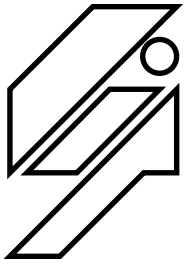
CAPNX, un environnement NX, PVM et MPI multi-utilisateurs sur MCS Capitan

Loïc Prylli



Laboratoire de l'Informatique du Parallélisme

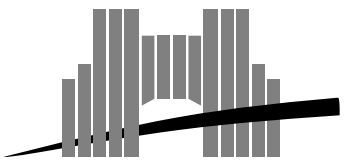
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

CAPNX, un environnement NX, PVM et MPI multi-utilisateurs sur MCS Capitan

Loïc PRYLLI

Décembre 1995

Research Report N^{o} 95-48



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

CAPNX, un environnement NX, PVM et MPI multi-utilisateurs sur MCS Capitan

Loïc PRYLLI

Décembre 1995

Abstract

We present here the use, the implementation and the performances, of the usual message-passing libraries NX, PVM and MPI on the Capitan machine. In particular, the implementation of a multi-user environment is described.

Keywords: Parallel computer, Capitan, CAPCASE, PVM, MPI

Résumé

Nous présentons ici, l'utilisation, l'implémentation et les performances des bibliothèques classiques de communication par échanges de messages NX, PVM et MPI sur la machine Capitan. En particulier l'implémentation de l'environnement multi-utilisateurs est décrite.

Mots-clés: Ordinateur parallèle, Capitan, CAPCASE, PVM, MPI

Table des matières

Introduction

L'environnement actuel de la Capitan offre une interface propriétaire CAPCASE au programmeur pour développer ses logiciels. Nous exposons ici comment nous avons construit au dessus de CAPCASE des bibliothèques fournissant respectivement les interfaces de programmation NX, PVM et MPI. Ainsi, la plupart des applications parallèles existantes peuvent désormais être exécutées sur la Capitan.

Cet environnement est entièrement basé sur la couche CAPCASE, ce qui permet de préserver l'investissement qui a été fourni pour CAPCASE tout en rendant le développement de cette sur-couche moins coûteux. Pour les systèmes d'échanges de messages, l'inconvénient d'une sur-couche se ressent en général par des performances moindres en comparaison d'une implémentation directe au dessus du matériel, dans notre cas les tests de performances montrent que les délais supplémentaires introduits par notre couche logicielle sont très faibles, et pratiquement négligeables par rapport au coût total des communications.

1 L'environnement de base de la Capitan

1.1 Programmation avec CAPCASE

1.1.1 Vue d'ensemble

Pour utiliser CAPCASE, le programmeur doit déclarer séparément dans un fichier à part, le fichier *LSL*, la répartition des tâches de l'application et les canaux de communication utilisées entre les tâches, l'allocation des tampons utilisés pour les communication se fait statiquement suivant les indications fournies par le programmeur dans le fichier *LSL*.

Le code de l'application utilise lui une interface propriétaire, les primitives de bases offrent des envois asynchrones, les primitives de réception usuelles, bloquantes ou asynchrones, avec sélection facultative sur la source. D'autre part on dispose d'une primitive de diffusion pour envoyer un message à un groupe de tâches déclarés statiquement dans le fichier *LSL*.

1.1.2 Discussion

L'informatique du parallélisme arrive dans une phase de maturation et aujourd'hui trois standards se sont imposés : HPF pour la programmation data-parallèle, PVM et MPI pour la programmation par échange de messages. Il paraît donc très improbable que qui que ce soit veuille investir des moyens importants dans un logiciel utilisant l'environnement CAPCASE de base et qui ne serait pas utilisable et portable sur les autres machines actuelles et futures du marché (sauf dans le cas ciblé du temps réel embarqué). Par conséquent il apparaît indispensable dans l'optique d'une machine généraliste que l'utilisateur puisse disposer au moins des standards de base tels que MPI[BHLS+95, Mes94], PVM[GBD+94]. De plus une quantité remarquable d'outils au niveau compilation, visualisation, analyse de performances existent déjà au dessus de ces interfaces. Cela permet donc au constructeur de concentrer ses efforts de développement uniquement sur la couche de bas niveau. C'est un avantage important lorsqu'on connaît le coût de développement d'un environnement complet de haut niveau.

1.2 Environnement d'exécution

Avec CAPCASE, les tâches de l'application sont réparties statiquement à l'aide du fichier LSL, le code de ces applications est lié avec le noyau d'exécution pour former une image par processeur qui est chargée lors de l'initialisation de la machine. Ces opérations sont faites à l'aide d'outils CAPCASE appropriés. L'exécution de deux applications différentes nécessite une réinitialisation complète de la machine, en effet le code de l'application, le noyau CAPCASE pour un processeur sont liés ensemble et CAPCASE ne supporte pas les tâches dynamiques. D'autre part pour une application donnée on a autant d'images exécutables que de nœuds dans la machine, même pour un code de type SPMD, ce qui a pour conséquence un temps de compilation relativement important.

2 Implémentation de NX au dessus de CAPCASE

2.1 Pourquoi NX?

La plupart des interfaces de programmation parallèle par échange de messages (NX, CMMD, PVM, MPI), gèrent dynamiquement les types de messages, et offrent une vue complètement connectée de la machine contrairement à l'environnement CAPCASE.

Nous avons décidé d'implémenter la librairie NX au dessus de CAPCASE. Le choix de NX a été fait pour deux raisons. Premièrement c'est une librairie bien représentative de la programmation par échanges de messages, deuxièmement il existe des implémentations de PVM et de MPI au dessus de NX, ce qui nous a permis de porter PVM et MPI avec un investissement en temps non prohibitif.

2.2 Utilisation des canaux CAPCASE pour NX

Ici nous appellerons « message NX » les messages échangés au niveau de l'application à l'aide des primitives NX, « message CAPCASE », ceux utilisés par la couche inférieure de notre implémentation, et qui permettent de réaliser effectivement la communication à l'aide des primitives CAPCASE.

Un fichier LSL est créé automatiquement par notre logiciel en fonction du nombre de tâches NX que l'utilisateur veut utiliser. Actuellement notre système répartit de manière modulaire les « nœuds » NX sur les processeurs de la Capitan et déclare en plus une tâche sur le processeur hôte qui s'occupera des entrées-sorties (on l'appellera tâche d'interface).

Notre implémentation déclare trois types de canaux de communication :

- Un canal pour les entrées sorties, il relie de manière bidirectionnelle chaque tâche de calcul à la tâche d'interface et transmet dans un sens les requêtes d'entrées-sorties et dans l'autre les réponses correspondantes. Chaque message commence par un entier identifiant le type de la requête. On a un type pour chaque primitive d'entrée-sortie implémentée : `open`, `read`, `write`, `close`, `ioctl`, `dup`, `stat`, `access`, ... Toutes les primitives POSIX concernant les entrées-sorties sont implémentées. L'utilisation de ce canal est détaillée dans la section 3.

- Un premier canal de messagerie : il relie chaque paire de tâches de calcul. Sur ce canal transiteront des en-têtes d'informations correspondant aux messages NX de l'application. Les petits messages NX sont complètement inclus dans cet en-tête. Les messages circulant sur ce canal sont de petite taille et bornés, ce qui permet à un récepteur de recevoir de tels messages dans un tampon de taille fixe sans en connaître a priori le contenu.

- Un deuxième canal de messagerie : il relie aussi chaque paire de tâches de calcul. Sur ce canal transitent les messages NX de taille importante, il est déclaré dans le fichier LSL avec une taille maximale de message très importante (supérieure à la mémoire disponible par nœud), mais en communication par paquets de sorte que la mémoire réservée par CAPCASE pour ses tampons de communications est limitée. Lorsqu'un message NX de taille importante doit être transféré, un en-tête est d'abord envoyé sur le premier canal de messagerie. Le destinataire en recevant cet en-tête peut alors prendre les dispositions nécessaires pour recevoir les données réelles sur le deuxième canal.

Cette organisation avec deux canaux de messagerie permet de ne pas imposer de limite statique à la taille des messages échangés par les applications sans réserver excessivement de mémoire. D'autre part en choisissant bien le seuil entre petit et gros messages, la latence supplémentaire introduite est tout à fait négligeable, en effet les petits messages NX sont envoyés comme un seul message CAPCASE sur le premier canal, et dans le cas des gros messages, le temps de communication de l'en-tête sur un canal séparé est négligeable devant le coût total de communication.

2.3 Contrôle de flot

Les primitives CAPCASE opèrent à bas niveau et ne permettent pas de mettre en file les envois de messages. C'est à dire, avant d'envoyer un message entre deux tâches, on doit être sûr qu'il n'y a pas déjà un message en cours sur le même parcours (ou tout du moins qu'il a déjà quitté complètement l'hypernœud de départ), sinon le premier message risque d'être écrasé partiellement par le second. CAPCASE positionne une variable dès qu'il est possible d'envoyer à nouveau un message. Or les primitives NX doivent implémenter l'envoi asynchrone de messages et les mettre en file de manière transparente pour l'application. Nous utilisons pour cela un mécanisme de mémorisation locale des messages sur le nœud émetteur. Les messages en attente sont mis dans une liste chaînée, et dès que le canal est disponible, le premier (s'il y en a un) est envoyé.

Le problème qui se pose est que la disponibilité du canal n'est testée que lorsque la tâche est en train d'effectuer une primitive NX. La situation suivante peut se produire : une tâche envoie deux messages consécutivement, et effectue de long calculs avant de rappeler une primitive NX. Le second message sera bufferisé localement, et bien que le canal soit libéré éventuellement assez tôt, ne sera envoyé que bien plus tard.

Pour résoudre cet inconvénient, il est utile dans certains cas de se bloquer en attente de la disponibilité d'un canal plutôt que de rendre la main, néanmoins il ne faut pas tomber dans la situation où tout le monde attend la disponibilité d'un canal ce qui provoquerait une étreinte fatale, on peut éviter ceci en adoptant une technique de « polling », on teste alternativement la disponibilité d'un canal et la présence d'un message à réceptionner, mais cette solution peut aussi provoquer des délais incongrus dans l'application : si une tâche A envoie à B et C, que A reste bloquée un certain temps dans le premier envoi, parce que B n'est pas prêt à réceptionner, le message de A à C sera pris en compte plus tard que nécessaire.

Pour éviter cela, nous avons choisi d'utiliser une propriété particulière de CAPCASE. Étant donné qu'un message CAPCASE implique des bufferisations à plusieurs niveaux, on peut réussir à lancer deux messages même si le destinataire n'a pas effectuée la primitive CAPCASE de réception (ceci est valable uniquement pour les messages de type court). Autrement dit, après un message de type court, le canal redevient disponible immédiatement indépendamment du destinataire (s'il n'y a pas déjà un message sur le même canal). Pour détecter cette situation (pas de message en cours sur un canal), nous rajoutons dans les en-têtes des messages des champs d'acquittement. Ainsi dans le cas où seul le dernier message envoyé de A à B n'a pas encore été acquitté par B, on est sûr que l'on peut attendre la disponibilité du canal $A \rightarrow B$ sans bloquer (en fait le canal est à nouveau disponible pratiquement immédiatement). Bien qu'il existe encore des cas où l'on puisse avoir des délais inutiles, ceux-ci correspondent à des situations quasiment inexistantes dans les applications courantes (A envoie plus de trois messages à la suite à B sans que B n'effectue de reception ni n'envoie de message à A) et de toute manière, même lorsqu'elles surviennent, la sémantique des primitives NX est respectée et garantit l'absence d'étreinte fatale dans une application correcte.

2.4 Implémentation des traces

Une facilité de traçage est implémentée dans la bibliothèque NX. Elle peut être activée ou désactivée au chargement de l'application, par une option appropriée. Après activation, chaque appel à une primitive de communication entraîne une écriture estampillée de la requête dans un tampon local au processeur, quand ce tampon est plein, il est écrit dans un fichier, ce système de tampon permet de limiter la perturbation introduite par le traçage dans le déroulement de l'application. D'autre part au début et à la fin de l'application, la différence de valeur entre les horloges des différents processeurs est évaluée, et les estampilles recueillies sont corrigés pour représenter un temps physique cohérent entre les différents processeurs. Nous supposons que chaque horloge est de la forme $d + (1 + \epsilon)t$ où t est le temps physique, d un décalage de base, et ϵ est le biais de l'horloge par rapport au temps physique. Des techniques plus évolués permettraient de prendre en compte les variations du biais lui-même au cours du temps. Elles s'avèrent superflues pour l'instant.

Les traces sont finalement converties au format PICL[Wor92] pour être visualisées avec par exemple l'outil Paragraph[HE91].

3 Implémentation des entrées-sorties

Avec CAPCASE est fourni un système permettant de faire des entrées-sorties à partir de n'importe quelle tâche. Ce système n'est pas documenté, et nécessite à la fois des ajouts dans le fichier LSL et l'appel de fonctions d'initialisation par les différentes tâches. C'est une première raison qui nous a poussé à écrire notre propre système d'entrées-sorties, utilisable de manière complètement transparente pour l'utilisateur. Cependant la principale raison vient du fait que le système original n'offre pas la flexibilité indispensable pour l'environnement dynamique multi-utilisateur réalisé et décrit dans la section 5.

Avec notre implémentation, toutes les entrées-sorties effectués par les tâches sont redirigées sur un processus Unix tournant sur la machine (ou le processeur hôte). La figure 1 représente les différentes couches logicielles impliqués dans l'exécution d'une opération d'entrée-sortie.

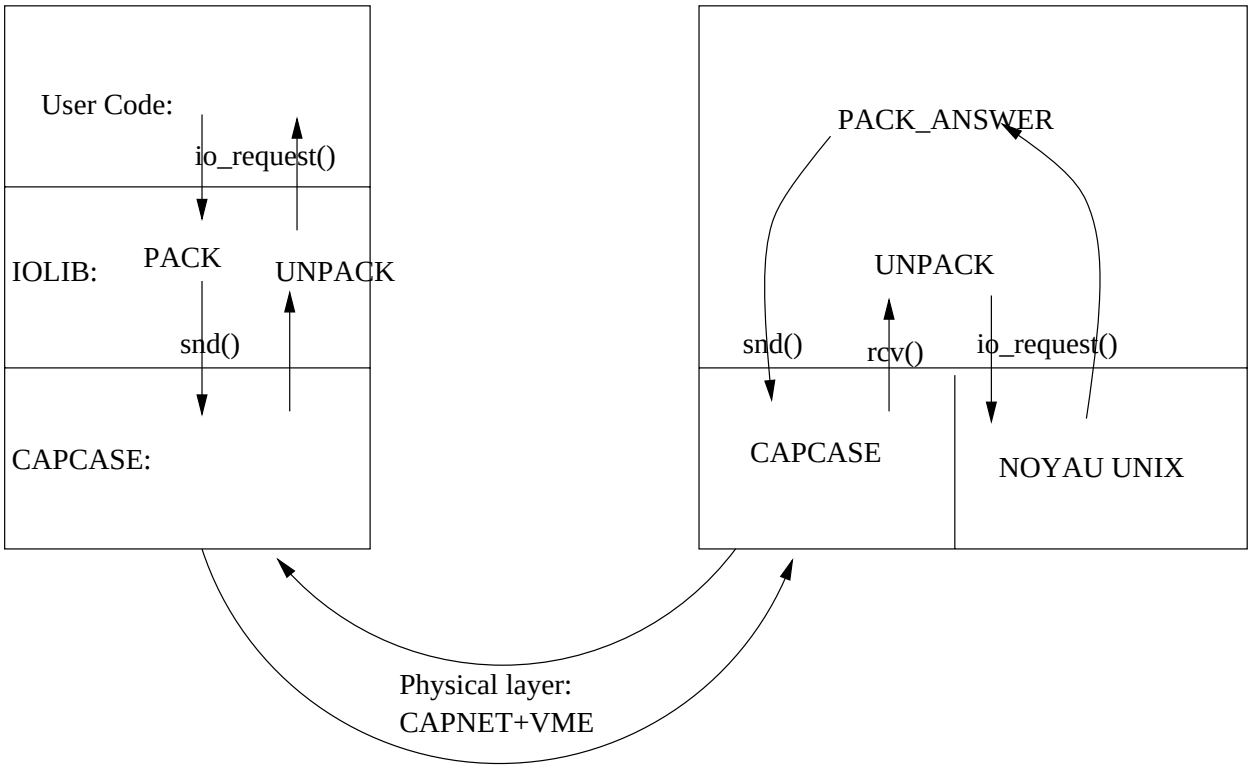


FIG. 1 - *Implémentation des entrées-sorties.*

Les E/S s'effectuent en utilisant un canal de communication dédié. Une opération d'entrée-sortie d'une tâche génère un message à destination de la tâche d'interface, dans lequel la requête a été encodée. La tâche d'interface effectue alors l'opération d'entrée-sortie proprement dite (sauf en configuration multi-utilisateurs, cf. ci-dessous), et retourne le résultat à la tâche de départ.

Notons quelques conséquences de ce mécanisme: l'ensemble des descripteurs de fichiers (au sens POSIX) est partagé par toutes les tâches d'une application, qu'elles soient sur le même processeur ou pas. Si chaque tâche ouvre un fichier, on aura autant de descripteurs ouverts que de tâches, mais on peut avoir une tâche qui ouvre un fichier et qui communique le descripteur de fichier aux autres tâches.

Dans le cadre d'une utilisation multi-utilisateurs (cf. section 5), ce n'est pas la tâche d'interface qui effectue l'appel système, elle se contente de faire passer le message au processus de contrôle de l'application (cf. section 5), qui lui effectue la requête, et répond à la tâche d'interface, qui fait passer cette réponse à la tâche originale. Ceci permet d'assurer que les opérations d'entrées-sorties sont bien faites dans le contexte

de l'utilisateur final (numéro d'`uid`, répertoire courant, droits d'accès, terminal de contrôle, entrée et sortie standards).

4 Chargement statique

Notre environnement offre en fait deux modes de fonctionnements, dans le premier cas l'application de l'utilisateur à laquelle a été lié notre bibliothèque, est utilisé directement comme tâche CAPCASE. Deux scripts sont alors utilisés : `allocnode` pour indiquer le nombre de tâches utilisées, ce script crée un fichier LSL adapté fonction du nombre de nœuds demandé et appelle les outils CAPCASE pour le compiler. Puis le script `sload` (pour Static LOAD) permet d'exécuter un programme sur la Capitan. Si c'est la première fois qu'on exécute ce programme, l'édition de liens finale est faite avec les outils CAPCASE (cf §1.2) de manière transparente pour l'utilisateur. Puis l'application est chargée et exécutée sur la Capitan.

La figure 2 présente l'interaction des différents composants.

5 Environnement dynamique multi-utilisateurs

La sur-couche que nous avons introduite jusqu'ici permet d'offrir un environnement sur la Capitan semblable aux autres machines parallèles du marché pour les programmes de type SPMD. Mais il ne résout pas l'un des problèmes de la Capitan : l'environnement mono-utilisateur, mono-tâche, avec reboot obligatoire (de l'ordre de cinq minutes) entre deux exécutions. Si ce n'est pas un problème en mode de production mono-tâche, cette restriction est rédhibitoire en mode de développement, où si l'on veut pouvoir partitionner les nœuds entre plusieurs utilisateurs. Cette section présente notre implémentation d'un environnement multi-applications, multi-utilisateurs toujours au dessus de CAPCASE.

La figure 3 présente les connections entre les différents composants logiciels et matériels nécessaire pour notre environnement dynamique.

En permanence, sur la Capitan tourne une application au sens CAPCASE. Cette application est constituée d'un pseudo-noyau sur chaque nœud, et d'une tâche d'interface sur l'hôte de la Capitan, cette dernière a cette fois un rôle plus complexe, elle communique avec les pseudo-noyaux pour permettre le chargement dynamique d'applications. Un pseudo-noyau sur requête de la tâche d'interface, réserve sur son processeur la place nécessaire à une nouvelle tâche, y copie le code et les données de la tâche que lui transmet la tâche d'interface, et donne le contrôle à cette application qui démarre. Lorsque l'application se termine, le pseudo-noyau reprend le contrôle. Il peut aussi reprendre le contrôle sur requête de la tâche d'interface, et « supprimer » une application en cours. Le processeur devient libre pour une nouvelle application. Les primitives CAPCASE sont disponibles pour l'application grâce à une table de fonctions initialisée par le pseudo-noyau. L'émulation de l'interface NX se fait à l'aide d'exactement la même sur-couche que dans le cas statique. La correspondance entre les nœuds au sens NX et les processeurs de la Capitan est choisie dynamiquement par la tâche d'interface et transmise à cette couche par l'intermédiaire du pseudo-noyau.

Au niveau de la machine hôte, le lancement de l'application est fait par un processus Unix que nous appellerons processus de contrôle. Le processus de contrôle se connecte à la tâche d'interface (qui est aussi un processus Unix) grâce à un « socket Unix ». Une fois la connection établie, le processus de contrôle fournit par ce lien de communication le nombre de nœuds requis et l'exécutable de l'application à lancer. Dès qu'il y assez de nœuds disponibles sur la Capitan, la tâche d'interface alloue le nombre de nœuds requis, envoie les messages d'initialisation appropriés aux pseudo-noyaux correspondants, et répond au processus de contrôle par un message d'acquittement. D'autre part, durant la durée d'exécution de l'application, tous les messages reçus par la tâche d'interface et concernant les entrées-sorties sont transmis au processus de contrôle des tâches correspondantes. Celui-ci effectue alors l'entrée-sortie dans le bon contexte (celui de l'utilisateur qui a lancé l'application), et transmet le résultat de la requête à la tâche d'interface qui le fait passer à la tâche destination.

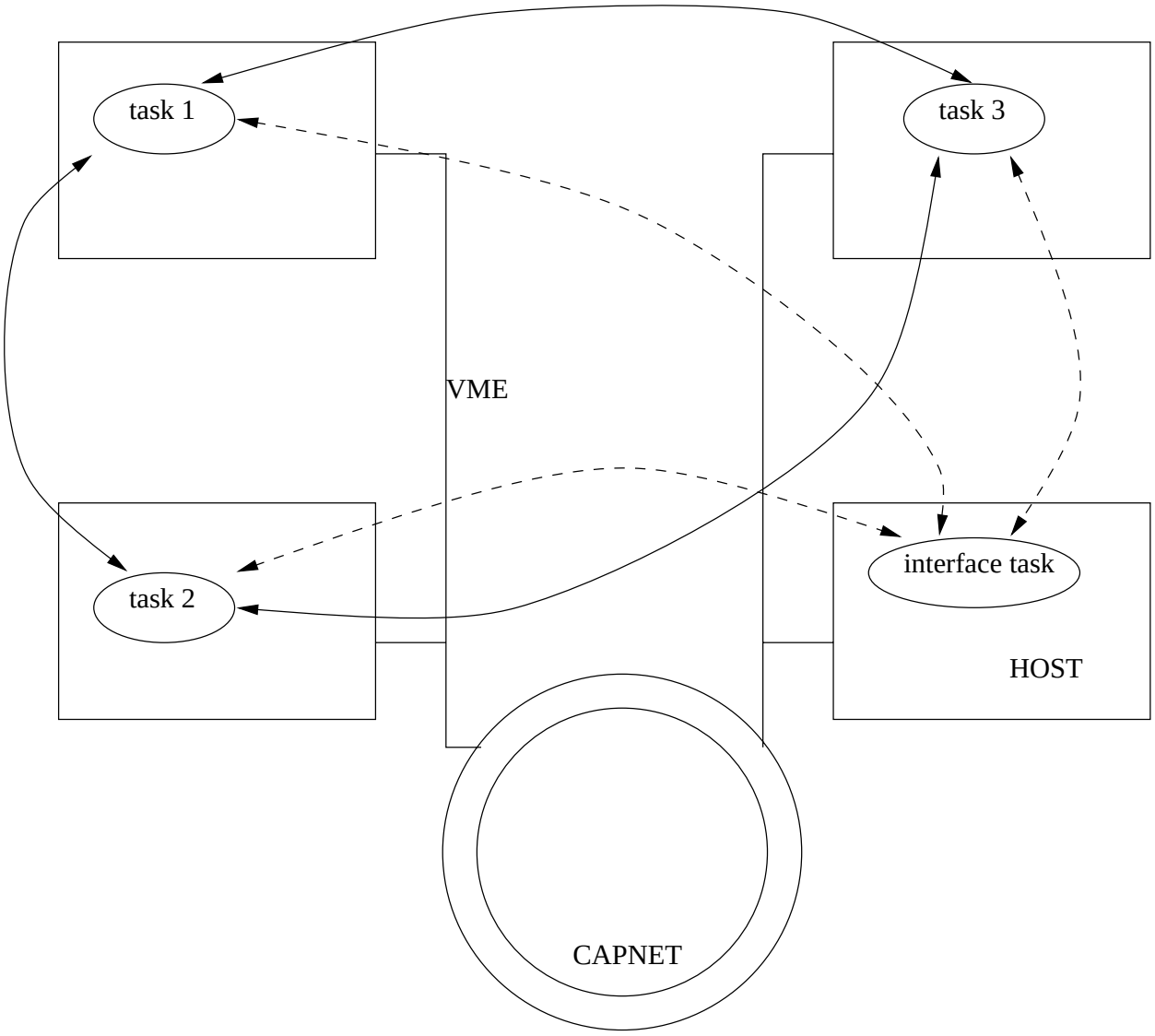


FIG. 2 - *Configuration en chargement statique.*

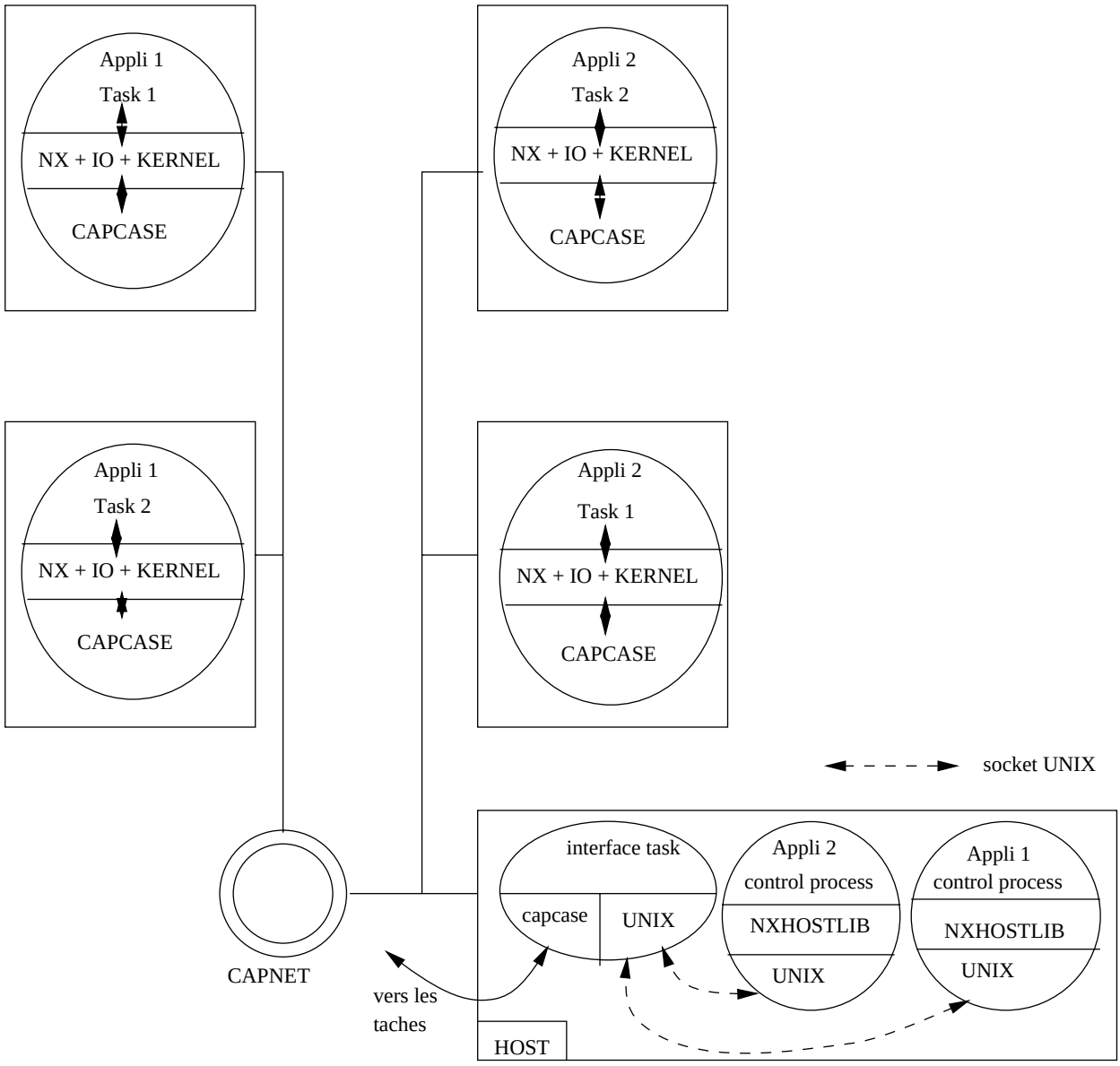


FIG. 3 - *Configuration en mode dynamique.*

Tout le protocole nécessaire pour communiquer avec la tâche de contrôle est implémenté dans une bibliothèque semblable à la bibliothèque NX disponible sur le Paragon pour les nœuds de service. Il y a une fonction pour initialiser une partition `nx_initve`, une fonction pour charger les exécutables sur les nœuds `nx_load`. On notera que l'on dispose seulement d'un sous-ensemble de l'interface disponible sur le Paragon pour le chargement et la gestion des nœuds d'une partition. Notre bibliothèque permet aussi de disposer comme dans le cas de l'iPSC/860 ou du Paragon des fonctions d'échange de message (`csend`, `crecv`,...) entre le processus de contrôle et les nœuds de calculs (ceci est capital pour l'implémentation de PVM).

En général, l'utilisateur final se contentera d'invoquer le programme `load`, avec en argument la taille de la partition et le nom de l'exécutable à charger sur les nœuds. De manière interne, `load` initialise la partition, charge les exécutables, et attend la fin de l'application.

Les nœuds sont alloués automatiquement lors du chargement d'un programme (NX, PVM ou MPI), et libérés dès la fin du programme (qu'elle soit normale ou par interruption, ou accidentelle du style accès mémoire illégal), ce qui optimise la disponibilité des nœuds en situation multi-utilisateurs.

6 Portage de PVM

La distribution PVM standard de UT/ORNL[GBD+94] a été installée avec succès sur la Capitan. Elle utilise l'interface NX, quelques modifications lui ont été apportées, pour gérer les différences entre une vraie machine Intel et la Capitan munie de notre environnement (les modifications concernent uniquement le format des données lors de la conversion XDR utilisé par PVM, et l'introduction d'un nouveau nom pour cette nouvelle architecture). La Capitan peut être utilisée sous PVM seule ou avec d'autres machines. PVM sur machines parallèles a quelques spécificités par rapport à la version réseau (cf. documentation PVM[GBD+94]), notre environnement n'ajoute aucune restriction par rapport à la version PVM sur d'autres machines parallèles.

PVM utilisant par nature le lancement dynamique de tâches, pour l'utiliser, il faut obligatoirement que la Capitan soit en environnement multi-utilisateurs. En fait, certaines machines parallèles, comme par exemple le Cray T3D imposent un chargement statique des tâches PVM en une seule fois et avec un seul exécutable. Cette restriction nécessitant généralement un port des programmes PVM écrit pour la version réseau, nous avons préféré bien sûr avec l'environnement Capitan en mode dynamique, offrir une compatibilité PVM totale, sans portage des programmes.

PVM sur Capitan a été validé grâce au jeu de tests fourni avec la version standard.

7 Portage de MPI

Nous avons utilisé MPICH : une version publique de MPI construite au dessus de l'interface NX développée conjointement par the Argonne National Laboratory and the Mississipi State National Laboratory. Cette version de MPI consiste principalement en une bibliothèque de fonctions MPI utilisant de manière internes les primitives NX. Un programme MPI s'exécute uniquement en mode SPMD, il n'y a pas de notion de chargement de programme. La compilation de cette bibliothèque de fonctions a permis de rendre disponible MPI sur la Capitan.

MPI a été validé sur la Capitan en faisant tourner la suite de tests de validation fournie indépendemment. Les programmes MPI peuvent être exécutés soit avec un chargement statique (commande `sload`) ou avec le mode de chargement dynamique (commande `load`).

8 Quelques tests de performances

Nous avons cherché ici à mesurer ce que coûtait en performance l'utilisation de notre environnement qui offre des fonctionnalités supplémentaires, des bibliothèques standards et un environnement dynamique. Dans

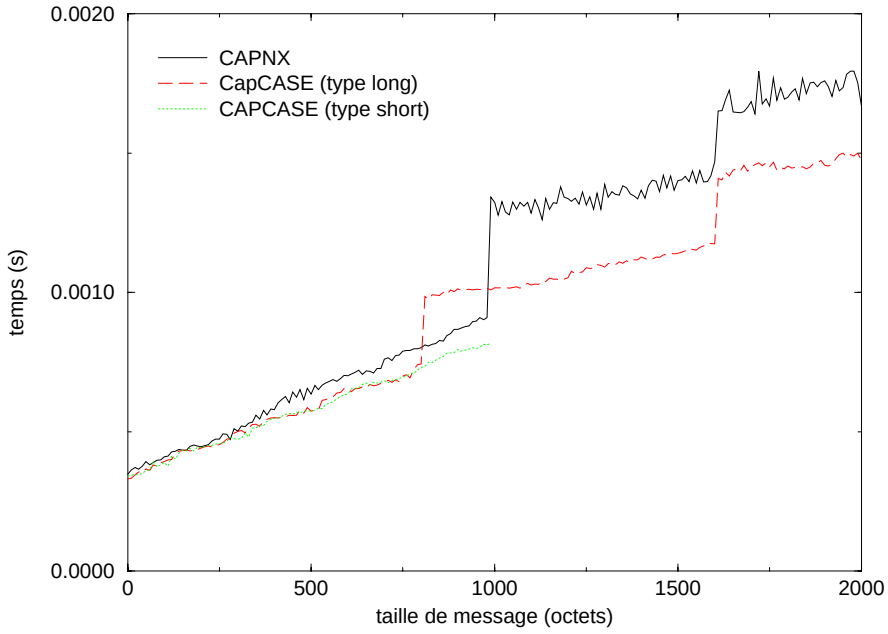


FIG. 4 - *Temps de transmission pour des petits messages directement avec CAPCASE ou avec notre environnement*

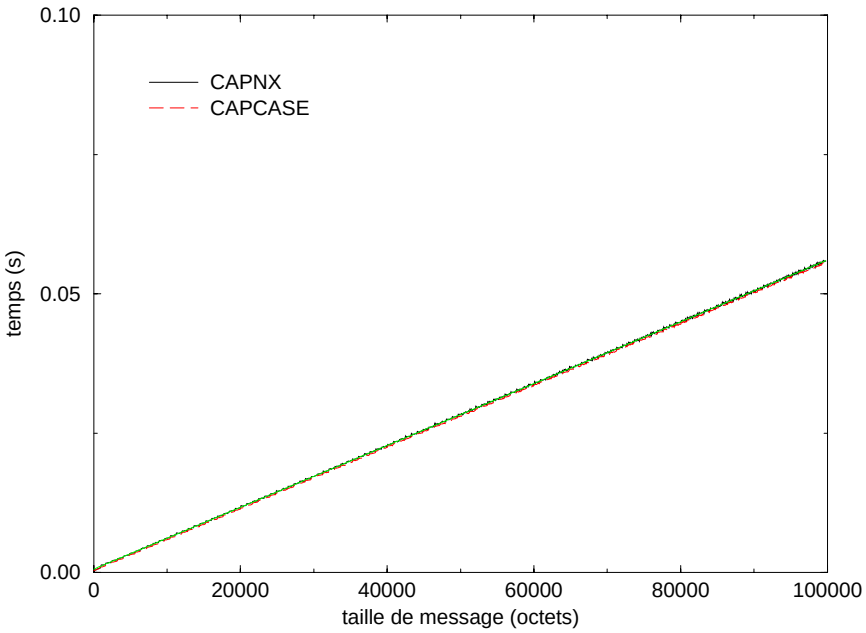


FIG. 5 - *Temps de transmission pour des gros messages*

la figure 4, nous avons représenté les temps de transmission en utilisant directement les primitives CAPCASE sur des messages de « type short » au sens CAPCASE ainsi que sur des messages de « type long ». La dernière courbe représente les temps de transmission en utilisant notre implémentation des primitives NX d'envoi et de réception, le résultat est bien conforme à notre stratégie décrite dans la section 2.2. Pour les petits messages, on utilise un canal CAPCASE de « type short », le temps supplémentaire par rapport à CAPCASE est très faible et vient essentiellement d'une recopie mémoire, nécessaire pour avoir une sémantique d'envoi non bloquante, tout en libérant le buffer utilisateur. Pour des gros messages, nous envoyons à la fois un message de « type short » pour l'en-tête et un message de « type long » pour les données, c'est ce qui provoque un saut important dans la courbe. On notera que la forme des courbes est liée très étroitement aux constantes utilisés pour les paquets CAPCASE et pour la définition des gros messages.

La deuxième figure montre une comparaison sur une échelle de taille de messages beaucoup plus large, et on constate qu'on obtient les mêmes performances avec notre implémentation qu'en programmant directement à plus bas niveau, le délai supplémentaire introduit est tout à fait négligeable.

En conclusion, la couche supplémentaire que nous avons introduite ne dégrade absolument pas les performances. La plupart du temps l'overhead introduit est très inférieur à 10%, seul la plage des tailles 1000 à 2000 dépassent ce seuil, l'overhead est alors de 20% à 25% environ. Nous envisageons de modifier le seuil où l'on sépare l'en-tête et les données, de manière à pratiquement supprimer cette zone, en augmentant la tailles des tampons pour les messages de « type short ».

9 Conclusion

Notre environnement permet de disposer sur la Capitan d'un environnement multi-utilisateurs offrant la flexibilité disponible actuellement sur toutes les machines parallèles récentes. Nous avons une première version testée et ayant prouvé sa fiabilité, elle possède une restriction : le partitionnement des processeurs n'est pas implémenté, et les applications des différents utilisateurs sont en fait exécutées à tour de rôle (le serveur gère une file d'attente). Une version complète avec plusieurs utilisateurs simultanés répartis sur les noeuds de la machine est en beta-test.

La machine Capitan peut aussi bien être utilisée en mode de production qu'en mode de développement avec de multiples exécutions courtes successives (les délais de chargement d'une application étant maintenant négligeables), mais aussi en mode « serveur de calcul » où des utilisateurs multiples peuvent soumettre des travaux à effectuer, qui seront traités dès que le nombre de processeurs disponibles le permet. La plupart des applications parallèles disponibles utilisent les interfaces PVM, MPI ou NX, et donc peuvent être exécutées sur la Capitan.

Des extensions peuvent être envisagées pour permettre le déboguage des tâches tournant sur les nœuds, et une sécurité accrue de l'environnement multi-utilisateurs. Des améliorations de performance peuvent être effectuées en intégrant la couche CAPCASE avec notre environnement de manière à pouvoir traiter les communications à un plus bas niveau. D'autre part la disponbilité récente de du noyau « Sphynx » sur la Capitan va permettre de nouvelles possibilités, en particulier plusieurs tâches sur un seul processeur.

A Mode d'emploi de NX et MPI sur Capitan

Pour utiliser les différentes commandes de l'environnement il faut exécuter (ou ajouter dans votre .cshrc) sur la machine capitan (pollux):

```
# commandes generales
set path=(~lprylli/capnx/bin $path)

# pour PVM
```

```
setenv PVM_ROOT /home/lprylli/pvm3
setenv PVM_ARCH MATRA
set path=(~lprylli/pvm3/lib $path)
```

A.1 Compilation d'un programme NX

```
icc <.c> <.o> [ -o progname ]
```

`icc` accepte les options standarts des compilateurs C et a de plus une option d'aide en ligne, vous pouvez faire `icc -help`.

Il est recommandé d'inclure le fichier d'entête `"nx.h"` dans les modules utilisant les primitives NX.

A.2 Compilation d'un programme MPI

Pour compiler un programme MPI, rajouter la librairie MPI:

```
icc <.c> <.o> [ -o progname ] -lmpi
```

Il est nécessaire d'inclure le fichier d'entête `"mpi.h"` dans les modules utilisant les primitives MPI.

A.3 Exécution d'un programme NX ou MPI

Une seule commande très simple de chargement :

```
load [ -sz <nbnode> ] [-trace] <pathname> [ program arguments ...]
```

Vous pouvez faire `load -help` pour l'aide en ligne.

A.4 Exemple complet NX

Programme jeton écrit avec la bibliothèque NX :

```
#include <stdio.h>
#include <cube.h>


/* ce programme fait circuler un jeton le long d'un anneau forme par les differents processeurs*/

#define TYPE 0

int main()
{
  int next;
  static char buf[BUFSIZ];
  setvbuf(stdout,buf,_IOLBF,BUFSIZ);
```

```
  next = mynode() + 1;
  if (next == numnodes())
    next = 0;
  if (mynode () == 0) {
    printf("jeton demmarre sur noeud 0\n");
    csend(TYPE,NULL,0,next,0);
    crecv(TYPE,NULL,0);
    printf("jeton arrive sur 0\n");
  } else {
    crecv(TYPE,NULL,0);
    printf("jeton sur %d\n",mynode());
    csend(TYPE,NULL,0,next,0);
  }
}
```

Compilation et exécution :

```
pollux%icc jeton.c -o jeton
pollux%load jeton
from 0:Application started on 4 nodes
jeton demmarre sur noeud 0
jeton sur 1
jeton sur 2
jeton sur 3
jeton arrive sur 0
all nodes have exited
from -1:end of the application
pollux%
```

A.5 Exemple complet MPI

Programme jeton écrit avec la bibliothèque MPI :

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define TAG 10

int main(int argc,char **argv)
{
  MPI_Status stat;
  int myid,numnodes,next;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myid);
  MPI_Comm_size( MPI_COMM_WORLD, &numnodes);
  next = myid + 1;
  if (next == numnodes)
    next = 0;
```

```
  if (myid == 0) {
    printf("jeton demmarre sur noeud 0\n");
    MPI_Send(NULL, 0 , MPI_INT, next, TAG, MPI_COMM_WORLD );
    MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD,&stat);
    printf("jeton arrive sur 0\n");
  } else {
    MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD,&stat);
    printf("jeton sur %d\n",mynode());
    MPI_Send(NULL, 0 , MPI_INT, next, TAG, MPI_COMM_WORLD );
  }
  MPI_Barrier( MPI_COMM_WORLD );
  MPI_Finalize();
}
```

Compilation et exécution :

```
pollux%icc jeton.c -lmpi -o jeton
pollux%load jeton
from 0:Application started on 4 nodes
jeton demmarre sur noeud 0
jeton sur 1
jeton sur 2
jeton sur 3
jeton arrive sur 0
all nodes have exited
from -1:end of the application
pollux%
```

B Mode d'emploi de PVM sur Capitan

B.1 Compilation de tâches PVM

Pour une tâche en C qui tournera sur la CAPITAN, se logger sur la Capitan:

```
pollux% icc -I$PVM_ROOT/include <.c> <.o> -L$PVM_ROOT/lib/MATRA -lgpvm3 -lpvm3pe
```

En FORTRAN:

```
pollux% if77 -I$PVM_ROOT/include <.f> <.o> -L$PVM_ROOT/lib/MATRA -lfpvm3 -lgpvm3 -lpvm3pe
```

Placer l'executable dans `~/pvm3/bin/MATRA` (ou créer un hostfile approprié, cf. documentation PVM).

Pour compiler une tâche en C qui doit tourner sur le processeur hôte de la Capitan (par exemple un tâche maître dans un schéma maître/esclave) faire :

```
pollux%gcc -I$PVM_ROOT/include <.c> <.o> -L$PVM_ROOT/lib/MATRA -lgpvm3 -lpvm3
```

En FORTRAN:

```
pollux% f77 -I$PVM_ROOT/include <.f> <.o> -L$PVM_ROOT/lib/MATRA -lfpvm3 -lgpvm3 -lpvm3
```

B.2 Lancement de tâches PVM

Après avoir démarré PVM sur l'hôte de la Capitan, un programme PVM peut être lancé avec un `pvm_spawn` dans une tâche PVM tournant soit sur l'hôte soit sur une autre machine, ou en faisant un spawn dans une console PVM (cf. doc PVM). Si l'environnement NX n'est pas chargé, celui-ci sera chargé automatiquement, ce chargement entrainant un délai lors de la première utilisation.

B.3 Exemple PVM

Listing du programme jeton pour PVM :

```
/* jeton.c */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include <pvm3.h>

#define NBTASKS 4
#define GROUP "token_ring"

#define TAG_INIT 1
#define TAG_TOKEN 2

/* to use this program type "spawn -4 token" at the PVM console prompt */

int main()
{
  int mytid;
  int next;
  int me;
  int parent;

  setvbuf(stdout,NULL,_IOLBF,BUFSIZ);
  pvm_setopt(PvmAutoErr,1);

  mytid = pvm_mytid();
  me = pvm_joingroup(GROUP);
  printf("node %d started,tid=%d\n",me,mytid);
  pvm_barrier(GROUP,NBTASKS);

  next = pvm_gettid(GROUP,me == NBTASKS -1 ? 0 : me + 1);
  if (me == 0) {
    printf("token start on 0\n");
    pvm_initsend(PvmDataDefault);
    pvm_send(next,TAG_TOKEN);
    pvm_recv(-1,TAG_TOKEN);
    printf("token arrive on 0\n");
  } else {
    pvm_recv(-1,TAG_TOKEN);
    printf("token is on %d\n",me);
    pvm_initsend(PvmDataDefault);
```

```
    pvm_send(next,TAG_TOKEN);
  }
  pvm_exit();
  return 0;
}
```

Compilation et exécution :

```
pollux%icc -I$PVM_ROOT/include -L$PVM_ROOT/lib/MATRA jeton.c \
                -lgpvm3 -lpvm3pe -o $HOME/pvm3/bin/MATRA/jeton
pollux%pvm
pvm> spawn -4 jeton
4 successful
t40002
t40003
t40004
t40005
pvm> quit

pvmd still running.
gueuze$cat /tmp/pvml.536
[t80040000] ready  3.3.7   Thu Oct  5 13:14:27 1995
[t80040000] [t40011] node 0 started,tid=262161
[t80040000] [t40012] node 1 started,tid=262162
[t80040000] [t40010] node 2 started,tid=262160
[t80040000] [t40013] node 3 started,tid=262163
[t80040000] [t40011] token start on 0
[t80040000] [t40012] token is on 1
[t80040000] [t40010] token is on 2
[t80040000] [t40013] token is on 3
[t80040000] [t40011] token arrive on 0
```

Références

[BHLS+95] Christian Bischof, Steven Huss-Lederman, Xiaobai Sun, Anna Tsao, and Thomas Turnbull. A case study of mpi: Portable and efficient libraries. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, 1995.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. Scientific and Engineering Computation. MIT Press, 1994.

[HE91] M. T. Heath and J. A. Etheridge. Visualizing performance of parallel programs. Technical report, Oak Ridge National Laboratory, 1991.

[Mes94] Message Passing Interface Forum. *Document for a Standard Message-Passing Interface*, November 1994.

[Wor92] P. Worley. A new picl trace file format. Technical Report TM-12125, Oak Ridge National Laboratory, October 1992.