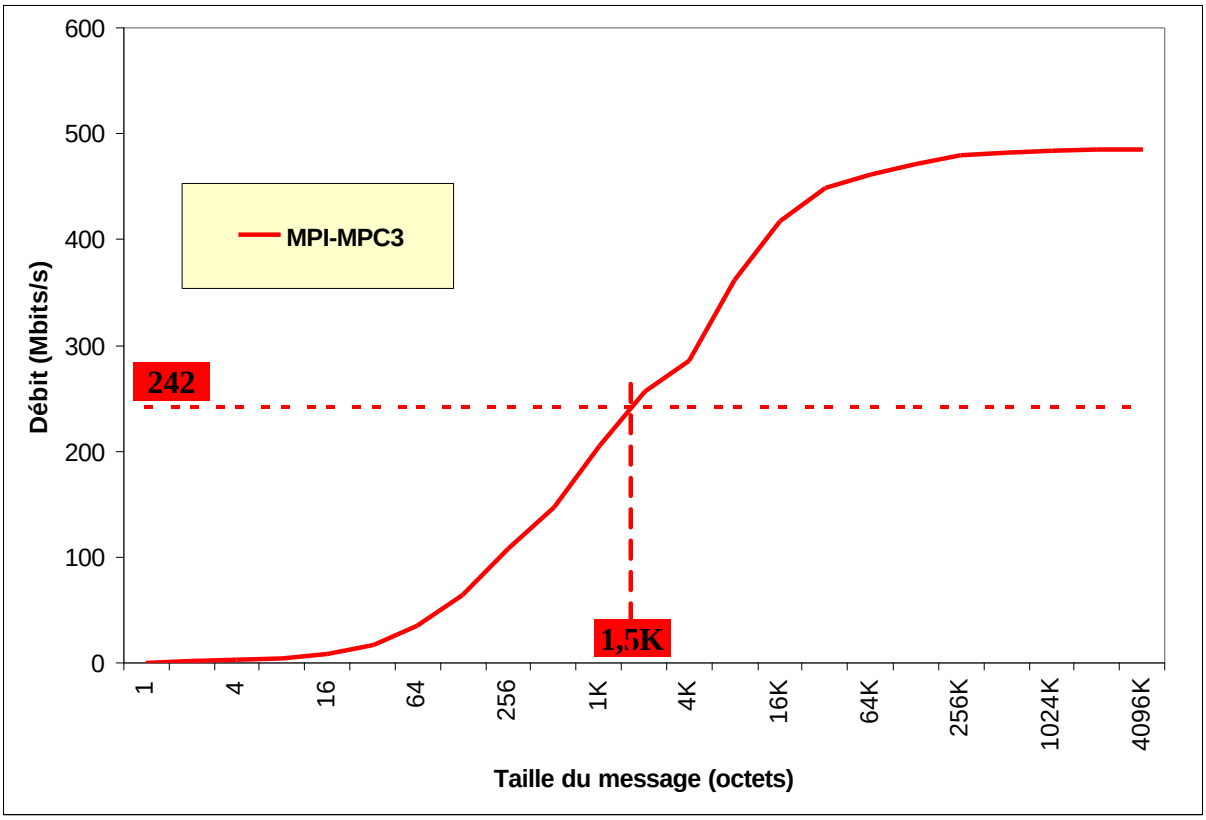


Figure VI-13 : Demi débit de MPI-MPC3

VI.5.6. Tableau récapitulatif

Le Tableau VI-1 récapitule les performances de MPI-MPC3. Les performances que nous avons obtenues pour MPI-MPC1 et MPI-MPC2 sont rappelées.

La latence mesurée avec le protocole de rendez-vous diminue : elle est de 49µs avec MPI-MPC3 alors qu'elle était de 72µs pour MPI-MPC2. Ce gain de 23µs provient de la suppression, dans MPI-MPC3, du verrouillage et des traductions d'adresse des tampons d'émission et de réception. Ce chiffre de 49µs est cohérent : il correspond aux transferts de deux messages *CTRL* et d'un message *DATA* d'un octet.

	Latence (µs)	Débit Max. (Mb/s)	Seuil (Ko)	Demi débit (Ko)	Latence RDV (µs)
MPI-MPC1	26	419	16	2	105
MPI-MPC2	15	421	8	1	72
MPI-MPC3	**15**	**485**	**4**	**1,5**	**49**

Tableau VI-1 : Performances de MPI-MPC3

Le coût pour verrouiller et traduire un tampon d'une taille d'un octet est estimé à **11,5µs** sur notre plate-forme expérimentale (ces deux opérations nécessitent deux appels système).

L'Annexe A explique comment les traductions d'adresse virtuelle/physique sont réalisées.

VI.5.7. Analyse des performances de MPI-MPC3

Le Tableau VI-2 récapitule, pour MPI-MPC3, le nombre de chacun des facteurs que nous avons définis à la section IV.6.7 : appels système, interruptions, recopies des données de l'application, écritures distantes.

	Appels systèmes		Interruptions		Recopies		RDMA_SEND	
Protocole	short	rdv	short	rdv	short	rdv	short	rdv
MPI-MPC1	1	1+6F	2	2+4F	2	0	1	1+F+NB_DMA*F
MPI-MPC2	0	4F	0	0	2	0	1	1+F+NB_DMA*F
MPI-MPC3	**0**	**0**	**0**	**0**	**2**	**0**	**1**	**1+F+F**

Tableau VI-2 : Analyse des performances de MPI-MPC3

Nous rappelons que F représente le nombre de messages *DATA* envoyés pour transmettre un message de l'application. NB_DMA est le nombre d'écritures distantes nécessaires pour transmettre un message *DATA* avec MPI-MPC1 et MPI-MPC2 (c'est-à-dire le nombre d'appels de la fonction `RDMA_SEND`). Il dépend de la description en mémoire physique des tampons d'émission et de réception.

La redistribution de la mémoire apporte trois améliorations par rapport à MPI-MPC2 :

- ✓ La première est la suppression des 4F appels système pour le verrouillage et la traduction d'adresse du tampon d'émission et de réception : dans MPI-MPC2, l'émission de chaque message *DATA* nécessite deux appels système sur l'émetteur et deux sur le récepteur.
- ✓ la deuxième concerne le nombre d'écritures distantes nécessaires pour transmettre un message *DATA* : il était de NB_DMA pour MPI-MPC2 ; avec la redistribution de la mémoire, une seule écriture distante est nécessaire du fait de la contiguïté des données en mémoire physique. Plusieurs centaines d'appels à la primitive `RDMA_SEND` pouvaient être nécessaires pour transmettre un message de 1024K avec MPI-MPC1 et MPI-MPC2.
- ✓ La troisième amélioration n'apparaît pas dans le Tableau VI-2 mais a une grande part dans l'amélioration des performances : il s'agit de la taille du message *RSP* dans le protocole de rendez-vous. Avec MPI-MPC3, le message *RSP* transporte la description d'une seule zone de mémoire physique, sa taille est de 28 octets quelle

que soit la taille du tampon de réception des données. Avec MPI-MPC2, la taille du message *RSP* pouvait être de plusieurs Kilo-octets.

Le gain de MPI-MPC3 par rapport à MPI-MPC2 concerne le transfert des messages de grande taille. La redistribution de la mémoire permet une augmentation significative du débit maximum mais n'a aucune influence sur la latence. Avec MPI-MPC3, les appels système ont totalement disparu des phases de communication.

La Figure VI-14 représente le gain sur les temps de transfert de MPI-MPC3 par rapport à ceux de MPI-MPC2, pour toutes les tailles de message. MPI-MPC3 n'apporte aucune amélioration tant que les données sont encapsulées dans un message *CTRL*. Lorsque le protocole de rendez-vous est utilisé, la différence entre les temps de transfert de MPI-MPC2 et ceux de MPI-MPC3 augmente avec la taille du message : plus le message de l'application est gros, plus la redistribution de la mémoire est avantageuse.

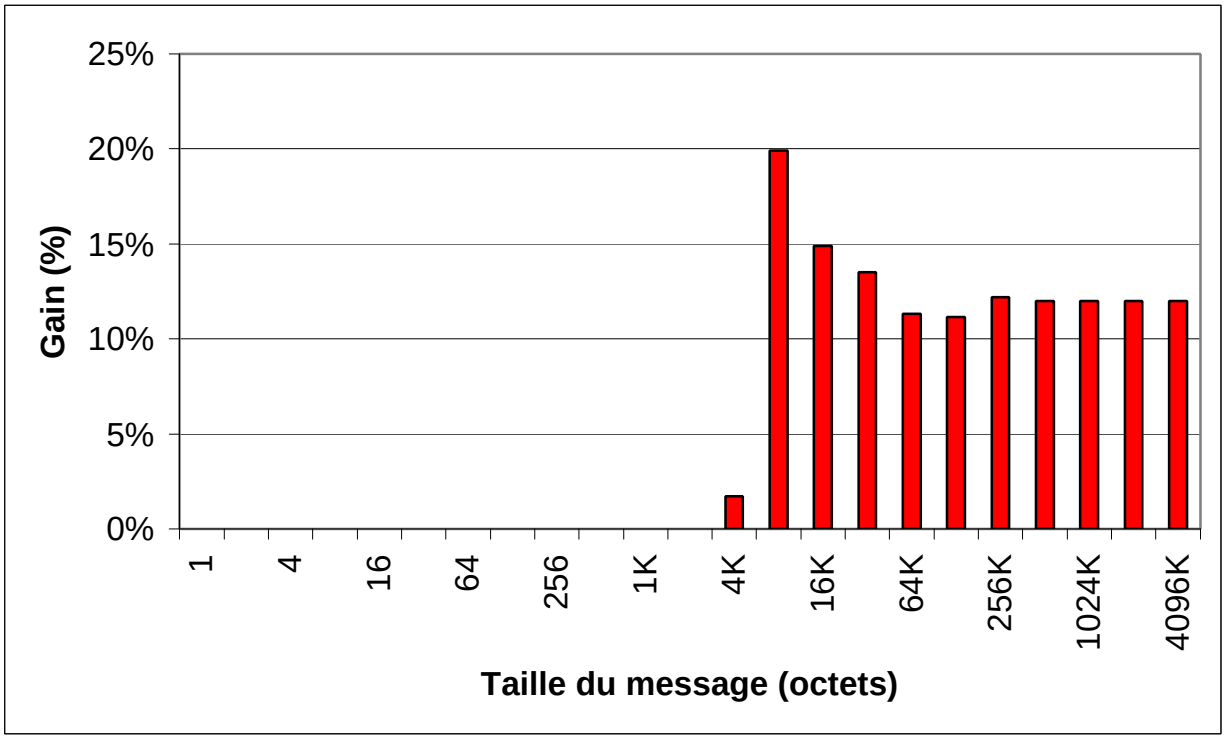


Figure VI-14 : Gain de MPI-MPC3 / MPI-MPC2

Pour une taille de message de quelques Kilo-octets, le gain mesuré est principalement lié à la suppression du verrouillage des tampons d'émission et de réception, et à la simplification des traductions d'adresse. Pour des messages de plus grande taille, la source du gain provient surtout de la diminution de la taille du message *RSP* et du nombre d'écritures distantes. Le gain se stabilise autour de 12% à partir d'une taille de message de 256Ko.

VI.5.8. Le coût de la redistribution de la mémoire

Nous n'avons pas encore parlé du coût de la redistribution de la mémoire, c'est-à-dire du temps nécessaire à la réalisation des opérations décrites en VI.4.5. Ces opérations sont réalisées lors de l'initialisation de l'application. Le coût principal concerne l'opération de sauvegarde sur le disque des segments du processus et l'opération de lecture de ces segments pour les recopier dans les zones « pile » et « données ». Le reste des opérations a un coût négligeable. En ce qui concerne la pile, la taille sauvegardée est très faible car la taille de la pile est de quelques Kilo-octets. En revanche, concernant le tas, nous sommes obligés de sauvegarder la taille de la zone « données ». La taille de cette zone dépend de la mémoire physique disponible sur la machine, du nombre de processus utilisant la redistribution de la mémoire sur le nœud local, et de la taille réservée à la zone « pile ». La zone « données » peut donc faire plusieurs centaines de Mega-octets.

Nous avons mesuré le coût total de la redistribution de la mémoire sur notre plate-forme expérimentale. Il est de sept secondes quand la taille de la zone « données » est de 100Mo. Notre plate-forme est équipée de disques durs IDE standard. L'écriture et la lecture des 100Mo sont réalisées sur le disque local.

Le coût de la phase de redistribution de la mémoire réalisée lors de l'initialisation de l'application est de l'ordre de quelques secondes : ce coût dépend de la quantité de mémoire physique dont chaque processus a besoin pour le tas.

VI.6. Conclusion

Nous avons décrit dans ce chapitre notre troisième implémentation (MPI-MPC3) de MPI au-dessus de l'API RDMA. Sa particularité est d'éliminer les opérations de verrouillage et de traduction d'adresse des tampons d'émission et de réception des données.

Nous avons décrit un mécanisme qui permet de se ramener à une situation dans laquelle les segments mémoire de chaque processus de l'application correspondent à des zones de mémoire physique contiguës. Le principe de notre solution est d'attribuer statiquement toute la mémoire physique disponible d'un nœud de calcul à l'ensemble des processus de l'application s'exécutant sur ce nœud.

Cette redistribution de la mémoire du processus est réalisée lors de l'initialisation de l'application. Elle ne nécessite aucune modification, ni du système d'exploitation, ni de la librairie C, ni du code de l'application. Notre méthode utilise une librairie dynamique écrite en C++ qui nous permet d'intervenir entre le chargement du processus en mémoire et le début de son exécution. Pour bénéficier de la redistribution de la mémoire, il suffit que l'application soit liée à notre librairie dynamique lors de la phase de compilation.

Avec la redistribution de la mémoire, il n'y a plus besoin de verrouiller les tampons de communication en mémoire physique : les zones « données » et « pile » ne peuvent pas être évincées de la mémoire physique par le système d'exploitation. Les traductions d'adresse sont simplifiées à l'extrême : il suffit de conserver au niveau de chaque processus de l'application et pour chacune des deux zones de mémoire physique contiguë un triplet (adresse virtuelle, adresse physique, longueur). Lorsqu'un processus veut obtenir l'adresse physique correspondant à une adresse virtuelle, il lui suffit de trouver la zone correspondante, de récupérer l'adresse physique du début de la zone et de soustraire le décalage à l'adresse virtuelle considérée.

La redistribution de la mémoire permet :

- ✓ la suppression des appels système liés aux opérations de verrouillage et de traduction d'adresse,
- ✓ la simplification des conversions d'adresse virtuelle en adresse physique,
- ✓ la réduction de la taille du message *RSP* dans le protocole de rendez-vous à 28 octets quelle que soit la taille du message transmis,
- ✓ la transmission d'un message *DATA* à l'aide d'une seule écriture distante quelle que soit sa taille.

Les performances mesurées sur notre plate-forme expérimentale montrent que la redistribution de la mémoire permet d'atteindre un débit maximum au niveau applicatif (MPI) très proche du débit maximum utile de la plate-forme. En revanche, elle n'a aucune influence sur la latence.

Chapitre VII : Résultats expérimentaux sur des applications réelles

Sommaire

Nous réalisons dans ce chapitre des mesures de performances sur deux applications réelles, avec les trois implémentations de MPI décrites aux chapitres IV, V, et VI. L'objectif est d'évaluer si les conclusions générales qui ont pu être tirées de la comparaison des trois implémentations de MPI dans le cas du ping-pong restent valables dans le cas d'une application réelle.

VII.1. Introduction

Nous avons choisi deux applications utilisant un schéma de communication simple et reproductible afin de pouvoir analyser les différences de performances entre nos implémentations :

- ✓ une résolution d'un système linéaire par la méthode de Gauss qui utilise une implémentation parallèle de la méthode CESTAC [Vignes,1974] permettant de prendre en compte les erreurs d'arrondi dans les calculs flottants,
- ✓ une résolution de l'équation de Laplace en parallèle par la méthode itérative de Jacobi.

Nos travaux autour d'une implémentation optimisée de MPI ont permis une collaboration avec l'équipe ANP (Algorithmique Numérique et Parallélisme) du LIP6 qui a développé le logiciel CADNA [CADNA] [Chesneaux,1995] [Vignes,1993]. Cette bibliothèque remplace l'arithmétique classique IEEE sur les nombres à virgules flottantes par une arithmétique aléatoire permettant de prendre en compte les erreurs de propagation d'arrondi dans les programmes scientifiques. Nous avons utilisé l'implémentation parallèle de CADNA (réalisée par Jean-Luc Lamotte) pour résoudre un système linéaire par la méthode de Gauss avec recherche du pivot maximum. Nous avons choisi cette application pour comparer nos implémentations MPI-MPC1 et MPI-MPC2 car, elle suppose l'échange de nombreux messages de petite taille et il suffit d'augmenter la taille du système linéaire pour augmenter le nombre de communications. Pour cette application, nous n'avons pas réalisé de mesures avec MPI-MPC3 car elle n'apporte aucune amélioration par rapport à MPI-MPC2 pour les messages d'une taille inférieure à 4Ko.

La résolution de l'équation de Laplace est une application classique qui a des implications dans divers domaines des mathématiques et de la physique. La parallélisation de cette application nécessite l'échange de messages de grande taille. Elle nous permettra de comparer les performances de MPI-MPC2 et MPI-MPC3. Nous avons choisi cette application car elle fait l'objet d'une collaboration avec le professeur Carl Hamacher de Queen's University au Canada.

VII.2. Résolution d'un système linéaire avec CADNA

Cette application permet de résoudre un système linéaire par la méthode de Gauss en prenant en compte la propagation des erreurs d'arrondi, grâce au logiciel CADNA, développé au Laboratoire d'Informatique de Paris 6 (LIP6). L'intérêt de cette application parallèle est qu'elle nécessite beaucoup de communications de messages de petite taille.

VII.2.1. La méthode CESTAC et le logiciel CADNA

Les algorithmes numériques utilisent principalement l'arithmétique des nombres réels. La plupart des nombres réels ne pouvant pas être représentés exactement dans la mémoire d'un ordinateur (par exemple, le nombre π), il est donc nécessaire de les tronquer ou de les arrondir pour les faire tenir dans les 32 ou 64 bits généralement alloués pour les nombres réels. Cette petite erreur d'arrondi peut dans certains cas se propager et entacher irrémédiablement tout résultat numérique d'un programme scientifique. La méthode CESTAC [Vignes,1974], créée par J. Vignes et M. Laporte en 1974, propose une nouvelle méthode de contrôle et d'estimation des erreurs de propagation d'arrondi. Elle repose sur la définition d'une nouvelle arithmétique appelée l'arithmétique stochastique [Chesneaux,1990]. Elle propose une redéfinition de l'opération d'égalité, des relations d'ordre (prenant en compte la précision de chaque opérande), et du nombre zéro (un zéro informatique est un nombre avec aucun chiffre significatif ou un zéro mathématique [Vignes,1993]).

La bibliothèque CADNA (*Control of Accuracy and Debugging for Numerical Applications*) est une implémentation synchrone de la méthode CESTAC. Chaque opération arithmétique est effectuée trois fois avant d'effectuer la suivante. Tout se passe comme si trois programmes identiques se déroulaient simultanément sur trois ordinateurs synchronisés utilisant chacun une arithmétique aléatoire. L'arithmétique aléatoire consiste à attribuer à chaque résultat d'une opération sa valeur par excès ou sa valeur par défaut avec une probabilité de ½. Les trois programmes conduisent alors à trois résultats différents : (R_1), (R_2) et (R_3). Le résultat informatique ($\overline{R}$) est alors la moyenne de tous les (R_i) et son nombre de chiffres significatifs est ($C_{\overline{R}}$) :

$$\overline{R} = \frac{1}{3} * \sum_{i=1}^{3} R_i \ , \quad C_{\overline{R}} = Log_{10}\left(\frac{\sqrt{3} * |\overline{R}|}{s * t_b} \right) \quad \text{avec} \quad s^2 = \frac{1}{2} * \sum_{i=1}^{3} \left(R_i - \overline{R} \right)^2$$

où t_β est une constante statistique.

CADNA permet au programmeur de disposer de nouveaux types numériques : les types stochastiques. Le contrôle des erreurs d'arrondi s'effectue uniquement sur ces types. On en dénombre deux : `single_st` (type stochastique simple précision) et `double_st` (type stochastique double précision). Ils tiennent compte de la précision

des opérandes. CADNA surcharge chaque opérateur arithmétique et chaque relation d'ordre avec ces deux types. Tous les types réels d'un programme sont remplacés par un type stochastique. Une opération revient alors à exécuter les instructions suivantes :

```
pour i=1 à 3
faire
     choisir aléatoirement l'arrondi par excès ou par défaut
     calculer le résultat Ri de l'opération
fin
```

La méthode peut fournir le nombre de chiffres significatifs de chaque résultat (intermédiaire ou final) et contrôler la validité des tests et des branchements qui en résultent. En sortie, seuls les chiffres significatifs exacts sont affichés ce qui permet de visualiser très facilement la précision des résultats. Dans le cas de zéros stochastiques, c'est-à-dire de résultats non significatifs, CADNA affiche « @.0 ». La détection des erreurs d'arrondi est dynamique : elle porte, non pas sur la validité du programme, mais sur les capacités de l'ordinateur à fournir des résultats corrects en l'exécutant. Par exemple, les deux instabilités fondamentales sont :

- ✓ DIVISION INSTABLE : cela signifie que lors d'une division, le dénominateur est un zéro stochastique,
- ✓ TEST INSTABLE : cela signifie que dans l'évaluation de $A \leq B$, $(A-B)$ est un zéro stochastique. Par application de la définition stochastique correspondante, la branche de l'égalité sera exécutée. Mais l'utilisateur est averti du fait que la réponse mathématique peut être l'opposé de la réponse informatique.

Les instabilités détectées sont enregistrées dans un fichier pendant l'exécution. L'utilisation de CADNA dans un programme scientifique a un coût : (1) le temps de calcul est multiplié par un facteur allant de 5 à 10 et que seule l'estimation de la précision est réalisée ; (2) la taille mémoire des données est multipliée par un facteur 3.

VII.2.2. Description de l'application

Une implémentation parallèle de la méthode CESTAC a été réalisée par Jean-Luc Lamotte pour permettre de réduire le surcoût résultant de l'utilisation de CADNA. Cette implémentation utilise la bibliothèque de communication MPI. Le principe consiste à calculer chaque résultat R_i d'une opération stochastique sur un processeur différent. L'application est donc divisée en trois processus MPI qui s'exécutent en parallèle. Lors d'une opération stochastique, les trois processus doivent s'échanger leur résultat R_i pour calculer le résultat moyen $\overline{R}$ qui est ensuite utilisé dans la suite du programme. L'exécution aboutit donc au même résultat sur chacun des trois processus. Le schéma de communication est présenté sur la Figure VII-1.

Les opérations stochastiques nécessitant un tel échange sont :

- ✓ les tests de comparaisons,
- ✓ la valeur absolue,
- ✓ les fonctions d'affichage : le résultat est la moyenne des trois valeurs et l'affichage n'affiche que les chiffres significatifs.

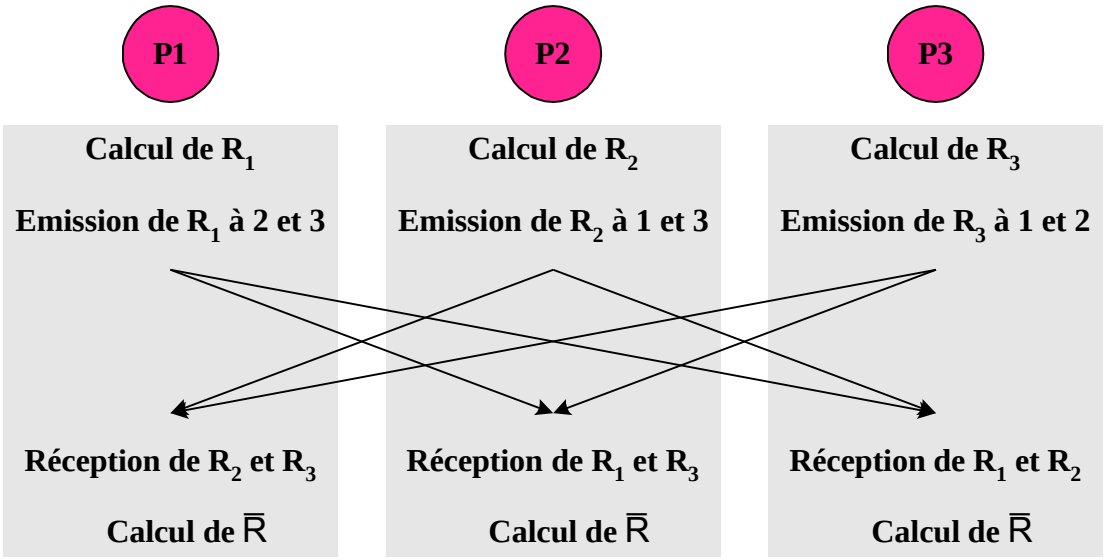


Figure VII-1 : Le schéma de communication dans la version parallèle de CADNA

Pour tester cette implémentation parallèle de la méthode CESTAC, nous avons utilisé la résolution d'un système linéaire par la méthode de Gauss. Cette application est intéressante car elle permet de résoudre des systèmes de tailles différentes et l'algorithme peut être réalisé avec ou sans la recherche du pivot maximum Sans recherche du pivot, il n'y a aucune communication qui est engendrée par la méthode CESTAC. Dans l'autre cas, les tests de comparaison et la valeur absolue permettant de trouver le pivot génèrent beaucoup de communications.

VII.2.3. Résultats sur notre plate-forme expérimentale

Les résultats présentés dans cette section ont été publiés dans [Glück,2002]. La plate-forme expérimentale que nous utilisons est la même que celle des chapitres IV, V et VI : nous utilisons 3 processeurs PII-350 interconnectés par le réseau HSL de la machine MPC. Chaque nœud de calcul possède 256Mo de mémoire. Nous avons fait deux séries de mesures : une avec l'implémentation MPI-MPC1 décrite au chapitre IV et une avec l'implémentation MPI-MPC2 décrite au chapitre V. Dans chacun des deux cas, nous avons mesuré le temps d'exécution global pour résoudre des systèmes linéaires de différentes tailles (800, 1200, 1600 et 2000 équations), par la méthode de Gauss, en utilisant la bibliothèque CADNA, avec et sans la recherche du pivot maximum. Les temps d'exécution sont présentés en secondes dans le Tableau VII-1.

	Sans recherche du pivot	Avec recherche du pivot		Différence entre avec et sans recherche du pivot	
Taille du système linéaire	Temps d'exécution (sec.)	Temps d'exécution (sec.)		Temps pour rechercher le pivot et permuter les lignes (sec.)	
	Pas de com.	MPI-MPC1	MPI-MPC2	MPI-MPC1	MPI-MPC2
800	100	151	131	51	31
1200	348	449	414	101	66
1600	786	977	914	191	128
2000	1580	1868	1757	288	177

Tableau VII-1 : Temps d'exécution de CADNA avec MPI-MPC1 et MPI-MPC2

Sans la recherche du pivot maximum, la résolution du système avec CADNA n'implique aucune communication, et donc aucun appel à notre bibliothèque MPI car les fonctions de test et de valeur absolue ne sont pas utilisées. Comme prévu, les temps d'exécution avec MPI-MPC1 et MPI-MPC2 sans la recherche du pivot sont identiques : nous avons présenté une seule colonne dans le tableau ci-dessus.

Avec la recherche du pivot maximum, CADNA ajoute, non seulement beaucoup de communications (1 échange lors de chaque test de comparaison et lors de chaque utilisation de la valeur absolue), mais aussi du temps de calcul supplémentaire (faible par rapport au temps de communications) pour rechercher le pivot et permuter les lignes. Comme prévu, le temps d'exécution global est plus important avec MPI-MPC1 qu'avec MPI-MPC2, quelle que soit la taille du système résolu. Les deux dernières colonnes du Tableau VII-1 représentent le coût supplémentaire de la recherche du pivot maximum pour MPI-MPC1 et MPI-MPC2. Ce temps est déduit des colonnes précédentes du tableau par une soustraction. Il inclut l'ensemble des temps de communications.

VII.2.4. Analyse des performances de MPI-MPC1 et MPI-MPC2

Nous cherchons ici à vérifier si les résultats obtenus dans le Tableau VII-1 sont cohérents avec les mesures de latence pour MPI-MPC1 et MPI-MPC2 présentées aux chapitre IV et V. Nous avons reporté, dans le Tableau VII-2, la différence entre les temps d'exécution (en secondes) avec et sans la recherche du pivot maximum. Nous avons comptabilisé, dans l'application, le nombre d'échanges suivant le schéma de la Figure VII-1. En négligeant le temps de permutation des lignes, nous avons déduit, dans les deux dernières colonnes du Tableau VII-2, le temps moyen d'un échange en micro secondes. La valeur moyenne est présentée dans la dernière ligne du tableau pour MPI-MPC1 et MPI-MPC2.

Taille du système linéaire	Nombre d'échanges	Différence entre avec et sans recherche du pivot (sec.)		Temps d'un échange (µs)	
		MPI-MPC1	MPI-MPC2	MPI-MPC1	MPI-MPC2
800	646682	51	31	79	48
1200	1450450	101	66	70	46
1600	2574140	191	128	74	50
2000	4018285	288	177	72	44
Temps moyen d'un échange (µs)				**74**	**47**

Tableau VII-2 : Analyse des performances de MPI-MPC1 et MPI-MPC2

Au niveau de chaque processus, un échange correspond à deux émissions et deux réceptions MPI bloquantes (`MPI_Send` et `MPI_Recv`) auxquelles s'ajoute du calcul lié à CADNA. La taille des données échangées est de 8 octets. Ces données sont encapsulées dans un message *CTRL* de notre implémentation de MPI. Le temps moyen d'un échange est de 74µs pour MPI-MPC1 et 47µs pour MPI-MPC2. Cela correspond à un gain de 36% avec MPI-MPC2. La différence est de 27µs. En divisant ce temps par deux, on obtient 13,5µs ce qui est cohérent avec les mesures présentées aux chapitres IV et V : l'écart entre la latence de MPI-MPC1 et celle de MPI-MPC2 est de 11µs.

VII.3. Résolution de l'équation de Laplace

La résolution de l'équation de Laplace a de nombreuses applications dans les domaines des mathématiques appliquées, de la mécanique des fluides, de la résistance des matériaux, de l'électromagnétisme, etc.

VII.3.1. Description séquentielle

L'équation de Laplace est une équation aux dérivées partielles :

$$\frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} = 0 \text{, parfois notée } \nabla^2 F(x, y) = 0$$

La résolution de cette équation consiste à trouver les valeurs de F(x,y) sur un domaine Ω inclus dans $\mathbb{R}^2$ dont la frontière est Γ. La seule donnée de cette équation n'est pas suffisante à la détermination d'une solution unique. Pour cela, il faut donner les conditions limites (ou conditions initiales). Les conditions de Dirichlet consistent en la donnée des valeurs de F sur Γ, la frontière du domaine de résolution.

Pour résoudre cette équation, il existe des méthodes directes (utilisant des inversions de matrices bandes) et des méthodes itératives. Les méthodes directes sont parfois plus rapides mais les méthodes itératives ont l'avantage de permettre le choix de la précision de résolution et donc de pouvoir accélérer la convergence. Nous nous intéressons aux méthodes itératives avec des conditions limites de Dirichlet.

En utilisant une discrétisation selon x et y, on montre que la résolution de l'équation de Laplace revient à mettre à jour chaque point intérieur de la grille par la moyenne de ses quatre voisins, les valeurs des frontières étant fixées par les conditions initiales [Freeman,1992 (p269)]. La mise à jour s'arrête lorsque chacun des points de la grille est suffisamment proche de la moyenne de ses quatre voisins : il y a alors convergence. Sur la Figure VII-2, les frontières ont une valeur égale à 100.

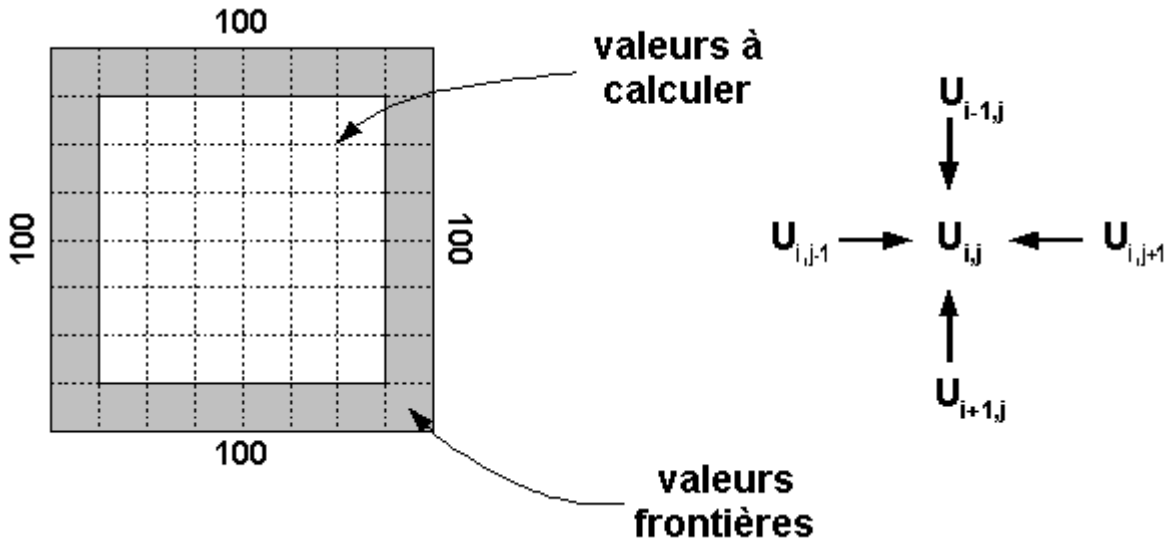


Figure VII-2 : Résolution séquentielle de l'équation de Laplace

Il existe différentes méthodes itératives permettant cette mise à jour. Nous avons utilisé la méthode de Jacobi, qui utilise l'itération suivante (appelée « itération de Jacobi ») :

$$U_{i,j}^{(k+1)} = \frac{1}{4} * (U_{i+1,j}^{(k)} + U_{i-1,j}^{(k)} + U_{i,j+1}^{(k)} + U_{i,j-1}^{(k)})$$

où $U^{(k)}_{i,j}$ est la valeur du point de coordonnées (i,j) après la $k^{ième}$ itération. La mise à jour des valeurs calculées à l'itération k+1 utilise uniquement les valeurs calculées à l'itération k. Donc, cette méthode nécessite de conserver pour chaque nouvelle itération les valeurs calculées à l'itération précédente.

Le pseudo-code réalisant la résolution séquentielle par la méthode de Jacobi sur une grille [N*N] est le suivant :

```
Initialisations
max_change = 0 ;
tant que (max_change > précision ou niter <= MAX_ITER)
faire
  niter ++ ;
  max_change = 0 ; //le plus grand écart entre deux itérations
successives
  pour i=2 à N-1   // traitement lignes
    pour j allant de 2 à N-1 // traitement colonnes
      Uold(i,j) = U(i,j) ;  // sauvegarde de l'ancienne valeur
      // mise à jour par l'itération de Jacobi
      U(i,j)  =  [Uold(i-1,j)  +  Uold(i+1,j)  +  Uold(i,j-1)  +
Uold(i,j+1)]/4 ;
      max_change = max ( max_change, abs(Uold(i,j) – U(i,j))
) ;
    fin pour
  fin pour
fin tant que
```

VII.3.2. Description parallèle

Pour paralléliser cette application, la principale question est celle de la décomposition du problème entre les différents processus. [Freeman,1992 (p267)] [Culler,1999 (p92)] [Pfister,1998 (p239)] traitent de la parallélisation de cette application. Afin de simplifier le schéma de communication, nous avons choisi de découper la matrice (grille [N*N]) en plusieurs bandes de M lignes. Chacune des bandes (ou sous-grilles) est attribuée à un processus. La Figure VII-3 représente le découpage en bandes d'une matrice [20,20] sur quatre processus. Lors de l'initialisation, le processus maître (par exemple, P1 sur la figure) envoie chaque sous-grille aux autres processus. A chaque itération, les processus échangent leurs frontières avec leur(s) voisin(s), puis mettent à jour leur partie de la grille. Par exemple, les processus « intérieurs » (P2 et P3) échangent leurs frontières inférieure et supérieure avec leurs deux voisins. La réception des frontières est bloquante car un processus ne peut pas commencer à calculer tant qu'il n'a pas reçu les frontières voisines. Il y a donc une synchronisation entre les processus voisins lors de chaque itération. Le contrôle de la convergence est réalisé par le processus maître P1 dont le rôle est de rassembler tous les écarts de convergence locaux puis de comparer le plus grand d'entre eux à la précision souhaitée. Ce processus maître indique alors aux autres processus s'ils doivent continuer la mise à jour ou non. Les tests de convergence peuvent se faire uniquement toutes les 50, 100 ou 1000 itérations afin de limiter les synchronisations globales qu'ils engendrent. Une fois le calcul terminé, les processus P2, P3 et P4 envoient leur sous-grille au processus P1 qui rassemble les résultats.

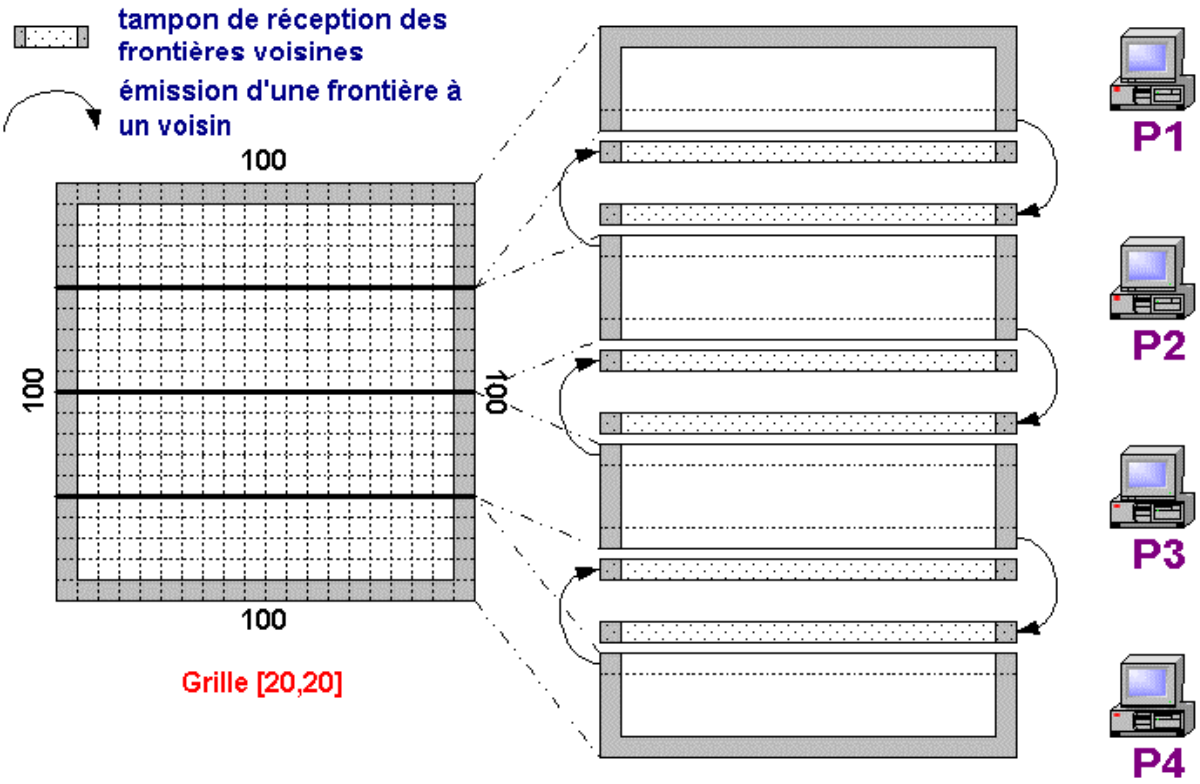


Figure VII-3 : Résolution parallèle de l'équation de Laplace

VII.3.3. Résultats sur notre plate-forme expérimentale

Nous avons utilisé la parallélisation de la résolution de l'équation de Laplace par la méthode de Jacobi pour comparer les performances de MPI-MPC2 et MPI-MPC3. La plate-forme expérimentale est la même que précédemment : nous utilisons quatre processeurs PII-350 interconnectés par le réseau HSL de la machine MPC.

Nombre de processeurs	Nombre de lignes	Nombre de colonnes	Taille Matrice (Ko)	Taille Bande (Ko)	Taille Frontière (Ko)
4	60	3200	750	187,5	12,5
4	60	6400	1500	375	25
4	60	12800	3000	750	50
4	60	25600	6000	1500	100
4	60	51200	12000	3000	200

Tableau VII-3 : Les données de l'application

Le Tableau VII-3 présente les données de l'application : nous avons fait nos mesures avec des matrices réelles de 60 lignes, le nombre de colonnes variant de 3200 à 51200.

Nous faisons varier le nombre de colonnes afin d'augmenter la taille des messages envoyés lors de l'échange des frontières. Nous avons choisi de fixer le nombre de lignes, d'une part pour que les deux matrices (la matrice des points mis à jour à l'itération précédente et celle des points en cours de mise à jour) ne prennent pas trop de place en mémoire et, d'autre part pour ne faire varier qu'un seul paramètre à la fois. Nous exécutons l'application sur quatre processeurs : chaque processeur fait ses calculs sur une bande de 15 lignes. La taille des messages lors de l'échange des frontières varie entre 12,5Ko et 200Ko. Nous n'avons donc pas réalisé de mesures avec MPI-MPC1 car les performances de MPI-MPC1 et MPI-MPC2 sont très proches lorsque la taille des messages est supérieure à 8Ko. La taille maximale prise en mémoire par les deux matrices étant proche de 24Mo, nous avons réservé 30Mo de mémoire physique contiguë pour l'exécution de l'application avec MPI-MPC3. On précise que ces deux matrices se trouvent dans le segment de données des processus.

Les résultats expérimentaux sont présentés dans le Tableau VII-4. Nous y avons reporté les temps d'exécution en secondes de l'application séquentielle et de l'application parallèle en utilisant MPI-MPC2 et MPI-MPC3. Dans tous les cas, le nombre d'itérations nécessaire pour atteindre la convergence est 3150. Précisons que nous avons choisi de tester la convergence toutes les 50 itérations avec une précision de 10^{-2}.

Nombre de colonnes	Nombre d'itéra-tions	Temps séquentiel (sec.)	Temps d'exécution parallèle (sec.)		*Speed-up*	
			MPI-MPC2	MPI-MPC3	MPI-MPC2	MPI-MPC3
3200	3150	70,2	18,3	19,4	3,8	3,6
6400	3150	163,8	46,8	47,5	3,5	3,4
12800	3150	327,6	92,6	92,3	3,5	3,5
25600	3150	689,3	187,9	185,4	3,7	3,7
51200	3150	1441,3	391,5	384,8	3,7	3,7

Tableau VII-4 : Temps d'exécution avec MPI-MPC2 et MPI-MPC3

Nous constatons que les temps d'exécution de l'application parallèle sont très voisins, mais que les performances de MPI-MPC3 sont légèrement inférieures à celles de MPI-MPC2 pour un nombre de colonnes égal à 3200 et 6400. Nous expliquons pourquoi dans la section suivante. Les deux dernières colonnes du tableau présentent le *speed-up* de l'application parallèle avec MPI-MPC2 et MPI-MPC3 : il s'agit du temps d'exécution séquentiel divisé par le temps d'exécution parallèle. Nous constatons que la résolution de l'équation de Laplace en parallèle conduit à de bons résultats puisque le *speed-up* est proche de quatre lorsque le calcul est réalisé par quatre processeurs.

VII.3.4. Analyse des performances de MPI-MPC2 et MPI-MPC3

Pour un nombre de colonnes de la matrice égal à 3200 et 6400, les performances de MPI-MPC3 sont légèrement inférieures à celles de MPI-MPC2. Cela s'explique par le fait que le coût initial de la redistribution de la mémoire des processus dans MPI-MPC3 est trop important par rapport au gain de temps réalisé sur les communications. En effet, ce coût initial consiste à lire et écrire sur le disque dur 30Mo. Nous avons évalué ce temps à deux secondes sur notre plate-forme expérimentale.

Nous analysons maintenant les performances de MPI-MPC2 et MPI-MPC3 concernant les temps de communication. Pour cela, nous avons mesuré les temps liés à l'échange des frontières sur chacun des quatre processus, et le temps nécessaire au processus maître pour envoyer et recevoir les sous-grilles aux trois autres processus. Les résultats sont présentés dans le Tableau VII-5 et le Tableau VII-6.

Nombre de colonnes	Taille Frontière (Ko)	Nombre d'échanges de frontières	Temps d'échanges des frontières (sec.)		Différence (sec.)	Gain (%)
			MPI-MPC2	MPI-MPC3		
3200	12,5	1,5	4,1	3,5	0,6	15%
6400	25	1,5	8,1	7,0	1,1	14%
12800	50	1,5	15,8	13,9	1,9	12%
25600	100	1,5	31,0	26,9	4,1	13%
51200	200	1,5	63,4	55,3	8,1	13%

Tableau VII-5 : Les temps de communication pour échanger les frontières

Nous avons présenté, dans le Tableau VII-5, les temps moyens de communication liés aux échanges des frontières pour les processus « extérieurs » (P1 et P4 sur la Figure VII-3) et les processus « intérieurs » (P2 et P3). Les processus intérieurs échangent deux frontières lors de chaque itération : une avec le voisin inférieur et une avec le voisin supérieur. Les processus extérieurs ne réalisent qu'un seul échange avec leur unique voisin. Ainsi, le nombre d'échanges moyen par itération est de 1,5. L'avant dernière colonne du tableau correspond à la différence entre le temps moyen d'échanges des frontières avec MPI-MPC2 et celui avec MPI-MPC3. Le temps gagné est de l'ordre de la seconde pour un nombre de colonnes égal à 3200 et 6400, ce qui explique pourquoi la redistribution de la mémoire n'est pas avantageuse pour cette dimension de matrice. La dernière colonne du tableau présente le gain en pourcentage sur les temps de communication de MPI-MPC3 par rapport à MPI-MPC2. Nous constatons qu'il est environ de 13% ce qui correspond aux résultats présentés au chapitre VI avec un ping-pong.

Nombre de colonnes	Taille de chaque sous-grille (Ko)	Temps pour émettre/recevoir les sous-grilles (msec.)		Différence (msec.)	Gain (%)
		MPI-MPC2	MPI-MPC3		
3200	187,5	28,8	25,4	3,4	12%
6400	375	60,6	53,8	6,8	11%
12800	750	116,5	103,3	13,2	11%
25600	1500	232,9	206,0	26,9	12%
51200	3000	468,6	416,2	52,4	11%

Tableau VII-6 : Les temps de communications des sous-grilles

Le Tableau VII-6 présente le temps nécessaire au processus P1 pour envoyer les trois sous-grilles aux processus P2, P3 et P4 avant de commencer le calcul et pour les recevoir une fois le calcul terminé. Les temps sont donnés en milli-secondes : le temps lié à l'envoi et la réception des sous-grilles est faible comparé au temps d'échanges des frontières. Nous constatons que le gain se situe entre 11% et 12% ce qui est cohérent avec les résultats précédents.

VII.4. Conclusion

Nous avons choisi deux applications réelles pour confirmer les conclusions générales qui ont pu être tirées de la comparaison des trois implémentations de MPI dans le cas du ping-pong :

- ✓ une résolution d'un système linéaire par la méthode de Gauss qui utilise une implémentation parallèle de la méthode CESTAC permettant de prendre en compte les erreurs d'arrondi dans les calculs flottants,
- ✓ une résolution de l'équation de Laplace en parallèle par la méthode de Jacobi.

Comme ces deux applications utilisent un schéma de communication simple et reproductible, nous avons pu analyser les différences de performances entre nos trois implémentations de MPI. La particularité de la première application est qu'elle échange beaucoup de messages d'une taille de 8 octets et qu'il suffit d'augmenter la taille du système linéaire pour augmenter le nombre de communications. Elle nous a permis de comparer nos implémentations MPI-MPC1 et MPI-MPC2. Pour cette application, nous n'avons pas réalisé de mesures avec MPI-MPC3 car elle n'apporte aucune amélioration par rapport à MPI-MPC2 pour les messages d'une taille inférieure à 4Ko. Les mesures réalisées ont confirmé l'écart de latence mesuré entre MPI-MPC1 et MPI-MPC2 à l'aide d'un ping-pong MPI avec une application échangeant plusieurs millions de messages. Nous avons utilisé la parallélisation de la résolution de l'équation de Laplace pour comparer nos implémentations MPI-MPC2 et MPI-MPC3. Cette application échange de nombreux messages de grande taille. Nous avons choisi les tailles du problème afin de fixer le nombre de communications et d'augmenter la

taille des messages échangés entre les différents processus de l'application. Les mesures réalisées nous ont permis de constater que le gain sur les temps de communications, lors de l'échange de messages de grandes tailles, de MPI-MPC3 par rapport à ceux de MPI-MPC2 est de l'ordre de 12%, ce qui a confirmé les mesures de débit réalisées avec un ping-pong. Nous avons également constaté que le *speed-up* concernant le temps d'exécution global de l'application parallèle par rapport à celui de l'application séquentielle est proche de quatre lorsque la parallélisation est réalisée sur quatre processeurs.

Ces mesures sur des applications réelles nous ont également permis de vérifier la robustesse de nos trois implémentations de MPI lors de l'échange fréquent de messages entre un nombre de processeurs supérieur à deux.

Chapitre VIII : Conclusions et perspectives

Notre objectif était de construire un ensemble de couches de communication, implantant la bibliothèque de communication MPI, au-dessus d'une primitive d'écriture en mémoire distante, dans le contexte des machines parallèles de type « grappe de PCs ». Dans ce cadre, notre souci constant a été de remonter au niveau applicatif MPI les très bonnes performances du matériel réseau utilisé.

Pour atteindre cet objectif, nous avons commencé par étudier les caractéristiques de la primitive d'écriture distante et les contraintes qu'elle nous imposait. L'utilisation de tampons décrits en adresses physiques et la sémantique de la primitive d'écriture distante obligeant l'émetteur à connaître la localisation en mémoire physique des tampons distants ont été les deux points sensibles de notre travail.

Notre premier apport a été de définir une API générique d'écriture en mémoire distante, appelée RDMA, permettant de porter facilement notre implémentation de MPI sur n'importe quelle plate-forme matérielle disposant d'une primitive d'écriture en mémoire distante. Nous avons ainsi formalisé les services fournis par la primitive d'écriture distante pour masquer les particularités des réseaux utilisant ce mécanisme.

Nous avons décrit une première implémentation de MPI au-dessus de l'API RDMA : MPI-MPC1. Deux protocoles ont été implantés. Le premier utilise des tampons intermédiaires, contigus en mémoire physique et projetés dans la mémoire virtuelle de l'application. L'avantage de ce protocole est d'éviter les opérations de verrouillage et de traduction d'adresse des tampons d'émission et de réception de l'application. L'inconvénient est qu'il utilise des recopies intermédiaires des données sur l'émetteur et le récepteur. Le deuxième protocole est un protocole de rendez-vous permettant des transmissions de type « zéro-copie ». Le récepteur utilise un message de contrôle pour transmettre la description en mémoire physique du tampon de réception de l'application. Ce protocole nécessite un verrouillage en mémoire physique et une traduction d'adresse des tampons d'émission et de réception de l'application. Comme prévu, le premier protocole s'est avéré efficace pour la transmission des messages de petite taille alors que le protocole de rendez-vous est plus performant pour les messages de taille importante. Nous avons également montré que ces deux modes de transfert permettaient de respecter la sémantique des primitives de communication point à point définies par le standard MPI. Enfin, cette première implémentation a prouvé qu'il est possible d'obtenir des performances assez proches des performances obtenues sur le réseau Myrinet (MPI/BIP, MPI/GM, MPI/PM), alors que les services fournis par le réseau sont assez différents : notre implémentation MPI n'utilise qu'une primitive d'écriture distante en mémoire physique, et ne suppose pas l'existence d'une primitive de réception. Par opposition, le contrôleur réseau de Myrinet étant programmable, des bibliothèques de communication PM, GM ou BIP peuvent y stocker, dans une table, la correspondance entre les adresses virtuelles et les adresses physiques. Ces protocoles utilisent même un cache logiciel, accessible directement par le contrôleur réseau, sauf dans le cas de BIP. Cela permet non seulement d'accélérer la traduction d'adresse mais aussi de réaliser des communications sur des canaux virtuels (un *tag* dans BIP, un *port* dans GM ou l'adresse virtuelle du tampon de réception dans

PM). L'émetteur n'a donc pas besoin de connaître les adresses physiques du tampon de réception de l'application, ce qui n'est pas le cas avec l'API RDMA. Par ailleurs, BIP, GM et PM fournissent des primitives de réception qui facilitent une signalisation par scrutation.

Cependant, les performances de MPI-MPC1 étant tout de même assez éloignées des performances « brutes » de la plate-forme matérielle utilisée, une analyse nous a montré que les limites provenaient principalement des appels système et de l'utilisation d'un mécanisme de signalisation par interruption matérielle. Nous avons proposé une deuxième implémentation (MPI-MPC2) dont l'objectif était l'élimination, durant les phases de communication, des interruptions matérielles et des appels système, à l'exception de ceux résultants des traductions d'adresse (virtuelle/physique). Nous avons montré comment passer d'une primitive d'écriture distante se trouvant dans le noyau et utilisant une signalisation par interruption, à une écriture distante en mode utilisateur reposant sur une signalisation par scrutation. Nous avons résolu le problème du partage des ressources réseau entre différents processus se trouvant sur un même nœud de calcul. Nous avons proposé un mécanisme de signalisation par scrutation et nous avons analysé les problèmes qu'elle pouvait engendrer. La scrutation s'est avérée très efficace sur notre plate-forme matérielle, à condition de ralentir la boucle de scrutation pour ne pas surcharger les transferts sur le bus PCI. Le fait de sortir les couches de communication du système d'exploitation a permis de rendre MPI-MPC2 nettement plus portable que MPI-MPC1. Les mesures de performances de MPI-MPC2 ont montré que l'utilisation d'une primitive d'écriture distante en mode utilisateur et sans interruption est très efficace pour les petits messages : avec MPI-MPC2, la latence logicielle est du même ordre de grandeur que la latence matérielle. En revanche, le débit maximum de MPI-MPC2 reste très inférieur au débit utile maximum de notre plate-forme matérielle, du fait de la discontinuité en mémoire physique des tampons de l'application.

Notre troisième implémentation (MPI-MPC3) visait à résoudre le problème de discontinuité en mémoire physique des tampons de l'application. Nous avons analysé différentes solutions qui ne résolvaient que partiellement le problème avant de proposer un mécanisme qui permet de se ramener à une situation où les segments mémoire des processus constituant l'application correspondent à des zones contiguës en mémoire physique. Le principe est d'intervenir entre le chargement des processus en mémoire et le début de leur exécution pour leur allouer statiquement toute la mémoire physique disponible sur chacun des nœuds de calcul. Nous avons montré que cette redistribution de la mémoire d'un processus pouvait se faire sans modifier, ni le système d'exploitation, ni la librairie C, ni l'application. Notre méthode utilise une librairie dynamique écrite en C++ à laquelle l'application doit être liée lors de la compilation. La redistribution de la mémoire permet non seulement, d'éliminer les opérations de verrouillage et de traduction d'adresse des tampons de l'application mais aussi, de réduire la taille du message *RSP* dans le protocole de rendez-vous et de limiter le nombre d'écritures distantes nécessaires à la transmission des données de l'application. Les performances mesurées avec MPI-MPC3 montrent que la redistribution de la mémoire permet d'atteindre un débit maximum au niveau

applicatif MPI très proche du débit maximum utile de la plate-forme utilisée. En revanche, elle n'a aucune influence sur la latence.

La plate-forme matérielle utilisée pour nos expérimentations est la machine MPC du LIP6. La Figure VIII-1 résume les performances mesurées en termes de latence et de débit maximum avec des processeurs cadencés à 350MHz. Les performances « brutes » de la plate-forme matérielle MPC ont été mesurées par Alexandre Fenÿo [Fenÿo,2001] : la latence est de 5µs et le débit utile maximum de 494Mbits/s. Au niveau applicatif MPI, avec MPI-MPC3, nous avons obtenu une latence de 15µs et un débit maximum de 485Mbits/s : nous avons atteint un de nos objectifs consistant à restituer à l'application les très bonnes performances du matériel réseau utilisé.

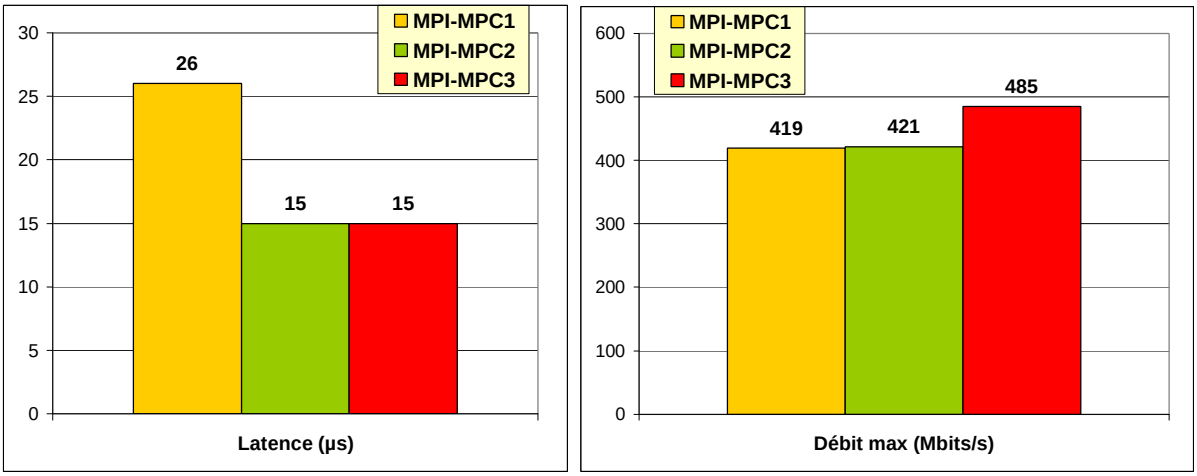


Figure VIII-1 : Latence et débit maximum de nos trois implémentations

La latence logicielle d'un transfert au niveau applicatif, résultant de la traversée de nos couches de communication, est de 10µs sur des processeurs cadencés à 350MHz (soit 3500 cycles), ce qui constitue un excellent résultat. Celle-ci étant proportionnelle à la fréquence du processeur, elle serait inférieure à la latence matérielle sur un processeur cadencé à 1GHz.

La Figure VIII-2 représente les gains cumulés de MPI-MPC2 et MPI-MPC3 par rapport à MPI-MPC1, pour des tailles de message allant de 1 octet à 4Mo.

Nous avons montré par nos expérimentations que l'influence des appels système et des interruptions matérielles est importante pour le transfert des messages de petite taille alors que le problème de la discontinuité des tampons de l'application en mémoire physique intervient principalement pour le transfert des messages de grande taille.

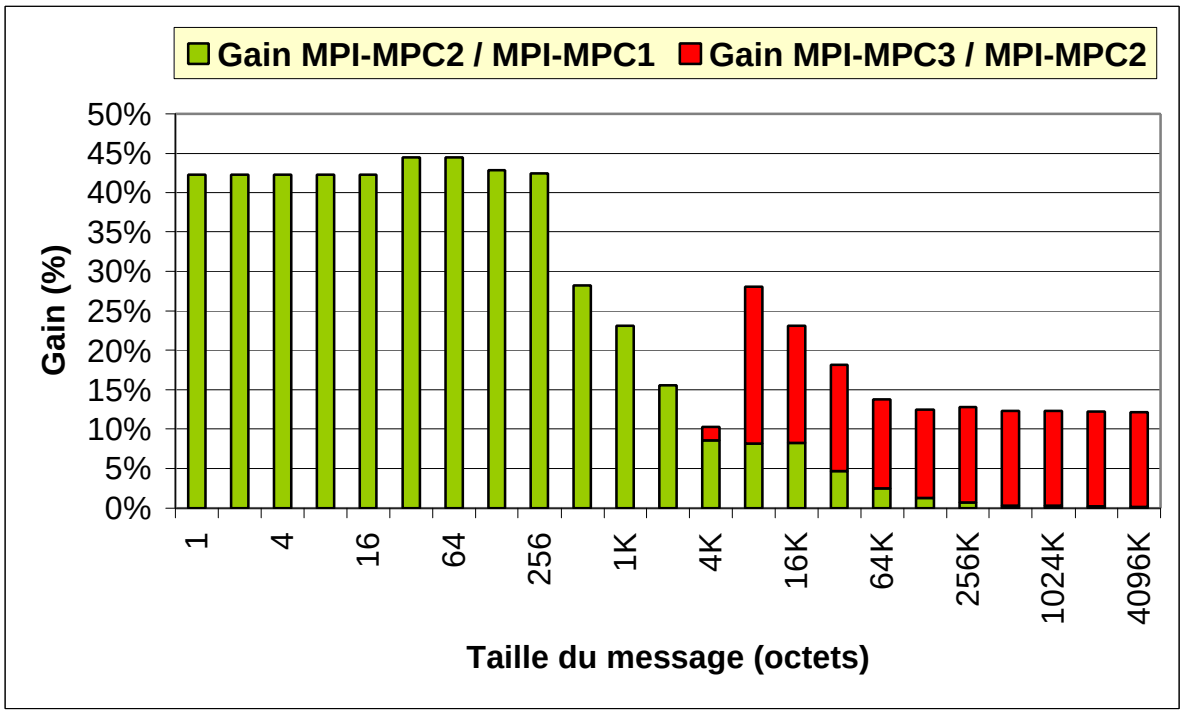


Figure VIII-2 : Gains cumulés de MPI-MPC2 et MPI-MPC3

Enfin, nous avons confirmé ces résultats sur deux applications réelles : la résolution d'un système linéaire utilisant une bibliothèque parallèle permettant la prise en compte des erreurs d'arrondi dans les calculs flottants et la résolution de l'équation de Laplace en parallèle par la méthode itérative de Jacobi. La première application a tourné sur trois processeurs et nous a permis de comparer les performances de MPI-MPC1 et MPI-MPC2 pour des applications qui échangent de nombreux messages de petite taille (8 octets) : les mesures sur notre plate-forme expérimentale ont confirmé l'écart de latence entre les deux implémentations. La deuxième application s'est exécutée sur quatre processeurs. Elle échange des messages de grande taille entre les différents processus et nous a permis de confirmer les performances en termes de débit de MPI-MPC3 par rapport à celles de MPI-MPC2. Enfin, les exécutions de ces deux applications ont montré la robustesse de nos trois implémentations de MPI dans le cadre d'applications réelles.

Les trois implémentations de MPI décrites dans ce manuscrit ne permettent pas de lancer plusieurs tâches d'une même application sur un même nœud de calcul. Cependant, les mécanismes décrits aux chapitres V et VI tiennent compte de cette possibilité et sont applicables à ce cas de figure.

Nous n'avons pas abordé les problèmes de recouvrement calcul/communication dans ce manuscrit. Ils sont abordés dans [Westrelin,2001] et [Chaussumier,2001]. Pour résoudre ces problèmes, il serait par exemple intéressant de passer à une implémentation *multi-threads* de MPI dans laquelle un ou plusieurs *threads* seraient en charge de la gestion des communications.

Les implémentations de MPI décrites sont basées sur le standard MPI-1 [MPI,1994]. Le standard MPI-2 [MPI,1997] offre de nouvelles fonctionnalités. L'architecture de MPICH est en cours de redéfinition [Gropp,2002] pour tenir compte de ces nouvelles fonctionnalités, des travaux récents sur les systèmes de communication de niveau utilisateur, et de l'évolution du matériel réseau. Cette nouvelle architecture intégrerait, entre autres, la création dynamique de tâches et des primitives de communication unidirectionnelle (*one-sided communications*) haut niveau. La sémantique de certaines primitives de communication unidirectionnelles est très proche de celle de la primitive d'écriture distante. On pourrait espérer une amélioration des performances en intégrant notre travail dans cette nouvelle architecture de MPICH. En revanche, la création dynamique de tâches est incompatible avec le mécanisme de redistribution de la mémoire que nous avons proposé, à moins de réserver de la mémoire physique pour d'éventuelles tâches créées dynamiquement.

Annexe A : Les traductions d’adresse

Sommaire

Nous rappelons comment la gestion des adresses mémoire est réalisée sur une architecture matérielle utilisant un processeur Intel 80x86. Nous décrivons comment nous avons implanté les traductions d'adresses virtuelles en adresses physiques dans le système d'exploitation LINUX. Notre but n'est pas de faire une description détaillée des mécanismes de traduction d'adresse mais plutôt une description simplifiée (pour plus de détails, se référer à [Bovet,2001]).

A.1. Principe général

Il existe trois types d'adresse sur les architectures Intel 80x86 :

- ✓ Les adresses logiques : il s'agit des adresses manipulées par le noyau ou les processus. Elle est composée d'un identificateur de segment sur 16 bits et d'un déplacement dans le segment sur 32 bits.
- ✓ Les adresses linéaires : elles sont représentées sur 32 bits ce qui permet d'adresser 4 Go de mémoire.
- ✓ Les adresses physiques : elles sont représentées sur 32 bits et permettent d'adresser les cellules de mémoire physique de la machine.

La segmentation :
Sur l'architecture 80x86, l'accès à la mémoire est effectué en utilisant des segments. Une adresse logique est composée d'un identificateur de segment (appelé *segment selector*) et d'un déplacement dans le segment. Les descripteurs de segments sont stockés dans une table des segments, appelée GDT (*Global Descriptor Table*) et stockée en mémoire centrale. Un descripteur de segment est composé de huit octets. Il contient l'adresse de base du segment sur 32 bits ainsi que des informations caractérisant le segment. Pour obtenir l'adresse linéaire à partir de l'adresse logique, il faut : (1) retrouver le descripteur de segment à partir de l'identificateur de segment et de l'adresse de la table des segments ; (2) extraire l'adresse de base du segment à partir du descripteur de segment ; (3) additionner cette adresse de base au déplacement contenu dans l'adresse logique pour obtenir l'adresse linéaire.

La pagination :
Il s'agit du mécanisme pour déduire l'adresse physique à partir de l'adresse linéaire. La mémoire physique est divisée en pages de 4Ko. Pour retrouver l'adresse physique de la page associée à une adresse linéaire, il faut parcourir une table de translation à deux niveaux sur les processeurs 32 bits (cf. Figure A-1). Le premier niveau est le « catalogue des pages », le deuxième niveau est la « table des pages ». L'adresse linéaire se décompose en trois parties : le champ « catalogue » sur 10 bits, le champ « table » sur 10 bits et le champ « déplacement » sur 12 bits. Le champ catalogue permet de retrouver l'entrée associée à l'adresse linéaire dans le « catalogue des pages » à partir de la valeur du registre cr3 qui contient l'adresse en mémoire centrale du catalogue des pages. Le contenu de cette entrée contient l'adresse de la table des pages correspondante. En y ajoutant le champ table, on trouve l'entrée correspondante

dans la table des pages qui contient l'adresse physique de la page associée à l'adresse linéaire. Le champ déplacement est ajouté à l'adresse de la page pour en déduire l'adresse physique correspondant à l'adresse linéaire.

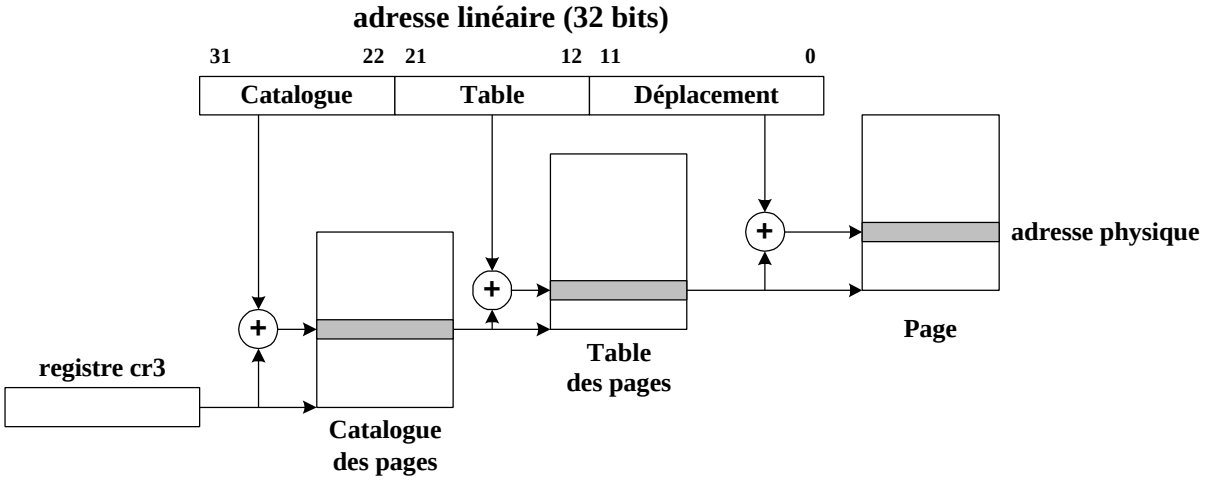


Figure A-1 : La pagination dans les processeurs 80x86

A.2. Implantation sous LINUX

Nous décrivons comment l'architecture 80x86 est supportée par LINUX et comment nous avons réalisé les traductions d'adresse dans notre implémentation de MPI.

La segmentation :
LINUX n'utilise la segmentation que pour distinguer les protections associées à chaque segment. Tous les descripteurs de segment utilisent une adresse de base égale à zéro. Ainsi, l'adresse logique est égale à l'adresse linéaire. Nous l'appelons « adresse virtuelle ». Les adresses manipulées par les processus et le noyau sont des adresses virtuelles.

La pagination :
LINUX gère la mémoire centrale et les tables de pages. Ces dernières sont utilisées pour convertir les adresses virtuelles en adresses physiques. Il implémente une gestion de la mémoire qui est largement indépendante du processeur sur lequel il s 'exécute. En fait, la gestion mémoire implémentée par LINUX considère qu'elle dispose d'une table des pages à trois niveaux pour supporter les architectures 64 bits :

- le catalogue global des pages (*page global directory*) dont les entrées contiennent les adresses des catalogues intermédiaires,
- les catalogues intermédiaires des pages (*page middle directory*) dont les entrées contiennent les adresses des tables de pages,
- les tables de pages (*page table*) dont les entrées contiennent les adresses de pages mémoire contenant le code ou les données utilisés par les processus ou le noyau.

Dans le cas de l'architecture 80x86 32 bits, LINUX considère qu'un catalogue intermédiaire ne contient qu'une seule entrée.

Les régions de mémoire virtuelle :
Un processus est constitué de plusieurs zones de mémoire virtuelle. Une région mémoire est décrite par la structure de données `vm_area_struct` déclarée dans le fichier d'en-tête <`linux/mm.h`>. Ces régions mémoire sont chaînées. LINUX maintient un descripteur de l'espace d'adressage associé à chaque processus. Il est accessible par le champ `mm` contenu dans le descripteur du processus. La structure de données `mm_struct`, déclarée dans le fichier d'en-tête <`linux/sched.h`>, définit le format du descripteur de l'espace d'adressage. L'adresse physique du catalogue global des pages utilisé par le processus est contenu dans le champ `pgd` de la structure `mm_struct`.

Nous présentons ci-dessous le code simplifiée de la fonction que nous avons écrite pour réaliser la traduction d'une adresse virtuelle en une adresse physique. Il implémente les opérations symbolisées sur la Figure A-1. Nous avons enlevé du code certaines vérifications concernant les entrées des tables de pages parcourues. Les macros et fonctions utilisées sont déclarées dans <`asm/page.h`> et <`asm/pgtable.h`>.

```
// conversion de l'adresse virtuelle « addr » en une adresse physique
// virt2phys retourne l'adresse physique correspondante ou 0 si la page
// correspondante n'est pas en mémoire physique
unsigned long virt2phys(struct mm_struct *mm,unsigned long addr)
{
  pgd_t * mypgd; // pointeur sur une entrée du catalogue global des pages
  pmd_t * mypmd; // pointeur sur une entrée d'un catalogue intermédiaire
  pte_t * mypte; // pointeur sur une entrée d'une table des pages
  unsigned long offset;

  // offset contient le déplacement dans la page
  offset = addr & ~PAGE_MASK;
  // addr est aligné sur le début d'une page
  addr &= PAGE_MASK;

  mypgd = pgd_offset(mm,addr);
  mypmd = pmd_offset(mypgd,addr);
  mypte = pte_offset(mypmd,addr);

  if (!pte_present(*mypte)) return 0;

  return __pa(pte_page(*mypte)) + offset;
}
```

Pour traduire une zone de mémoire virtuelle (`addr, size`), il faut appeler la fonction `virt2phys` autant de fois qu'il y a de pages mémoire dans la zone.

Annexe B : L'API RDMA sur la machine MPC

Sommaire

Cette annexe présente l'implantation de l'API RDMA sur la machine MPC, notre plate-forme expérimentale.

B.1. L'API RDMA

L'API RDMA a été décrite au chapitre IV. Ses primitives sont rappelées dans le tableau ci-dessous :

a) La primitive d'écriture distante
❶ `RDMA_SEND(nsrc, ndst, plad, prad, len, ctrl, sid, rid, ns, nr)` Rôle : écriture en mémoire distante d'un tampon contigu en mémoire physique `nsrc` : numéro du nœud local sur lequel les données sont lues `ndst` : numéro du nœud distant sur lequel les données sont écrites `plad` : adresse physique locale des données à transmettre (*physical local address*) `prad` : adresse physique distante du tampon de réception (*physical remote address*) `len` : taille des données à émettre `ctrl` : booléen indiquant s'il s'agit d'un message *CTRL* ou d'un message *DATA* `sid` : identificateur de la requête d'émission `rid` : identificateur de la requête de réception `ns` : booléen indiquant si la fin d'émission doit être signalée sur l'émetteur (*notify sent*) `nr` : booléen indiquant si la fin de réception doit être signalée sur le récepteur (*notify recv*)
b) Les fonctions de notification
❷ `RDMA_SENT_NOTIFY(ctrl, sid)` Rôle : signaler la fin d'émission d'un message (contrôle ou données) `ctrl` : booléen indiquant s'il s'agit d'un message *CTRL* ou d'un message *DATA* `sid` : identificateur de la requête d'émission
❸ `RDMA_RECV_NOTIFY(nsrc, ctrl, rid)` Rôle : signaler la fin de réception d'un message (contrôle ou données) `nsrc` : numéro du nœud distant duquel proviennent les données `ctrl` : booléen indiquant s'il s'agit d'un message *CTRL* ou d'un message *DATA* `rid` : identificateur de la requête de réception
c) La signalisation par scrutation (optionnelle)
❹ `RDMA_NET_LOOKUP(blocking)` Rôle : scrutation des événements réseau et appel des primitives RDMA_SENT_NOTIFY ou RDMA_RECV_NOTIFY le cas échéant `blocking` : booléen indiquant si la scrutation doit être bloquante

Tableau B-1 : L'API RDMA

B.2. L'API PUT

Implanter l'API RDMA sur la machine MPC consiste à réaliser l'interface avec l'API PUT qui constitue la couche de communication bas niveau de la plate-forme. PUT est l'interface entre les applications et le contrôleur réseau de la machine MPC. Les principales primitives de PUT sont présentées dans le Tableau B-2 :

a) La primitive d'écriture distante
❶ `PUT_ADD_ENTRY(ndst, plad, prad, len, mi, ns, nr, lmp)` Rôle : écriture en mémoire distante d'un tampon contigu en mémoire physique par ajout d'une entrée dans la LME `ndst` : numéro du nœud distant sur lequel les données sont écrites `plad` : adresse physique locale des données à transmettre (*physical local address*) `prad` : adresse physique distante du tampon de réception (*physical remote address*) `len` : taille des données à émettre `mi` : identifiant du message de l'application (*message identifier*) `ns` : booléen indiquant si la fin d'émission doit être signalée sur l'émetteur (*notify sent*) `nr` : booléen indiquant si la fin de réception doit être signalée sur le récepteur (*notify recv*) `lmp` : booléen indiquant si le tampon à émettre est le dernier tampon constituant le message de l'application (*last message page*)
b) Les fonctions de notification
❷ `PUT_SENT_NOTIFY(ndst, plad, prad, len, mi)` Rôle : signaler la fin d'émission d'un message de l'application `ndst` : numéro du nœud distant sur lequel les données sont écrites `plad` : adresse physique locale du dernier tampon transmis pour un message donné `prad` : adresse physique distante du dernier tampon de réception pour un message donné `len` : taille des données du dernier tampon `mi` : identifiant du message émis (*message identifier*)
❸ `PUT_RECV_NOTIFY(mi)` Rôle : signaler la fin de réception d'un message de l'application `mi` : identifiant du message reçu (*message identifier*)
c) La signalisation par scrutation (optionnelle)
❹ `PUT_FLUSH_LME()` Rôle : scrutation des fins d'émission et appel de la primitive PUT_SENT_NOTIFY le cas échéant ; cette fonction retourne le nombre de fins d'émission signalées
❺ `PUT_FLUSH_LMR()` Rôle : scrutation des fins de réception et appel de la primitive PUT_RECV_NOTIFY le cas échéant ; cette fonction retourne le nombre de fins de réception signalées

Tableau B-2 : L'API PUT

PUT permet de distinguer la notion de message de l'application de la notion de tampon contigu en mémoire physique : un message est constitué d'un ou plusieurs tampon(s) contigu(s) en mémoire physique. L'identifiant d'un message (`mi`) est un champ de 24 bits dont l'allocation est à la charge de l'application utilisatrice de PUT. La seule contrainte imposée à l'application pour l'attribution du `mi` est de garantir que deux

messages circulant sur le réseau HSL à un instant donné ne peuvent pas avoir le même identifiant. Les champs `ndst`, `prad`, `len`, `mi` et `nr` sont transmis dans chaque paquet du réseau HSL.

B.3. L'interface RDMA/PUT

La primitive d'écriture distante :
Un appel à la primitive `RDMA_SEND` se traduit par un appel à la primitive `PUT_ADD_ENTRY`. Les paramètres `ndst`, `plad`, `prad`, `len`, `ns` et `nr` sont transmis tels quels à `PUT_ADD_ENTRY`. Le champ `lmp` est positionné à 1 si `ns` et `nr` sont à 1 : dans la couche RDMA, le dernier tampon contigu en mémoire physique constituant un message est signalé en émission et en réception. Les paramètres `nsrc`, `ctrl`, `sid` et `rid` de `RDMA_SEND` sont stockés dans le `mi`, comme cela est présenté sur la Figure B-1. Le champ `ctrl` utilise un seul bit, les champs `sid` et `rid` utilisent chacun quatre bits et le champ `nsrc` utilise quinze bits.

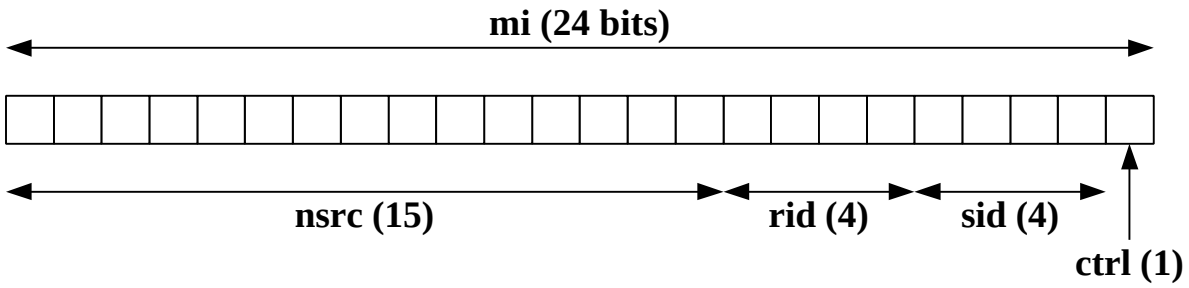


Figure B-1 : L'utilisation du mi par l'API RDMA

Les fonctions de notification :
Les fonctions `PUT_SENT_NOTIFY` et `PUT_RECV_NOTIFY` sont des fonctions de rappel enregistrées par la couche RDMA lors de l'initialisation de l'application. Elles sont appelées, soit par le gestionnaire d'interruptions de la couche PUT, soit par les fonctions de signalisation par scrutation. La primitive `PUT_SENT_NOTIFY` extrait du `mi` les champs `ctrl` et `sid` avant d'appeler `RDMA_SENT_NOTIFY`. De la même façon, `PUT_RECV_NOTIFY` extrait du `mi` les champs `nsrc`, `ctrl` et `rid` avant d'appeler `RDMA_RECV_NOTIFY`.

La signalisation par scrutation :
Au niveau de la couche PUT, la signalisation par scrutation consiste à parcourir les tables LME (en émission) et LMR (en réception). La primitive `RDMA_NET_LOOKUP` appelle les primitives `PUT_FLUSH_LME` et `PUT_FLUSH_LMR`. Si le paramètre `blocking` vaut 1, `RDMA_NET_LOOKUP` appelle ces primitives jusqu'à ce qu'une fin d'émission ou de réception soit détectée.

Annexe C : Les primitives de communication point à point dans MPI

Sommaire

Nous présentons dans cette section les principales primitives de communication point à point définies dans le standard MPI [MPI,1994].

C.1. Les primitives de réception

❶	`MPI_Recv(buf, count, datatype, source, tag, comm, status)` Rôle : réception bloquante `buf` : adresse du tampon de réception des données `count` : nombre d'éléments à recevoir `datatype` : type MPI de chaque élément à recevoir `source` : numéro de la tâche émettrice `tag` : tag MPI du message `comm` : communicateur MPI `status` : status de la communication
❷	`MPI_Irecv(buf, count, datatype, source, tag, comm, request)` Rôle : réception non bloquante `buf` : adresse du tampon de réception des données `count` : nombre d'éléments à recevoir `datatype` : type MPI de chaque élément à recevoir `source` : numéro de la tâche émettrice `tag` : tag MPI du message `comm` : communicateur MPI `request` : identificateur de la requête de réception

Tableau C-1 : Les primitives de réception point à point

C.2. Les primitives d'émission

❶	`MPI_Send(buf, count, datatype, dest, tag, comm)` Rôle : émission bloquante en mode *standard* `buf` : adresse du tampon d'émission `count` : nombre d'éléments à envoyer `datatype` : type MPI de chaque élément à envoyer `dest` : numéro de la tâche destinataire `tag` : tag MPI du message `comm` : communicateur MPI
❷	`MPI_Ssend(buf, count, datatype, dest, tag, comm)` Rôle : émission bloquante en mode *synchrone* (paramètres identiques à `MPI_Send`)
❸	`MPI_Rsend(buf, count, datatype, dest, tag, comm)` Rôle : émission bloquante en mode *ready* (paramètres identiques à `MPI_Send`)
❹	`MPI_Bsend(buf, count, datatype, dest, tag, comm)` Rôle : émission bloquante en mode *bufferisé* (paramètres identiques à `MPI_Send`)

❺	`MPI_Isend(buf, count, datatype, dest, tag, comm, request)` Rôle : émission non bloquante en mode *standard* `buf` : adresse du tampon d'émission `count` : nombre d'éléments à envoyer `datatype` : type MPI de chaque élément à envoyer `dest` : numéro de la tâche destinataire `tag` : tag MPI du message `comm` : communicateur MPI `request` : identificateur de la requête d'émission
❻	`MPI_Issend(buf, count, datatype, dest, tag, comm, request)` Rôle : émission non bloquante en mode *synchrone* (paramètres identiques à `MPI_Isend`)
❼	`MPI_Irsend(buf, count, datatype, dest, tag, comm, request)` Rôle : émission non bloquante en mode *ready* (paramètres identiques à `MPI_Isend`)
❽	`MPI_Ibsend(buf, count, datatype, dest, tag, comm, request)` Rôle : émission non bloquante en mode *bufferisé* (paramètres identiques à `MPI_Isend`)

Tableau C-2 : Les primitives d'émission point à point

Avant de faire une émission en mode *bufferisé*, l'application doit appeler la primitive `MPI_Bsend_init` et réserver un tampon pour la préparation des émissions à l'aide de la primitive `MPI_Buffer_attach`.

C.3. Les primitives de complétion

Les primitives de complétion permettent d'attendre ou de tester la terminaison d'une requête d'émission ou de réception non bloquante.

❶	`MPI_Wait(request, status)` Rôle : attente de la terminaison d'une émission ou d'une réception `request` : identificateur de la requête attendue `status` : status de la communication
❷	`MPI_Test(request, flag, status)` Rôle : teste la terminaison d'une émission ou d'une réception `request` : identificateur de la requête attendue `flag` : vrai si la requête est terminée `status` : status de la communication

Tableau C-3 : Les primitives de complétion

Bibliographie

[Agerwala,1995] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. *SP2 system architecture*. IBM system journal, 34(2):152-184, 1995. *http://www.research.ibm.com/journal/sj/342/agerwala.html*

[Anderson,1995] T. Anderson, D. Culler, and D. Patterson. *A case for Networks of Workstations*. IEEE Micro, 1995. *http://now.cs.berkeley.edu/Case/case.html*

[Araki,1998] S. Araki, et al. *User-Space Communication: A Quantitative Study*. In Supercomputing (SC'98), Orlando, FL, Nov. 1998.

[ATM,1995] ATM Forum. *ATM User Level Network Interface Specification*. Prentice Hall, NJ, June 1995.

[Aumage_1,2001] O. Aumage, L. Bougé, J-F. Méhaut, and R. Namyst. *Madeleine II: A portable and efficient communication library for high-performance cluster computing*. In Parallel Computing, March 2001.

[Aumage_2,2001] O. Aumage, G. Mercier, and R. Namyst. *MPICH/Madeleine: a true multi-protocol MPI for high-performance networks*. In Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), p51, San Francisco, April 2001. IEEE. Extended proceedings in electronic form only.

[Bailey,1994] D. Bailey et al. *The NAS Parallel Benchmarks*. RNR Technical Report RNR-94-007, NASA Ames Research Center, March 1994. *http://www.nas.nasa.gov*

[Baker,1996] M.A. Baker, G.C. Fox, and H.W. Yau. *Review of Cluster Management Software*. NHSE Review, May 1996. *http://www.nhse.org/NHSEreview/CMS*

[Barak,1999] A. Barak, I. Gilderman, and I. Metrik. *Performance of the communication layers of TCP/IP with the myrinet gigabit LAN*. In Computer Communications, July 1999.

[Beowulf] The Beowulf project. *http://www.beowulf.org*

[Bertozzi,1999] M. Bertozzi, F. Boselli, G. Conte, and M. Reggiani. *An MPI Implementation on the Top of the Virtual Interface Architecture*. In Proc. of Euro PVM/MPI, vol. 1167, p199-206, Barcellona, Sept. 1999. *http://www.ce.unipr.it/research/parma2/via/via.html*

[Bertozzi,2001] M. Bertozzi, M. Panella, and M. Reggiani. *Design of a VIA based communication protocol for LAM/MPI suite*. In Proc. of 9th Euromicro Workshop on Parallel and Distributed Processing PDP 2001, p27-33, Mantova, Italy, February 2001.
Available on *http://www.ce.unipr.it/people/bertozzi/publications*

[Bhoedjang,1998] R. Bhoedjang, T. Ruhl, and H.E. Bal. *User-Level Network Interface Protocols.* In IEEE Computer, 31(11):53-60, November 1998.

[BIP] The BIP project. *http://www.ens-lyon.fr/LIP/RESAM/index_bip.html*

[Blum,1996] J. M. Blum, T. M. Warschko, and W. F. Tichy. *PSPVM: implementing PVM on a high-speed interconnect for workstation clusters.* In LNCS, Ed., Proc. of EuroPVM'96, n°1156, p235-242, 1996.
http://www.ubka.uni-karlsruhe.de/vvv/ira/1996/17

[Blumrich,1994] M. Blumrich et al. *A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer*. In Proc. of the 21st Annual Symposium on Computer Architecture, p142-153, April 1994.

[Boden,1995] N. Boden et. al. *Myrinet : A Gigabit-per-Second Local-Area Network.* In IEEE Micro, 15(1):29-36, Feb. 1995, *http://www.myri.com*

[Bovet,2001] D. P. Bovet, M. Cesati. *Understanding the Linux Kernel*. O'REILLY, ISBN 0-596-00002-2 , 2001.

[Brightwell,2000] R. Brightwell, A.B. Maccabe. *Scalability Limitations of VIA-Based Technologies in Supporting MPI*. In Proc. of the Fourth MPI Developer's and User's Conference, March 2000.

[Buonadonna,1998] P. Buonadonna, A. Geweke, D. E. Culler. *An Implementation and Analysis of the Virtual Interface Architecture.* In Proc. of SC98, Orlando, Florida, Nov. 1998.

[Burns,1994] G. Burns, R. Daoud, J. Vaigl. *LAM: An Open Cluster Environment for MPI.* Proceedings of Supercomputing Symposium '94, p379-386. University of Toronto, 1994.

[Buyya,1999] R. Buyya(ed.). *High Performance Cluster Computing: Systems and Architectures*. Vol. 1, Prentice Hall PTR, ISBN 0-13-013784-7, NJ, USA, 1999.
R. Buyya(ed.). *High Performance Cluster Computing: Programming and Applications*. Vol. 2, Prentice Hall PTR, ISBN 0-13-013785-5, NJ, USA, 1999.

[CADNA] Laboratoire LIP6, thème ANP. *Le logiciel CADNA*.
http://www-anp.lip6.fr/cadna/index.html

[Cappello,1998] F. Cappello, O. Richard. *Architectures parallèles à partir de réseaux de stations de travail : réalités, opportunités, enjeux.* Calculateurs Parallèles, Hermes, 10(1), Septembre 1998.

[Cappello,1999] F. Cappello. *Evolution des mécanismes de communication par passage de messages dans les architectures parallèles*. TSI (Technique et Science Informatiques), Hermes, 18(1), Janvier 1999.

[Chandra,2000] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald. *Parallel Programming in OpenMP*. M. K. Publishers, October 2000. *http://www.openmp.org*

[Chaussumier,2001] F. Chaussumier. *Recouvrement des calculs et des communications : du matériel au logiciel*. Thèse de doctorat de l'Ecole Nationale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, 2001.

[Chelius,2001] G. Chelius. *Implementing virtual interface architecture on top of the GM message passing interface*. In IEEE Computer Society, editor, IEEE Int. Symposium on Cluster Computing and the Grid (CCGrid2001), Brisbane, Australia, May 2001.

[Chesneaux,1990] J-M. Chesneaux. *Study of the computing accuracy by using probabilistic approach.* In contribution to Computer Arithmetic and Self-Validating Numerical Methods, ed. C. Ulrich, (J.C. Baltzer), p19-30, 1990.

[Chesneaux,1995] J-M. Chesneaux, *L'arithmétique Stochastique et le Logiciel CADNA.* Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, Novembre 1995.

[Chun,1998] B. N. Chun, A. M. Mainwaring, D. E. Culler. *Virtual Network Transport Protocols for Myrinet*. In Proc. of Hot Interconnects V: a Symposium on High Performance Interconnects, Stanford University, August 1997 (Award Paper). Also appears in IEEE Micro, January/February 1998.

[Chung,2000] S. Chung et al. *A CC-NUMA prototype card for SCI-based PC clustering*. In Proc. IEEE Int. Conf. on Cluster Computing, CLUSTER'2000, 2000.

[Clark,1989] D. D. Clark, V. Jacobson, J. Romkey, H. Salwen. *An analysis of TCP processing overhead*. IEEE Communication mag., p23-29, June 1989.

[CM] Thinking Machines Corporation. *http://www.npac.syr.edu/nse/hpccsurvey/orgs/tmc/tmc.html*

[CRAY] Cray Systems. *http://www.cray.com/products/systems*

[Culler,1993] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. V. Eicken. *LogP : Towards a Realistic Model of Parallel Computation*. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1993.

[Culler,1999] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture : A Hardware/Software Approach*. M. K. Publishers, ISBN 1-55860-343-3, San Francisco, CA, 1999.

[Cyliax,2000] I. Cyliax. *Catching the PCI bus – A spin through the background*. Circuit Cellar Ink, 2000.

[Cypher,1993] R. Cypher, A. Ho, S. Konstantinidou, P. Messina. *Architectural Requirements of Parallel Scientific Applications with Explicit Communication*. Proceedings of the 20th annual international symposium on Computer architecture, p2-13, May 1993.

[Dimitrov,1998] R. Dimitrov, A. Skjellum. *Efficient MPI for Virtual Interface (VI) Architecture.* In Proc. of the 1999 Int. Conf. on Parallel and Distributed Processing Techniques and Applications, Vol. 6, p3094-3100, Las Vegas, Nevada, USA, June 1999.

[Dolphin] Dolphin interconnect. *http://www.dolphinics.com*

[Dongarra,1994] J. J. Dongarra. *Performance of Various Computers Using Standard Linear Equations Software*. Computer Science Department, University of Tennesee, CS-89-95, 1994.

[Dormanns,1997] M. Dormanns, W. Sprangers, H. Ertl, T. Bemmerl. *A Programming Interface for NUMA Shared-Memory Clusters*. In Proc. High Performance Computing and Networking (HPCN), p608-707, LNCS 1225, Springer, 1997.

[Dubnicki,1997] C. Dubnicki et al. *VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication*. In Proc. of Hot Interconnects V: a Symposium on High Performance Interconnects, p37-46, Stanford University, August 1997.

[Dunning,1998] D. Dunning et al. *The Virtual Interface Architecture*. In IEEE Micro, 18(2):66-76, March/April 1998.

[Eicken,1992] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. *Active Messages : A mechanism for integrated communication and computation.* In Proc. of the 19th Int'l Symp. on Computer Architecture, 20(2):256-266, Gold Coast, Australia, May 1992.
http://wwwbode.cs.tum.edu/~weissc/am.html

[Eicken,1995] T. Eicken, A. Basu, V. Buch, W. Vogels. *U-Net: A User-Level Network Interface for Parallel and Distributed Computing.* In Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP), p40-53, Copper Mountain, Colorado, December 1995.

[Felderman,1994] R. Felderman et al. *ATOMIC: A high speed local communication architecture*. Journal of High Speed Networks, 3(1):1-30, 1994.

[Fenÿo,2001] A. Fenÿo. *Conception et réalisation d'un noyau de communication bâti sur la primitive d'écriture distante, pour machines parallèles de type « grappe de PCs »*. Thèse de doctorat de l'Université de Paris VI, 2001.

[Flynn,1972] M. Flynn. *Some computer organizations and their effectiveness*. In IEEE Trans. Computers, 21, p948-960, 1972.

[Freeman,1992] T. L. Freeman, and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, ISBN 0-13-651597-5, October 1992.

[Gauchard,2000] D. Gauchard, T. Monteil, C. Fournié, B. Lecussan. *Communication mechanisms for Myrinet and MPC in the Parallel Environment LANDA-HSN*. In First Myrinet User Group Conference, Lyon, France, September 2000 .

[Giacomini,1999] F. Giacomini et al. *Low-level SCI software functional specification*. Esprit Project 23174, CERN, Geneva and Dolphin Interconnect Solutions AS, Oslo, Version 2.1.1, March 1999.

[Giannini,1998] L. A. Giannini, A. Chien. *A software architecture for global address space communication on clusters: Put/Get on Fast Messages*. In Proc. of High-Performance Distributed Computing Conference, 1998.

[Glück,2001] **O. Glück, A. Zerrouki, J.L. Desbarbieux, A. Fenyö, A. Greiner, F. Wajsbürt, C. Spasevski, F. Silva and E. Dreyfus. *Protocol and Performance Analysis of the MPC Parallel Computer*. In proceedings of 15th International Parallel & Distributed Processing Symposium (IPDPS'2001), p52, San Francisco, USA, April 23-27, 2001.**

[Glück,2002] **O. Glück, J-L. Lamotte, A. Greiner. *The influence of system calls and interrupts on the performance of a PC cluster using a remote DMA communication primitive*. To be presented at the 3rd Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'02), accepted paper number 101, Kanazawa, Japan, September 4-6, 2002.**

[GM,2000] Myricom. *The GM Message Passing System*. 2000. *http://www.myri.com/scs/GM/doc/gm.pdf*

[GM,2001] Myricom. *Performance Measurements of GM 1.4.* 2001. *http://www.myri.com/myrinet/performance/index.html*

[Greiner,1998] A. Greiner, P. David, J.L. Desbarbieux, A. Fenÿo, J.J. Lecler, F. Potter, V. Reibaldi, F. Wajsbürt, and B. Zerrouk. *La machine MPC*. In Hermes, editor, Calculateurs Parallèles, Réseaux et Systèmes répartis, Vol. 10, p71-84, Feb. 1998.

[Gropp,1994] W. Gropp, E. Lusk. *An abstract device definition to support the implementation of a high-level point-to-point message-passing interface*. Preprint MCS-P342-1193, Argonne National Laboratory, 1994.

[Gropp,1995] W. Gropp, E. Lusk. *MPICH working note: Creating a new MPICH device using the channel interface.* Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1995.

[Gropp,1996] W. Gropp, E. Lusk, N. Doss, A. Skjellum. *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*. In Parallel Computing, 22(6):789-828, September 1996.

[Gropp,2002] W. Gropp, E. Lusk. *MPICH Abstract Device Interface Version 3.3 Reference Manual.* Draft MCS-TM-00, Argonne National Laboratory, 2002. *http://www-unix.mcs.anl.gov/mpi/mpich/adi3*

[Gustavson,1992] D. B. Gustavson. *The Scalable Coherent Interface and related standards projects.* IEEE Micro, Vol. 12, No. 1, p10-22, Février 1992.

[HSL,1994] IEEE 1355. *IEEE 1355 Standard for Heterogeneous Interconnect (HIC) Low Cost Low Latency Scalable Serial Interconnect for Parallel System Construction*. IEEE Standards Department, August 1994.

[Hwang,1998] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. WCB/McGraw-Hill, NY, 1998.

[Ibel,1997] M. Ibel et al. *High-Performance Cluster Computing Using SCI*. In 7th International Workshop on SCI-based Low-cost/High-performance Computing (SCIzzL-7), Santa Clara, CA, March 1997.

[Koelbel,1994] C. Koelbel et al. *The High Performance Fortran Handbook.* The MIT Press, Massachusetts, 1994.

[LAM] LAM/MPI Parallel Computing. *http://www.lam-mpi.org*

[Lauria,1997] M. Lauria, A. Chien. *MPI-FM: High Performance MPI on Workstation Clusters.* J. of Parallel and Distributed Computing, 40(1):4-18, January 1997.

[Lauria,1998] M. Lauria, S. Pakin, A. Chien. *Efficient Layering for High Speed Communication: Fast Messages 2.x.* In Proc. 7th IEEE Int'l Symp, HPDC-7, Chicago, IL, July 1998.

[Lauria,1999] M. Lauria, S. Pakin, A. Chien. *Efficient Layering for High Speed Communication: the MPI over Fast Messages (FM) Experience*. In Cluster Computing. 2(2):107-116, Sep. 1999.

[LINPACK] The Linpack Benchmark. *http://www.netlib.org/linpack*

[Liu,1999] X. Liu. *Performance Evaluation of a Hardware Implementation of VIA*. Technical Report of Concurrent Systems Architecture Group (CSAG), Univ. of California, San Diego, 1999.
http://www-csag.ucsd.edu/papers/csag/external/VIA-Evaluation.pdf

[Mainwaring,1995] A. M. Mainwaring, D. E. Culler. *Active Message Applications Programming Interface and Communication Subsystem Organization*. Technical Report CSD-96-918, Computer Science Division, University of California, Berkeley, 1995.

[Martin,1997] R. P. Martin, A. M. Vahdat, D. E. Culler and T. E. Anderson. *Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture*. International Symposium on Computer Architecture, p85-97, Denver, CO, USA, 1997.

[Monteil,1995] T. Monteil, J. Garcia, P. Guyaux. *LANDA : une machine virtuelle parallèle.* Revue calculateurs parallèles, 7(2):119-137, 1995.

[Moore,1965] Gordon E. Moore. *Cramming more components onto integrated circuits.* In Electronics, Vol. 38, 1965.

[MPC] Le projet MPC. *http://mpc.lip6.fr*

[MPI] The MPI Forum. *http://www.mpi-forum.org*

[MPI,1994] Message Passing Interface Forum. *MPI: a Message-Passing Interface Standard*. May 5, 1994.
Available from *http://www.mpi-forum.org/docs/docs.html*

[MPI,1997] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. July 18, 1997.
Available from *http://www.mpi-forum.org/docs/docs.html*

[MPI-AM] Berkeley University. MPI performance on the NOW clusters.
http://now.cs.berkeley.edu/Fastcomm/MPI/performance

[M-VIA] National Energy Research Scientific Computing Center. *M-VIA: A High Performance Modular VIA for Linux.*
http://www.nersc.gov/research/FTG/via

[MVICH] National Energy Research Scientific Computing Center. *MPI for virtual interface architecture. http://www.nersc.gov/research/FTG/mvich*

[Myricom] Myricom, inc. *http://www.myri.com*

[Nevin,1996] N. Nevin. *The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster*. Technical Report OSC-TR-1996-4, Ohio Supercomputer Center, Columbus, Ohio, 1996.

[NOW] UC Berkeley NOW project. *http://now.cs.berkeley.edu*

[Ocarroll_1,1998] F. O'Carroll, A. Hori, H. Tezuka, Y. Ishikawa. *MPI-PM: Design and Implementation of Zero Copy MPI for PM*. Technical Report TR-97011, RWC, March 1998.

[Ocarroll_2,1998] F. O'Carroll, H. Tezuka, A. Hori, Y. Ishikawa. *The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network*. In Int. Conf. on Supercomputing, ACM SIGARCH ICS'98, p243-250, July 1998.

[Oliveira,2000] F. A. D. Oliveira, M. E. Barreto, R. B. Ávila, P. O. A. Navaux. *A Comparative Study on Low-level APIs for Myrinet and SCI-based Clusters*. In 4th World Multiconference on Systemics, Cybernetics and Informatics, vol. 4, p1-6, Orlando, Florida, USA, July 2000.

[Ong,2000] H. Ong, P. A. Farrell. *Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network*. In Proc. of Linux 2000, 4th Annual Linux Showcase and Conference, Extreme Linux Track, p.353-362, Atlanta, 2000.

[Pakin,1995] S. Pakin, M. Lauria, A. Chien. *High performance messaging on workstations : Illinois Fast Messages (FM) for Myrinet*. In Proceedings of ACM, 1995.

[Pakin,1997] S. Pakin, V. Karamcheti, and A.A. Chien. Fast Messages (FM) : *Efficient, Portable Communication for Workstation Clusters and MPPs*. IEEE Concurrency, 5(2):60-73, April-June 1997.

[Pant,2000] A. Pant, et al. *VMI: An Efficient Messaging Library for Heterogeneous Cluster Communication*. Extended Abstract submitted to HPDC-9, Pittsburgh, Pennsylvania, August 1-4, 2000. Available from : *http://archive.ncsa.uiuc.edu/General/CC/ntcluster/Articles.html*

[PARMA^2] Département Informatique de l'Université de Parme (Italie). *PARMA^2: porting VIA in LAM/MPI*. *http://www.ce.unipr.it/research/pardis/parma2*

[Petri,1999] S. Petri et al. *Performance Comparison of different High-Speed Networks with a Uniform Efficient Programming Interface*. In Proc. of SCI Europe'99, p83-90, Toulouse, France, September 1999.

[Pfister,1998] G. F. Pfister. *In Search of Clusters*. Prentice Hall PTR, Second Edition, ISBN 0-13-899709-8, 1998.

[Potter,1996] F. Potter. *Conception et réalisation d'un réseau d'interconnexion à faible latence et haut débit pour machine multiprocesseur*. Thèse de doctorat de l'Université de Paris VI, 1996.

[Prylli,1998] L. Prylli and B. Tourancheau. *BIP : a New Protocol Designed for High Performance Networking on Myrinet*. In Proc. Workshop PC-NOW, IPPS/SPDP'98, LNCS N°1388, Springer, p472-485, Orlando, FL, April 1998.

[Prylli,1999] L. Prylli, B. Tourancheau, R. Westrelin. *The design for a high performance MPI implementation on the Myrinet network.* In Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 6th European PVM/MPI Users' Group (EuroPVM/MPI '99), vol. 1697 of Lect. Notes in Comp. Science, p223-230, Barcelona, Spain, 1999.

[Prylli,2000] L. Prylli, R. Westrelin. *Current issues in available implementations of MPI on Myrinet*. In Proc. of First Myrinet User Group Conference, p149-155, Lyon, France, September 2000.

[PVM,1994] A. Geist et al. *PVM 3 User's Guide and Reference Manual*. ORNL/TM-12187, september 1994.

[PVM] Parallel Virtual Machine. *http://www.epm.ornl.gov/pvm/pvm_home.html*

[Reibaldi,1997] V. Reibaldi. *Conception et réalisation d'un routeur de paquets à hautes performances*. Thèse de doctorat de l'Université de Paris VI, 1997.

[Renault_1,2000] E. Renault. *Etude de l'impact de la sécurité sur les performances dans les grappes de PC*. Thèse de doctorat de l'Université de Versailles, Saint-Quentin-en-Yvelines, France, Décembre 2000.

[Renault_2,2000] E. Renault. *Sécurisation des blocs de mémoire contiguë*. In Ren-Par'12, p61-66, Juin 2000.

[Renault_3,2000] E. Renault. *PAPI User Manual.* Rapport de recherche, Laboratoire PRISM, Université de Versailles, Saint-Quentin-en-Yvelines, France, Mai 2000.

[Renault,2001] E. Renault, P. David. *Performance and Analysis of the PCI-DDC Remote-Write Implementation*. In IEEE International Conference on Cluster Computing, p197-203, Newport Beach, CA, October 2001.

[RWCP,1996] Myricom. *RWCP PC Cluster*. 1996. *http://www.myri.com/news/96a25.html*

[SCAMPI] SCALI, inc. (Scalable Linux Systems). *ScaMPI – high performance MPI. http://www.scali.com/products/scampi.html*

[SCI] SCI Association. *http://www.SCIzzL.com*

[Seifert_1,2001] F. Seifert, J. Worringen, W. Rehm. *Using Arbitrary Memory Regions for SCI Communication.* In Proc. SCI Europe 2001, p59-64, Dublin, Ireland, October 2001.

[Seifert_2,2001] F. Seifert, W. Rehm. *Reliably Locking System V Shared Memory for User Level Communication in Linux.* In Proc. of the IEEE International Conference on Cluster Computing CLUSTER'2001, Newport Beach, California, USA, Oct. 8-11, 2001.

[Seitz,1993] C. L. Seitz et al. *The design of the caltech mosaic c multicomputer*. In MIT Press, editor, Research on Integrated Systems Symposium, p1-22, 1993.

[Shoinas,1998] I. Schoinas, M. D. Hill. *Address Translation Mechanisms in Network Interfaces*. In Proc. of the 4th International Symposium on High-Performance Computer Architecture (HPCA), February 1998.

[Silva,1998] F. Silva and K. Mana. *Portage de PVM sur MPC*. Technical Report, Laboratoire d'Informatique de Paris VI, 1998. *http://mpc.lip6.fr/pvm.html*

[Spector,2000] D. Spector. *Building Linux Clusters*. O'Reilly, 2000.

[Sterling,1995] T. Sterling et al. *BEOWULF: A Parallel Workstation for Scientific Computation*. In Proc. 24th Int. Conf. on Parallel Processing, Oconomowoc, Wisconsin, August 1995.

[Sterling,1999] T. Sterling, J. Salmon, D. Becker, and D.Savarese. *How to Build a Beowulf*. MIT Press, 1999.

[Stunkel,1995] C. B. Stunkel, et col.. The SP2 high-performance Switch. IBM system journal, 34(2):185-204, 1995.

[Sun Microsystems] Sun Microsystems. *Sbus*. *http://www.sun.com/io_technologies/sbus*

[Tezuka,1997] H. Tezuka, A. Hori, Y. Ishikawa, M. Sato. *PM: An Operating System Coordinated High Performance Communication Library*. In Peter Sloot Bob Hertzberger, editor, High-Performance Computing and Networking, vol. 1225 of Lecture Notes in Computer Science, p708-717, Springer-Verlag, April 1997.

[Tezuka,1998] H. Tezuka, F. O'Carroll, A. Hori, Y. Ishikawa. *Pin-down Cache: A virtual Memory Management Technique for Zero-copy Communication*. In Proc. IEEE of Int. Conf. on IPPS/SPDP'98, p308-314, April 1998.

[theHIVE] TheHIVE: Highly-parallel Integrated Virtual Environment. *http://newton.gsfc.nasa.gov/thehive*

[TOP500] TOP500 Supercomputer Sites. *http://www.top500.org*

[UNET] M. Welsh. *U-Net: A User-level Network Interface Architecture*. 1998. *http://www.cs.berkeley.edu/~mdw/proj/unet*

[VIA] Compaq, Intel and Microsoft Corporations. *Virtual Interface Architecture Specification v1.0. http://www.viarch.org*

[VI-GM] Myricom. *VI-GM: Virtual Interface on Myrinet 1.0. http://www.myri.com/scs/VI-GM/doc/html*

[Vignes,1974] J. Vignes and M. La Porte. *Error analysis in computing*. In Information Processing 74, North-Holland, 1974.

[Vignes,1993] J. Vignes. *A stochastic arithmetic for reliable scientific computation*. In Mathematics and Computers in Simulation, Vol. 35, p233-261, July 1993.

[VMEbus,1996] VMEbus. *VMEbus: The Systems Architecture for the 21st century*. In Proc. of VITA, Europe Congress, 1996.

[VMMC,1999] Site internet de VMMC, Princeton University, 1999. *http://www.cs.princeton.edu/shrimp/html/vmmc.html*

[Wajsbürt,1997] F. Wajsbürt, J.L. Desbarbieux, A. Greiner, C. Spasevski, S. Penain. *An Integrated PCI component for IEEE 1355 Networks.* In Proc. of EMMSEC'97, Florence, Italy, 1997.

[Welsh,1996] M. Welsh, A. Besu, T. v. Eicken. *Low-Latency Communication over Fast Ethernet*. In Proc. of Euro-Par'96, Lyon, France, 1996.

[Welsh,1997] M. Welsh, A. Besu, T. v. Eicken. *Incorporating memory management into user-level network interfaces*. Technical Report TR97-1620, Department of Computer Science, Cornell University, Ithaca, 1997.

[Westrelin,2001] R. Westrelin. *Propositions autour de l'architecture logicielle des interfaces réseau programmables*. Thèse de doctorat de l'Université Claude BERNARD, Lyon, 2001.

[Wilkinson,1999] B. Wilkinson and M. Allen. *Parallel Programming*. Prentice Hall, 1999.

[Wong,1999] F. C. Wong, A. C. Arpaci-Dusseau, D. E. Culler. *Building MPI for Multi-Programming Systems using Implicit Information*. In the 6th European PVM/MPI User's Group Meeting, Aug. 1999.

[Worringen,1999] J. Worringen, T. Bemmerl. *MPICH for SCI-Connected Clusters*. In Proc. SCI-Europe '99 (Conference Stream of Euro-Par '99), p3-11, Toulouse, France, September 1999.

[Worringen,2000] J. Worringen. *SCI-MPICH - The Second Generation*. In Proc. SCI-Europe 2000 (Conference Stream of Euro-Par 2000), p10-20, Munich, Germany, August 2000.

[Worringen,2001] J. Worringen, F. Seifert, T. Bemmerl. *Efficient Asynchronous Message Passing via SCI with Zero-Copying*. In Proc. SCI Europe 2001, p65-74, Dublin, Ireland, October 2001.

[Worringen,2002] J. Worringen, A. Gäer, F. Reker. *Exploiting Transparent Remote Memory Access for Non-Contiguous and One-Sided-Communication*. In Proc. of ACM/IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002), Workshop for Communication Architecture in Clusters (CAC 02), Fort Lauderdale, USA, April 2002.

[Zerrouki,2000] A. Zerrouki, O. Glück, J.L. Desbarbieux, A. Fenyö, A. Greiner, C. Spasevski, F. Wajsbürt, F. Silva and E. Dreyfus. *The MPC Parallel Computer: Hardware, Low-level Protocols and Performances*. In Proceedings of Parallel and Distributed Computing and Systems (PDCS'2000), Vol. 1, p87-92, Las Vegas, USA, November 6-9, 2000.

Glossaire

ADI (*Abstract Device Interface*) [Gropp,1994] : couche de communication de MPICH définissant un ensemble restreint de primitives de communication point à point permettant de réaliser le portage de MPI sur différentes architectures

AM (*Active Messages*) [Eicken,1992] [Mainwaring,1995] : les Messages Actifs issus de l'Université de Berkeley ; bibliothèque de communication bas niveau

API (*Applications Programming Interface*) : ensemble de fonctions décrivant une interface de programmation

ATM (*Asynchronous Transfer Mode*) [ATM,1995] : réseau d'interconnexion

BEOWULF [Sterling,1995] [Sterling,1999] [Beowulf] : le projet Beowulf

BIP (*Basic Interface for Parallelism*) [Prylli,1998] [BIP] : bibliothèque de communication bas niveau pour le réseau Myrinet

CADNA (*Control of Accuracy and Debugging for Numerical Applications)* [CADNA] [Chesneaux,1995] [Vignes,1993] : bibliothèque logicielle pour le calcul numérique permettant de prendre en compte les erreurs d'arrondi dans les calculs flottants

CESTAC [Vignes,1974] : méthode de contrôle et d'estimation des erreurs de propagation d'arrondi

CH_RDMA : couche de communication présentée dans ce manuscrit réalisant l'interface entre MPICH et l'API RDMA

CMEM : module noyau permettant l'allocation de blocs contigus en mémoire physique

CREDIT : message de contrôle dans notre implémentation de MPI permettant l'échange de crédits pour le contrôle de flux entre deux processus MPI

CTRL : message de contrôle dans notre implémentation de MPI

DATA : message de données dans notre implémentation de MPI

DMA (*Direct Memory Access*) : accès direct à la mémoire centrale de la part d'un composant matériel d'un nœud de calcul (contrôleur réseau, processeur, etc.)

FIFO (*First In First Out*) : un réseau est dit FIFO si l'ordre d'arrivée des messages sur le nœud récepteur correspond à l'ordre dans lequel les messages ont été émis sur le nœud émetteur

FM (*Fast Messages*) [Pakin,1997] [Lauria,1998] : les Fast Messages issus de l'Université de l'Illinois ; bibliothèque de communication bas niveau de type Messages Actifs pour le réseau Myrinet

GDT (*Global Descriptor Table*) : la table globale de stockage des descripteurs de segments sur les architectures 80x86

GM [GM,2000] : bibliothèque de communication bas niveau pour le réseau Myrinet, fournie par la société Myricom

HPCC (*High Performance Cluster Communication*) [Petri,1999] : API qui fournit au niveau applicatif un accès direct et efficace à différents réseaux haut débit

HPF (*High Performance Fortran*) [Koelbel,1994] : langage de programmation parallèle

HSL (*High Speed Link*) [Reibaldi,1997] [HSL,1994] : réseau haut débit développé au LIP6 et défini par la norme IEEE-1355

LAM [Burns,1994] [LAM] : implémentation du standard MPI issu de l'Ohio Supercomputer Center

LAN (*Local Area Network*) : réseau de plusieurs stations interconnectées entre elles souvent par un réseau de type Ethernet, token ring, FDDI, etc.

LANai : processeur RISC programmable de l'interface réseau Myrinet

LANDA (*Local Area Network for Distributed Applications*) [Monteil,1995] : projet de recherche dont l'un des objectifs est de fournir un environnement de programmation parallèle pour les clusters permettent de communiquer avec différents réseaux d'interconnexion simultanément

LINPACK [LINPACK] [Dongarra,1994] : ensemble de « benchmarks » d'algèbre linéaire utilisé pour comparer les performances des machines parallèles

LIP6 (*Laboratoire d'Informatique de Paris 6*) : laboratoire de recherche dans lequel cette thèse a été réalisée

LME (*Liste des Messages à Emettre*) : liste contenant les descripteurs DMA correspondant aux messages à transférer sur un réseau disposant d'une primitive d'écriture distante

LMM (*Locked Memory Manager*) [Seifert_2,2001] : gestionnaire de mémoire verrouillée développé à l'Université Technologique de Chemnitz dans le cadre d'une implémentation de VIA ; module noyau LINUX qui permet de verrouiller/déverrouiller les tampons de l'application et de retrouver efficacement leurs adresses physiques à l'aide d'un mécanisme appelé « kiobufs »

LMR (*Liste des Messages Reçus*) : liste contenant la liste des messages reçus sur un réseau disposant d'une primitive d'écriture distante

LRU (*Least Recently Used*) : méthode utilisée dans la gestion des caches pour leur mise à jour

MCP (*Myrinet Control Program*) : code exécuté par le LANai sur l'interface réseau Myrinet

MPC (*Multi-PC*) [MPC] : projet de recherche démarré en 1995 au laboratoire LIP6 dont le but est la réalisation de machines parallèles performantes à faible coût ; machine parallèle de type grappe de PCs utilisant le réseau HSL

MPI (*Message Passing Interface*) [MPI] [MPI,1994] [MPI,1997] : standard définissant un environnement de programmation parallèle à passage de messages

MPICH [Gropp,1996] [Gropp,2002] : implémentation du standard MPI permettant un portage efficace sur différentes architectures grâce à son organisation en couches ; probablement l'implémentation la plus utilisée actuellement

MPI-MPC : implémentation optimisée de MPI sur des architectures matérielles disposant d'une primitive d'écriture en mémoire distante

M-VIA [M-VIA] : implémentation du standard VIA sur des cartes Fast-Ethernet ou Giga-Ethernet sous Linux

MVICH [MVICH] : implémentation spécifique de MPI sur VIA qui utilise M-VIA pour accéder au réseau Gigabit Ethernet

NIC (*Network Interface Controller*) : contrôleur réseau se trouvant sur les cartes d'interface

NOW (*Networks Of Workstations*) [Anderson,1995] [NOW] : projet de recherche de l'Université de Berkeley dont le but est la réalisation de machines parallèles à faible coût en interconnectant des stations de travail individuelles entre elles

OpenMP [Chandra,2000] : standard de programmation parallèle définissant une API pour le modèle de programmation à mémoire partagée

PAPI (*Pci-ddc Application Programming Interface*) [Renault_3,2000] : interface en mode utilisateur de la primitive d'écriture distante de la machine MPC, réalisée au laboratoire PRiSM de l'Université de Versailles

PARMA^2 [PARMA^2] : projet de recherche de l'Université de Parme proposant une implémentation de LAM/MPI sur M-VIA

PCI (*Peripheral Components Interface*) [Cyliax,2000] : standard définissant le bus PCI

PIO (*Programmed I/O*) : méthode pour l'échange de données entre l'interface réseau et le nœud hôte

PM [Tezuka,1997] : bibliothèque de communication pour le réseau Myrinet faisant partie d'un environnement de programmation parallèle développé par le Real World Computing Partnership (RWCP)

PUT [Fenÿo,2001] : bibliothèque de communication bas niveau de la machine MPC fournissant une primitive d'écriture distante en mode noyau

PVM (*Parallel Virtual Machine*) [PVM,1994] [PVM] : environnement de programmation parallèle à passage de messages

RDMA (*Remote Direct Memory Access*) : API générique d'écriture en mémoire distante définie dans ce manuscrit

REQ : message de contrôle de type « requête » dans notre implémentation de MPI

RHANDLE : structure de données de MPICH associée à une requête de réception

RSP : message de contrôle de type « réponse » dans notre implémentation de MPI

Sbus [Sun Microsystems] : bus des SPARC de Sun Microsystems

SCAMPI [SCAMPI] : implémentation commerciale de MPI pour le réseau SCI, réalisée par la société Scali

SCI (*Scalable Coherent Interface*) [Gustavson,1992] [SCI] : standard IEEE 1596-1992 dont le but est de fournir une mémoire partagée globale à faible latence aux processeurs constituant un cluster ; le principal fournisseur de matériel SCI est la société Dolphin

SISCI [Giacomini,1999] : bibliothèque de communication bas niveau pour le réseau SCI qui permet de communiquer avec le pilote IRM de l'adaptateur PCI-SCI de la société Dolphin

SHANDLE : structure de données de MPICH associée à une requête d'émission

SHORT : message de contrôle dans notre implémentation de MPI dans lequel les données de l'application sont encapsulées pour être transférées

SHRIMP (*Scalable High-performance Really Inexpensive Multi-Processor*) [Blumrich,1994] : projet de recherche de l'Université de Princeton dont le but est la réalisation d'une machine parallèle à faible coût, l'idée générale étant de faire réaliser par le matériel réseau des fonctionnalités traditionnellement fournies par le système d'exploitation ; nom d'une machine parallèle

SMI (*Shared Memory Interface*) [Dormanns,1997] : bibliothèque de communication bas niveau réalisant l'interface entre l'ADI de MPICH et l'API SISCI

SMP (*Symmetric MultiProcessor*) : modèle de programmation à mémoire partagée pour des processeurs identiques interconnectés par un bus mémoire

TLB (*Translation Lookaside Buffer*) : table contenant la correspondance entre les adresses virtuelles et les adresses physiques

U-Net [UNET] [Eicken,1995] [Welsh,1997] : bibliothèque de communication bas niveau pour les réseaux Ethernet, ATM et Myrinet

UTLB (*User-managed TLB*) [Dubnicki,1997] : table logicielle contenant la correspondance entre les adresses virtuelles/physiques des zones de communication de la bibliothèque VMMC

VI (*Virtual Interface*) : entité permettant à deux processus de communiquer via le standard VIA

VIA (*Virtual Interface Architecture*) [Dunning,1998] [VIA] : standard pour les systèmes de communication de niveau utilisateur, proposé par Compaq, Intel et Microsoft

VME [VMEbus,1996] : bus reliant des composants périphériques à l'hôte

VMI (*Virtual Machine Interface*) [Pant,2000] : bibliothèque de communication haut niveau, réalisée par le National Center for Supercomputing Applications de l'Université de l'Illinois, permettant de choisir dynamiquement le meilleur réseau pour communiquer

VMMC (*Virtual Memory-Mapped Communication*) [Dubnicki,1997] : bibliothèque de communication en mode utilisateur, issue du projet SHRIMP, pour la machine parallèle SHRIMP et le réseau Myrinet

ZCSP (*Zero Copy Secured Protocol*) : protocole de communication pour le réseau HSL, réalisant une primitive d'écriture en mémoire distante sécurisé avec correction des erreurs, implanté au niveau du matériel réseau