Université de Nice Sophia-Antipolis
Laboratoire I3S
Diplôme d'Etudes Approfondies, Section Informatique

Accès à PVM par l'intermédiaire du langage ADA dans un objectif de programmation conviviale d'applications parallèles

Nathalie FURMENTO
furmento@essi.cerisi.fr

Responsables :
Mme. Françoise BAUDE
M. Daniel LAFAYE de MICHEAUX
Laboratoire I3S,
Sophia Antipolis

Rapporteur :
M. Philippe MUSSI
Laboratoire INRIA,
Sophia Antipolis

24 juin 1994

Remerciements

Je tiens à remercier mes responsables, Mme Françoise Baude et M. Daniel Lafaye de Micheaux pour l'aide et les conseils qu'ils m'ont apportés tout au long de ce stage, et également pour le temps qu'il m'ont consacrés.

Table des matières

Liste des figures

Chapitre 1

Introduction

1.1 Motivation

Ces derniers temps, les applications utilisant le calcul parallèle ont connu un essor considérable. Afin d'augmenter la rentabilité de ces applications, il serait souhaitable pour les développeurs que celles-ci soient portables sur plusieurs types de machine parallèles différentes. Mais le problème avec la plupart des architectures massivement parallèles, est qu'à chaque modèle de machine est associé un modèle de programmation et donc un certain type de langage [Paz93].

La solution proposée dans le cadre de ce stage est de fournir une librairie permettant d'écrire des applications parallèles qui pourraient s'exécuter sur des machines connectées en réseau, sans pour autant disposer d'une machine parallèle. Ceci va être possible grâce au logiciel PVM 3 (acronyme de **P**arallel **V**irtual **M**achine) qui permet de considérer un réseau de machines hétérogènes comme un seul et unique ordinateur multi-processeurs à mémoire distribuée. Il offre en effet la possibilité de gérer efficacement des tâches réparties sur les différentes machines d'un réseau (stations de travail, par exemple), et il est également disponible sur certaines machines parallèles [GBD+93]. On va ainsi pouvoir récupérer de la puissance inutilisée dans les réseaux de stations de travail existant et pouvoir s'adapter aux nouveaux multi-processeurs multi-unix.

Cependant l'utilisation de PVM 3 est d'un niveau très bas puisque par exemple, réaliser une communication nécessite l'appel à plus d'une primitive auxquelles un certain nombre de paramètres doivent être fournis. Il serait souhaitable d'avoir un langage de programmation dont la syntaxe serait plus proche de celle utilisée pour l'écriture d'algorithmes parallèles. Le langage choisi est le langage Ada.

Le but de ce stage est donc de réaliser une interface entre un langage de haut niveau — le langage Ada — et la librairie PVM 3 afin de fournir un outil adapté à l'écriture aisée d'applications portables dans le domaine du calcul parallèle orientée données.

1.2 Travail réalisé et plan du rapport

Les différentes étapes de travail réalisées au cours de ce stage sont :

- apprentissage du langage Ada,
- étude de l'environnement PVM 3 (chapitre 2),

- réalisation de la librairie Ada parallèle, et étude d'un standard de système d'échange de messages (chapitre 3),
- évaluation des temps de communication des primitives PVM 3 (chapitre 4),
- étude des communications globales que l'on pourrait rajouter à PVM 3 (chapitre 5),
- réalisation d'une application destinant à prouver l'utilité et l'efficacité de notre système (chapitre 6).

Chapitre 2

Présentation de PVM 3

2.1 Pourquoi PVM 3 ?

Comme nous l'avons dit précédemment, le choix de PVM 3 a été principalement imposé par le fait que l'on voulait pouvoir programmer des applications parallèles sans pour autant disposer de machines massivement parallèles.

La couche de bas niveau proposée par PVM 3 aurait pu également être programmée à partir du noyau Unix mais il se trouve que la sous-partie de PVM 3 qui permet d'établir des communications entre des tâches se trouvant sur des serveurs différents repose essentiellement sur le noyau Unix (sockets, RPC). Il n'y aurait donc pas eu d'intérêt à reprogrammer cette couche, et de plus l'utilisation de PVM 3 nous permet d'être indépendants du système sous-jacent. Et de ce fait, des programmes écrits en PVM 3 pourront s'exécuter sur plusieurs architectures différentes sans qu'il soit nécessaire de réécrire à chaque fois du code spécifique pour les couches de bas-niveau [GBD+93].

2.2 L'architecture PVM 3

PVM 3 est un environnement de programmation permettant le développement (écriture et exécution) d'applications parallèles, ces applications étant destinées à s'exécuter sur des réseaux de processeurs. Pour cela, PVM 3 permet de considérer un réseau de machines hétérogènes comme un seul et unique ordinateur à mémoire distribuée.

Les différents ordinateurs réunis pour constituer cette machine parallèle peuvent être de différents types (séries, parallèles ou vectoriels), ils sont interconnectés par un ou plusieurs réseaux qui peuvent eux-mêmes être de différents types (réseau Ethernet, réseau d'interconnection de machines parallèles ou réseau de fibres optiques).

Le terme *machine virtuelle* sera utilisé pour désigner l'ensemble de ces ordinateurs, et on parlera de *serveurs* pour désigner les ordinateurs eux-mêmes. Une fois la machine virtuelle définie, on va alors pouvoir y exécuter des *tâches* en parallèle. Les différentes tâches pourront communiquer entre elles à l'aide de signaux ou de messages et pourront également se synchroniser.

Le logiciel PVM 3 est disponible en deux langages :

- Le langage C, qui a été choisi dans le cadre de ce stage.

– Le langage Fortran77.

2.3 Un système hétérogène

Les applications écrites en PVM 3 considèrent le système sous-jacent comme une ressource parallèle qui fournit à la fois un mécanisme de mémoire partagée et un système d'échange de messages. Le logiciel PVM 3 est en fait hétérogène à trois niveaux différents : application, machine et réseau.

Explicitons plus précisement le premier point. Une application hétérogène est une application dont les tâches diffèrent de manière significative les unes des autres, que ce soit au niveau de leur traitement ou de leurs données. On peut par exemple avoir une application devant à la fois faire du traitement vectoriel, des calculs de type SIMD (*Single Instruction Multiple Data*) et du graphisme 3-D. PVM 3 va permettre d'exécuter chacune de ces tâches sur l'architecture la plus adaptée. Il fournit pour cela trois modes différents pour le placement des tâches :

- *mode transparent* : PVM 3 choisit parmi l'ensemble des serveurs actuellement présents sur la machine virtuelle celui qui est le plus approprié pour y exécuter la tâche (utilisation d'une heuristique basée sur la charge et le taux de performance des serveurs).
- *mode dépendant de l'architecture* : l'utilisateur spécifie une architecture (ex : SUN4, I860 ou CM2) sur laquelle la tâche doit être démarrée. PVM 3 devra alors exécuter la tâche sur l'un des serveurs de la machine virtuelle ayant cette architecture.
- *mode de bas niveau* : dans ce cas, l'utilisateur devra donner l'identificateur du serveur sur lequel il souhaite que la tâche s'exécute.

2.4 Le modèle de communications

Le modèle fourni est basé sur un système d'échange de messages. Ces messages vont être échangés à l'aide d'un programme PVM 3 — le *démon* — qui doit être installer sur chacun des serveurs de la machine virtuelle afin de les connecter entre eux.

Le mode de routage utilisé pour les communications entre tâches est le suivant : le message va tout d'abord passer par le démon du serveur de la tâche émettrice, puis sera routé vers le démon de la deuxième tâche, avant d'être finalement délivré à la tâche destinataire. Toutefois, afin d'augmenter la performance de la communication, l'utilisateur a la possibilité d'établir un lien direct entre deux tâches. Le temps de communication peut ainsi être divisé par deux ou plus, mais en contre partie, il faut savoir que le nombre de liens pouvant être établis est limité par le système Unix [GBD+93].

La figure 2.1 illustre les cas précédemment cités :

– Les tâches $2a$ et $2b$ bien qu'étant sur le même serveur communiquent à travers leur démon,

– Les tâches $1a$ et $1b$ ont établi un lien de communication direct,

– Les tâches $1c$ et $2b$ communiquent via leurs démons respectifs.

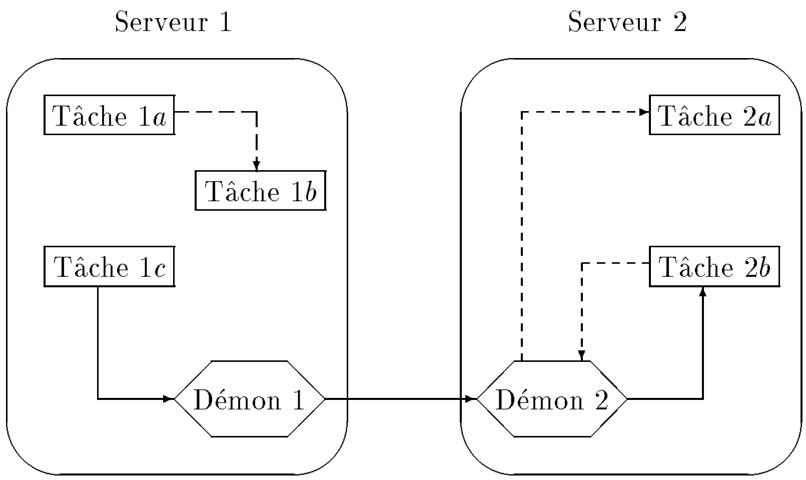

Figure 2.1: Les communications entre tâches en PVM 3

2.5 Les fonctionnalités de PVM 3

Les fonctionnalités de PVM 3 peuvent être regroupées de la manière suivante :

- gestion de la machine virtuelle,
- gestion des tâches,
- les communications,
- les groupes,
- gestion des erreurs.

On va maintenant présenter de manière plus détaillée les fonctions permettant l'échange de messages et la gestion des groupes. Une présentation plus complète de la totalité des fonctions PVM 3 pourra être trouvée dans l'annexe A et dans [GBD+93].

2.5.1 La machine virtuelle

2.5.2 Les tâches

2.5.3 Les primitives d'échange de messages

Une application parallèle écrite en PVM 3 va donc être découpée en plusieurs tâches qui vont coopérer à la résolution de l'application en s'envoyant mutuellement des données. Les primitives de communication disponibles permettent de faire de la communication point à point et de la diffusion (envoi d'un même message à plusieurs tâches). Bien que [GBD+93] dise que la diffusion soit basée sur un algorithme utilisant un arbre de recouvrement entre les différents démons de la machine virtuelle, ce n'est le cas que si PVM 3 est installé sur des machines du genre Paragon, pour les autres cas, l'algorithme est présenté en annexe B.

- **Envoi de message** : pour les deux types de communication (point à point ou globale), la primitive est asynchrone bloquante c'est-à-dire que la main est rendue au processus appelant, dès que le message est en route, de façon sécuritaire (la tâche peut alors réutiliser son buffer d'émission), vers le processus destination.

- **Réception de message** : Deux primitives sont disponibles :

  - Une primitive non-bloquante : elle se termine de suite même si aucun message n'a été reçu.
  - Une primitive bloquante : elle est bloquée jusqu'à ce qu'un message ait été effectivement reçu.

Pour envoyer un message à une tâche ou à un ensemble de tâches, il va d'abord falloir stocker les différentes données que l'on désire envoyer dans un buffer d'envoi. Ce buffer va ensuite être envoyé dans le buffer réception de la tâche destinataire qui aura alors la charge d'en extraire les différentes données.

Du fait que les deux tâches participant à la communication peuvent se trouver sur deux serveurs différents et donc utiliser des formats de données différents, il sera parfois nécessaire d'utiliser le codage XDR (*eXternal Data Representation*) qui permet d'avoir une représentation de données qui soit indépendante de la structure physique de la machine [Rif90]. Il faut noter que ce n'est pas à l'utilisateur de préciser si le codage XDR doit être utilisé ou non : ce dernier sera automiquement utilisé par PVM 3 si une configuration hétérogène a été détectée.

2.5.4 La notion de groupe

Il existe en PVM 3 la possibilité de regrouper des tâches dans un même ensemble appelé *groupe*. On pourra alors désigner l'ensemble de ces tâches en donnant uniquement leur identificateur de groupe.

Une même tâche peut appartenir à plusieurs groupes de manière simultanée, et le nombre de tâches présentes dans un groupe peut varier dynamiquement au cours d'une application.

Les groupes sont notamment utilisés pour faire de la synchronisation entre tâches, et de la diffusion (envoi du même message à toutes les tâches membres du groupe).

2.5.5 La gestion des erreurs

2.6 Les extensions de PVM 3

Il existe actuellement deux extensions importantes de PVM 3 :

- **EASYPVM** qui a pour but de faciliter l'utilisation de PVM 3,
- **DynamicPVM** qui traite le problème d'une utilisation efficace des ressources de calcul existantes lors de l'exécution d'une application PVM 3.

2.6.1 EASYPVM

EASYPVM ([Saa94]) est une bibliothèque construite au dessus de la version 3.2 de PVM, fournissant un petit ensemble de fonctions permettant de s'abstraire de certaines notions de PVM 3 pour la communication entre tâches (notamment la notion de buffer) et la création de processus.

Toutes les fonctions de communications proposées sont "génériques" sur le type de la donnée à traiter. La notion de généricité que l'on peut trouver dans cette bibliothèque n'est pas la même que celle existant en C++ ou en Ada, les langages C et Fortran 77 n'offrant pas la possibilité d'instancier de nouvelles fonctions à partir d'un modèle unique.

La solution apportée dans EASYPVM est d'appeler avant chaque fonction de communication une fonction qui va permettre de fixer le type de la donnée sur lequel on désire travailler. L'inconvénient de cette méthode, ajoutée au fait que la notion de buffer a été "cachée", est d'empêcher l'envoi de messages contenant des données de différents types.

Cette bibliothèque fournit également des routines de communications globales telles que la synchronisation[1], la réduction ou le rassemblement. Toutefois la fonction de réduction proposée ne peut être appelée qu'avec un nombre limité d'opérateurs de réduction (somme, produit, minimum, maximum) qui sont en fait définis dans la bibliothèque. Ce modèle est donc assez limité puisqu'il ne permet pas à l'utilisateur d'effectuer une réduction avec un opérateur qu'il aurait lui-même défini. Une solution à ce problème est proposé dans [BS94a] : l'idée est d'utiliser des types de données catégoriques pour pouvoir construire un modèle pour le calcul parallèle qui soit suffisamment abstrait pour être porté sur n'importe quel type d'architecture.

2.6.2 DynamicPVM

Bien que l'utilisateur ait la possibilité en PVM 3 d'indiquer sur quels serveurs (ou architectures) les tâches de son application doivent s'exécuter[2], ce placement est statique c'est-à-dire que les tâches ne pourront pas être déplacées en cours d'exécution. L'environnement DynamicPVM fournit des mécanismes de réordonnancement des tâches qui permettent d'équilibrer les charges de traitement sur l'ensemble des serveurs de la machine virtuelle et donc d'utiliser au mieux les ressources de calcul disponibles à tout moment ([DvdLVS94]).

La solution proposée est l'utilisation d'un système d'ordonnancement global, Condor, qui grâce à une méthode d'équilibrage des charges permet de démarrer les nouvelles tâches sur les processeurs les plus faiblement chargés, et de déplacer au besoin des tâches se trouvant sur des processeurs dont la charge serait devenue trop importante à la suite de changements d'environnement.

DynamicPVM a déjà été implementé avec succès sur un réseau de IBM RS/6000 et d'ordinateurs AIX32.

[1] sur l'ensemble des tâches et non juste sur un groupe

[2] voir section 2.4

2.7 Conclusion

On peut donc voir que l'environnement de programmation PVM 3 est un environnement bien adapté à l'utilisation d'un réseau de machines pour le codage d'une application parallèle.

De plus, à la différence des architectures massivement parallèles, on va pouvoir coder un ensemble plus large d'applications du fait que des machines possédant des caractéristiques très différentes puissent être regroupées dans une même configuration.

Toutefois, les implémentations actuellement disponibles de PVM 3 ont une syntaxe beaucoup trop éloignée de celle utilisée en algorithmique. C'est pour cela que proposons une interface de PVM 3 dans le langage Ada afin d'avoir une librairie de calcul parallèle qui soit à la fois efficace et facile d'écriture.

Chapitre 3

La librairie ADA parallèle

3.1 Pourquoi ADA ?

Le langage Ada a été choisi parmi les différents langages de haut niveau disponibles pour les raisons suivantes :

* Système de modules puissant,
* Langage fortement typé, modulaire,
* Utilisé par beaucoup de développeurs dans le cadre numérique,
* Syntaxe proche de celle utilisée en algorithmique.

Le parallélisme en Ada :

- On a en Ada la notion de "tâches" qui permet à l'aide de mécanismes de rendez-vous d'établir des synchronisations et des échanges de données. Mais ce système ne permet de faire que du pseudo-parallélisme et ne pourrait pas être utilisé pour l'écriture d'une librairie pour le calcul parallèle.

- Il existe déjà un système Ada parallèle sur un multi-processeur MIMD (*Multiple Instruction Multiple Data*) expérimental [Lun89]. Ce système permet à un programme Ada contenant diverses tâches de s'exécuter en parallèle sur différents processus. Les stratégies d'allocation et de routage sont contrôlées directement par l'exécutif du système, et sont de ce fait complètement transparentes au programmeur Ada. Le but de cette recherche était de pouvoir exécuter du code Ada sur une machine parallèle sans avoir à modifier le source du programme, seul le compilateur a été modifié. Le compilateur initial était un compilateur Ada DDC, auquel différentes parties ont été rajoutées afin de résoudre les problèmes de synchronisation, de communication, d'ordonnancement des tâches, d'équilibrage des charges, ...

  On a ainsi la possibilité de pouvoir exécuter le programme Ada soit sur une machine séquentielle soit sur une machine parallèle. Ce système propose en fait de pouvoir "paralléliser" des programmes Ada comprenant des tâches.

  Ceci est différent de l'étude poursuivie dans le cadre de ce stage puisque les programmes Ada écrits à l'aide de notre bibliothèque contiendront explicitement des instructions parallèles et de ce fait n'auront pas la même sémantique qu'une version séquentielle.

La solution proposée par [Lun89] n'a pas été retenue car elle présente l'inconvénient de devoir utiliser deux compilateurs Ada différents selon le type de la machine résultat (séquentielle ou parallèle).

- [LPS90] propose une bibliothèque Ada fournissant les routines nécessaires pour le calcul vectoriel de type SIMD. Le langage Ada a été choisi du fait que c'est un langage portable, et possédant les notions de surcharge, de généricité et de multi-tâches. Les caractéristiques principales de cette bibliothèque sont :
  - Exécution des opérations sur certains éléments du vecteur plutôt que sur la totalité du vecteur,
  - Instructions conditionnelles (identique à la notion de "masque" en Fortran [Caq94]) et instructions de bloc (extension du parallélisme),
  - Affectations de vecteurs par des scalaires,
  - Alignement de données : fonctions permettant de décaler les données à l'intérieur d'un vecteur (`shift` ou `rotate`).

  L'implémentation de ces fonctionnalités a été réalisé à l'aide de tâches Ada, qui seront exécutées en parallèle sur des configurations multi-processeurs.

3.2 L'interface ADA-PVM

Un package Ada a donc été réalisé, celui-ci associe à chaque primitive PVM 3 une procédure ou fonction de même nom. Du fait qu'Ada contrairement à C est un langage fortement typé, et de surcroît possédant la notion de généricité, certaines modifications ont été apportées aux primitives PVM 3.

Ainsi, pour les fonctions de communications, un package interne, générique sur le type de la donnée à échanger a été défini. Et donc contrairement à PVM 3 où l'on dispose d'autant de fonctions ayant des noms différents qu'il y a de types de données élémentaire (`char`, `float`, `double`, `int`, ...), l'utilisateur du package Ada n'aura qu'à instancier le package générique sur le type de son choix.

Le fait qu'Ada soit un langage fortement typé a notamment permis d'être plus restrictif au niveau des valeurs attendues pour les paramètres d'une fonction. Prenons l'exemple de la fonction PVM 3 `pvm_advise` qui permet d'utiliser des liens directs pour la communication entre tâches[1]. Cette fonction a un unique paramètre de type entier, qui ne devrait normalement pouvoir prendre que les 3 valeurs `PvmDontRoute`, `PvmAllowDirect` ou `PvmRouteDirect` selon le type de lien que l'on désire utiliser (voir [GBD+93]). Mais le langage C ne dispose d'aucun moyen d'empêcher l'appel de la fonction avec une valeur illicite, seul le code retourné par la fonction permettra de s'assurer que l'appel était correct. En Ada, pour éviter ce genre de problème, on a utilisé un type énuméré, et on sera donc sûrs que la procédure sera appelée avec une valeur correcte [Bar91].

Le mécanisme de gestion des erreurs a également été modifié. En C, chaque primitive renvoie un code qui doit être testé pour savoir si l'exécution de la primitive s'est bien déroulée. Dans le cas contraire, la valeur du code retourné permet d'identifier le problème. Ceci a été remplacé en Ada par le mécanisme des exceptions : une exception

[1] voir section 2.4

est un moyen offert par le langage Ada pour traiter les erreurs et autres situations exceptionnelles qui surviennent lors de l'exécution d'un programme [Als87]. Une exception a donc été définie pour chaque problème pouvant survenir lors de l'exécution d'une primitive PVM 3. L'utilisateur devra alors décider soit de traiter les cas d'erreurs à l'aide d'un traite-exception soit de laisser les exceptions se propager.

Ce mécanisme de traitement des erreurs permet d'obtenir un programme plus fiable puisque l'on est sûr que l'exécution normale du programme s'arrêtera en cas de problème, et que l'on obtiendra des messages clairs identifiant le problème.

3.2.1 Redéfinition de l'appel à certaines fonctions PVM 3

Afin de faciliter l'utilisation des primitives PVM 3, il nous a également paru nécessaire de modifier l'interface de certaines des primitives PVM 3. Par exemple, pour envoyer une donnée d'une tâche vers une autre, il faut appeler deux procédures de la manière suivante :

```
pvm_pack(&donnee, 1, 1); /* Stockage dans le buffer. */
pvm_send(dest, msgtag); /* Envoi du message. */
```

où `donnee` est la variable que l'on désire envoyer, `dest` l'identificateur de la tâche destinataire, et `msgtag` la marque du message envoyé. Ce que l'on propose de faire ici est de cacher la notion de buffer à l'utilisateur et de lui permettre d'envoyer des données en appelant juste une fonction dont la syntaxe serait la suivante :

```
send(donnee, dest, msgtag)
```

De même pour la réception de donnée, au lieu de devoir appeler les deux fonctions :

```
pvm_recv(source, msgtag); /* Réception du message. */
pvm_upkint(&donnee, 1, 1); /* Extrait la donnée du buffer. */
```

on fournit la fonction suivante :

```
recv(donnee, source, msgtag)
```

Toutefois, afin de permettre l'envoi de plusieurs données de type différents dans un même message, les deux fonctions `send` et `recv` auront un paramètre supplémentaire, qui aura une valeur par défaut et sera donc optionnel [Als87]. Ce paramètre va permettre de distinguer deux cas pour chacune des deux fonctions.

- **Fonction `send`** : son rôle dans les deux cas sera de stocker la donnée dans le buffer d'envoi, et le paramètre booléen `envoi` permettra de préciser si l'on veut que l'envoi soit effectivement fait (valeur par défaut) ou non ce qui signifiera que le message construit sera envoyé ultérieurement. On a donc :

  ```
  send(donnee, dest, msgtag, envoi)
  ```

- **Fonction `recv`** : le paramètre booléen `nouveau` servira à préciser si un nouveau message doit être attendu (valeur par défaut), ou alors si on utilise un message préalablement reçu. Puis dans les deux cas, la donnée sera extraite du message courant. La syntaxe d'appel est donc :

  ```
  recv(donnee, source, msgtag, nouveau)
  ```

3.2.2 Résultats

Ce package a été testé avec succès sur les programmes fournis en exemple dans [GBD+93] : une application maître-esclave avec des communications entre les différents esclaves, et une application écrite dans le mode SPMD (*Single Program Multiple Data*). Pour l'application maître-esclave, on a pu également vérifier que l'on pouvait, pour une exécution donnée, associer aussi bien le programme Maître écrit en Ada avec le programme Esclave écrit en Ada, que le programme Maître écrit en C avec le programme Esclave écrit en Ada. Les résultats obtenus lors des différentes exécutions sont rigoureusement identiques. Il est ainsi possible d'intégrer dans une même application des exécutables, faisant appel à PVM 3, issus de langages différents.

3.3 Le standard MPI

Beaucoup de machines parallèles, et spécialement celles à mémoire distribuée utilisent un modèle de communications basé sur l'échange de messages. Et bien que les concepts primaires de communication entre processus à l'aide de messages soient bien définis, il n'existe aucune norme et chaque machine possède sa propre interface.

Au début des années 90, plusieurs personnes se sont donc réunies pour définir le standard MPI (**M**essage **P**assing **I**nterface) [Mes94]. Ce standard qui est toujours en cours d'élaboration a pour but essentiel d'obtenir un système d'échange de messages qui soit à la fois :

- efficace,
- portable sur plusieurs types de machine différentes,
- facilement utilisable.

L'étude de MPI a été intéressante car ce standard regroupe les caractéristiques les plus importantes de la plupart des systèmes d'échange de message existants (Express, PARMACS, PVM, ...). Les primitives fournies peuvent être regroupées de la manière suivante :

- les communications point à point,
- les communications collectives (synchronisation, diffusion, distribution, ...),
- la gestion de groupes de processus,
- la gestion des topologies de processus,
- la gestion de l'environnement MPI.

Un programme MPI est constituée de processus autonomes exécutant chacun leur propre code sur une machine de type MIMD, ou sur un réseau. Chaque processeur possède donc son propre séquenceur et exécute son programme sur des données locales (machine à mémoire distribuée) ou sur des données globales (machine à mémoire partagée) [BS94b].

Les communications entre ces processus se font à l'aide des primitives MPI. Le modèle d'exécution de chaque processus n'est pas précisé par la norme : un processus pourra être soit séquentiel, soit parallèle. On ne dispose pas de mécanismes pour spécifier l'allocation initiale des processus lors d'une exécution, ni leur lien avec les processeurs physiques. Ces mécanismes devront être fournis par les différents systèmes de développement, à l'exemple d'un fichier `parfile` qui est utilisé en complément de la bibliothèque **CS_TOOLS** sur la machine Meiko afin de pouvoir placer les processus et de choisir une topologie [Com90].

3.3.1 Inconvénients de MPI

Pour le moment, aucune implémentation complète de MPI n'est encore disponible. Seules les syntaxes des types, constantes, et prototypes de primitives sont fournies à la fois pour le langage C et le langage Fortran.

Initialement on avait pensé fournir une implémentation de la norme MPI en ADA, implémentation qui serait basée sur les primitives disponibles en PVM 3. Cette idée a été rejetée pour trois raisons principales :

- La syntaxe utilisée en MPI est beaucoup trop proche de celle du langage C,
- Certaines notions de la norme ne sont pas absolument pas conformes à la philosophie Ada (notamment les `datatypes` qui permettent de spécifier le type d'une donnée, et qui doivent être passés en paramètre de certaines fonctions en plus de la donnée),
- Les fonctionnalités proposées par MPI sont beaucoup trop nombreuses et elles ne constituent pas une synthèse des mécanismes propres au calcul parallèle, mais plutôt un catalogue.

Il nous a donc paru plus intéressant d'extraire de la norme MPI les notions qui nous ont semblé utiles pour le domaine d'application visé, à savoir le calcul parallèle orientée données (dans un style SPMD).

Chapitre 4

Evaluation des temps de communication

4.1 Généralités

Dans les machines parallèles où la communication se fait selon un modèle temps linéaire, la transmission d'un message de longueur L entre deux processeurs voisins est modélisé par la somme :

- d'un temps d'initialisation (coût fixe) β,
- et d'un temps de propagation $L\tau$ (coût variable) directement proportionnel à la taille L du message communiqué.

$$T = \beta + L\tau$$

Du fait que dans notre cas, la communication se fait sur un réseau et donc sur deux machines différentes, le temps d'initialisation est en fait découpé en :

- un temps d'initialisation sur le processeur émetteur β_e,
- et un temps de réception du message sur le processeur destinataire β_r.

Et on a donc :

$$T = \beta_e + L\tau + \beta_r$$

Afin de concevoir des algorithmes de communications globales qui soient le plus performant possible, il est nécessaire de connaître les valeurs de ces paramètres. Nous avons effectué pour cela plusieurs séries de test qui nous ont donc permis de mesurer les temps d'une communication.

Le principe du programme de mesure est le suivant : pour un nombre donné de processeurs p et une taille de message donnée L, mesurer le temps nécessaire pour que le processeur numéro 1 envoie un message de longueur L à chacun des $p-1$ autres processeurs, et que le dernier processeur renvoie le message reçu au processeur numéro 1. Afin d'obtenir une mesure qui soit la plus fiable possible, le programme a un troisième paramètre qui est le nombre de fois que la communication doit être répetée lors de la mesure.

L'algorithme du programme est donc :

$processeur\_principal(nb\_processeurs, taille\_message, nb\_itérations)$
- $créer(buffer, taille\_message)$
- démarrer les tâches sur les $nb\_processeurs - 1$ processeurs

- $start = horloge()$
- **pour** i **de** 1 **à** $nb\_itérations$
  faire
  - **pour** i **de** 1 **à** $nb\_processeurs - 1$
    faire
    - $envoyer(buffer, processeur(i))$

    finfaire
  - $recevoir(message, dernier\_processeur)$

  finfaire
- $stop = horloge()$
- $temps = (stop - start)/nb\_itérations$

$autres\_processeurs(nb\_processeurs, taille\_message, nb\_itérations,$
$mon\_numéro)$
- $initialisation()$
- **pour** i **de** 1 **à** $nb\_itérations$
  faire
  - $recevoir(buffer, processeur\_parent)$
  - **si** $mon\_numéro = nb\_processeurs - 1$
    alors
    - $envoyer(buffer, processeur\_parent)$

    finsi

  finfaire

Lors des premiers tests effectués, on s'est aperçu que le temps de communication ne pouvait pas être exprimé sous la forme d'une fonction linéaire dépendant uniquement de la longueur du message. En effet, pour des messages petits (taille entre 0 et 4096 octets), le temps nécessaire à une communication est pratiquement constant, et ce n'est que pour des messages de longueur supérieure que le temps augmente de façon linéaire. Deux cas ont donc été distingués : celui où le coût fixe de la communication est négligeable par rapport au coût variable (message de taille importante), et celui au contraire où c'est le coût fixe qui est supérieur au coût variable.

Cette approche semble d'ailleurs être confirmée par le fait que lors d'une communication PVM 3, la taille maximale d'un paquet échangé sur le réseau est de 4096 octets.

Revenons plus en détail sur ces deux cas :

1. **coût variable ($L\tau$) supérieur au coût fixe (β).** On peut supposer que dans ce cas les phases de communication sont ordonnées selon la figure 4.1, et donc que la durée globale T que l'on peut mesurer sur le processeur maître est :

$$T = pL\tau + 2(\beta_r + \beta_e)$$

Supposons que β_e et β_r dépendent de la longueur du message, et posons :

$$\beta_e = \beta_e^{'} + L\tau_e^{'}$$

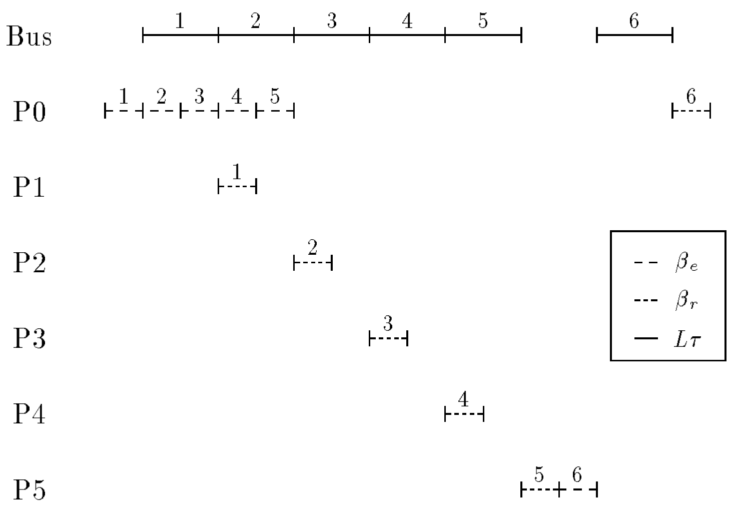

Figure 4.1: Coût variable > Coût fixe

$$\beta_r = \beta_r^{'} + L\tau_r^{'}$$

Il vient alors :

$$T = pL\tau + 2(\beta_e^{'} + L\tau_e^{'} + \beta_r^{'} + L\tau_r^{'})$$
$$T = 2(\beta_r^{'} + \beta_e^{'}) + 2(\tau_r^{'} + \tau_e^{'})L + \tau Lp$$

2. **coût fixe supérieur au coût variable**. D'après la figure 4.2, il vient :

$$T = p\beta_e + 2L\tau + 2\beta_r$$

Soit :

$$T = p(\beta_e^{'} + L\tau_e^{'}) + 2L\tau + 2(\beta_r^{'} + L\tau_r^{'})$$
$$T = 2\beta_r^{'} + 2(\tau + \tau_r^{'})L + \beta_e^{'}p + \tau_e^{'}Lp$$

4.2 Résultats obtenus

Les figures 4.3 et 4.4 présentent les résultats des mesures effectuées. Les résultats sont regroupés par le nombre de processeurs (2, 4 ou 8) ayant participé à la communication. Plusieurs mesures ont été faites sur plusieurs nuits, et on donne à chaque fois le temps minimum obtenu pour une longueur de message donnée et un nombre de processeur donné. A noter que la précision de l'horloge utilisée est de l'ordre de $0,000044s$.

Soit t le temps nécessaire à la communication, p le nombre de processeurs et L la longueur du message, en faisant une régression sur l'ensemble des valeurs obtenues, on a :

- Messages de petite taille (voir figure 4.3)

$$t = 0,00721 + 0,00000212L + 0,00263p + 0,00000135Lp \tag{4.1}$$

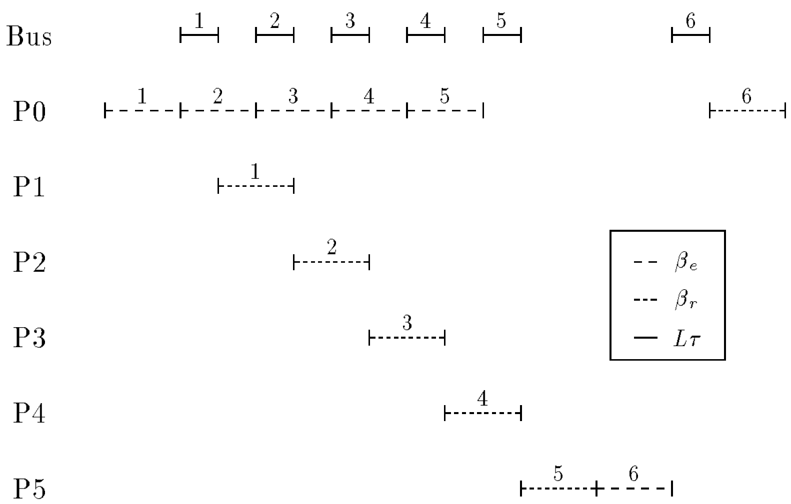

Figure 4.2: Coût fixe > Coût variable

- Messages de grande taille (voir figure 4.4)

$$t = 0,0292 + 0,000001L + 0,00002Lp \tag{4.2}$$

De plus, on a pu voir lors des régressions effectuées que les valeurs obtenues pour les coefficients de p, L et Lp étaient très significatifs sur l'ensemble des mesures proposées.

4.2.1 Analyse des résultats

On s'intéresse ici aux mesures prises pour $p = 2$.

Cas des petits messages

D'après la formule 4.1, on a :

$$\begin{aligned} 2\beta_r^{'} &= 0,00721 \\ 2(\tau + \tau_r^{'}) &= 2,12.10^{-6} \\ \beta_e^{'} &= 0,00263 \\ \tau_e^{'} &= 1,35.10^{-6} \end{aligned}$$

En remplaçant p par 2 dans la formule 4.1, il vient :

$$\begin{aligned} T &= 2(\beta_r^{'} + \beta_e^{'}) + 2(\tau_e^{'} + \tau_r^{'} + \tau)L \\ T &= 0,01247 + 4,82.10^{-6}L \end{aligned}$$

Or en effectuant une régression sur des mesures prises uniquement pour $p = 2$, on avait trouvé la formule suivante :

$$t = 0,0128 + 5,11.10^{-6}L \tag{4.3}$$

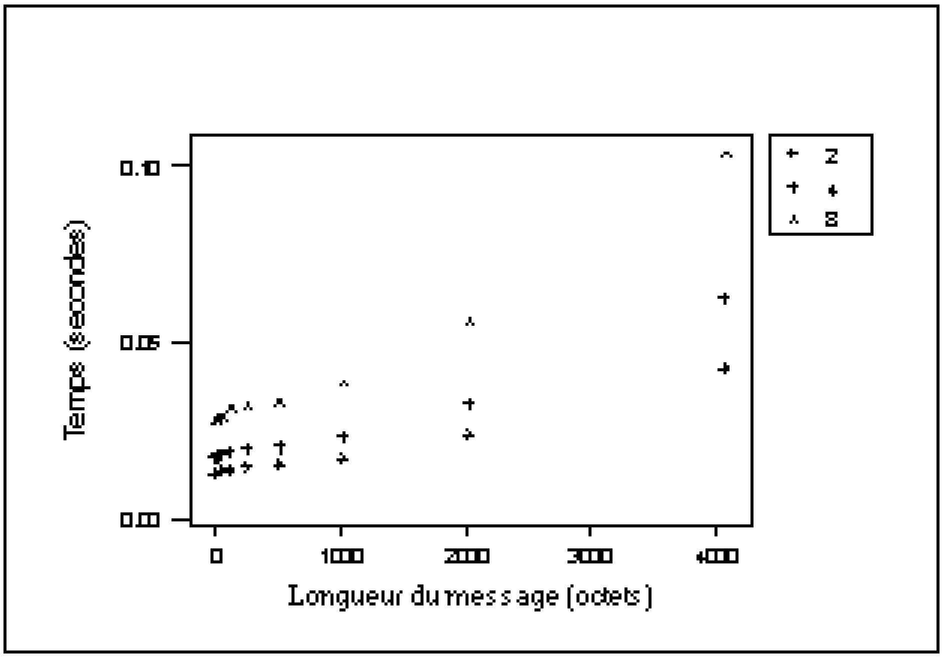

Figure 4.3: Résultats pour des messages de petite taille

Les termes constants dans les deux cas sont très proches, et la différence au niveau des coefficients de L peut s'expliquer par le fait que l'écart type de cette variable lors de la régression est de $5,8.10^{-7}$.

Cas des longs messages

D'après la formule 4.2, on a :

$$\begin{aligned} 2(\beta_e^{'} + \beta_r^{'}) &= 0,0292 \\ 2(\tau_e^{'} + \tau_r^{'}) &= 1,38.10^{-6} \\ \tau &= 1,80.10^{-6} \end{aligned}$$

Ce qui donne pour $p = 2$:

$$\begin{aligned} T &= 2(\beta_r^{'} + \beta_e^{'}) + 2(\tau_e^{'} + \tau_r^{'} + \tau)L \\ T &= 0,0292 + 4,98.10^{-6}L \end{aligned}$$

Or en effectuant une régression sur des mesures prises uniquement pour $p = 2$, on avait trouvé la formule suivante :

$$t = 0,0202 + 5,2.10^{-6}L \tag{4.4}$$

Là encore la différence obtenue au niveau du coefficient de L peut s'expliquer en regardant l'écart type de la régression (qui ici est de 25%).

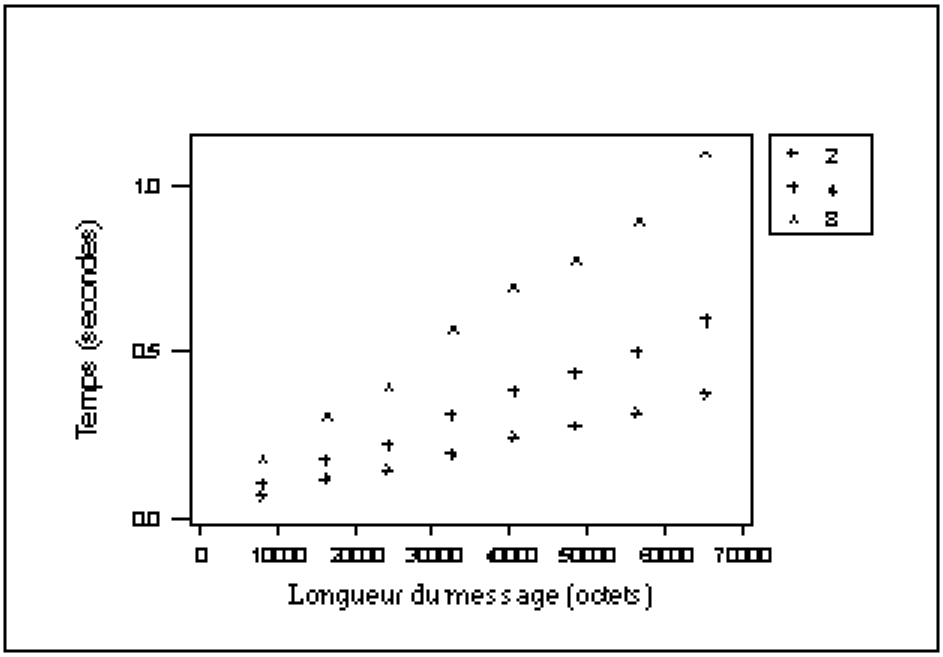

Figure 4.4: Résultats pour des messages de taille importante

En regardant les formules 4.3 et 4.4, on peut voir que pour les deux modèles considérés, les coefficients de L sont très proches. La différence au niveau du terme constant peut s'expliquer par le fait que pour des messages de taille supérieure à 4096 octets, le coût fixe est en fait la somme du coût fixe calculé pour les petits messages et d'un certain temps (noté β'') dû au fait que le message va devoir être découpé en plus d'un paquet.

4.3 Conclusion

- Les figures 4.5, 4.6 et 4.7 montrent les fréquences des temps de mesures obtenues sur 100 mesures pour des messages de longueur donnée. On s'est aperçus en faisant plusieurs fois la même mesure pour des valeurs de paramètres identiques, que les différences de temps observées étaient dûes à des évènements extérieurs de temps fixe. On a notamment pu observer l'apparition d'un évènement d'une durée d'une seconde. De plus, ces évènements apparaissent de manière aléatoire au cours d'une expérience. C'est donc pour cela qu'il est important lors des calculs de la régression de ne considérer que les valeurs minimums obtenues.

- Le fait d'avoir considéré que les temps d'initialisation de la communication dépendaient de la longueur du message envoyé a été confirmé par le fait que :

  - pour les petits messages, le coefficient β_e' de Lp est significativement différent de 0,

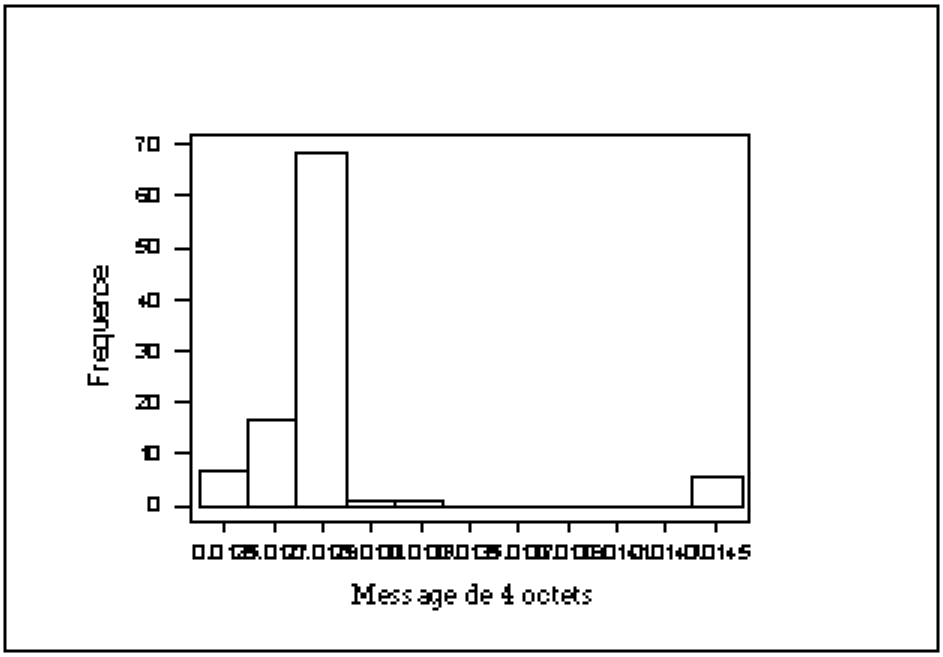

Figure 4.5: Fréquence de temps pour messages de 4 octets

- pour les longs messages, le coefficient de L est également significativement différent de 0.

On a également pu vérifier que dans le cas des longs messages le temps ne dépendait pas du nombre de processeurs : en ajoutant p lors du calcul de la régression, on s'aperçoit que le coefficient de ce dernier n'est pas significativement différent de 0.

- L'identification des paramètres donne :

$$\begin{aligned} \beta_e^{'} &\approx 2,63.10^{-6} \\ \tau_e^{'} &\approx 1,35.10^{-6} \\ \beta_r^{'} &\approx 3,60.10^{-6} \\ \tau_r^{'} &\approx 1,50.10^{-6} \\ \tau &\approx 1,80.10^{-6} \\ \beta^{''} &\approx 7,40.10^{-6} \end{aligned}$$

- Afin d'avoir des résultats plus fiables, il va être nécessaire de :
  - multiplier les répétitions d'une même mesure afin d'obtenir des temps minimums qui soient plus fiables,
  - multiplier les valeurs de p et de L afin de mieux mettre en évidence les seuils de taille des buffers utilisés lors d'une communication.

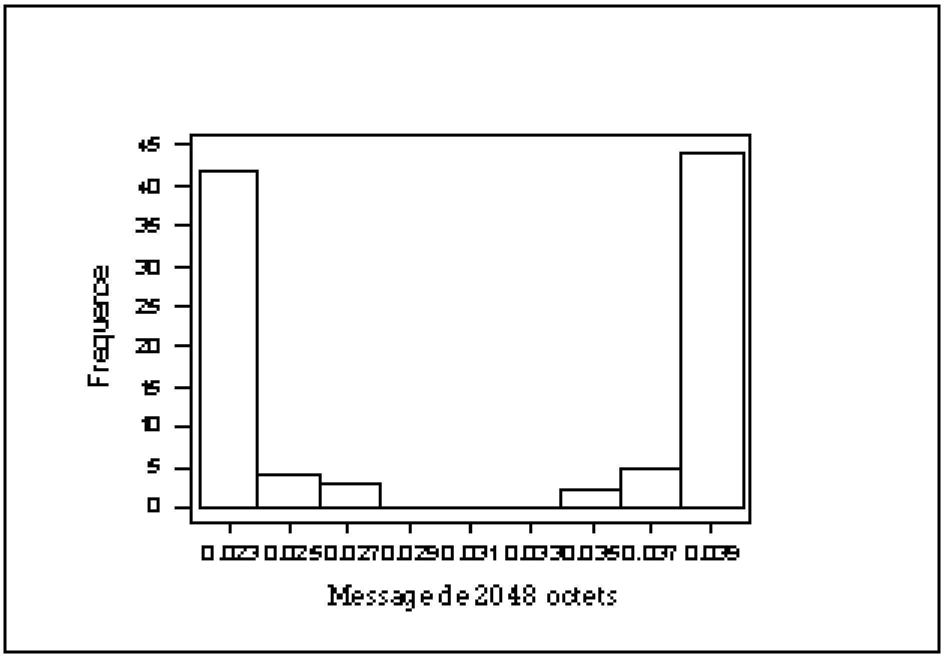

Figure 4.6: Fréquence de temps pour messages de 2048 octets

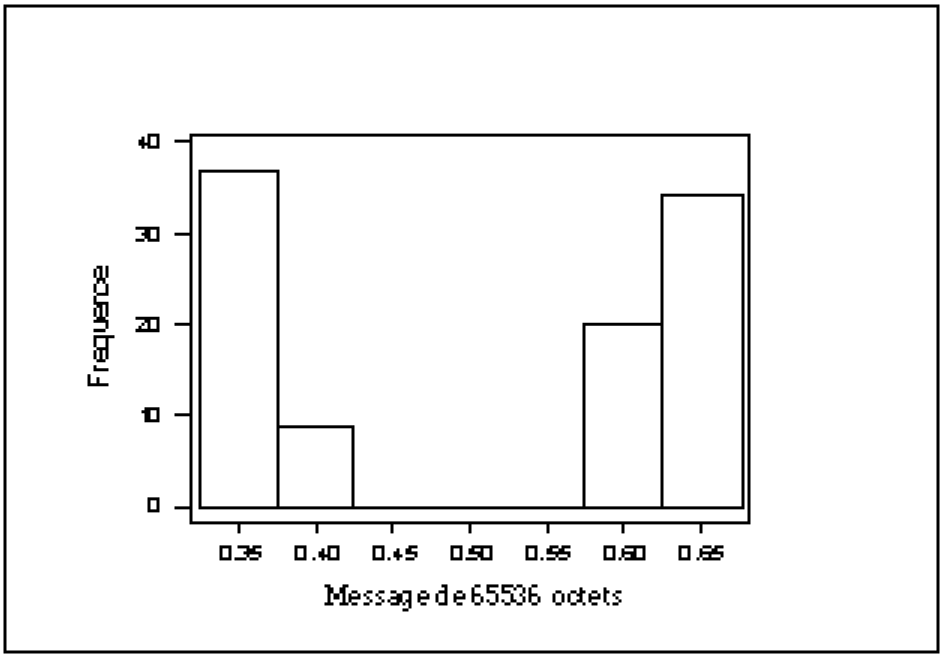

Figure 4.7: Fréquence de temps pour messages de 65536 octets

Chapitre 5

Les communications globales

5.1 Définition

Soit XR et YR deux objets répartis sur p processeurs. On note $XR_k(i)$, le $i^{ème}$ élément de la composante de l'objet XR, qui se trouve localisée sur le processeur P_k.

Pour chacune des fonctionnalités suivantes, le groupe pourra désigner soit un ensemble limité de tâches[1], ou alors l'ensemble des tâches d'une application. Les différentes opérations de communications globales peuvent être définies de la manière suivante :

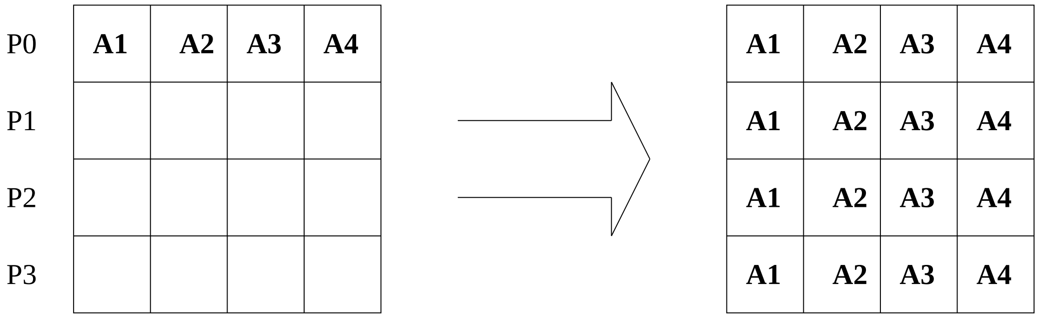

Figure 5.1: La diffusion

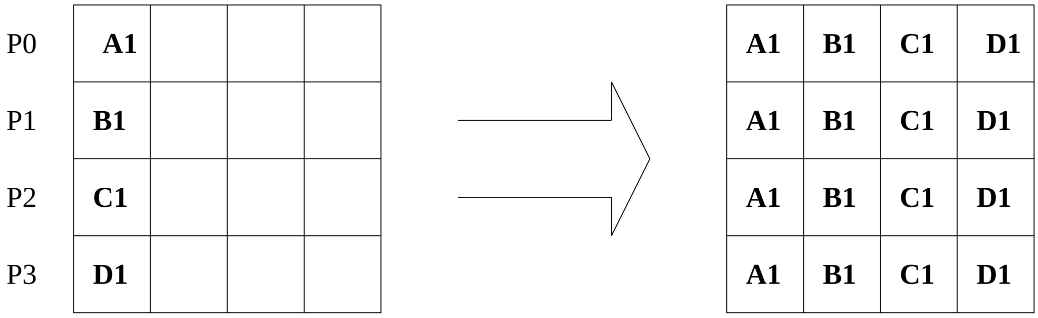

Figure 5.2: La multi-diffusion

- Synchronisation entre processus d'un même groupe.

[1]de la même manière qu'en PVM 3

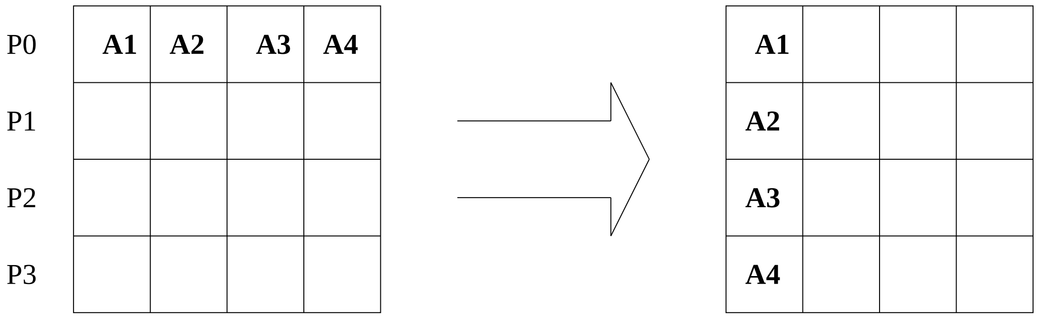

Figure 5.3: La distribution

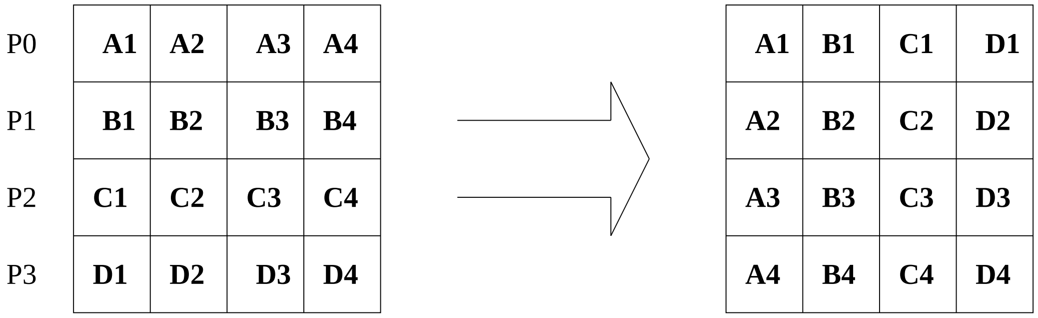

Figure 5.4: La multi-distribution

- Diffusion : Envoi d'un même message à toutes les tâches d'un groupe, à partir d'un processeur initiateur P_s, qui n'est pas forcément membre du groupe.

$$YR_k \leftarrow diffuser(XR, s) \equiv$$

$$YR(k) \leftarrow XR_s \ \forall k \in \{0, \ldots, p-1\}$$

- Multi-diffusion (échange total) : Diffusion à partir de toutes les tâches d'un groupe, de façon simultanée, vers tous les membres du groupe.

$$YR \leftarrow multi\_diffuser(XR) \equiv$$

$$YR_k(i) \leftarrow XR_i \ \forall i, k \in \{0, \ldots, p-1\}$$

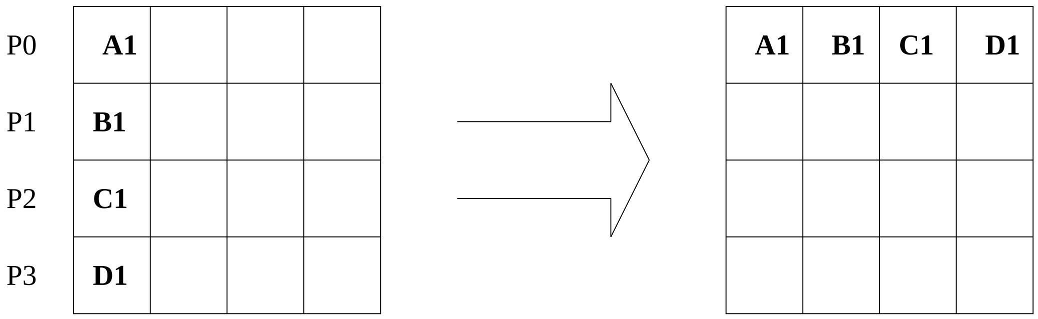

Figure 5.5: Le rassemblement

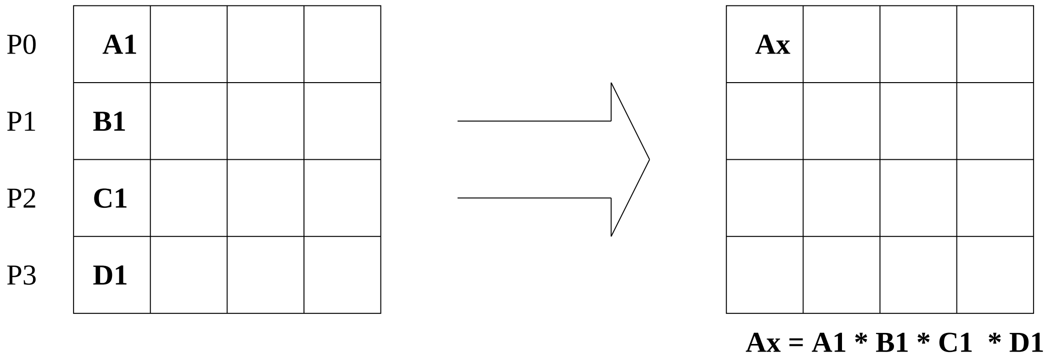

Figure 5.6: La combinaison centralisée

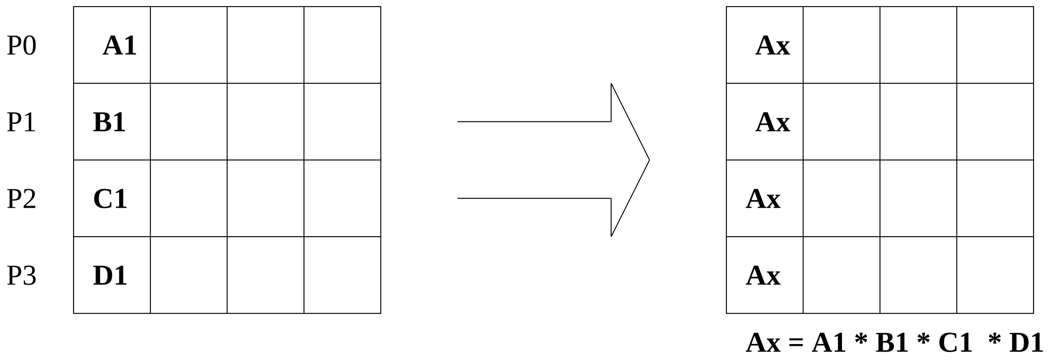

Figure 5.7: La combinaison diffusée

- Distribution : Envoi à partir d'un processeur initiateur P_s d'un message différent à chacune des tâches d'un groupe.

$$YR \leftarrow distribuer(XR, s) \equiv$$

$$YR_k \leftarrow XR_s(k) \; \forall k \in \{0, \ldots, p-1\}$$

- Multi-distribution : Distribution simultanée à partir de toutes les tâches c'est-à-dire que chaque tâche envoie un message différent à chacune des autres tâches du groupe.

$$YR \leftarrow multi\_distribuer(XR) \equiv$$

$$YR_k(i) \leftarrow XR_i(k) \; \forall i, k \in \{0, \ldots, p-1\}$$

- Rassemblement : Opération inverse de la distribution, qui consiste à récupérer sur une tâche unique des données venant de toutes les autres tâches.

$$YR \leftarrow rassembler(XR, d) \equiv$$

$$\begin{cases} YR_d(i) \leftarrow XR_i & \forall i \in \{0, \ldots, p-1\} \\ YR_j \leftarrow \emptyset & j \neq d \end{cases}$$

- Combinaison (appelée souvent réduction) : Opération qui permet de combiner les données venant de toutes les tâches d'un groupe à l'aide d'un opérateur donné (min, max, somme, ...) noté $\otimes$. On distingue 4 types de réduction :

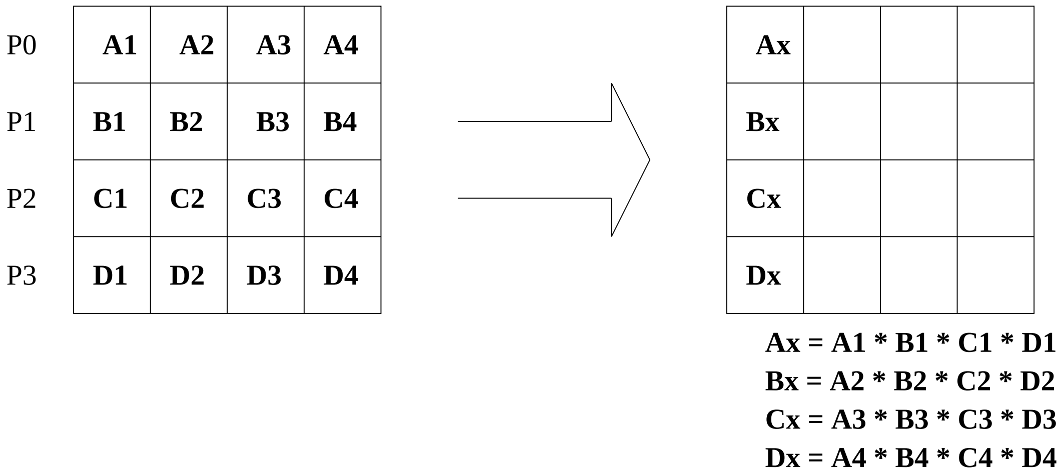

Figure 5.8: La combinaison multiple

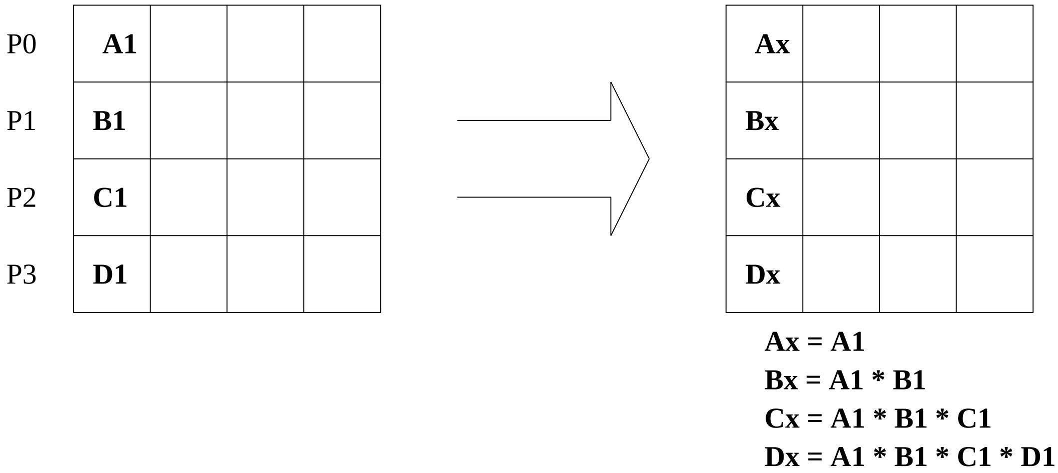

Figure 5.9: La combinaison partielle

– centralisée : le résultat de la combinaison est envoyée vers une tâche unique,

$$YR \leftarrow combiner(XR, d) \equiv$$

$$YR_d \leftarrow XR_0 \otimes \ldots \otimes XR_{p-1}$$

– diffusée : le résultat est envoyé vers toutes les tâches du groupe,

$$YR \leftarrow D\_combiner(XR) \equiv$$

$$YR_k \leftarrow XR_0 \otimes \ldots \otimes XR_{p-1} \; \forall k \in \{0, \ldots, p-1\}$$

– distribuée : le résultat est distribué vers toutes les tâches (utilisée dans le cas où le résultat est sous la forme d'un vecteur),

– multiple : combinaisons simultanées à partir de toutes les tâches, et à destination de chacune d'elles

$$YR \leftarrow M\_combiner(XR, d) \equiv$$

$$YR_k \leftarrow XR_0(k) \otimes \ldots \otimes XR_{p-1}(k) \; \forall k \in \{0, \ldots, p-1\}$$

- Combinaison partielle : Supposons qu'il y ait un ordre sur les tâches d'un groupe. Chacune va recevoir le résultat de la combinaison des données des tâches qui sont plus petites qu'elles selon l'ordre.

$$YR \leftarrow P\_combiner(XR, d) \equiv$$

$$YR_k \leftarrow XR_0 \otimes \ldots \otimes XR_k \; \forall k \in \{0, \ldots, p-1\}$$

Toutes ces opérations de communication peuvent en fait être exprimées en fonction de la diffusion et/ou de la distribution et/ou de la combinaison centralisée, puisque :

- le *rassemblement* peut être considéré comme une combinaison centralisée où l'opérateur de combinaison est une fonction de concaténation,
- la solution pour une *combinaison diffusée* peut être de d'abord effectuer une combinaison centralisée sur l'un des processeurs du groupe puis de diffuser le résultat obtenu vers les autres processeurs concernés par la communication.
- *la combinaison partielle* peut être vu comme une multi-diffusion suivie d'un calcul de la réduction sur chacun des processeurs sur des composantes selectionnées.

Toutefois, il faut préciser que ces solutions ne représentent pas une implémentation optimale de ces fonctions de communication.

5.2 Comparaison des différentes solutions

Nous allons maintenant essayer de voir quelle serait la solution la plus adaptée pour l'implémentation des fonctions de communication principales. Du fait que ces communications vont devoir s'effectuer sur un réseau de stations de travail reliées par Ethernet (qui fonctionne comme un bus), il va falloir essayer de minimiser le plus possible le nombre de messages à envoyer entre les différents processeurs participant à la communication. On va donc aussi s'intéresser aux solutions proposées dans les systèmes distribués.

5.2.1 La distribution

Du fait même de sa définition, l'algorithme le mieux adapté semble être celui utilisé dans les machines parallèles ayant une architecture en forme d'étoile : le processeur initiateur va envoyer successivement un message différent à chacun des processeurs du groupe. Pour un groupe de p processeurs, le nombre de messages échangés est donc p (ou $p-1$ si l'initiateur appartient au groupe).

5.2.2 La diffusion

On peut ici comparer les algorithmes utilisés par PVM 3, et ceux utilisés avec des architectures en forme d'arbre et d'étoile.

Pour PVM 3 (voir annexe B), lorsque plusieurs tâches participant à la communication se trouvent être sur le même serveur, un seul message sera envoyé à ce serveur, mais le nombre de communications nécessaires à la diffusion est tout de même beaucoup trop important. Toutefois, les solutions proposées pour des architectures d'arbre ou d'étoile ne semblent pas plus intéressantes dans la mesure où elles ne diminuent pas le nombre de messages à envoyer. La solution de l'arbre aurait pu être intéressante dans le cas où il aurait été possible d'envoyer deux messages en parallèle (sur des liens de communications différents, les deux fils d'un nœud peuvent envoyer simultanément), mais dans notre cas, on utilise un réseau et donc il n'y a qu'un message qui peut passer sur le bus à un moment donné.

5.2.3 La combinaison centralisée

Les solutions les plus adaptées au problème de la combinaison centralisée ont été trouvées dans la littérature des systèmes distribués. Il existe en effet dans ces systèmes une notion de leader (processeur qui régit l'ensemble des autres processeurs), et en cas de panne de ce processeur leader, il est très important de réelire le plus vite possible un nouveau leader. La technique utilisée dans la plupart des cas est d'ordonner les processeurs selon un critère donné, puis de choisir comme leader celui ayant la valeur la plus petite ou la plus grande selon cet ordre. Il va donc falloir qu'une information soit échangée sur l'ensemble des processeurs.

Ce problème peut être ramené à celui de la réduction centralisée si on considère que lorsqu'un processeur reçoit une donnée, au lieu de la comparer à sa valeur d'ordre, il effectue la combinaison entre sa propre valeur et la valeur reçue, et il envoie le résultat aux processeurs suivants. Un tel algorithme peut être trouvé dans [KKS90].

Chapitre 6

Un exemple d'application

6.1 Présentation de l'application d'astronomie

6.1.1 Enoncé du problème

L'observation d'étoiles à partir de la Terre est très perturbée par la couche atmosphérique sans cesse en mouvement, bien plus en réalité que par la diffraction liée au diamètre du télescope. Ce phénomène se manifeste ainsi : lors de l'observation d'une étoile simple, au lieu de voir un seul point lumineux, nous observons des "tavelures" c'est à dire une sorte d'éclatement de la lumière dans toutes les directions, dû au mouvement aléatoire de l'atmosphère. Le but des astronomes est alors de créer des méthodes pour reconstituer l'étoile simple à partir de ces images de "tavelures".

Diverses méthodes sont proposées pour diminuer, voire annuler ce phénomène dû à la perturbation atmosphérique : télescope spatial, miroir adaptatif. Une autre solution est d'utiliser les propriétés des *statistiques d'ordre supérieur*. En effet, l'utilisation du spectre d'ordre trois permet de bien atténuer la dégradation de l'image dûe à l'atmosphère.

Ce travail a été réalisé en collaboration avec deux chercheurs Joel Le Roux et Delphine Rossille, du laboratoire I3S de l'université de Nice Sophia-Antipolis, qui travaillent sur l'utilisation des statistiques d'ordre supérieur en vue de la reconstitution d'images, en particulier dans le cas d'images d'astrophysique.

6.1.2 Résolution grâce aux statistiques d'ordre supérieur

Nous disposons d'un ensemble d'images bruitées par l'atmosphère. Un simple moyennage de ces images temporelles ne donne aucun résultat car il brouille encore plus l'image. La méthode présentée ici consiste à générer le spectre d'ordre trois (ou bispectre)[1] de chaque image bruitée, puis de les moyenner, ce qui permet d'annuler l'effet de la couche atmosphérique. Nous obtenons donc un bispectre moyenné qui n'est plus perturbé par le bruit. Il faut alors reconstituer l'image originale $o(\mathbf{x})$ à partir de ce bispectre moyenné. Nous utiliserons pour ce faire une méthode classique en astronomie : la méthode de Lohmann, Weigelt et Wirnitzer.

[1] Le bispectre est la transformée de Fourier de la corrélation d'ordre trois du signal

Moyenne des bispectres

Soit $o(\mathbf{x})$, l'objet astronomique que nous voulons observer. C'est une image bidimensionnelle déterministe de dimension N x N et de transformée de Fourier $O(\mathbf{u})$.

L'atmosphère dégrade de manière aléatoire cette image initiale $o(\mathbf{x})$. Elle correspond en fait à l'ajout d'une phase aléatoire sur $O(\mathbf{u})$.

Appelons $f_k(\mathbf{x})$, l'image dégradée que l'on observe de la Terre à travers un télescope à un instant k et $F_k(\mathbf{u})$ sa transformée de Fourier. Son bispectre est défini par :

$$B_k(\mathbf{u}, \mathbf{v}) \quad = \quad F_k(\mathbf{u})\ F_k(\mathbf{v})\ F_k^*(\mathbf{u}+\mathbf{v})$$

où z^* désigne le conjugué du nombre complexe z.

Le calcul doit être réalisé sur chaque image $f_k(\mathbf{x})$ pour k variant de 1 à K, avec K le nombre total d'images bruitées que nous avons à notre disposition. Nous calculons ensuite la moyenne $\widehat{B(\mathbf{u}, \mathbf{v})}$ (en partie réelle et partie imaginaire) de tous ces bispectres. Cette moyenne permet d'annuler la phase aléatoire dûe à l'atmosphère.

$$\widehat{B(\mathbf{u}, \mathbf{v})} \quad = \quad \frac{1}{K} \sum_{k=1}^{K} B_k(\mathbf{u}, \mathbf{v})$$

Il s'agit donc de calculer pour chaque image sa transformée de Fourier puis son bispectre associé, d'où l'idée de paralléliser cette portion de programme afin de diminuer le temps de calcul et de bénéficier des temps de non-utilisation d'un grand nombre de serveurs d'un réseau local.

Moyenne des carrés des modules

En même temps que le calcul des bispectres, il faut calculer le module au carré de chaque image. En effet, le module utilisé pour la reconstitution de l'image est pris comme étant la racine carrée de la moyenne des carrés des modules de chaque image car l'atmosphère n'affecte que la phase de l'image bruitée. Ainsi,

$$|\widehat{O(u)}| \quad = \quad \sqrt{\frac{1}{K} \sum_{k=1}^{K} |F_k(u)|^2}$$

Ce calcul est également à réaliser pour chaque image bruitée, d'où l'interêt de sa parallélisation.

Méthode de reconstitution

La méthode de reconstitution est réalisée sur la phase du bispectre moyenné, le module étant en fait la racine carrée de la moyenne des carrés des modules de chaque image. Sachant que :

$$\begin{aligned} O(\mathbf{u}) &= |O(\mathbf{u})|\ e^{-j\phi(\mathbf{u})} \\ B(\mathbf{u}, \mathbf{v}) &= |B(\mathbf{u}, \mathbf{v})|\ e^{-j\psi(\mathbf{u}, \mathbf{v})} \end{aligned}$$

et en utilisant le caractère impair de la phase $\phi(u)$ associée à un signal réel, on peut vérifier que :

$$\psi(\mathbf{u},\mathbf{v}) \quad = \quad \phi(\mathbf{u}) + \phi(\mathbf{v}) - \phi(\mathbf{u}+\mathbf{v})$$

et donc :

$$\phi(\mathbf{u}+\mathbf{v}) \quad = \quad \phi(\mathbf{u}) + \phi(\mathbf{v}) - \psi(\mathbf{u},\mathbf{v})$$

La méthode de Lohmann, Weigelt et Wirnitzer est récursive. Elle correspond au cas particulier où v est une constante v_0. Cette valeur constante devient le pas de récursion permettant de déduire toutes les autres valeurs de la phase du signal original. L'équation de récursion permet d'évaluer l'estimation $\widehat{\phi}$ de la phase ϕ à partir de la phase estimée $\widehat{\psi}$ du bispectre B.

$$\begin{aligned} \widehat{\phi}(u+v_0) &= \widehat{\phi}(u) + \widehat{\phi}(v_0) - \psi(u,v_0) \\ \widehat{\phi}(u+kv_0) &= \widehat{\phi}(u+(k-1)v_0) + \widehat{\phi}(v_0) - \psi(u+(k-1)v_0, v_0) \end{aligned}$$

La récurrence permet donc de calculer $\phi(\widehat{u+mv_0})$ à partir de $\widehat{\phi(u)}$ et $\widehat{\psi}(u+kv_0,v_0)$ pour k de 1 à $m-1$.

Cas monodimensionnel

Prenons $v = 1$. L'équation de récursion devient alors :

$$\phi(u+1) \quad = \quad \phi(u) + \phi(1) - \psi(u,1)$$

Il suffit ainsi de calculer les valeurs suivantes de chaque bispectre :

$$B(u,v) \text{ pour } u = 0,\ v = 0, \text{ et pour } u \in [1, N/2],\ v = 1$$

et leur moyenne $\widehat{B}$ pour ces mêmes valeurs de u et v, ainsi que le module carré :

$$|\widehat{O(u)}|^2 = \frac{1}{N}\sum_{k=1}^{K} |F_k(u)|^2 \text{pour } u \in [-N/2+1, N/2]$$

La reconstruction se fait alors en calculant la transformée de Fourier de l'image originale par :

$$\widehat{O(u)} \quad = \quad |\widehat{O(u)}|\ e^{-j\,\widehat{\phi(u)}}$$

Sa transformée de Fourier inverse nous donnera alors l'image reconstituée $\widehat{o}(x)$.

Cas bidimensionnel

L'image bruitée ainsi que sa transformée de Fourier sont alors :

$$f_k \begin{pmatrix} x \\ y \end{pmatrix} \quad \leftrightarrow \quad F_k \begin{pmatrix} u_x \\ u_y \end{pmatrix}$$

Son bispectre est donc à quatre dimensions :

$$B_k \begin{pmatrix} u_x & v_x \\ u_y & v_y \end{pmatrix} \quad = \quad F_k \begin{pmatrix} u_x \\ u_y \end{pmatrix} F_k \begin{pmatrix} v_x \\ v_y \end{pmatrix} F_k^* \begin{pmatrix} u_x + v_x \\ u_y + v_y \end{pmatrix}$$

Le même principe de résolution est utilisé : moyenne des bispectres et moyennes des carrés des modules.

La méthode de reconstitution de la phase est donnée par :
Pour u_x de 0 à $N/2$

$$\psi\begin{pmatrix} u_x \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \phi\begin{pmatrix} u_x \\ 0 \end{pmatrix} + \phi\begin{pmatrix} 1 \\ 0 \end{pmatrix} - \phi\begin{pmatrix} u_x + 1 \\ 0 \end{pmatrix}$$

Pour u_y de 1 à $N/2$
Pour u_x de $-N/2+1$ à $N/2$

$$\psi\begin{pmatrix} u_x \\ u_y \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \phi\begin{pmatrix} u_x \\ u_y \end{pmatrix} + \phi\begin{pmatrix} 0 \\ 1 \end{pmatrix} - \phi\begin{pmatrix} u_x \\ u_y + 1 \end{pmatrix}$$

Il suffit donc de calculer les valeurs suivantes de chaque bispectre :

$$B_k\begin{pmatrix} u_x \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ pour } u_x \in [1, N/2]$$

$$B_k\begin{pmatrix} u_x \\ u_y \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \text{ pour } u_y \in [1, N/2] \text{ et } u_x \in [-N/2+1, N/2]$$

De la même manière, la moyenne des carrés des modules doit être calculée pour :

$$|F_k\begin{pmatrix} u_x \\ u_y \end{pmatrix}|^2 \text{ pour } u_x \in [-N/2+1, N/2] \text{ et } u_y \in [0, N/2]$$

6.1.3 Réalisation pratique et résultats

La programmation s'est faite dans un premier temps en monodimensionnel. Les signaux utilisés sont des projections d'images d'une étoile double appelée α Coma. Nous avions à notre disposition environ 560 signaux bruités de dimension 256 pixels.

Nous avons donc parallélisé le calcul de la moyenne des bispectres et de la moyenne des carrés des modules, en répartissant les calculs sur un ensemble de serveurs.

Le résultat obtenu aprés reconstitution ne montre pas de façon nette les deux pics correspondant à nos deux étoiles. Ceci est en fait dû à la qualité de nos images. En effet, ces images sont très faibles en amplitude car les étoiles sont trés peu lumineuses. Il nous faudrait en réalité beaucoup plus d'images pour que le programme converge vers la solution.

Nous ne pouvons vous montrer les résultats de la reconstitution dans le cas d'images bidimensionnelles. En effet, la reconstitution en bidimensionnel n'a pu être mise au point avant la fin du stage. Toutefois, le bispectre ainsi que la moyenne des carrés des modules ont été calculés dans le cas d'images 2D. La parallélisation a donc été programmée dans le cas 2D.

6.2 Package générique de traitement

La solution proposée pour implémenter l'application d'astronomie est l'utilisation d'une librairie de calcul réparti. Cette librairie est en fait constituée de deux parties : un programme "maître" et un programme "esclave".

Afin d'être réutilisable pour d'autres applications fonctionnant selon le même principe, ce package sera générique. Les principaux paramètres de généricité sont les suivants :

- Type permettant d'identifier les éléments à traiter (ex : numéro d'image),
- Type résultat du traitement effectué sur ces éléments (ex : résultat du calcul du bispectre et du module carré sur une image). La fonction de traitement permettant de passer d'un élément à un résultat est donc aussi générique.
- Fonction permettant de combiner (réduction) deux résultats (ex : sommation des bispectres et des modules carrés de deux images).

Le principe utilisé est le suivant : on a tout d'abord une partie initialisation (qui n'est pas générique) où le "maître" va récupérer les différents paramètres de l'application, créer les tâches esclaves, et leur envoyer les paramètres leur étant nécessaires. On a alors à cet instant de l'application, un processus maître et plusieurs processus esclaves tous correctement initialisés. L'ensemble des processus (maître et esclaves) vont alors pouvoir instancier les fonctions génériques qui vont leur permettre d'effectuer leur traitement respectif.

La répartition du travail sur l'ensemble des esclaves (ou serveurs) se fait de la manière suivante : le processus maître rentre dans une boucle où il va attendre que les processus esclaves lui demandent du "travail". En réponse à cette demande, le maître va alors envoyer à l'esclave un certain nombre d'éléments à traiter. A chaque fois qu'un esclave va recevoir un nouvel élément du maître, il va le traiter puis combiner le résultat obtenu avec les résultats précédemment obtenus.

Puis une fois que tous les éléments de l'application auront été envoyés aux esclaves pour traitement, le maître va alors demander à ses esclaves de lui envoyer leur résultat respectif. Puis pour finir, il va combiner ces résultats entre eux afin d'obtenir le résultat final. Le programme suppose, ce qui est vrai dans notre application, que l'opérateur de combinaison est associatif et commutatif.

6.2.1 Eléments génériques communs aux deux algorithmes

- type $type\_ident$
- type $type\_result$

6.2.2 Algorithme Maître

Eléments génériques :

- constante $first\_ident : type\_ident\_item$
- constante $neutral\_result : type\_result$
- fonction $next\_ident : type\_ident\_item \rightarrow type\_ident\_item$
- fonction $operator : type\_result * type\_result \rightarrow type\_result$

Paramètres :

- entier nb_slaves
- entier nb_item_total
- entier $block\_item\_size$

Implémentation :

- - Envoi du travail à effectuer aux tâches esclaves.
- $nb\_item\_processed \leftarrow 0$
- $nb\_item\_to\_send \leftarrow block\_item\_size$
- **répéter**
    - **si** $nb\_item\_processed + nb\_item\_to\_send \geq nb\_item\_total$
      alors
        - $nb\_item\_to\_send \leftarrow nb\_item\_total - nb\_item\_processed$

      finsi

    - Attendre une demande de travail d'une des tâches esclaves
    - Récuperer l'identificateur de la tâche ayant fait la demande
    - Envoyer à cette tâche le nombre $nb\_item\_to\_send$
    - Envoyer à cette tâche les identificateurs des $nb\_item\_to\_send$ prochains éléments à traîter
    - $nb\_item\_processed \leftarrow nb\_item\_processed + nb\_item\_to\_send$

  jusqu'à $nb\_item\_processed = nb\_item\_total$

- **pour** i **de** 1 **à** nb_slaves
  faire
    - Attendre une demande de travail d'une des tâches esclaves
    - Récuperer l'identificateur de la tâche ayant fait la demande
    - Envoyer à cette tâche un message de fin de travail

  finpour

- - Calcul du résultat final.
- $final\_result \leftarrow neutral\_result$
- **pour** i **de** 1 **à** nb_slaves
  faire
    - Attendre un envoi de résultat d'une des tâches esclaves
    - Lire le résultat envoyé dans new_result
    - $final\_result \leftarrow operator(final\_result, new\_result)$

  finpour
- Afficher $final\_result$

6.2.3 Algorithme Esclave

Eléments génériques :

- type $type\_item$
- constante $neutral\_result : type\_result$

- fonction $read$: $type\_ident \rightarrow type\_item$
- fonction $processing$: $type\_item \rightarrow type\_result$
- fonction $operator$: $type\_result * type\_result \rightarrow type\_result$

Implémentation :

- $master \leftarrow tid\_tâche\_parent$
- $end\_work \leftarrow FALSE$
- $result \leftarrow neutral\_result$
- **tantque** non end_work
  faire
    - Envoyer à $master$ une demande de travail
    - Recevoir la réponse de $master$ dans nb_item
    - **si** $nb\_item \neq 0$
      alors
        - **pour** i **de** 1 **à** nb_item
          faire
            - Recevoir un identificateur d'élément dans $ident$
            - $elem \leftarrow read(ident)$
            - $partial\_result \leftarrow processing(elem)$
            - $result \leftarrow operator(result, partial\_result)$

          finpour

      sinon
        - $end\_work \leftarrow TRUE$
        - Envoyer $result$ à $master$

      finsi

  fintantque

6.3 Conclusion

L'application d'astronomie précédemment décrite a donc été implementée avec succès. La procédure permettant de calculer la FFT d'une image a été écrite à partir des librairies de calcul numérique proposés par [Rom94]. Toutefois, le calcul de la FFT n'est pas encore totalement optimisé, et pourrait encore être amélioré afin d'obtenir un programme qui soit plus rapide et plus efficace.

Chapitre 7

Conclusion

Ce stage a été enrichissant par de multiples aspects. Tout d'abord, les objectifs initialement fixés ont été atteints à savoir la réalisation d'une librairie Ada pour le calcul parallèle orientée données. De plus, l'utilité de cette librairie a pu être démontrée par l'écriture d'une application de reconstruction d'images astronomiques. On a pu ainsi voir que le langage Ada se prêtait fort bien au parallélisme, et que l'on pouvait également avoir des exécutions dans une configuration hétérogène (exécutables provenant de langages différents).

Toutefois, le travail effectué offre encore de nouvelles voies de recherche notamment sur le problème des communications globales. Nous n'avons pas encore trouvé les meilleurs moyens pour implémenter celles-ci sur des réseaux de station. Et on pourra également se pencher sur le problème des recouvrements de temps de calcul/communication. Ce problème a été abordé dans [Des94] et il pourrait être intéressant d'essayer de l'adapter aux architectures de réseaux hétérogènes utilisées par PVM 3.

Bibliographie

[Als87] Alsys. *Manuel de Référence du langage de programmation Ada*, Février 1987.

[Bar91] John Barnes. *Programmer en Ada*. Addison-Wesley, 1991.

[BS94a] Françoise Baude and David B. Skillicorn. Vers de la programmation parallèle structurée fondée sur la théorie des catégories. *TSI*, 1994. A paraître.

[BS94b] Jean-Claude Bermond and Michel Syska. *Architectures Parallèles*. Cours ESSI-3 et DEA, 1993-1994.

[Caq94] C. Caquineau. *CMF For beginners : what you should know*. Cours ESSI-3 et DEA, 1993-1994.

[Com90] Computing Surface Meiko. *CSTools Tutorial For C Programmers*, 1990.

[Des94] Frédéric Desprez. *Procédures de Base Pour le Calcul Scientifique sur Machines Parallèles à Mémoire Distribuée*. PhD thesis, CNRS, Lyon, Janvier 1994.

[DvdLVS94] Leen Dikken, Frank van der Linden, Joep Vesseur, and Peter Sloot. DynamicPVM, Dynamic Load Balancing on Parallel Systems. In *High-Performance Computing and Networking*, number 797. Lecture Notes in Computer Science, Avril 1994.

[GBD+93] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Ro bert Manchek, and Vaidy Sunderam. *PVM 3 USER'S GUIDE AND REFERENCE MANUAL*. Engineering Physics and Mathematics Division, Mai 1993.

[KKS90] E. Korach, S. Kutten, and S.Moran. A Modular Technique for the Design of Efficient Doistribued Leader Finding Algorithms. In *ACM Transactions on Programming Languages and Systems*, volume 12, Janvier 1990.

[LPS90] R.W. Little, R.H. Perrot, and P. Sritharan. Modeling SIMD-Type Parallelism in Ada. *Journal of Pascal, Ada & Modula-2*, 9, Mars/Avril 1990.

[Lun89] Lars Lundberg. A Parallel Ada System on an Experimental Multiprocessor. *Software — Practice and Experience*, 19, Août 1989.

[Mes94] Message Passing Interface Forum. *Document for a Standart Message-Passing Interface*, Février 1994.

[Paz93] J.L. Pazat. *Conception et Analyse d'algorithmes parallèles.* Ecole d'Automne CAPA, 1993. A paraître.

[Rif90] Jean-Marie Rifflet. *La communication sous Unix*, chapter 13. 1990.

[Rom94] Jean-Etienne Rombaldi. *Algorithme numérique en Ada.* Collection Logique Mathématiques Informatique, 1994.

[Saa94] Sami Saarinen. EASYPVM, EnhAnced Subroutine LibrarY for PVM. In *High-Performance Computing and Networking*, number 797. Lecture Notes in Computer Science, Avril 1994.

Annexe A

Mémento PVM 3

Les fonctions fournies par PVM 3 pour l'écriture de programmes parallèles peuvent être regroupées de la manière suivante :

- gestion de la machine virtuelle,
- gestion des tâches,
- les communications (signaux ou messages),
- les groupes (ensemble de tâches),
- gestion des erreurs

A.1 La machine virtuelle

Comme il a été dit précédemment, la machine virtuelle est constituée de plusieurs serveurs, ces serveurs étant connectés entre eux à l'aide de *démons*. On dispose de fonctions qui vont permettre d'obtenir des informations sur la configuration de la machine virtuelle, ou d'ajouter ou de retirer dynamiquement des serveurs en cours d'exécution.

Configuration de la machine virtuelle

Fonction : int pvm_config(int *nhost, int *narch, struct hostinfo **hostp)

Rôle : Cette fonction permet d'obtenir les informations suivantes :

- nombre de serveurs actuellement présents dans la machine virtuelle.
- nombre de formats de données[1] utilisées.
- informations diverses sur les serveurs.

[1] voir page 42.

Etat d'un serveur

a *Fonction* : int pvm_mstat(char *host)

a *Rôle* : Cette fonction qui a pour paramètre le nom d'un serveur permet de connaître l'état de ce serveur par rapport à la machine virtuelle. Le code retourné va indiquer si le serveur appartient ou n'appartient pas à la machine virtuelle, ou alors si le serveur est défectueux.

Ajout de serveurs

Fonction : int pvm_addhosts(char **hosts, int nhost, int *infos)

Rôle : Cette fonction permet d'ajouter un nombre `nhost` de serveurs, dont les noms sont passés en premier paramètre, à la configuration existante de la machine virtuelle. Si un problème survient au niveau de l'un des serveurs à ajouter, la fonction se terminera quand même mais va renvoyer un code d'erreur[2]. Il va alors falloir parcourir le tableau d'entiers passé en troisième paramètre pour déterminer lequel des serveurs a provoqué le problème.

Retrait de serveurs

Fonction : int pvm_delhosts(char **hosts, int nhost, int *infos)

Rôle : Le principe est le même que pour la fonction précédente si ce n'est que les serveurs dont les noms sont passés en paramètre vont être retirés de la machine virtuelle. A noter que si des tâches étaient en train de s'exécuter sur ces serveurs, elles seront tuées.
En plus de cette fonction, si le système PVM 3 s'aperçoit en cours d'exécution d'une application qu'un des serveurs est défectueux, celui-ci sera automatiquement retiré de la configuration. L'utilisateur devra de ce fait s'assurer de l'existence d'un serveur dans la machine virtuelle avant d'y exécuter un processus.

A.2 Les tâches

Une tâche PVM 3 est définie comme une unité de traitement[3]. L'application parallèle va donc être découpée en plusieurs tâches qui vont coopérer à la résolution de l'application en s'envoyant mutuellement des données.

Pour gérer l'ensemble des processus PVM 3 d'une application, l'utilisateur dispose donc de fonctions lui permettant de créer ou de détruire des tâches, ou alors d'obtenir des informations sur ces tâches. Une tâche sera représentée en PVM 3 par un entier appelé *identificateur de tâche*, qui est unique pour une tâche donnée.

Création d'un processus

Fonction : int pvm_mytid(void)

---

[2] voir page 46.

[3] de manière analogue à un processus Unix.

Rôle : Si cette fonction est appelée pour la première fois par le programme, celui-ci devient alors un nouveau processus PVM 3 et on a génération d'un nouvel identificateur de tâche — à moins que le processus n'est été créé par un appel à la fonction pvm_spawn.
Dans tous les cas, la fonction retourne l'identificateur du processus appelant.

Démarrage de tâches

Fonction : int pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)

Rôle : Cette fonction permet de démarrer `ntask` copies du processus exécutable `task`, avec `argv` en paramètres. Le paramètre `flag` permet de déterminer sur quels serveurs les nouveaux processus PVM 3 doivent être exécutés. Les valeurs possibles sont :

- `PvmTaskDefault` : PVM 3 choisit lui-même les serveurs.
- `PvmTaskHost` : Le paramètre `where` spécifie le serveur.
- `PvmTaskArch` : Le paramètre `where` spécifie l'architecture du serveur qui va être choisi par PVM 3.

On a également la possibilité de démarrer les processus sous un débugueur, ou alors de générer des traces d'exécution.
Normalement cette fonction retourne le nombre de processus qui ont été effectivement démarrés. En cas d'erreur, le tableau `tids` permettra d'obtenir les codes d'erreurs des processus ayant échoués.

Obtention de la tâche appelante

Fonction : int pvm_parent(void)

Rôle : Si le processus appelant a été démarré par un autre processus PVM 3 (appel à la fonction pvm_spawn), la fonction retourne l'identificateur de ce processus. Sinon, la valeur `PvmNoParent` est retournée.

Quitter PVM

Fonction : int pvm_exit(void)

Rôle : Cette fonction permet de dire au démon local que le processus appelant quitte PVM 3, et redevient donc un processus normal s'exécutant de manière séquentielle.

Tuer une autre tâche

Fonction : int pvm_kill(int tid)

Rôle : Cette fonction permet de tuer le processus ayant `tid` comme valeur d'identificateur de tâche, elle renvoie 0 en cas de succès.
A noter que cette fonction ne doit pas être utilisée par un processus pour se tuer soi-même, pour cela il devra appeler la fonction pvm_exit.

Etat d'une tâche

Fonction : int pvm_pstat(int tid)

Rôle : Cette fonction permet de connaître le statut du processus identifié par l'entier `tid`, le statut peut être :

- `PvmOk` : tâche en cours d'exécution.
- `PvmNoTask` : identificateur inconnu.

Informations sur les tâches

Fonction : int pvm_tasks(int where, int *ntask, struct taskinfo **taskp)

Rôle : De manière plus générale que la fonction précédente, cette fonction permet d'obtenir des informations sur un ensemble de tâches par le paramètre `taskp`[4]. Le paramètre `where` permet de spécifier les tâches pour lesquelles on veut obtenir des informations. La valeur 0 désigne l'ensemble des tâches de la machine virtuelle, mais on peut également par un identificateur de démon local désigner l'ensemble des tâches d'un serveur, ou alors désigner une tâche unique par son identificateur. Dans tous les cas, le paramètre `ntask` renvoie le nombre de tâches pour lesquelles des informations ont été obtenues.

A.3 Les communications

Les différentes tâches d'une application PVM 3 peuvent communiquer entre elles par l'envoi de données (messages) ou signaux. L'envoi de signaux est similaire à celui utilisé dans un traitement séquentiel.

L'envoi d'un message est découpé en 3 phases :

- Initialisation d'un buffer d'envoi,
- Stockage des données à envoyer dans le buffer,
- Envoi du message à un autre processus.

Le processus de réception d'un message est découpé en 2 étapes :

- Réception du message dans un buffer (ceci peut être fait à l'aide d'une routine bloquante ou non-bloquante),
- Déstockage des données reçues.

A.3.1 Gestion des buffers d'envoi/réception

Un processus PVM 3 peut gérer plusieurs buffers d'envoi et de réception, mais à un moment donné du déroulement de l'application, il n'y a qu'un seul buffer d'envoi *actif* et qu'un seul buffer de réception *actif*.

Un buffer d'envoi ou de réception est représenté par un entier, un *identificateur de buffer* qui est unique pour un buffer donné et un processus donné.

[4] cf p99 de "PVM 3 USER'S GUIDE AND REFERENCE MANUAL" pour le contenu de cette variable.

Création de buffer

Fonction : int pvm_mkbuf(int encoding)

Rôle : Cette fonction va créer un nouveau buffer de message, qui va utiliser le codage de données spécifié par le paramètre `encoding`. La valeur renvoyée est l'identificateur du nouveau buffer créé.

Initialisation de buffer

Fonction : int pvm_initsend(int encoding)

Rôle : Cette fonction permet d'initialiser le buffer d'envoi actif afin qu'il puisse véhiculer un nouveau message. Le rôle du paramètre `encoding` est le même que pour la fonction précédente.

Fonction : int pvm_freebuf(int bufid)

Rôle : Cette fonction permet de libérer la mémoire occupée par le buffer identifié par bufid afin qu'il puisse être utilisé pour un nouveau message.

Gestion des identificateurs de buffer

Fonction : int pvm_get*X*buf(void)

Rôle : Cette fonction retourne l'identificateur du buffer actif d'envoi (remplacer X par s) ou de réception (remplacer X par r). Si aucun buffer n'est actif, la fonction renvoie 0.

Fonction : int pvm_set*X*buf(int bufid)

Rôle : Cette fonction permet de changer le buffer actif d'envoi (remplacer X par s) ou de réception (remplacer X par r). L'identificateur du nouveau buffer est `bufid`, et l'identificateur de l'ancien buffer est renvoyé par la fonction.

Informations sur un buffer

Fonction : int pvm_bufinfo(int bufid, int *bytes, int *msgtag, int *tid)

Rôle : Cette fonction permet d'obtenir des informations sur le message contenu dans le buffer identifié par `bufid`. Les informations renvoyées sont :

– `bytes` : Longueur du message,
– `msgtag` : Marque du message[5],
– `tid` : Identificateur de la tâche ayant envoyée le message.

A.3.2 Stockage et déstockage des données

Pour les deux fonctions qui suivent, la figure A.1 donne la liste des terminaisons des noms de fonction de stockage ou de déstockage ($XXXX$), et pour chacune d'elle le type des données attendues par la fonction ($YYYY$).

[5]voir section A.3.3.

Stockage de données

Fonction : int pvm_pk*XXXX*(*YYYY* *data, int nitem, int stride)

Rôle : Les fonctions pvm_pk*XXXX* permettent de stocker dans le buffer actif d'envoi des données de type *YYYY*. Le paramètre `data` est un tableau du même type que la donnée, `nitem` est le nombre de données à stocker, et `stride` est la distance dans le tableau entre les différentes données à envoyer (vaut 1 en général).

Déstockage de données

Fonction : int pvm_upk*XXXX*(*YYYY* *data, int nitem, int stride)

Rôle : Les fonctions pvm_upk*XXXX* permettent de récupérer dans des tableaux de type *YYYY* les données stockées dans le buffer actif de réception. Le rôle des paramètres est le même que pour les fonctions pvm_pk*XXXX*.

Nom de la fonction (*XXXX*)	Type de la donnée (*YYYY*)
byte	char
cplx	float
dcplx	double
double	double
float	float
int	int
long	long
short	short
str	char

Figure A.1: Correspondance noms fonctions, type de données

A.3.3 Envoi et réception de données

Toutes les fonctions PVM 3 permettant d'envoyer ou de recevoir des données ont un paramètre supplémentaire appelé *identificateur de message* qui va permettre à une tâche d'affiner la sélection des messages attendus. Ainsi une tâche peut accepter de recevoir :

- n'importe quel message,
- tous les messages d'une source spécifiée,
- tous les messages avec un identificateur de message spécifié,
- les messages d'une source donnée et avec un identificateur donné.

Envoi de message

Fonction : int pvm_send(int tid, int msgtag)

Rôle : Cette fonction envoie le message qui se trouve dans le buffer actif d'envoi à la tache identifiée par `tid`, et avec `msgtag` comme identificateur de message.
Cette fonction est asynchrone bloquante : la main est rendue au processus appelant dès que le message est en route, de façon sécuritaire, vers le processus récepteur.

Multi-diffusion

Fonction : int pvm_mcast(int *tids, int ntask, int msgtag)

Rôle : Cette fonction permet d'envoyer un même message à `ntask` tâches dont les identificateurs sont rangés dans le tableau `tids`. Au niveau de chaque tâche, le traitement est le même que pour la fonction précédente.

Réception de message

Fonction : int pvm_nrecv(int tid, int msgtag)
int pvm_recv(int tid, int msgtag)

Rôle : Ces fonctions permettent à la tache appelante de recevoir un message de la tâche `tid` avec un identificateur de message égal à `msgtag`. Le message sera stocké dans le buffer actif de réception. En donnant -1 comme valeur à l'un ou aux deux paramètres, la tâche sera en mesure d'accepter la réception de plusieurs messages de sources ou d'identificateurs différents.
La fonction pvm_recv est bloquante : elle ne se termine que lorsqu'un message a été effectivement reçu, contrairement à la fonction pvm_nrecv qui elle se termine immédiatement. La valeur retournée permettra de savoir si un message a été reçu (0 si c'est le cas, < 0 sinon).

Test de réception d'un message

Fonction : int pvm_probe(int tid, int msgtag)

Rôle : Cette fonction permet de savoir si un message avec `msgtag` comme identificateur est arrivé de la tâche `tid`. Les deux paramètres peuvent comme précédemment prendre la valeur -1.
Si un message est arrivé, la fonction retourne un identificateur de buffer qui pourra être utilisé dans un appel à la fonction pvm_bufinfo[6].

A.3.4 Les autres fonctions de communication

Génération automatique de message

Fonction : int pvm_notify(int what, int msgtag, int ntask, int *tids)

Rôle : Après l'appel de cette fonction, si l'événement `what` a lieu, un message va être généré puis envoyé aux `ntask` tâches listées dans le tableau `tids`. L'identificateur de ce message sera `msgtag`. L'événement peut être `PvmTaskExit` ou `PvmHostDelete`, ou `PvmHostAdd`.

[6]voir page 42.

Envoi de signal

Fonction : int pvm_sendsig(int tid, int signum)

Rôle : Cette fonction permet d'envoyer le signal `signum` au processus PVM 3 identifié par `tid`. Le signal envoyé est l'un des signaux disponibles sous le système Unix.

A.4 Les groupes

Il existe en PVM 3 la possibilité de regrouper les tâches dans un même ensemble appelé *groupe*. On pourra alors désigner l'ensemble de ces tâches en donnant uniquement leur identificateur de groupe, ce qui sera notamment utile pour l'envoi de messages ou la synchronisation.

Une même tâche peut appartenir à plusieurs groupes de manière simultanée, et le nombre de tâches présentes dans un groupe peut varier dynamiquement au cours d'une application.

Rejoindre un groupe

Fonction : int pvm_joingroup(char *group)

Rôle : Cette fonction permet à la tâche appelante de rejoindre le groupe ayant pour nom `group`. Si ce groupe n'existe pas encore, il est créé.
Dans les deux cas, la fonction retourne le numéro d'instance de la tâche à l'intérieur du groupe, qui est unique pour une tâche donnée et un groupe donné.

Quitter un groupe

Fonction : int pvm_lvgroup(char *group)

Rôle : Cette fonction permet à la tâche appelante de quitter le groupe nommé `group`.

Informations sur un groupe

Fonction : int pvm_getinst(char *group, int tid)

Rôle : Pour un nom de groupe donné — `group`, et un identificateur de tâche donné — `tid`, cette fonction retourne le numéro d'instance de la tâche dans le groupe.

Fonction : int pvm_gettid(char *group, int inum)

Rôle : Pour un nom de groupe donné — `group`, et un numéro d'instance donné — `inum`, cette fonction retourne l'identificateur de la tâche ayant ce numéro d'instance dans le groupe.

Fonction : int pvm_gsize(char *group)

Rôle : Si le paramètre `group` est un nom de groupe existant, cette fonction retourne le nombre de tâches actuellement présentes dans ce groupe.

Synchronisation

Fonction : int pvm_barrier(char *group, int count)

Rôle : Cette fonction permet de synchroniser des tâches d'un même groupe. Pour cela, elle va bloquer le processus appelant jusqu'à ce que `count` membres du groupe nommé `group` aient appelé la fonction avec la même valeur pour le deuxième paramètre.
Il est nécessaire que la tâche appelante soit membre du groupe.

Envoi de message

Fonction : int pvm_bcast(char *group, int msgtag)

Rôle : Cette fonction permet d'envoyer un message avec `msgtag` comme marque de message[7] à toutes les tâches membres du groupe `group`.

A.5 Les erreurs

Toutes les fonctions que nous avons vues renvoient un code d'erreur si un problème a été détecté durant leur exécution. Ces problèmes peuvent être : l'inexistance de la tâche, du groupe ou du serveur pour lequel on veut obtenir ou modifier des informations, un problème de mémoire ou de système, ou alors un mauvais type de paramètre, ...[8] Et un code particulier est évidemment renvoyé pour signaler que l'exécution de la fonction PVM 3 s'est déroulée sans aucun problème.

Affichage d'un message d'erreur

Fonction : int pvm_perror(char *msg)

Rôle : Cette fonction affiche le message d'erreur relatif au dernier appel d'une fonction PVM 3. La chaîne `msg` sera affichée en tête de ce message d'erreur.

[7] voir section A.3.3.

[8] cf p34 de "PVM 3 USER'S GUIDE AND REFERENCE MANUAL" pour avoir une liste complète des codes d'erreur.

Index

Annexe B

La diffusion en PVM 3

Le programme utilisé pour réaliser une diffusion sur un ensemble de tâches précis (fonction `pvm_mcast`) ou sur un groupe (fonction `pvm_bcast`) est le même puisque l'appel de la fonction `pvm_bcast` avec un identificateur de groupe se ramène à un appel de la fonction `pvm_mcast` avec la liste des tâches actuellement présentes dans le groupe.

Algorithme de $bcast(groupe, msg)$

- Demande de l'identificateur du gestionnaire des groupes au démon
- Envoi au gestionnaire de groupes d'un message disant qu'une diffusion va être effectué sur le groupe *groupe*.
- Le gestionnaire des groupes renvoie la liste des identificateurs de tâches se trouvant actuellement dans le groupe.
- Appel de $mcast(liste\_tâches, msg)$

Algorithme de $mcast(liste\_tâches, msg)$

- Demande de connexion vers le démon PVM 3
- Envoi au démon d'un message indiquant qu'une diffusion va être faite, puis attente d'un accusé de réception.
- Travail du démon :
  - Allouer un nouvel identificateur de tâche qui servira uniquement pour la diffusion (*mid*)
  - Envoi à chacun des démons (y compris lui même) d'un message contenant *mid* et la liste des tâches du démon concernées par la diffusion.
- Les différents démons concernés par la diffusion mémorisent le *mid* et la liste des tâches locales.
- La tâche exécute un $mroute(msg, mid)$
  - Le démon envoie à chacun des autres démons un message contenant *msg* et *mid*.
  - Sur réception de ce message, chaque démon pvm va alors diffuser à ses tâches le message initial.

Annexe C

Interfacer C en ADA

On présente dans cette annexe la démarche à suivre pour pouvoir établir une interface entre les langages C et Ada.

Supposons que l'on veuille appeler en Ada une fonction C d'incrémentation qui prenne deux paramètres : la variable à incrémenter, et la valeur d'incrémentation. Les étapes à suivre sont les suivantes :

- **Ecriture de la fonction C** : fichier `incr.c`

  ```
  int incr(variable, valeur)
  int *variable; int valeur;
  {
       *variable += valeur;
       return 1;
  }
  ```

- **Compilation de la fonction C** : on obtient donc le code objet de la fonction dans `incr.o`

- **Ecriture du code ADA** : Il va d'abord falloir déclarer la fonction C comme une fonction externe (fichier `test.ada`) :

  ```
  procedure test is
      function incr(x : ADDRESS; inc : INTEGER) return INTEGER;
      pragma interface(c, incr);
  ```

  L'utilisation se fait alors de la manière suivante :

  ```
      x, ret : INTEGER;
  begin
      ret := incr(x'ADDRESS, 12);
  end test;
  ```

- **Compilation du programme ADA** : Il va alors falloir indiquer a l'éditeur de liens ADA où il peut trouver le code objet de la fonction C.

```
$ ada test.ada
$ auglib test incr.o
$ bamp test
```

On obtient ainsi un exécutable de la procedure `test`.

Remarques diverses :

- *le passage des paramètres par référence* : pour pouvoir récuperer les modifications effectués dans une procédure C sur une variable pointée (mode out en Ada), il faut passer à l'appel de la procédure non pas la variable mais son adresse.
- *pour les chaînes*, l'allocation de la place mémoire devra être faite dans le programme Ada[1].
- on aura également un problème si la fonction C a un paramètre qui est un pointeur sur une structure. Ce paramètre ne pourra pas être récupéré tel quel en Ada. Il va falloir écrire une autre fonction C qui se chargera de récuperer la structure et de passer les différents champs à la procédure Ada appelante dans des variables différentes[2].

[1] voir annexe D pour plus de détails
[2] voir implémentation de la fonction `config` dans annexe D

Annexe D

L'implémentation de la librairie ADA