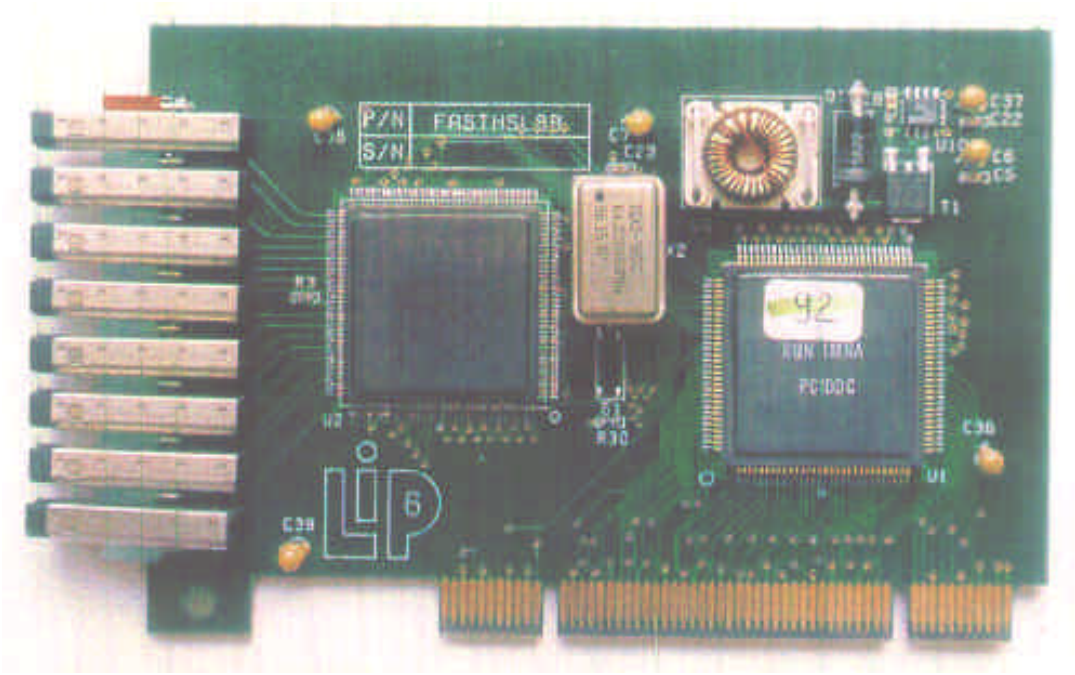
Le Laboratoire d'Informatique de Paris 6

L'université d'EVRY

RAPPORT DE STAGE DEA

NON CONFIDENTIEL



Etudes de performance sur une machine parallèle de type « Grappe de PCs » : la machine MPC

Olivier GLÜCK

Mars-Août 1999

Sous l'encadrement de Pr. Alain Greiner (Alain.Greiner@lip6.fr)

STAGE DEA 1998 / 1999

Laboratoire Informatique de Paris 6

Nom du stagiaire : GLÜCK Olivier

Option **:** DEA Informatique

Laboratoire **:** Laboratoire Informatique de Paris 6

Dates **:** mars – août 1999

Sujet **:** Etudes de performance sur la machine parallèle MPC

RESUME :

Ce stage s'inscrit dans le cadre du projet de recherche Multi-PC, qui a démarré en janvier 1995, sous la responsabilité d'Alain GREINER. Il vise la conception et la réalisation d'une machine parallèle à faible coût dont les nœuds de calcul sont des PCs interconnectés par un réseau haut débit.

Le stage a consisté en la comparaison des performances du réseau MPC et d'un réseau ETHERNET 100 Mbits/s. Cette étude s'est faite à différents niveaux des couches logicielles et, en particulier, par la parallélisation de la résolution de l'équation de Laplace sur l'environnement de programmation parallèle PVM.

Enfin, des propositions ont été faites afin d'espérer améliorer les performances au niveau des couches logicielles de la machine MPC.

ABSTRACT :

This training period is a part of the global research project called Multi-PC which started in 1995, under the responsibility of Pr. Alain GREINER. This project deals with the design and the realisation of a low-cost parallel computer composed of calcul-nodes, using standard pentium based PC-motherboards as processing nodes, Unix FreeBSD operating system, and a high speed communication network based on the IEEE 1355/HSL technology.

This training period deals with the comparison of an 100 Mbits/s ETHERNET network and the MPC network. This study was done at different levels of software layers. In particular, the parallelisation of Laplace equation was done under the Parallel Virtual Machine environment.

Finally, propositions have been done in order to improve performances of MPC communication layers.

Remerciements

Je tiens à remercier les enseignants - chercheurs, les thésards, et plus généralement toute l'équipe du département ASIM du LIP6 pour le soutien qu'ils ont pu m'apporter, leur ouverture d'esprit, et leur bonne humeur.

Je remercie, en particulier, Alain GREINER, non seulement pour m'avoir permis d'effectuer ce stage, mais aussi pour m'avoir aidé lors des différentes réflexions concernant Fast-PVM.

Je remercie également, de tout cœur, Daniel MILLOT et Philippe LALEVEE (INT) sans qui ce stage n'aurait pas pu se faire. Je les remercie en particulier pour la relecture de mes documents, leurs nombreux conseils et une aide permanente.

Table des matières

Table des figures

Chapitre I :

Introduction et contexte du stage

Chapitre I : Introduction et contexte du stage

Ce chapitre a pour but de présenter le contexte du stage. Il s'agit d'une part de replacer mon stage et mon travail dans le cadre du projet MPC de l'équipe de recherche qui m'a accueillie. D'autre part, il s'agit d'expliciter le sujet de mon stage. Enfin, il s'agit de présenter quelques généralités sur les architectures parallèles construites à partir d'une grappe de PCs afin de dégager l'intérêt du projet MPC et de mon stage.

I. Le projet MPC

I.1 Le projet MPC au sein du LIP6

Mon stage s'est effectué dans le laboratoire de recherche du département *ASIM* (Architecture des systèmes intégrés et micro-électronique / http://www-asim.lip6.fr/) du *LIP6* (Laboratoire d'Informatique de Paris 6) au sein de l'équipe de recherche dont le travail repose sur trois projets (*Figure 1*):

- La machine MPC
- Projet d'indexation multimédia
- Projet CAO de circuits et systèmes (Alliance)

Le LIP6 est l'instrument de la politique scientifique de l'université Pierre et Marie Curie en matière d'informatique, exprimée dans le cadre du contrat quadriennal 1997-2000 qui lie l'université, le CNRS et le ministère. Il est composé de 9 thèmes de recherche dont les activités recouvrent une large part de l'informatique (http://www.lip6.fr/).

Outre leurs activités propres, ces thèmes collaborent dans diverses actions transversales et sont engagés dans de nombreux projets en partenariat avec des entreprises.

Le projet MPC est l'un de huit projets transversaux qui déterminent la politique scientifique du LIP6.

Il est un projet de longue haleine. L'idée générale est d'aider certains demandeurs de puissance de calcul du LIP6 à mener des expériences sur la machine MPC développée au laboratoire dans le thème ASIM. Mais surtout, le but du projet MPC est la conception d'une machine parallèle performante et à faible coût. La contribution du laboratoire s'est traduite par l'acquisition d'un réseau à 4 nœuds. Ceci est suffisant pour mettre au point des expériences mais il serait bon d'avoir au moins huit nœuds pour obtenir des résultats significatifs.

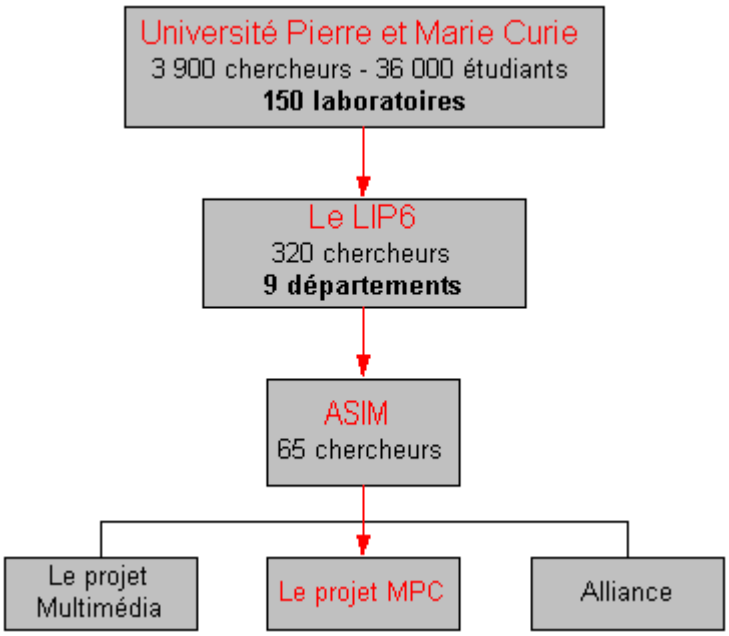


Figure 1 : De l'UPMCP6 au projet MPC

I.2 Pourquoi le projet MPC ?

Les années 1990 ont vu l'émergence des réseaux locaux haut-débit. Citons par exemple, ATM, Ethernet 100 Mb/s, Ethernet Giga-bit, Myrinet, SCI, etc. Ces progrès sont la conséquence logique de l'avancement de la technologie et en particulier d'une augmentation de la densité d'intégration des composants électroniques. La latence de ces réseaux est de l'ordre de la dizaine de micro-secondes et le débit dépasse souvent le Giga-bit.

Une conséquence directe de cette évolution est l'apparition d'un nouveau type d'architectures parallèles à base d'un réseau de stations. En effet, les réseaux d'interconnexion à haute performance permettent à différents processeurs de collaborer grâce à l'échange de données entre les différents nœuds de calcul. Le rapport coût/performance de ces nouvelles machines parallèles s'approche de celui des calculateurs parallèles classiques (cf. *III. Les architectures parallèles et les réseaux de stations*).

Le projet MPC (*Multi-PC*), démarré en janvier 1995 sous la responsabilité d'Alain GREINER, s'intègre complètement dans cette évolution. Il a d'abord consisté, à partir de 1993, à développer une technologie d'interconnexion haute performance. Il s'agit de la technologie HSL qui est devenue le standard IEEE 1355. Cette technologie s'appuie sur deux composants VLSI développés au département ASIM du LIP6 : **PCI-DDC** et **RCUBE** (cf. **[1][2][5]**).

L'objectif du projet MPC vise à la réalisation d'une machine parallèle performante à faible coût, de type *grappe de PCs*, basée sur un réseau d'interconnexion utilisant la technologie HSL. Ce réseau utilise des liaisons séries, point à point, bidirectionnelles et full-duplex, à 1 Gbit/s. La latence matérielle est inférieure à 10 micro-secondes. Le matériel spécifique se

limite à une carte d'extension réseau HSL. La cellule processeur est une carte mère standard pour PC possédant un bus PCI.

Différents industriels (Bull, SGS-Thomson, Parsytec, Thomson) participent au projet MPC dans le cadre de quatre projets européens visant le développement ou l'exploitation de la technologie HSL. Par ailleurs, le projet MPC fait l'objet d'une collaboration avec d'autres laboratoires de recherche : PRISM (Versailles), INT (Evry), ENST (Paris), Amiens, Toulouse, etc. Les cartes réseaux FastHSL et les composants VLSI sont commercialisés par la société Tachys Technologies, qui est une « start-up » du département ASIM du LIP6.

Les projets comparables au projet MPC sont des projets industriels : SCI, Myrinet et Memory Channel. SCI est une norme (standard IEEE 1596) qui s'appuie sur la connexion de plusieurs anneaux par des passerelles. Myrinet est un réseau qui utilise des liaisons point à point à 1 Gbit/s. A la différence du réseau HSL, Myrinet utilise des liaisons parallèles. De plus, la fonction de routage est centralisée et nécessite un boîtier séparé alors que chaque nœud du réseau HSL possède un ou plusieurs routeurs RCUBE. Myrinet utilise également des cartes d'interface PCI. La densité d'intégration des composants électroniques utilisés par la technologie HSL est plus importante que celle de la technologie Myrinet. Enfin, Memory Channel est un réseau basé sur une carte PCI, des liaisons point à point et une fonction de routage centralisée.

II. Le sujet du stage

II.1 Objectif

L'objectif du stage est à travers l'étude de l'implémentation d'une application classique notamment, d'évaluer les performances de la machine parallèle MPC du laboratoire LIP6, non seulement au niveau de l'environnement PVM de développement fourni avec celle-ci, mais aussi au niveau des couches logicielles basses. L'application choisie est souvent utilisée comme benchmark pour comparer différentes parallélisations, différentes machines, ou différents environnements de programmation parallèle : il s'agit de la résolution de l'équation de Laplace sur une grille bidimensionnelle.

II.2 Description

Un outil d'administration de la machine MPC a été réalisé durant le début du stage qui constituait le stage de fin d'études « ingénieur » de ma formation à l'INT afin, d'une part, de pouvoir gérer le chargement des couches logicielles et, d'autre part, de permettre aux différents utilisateurs de tester leurs applications.

Les nœuds de calcul de la machine MPC sont interconnectés non seulement par les couches de communications rapides MPC, mais aussi par un réseau de contrôle ETHERNET 100 Mbit/s.

L'environnement standard de programmation distribuée PVM a été porté sur l'interface de programmation proposée par les couches de communication MPC. Cependant, aucune véritable application n'a encore été testée sur cette plate-forme. Il s'agit donc de faire les premiers tests en parallèlisant la résolution de l'équation de Laplace sur cette architecture.

Par ailleurs, d'autres procédures de tests pourront être mises en place afin de comparer les performances des couches de communications MPC et de PVM-MPC avec Ethernet et un PVM standard. On s'attachera à identifier les forces et les faiblesses de l'implémentation existante de PVM.

Enfin, des propositions pourront être faites afin de gagner encore en performances, par exemple en adaptant au mieux les couches logicielles de communications à l'architecture matérielle de la machine. L'enjeu est de faire bénéficier les applications de la très faible latence matérielle du réseau en minimisant le coût des couches logicielles.

Ce sujet s'insère dans le cadre d'une coopération entre le LIP6 et l'Université de Queens (Prof. C. Hamacher).

II.3 Responsables

Directeur de stage : **Professeur Alain Greiner**, Alain.greiner@lip6.fr (LIP6)
Conseillers d'études : **Daniel Millot** et **Philippe Lalevée**, (Université d'Evry, INT)

III. Les architectures parallèles et les réseaux de stations

Cette section a pour but le rappel de quelques éléments importants concernant les réseaux de stations et les « *clusters* ».

III.1 Des architectures parallèles type « grappe de PCs »

Un nouveau type d'architecture parallèle s'est fortement développé grâce aux progrès réalisés dans le domaine des réseaux de stations. Pour réaliser une machine parallèle, il suffit de connecter des PC par un réseau d'interconnexion. L'utilisation d'une grappe de PCs en tant qu'une architecture parallèle est de plus en plus fréquente.

L'émergence des réseaux locaux haut-débit comme Myrinet, ATM, SCI ou MPC permet d'avoir des mécanismes de communication tout à fait compétitifs face à ceux des architectures parallèles classiques comme les réseaux de calculateurs parallèles (l'IBM SP par exemple). Cette nouvelle génération de réseaux d'interconnexion permet généralement des communications point à point de plus d'1 Gbit/s avec une latence de l'ordre de 10 micro-secondes.

Enfin, le rapport coût/performance des clusters est nettement favorable quand on le compare à celui des architectures classiques. Du fait de l'utilisation de composants standards produits en très grands volumes, le coût d'une grappe de PCs est très faible comparativement au coût des « supercomputers » comme une T3E (Cray) ou un SP2 (IBM). Or, les performances sont comparables.

III.2 Le modèle de programmation

Une machine parallèle sert à programmer des applications parallèles. Un modèle de programmation est donc nécessaire.

Les réseaux de stations ne disposent pas d'une mémoire partagée. Le modèle de programmation le plus utilisé est donc basé sur l'échange de messages entre les différentes stations. C'est au programmeur de gérer l'ensemble de ses données et de savoir quand communiquer quelles données et à qui. Il existent d'autres modèles de programmation sur les réseaux de stations comme des modèles à mémoire partagée mais ils sont encore souvent à l'état de prototype.

Le modèle de programmation à passage de messages est très répandu. Il est basé sur deux primitives (***send*** et ***receive***) et sur des mécanismes de synchronisation entre les différents processeurs. Le *send* permet d'envoyer des messages à un processus distant et le *receive* d'en recevoir. Ainsi, les différents processus d'une application parallèle peuvent se communiquer les paramètres et les résultats qui concernent leur domaine de calcul. Il s'agit de communications point à point.

La primitive *send* peut être ***synchrone*** ou ***asynchrone*** : dans le premier cas, le *send* n'est terminé que lorsque le récepteur a bien reçu le message (l'émetteur et le récepteur sont synchronisé) alors qu'un *send asynchrone* n'attend pas de savoir si le récepteur a reçu le message.

Enfin, les primitives *send* et *receive* peuvent être bloquantes ou non bloquantes :

- ***send bloquant*** : ne retourne que lorsque les données à émettre ont été envoyées ou recopiées dans un buffer d'émission ; le contenu du message ne peut plus être modifié après le *send*.

- ***send non bloquant*** : retourne aussitôt (dès que l'ordre d'envoi a été émis) ; si les données sont modifiées après le *send*, le contenu du message est changé.

- ***receive bloquant*** : ne retourne que lorsque les données sont disponibles.

- ***receive non bloquant*** : retourne aussitôt ; indique si les données sont disponibles ou non par un drapeau.

Il existe principalement deux bibliothèques de passage de messages utilisées dans les réseaux de stations : **PVM** (Parallel Virtual Machine) qui est devenu un standard et **MPI** (Message Passing Interface) qui définit une norme. Elles sont construites par dessus des langages de programmation séquentielle standard comme le C et le Fortran. Nous nous intéresserons désormais plus particulièrement à PVM qui est la couche portée sur la machine MPC.

III.3 Les performances dans les clusters

L'objectif de cette section est d'évaluer à priori, dans quelles mesures les performances dans une architecture parallèle à base d'un cluster peuvent être améliorées. Il s'agit également de mettre en valeur les contraintes que ces améliorations peuvent engendrer.

Nous considérons dans cette section les performances lors de communications point à point entre processus communicants grâce à une bibliothèque de passage de messages. Ces performances de communication sont très importantes pour la performance globale d'une application parallèle qui utilise le modèle de programmation à passage de messages.

Les architectures parallèles comportent 4 ressources principales : les processeurs, la mémoire, les entrées/sorties et le réseau de communication interne. L'évolution des performances des trois premières ressources est guidée par la technologie grâce à l'augmentation de la densité d'intégration des composants électroniques. En revanche, les performances des réseaux de communications dépendent en grande partie du temps passé en dehors des couches matérielles, c'est à dire dans les couches logicielles.

III.3.1 Les facteurs qui influent sur les performances

A la fin des années 1980, les temps de traversée des couches logicielles et matérielles d'un réseau de communication étaient semblables. Aujourd'hui, avec les progrès considérables au niveau matériel, il y a une émergence de nombreux réseaux locaux haut-débit (ATM, Myrinet, SCI, MPC, etc.). La conséquence de ces progrès est le fait que le temps de traversée des couches matérielles est devenu très faible comparativement au temps de traversée des couches logicielles. C'est donc au niveau de l'interface logicielle entre le processeur et le réseau que les performances des réseaux haut-débit doivent être améliorées.

Le point clé est l'utilisation ou non du système d'exploitation comme mécanisme intermédiaire de gestion de l'interface matérielle **[7][8]**. Il est nécessaire de passer par le système d'exploitation lorsque les mécanismes de communications sont accessibles par différents utilisateurs simultanément, et ce pour des raisons de sécurité et fiabilité. Dans le cas de plusieurs utilisateurs, le système d'exploitation sert d'intermédiaire pour communiquer avec les éléments matériels du réseau. La Figure 2 présente les différentes relations entre un processus utilisateur, le système d'exploitation et l'interface réseau.

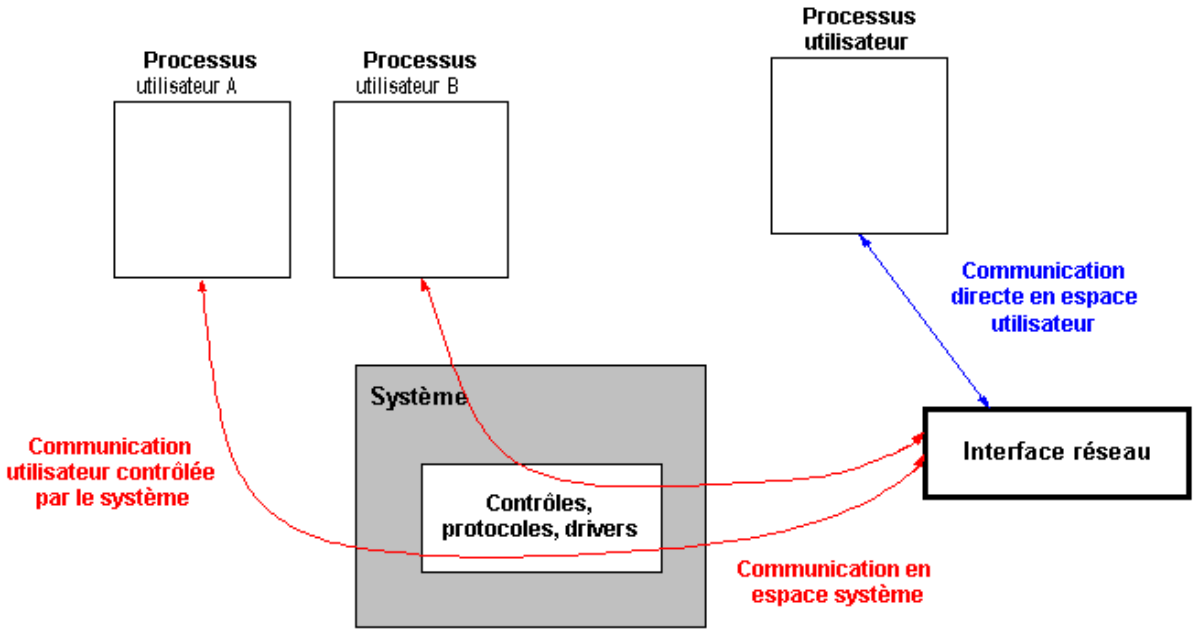


Figure 2 : Accès à l'interface réseau depuis un processus utilisateur

Si l'on se place dans le cas d'un seul utilisateur, cela signifie qu'une seule application parallèle peut s'exécuter à la fois. Par exemple, Myrinet se trouve dans ce cas de figure. S'il n'y a qu'un seul accès à l'interface réseau, il n'y a alors plus besoin d'utiliser le système d'exploitation pour introduire des mécanismes de contrôles d'accès et de protection du système. L'interface réseau peut alors être accédée directement depuis et vers l'espace utilisateur.

D'une façon générale, c'est plutôt la configuration multi-utilisateurs qui a été choisie. Pourtant, la traversée du système d'exploitation est très coûteuse : elle engendre **à chaque communication**, d'une part, des appels systèmes et d'autre part, des recopies de données utilisateurs de l'espace mémoire du processus vers l'espace mémoire du système.

Ainsi, le nombre de recopies et d'appels systèmes effectués à chaque communication est déterminant pour les performances des communications.

III.3.2 L'amélioration des performances

Le plus important pour améliorer les performances est de réduire le chemin critique logiciel que constitue la traversée de la pile de protocoles (ou de couches logicielles) qui sépare l'appel à une fonction de communication à partir du processus utilisateur de la prise en compte effective du message par le matériel réseau. Les rôles de ces différentes couches logicielles sont multiples : la fiabilité de la transmission, la protection des différents utilisateurs (au niveau de leur espace d'adressage), le partage virtuel des ressources matérielles, la gestion de l'hétérogénéité, etc. Cela engendre des appels systèmes, des recopies de messages, des formatages de messages, du calcul de codes de détection d'erreurs, des mécanismes de contrôle de flux, etc.

Pour améliorer les performances de communications dans les clusters, il faut donc essayer de réduire et de simplifier les mécanismes énoncés précédemment et ce dans la mesure du possible.

Par exemple, si l'ensemble des nœuds de calcul du réseau de stations est homogène, il est inutile avant chaque transmission de traverser une couche logicielle faisant de l'encodage XDR.

Si le réseau physique est considéré comme très fiable (il ne génère quasiment aucune erreur), il est peut être inutile de faire de la détection d'erreur. C'est par exemple le cas de la machine MPC.

Les recopies sont d'autant plus coûteuses que la taille du message est grande. S'il n'y a pas lieu de formater les données avant de les transmettre, il est sans doute possible d'éviter une recopie du message. De même, s'il n'y a pas de mécanismes de retransmission des données en cas de mauvaise transmission, une recopie peut être évitée.

Un appel système coûte quelques micro-secondes. Si les ressources matérielles du réseau ne sont pas partagées, et sont dédiées à une seule application, des appels systèmes peuvent être évités.

Quelques pistes ont été données pour tenter d'améliorer le temps de traversée des couches logicielles. Cependant, il est clair qu'une étude au cas par cas est nécessaire en fonction de l'architecture dont on dispose. Par ailleurs, un compromis entre sécurité, fiabilité et performances doit être trouvé.

III.3.3 Les mesures de performances

Les performances des réseaux de communications sont évaluées grâce aux deux paramètres que sont la ***latence*** et le ***débit***. La latence se mesure en unité de temps (généralement la micro-seconde) et le débit en octets par seconde ou bits par seconde.

La définition théorique de la latence dit qu'elle est le temps d'acheminement d'un message de zéro octet entre un processus émetteur et un processus récepteur. Un message de zéro octet n'existe pas. La latence n'est donc qu'une extrapolation d'autres mesures.

Pour mesurer le débit, il faut qu'un processus émetteur envoie à un processus récepteur des messages de taille fixe et si possible, suffisamment grande pour diminuer l'influence de la latence.

Pratiquement, on procède par exemple comme suit. Il y a un processus émetteur et un processus récepteur sur deux nœuds distincts. L'émetteur envoie successivement 100 messages de 1 octet, 100 messages de 2 octets, puis 100 messages de 4 octets, et ainsi de suite pour toutes les puissances de 2 jusqu'à une taille de 64Ko par exemple. De son coté, le récepteur renvoie tous les messages qu'il reçoit à l'émetteur. L'émetteur mesure le temps que chaque message met pour faire un aller-retour à l'aide d'une fonction comme *gettimeofday()*, par exemple. Ensuite, il fait une moyenne du temps de transfert (aller-retour/2) sur les 100 messages de chaque taille. Il est alors possible de tracer une courbe donnant le temps de transfert du message en fonction de sa taille (cf. Chapitre 5). Cette courbe est le plus souvent linéaire car les temps de transfert sont généralement proportionnels à la taille des messages (surtout pour des messages de tailles importantes). Par une régression linéaire, on obtient l'équation de droite qui approche au mieux l'ensemble des points (taille, temps de transfert) :

$$\boxed{\textbf{\textit{Temps transfert}} = \textit{PENTE} * \textbf{\textit{Taille}} + \textit{LATENCE}}$$

PENTE et LATENCE sont les deux constantes qui sont déterminées par la régression linéaire. LATENCE représente effectivement la latence puisque cette constante correspond au temps de transfert pour une taille de message nulle.

Pour obtenir le débit, il suffit de considérer le temps de transfert du plus gros message (64Ko par exemple), la latence étant alors négligeable :

$$\boxed{\textit{Débit} = \frac{64 * 1024 * 8}{\textit{Temps}(\text{sec})}}$$

Les mesures de latence et de débit peuvent se faire à différents niveaux des couches logicielles ; tout dépend des primitives utilisées pour envoyer et recevoir les messages.

D'autre part, la latence peut se décomposer en deux composantes : la latence logicielle (latence due à la traversée des couches logicielles) et la latence matérielle. Pour obtenir la latence logicielle, il suffit de faire les mesures décrites ci-dessus en « loop back » ce qui signifie que l'émetteur et le récepteur sont sur le même nœud ; il n'y a donc plus le coût de la traversée du réseau physique.

Chapitre II: La machine MPC

Chapitre 2 : La machine MPC

Ce chapitre présente la machine MPC. Il se décompose en deux parties : la présentation des couches matérielles puis la présentation des couches basses de communication MPC.

I. Architecture matérielle

La machine MPC du LIP6 est une machine parallèle de type « grappe de PCs ». Elle est constituée d'un ensemble de PCs standards interconnectés par un réseau à très haut débit dont la technologie a été développée par le laboratoire LIP6. Elle est composée de 4 nœuds de calcul bi-Pentium haut de gamme. La machine MPC a été imaginée dans l'idée de pouvoir constituer des machines pouvant avoir un très grand nombre de nœuds de calcul mais les machines actuelles sont expérimentales et ne possèdent que 4 ou 8 nœuds.

Tous les nœuds sont équipés d'une carte Ethernet afin de constituer un réseau de contrôle nécessaire à l'établissement du réseau HSL.

Le réseau d'interconnexions rapides est un réseau à 1 Gigabit/s en full duplex à la norme IEEE 1335 HSL. Il est composé de **liens HSL** et d'une **carte FastHSL** sur chaque nœud.

Le lien HSL est un lien série, haut débit, point à point et bidirectionnel, qui permet d'avoir un débit matériel (maximum) de 1 Gbit/s dans les deux sens de communication simultanément. Il est constitué de deux gaines coaxiales. La communication est asynchrone ; elle utilise un bit de Start et un bit de Stop.

La carte FastHSL a été conçue au laboratoire LIP6. Elle contient, en particulier, 2 circuits VLSI (cf. *Figure 3*):

- Un contrôleur de bus PCI intelligent appelé ***PCI-DDC*** qui réalise le protocole de communication
- Un routeur rapide possédant 8 liens HSL à 1 Gbit/s appelé ***RCUBE*** qui assure le routage des paquets

Le routeur est un cross-bar 8*8 entre 8 liens HSL. Les paquets sont constitués d'une entête de deux octets qui spécifie la destination des paquets. Un paquet peut avoir une taille quelconque ; il se termine par un caractère spécial nommé EP (*End of Packet*). Le routeur utilise une technique de routage par intervalles avec une stratégie de type *wormhole* **[3]**. La bande passante maximum au niveau du composant PCI-DDC est de 640 Mo par seconde. Mais le bus PCI constitue un goulot d'étranglement dans la mesure où il assure un débit maximum de 80 Mo par seconde.

Le composant PCI-DDC permet de réaliser le protocole de communication d'écriture distante. Il assure l'interface entre le bus PCI et RCUBE. Il peut accéder directement à la mémoire (par accès DMA) du processeur local, en écriture comme en lecture, ce qui décharge le processeur local **[1][2]**.

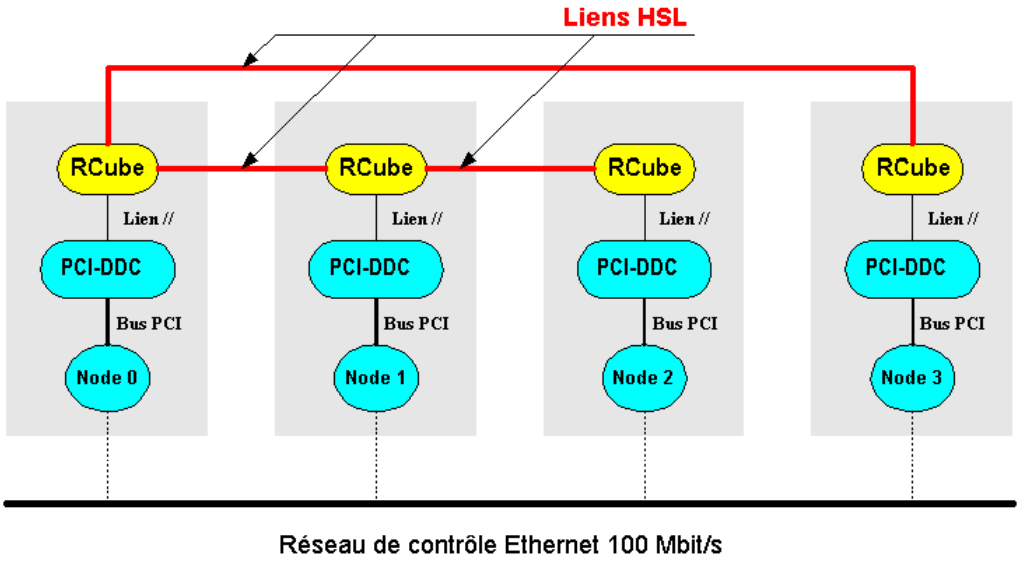


Figure 3 : Quatre cartes Fast-HSL

Le réseau ainsi constitué fournit un mécanisme de communication extrêmement efficace d'écriture en mémoire distante, qui peut être assimilé à un « Remote DMA ». L'enjeu est de faire bénéficier les applications de la très faible latence matérielle du réseau, en minimisant le coût des couches logicielles. La machine MPC promet donc de très bonnes performances via son réseau d'interconnexion haut-débit.

II. Architecture logicielle

II.1 L'écriture distante

II.1.1 Principe de l'écriture distante

Le principe général de l'écriture distante ou « Remote-write » est de permettre à un processus utilisateur du réseau HSL et local à un nœud, d'aller écrire dans la mémoire physique d'un nœud distant. Cette opération peut se faire par l'intermédiaire de PCI-DDC qui peut accéder directement à la mémoire physique locale. Il prend ses ordres dans une structure de données présente en mémoire centrale, appelée LPE (*Liste des Pages à Emettre*). Les couches de communication MPC permettent aux processus émetteurs d'ajouter dans la LPE des descripteurs de type DMA qui définissent les paramètres du transfert à effectuer, et de prévenir le composant PCI-DDC de cet ajout. Il se charge alors du transfert et signale au processeur émetteur et/ou récepteur la fin du transfert.

Chaque entrée de LPE définit une page de données à transmettre. Nous parlons ici d'une page au sens HSL : une page est une zone de mémoire physique **contiguë** dans la mémoire de l'émetteur comme dans la mémoire du récepteur.

Un descripteur de page (= une entrée de LPE) contient les champs suivants :

- Message Identifier (MI) : permet d'étiqueter les messages
- numéro du nœud destinataire
- adresse physique locale des données à émettre
- nombre d'octets à transférer
- adresse physique distante (dans la mémoire du récepteur) où écrire les données
- des drapeaux

La transmission d'un message se décompose en trois principales étapes (cf. Figure 4) :

Préparation : **(1)** Une entrée est ajoutée dans la LPE. Désormais, le processeur émetteur n'intervient plus. **(2)** PCI-DDC est informé par les couches de communication de la modification de la LPE.

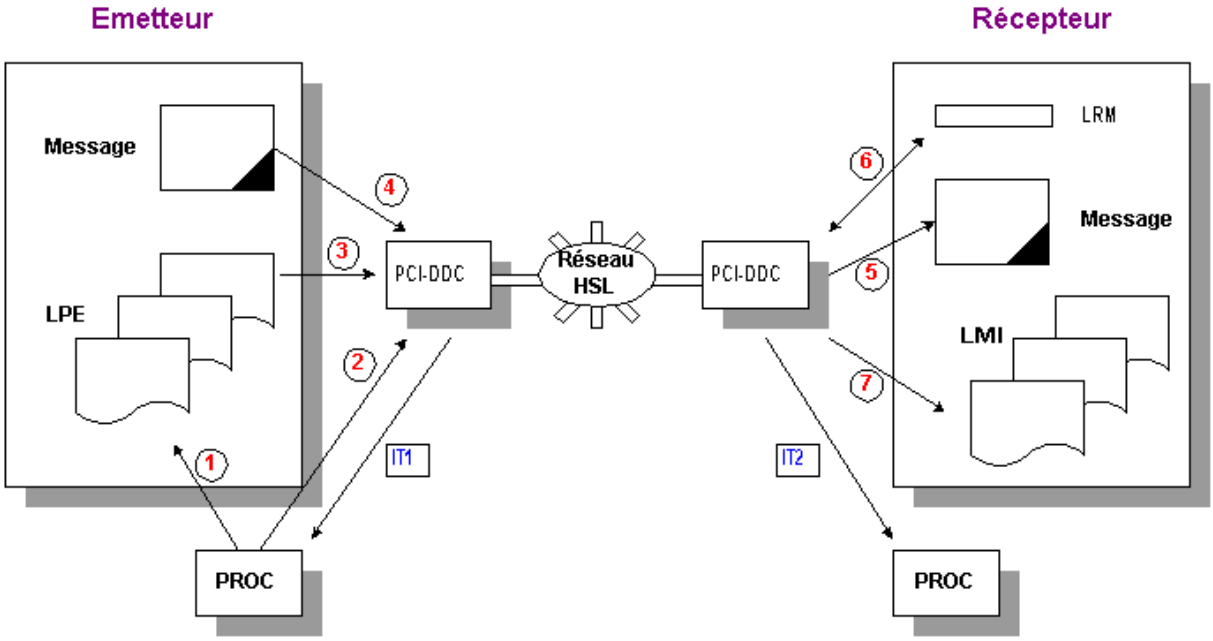


Figure 4 : Les étapes d'une écriture distante

Transmission : **(3)** Le PCI-DDC émetteur demande le bus PCI et lit le descripteur de LPE par accès DMA. **(4)** Il transmet les données, toujours par accès DMA. PCI-DDC décompose le message en plusieurs paquets qui contiennent chacun l'adresse physique distante où les paquets doivent être écrits. Quand le dernier paquet est parti, le processeur émetteur peut être prévenu par une interruption **(IT1)** (si un drapeau est positionné).

Réception : **(5)** Dès que le PCI-DDC récepteur commence à recevoir des paquets, il demande le bus PCI et écrit les données en mémoire centrale par accès DMA. **(6)** Dans les cas d'un réseau adaptatif, les paquets peuvent arriver dans un ordre quelconque (dépendant de la configuration des tables de routage) ; PCI-DDC utilise la LRM (*List of Received Messages*) pour compter les paquets reçus. Ce n'est pas le cas de la machine MPC actuelle. **(7)** Une fois que le dernier paquet a été reçu, PCI-DDC prévient le processeur récepteur soit par un signal

d'interruption **(IT2)** soit par une écriture dans la LMI (*List of Message Identifiers*). Chaque message est étiqueté par un MI.

Le message est l'entité au niveau applicatif. Un message étant défini en mémoire virtuelle, il peut être discontinu en mémoire physique s'il est réparti sur plusieurs pages. Il lui correspond alors plusieurs entrées de LPE. C'est pourquoi, il est nécessaire d'étiqueter le message par un MI afin de pouvoir reconstituer le message à la réception ; chaque descripteur de page contient le MI. Toutes ces manipulations (traduction adresses physiques en adresses virtuelles et réciproquement, gestion de MI, gestion de la LPE, etc.) sont gérées par les couches de communication MPC.

II.1.2 Avantages et inconvénients du « Remote-write »

L'avantage principal de l'écriture distante est qu'elle utilise un protocole simple. Mais surtout, ce protocole est « zéro copie ». PCI-DDC prend directement les données dans la mémoire de l'émetteur et écrit directement dans la mémoire centrale du récepteur. Cela permet de bénéficier de la très faible latence matérielle.

Un autre avantage est l'utilisation d'un lien bidirectionnel . Les communications peuvent se faire dans les deux sens, sans concurrences. Le contrôle de flux se fait au niveau des couches matérielles.

En revanche, ce protocole n'est pas classique dans la mesure où le récepteur est actif. Le modèle de communication est le passage de messages sous le mode *Receicer-Driven*. L'écriture est asynchrone vis à vis du récepteur. Autrement dit, le transfert du message se déroule sous le contrôle du récepteur. Ce modèle suppose un rendez-vous préalable entre l'émetteur et le récepteur sinon, l'émetteur ne sait pas où il peut écrire dans la mémoire du récepteur. Le rendez-vous permet au récepteur de transmettre à l'émetteur l'adresse physique dans la mémoire du récepteur. De plus, l'émetteur ne prévient pas quand il va écrire. Il écrit directement en mémoire physique ce qui suppose que les applications utilisateurs ont verrouillé à l'avance des tampons en mémoire physique. Enfin, le récepteur ne connaît pas la taille des messages qu'il reçoit.

De ce fait, il est difficile d'adapter des environnements de communication comme PVM aux couches basses MPC car la philosophie n'est pas la même. Par exemple, une émission asynchrone au niveau de PVM peut être pénalisée par le fait qu'une émission suppose d'attendre au préalable un message du récepteur indiquant l'adresse physique dans la mémoire du récepteur.

II.2 Les couches basses de communication MPC

La machine MPC fonctionne actuellement sur UNIX FreeBSD 3.1. Une pile de drivers noyaux et de démons forment les couches de communication MPC. Cet ensemble porte le nom de noyau MPC-OS. Il sera bientôt disponible sous LINUX.

Le noyau MPC-OS est l'ensemble des couches logicielles qui permettent d'utiliser au mieux les performances matérielles de la carte FastHSL. Les différentes couches de communication MPC sont représentées Figure 5.

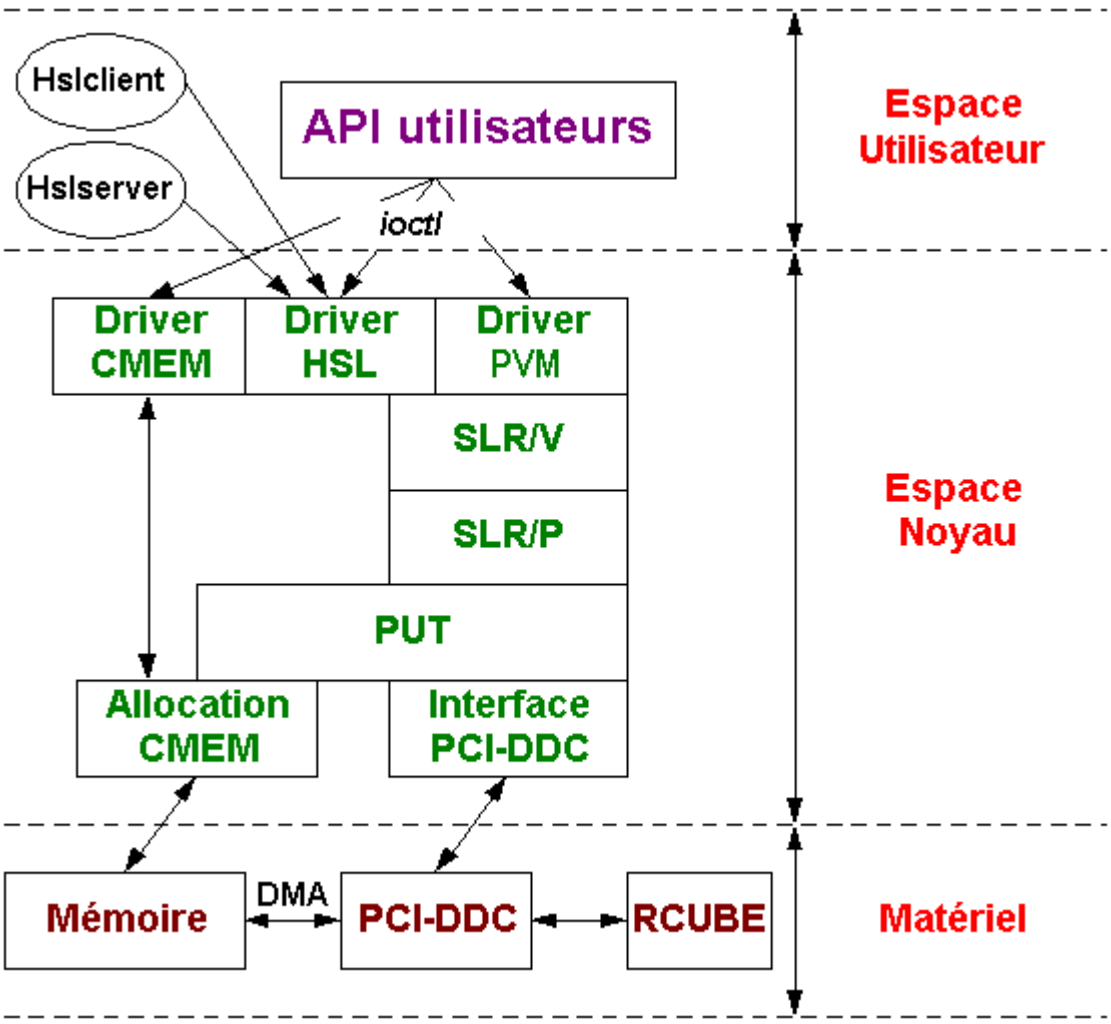


Figure 5 : Les couches de communication MPC

Allocation CMEM et driver CMEM :

Le modèle de communication de la machine MPC nécessite l'utilisation de zones contiguës en mémoire physique. C'est pourquoi, il existe un système d'allocation (CMEM) d'une plage de mémoire contiguë (mappée dans l'espace d'adressage virtuel du noyau) pour les besoins des couches de communication. Cette réservation se fait au démarrage de la machine. Le driver CMEM permet de fournir des morceaux de cette plage aux modules qui en feront la demande.

Driver HSL :

Le driver HSL permet d'accéder aux fonctionnalités des couches PUT, SLR/P et SLR/V. Il utilise de la mémoire allouée par CMEM. Il permet d'initialiser le réseau HSL car les deux démons *hslclient* et *hslserver* l'utilisent.

Les démons MPC (*hslclient* et *hslserver*) :

Ces deux démons utilisent le réseau de contrôle (ETHERNET) pour initialiser le réseau HSL. Ces deux exécutables sont lancés simultanément sur tous les nœuds de la machine MPC et dialoguent entre eux grâce à l'utilisation de RPC (Remote Procedure Call). Ils se chargent de la configuration des couches logicielles MPC sur les différents nœuds, en particulier grâce à un fichier de configuration et des tables de routage.

Les API utilisateurs :

Elles font appel aux fonctions des différents drivers par l'intermédiaire de la primitive UNIX *ioctl*.

La couche PUT :

La couche PUT constitue le service de communication de plus bas niveau et fournit la primitive d'écriture distante de l'espace d'adressage physique local vers l'espace physique d'un nœud distant. Elle est responsable de la gestion de l'accès à la LPE, de la gestion des MI (Message Identifiers). Enfin, la couche PUT joue le rôle de multiplexeur pour la distribution des interruptions.
Une version de PUT en mode utilisateur est en cours de développement au PRISM (Versailles) afin de s'affranchir des appels systèmes. Elle portera le nom « PAPI ».

La couche SLR/P :

La couche SLRP fournit un service d'échange de zones de mémoire physique, entre nœuds, sur des canaux virtuels. Elle fournit une primitive d'émission et de réception. Elle s'appuie sur la couche PUT et une utilisation particulière des identificateurs de messages (MI) pour fournir à ses utilisateurs un service de communication à travers des canaux virtuels, sur lesquels les données sont reçues en séquence (même si le réseau n'est pas adaptatif). A la différence de la couche PUT, l'utilisateur de la couche SLR/P n'a pas besoin de connaître l'adresse physique sur le nœud distant : il se contente d'indiquer le numéro de canal sur lequel il veut communiquer.

La couche SLR/V :

La couche SLRV fournit un service équivalent mais en manipulant des zones localisées dans des espaces de mémoire virtuelle. Elle fournit donc un système de traduction adresses logiques vers adresses physiques, et réciproquement. Elle sait traiter le cas de données discontiguës dans la mémoire physique car le passage en adresse physique peut engendrer une discontinuité dans les données. Avant l'utilisation de la couche SLR/V, il faut s'assurer que les données sont bien verrouillées en mémoire physique pendant toute la durée de l'opération d'émission ou de réception.

Aucune des couches logicielles ne fait de recopie des données utilisateurs. En revanche, comme ces couches sont implémentées dans le système d'exploitation, les utilisateurs y accèdent par des appels systèmes.

Chapitre III:

Un environnement de programmation parallèle : PVM

Chapitre 3 : Un environnement de programmation parallèle : PVM

Ce chapitre ne fait pas une présentation exhaustive de PVM. L'objectif est d'une part, de mettre en évidence certaines propriétés de PVM qui seront utiles dans la suite du mémoire, et d'autre part, d'expliquer comment le portage de PVM sur la machine MPC a été réalisé.

I. Généralités sur PVM

I.1 Fonctionnement général

PVM est constitué d'une librairie et d'un démon. C'est un système de communication permettant aux applications d'utiliser un ensemble de machines, potentiellement hétérogènes et de puissances variables, interconnectés par un réseau de communication, rendant ainsi les communications indépendantes du système d'exploitation.

PVM a été développé à la fin des années 80 à l'Oak Ridge National Laboratory pour le Département de l'énergie Américaine et atteint aujourd'hui une large popularité. Il est considéré comme un standard pour l'écriture de programmes parallèles.

L'ensemble des tâches est distribué sur des machines hôtes qui constituent la machine virtuelle. PVM garantit que chaque tâche de la machine virtuelle peut émettre un nombre quelconque de messages vers tout autre tâche de la machine. Ces tâches sont gérées par un ou plusieurs démons selon le type d'implémentation. Dans le cas de la machine MPC, la configuration est à démon unique, comme pour le cas de la machine parallèle Paragon d'Intel, qui gère l'ensemble des tâches de la machine virtuelle comme si elles étaient locales à son site. Ce type d'implémentation se retrouve dans les machines massivement parallèles qui disposent d'un réseau d'interconnexion rapide.

Dans cette configuration, nous distinguons la machine portant le démon (nœud de service) des autres machines du réseau (nœuds de calcul). Cette distinction nous permet d'identifier les étapes effectuées pour un échange de message dans chacun des deux cas.
Lorsqu'une tache, se trouvant sur la machine du démon, veut émettre un message, elle ne fait que l'insérer dans la file des messages à émettre, le confiant ainsi au démon. Ce dernier se charge alors de le router ou de le délivrer à sa tâche destinataire. La tâche émettrice est donc libre de retrouver son travail de fond.

I.2 Les communications PVM

Toutes les communications PVM sont asynchrones : l'émetteur n'attend pas de savoir si le récepteur a reçu le message pour rendre la main.

Les trois principales primitives de communication sont **[9]:**

- ***pvm_send****(tid,mstag)* : émission asynchrone bloquante
- ***pvm_recv****(tid,mstag)* : réception asynchrone bloquante
- *pvm_nrecv(tid,mstag)* : réception asynchrone non bloquante

où *tid* est l'identificateur d'une autre tâche PVM et *mstag* est un type de message. Ces fonctions utilisent un tampon d'émission et de réception. Des fonctions permettent la gestion de ces buffers. En particulier, ***pvm_initsend****(mode)* permet d'initialiser le tampon d'émission et de choisir si les données doivent être encodées (encodage XDR ou pas d'encodage (mode *PvmDataRaw*)). De plus, avant d'utiliser *pvm_send*, les données doivent être empaquetées dans le tampon d'émission à l'aide des fonctions ***pvm_pk****()*. De même, à la réception, il faut dépaqueter les données. Le schéma le plus classique d'un échange de données entre deux tâches (*tid1* et *tid2*) est le suivant :

Emission – tâche tid1

```
pvm_initsend(PvmDataRaw) ;
pvm_pkint(tabentiers,taille) ;
pvm_send(tid2,0) ;
```

Réception – tâche tid2

```
pvm_recv(tid1,0) ;
pvm_upkint(tabentiers,taille)
```

Dans un PVM standard (PVM sur TCP/IP par exemple), il existe un démon sur chaque nœud de calcul. Par défaut, toutes les communications passent par les démons respectifs de chaque tâche (cf. Figure 17). Il existe une connexion par socket UNIX entre chaque tâche et son démon. Les démons sont interconnectés par des liaisons UDP fiabilisées. Le démon a un rôle de multiplexeur pour toutes les tâches locales à son nœud. Cependant, il est possible de configurer PVM (mode *PVMDirectRoute*) pour qu'une tâche utilise une connexion TCP permanente avec toutes les tâches avec lesquelles elle souhaite communiquer. Ceci permet de court-circuiter les démons.

I.3 Des couches de communication PVM au réseau natif

La bibliothèque PVM est constituée de deux principales parties (cf. Figure 6). La première partie (PVM1) est indépendante du système d'exploitation. Elle regroupe toutes les fonctions de gestion des tâches PVM, des groupes, des tampons PVM ainsi que les fonctions de communication de haut niveau. La deuxième (PVM2), de plus bas niveau, comprend les fonctions de transfert des paquets sur le réseau natif. Cette partie reconnaît le système de la machine hôte et s'adapte à son architecture. C'est à ce niveau-là que vont intervenir les communications avec le réseau HSL. Le niveau de portage choisi a été la couche de communication SLR/V car elle permet d'échanger via le réseau HSL des zones de mémoire virtuelle sur des canaux virtuels.

Dans la figure ci-dessous, la partie de gauche concerne le PVM pour la machine MPC et la partie de droite concerne un PVM standard qui utilise un réseau ETHERNET.

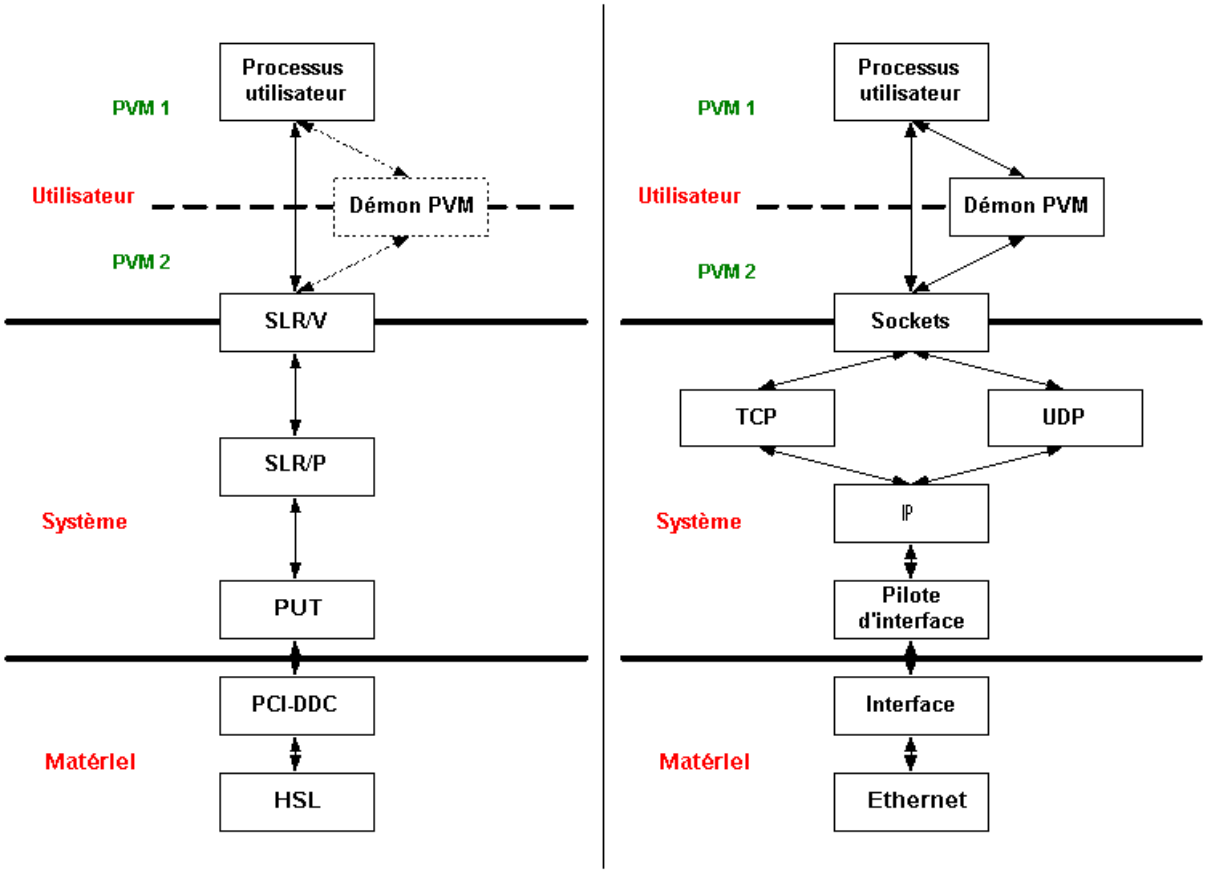


Figure 6 : Des couches PVM au réseau natif

I.4 Les recopies dans les couches hautes PVM

La fonctionnement de PVM engendre un certain nombre de recopies à l'émission comme à la réception des données. La Figure 7 représente de façon très schématique certaines de ces recopies.

La fonction *pvm_initsend* permet de préparer les structures de données qui composent le buffer d'émission. Les fonctions *pvm_pk*()* font une recopie systématique des données utilisateurs dans le tampon d'émission PVM. Les messages utilisateurs sont découpés en fragments PVM. Généralement, la taille d'un fragment est de 4 Ko . Le réseau natif est utilisé pour transmettre ces fragments. Par exemple, l'émission d'un fragment PVM-MPC correspond à une émission au niveau SLR/V. La fonction *pvm_send()* fait une nouvelle recopie des données dans les tampons d'émission du réseau natif (tampon d'émission d'une socket ou tampon d'émission SLR). Il a donc au moins deux recopies au niveau des couches PVM du côté émetteur. Il en est de même au niveau réception.

Par ailleurs, le fait de passer par le démon ou non peut également engendrer des recopies supplémentaires. De même, l'utilisation d'un encodage XDR (mode standard de PVM afin de supporter un environnement hétérogène) peut ajouter des recopies.

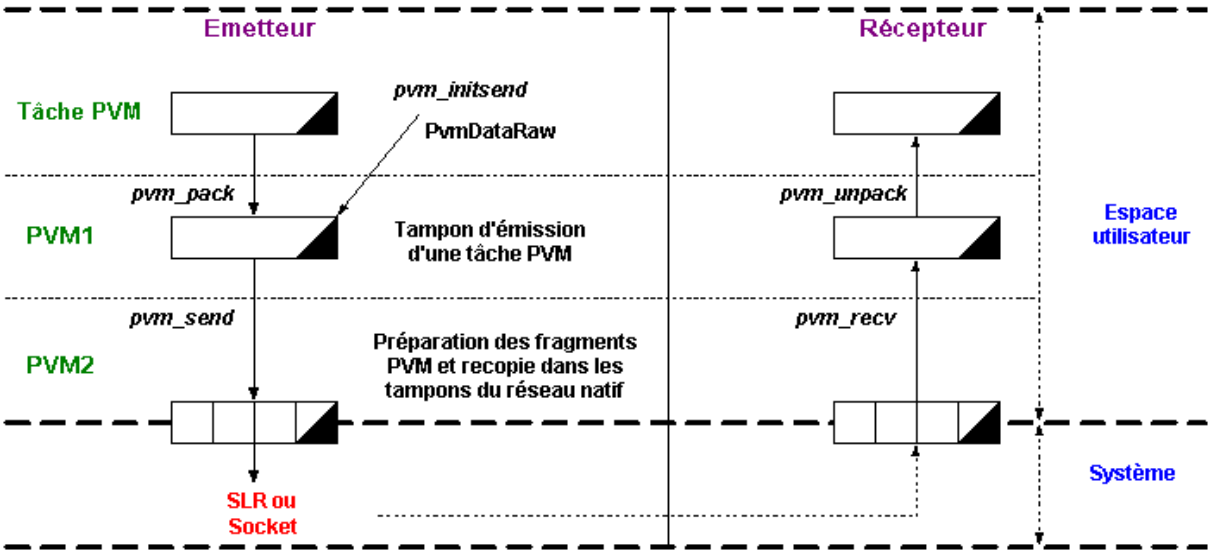


Figure 7 : Les recopies dans les couches hautes

II. PVM sur la machine MPC

Parmi les environnements de programmation parallèle, c'est l'environnement standard PVM (Parallel Virtual Machine) qui a été choisi pour la machine MPC. L'implémentation est basée sur la version 3.3 de PVM et est optimisée pour l'architecture de la machine MPC. Elle n'utilise qu'un seul démon pour toute la machine virtuelle. Celui-ci initialise la machine virtuelle et assure la gestion de toutes les tâches comme si elles étaient locales à son nœud.

II.1 Architecture générale de PVM-MPC

Le nœud sur lequel tourne le démon est appelé nœud de service ou bien nœud de démarrage **[15]** (il joue le rôle d'un frontal). Toutes les applications PVM-MPC devront être lancées sur le nœud de service. Les autres nœuds MPC sont des nœuds de calcul pour la machine virtuelle (cf. Figure 8). Cela permet de voir toute la machine MPC comme une seule machine gérée par un démon unique.

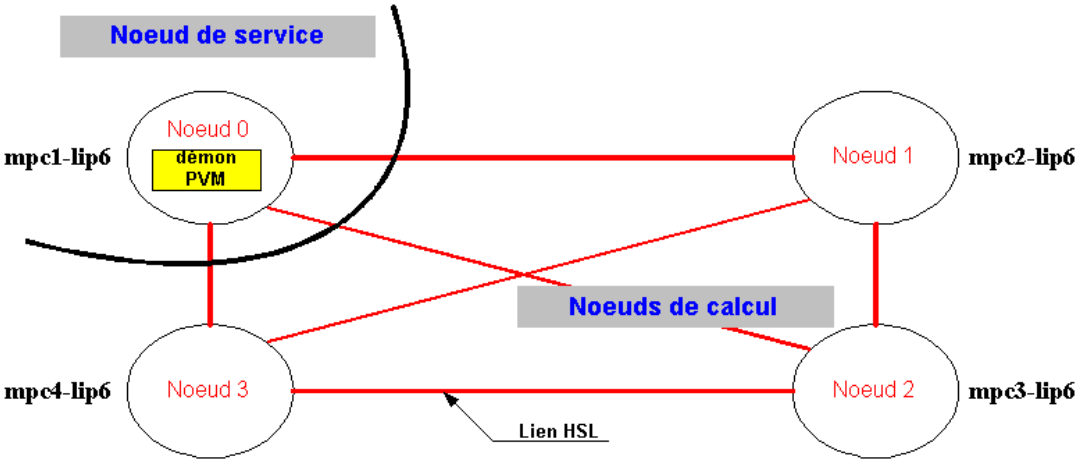


Figure 8 : Philosophie de PVM-MPC

Dans cette implémentation, une première tâche peut être lancée sur le nœud de démarrage et intégrer la machine virtuelle par l'établissement d'une connexion *Unix Socket* avec le démon. Dans la suite, nous appellerons cette tâche la tâche maître. Les tâches lancées par le démon sur le nœud de calcul seront les tâches esclaves. Par ailleurs, la seule manière de lancer les tâches sur les nœuds de calcul est d'effectuer un *pvm_spawn()* via la tâche maître, sur le nœud de démarrage. Le choix des nœuds de calcul devant supporter les tâches est laissé à la charge du démon pour un meilleur rendement. Ceci implique que, en particulier, la fonction *pvm_spawn()* a été modifié par rapport au PVM standard. Son appel est légèrement différent.

II.2 Communications entre une tâche de calcul et une tâche maître

La tâche maître se trouve sur le nœud de service. ***Toutes ses communications avec les tâches distantes se font via le démon.*** Les messages que le tâche maître veut émettre sont confier à ce dernier qui les transfère à la tâche destinataire grâce à une connexion HSL. De même les messages destinés à la tâche maître sont reçus par le démon puis délivrées à cette dernière.

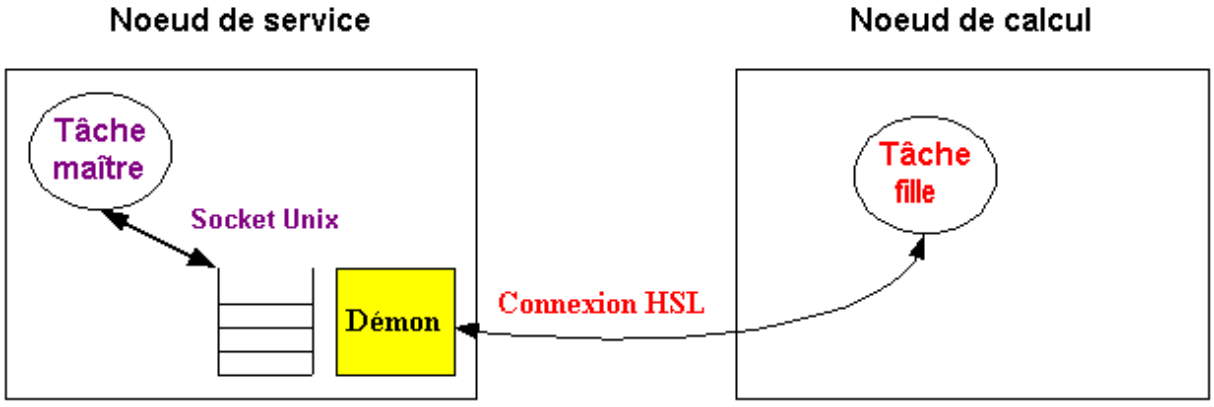


Figure 9 : Communications entre la tâche maître et une tâche fille

Le démon conserve une connexion HSL (par un canal SLR/V) avec chacune des tâches filles jusqu'à terminaison de l'application PVM. Un des inconvénients principaux de PVM-MPC est que les communications entre le démon et la tâche maître sont réalisées par Socket Unix locales ce qui ralentit considérablement les performances de la machine virtuelle (cf. *Figure 9*). Le passage systématique par les sockets Unix pour les communications avec la tâche maître pourrait constituer un goulot d'étranglement pour le reste du système.

II.3 Communications entre deux tâches de calcul distantes

Les communications entre deux tâches de calcul distantes se font par ***une connexion permanente sur un canal SLR/V***. La première opération consiste à établir cette connexion directe entre les deux tâches. Cette opération se fait par l'intermédiaire du démon. Après avoir créé deux tampons verrouillés en mémoire physique sur la tâche émettrice (un pour l'émission

et un pour la réception), cette dernière envoie une requête de connexion au démon portant le TID de la tâche à laquelle elle veut se connecter. Elle se met alors en attente d'une réponse de la part du démon. Ensuite, elle attend une acceptation de la connexion de la part de la tâche distante concernée. De son côté le démon reçoit la première requête, attribue le canal de connexion, renvoie à la première tâche le numéro du canal de connexion et envoie un message à la seconde avec le numéro de ce canal et le TID de la tâche désirant se connecter. Le démon alloue 2 tampons nécessaires à la connexion à la tâche réceptrice puis indique à la tâche émettrice qu'elle peut émettre. Chacune des deux tâches garde trace de cette connexion et l'utilise pour ses communications tout au long de sa vie.

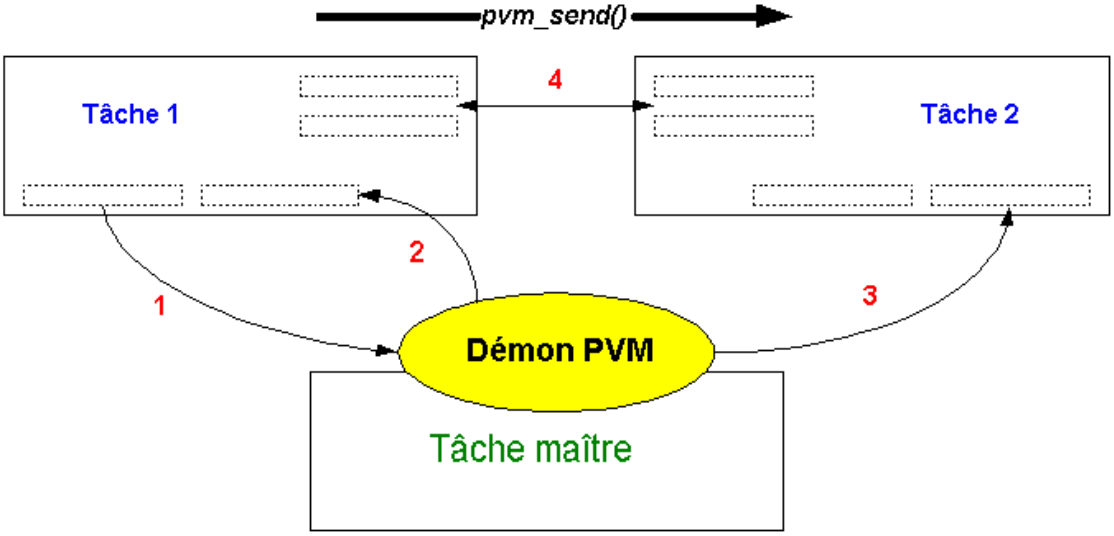


Figure 10 : Protocole de connexion HSL entre 2 tâches esclaves

Lorsque les deux tâches de calcul sont sur un même nœud, la mise sur le réseau des données est remplacée par une recopie des données au niveau local. Deux buffers sont alors nécessaires (au lieu de 4 pour des communications distantes) ; il y a un buffer par sens de communication Cependant, la connexion entre deux tâches locales s'établit toujours par l'intermédiaire du démon.

Chapitre IV:

Parallélisation de l'équation de Laplace sur une grille bi-dimensionnelle.

Chapitre 4 : Parallélisation de l'équation de Laplace sur une grille bi-dimensionnelle

Ce chapitre traite de la parallélisation de la résolution de l'équation de Laplace sur une grille bi-dimensionnelle. Il ne fait pas une liste exhaustive de toutes les façons de procéder afin d'arriver à ses fins. Dans un premier temps, je rappelle ce qu'est l'équation de Laplace et la méthode de résolution séquentielle. Ensuite, une première méthode de parallélisation est donnée (méthode SOR). Enfin, je présente une optimisation classique de cette parallélisation : la méthode « Red & Black ».

I. L'équation de Laplace

La résolution de l'équation de Laplace a de nombreuses applications dans les domaines suivants : mathématiques appliquées, mécanique des fluides, résistance des matériaux, électromagnétisme, équation de la chaleur, équation de poisson, etc. Il s'agit d'une équation aux dérivées partielles. Son équation en dimension deux est la suivante :

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

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 qui portent aussi parfois, le nom de conditions initiales. Il existe deux types de conditions limites : les conditions de *Neumann* et celles de *Dirichlet* [**17**]. Par exemple, 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 désormais 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 4 voisins, les valeurs des frontières étant fixées par les conditions initiales (cf. Figure 11, les valeurs initiales sont fixées à 100). La mise à jour s'arrête lorsque chacun des points de la grille est suffisamment proche de la moyenne de ses quatre voisins [**18**][**16**]. Il y a alors convergence.

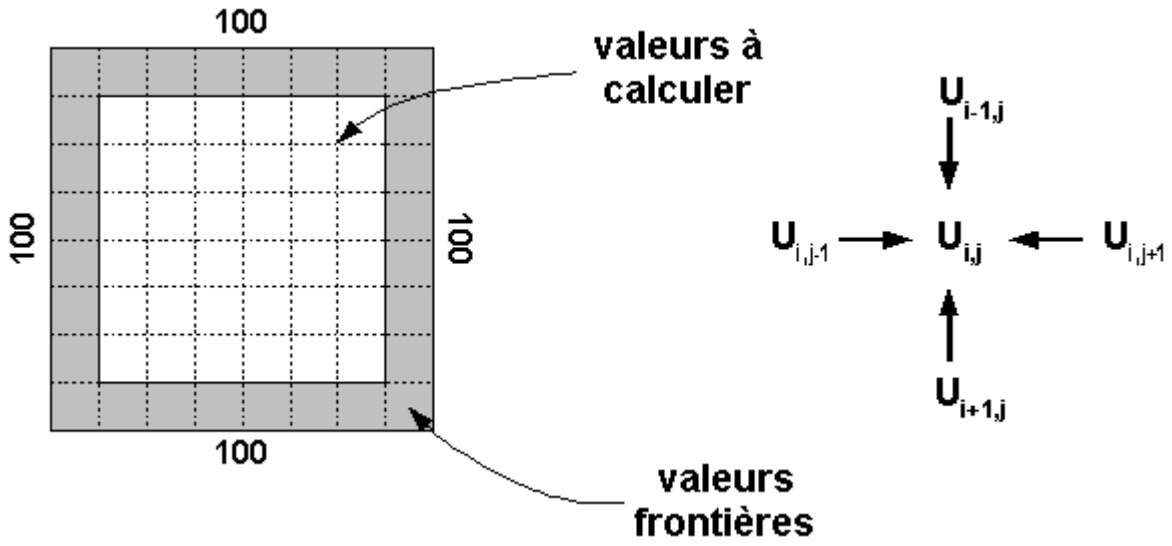


Figure 11 : Résolution de l'équation de Laplace

Les points de la grille peuvent être mis à jour suivant différentes méthodes itératives. Citons par exemple, l'***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 (i,j) après la $k^{ième}$ itération [**17**]. 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 avant chaque nouvelle itération les valeurs calculées à l'itération précédente.

Un exemple de pseudo-code décrivant la résolution séquentielle par la méthode de *Jacobi* sur une grille [N*N] est :

```
Initialisations
max_change = 0 ;
tant que (max_change > précision ou niter <= MAX_ITER)
  niter ++ ;
  max_change = 0 ; //le plus grand écart entre deux itérations successives
  pour i allant de 2 à N-1   // traitement des lignes sans les frontières
    pour j allant de 2 à N-1 // traitement des colonnes sans les frontières
      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
```

Une deuxième méthode de résolution classique est basée sur ***l'itération de Gauss-Seidel*** :

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

Le procédé est identique à la méthode de Jacobi mis à part le fait que la mise à jour des valeurs calculées à l'itération k+1 utilise les valeurs fraîchement calculées pendant cette itération (k+1) pour les voisins nord et ouest (cf. Figure 12). Il n'est donc plus nécessaire de conserver les valeurs de l'itération précédente. Cette méthode est parfois appelée la ***méthode SOR*** (Successive Over Relaxation) lorsqu'on utilise également l'ancienne valeur des Ui,j pour la mise à jour.

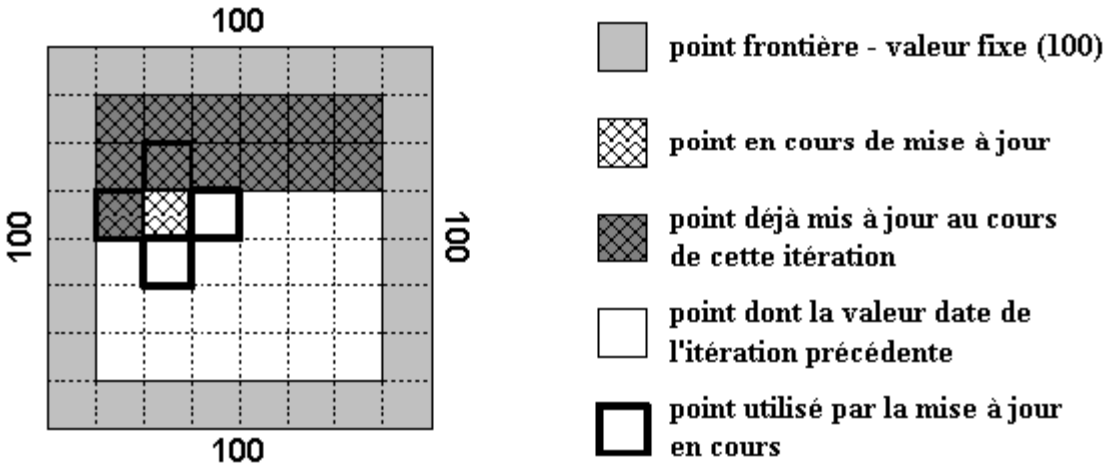


Figure 12 : L'itération de Gauss-Seidel

L'avantage de cette méthode est que la convergence est plus rapide puisqu'elle accélère la circulation de l'information sur la grille mais elle engendre un inconvénient pour la parallélisation (cf. section suivante).

II. Parallélisation par la méthode de Jacobi

La principale question est celle de la décomposition du problème sur les processus esclaves [**16**]. Une décomposition point par point ne serait pas adaptée à la machine MPC car l'utilisation d'un modèle de programmation à passages de messages (par ex. PVM) est efficace si les communications ne sont pas trop nombreuses par rapport aux calculs. Si un processus esclave ne mettait à jour qu'un seul point, il devrait effectuer huit communications (avec ses quatre processus voisins) à chaque itération et donc la synchronisation entre chaque itération serait beaucoup trop importante pour bénéficier du parallélisme. La synchronisation est réalisée par des réceptions bloquantes. L'utilisation de PVM est intéressante à la condition que le ratio temps de calcul / temps de communication est élevé.

On choisit de découper la matrice (grille [N*N]) en plusieurs bandes (M lignes par exemple) en raison de la simplicité des communications [Parallel Computer Archi]. Chacune des bandes (ou sous-grilles) est attribuée à un processeur esclave (cf. Figure 13).

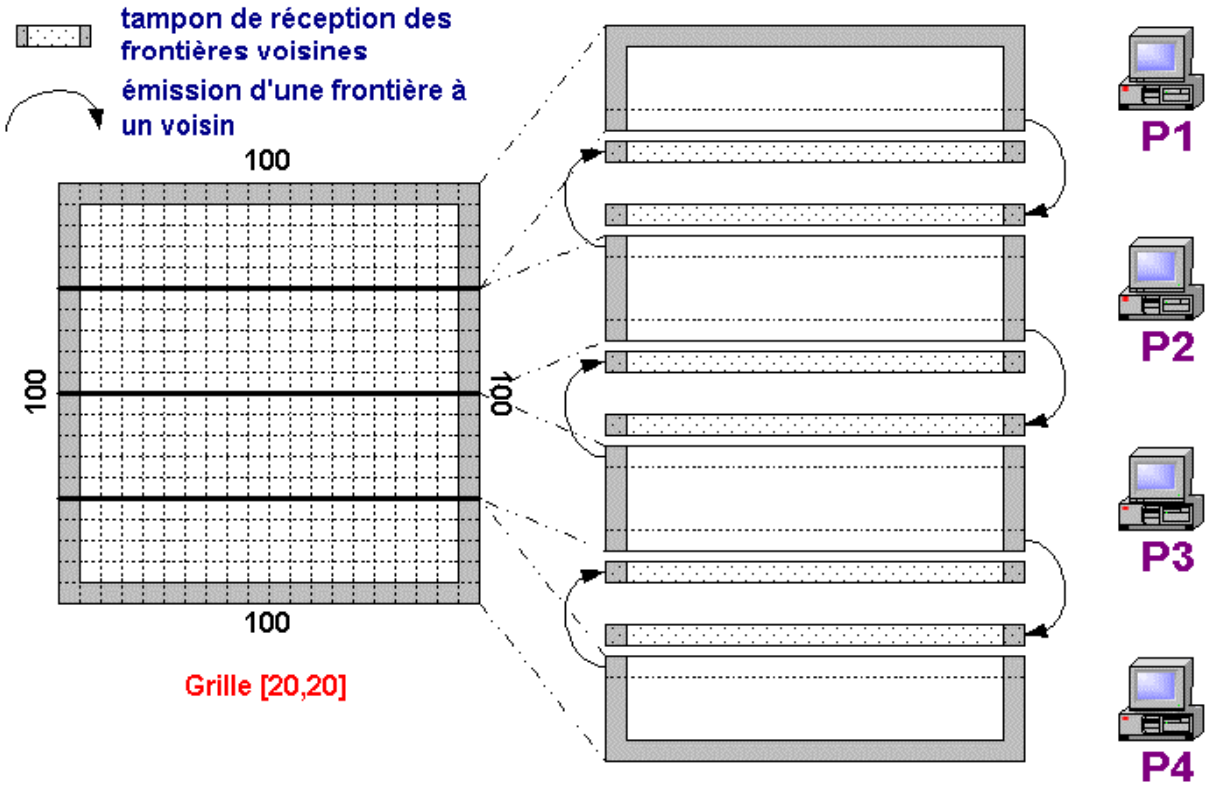


Figure 13 : Décomposition en bandes

A chaque itération, les processus échangent leurs frontières avec leur(s) voisin(s), puis mettent à jour leur partie de grille. Les processus « intérieurs » (comme P2 et P3) font quatre communications à chaque itération. Le contrôle de la convergence peut se faire par un processus maître P0 dont le rôle est de rassembler toutes les écarts de convergence locaux puis de comparer le plus grand d'entre eux à la précision souhaitée. Ce processus maître indique alors à tous les processus esclaves de continuer la mise à jour ou non. Les tests de convergence peuvent se faire uniquement toutes les 10, 100 ou 1000 itérations afin de limiter les synchronisations globales qu'ils engendrent. En revanche, il y a une synchronisation deux à deux des processus esclaves à chaque itération du fait des échanges de frontières.

Voici les différentes étapes d'un processus esclave qui utilise la méthode de Jacobi décrites dans un pseudo-code enrichi de deux primitives de communication : un *send* non bloquant et un *recv* bloquant.

```
initialisations
pour niter allant de 1 à MAX_ITER
   mylocal_change = 0 ;      //le plus grand écart localement
   recv(FrontièresVoisines)  // si niter > 1
   calcul(MesFrontières) // possible : utilise valeurs itération précédente
   send(MesFrontières)   // aux voisins
   calcul(Points_intérieurs)
   si test_convergence alors
      send(mylocal_change)    // envoi au maître
      recv(réponse)           // attendre réponse du maître
      si réponse=convergence alors quitter pour
   fin si
fin pour
```

Dans les étapes précédentes, le calcul comprend la mise à jour de *mylocal_change* et la recopie dans OldU ; d'autre part, la première réception (pour niter = 1) fait partie des initialisations.

L'intérêt de la méthode de Jacobi est qu'elle permet d'avoir un bon recouvrement temps de calcul / temps de communication car les processeurs esclaves peuvent calculer et émettre leur(s) frontière(s) dès la réception de(s) frontière(s) voisine(s) sans attendre d'avoir calculé tous les points intérieurs. On peut remarquer que ce schéma de parallélisation ne peut être appliqué à la méthode de Gauss-Seidel puisque celle-ci ne permet pas de calculer les frontières « sud » en avance. En effet, la mise à jour d'une frontière sud nécessite d'avoir calculé au préalable les points nord et ouest qu'elle utilise, c'est à dire en fait, d'avoir calculé tous les points intérieurs.

III. Parallélisation par la méthode « Red & Black »

Le principe de la parallélisation par la méthode R&B s'inspire de la parallélisation par la méthode de Jacobi et la méthode de Gauss-Seidel. Elle est basée sur la constatation suivante : supposons que la grille de résolution soit vue comme un échiquier de points rouges et noirs (cf. Figure 14) :

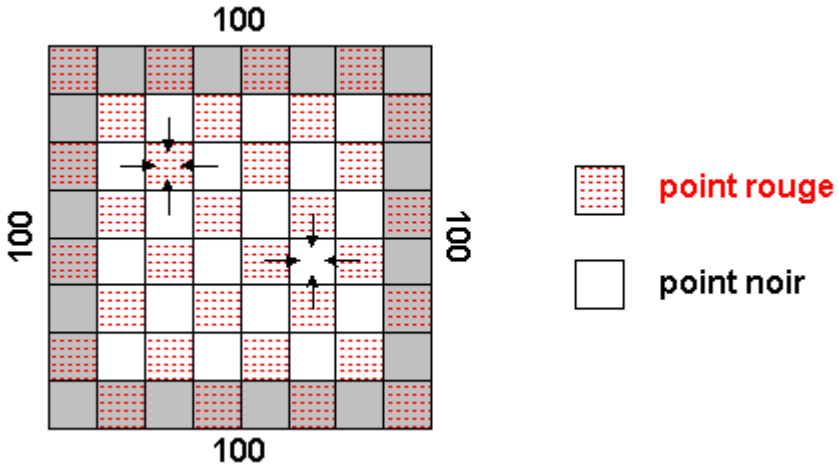


Figure 14 : La méthode Red & Black

Alors, un point rouge est systématiquement entouré par 4 points noirs et réciproquement. L'intérêt de ce quadrillage est que la mise à jour des points rouges se fait uniquement à partir des points noirs. De même, la mise à jour des points noirs utilise seulement les points rouges.

On peut alors imaginer une résolution en deux phases successives : la phase 1 consiste en la mise à jour des points rouges. La phase 2 utilise les nouvelles valeurs des points rouges pour mettre à jour les points noirs (cf. Figure 15). Chaque ligne de la grille est traitée en deux temps.

Voici les différentes étapes d'un processus esclave qui utilise la méthode R&B :

```
initialisations
pour niter allant de 1 à MAX_ITER
  mylocal_change = 0 ;      //le plus grand écart localement

  // Phase 1 : mise à jour des points rouges
  recv(FrontièresNoires_Voisines) // si niter > 1
  calcul(MesFronitèresRouges) // utilise uniquement les points noirs
  send(MesFrontièresRouges)   // aux voisins
  calcul(Points_intérieurs_Rouges) // utilise uniquement les points noirs

  // Phase 2 : mise à jour des points noirs
  recv(FrontièresRouges_Voisines) // si niter > 1
  calcul(MesFronitèresNoires) // utilise uniquement les points rouges
  send(MesFrontièresNoires)   // aux voisins
  calcul(Points_intérieurs_Noirs) // utilise uniquement les points rouges

  // test de la convergence toutes les 10 ou 100 itérations
  si test_convergence alors
    send(mylocal_change)    // envoi au maître
    recv(réponse)           // attendre réponse du maître
    si réponse=convergence alors quitter pour
  fin si
fin pour
```

Figure 15 : Les deux phases de la méthode R&B

Cette méthode combine les avantages des méthodes de Jacobi et Gauss-Seidel : elle permet de calculer les valeurs des frontières à l'avance et de les envoyer aussitôt afin de réduire les effets de synchronisation tout en évitant une recopie des anciennes valeurs avant chaque itération. De plus, la convergence est accélérée car à chaque itération, les points noirs utilisent les nouvelles valeurs des rouges.

Des études [19][20][16][17][18][21] ont montré l'intérêt de cette parallélisation.

La méthode R&B est très utilisée dans le monde du parallélisme. Elle permet de résoudre de nombreux problèmes au-delà de l'équation de Laplace. De plus, elle constitue un benchmark très utilisé (cf. [19][20]).

Chapitre V:

Mesures de performances sur la machine MPC

Chapitre 5 : Mesures de performances sur la machine MPC

Ce chapitre présente des mesures de performance qui ont été réalisées sur la machine MPC de l'I.N.T. Ces mesures sont multiples. D'une part, des mesures de latence et de débit ont été réalisées à différents niveaux des couches logicielles. D'autre part, une application parallèlisant la résolution de l'équation de Laplace sur PVM a été réalisée sur la machine MPC afin de comparer les performances entre un PVM standard et PVM-MPC.

Aucune véritable application n'a encore été portée sur PVM-MPC avant ce stage. L'objectif de ces mesures est de mettre en valeur les forces et les faiblesses de PVM-MPC et ce à différents niveaux des couches logicielles. En particulier, on essaiera de voir si PVM est bien adapté à une architecture telle que celle de la machine MPC.

I. Cadre des mesures

Les mesures ont été réalisées sur la machine MPC de l'I.N.T. dont l'architecture est équivalente à celle de la machine MPC du LIP6. Elle est constituée de 4 nœuds P2 450 MHz avec chacun 384 Mo de mémoire physique. Le réseau HSL est complet : il existe un lien HSL entre chacun des nœuds de la machine MPC. Enfin, le réseau de contrôle est un réseau ETHERNET 100 Mbits/s. Toutes les mesures ont été effectuées en l'absence d'applications concurrentes.

Théoriquement, le débit maximum au niveau matériel (sur le lien HSL) est de 1 Gbits/s et la latence matérielle est de l'ordre de 5 µs (de mémoire à mémoire) [**4**][**1**]. Cependant, ces performances ne sont pas atteintes actuellement car la première version du composant PCI-DDC possède quelques défauts :

- La LPE (Liste des Pages à Emettre) ne peut contenir qu'un seul descripteur DMA à la fois et donc PCI-DDC ne peut pas traiter plusieurs paquets simultanément.
- Normalement, PCI-DDC peut transmettre par une écriture distante des messages de 64 Ko, ce que la première version du composant ne peut pas faire.
- Enfin, il y a un problème d'alignement des octets entre l'émetteur et le récepteur.

Tous ces défauts matériels sont corrigés par les couches logicielles mais le débit maximum n'est jusqu'à présent que de 110 Mbits/s environ (le lien HSL étant bidirectionnel, il est possible d'avoir un débit de 110 Mbits/s dans chaque sens de communication). La dernière version du composant PCI-DDC ne possède plus ces défauts et devrait permettre d'obtenir des résultats bien meilleurs.

II. Les mesures de latences et de débit

Le but de ces mesures et de mettre en évidence le coût de traversée des différentes couches logicielles de PVM-MPC jusqu'à la couche PUT d'une part (réseau HSL), et d'un PVM standard à la couche IP d'autre part. (réseau ETHERNET 100 Mbits/s) (cf. Figure 6 : Des couches PVM au réseau natif).

II.1 Les programmes de tests

Plusieurs programmes ont été écrits afin de mesurer des temps de transfert pour des messages de taille variable. Le principe de tous ces programmes est, pour une taille M donnée, d'échanger 100 messages entre deux processus (cf. *Figure 16*). Le temps de transfert aller-retour de chaque message est mesuré. Une moyenne est faite sur les 100 aller-retour, puis ce temps est divisé par deux (l'écart-type n'a pas été calculé mais contrôlé à « vu d'œil »). Cela permet d'obtenir le temps de transfert d'un message de taille M. Les mesures sont faites pour des messages de 1 octet à 64 Ko.

```
// ping
pour i allant de 1 à 17
   taille = 2^i
   pour j allant de 1 à 100
      déclencher timer
      send(taille)
      recv(taille)
      arrêter timer
   fin pour
   calcul temps de transfert(i)
fin pour
```

```
// pong
pour i allant de 1 à 17
   taille = 2^i
   pour j allant de 1 à 100

      recv(taille)
      send(taille)

   fin pour

fin pour
```

Figure 16 : Pseudo-code des programmes de mesures de latence et de débit

Les mesures ont été réalisées au niveau des couches basses (SLR/V et Sockets) et des couches hautes (PVM-MPC et PVM standard). Le pseudo-code de ces programmes est celui de la Figure 16 ; seules les primitives d'émission (*send*) et de réception (*recv*) changent suivant la couche où l'on se place.

La taille des fragments PVM-MPC et PVM standard est de 4 Ko. Les tampons de réception et d'émission des sockets UNIX sont également de 4 Ko. De plus, les sockets utilisent une connexion TCP. On ne tient pas compte dans les mesures du temps d'établissement de la connexion qui est d'environ 250 µs. Enfin, les transferts de messages PVM se font dans le mode *PvmDataRaw*. Il n'y a donc pas d'encodage des données.

II.2 Tableau des temps de transfert

Les abréviations utilisées ont la signification suivante :

- **SLR** : les primitives de communication utilisées sont celles de la couche SLR/V (couche de communication MPC)
- **SOCK** : les primitives de communication sont celles des sockets avec une connexion TCP (mesures faites en dessous des couches PVM)
- **PVM-ETH** : communications entre deux tâches distantes PVM standard (cf. *Figure 17*)
- **PVM-ETH Direct** : communications entre deux tâches distantes PVM standard en mode *DirectRoute* (les communications ne passent pas par les démons ; une connexion TCP est établie entre les deux tâches distantes

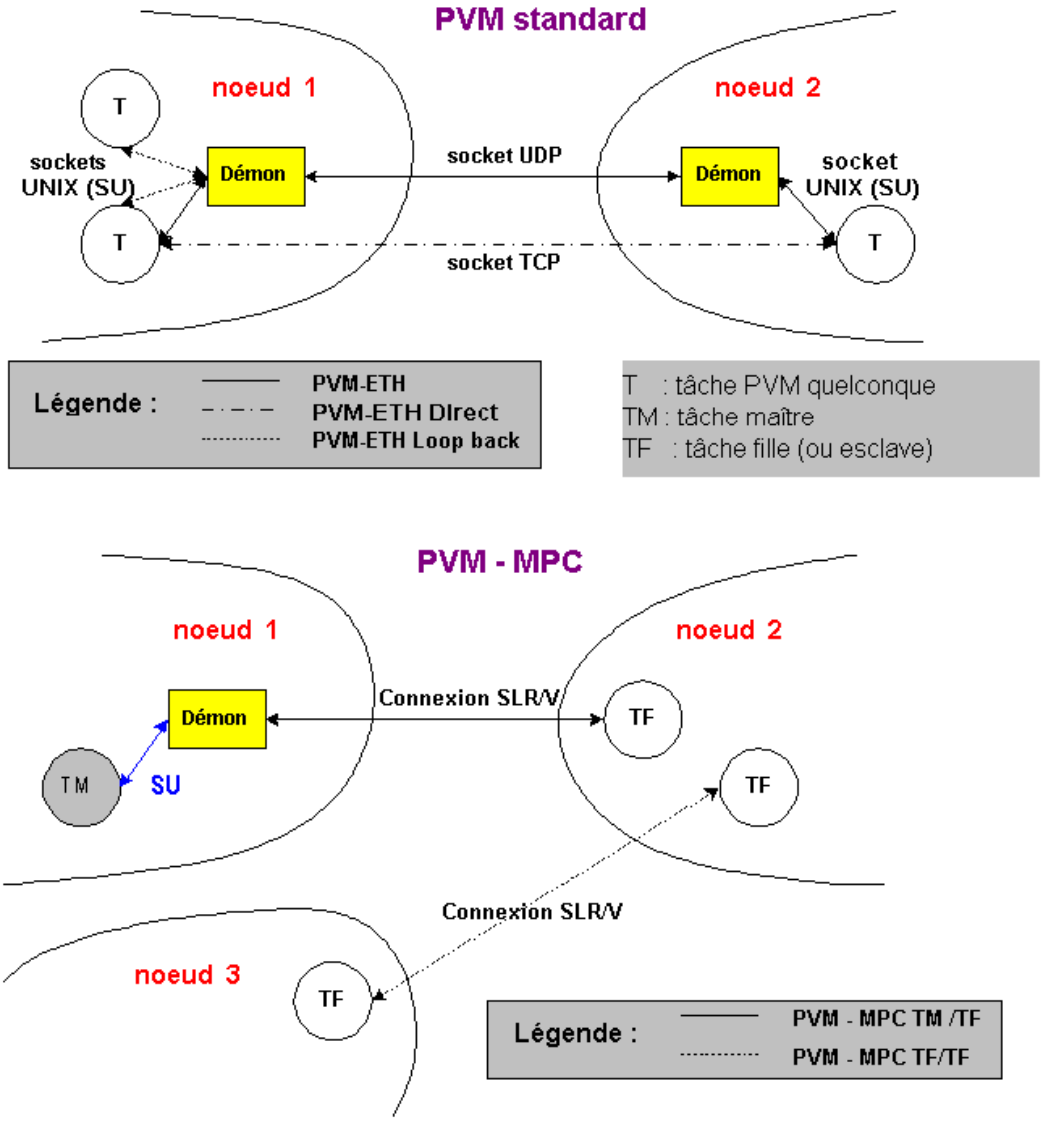


Figure 17 : Les communications PVM standard et PVM-MPC

- **PVM-ETH Loop Back** : communications PVM standard entre deux tâches locales à un même nœud

- **PVM-MPC TM/TF** : communications entre la tâche maître TM (sur le nœud du démon) et une tâche fille TF (sur un nœud distant). Les communications passent toutes par le démon unique. Les communications locales entre TM et le démon se font par une socket UNIX.

- **PVM-MPC TF/TF** : communications entre deux tâches fille distantes. Celles-ci utilisent un canal de communication SLR/V établi auparavant par l'intermédiaire du démon.

La Figure 18 ci-dessus donne les temps de transfert en micro-secondes.

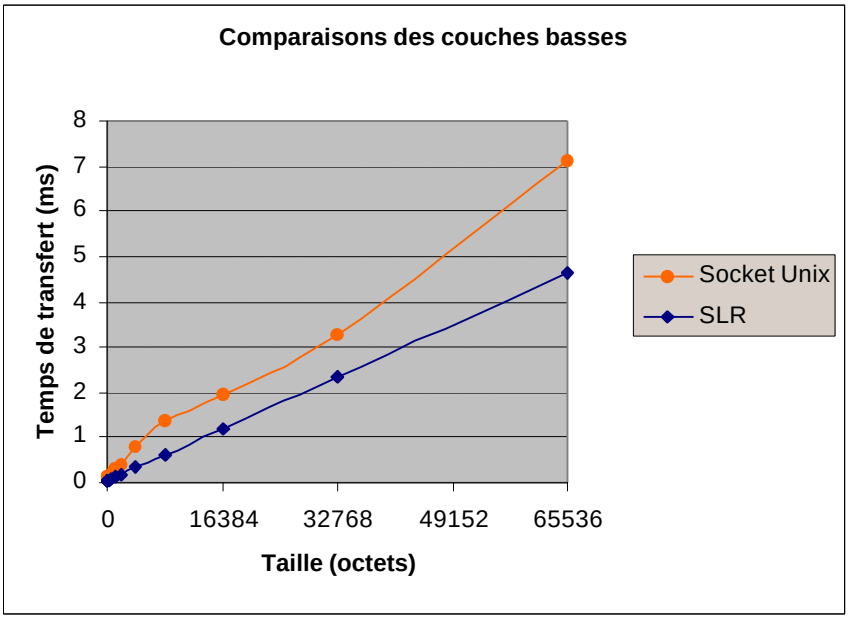
Taille (octets)	SLR	SOCK	PVM-ETH LoopB	PVM-ETH	PVM-MPC TM / TF	PVM-ETH Direct	PVM-MPC TF / TF
1	52	109	245	377	463	148	134
2	48	92	243	383	460	148	124
4	48	91	246	383	454	149	128
8	48	90	245	392	476	156	126
16	49	94	247	379	456	151	125
32	48	97	245	388	455	159	125
64	50	106	258	395	475	160	125
128	54	115	248	429	545	182	139
256	63	140	252	442	557	206	131
512	81	192	253	489	9888	260	140
1024	116	290	261	608	9978	345	150
2048	185	396	286	762	10466	486	186
4096	338	808	515	1244	9992	723	262
8192	621	1379	842	2104	19905	1120	517
16384	1195	1942	1499	3739	51339	1942	1533
32768	2342	3249	3580	7163	80123	3480	2664
65536	4642	7094	6393	16099	170159	6672	5790

Figure 18 : Temps de transfert

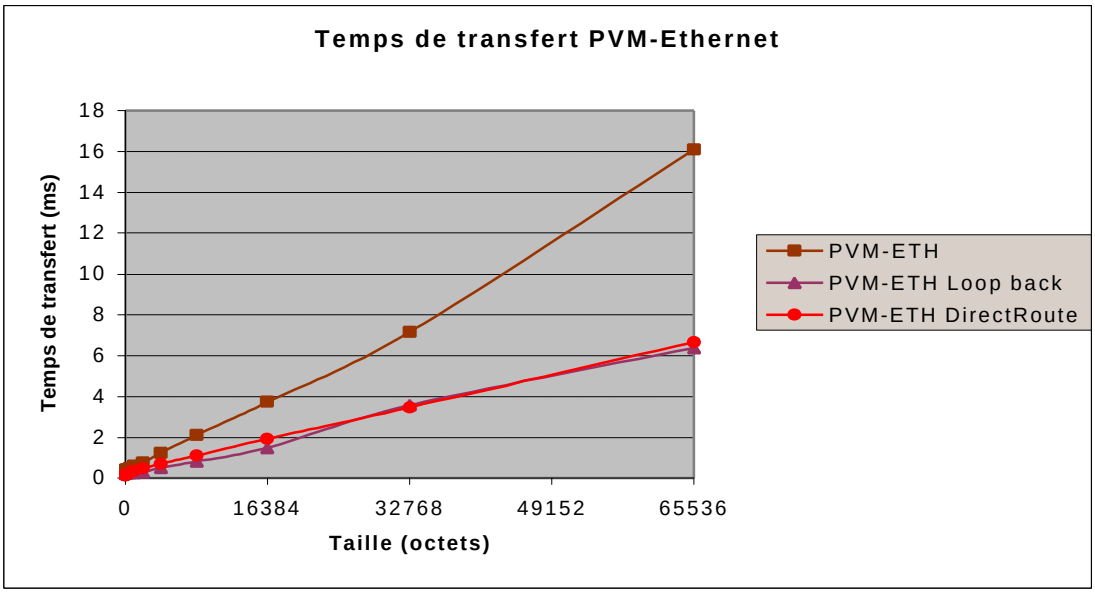
II.3 Les courbes de temps de transfert

II.3.1 Les couches basses : SLR et TCP / IP

La courbe ci-dessous nous permet de voir que les performances des couches basses de communication MPC sont meilleures que celles d'Ethernet 100Mb/s, et ce malgré les défauts actuels de PCI-DDC.

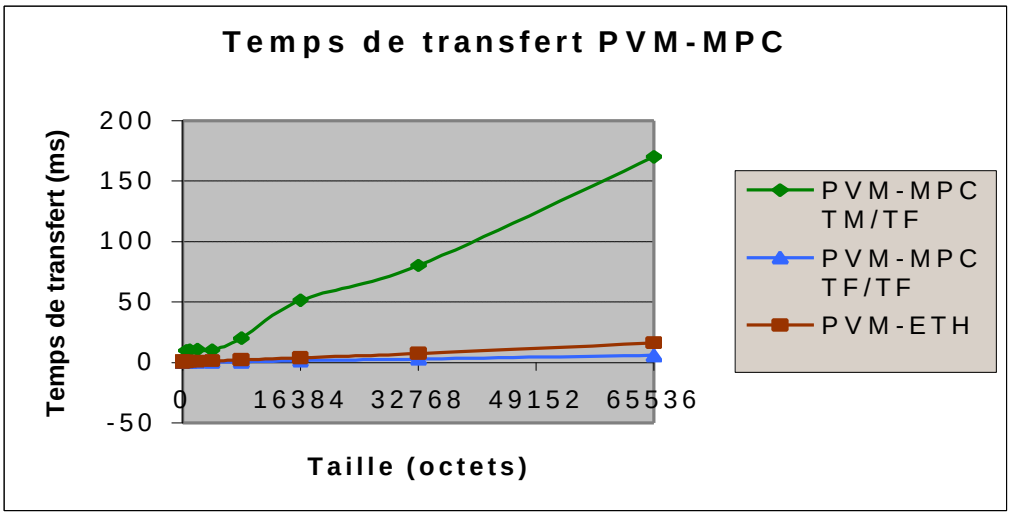


II.3.2 Les trois modes de transfert avec un PVM standard



Toutes les communications PVM – ETHERNET utilisent des sockets. On voit sur cette courbe qu'il est beaucoup plus performant d'établir une connexion directe (mode DirectRoute) entre deux tâches distantes. Cela s'explique par le fait qu'en mode normal (courbe PVM-ETH), les communications se font par l'intermédiaire des démons respectifs des deux tâches distantes, ce qui engendre l'utilisation d'une socket supplémentaire sur chaque nœud et donc des recopies et des appels systèmes supplémentaires. De la même façon, si les deux tâches sont locales à un même nœud, les performances sont meilleures car seules deux sockets locales sont utilisées et il n'y a pas de traversée du réseau physique.

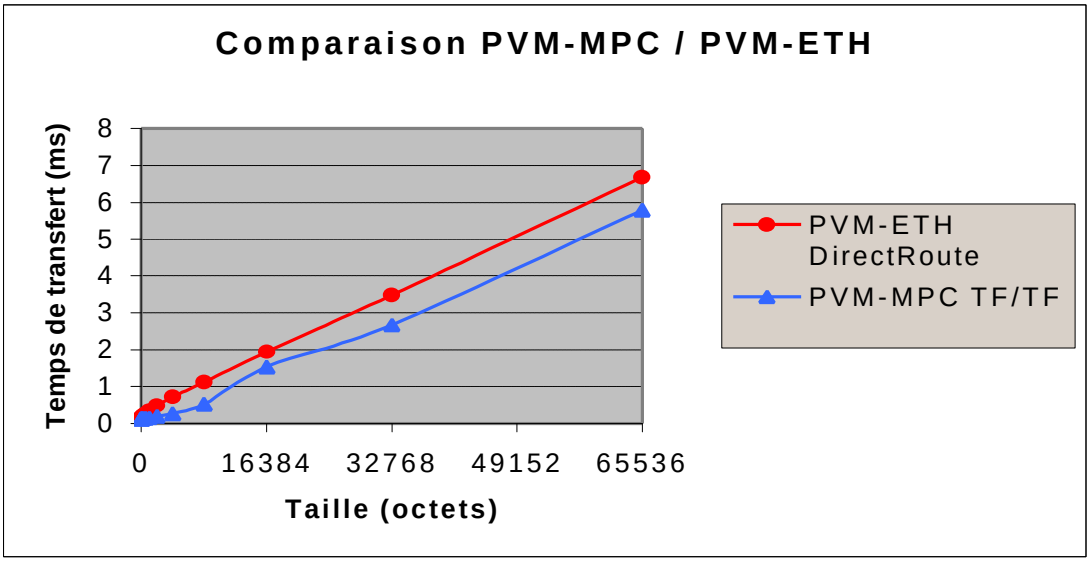
II.3.3 Les communications PVM-MPC



La courbe ci-dessus montre que PVM-MPC n'est pas encore au point en ce qui concerne les communications tâche maître (TM) / tâche fille (TF). En effet, les temps de transfert ne suivent pas une courbe linéaire mais surtout le temps de transfert d'un message de 64 Ko est 10 fois plus important qu'avec le mode normal de PVM-ETHERNET. En revanche, les performances des communications PVM-MPC entre deux tâches filles sont bonnes.

II.3.4 Comparaisons entre PVM-MPC et PVM-ETHERNET

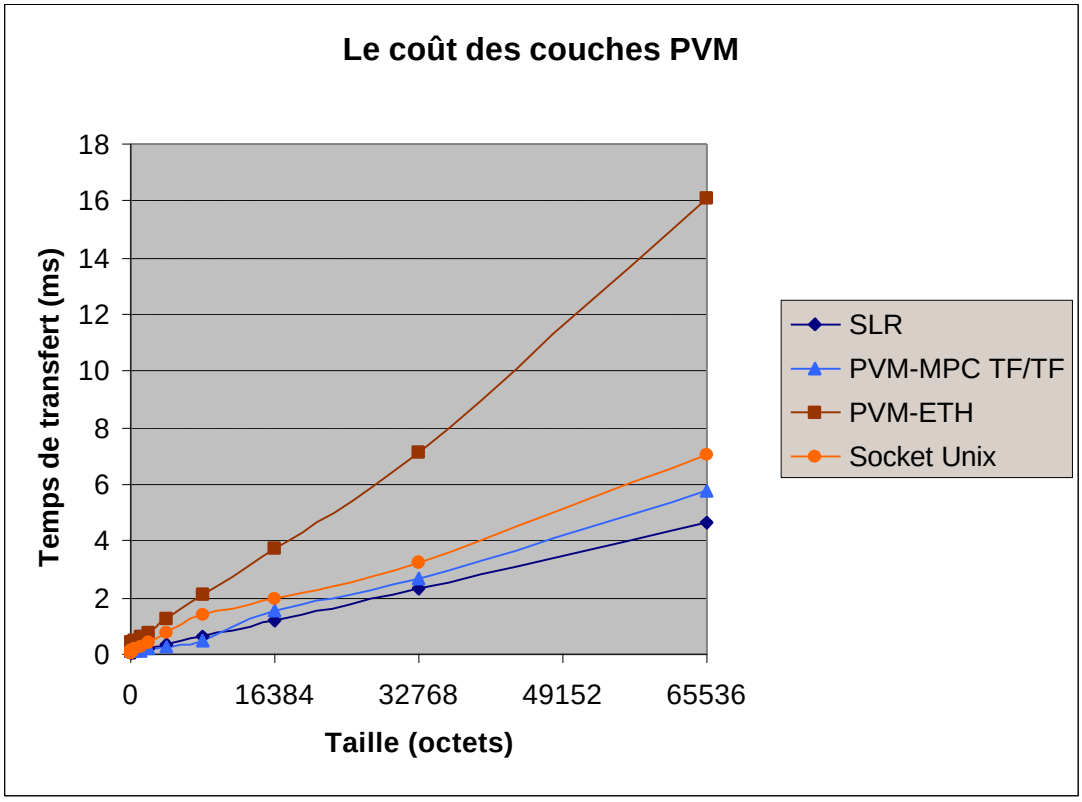
Pour que la comparaison ait un sens, on ne compare dans cette section que des communications directes. On compare l'échange de messages PVM entre deux tâches distantes qui utilisent soit une connexion SLR/V dans le cas de PVM-MPC, soit une connexion TPC dans le cas d'un PVM standard. On ne prend donc pas en compte les communications TM / TF de PVM-MPC qui ont besoin d'être revues (cf. section précédente).



On constate que les communications PVM-MPC sont meilleures que les communications PVM-ETHERNET. A partir de messages de 32 Ko, il y a quasiment une milliseconde d'écart. Ceci n'est pas surprenant dans la mesure où la traversée des couches SLR est « zéro copie » alors que celle des couches TCP/IP engendre un certain nombre de recopies. Les couches basses MPC permettent d'obtenir de meilleures performances que les couches basses ETHERNET.

II.3.5 Le coût des couches PVM

La courbe ci-dessous montre que l'ajout des couches de communication de PVM au dessus de la couche SLR/V (dans le cas du réseau MPC) ou au dessus de TCP/IP (dans le cas d'un réseau ETHERNET 100 Mbits/s) engendre une perte de performance surtout pour des messages de taille importante. Les couches PVM-MPC ajoute plus d'une milliseconde au temps de transfert d'un message de 64 Ko. Ceci peut s'expliquer en particulier par le coût de la gestion des tampons PVM et les recopies qu'elle engendre.



De même, l'ajout des couches PVM au dessus de TCP/IP abaisse très nettement les performances. Pour des messages de 64 Ko, le surcoût des couches PVM est de plus de huit millisecondes.

II.4 Tableau de latences et débits

Le tableau ci-dessous donne la latence en micro-secondes et le débit en Mbits/s pour chacun des types de communication vus précédemment. Ils ont été calculés par une régression linéaire (cf. *Chapitre 1 : III.3.3 Les mesures de performances*).

	SLR	SOCK	PVM-ETH LoopB	PVM-ETH	PVM-MPC TM / TF	PVM-ETH Direct	PVM-MPC TF / TF
Latence (µs)	**47**	**151**	**199**	**306**	**1864**	**203**	**78**
Débit (Mb/s)	**113**	**74**	**82**	**33**	**3**	**79**	**91**
Pente (REG)	0,07	0,1	0,09	0,23	2,5	0,1	0,087
Corrélation	1,0000	0,9935	0,9955	0,9957	0,9932	0,9991	0,9978

La ligne corrélation représente le coefficient de corrélation de la régression linéaire. Plus ce coefficient est proche de 1, et plus les points sont proches de la droite de régression linéaire. La ligne Pente (REG) est la pente de cette droite.

Les constations sur ces mesures sont les mêmes que pour les courbes de temps de transfert. On constate en particulier que le débit et la latence pour PVM-MPC TM/TF sont très mauvais. Le débit au niveau de la couche SLR est bien d'environ 110 Mbits/s. Enfin, le débit PVM-MPC TF/TF (91 Mbits/s) est bien un peu plus élevé que PVM-ETH Direct (79 Mbits/s).

II.5 Conclusions sur PVM

Les programmes de tests précédents ont permis de constater un certain nombre d'éléments sur l'environnement PVM.

Premièrement, il est clair que l'utilisation des couches hautes de PVM (que ce soit au dessus de TCP/IP ou au dessus de SLR/V) est pénalisante pour les performances. PVM est devenu un standard car il permet de marier des environnements hétérogènes mais cela a un coût du point de vue des performances. Il n'est donc pas forcément très adapté à un cluster tel que la machine MPC. De plus, quand PVM a été réalisé (dans les années 1980), les performances des réseaux de communication n'étaient pas ce qu'elles sont actuellement. Le coût des couches logicielles n'était donc pas aussi important comparativement à celui des couches matérielles.

D'autre part, les différents tests m'ont permis de voir qu'il y a quelques dysfonctionnements au niveau des couches de PVM-MPC. En effet, au-delà des très mauvaises performances pour les communications tâche maître / tâche fille, je me suis aperçu que les données transportées par PVM-MPC sont corrompues pendant le transfert pour des tailles de messages supérieures à 128 octets. Or, les tests au niveau de la couche SLR/V ne montrent aucune corruption des données.

Le portage de PVM sur MPC a été réalisé au niveau de la couche SLR/V. Cela induit à chaque émission d'un fragment PVM une synchronisation implicite entre l'émetteur et le

récepteur au niveau de la couche SLR/V [**15** (p 13)]. En effet, une émission SLR/V ne peut se faire qu'à la condition que le récepteur ait posté une acceptation du message au préalable.
Or, les primitives de PVM sont asynchrones. Ce procédé n'est donc pas forcément très adapté.

III. Mesures sur la parallélisation de l'équation de Laplace

Pour des raisons de corruptions des données avec l'environnement PVM-MPC (cf. paragraphe précédent), il m'a été impossible d'effectuer des mesures de performance concernant la parallélisation de la résolution de l'équation de Laplace. L'application tourne sur un PVM standard mais n'est pas utilisable sur PVM-MPC.

En effet, cette corruption des données empêche non seulement la convergence de l'application mais aussi le transfert des messages de services tels que les tableaux de *tid* (identificateurs de tâches PVM).

Chapitre VI:

Approche théorique d'une amélioration des performances : FAST-PVM

Chapitre 6 : Approche théorique en vue d'une amélioration des performances : FAST-PVM

Les mesures de performances décrites au chapitre 5 montrent que l'utilisation de PVM sur une architecture telle que celle de la machine MPC n'est pas très satisfaisante. Nous proposons donc une adaptation de PVM qui préserve au mieux la sémantique de PVM (sans que la compatibilité soit totale) en même temps que le niveau de performance du réseau HSL d'où son nom : Fast-PVM. L'objectif de ce chapitre est d'étudier d'un point de vue théorique le remplacement des couches logicielles actuelles (cf. Figure 6) par *FAST-PVM* pour bénéficier au maximum de la faible latence des couches matérielles.

I. Fast-PVM : une nouvelle approche

I.1 L'objectif de Fast-PVM

Fast-PVM serait une nouvelle couche de communication de la machine MPC. Elle serait disponible directement au niveau applicatif et utiliserait la couche PUT (cf. Chapitre II). L'idée de Fast-PVM est d'adapter la partie logicielle des couches de communication à la partie matérielle : l'architecture de la machine MPC est entièrement basée sur la primitive « remote-DMA » du réseau HSL. Fast-PVM doit donc s'adapter à la primitive d'écriture distante que fournissent les composants matériels et la couche PUT (cf. Figure 4) pour bénéficier de ses performances intrinsèques. L'objectif est de ne pas pénaliser les performances de la machine MPC par les couches logicielles.

En effet, nous avons vu aux chapitres III et V que les couches logicielles de PVM provoquent une baisse assez importante des performances, en particulier à cause des recopies mémoire. Par ailleurs, on ne souhaite plus utiliser la couche SLR/V car elle nécessite un mécanisme de rendez-vous entre l'émetteur et le récepteur. C'est pourquoi, Fast-PVM utilisera directement la couche PUT.

Cependant, on souhaite pour les besoins des utilisateurs que la sémantique de Fast-PVM soit la plus proche possible de celle de PVM. Plus encore, la possibilité d'une transformation rapide et automatique d'une application existante (ou d'une partie seulement) écrite avec PVM en une application Fast-PVM serait d'un grand intérêt.

Comment espérer avoir un gain de performances avec Fast-PVM ? En réduisant le chemin critique logiciel entre les applications et le réseau HSL.

Tout d'abord, Fast-PVM devrait utiliser directement la primitive PUT qui permet d'aller écrire dans la mémoire physique du récepteur en la rendant utilisable au niveau applicatif. On

rappelle que le portage de PVM sur MPC a été réalisé sur la couche SLR/V qui elle-même utilise la couche PUT (cf. Figure 5). L'utilisation directe de PUT permet d'être très proche des couches matérielles.

Ensuite, nous avons vu qu'indépendamment du réseau physique utilisé, les couches PVM engendrent un certain nombre de recopies des données (cf. *I.4 Les recopies dans les couches hautes PVM*). En particulier, le portage de PVM sur la couche SLR/V de la machine MPC conserve toutes ces recopies qui sont inhérentes au fonctionnement de PVM. En revanche, l'idée de Fast-PVM est de ne faire aucune recopie des données**.** On se rapproche ainsi de l'intérêt de l'écriture distante qui est d'être « zéro copie » ! Cependant, on perd la compatibilité totale avec PVM.

Enfin, pour réduire la latence au maximum, Fast-PVM doit permettre des communications sans appel système. C'est à dire que les communications se font directement en espace utilisateur (cf*. Figure 2 : Accès à l'interface réseau depuis un processus utilisateur*). Une conséquence directe des communications en espace utilisateur est qu'il ne doit y avoir qu'un seul utilisateur à la fois des couches de communication.

On retiendra que Fast-PVM doit permettre des communications **sans recopie** et **sans appel système**.

I.2 La sémantique de Fast-PVM

I.2.1 Fonctionnement de Fast-PVM

Afin de vérifier les propriétés de Fast-PVM énoncées ci-dessus, on enrichit la bibliothèque de programmation PVM avec 7 fonctions :

- fast_init()
- buf_map()
- buf_unmap()
- fast_send()
- fast_recv()
- fast_nrecv()
- fast_release()

La primitive d'écriture distante utilise un descripteur DMA. Elle suppose de connaître l'adresse physique du tampon de réception dans la mémoire du récepteur, ainsi que la longueur de cet espace mémoire (taille du message en octets).

Fast-PVM associerait une zone d'émission et un tampon de réception à chaque paire (identificateur de tâche distante - ***tid***, étiquette de message – ***mstag***). Il utiliserait une table associative sur chaque nœud émetteur qui contiendrait pour chaque paire (tid,mstag) l'adresse physique du « buffer » distant dans l'espace mémoire utilisateur du récepteur et l'adresse physique de la zone d'émission associé. La connaissance de ces informations permet de construire directement les descripteurs de DMA (entrées de LPE) sans protocole supplémentaire de communication. Fast-PVM n'utiliserait pas de tampon d'émission à

proprement dit : la zone des données à émettre est simplement verrouillée en mémoire physique et cette zone est non seulement l'espace d'émission mais aussi une zone de calcul. Les fonctions *buf_map()* et *buf_unmap()* permettent de modifier cette table qui se trouve dans l'espace utilisateur de l'émetteur. Il y a un et un seul buffer de réception et zone d'émission par couple (tid, mstag).

Une fois cette table constituée, la fonction *fast_send()* est alors utilisée pour faire une écriture distante en mode utilisateur. La primitive d'écriture distante existante est implémentée dans le driver HSL qui se trouve dans le noyau du système d'exploitation (cf. Figure 5). Il était donc nécessaire de faire un appel système à chaque appel de cette primitive. Ceci est une obligation dont on veut s'affranchir avec la fonction *fast_send()*. Par ailleurs, la fonction *fast_send()* rend la main dès qu'elle a ajouté une entrée dans la LPE : elle est non-bloquante.

De plus, les fonctions *fast_recv()* et *fast_nrecv()*, respectivement bloquante et non bloquante, permettraient de savoir si un message est arrivé ou non et, le cas échéant, d'attendre ou non sa réception. La compatibilité avec les fonctions correspondantes de PVM est totale.

Enfin, le rôle de la fonction *fast_init()* serait d'allouer à Fast-PVM les ressources de la carte réseau Fast-HSL, et d'empêcher des accès concurrentiels par un système de verrouillage ou de sémaphores. La fonction *fast_release()* permettrait de libérer la carte réseau et de désallouer toutes les structures de données créées par la fonction *fast_init()*.

I.2.2 Spécifications de Fast-PVM

Dans cette partie, nous spécifions ce que pourraient être les différentes primitives de Fast-PVM.

I.2.2.1 fast_init()

Cette fonction ne prend aucun paramètre. Elle est exécutée du côté émetteur et récepteur. Elle permettrait aux différentes tâches Fast-PVM de s'attribuer les ressources de la carte réseau « à la demande ». En particulier, elle serait chargée de configurer le composant PCI-DDC et de créer dans l'espace utilisateur de l'application toutes les structures de données utilisées par la carte Fast-HSL (LPE, LMI, etc.). Enfin, elle empêcherait tout accès concurrentiel au niveau de la carte réseau par un mécanisme à définir.

I.2.2.2 buf_map(nœud, tid, mstag, buf, taille)

entrées :

- **nœud** : émetteur ou récepteur
- **tid** : tâche distante PVM
- **mstag** : étiquette du message
- **buf** : adresse virtuelle des données à émettre
- **taille** : taille en octets des données à émettre

Cette fonction est exécutée non seulement sur le nœud récepteur mais aussi sur le nœud émetteur. Le paramètre d'entrée *nœud* permet d'indiquer s'il s'agit du nœud émetteur ou du nœud récepteur.

Cette fonction a les rôles suivants :

- côté récepteur : Elle verrouille une zone en mémoire physique de taille *taille* dans l'espace utilisateur de l'application et transmet l'adresse physique de ce buffer au nœud émetteur pour permettre la création des descripteurs DMA. Cette transmission peut se faire à l'aide des fonctions classiques de PVM (pvm_send() et pvm_recv()) ou bien par l'intermédiaire du réseau de contrôle.

- côté émetteur : Elle reçoit l'adresse physique du buffer de réception se trouvant sur la tâche *tid* correspondant à l'étiquette du message *mstag*. Elle verrouille en mémoire physique les données se trouvant à l'adresse *buf* et de taille *taille*. Enfin, elle ajoute une entrée dans la table associative contenant les adresses physiques des données à émettre et l'adresse physique du tampon de réception sur le nœud récepteur.

I.2.2.3 buf_unmap(tid, mstag)

entrées :
- **tid** : tâche distante PVM
- **mstag** : étiquette du message

Cette fonction est exécutée non seulement sur le nœud récepteur mais aussi sur le nœud émetteur.

Cette fonction a les rôles suivants : elle supprime l'entrée (tid, mstag) de la table associative et déverrouille les zones d'émission et de réception associées.

Les fonctions *buf_map* et *buf_unmap* font nécessairement un appel système pour verrouiller et déverrouiller des zones mémoires. L'idée de Fast-PVM est de n'utiliser ces fonctions qu'une fois pour toutes, respectivement au début et à la fin de chaque tâche locale (cf. *II. Les problèmes posés par Fast-PVM*).

I.2.2.4 fast_send(tid, mstag)

entrées :
- **tid** : tâche distante PVM
- **mstag** : étiquette du message

Cette fonction est exécutée du côté de l'émetteur. Elle fait une écriture distante en mode utilisateur à l'adresse physique correspondant au couple (*tid*, *mstag*) de la table associative. Les zones de données à émettre ont été préalablement verrouillées dans la mémoire physique du nœud émetteur par la fonction *buf_map*. L'écriture distante consiste en une écriture dans la LPE (Liste des Pages à Emettre) de PCI-DDC. La fonction *fast_send* est **non bloquante** dans

la mesure où elle retourne immédiatement après avoir ajouté une entrée dans la LPE. Les données peuvent être modifiées entre l'instant où le *fast_send* est exécuté et l'instant où les données sont effectivement transmises par le composant PCI-DDC. *Fast_send* n'engendre aucune recopie des données et aucun appel système.

I.2.2.5 fast_recv(tid, mstag)

entrées :

- **tid** : tâche distante PVM
- **mstag** : étiquette du message

Cette fonction est exécutée du côté du récepteur. Lorsqu'un message a été transmis par le réseau HSL à l'aide d'une écriture distante, une entrée est ajoutée dans la LMI (Liste des Message Identifiers, cf. Chapitre 2 : II.1.1 Principe de l'écriture distante) du récepteur par PCI-DDC. La fonction fast_recv fait une scrutation de la LMI. Fast_recv est **bloquante** : elle attend qu'un message correspondant au couple (*tid*, *mstag*) soit arrivé pour rendre la main.

I.2.2.6 fast_nrecv(tid, mstag)

entrées :

- **tid** : tâche distante PVM
- **mstag** : numéro de message

Cette fonction est exécutée du côté du récepteur. Elle fait exactement la même chose que *fast_recv* à la différence près que *fast_nrecv* est **non bloquante** : elle rend la main aussitôt en retournant « 1 » si un message est arrivé et « 0 » sinon, mais ne bloque pas en attendant que le message soit arrivé.

I.2.2.7 fast_release()

Cette fonction ne prend aucun paramètre. Elle est exécutée du côté émetteur et récepteur. Elle permet de libérer la carte réseau et de désallouer toutes les structures de données utilisées par Fast-PVM et créées par *fast_init.*

I.2.3 Différences sémantiques entre Fast-PVM et PVM

Bien que Fast-PVM tente de garder une sémantique proche de celle de PVM, il existe néanmoins plusieurs différences.

La première concerne les recopies de données. Le fait que Fast-PVM soit « zéro copie » modifie la sémantique de l'émission des données. En effet, la fonction standard *pvm_send()* est bloquante : elle ne retourne que lorsque les données ont été transmises ou bien recopiées dans l'espace système ; celles-ci n'étant plus accessibles, elles ne peuvent donc plus être

modifiées après le *pvm_send()* et le tampon d'émission PVM peut être réutilisé. En revanche, la fonction *fast_send()* ne fait aucune recopie des données ; il n'y a pas de buffer d'émission à proprement dit car l'espace de calcul des données est identique à l'espace de stockage avant émission. La fonction *fast_send()* n'est pas bloquante.

La notion « d'empaquettage » des données n'existe plus dans Fast-PVM (fonctions standards PVM : pvm_pk*). La gestion de la nature des données est à la charge de l'utilisateur.

Les fonctions *buf_map* et *buf_unmap* remplacent toutes les fonctions standards de gestion des tampons de PVM. Elles sont exécutées une fois pour toutes au début de l'application.

En revanche, les fonctions *fast_recv* et *fast_nrecv* ont la même sémantique que les fonctions standards *pvm_recv* et *pvm_nrecv*.

II. Les problèmes posés par Fast-PVM

II.1 Mise en évidence des contraintes liées à la sémantique de Fast-PVM

Le fait que Fast-PVM soit « zéro copie » et sans appel système engendre certaines contraintes pour les applications.

Les fonctions *buf_map* et *buf_unmap* font nécessairement un appel système alors que l'idée de Fast-PVM est d'éviter les appels systèmes. Il s'agit donc d'utiliser des tampons de réception « statiques » dans le sens où les applications initialisent leurs tampons une fois pour toutes au début de l'application ; ensuite, les communications entre tâches utilisent plusieurs fois ces mêmes buffers (typiquement dans une boucle).

Fast-PVM doit pouvoir traduire des adresses virtuelles en adresses physiques et, si besoin est, gérer des zones qui étaient contiguës dans l'espace d'adressage virtuel du processus et qui sont devenues discontiguës en mémoire physique. Cependant, ce problème est déjà traité en partie au niveau de la couche SLR/V.

Fast-PVM ne fait pas d'encodage des données comme cela peut se faire avec PVM (encodage XDR). En effet, un encodage de ce type est un frein pour les performances (recopies des données nécessaires). Cela suppose donc de travailler sur une architecture homogène comme l'est la machine MPC.

De plus, c'est au programmeur de gérer les données qui arrivent dans le buffer de réception. Il doit savoir quel est le type des données qu'il reçoit.

Enfin, les 2 principaux problèmes posés par la sémantique de PVM sont liés au fait qu'il n'y a aucune recopie des données et au caractère non bloquant de *fast_send*.

Le premier problème pouvant se poser concerne le nœud émetteur : si des calculs ultérieurs à un *fast_send* sont faits, ils risquent de modifier le contenu des données à émettre. Cela est illustré Figure 20. Prenons l'exemple d'un cas relativement usuel dans les applications parallèles qui est (sur chaque processeur esclave) une boucle itérative du type :

```
send(0)
pour i allant de 1 à N
      recv(i-1)
      calcul(i)
      send(i)
fin pour
```

Figure 19 : un cas fréquent

Considérons deux tâches P0 et P1 qui communiquent. Chacune calcule des données qui sont nécessaires à l'autre tâche. Le **recv** est bloquant. A chaque itération, la réception, le calcul et l'émission se font sur les mêmes données et donc dans les mêmes tampons mémoire. Déroulons séquentiellement 3 itérations de la boucle :

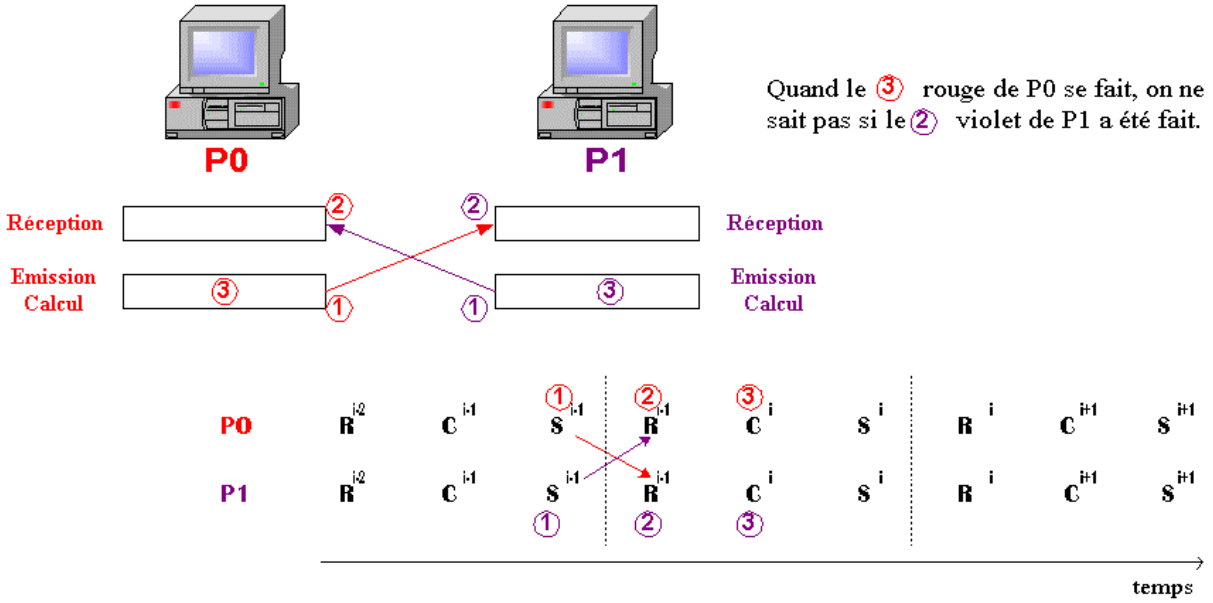


Figure 20 : Premier problème – modification des données avant l'envoi effectif

Le problème est le suivant : il se peut que les données qui sont envoyées à l'étape 1 de P0 soient modifiées par le calcul 3 de P0 avant même d'être effectivement émises car la réception bloquante 2 de P0 qui se trouve entre l'émission et le calcul suivant ne permet pas de savoir si la réception 2 sur P1 a été faite. Rappelons que le retour de *fast_send()* indique simplement qu'un descripteur DMA a été enlisté dans la LPE (émission postée) et qu'aucun mouvement de données n'est réalisé à ce moment-là.

Le deuxième problème concerne le nœud récepteur : il est lié au fait que deux émissions successives en direction du même destinataire écrivent dans le même tampon sur le nœud récepteur. Donc, il se peut fort bien que la deuxième émission soit exécutée par PCI-DDC avant que le récepteur n'ait consommé les données de la première, et le cas échéant, écrase celles-ci (cf. Figure 21).

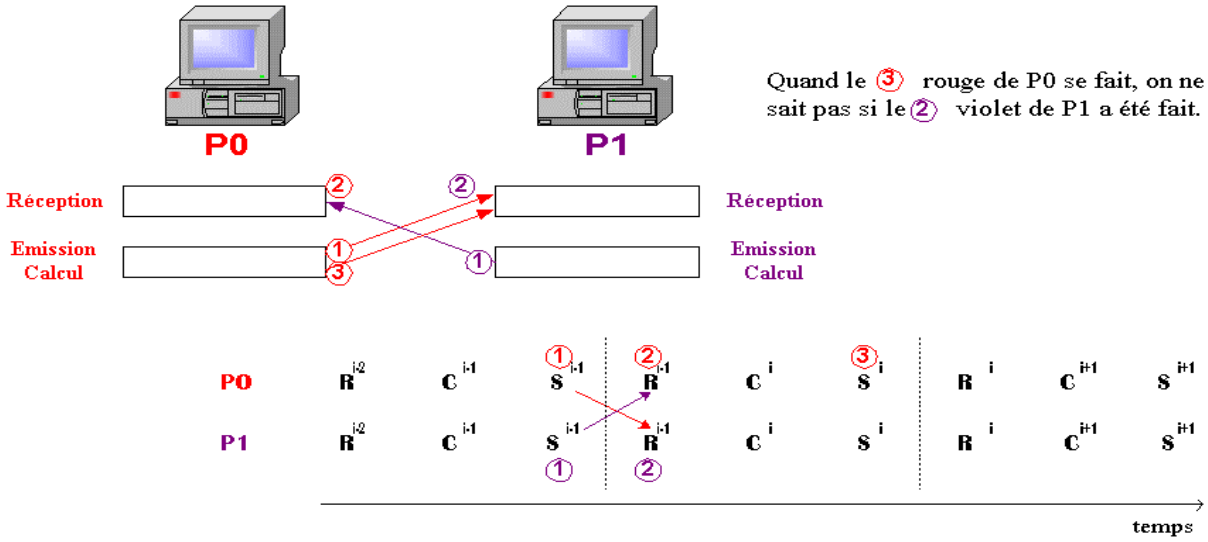


Figure 21 : Deuxième problème – écrasement des données avant la réception

L'émission 3 sur P0 peut écraser des données se trouvant dans le tampon de réception de P1 avant même que celles-ci aient été consommées par la réception 2 de P1.

Les deux problèmes présentés ci-dessus sont semblables dans la mesure où ce sont les réceptions bloquantes qui permettent d'avoir une information sur l'émission précédente.

II.2 Conséquences sur les applications : les bons cas

Il est à retenir du paragraphe précédent que certaines applications se prêtent bien à l'utilisation de Fast-PVM à condition de respecter les deux points suivants :

- Utilisation de tampons statiques : les communications entre deux tâches distantes doivent se faire au moyen de tampons **réutilisés** plusieurs fois au cours de l'application (typiquement dans une boucle). Ces tampons sont initialisés une fois pour toutes par la primitive *buf_map* afin de limiter les appels systèmes.

- Existence d'un mécanisme de synchronisation pour éviter la corruption des données : l'application doit mettre en œuvre un mécanisme de synchronisation deux à deux entre les tâches qui communiquent, et ce, afin d'éviter les deux problèmes présentés au paragraphe précédent.

Par exemple, replaçons nous dans le cas de la Figure 19. Ce type d'applications utilise effectivement toujours les mêmes tampons mémoire et nous avons vu que le mécanisme de synchronisation introduit par les réceptions bloquantes ne suffit pas à garantir la cohérence de l'application. Pour remédier à ce problème, introduisons sur chaque nœud un tampon de réception et une zone d'émission supplémentaires comme le montre la Figure 22. Le principe est le suivant : à chaque itération, P0 et P1 changent de tampons d'émission et de réception.

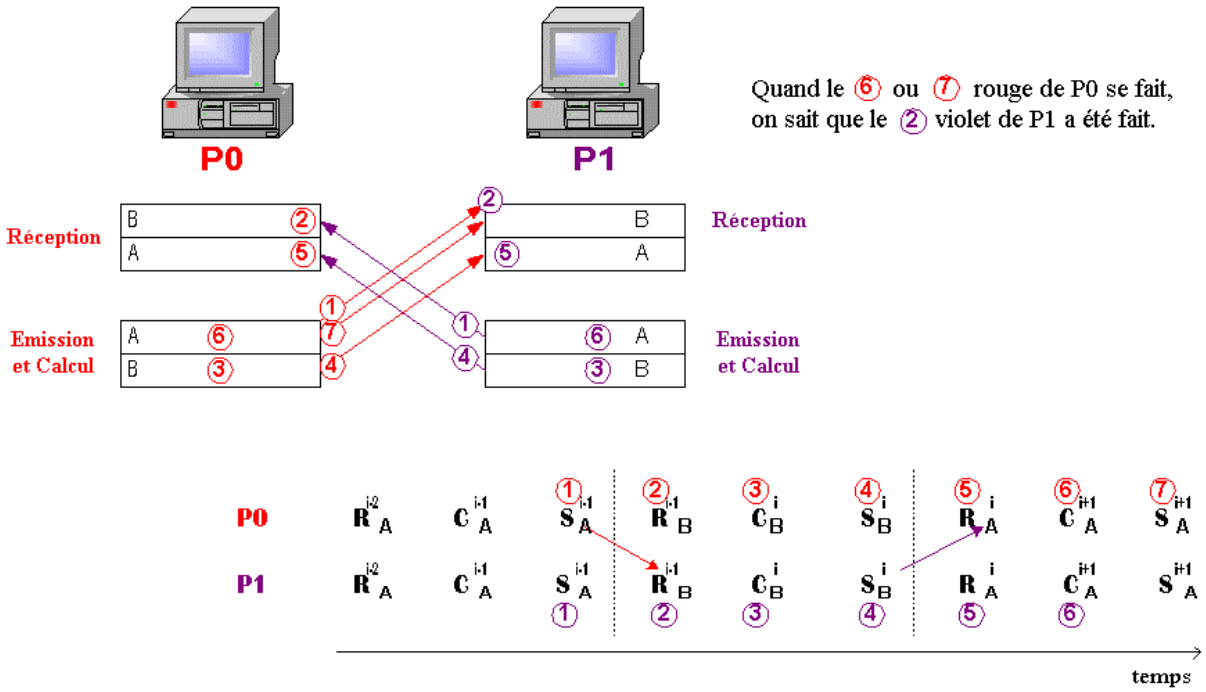


Figure 22 : Utilisation de deux tampons en réception et en émission

Cette alternance dans l'utilisation des tampons permet d'être sûr que la réception 2 sur P1 a été faite lorsque le calcul 6 ou l'émission 7 de P0 se font. Ce procédé permet à chaque itération de fournir un accusé de réception des messages de l'itération précédente.

Si l'on passe à des échanges entre plusieurs processeurs, l'analyse est la même car les problèmes de synchronisation ne se posent que pour des échanges successifs au sein d'un même couple de deux tâches distantes. En effet, si un processeur esclave communique avec deux autres processeurs distincts, il n'utilise pas les mêmes tampons mémoire pour communiquer avec l'un ou l'autre des processeurs car un tampon est associé à chaque couple (*tid*, *mstag*).

II.3 Transformation automatique PVM → Fast-PVM

Nous venons de voir que certaines applications peuvent bénéficier des performances prévisibles de Fast-PVM. Pour ces applications, il serait intéressant de disposer d'un outil de transformation automatique d'un code PVM en un code Fast-PVM. En particulier, il serait intéressant que cet outil puisse reconnaître des morceaux de code PVM se prêtant bien à l'utilisation de Fast-PVM.

La section précédente montre qu'il suffit que l'application utilise une alternance de (réception A, émission A) et de (réception B, émission B) dans des tampons A et B distincts pour être écrite en Fast-PVM. Typiquement, la méthode Red & Black de parallélisation de l'équation de Laplace satisfait ce critère (cf. Chapitre 4).

Si l'on se restreint aux applications utilisant sur chaque processeur esclave une boucle du type de celle de la Figure 19, la transformation suivante pourrait être faite :

```
send(0)
pour i allant de 1 à N
    recv(i-1)
    calcul(i)
    send(i)
fin pour
```

```
send(0)
pour i allant de 1 à N/2
    recvA(2i-1)
    calculA(2i)
    sendA(2i)
    recvB(2i)
    calculB(2i+1)
    sendB(2i+1)
fin pour
```

où les lettres A et B font référence à l'utilisation de tampons distincts comme cela a été explicité à la section précédente.

Ce type de transformation serait parfaitement adaptée à la parallélisation de la résolution de l'équation de Laplace qui suit exactement le schéma précédent en introduisant du calcul entre le *sendA* et le *recvB*. Qui plus est, cette insertion de calculs permet d'avoir un meilleur recouvrement calculs / communications.

Cependant ce type de transformation reste encore à étudier car il pose de nombreux problèmes :

Les données utilisateur d'une application PVM se trouvent dans la mémoire virtuelle du processus. Or, Fast-PVM utilise des données se trouvant en mémoire physique. La traduction des adresses (virtuelles vers physiques) reste un problème non résolu.

Par ailleurs, la transformation de la structure de boucle décrite précédemment oblige à reconnaître une telle structure, ce qui peut être parfois délicat.

Enfin, le passage de PVM à Fast-PVM nécessite que toutes les tâches de calcul qui communiquent entre elles soient réellement distantes car sinon une écriture distante ne peut pas se faire. Or, une application PVM peut avoir plusieurs tâches de calcul sur un même nœud.

Chapitre VII : Conclusions

Chapitre 7 : Conclusions

Il est maintenant temps de faire un bilan global du stage que j'ai effectué au département ASIM du Laboratoire d'Informatique de Paris 6. Il s'agit non seulement de faire un bilan du travail effectué mais aussi de voir qu'elles sont les perspectives dans un avenir à moyen terme.

I. Bilan du stage

I.1 Un outil d'administration pour la machine MPC

Le début de mon stage a consisté à réaliser un outil d'administration pour la machine MPC et un système de lancement d'applications PVM-MPC par l'intermédiaire de files d'attente afin de permettre l'exploitation de la machine par divers utilisateurs simultanément.

Cette partie du stage n'a pas été réellement abordée dans ce document car elle constituait mon stage ingénieur de fin d'études.

I.2 Adaptabilité de PVM à la machine MPC

La seconde partie du stage a consisté à analyser le fonctionnement interne de PVM. Il m'a d'abord fallu comprendre les principes généraux de PVM afin de voir comment a été réalisé le portage de PVM sur la machine MPC. L'architecture à démon unique de PVM-MPC est différente de l'architecture classique de PVM. En particulier, elle réserve un nœud au démon unique et au lancement de la tâche maître, et suppose donc de ne lancer des tâches filles PVM que sur 3 nœuds de calcul alors que la machine MPC en possède dans notre cas.

Après avoir compris les mécanismes de communication de PVM-MPC, j'ai tenté d'analyser du point de vu des performances, le coût de la traversée des différentes couches logicielles par l'utilisation de programmes de tests et de mesures de latences et débit. La comparaison aux communications d'un PVM standard utilisant un réseau ETHERNET 100 Mbits/s m'a permis de constater quelques dysfonctionnements de PVM-MPC.

Les premières mesures concernant des échanges de messages entre une tâche maître PVM-MPC et une tâche fille étaient très mauvaises voir aberrantes : il fallait plus de 18 secondes pour transférer un message de 64 Ko alors qu'il fallait pas plus de 17 millisecondes avec le PVM standard. D'autre part, les données au niveau de PVM-MPC étaient corrompues alors qu'elles ne l'étaient pas au niveau des couches basses de communication MPC (couche SLR/V). Ces constatations m'ont amené à rentrer encore un peu plus dans le code du portage de PVM sur le réseau HSL. Après avoir changer un mécanisme de synchronisation utilisant des attentes de 1 seconde au niveau du démon PVM-MPC, les performances sont devenues nettement meilleures mais les données étaient toujours corrompues.

En ce qui concerne des communications entre deux tâches filles PVM-MPC (qui utilisent une connexion HSL directe), les performances étaient à la hauteur de mes attentes. En revanche, la corruption des données étaient toujours présente.

Cela m'a permis de voir que le portage de PVM sur la machine MPC n'est pas encore tout à fait opérationnel. Cependant, les différentes mesures de temps de transfert m'ont permis de voir qu'il n'est pas forcément adapté d'utiliser des couches hautes telles que PVM au dessus des couches basses de communication MPC car les diverses recopies de données engendre un surcoût assez important. Par ailleurs, le principe de l'écriture distante n'est pas très adapté à un environnement tel que PVM car une émission asynchrone au niveau de PVM nécessite un mécanisme de synchronisation avec le récepteur dans les couches basses de communication MPC.

I.3 Etude de la parallélisation de la résolution de l'équation de Laplace

Une autre partie de mon stage a consisté à étudier les différentes méthodes de parallélisation de la résolution de l'équation de Laplace par des méthodes itératives. J'ai utilisé la méthode de Jacobi et la méthode Red & Black pour implémenter la résolution en parallèle de l'équation de Laplace sous PVM. Les résultats avec un PVM standard sur un réseau ETHERNET 100 montrent qu'il est intéressant d'utiliser PVM uniquement pour des tailles de matrices élevées (matrices 1000x1000). En effet, les méthodes de Jacobi et R&B engendrent de nombreuses communications. Cela m'a permis de vérifier que l'utilisation de PVM n'est pas adaptée à des applications qui communiquent beaucoup.

I.4 L'étude de nouvelles couches de communication

Les études précédentes ont donné lieu à une nouvelle partie de mon stage : la réflexion sur Fast-PVM qui serait une nouvelle couche de communication de la machine MPC et dont l'objectif principal est d'obtenir des performances les plus proches possibles de celles des couches matérielles. Fast-PVM permettrait aux applications utilisateurs de bénéficier de la primitive d'écriture distante tout en restant le plus proche possible de la philosophie de PVM.

Les premières réflexions montrent que cette idée pourrait très bien s'adapter à des applications bien particulières comme la résolution de l'équation de Laplace par la méthode R&B. L'idée de Fast-PVM pourrait être de reconnaître certaines portions de code dans des applications PVM existantes qui seraient bien adaptées à l'utilisation des primitives de Fast-PVM.

Cependant, de nombreux points sont encore à résoudre ou à étudier, en particulier en ce qui concerne une éventuelle transformation automatique de telles portions de code et la possibilité de partager les ressources réseaux de la carte Fast-HSL entre différents utilisateurs. En particulier, on souhaiterait pouvoir marier dans une même application des primitives PVM classiques avec des primitives de Fast-PVM.

II. Perspectives

II.1 Exploitation de la machine MPC

La première perspective à court terme est de rendre exploitable la machine MPC afin de fournir une puissance de calcul importante aux différents utilisateurs potentiels. Cette exploitation devrait se faire par l'intermédiaire de l'outil d'exploitation de la machine MPC que j'ai réalisé pendant la première partie de mon stage.

La correction des dysfonctionnements du composant PCI-DDC a permis d'obtenir une deuxième génération de cartes Fast-HSL qui sont encore en cours de tests. Ces nouvelles cartes devraient permettre d'atteindre les performances escomptées de la machine MPC avec un débit matériel maximum de 1 Gbit/s. Les nouvelles mesures de performance devraient être bien meilleures que celles qui ont été présentées dans ce document.

Enfin, il est souhaitable de fournir un environnement de programmation fiable aux utilisateurs potentiels. Le développement du portage de l'environnement PM² qui vient d'être réalisé par l'I.N.T. pourrait apporter une solution, mais il n'a pas encore été validé par l'exécution de grosse application. Par ailleurs, il est nécessaire de reprendre le portage de PVM sur la machine MPC afin de rendre cet environnement parfaitement fiable. Des études sont en cours concernant le portage de MPI et la réalisation de Fast-PVM.

II.2 Utilisation de PVM sur la machine MPC

Ce stage a permis d'étudier comment le portage de PVM sur la machine MPC a été réalisé. Cette étude montre qu'il est nécessaire de reprendre le portage afin d'une part, de régler le problème de la corruption des données lors du transfert de messages et d'autre part, d'essayer d'améliorer les performances au niveau des communications tâche maître / tâche fille.

Un autre problème du PVM-MPC actuel est qu'il ne permet pas de lancer des tâches fille sur le nœud de service (nœud sur lequel tourne le démon). Ceci est très pénalisant surtout sur les machines MPC actuelles qui ne comportent que 4 nœuds : il y a une perte de puissance de calcul de 25%.

Enfin, nous avons vu que le fait d'utiliser les couches hautes de PVM restera pénalisant sur la machine MPC à cause, en particulier, des nombreuses recopies de données. Par ailleurs, PVM n'est pas adapté aux applications qui utilisent beaucoup de communications. C'est pourquoi il serait intéressant d'offrir aux utilisateurs un environnement tel que Fast-PVM qui pourrait s'intégrer à l'environnement PVM actuel.

II.3 Validation de Fast-PVM

Nous avons tracé les grandes lignes (Chapitre 6) d'un environnement de programmation qui serait mieux adapté à l'architecture de la machine MPC et qui permettrait de réduire les coûts de traversée des différentes couches logicielles : Fast-PVM.

Il serait intéressant de réaliser une première implémentation afin de valider l'étude qui a été faite. Cependant, un certain nombre de points restent encore à étudier : la transformation automatique en Fast-PVM de certaines portions de code écrites en PVM et l'utilisation au sein d'une même application de primitives de PVM et de Fast-PVM simultanément.

II.4 Le projet MPC

Le département ASIM du LIP6 est fortement impliqué, depuis 1993, dans le développement d'une technologie d'interconnexion haute performance qui est devenue le standard IEEE 1355. Le département ASIM a été sollicité par différents industriels (Bull, SGS-Thomson, Parsytec, Thomson) pour participer à quatre projets européens visant le développement ou l'exploitation de cette technologie HSL. Les composants VLSI RCUBE et PCI-DDC, qui ont été entièrement conçus à l'Université Pierre et Marie Curie, sont actuellement utilisés dans plusieurs machines industrielles.

Le projet MPC, qui a démarré en janvier 1995 sous la responsabilité de Alain Greiner, possède un fort impact puisqu'il existe des plates-formes MPC à Versailles (PRISM), Evry (INT), Toulouse, Amiens, etc. De nombreuses équipes de recherche travaillent actuellement sur le projet MPC.

Les cartes FastHSL et les composants VLSI RCUBE et PCI-DDC sont commercialisés par la société Tachys Technologies, qui est une « start-up » du département ASIM.

Enfin, une nouvelle carte est en cours de réalisation : la carte Noé. Il s'agit d'une carte Fast-HSL utilisant un processeur intégré. Il serait intéressant de proposer un procédé de validation de cette carte en utilisant par exemple Fast-PVM.

Le projet MPC a donc encore beaucoup d'avenir devant lui. Je suis heureux de pouvoir continuer à travailler sur ce projet dans le cadre d'une thèse de doctorat que j'effectuerai avec Alain Greiner dans le département ASIM du LIP6.

Bibliographie

I. Sites Internet

Site de l'Université de **Paris 6** http://www.admp6.jussieu.fr/
Site du **LIP6** http://www.lip6.fr/
Site du département **ASIM** http://www-asim.lip6.fr/
Site de la machine **MPC** du LIP6 http://mpc.lip6.fr/ ou http://www-asim.lip6.fr/mpc

II. Documentations et ouvrages

II.1 La machine MPC

[1] *An integrated PCI component for IEEE 1355 Networks*, UPMC / LIP6
F. Wajsbürt, J.L. Desbarbieux, C. Spasevski, S. Penain, A. Greiner

[2] *PCI-DDC specifications*, version 1.3a, December 1996, UPMC / LIP6
Alain Greiner, J.L. Desbarbieux, J.J. Lecler, F. Potter, C. Spasevski, S. Penain, F. Wajsbürt

[3] *La machine MPC*, Calculateurs Parallèles, Volume 10, pages 71-84, 1998
P. David, A. Greiner, J.L. Desbarbieux, A. Fenyö, J.J. Lecler, F. Potter, F. Wajsbürt, V. Reibaldi

[4] *Noyau de communication sécurisé pour la machine parallèle MPC*, RenPar'10, 1998
P. David, A. Greiner, A. Fenyö

[5] *Conception et réalisation d'un réseau d'interconnexion à faible latence et haut débit pour machines multiprocesseurs*, Thèse de doctorat de l'Université de Paris VI, 1996
Frédéric Potter

[6] *Emulateur logiciel MPC/1 Manuel d'installation et d'utilisation*, version 3.0, Dec1996
P. David, A. Fenyö, Frédéric Potter

II.2 Modèle à passages de messages

[7] *Evolutions des mécanismes de communication par passage de messages dans les architectures parallèles*, LRI
F. Cappello

[8] *Architectures parallèles à partir de réseaux de stations de travail : réalités, opportunités et enjeux*, LRI
F. Cappello, O. Richard

[9] *PVM 3 USER'S GUIDE AND REFERENCE MANUAL*, ORNL, septembre 1994
A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam

[10] *The PVM Concurrent Computing System : Evolution, Experiences, and Trends*
V.S. Sunderam, G.A. Geist, J. Dongarra, R. Manchek

[11] *Message-Passing Performance of Various Computers*, ORNL, janvier 1997
J. Dongarra, T. Dunigan

[12] *The performance of PVM on MPP Systems*, ORNL, août 1995
H. Casanova, J. Dongarra, W. Jiang

[13] *PVM List of Frequently Asked Questions*, v1.6, juin 1995

[14] *Overview of PVM and MPI*, ORNL
J. Dongarra

[15] *Portage de PVM sur la machine MPC*, UPMC / LIP6, octobre 1998
K. Mana

II.3 Parallélisation de la résolution de l'équation de Laplace

[16] *Parallel Computer Architecture*, Morgan Kaufmann publishers, INC, 1999
David E. Culler, Jaswinder Pal Singh

[17] *Parallel Numerical Algorithms*, Prentice Hall
T.L. Freeman, C. Phillips

[18] *In search of Clusters*, Second Edition, Prentice Hall
Gregory F. Pfister

[19] *Message Passing Versus Distributed Shared Memory on Networks of Workstations*, RICE UNIVERSITY, mai 1995
Honghui Lu

[20] *Quantifying the Performance Differences Between PVM and TreadMarks*, RICE UNIVERSITY
Honghui Lu

[21] *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, 1988
Geoffrey C. Fox, Johnson, Lyzenga, Otto, Salmon, Walker

Annexes :

Annexe 1 : PVM pour MPC (compléments)

I. Création d'une tâche PVM sur la machine MPC

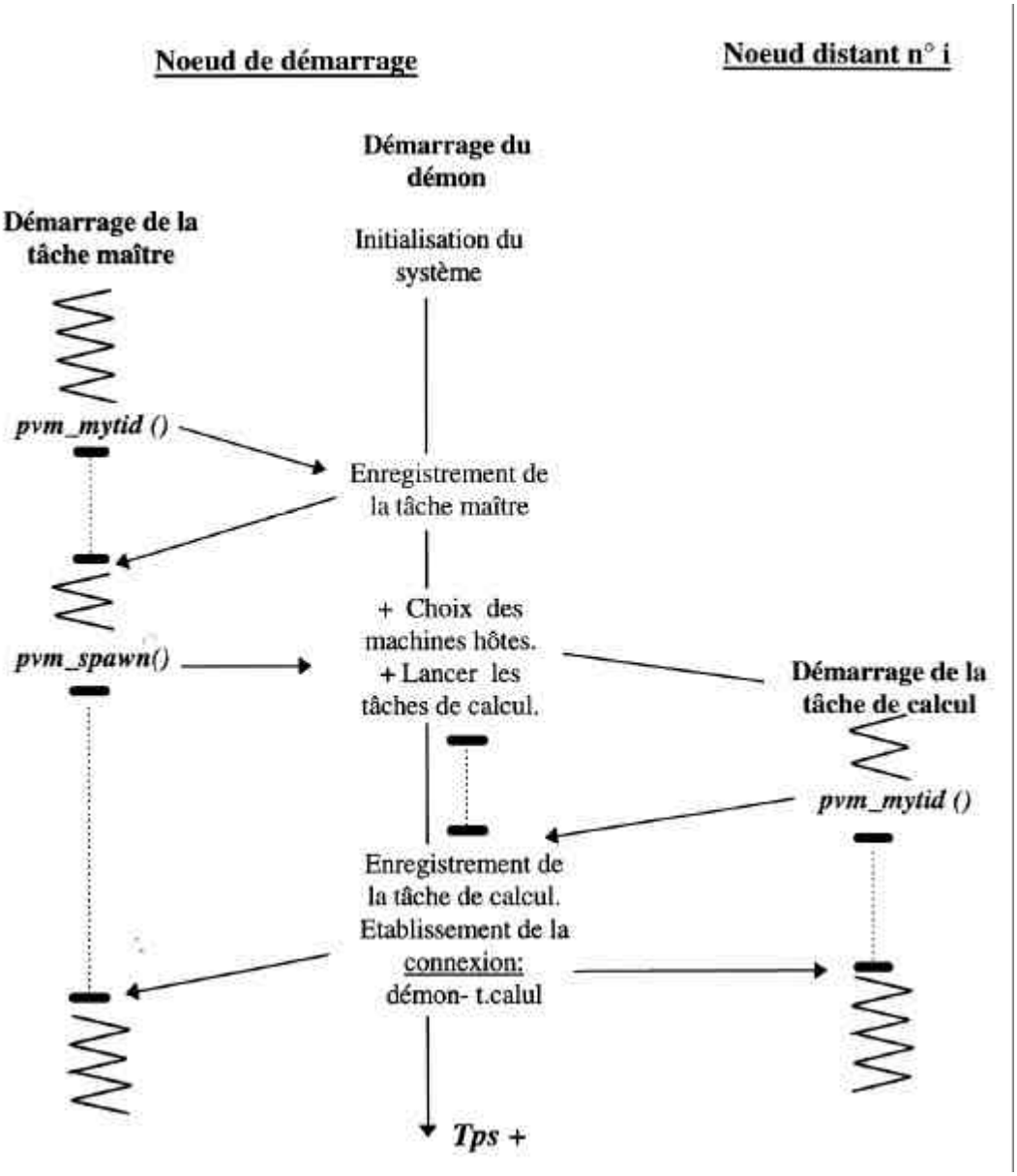


Figure 23 : Création et Initialisation d'une tâche PVM sur la machine MPC

Les tâches de calcul sont crées grâce à l'appel dans la tâche maître de la procédure *pvm_spawn()* (cf. *Figure 23*). Le démon se charge de la répartition de ces tâches sur les nœuds de calcul. La tâche de calcul est créée via l'exécution, par le démon, d'un **rsh** (remote shell) sur le nœud distant choisi pour l'accueillir. Le démon se met alors en attente jusqu'à la réception d'une demande de contexte de la part de la tâche lancée. Cette demande est produite lors de l'exécution d'un appel à la bibliothèque PVM. Le contexte de la tâche, contenant entre autres son tid (task identifier géré par PVM), lui est alors communiqué. Le tid est le seul moyen pour le démon de distinguer les tâches tournant sur la machine virtuelle. Dès lors, une connexion hsl est établie entre cette tâche et le démon. Cette connexion est maintenue tout au long de la durée de vie de cette tâche au sein de la machine virtuelle. Elle servira aux échanges de messages de contrôle avec le démon ainsi qu'aux échanges de données avec la tâche maître. Un canal de communication est négocié pour chaque nœud MPC en fonction du nombre de tâches déjà présentes sur ce dernier.

II. Initialisation de la machine virtuelle PVM-MPC

Avant d'initialiser la machine virtuelle, il est nécessaire d'initialiser le réseau HSL de la machine MPC. Nous supposons donc que les pilotes HSLDRIVER et CMEMDRIVER sont chargés et que les démons *hslclient* et *hslserver* sont lancés sur tous les nœuds.

Un autre pilote doit être chargé sur tous les nœuds pour que PVM-MPC puisse fonctionner. Il s'agit du pilote PVMDRIVER. Ce driver sert d'interface entre les communications PVM et les couches logicielles MPC. En particulier, le driver PVMDRIVER permet d'accéder aux primitives de la couche SLR/V.
La commande Unix modstat doit faire apparaître les 3 pilotes nécessaires à l'utilisation de PVM-MPC :

```
Type     Id Off Loadaddr Size Info     Rev Module Name
MISC      0   0 f5568000 0008 f5569000   1 daemon_saver_mod
DEV       1 128 f5571000 0013 f5572038   1 cmemdriver_mod
DEV       2 129 f7576000 05aa f7586058   1 hsldriver_mod
DEV       3 130 f76e4000 0008 f76e5038   1 pvmdriver_mod
```

Enfin, il faut lancer le démon PVM sur le nœud de service (nœud 0) avant de lancer des applications parallèles PVM. Désormais, la machine virtuelle PVM-MPC est initialisée.

III. Compilation d'une application PVM-MPC

L'implémentation de PVM-MPC est, rappelons le, à démon unique. Ceci implique que deux comportements différents sont possibles pour les tâches suivant qu'elles se trouvent sur le nœud du démon ou sur les nœuds de calcul. Deux librairies différentes sont utilisées suivant le cas :

- Les tâches maîtres doivent être liées avec la librairie *libpvm3.a* (librairie standard de PVM)
- Les tâches esclaves doivent être liées avec la librairie *libpvm3pe.a* (librairie PVM-MPC)

Le maître utilise la bibliothèque PVM standard. Un exemple de compilation d'une application PVM pour MPC est donné à la section suivante.

IV. Un exemple d'application PVM pour MPC

IV.1 Code source d'une application PVM-MPC

Voici un exemple très simple d'application PVM pour MPC. Il se décompose en 2 exécutables : le maître *master1* qui doit être lancé sur le nœud de service et les esclaves *slave1* qui sont lancés sur les nœuds de calcul par le démon grâce à la fonction *pvm_spawn()*.

IV.1.1 Code de master1.c

```
/* Creation  de taches de calculs recuperations des resultats */

#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"

main()
{
int mytid;            /* my task id */
int tids[24];         /* slave task ids */
int n, nproc, numt, i, who, msgtype;
float data[100], result[24];

/* enroll in pvm */
mytid = pvm_mytid();
printf("My tid  : %d\n",mytid);

/* Set number of slaves to start */
/* Can not do stdin from spawned task */
if ( pvm_parent() == PvmNoParent )
{
    puts("How many slave programs (1-24)?");
    scanf("%d", &nproc);
}

/* start up slave tasks */
numt=pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
if( numt < nproc )
{
   printf("Trouble spawning slaves. Aborting. Error codes are:\n");
   for( i=numt ; i<nproc ; i++ )
   {
         printf("TID %d %d\n",i,tids[i]);
   }
```

```
  for( i=0 ; i<numt ; i++ )
  {
        pvm_kill( tids[i] );
  }
  pvm_exit();
  exit();
  }

/* Begin User Program */
n = 100;
for ( i=0 ; i<n ; i++ ) data[i] = 1;

/* Broadcast initial data to slave tasks */

pvm_initsend(PvmDataDefault);
pvm_pkint(&nproc, 1, 1);
pvm_pkint(tids, nproc, 1);
pvm_pkint(&n, 1, 1);
pvm_pkfloat(data, n, 1);

for (i=0; i<nproc ; i++) pvm_send(tids[i],0);

/* Wait for results from slaves */
msgtype = 5;
for( i=0 ; i<nproc ; i++ )
{
      pvm_recv( -1, msgtype );
      pvm_upkint( &who, 1, 1 );
      pvm_upkfloat( &result[who], 1, 1 );
      printf("I got %f from %d\n",result[who],who);
}
 /* OK */
/* Program Finished exit PVM before stopping */
pvm_exit();
}
```

IV.1.2 Code de slave1.c

```
#include <stdio.h>
#include "pvm3.h"

FILE *fdlog;
char txt[128];
main()
{
int res;
int mytid;       /* my task id */
int tids[32];    /* task ids   */
int n, me, i, nproc, master, msgtype;
float data[100];
char  logfile[30];

sprintf(logfile,"/tmp/pvmlib.log.sla01.%d",(int) getpid());
fdlog = fopen(logfile,"w+");
if (!fdlog)  pvmlogperror("ERROR : Log file");

setvbuf(fdlog,(char *)NULL,_IONBF,0);
```

```
pvmlogerror("Log file OK!\n");
pvmlogerror("slave OK\n");

/* enroll in pvm */
mytid = pvm_mytid();

/* Receive data from master */
msgtype = 0;
pvmlogerror("Before  pvm_recv\n");
res=pvm_recv( -1, msgtype );
pvm_upkint(&nproc, 1, 1);
sprintf(txt,"Receive : number processors :  %x \n",nproc);
pvmlogerror(txt);

pvm_upkint(tids, nproc, 1);
sprintf(txt,"Receive : tid :  %x \n",tids[0]);
pvmlogerror(txt);

pvm_upkint(&n, 1, 1);
sprintf(txt,"Receive : n  :  %d \n",n);
pvmlogerror(txt);

pvm_upkfloat(data, n, 1);
sprintf(txt,"Receive : data  :  %f \n",data[0]);
pvmlogerror(txt);

/* Determine which slave I am (0 -- nproc-1) */
for( i=0; i<nproc ; i++ )
   if( mytid == tids[i] )
  {
             me = i;
             sprintf(txt,"me  :  %d \n",me);
             pvmlogerror(txt);
             break;
  }

/* Send result to master */
pvm_initsend( PvmDataDefault );
pvm_pkint( &me, 1, 1 );
pvm_pkfloat( &data, 1, 1 );
msgtype = 5;
master = pvm_parent();
sprintf(txt,"Slave : send - master :  %x \n",master);
pvmlogerror(txt);
pvmlogerror("pvm send\n");
pvm_send( master, msgtype );

/*  OK */
pvmlogerror("Pvm exit\n");
/* Program finished. Exit PVM before stopping */
pvm_exit();

fprintf(fdlog,"End of program...  slave1\n");
fclose(fdlog);
exit(0);
}
```


IV.2 Un exemple de compilation d'une application PVM-MPC

Voici, par exemple, le type de Makefile qu'il faut utiliser. Dans le Makefile suivant, la tâche maître est *master1* et la tâche esclave est *slave1*.

```
PVM_ARCH    =     MPC
ARCHLIB     =     -lrpcsvc

PVMDIR      =     ../..
PVMLIB      =     $(PVM_ROOT)/lib/$(PVM_ARCH)/libpvm3.a
PVMPELIB    =     $(PVMDIR)/lib/$(PVM_ARCH)/libpvm3pe.a
SDIR  =     ..
BDIR  =     $(PVM_ROOT)/bin
XDIR  =     $(BDIR)/$(PVM_ARCH)

CC          =     cc
CFLAGS      =     -g -I../../include
LIBS  =     $(PVMLIB) $(ARCHLIB)
NODELIBS    =     $(PVMPELIB) $(ARCHLIB)

default:    master1 slave1

$(XDIR):    $(BDIR)
      - mkdir $(XDIR)

$(BDIR):
      - mkdir $(BDIR)

clean:
      rm -f ../../bin/MPC/* *.o

#------------------------------------------>>>>>>

master1: $(SDIR)/master1.c  $(XDIR)
      $(CC) $(CFLAGS) -o master1 $(SDIR)/master1.c
$(LIBS)
      mv master1 $(XDIR)

slave1: $(SDIR)/slave1.c  $(XDIR)
      $(CC) $(CFLAGS) -o slave1 $(SDIR)/slave1.c
$(NODELIBS)
      mv slave1 $(XDIR)
```