Chapitre 7

Modèle de programmation à mémoire distribuée

Dans une architecture à mémoire distribuée, l'espace mémoire est fragmenté entre les différents processeurs. Afin de coopérer, les processeurs doivent se transmettre des informations tout au long du calculs. L'architecture à mémoire distibuée impose que ces échanges se fassent à travers des messages inter-processeurs, dans le même esprit que l'on envoie une lettre par la poste pour communiquer avec quelqu'un qui est éloigné.

Cette façon de communiquer donne lieu à un modèle de programmation spécifique, appelé **modèle par échange de messages**. Il est mis en oeuvre grâce à des appels à des bibliothèques d'échange de messages spécialisées, à partir de langages de programmation classiques tels que Fortran, C, C++ ou Java.

Les bibliothèques d'échange de messages les plus courantes sont MPI (Message Passing Interface) et PVM (Parallel Virtual Machine). Le but de ce chapitre n'est pas d'expliquer comment utiliser MPI ou PVM mais plutôt d'indiquer la philosophie de ce modèle de programmation, ses principes de base, son implémentation possible et des applications typiques.

7.1 Principes de base

Dans le modèle à échange de message, il n'y a pas de variables globales. Chaque processeur n'a accès qu'à sa propre mémoire et le programmeur est responsable de gérer la répartition des données et les transferts d'information.

En général, un programme utilisant le modèle à échange de message est du type SPMD (Single Program, Mutilple Data). Chaque processeur exécute ainsi le même programme, de façon asynchrone. Cependant, le comportement de chaque programme peut être différencié sur la base de l'identité du processeur qui l'exécute.

En effet, un ingrédient fondamental de la programmation par échange de

message est que chaque processeur se voit attribuer une adresse (ou encore une identité ou un rang) qui le distingue des autres. Par exemple dans MPI, les p processeurs [1] participants à une exécution parallèle sont numéroté de 0 à $p-1$. Ainsi, chaque processeur peut avoir accès au rang qui lui a été attribué par l'environnement d'exécution (`mpirun` par exemple) en appelant une fonction adéquate de la bibliothèque. Typiquement, dans une pseudo-syntaxe, la fonction

```
my_rank(id,nb_procs)
```

retourne, pour chaque processeur, une valeur différente entre 0 et $p-1$ dans la variable locale `id`. Le nombre de processeurs p choisi pour l'exécution en cours est aussi accessible à travers la variable `nb_procs`.

L'esprit de la programmation par échange de message est donc de paramétrer le programme en fonction de cette adresse `id` et du nombre de processeurs p.

Le deuxième élément essentiel du modèle de programmation par échange de message est justement sa capacité d'envoyer et recevoir des messages entre n'importe quelles paires de processeurs. Il y a donc deux primitives fondamentales de communication `SEND` et `RECEIVE`.

La fonction `SEND` prend en argument le rang du destinataire souhaité du message, ainsi que la nom de la variable contenant la valeur à communiquer à ce destinataire. De façon analogue, `RECEIVE` précise l'identité de l'expéditeur en provenance duquel un message est attendu, ainsi que le nom d'une variable où sera stocké l'information reçue.

L'exemple ci-dessous illustre l'envoi du contenu de la variable `a` contenue dans le processeur de rang 3 au processeur de rang 8. Ce dernier stocke la valeur reçue dans sa variable `b`

```
my_rank{id,nb_procs}
if(id==3){ a=100; send(a,8)}
if(id==8){receive(b,3)}
```

En pratique, d'autres arguments sont nécessaires pour les fonctions d'envoi et de réception. De même, il y a de nombreuses variantes de `SEND` et `RECEIVE` qui permettent de préciser plus finement les modes d'envoi ou de réception (bloquant ou non-bloquant,...) Nous renvoyons le lecteur au paragraphe 7.4 ou à un guide de programmation détaillé pour en savoir plus (p.ex. Parco, MPI doc,..).

Les bibliothèques d'échange de messages offrent aussi la plupart des primitives de communications collectives décrites au paragraphe 3.1.3, ainsi qu'une **barrière de synchronisation**, qui est une forme de communication collective, permettant de bloquer l'avancement des processeurs jusqu'à ce qu'ils aient tous effectué l'appel à cette primitive barrière. Cela peut être nécessaire pour gérer

1. Plus généralement, on devrait dire les p processus car le nombre de processeurs demandés peut excéder le nombre de processeurs physiquement disponibles.

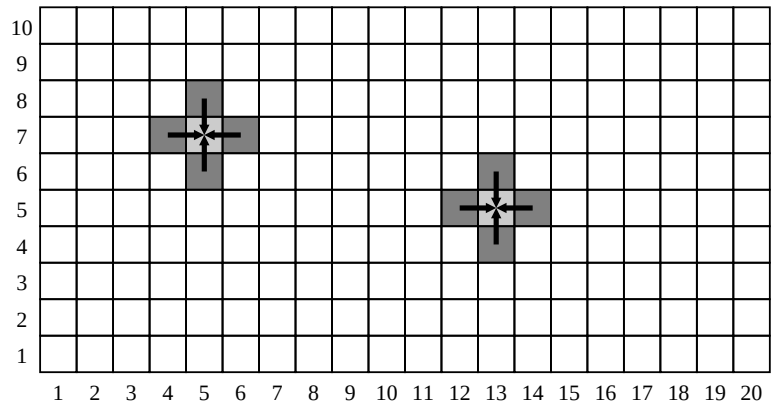

FIGURE 7.1 – *Domaine de calcul sur lequel on applique la relation itérative 7.1. Graphiquement, ce calcul peut s'illustrer par l'échange d'information indiqué par les flêches. Le point en gris clair est le point dont on calcul la nouvelle valeur et, ceux en gris foncé, sont les points dont la valeur est nécessaire.*

l'accès à une ressource partagée comme par exemple un fichier de données, ou mesurer le temps d'exécution d'un programme parallèle.

En général, les bibliothèques d'échange de messages permettent de diviser l'ensemble des processeurs disponibles en plusieurs groupes. Les opérations collectives sont alors restreintes à un groupe donné, identifié par `group_id` ou, dans MPI, son *communicator*. Des opérations inter-groupes sont évidemment aussi possibles.

7.2 Un exemple typique

Le modèle de programmation par échange de message est particulièrement bien adapté à la parallélisation du calcul itératif sur une grille. Par exemple, on considère un domaine de calcul en 2D, de taille $n \times m$ et on suppose que le but du calcul est d'itérer k fois la transformation :

$$s_{ij} = f(s_{i-1,j}, s_{i+1,j}, s_{i,j-1}, s_{i,j+1}) \tag{7.1}$$

où f est une fonction donnée et s_{ij} des valeurs numériques associée à chaque point de grille de coordonnée (i, j). On suppose que des valeurs initiales pour s_{ij} sont fixées et que le domaine de calcul global est périodique c'est à dire les les indices i et j sont pris respectivement modulo n et modulo m. La figure 7.1 illustre le domaine de calcul. Pour paralléliser un tel calcul, on commence par

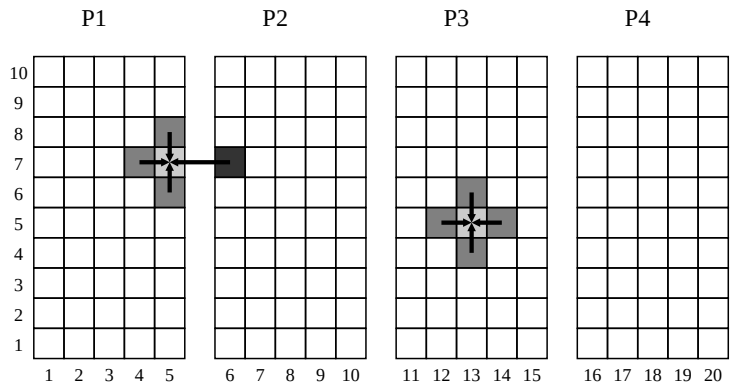

FIGURE 7.2 – *Partitionnement du domaine de calcul de la figure 7.1 en sous-domaines et illustration de la communication inter-processeur qui est necessaire pour calculer les points frontières.*

fractionner (on dit en pratique **partitionner**) le domaine de calcul en p tranches de taille $n \times (m/p)$, chacune assignée à un processeur différent. Dans chaque processeur, un tableau 2D est définit pour contenir ces valeurs, comme illustré dans la figure 7.2. Le calcul des s_{ij} qui sont strictement à l'intérieur de chaque sous-domaine se fait sans difficulté par chacun des processeurs en parallèle. Par contre, pour les points sur les fontières des sous-domaines le calcul demande une information non-locale, détenue par les processeurs voisins. Ces derniers ont d'ailleurs eux aussi besoin de l'information contenue chez leur propres voisins. Il faut donc que chaque processeur envoie aux autres les valeurs qui leur sont utiles et attendent de ces derniers la réciproque. La figure 7.3 montre comment les colonnes fontières de chaque sous-domaine sont envoyées par un processeur à son voisin. Une fois cet échange d'information terminé, les points frontières peuvent être calculés et on peut passer aux itérations suivantes. Le pseudo-code parallèle réalisant une telle itération de calcul est par exemple :

```
// compute internal nodes
 for(i=0,i<n,i++){
    for(j=1,j<m/p-1,j++){
        s_new(i,j)=f(s(i-1,j),s(i+1,j),s(i,j-1),s(i,j+1))
        }
     }

// exchange boundary nodes
 send(array=s(:,0),size=n,dest=id-1)
 send(array=s(:,m/p-1),size=n,dest=id+1)
 receive(array=left_column,size=n,source=id-1)
```

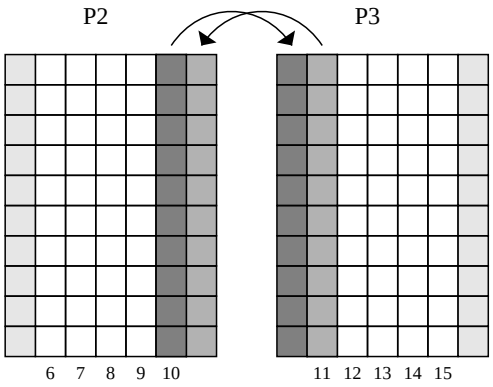

FIGURE 7.3 – *Illustration de la communication entre processeurs voisins afin de réaliser le calcul demandé par la relation 7.1. Pour des raisons d'efficacité de la communication interprocesseur, il est préférable d'envoyer les colonnes frontières en une seul fois et de les recevoir dans des colonnes que l'on adjoint aux sous-domaines de calcul.*

```
 receive(array=right_column,size=n,source=id+1)

// compute boundary nodes
 J=m/p-1
 for(i=0,i<n,i++){
        s_new(i,0)=f(s(i-1,0),s(i+1,0),left_column(i),s(i,1))
        s_new(i,J)=f(s(i-1,J),s(i+1,J),s(i,J-1),right_column(i))
        }
     }
```

Dans cet exemple on a implicitement supposé que `i`$\pm$`1` est correctement calculé modulo n, de même que `id`$\pm$`1`.

7.3 Décomposition en tranche, en morceaux et cyclique

Dans le modèle à échange de message, la répartition des données sur les processeurs est laissée au soin du programmeur. Ce paragraphe illustre les partitionnements courrants lorsque le domaine de calcul est régulier.

Nous considérons ici le cas de 16 valeurs a_{ij}, $i, j = 1, ..., 4$, à répartir sur un réseau carré de 2×2 processeurs. Le découpage ou répartition en tranche discuté ci-dessus est illustré dans la figure 7.4.

Mais on peut aussi choisir un découpage par **morceau** ou **blocs**. Dans notre

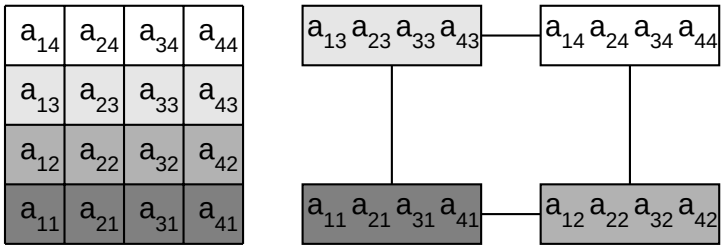

FIGURE 7.4 – *Répartition en tranches (ici en lignes) de la structure de données de gauche sur quatre processeurs (partie droite de l'image) interconnectés selon une grille* 2×2.

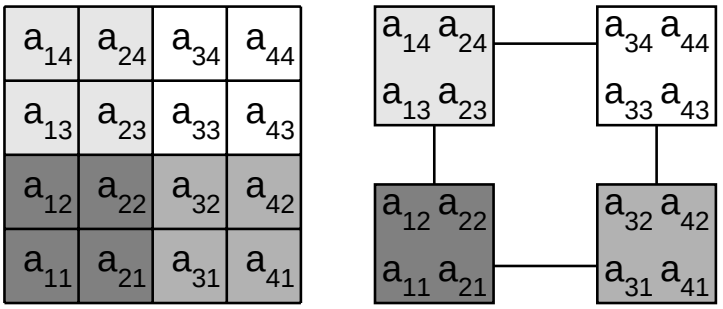

FIGURE 7.5 – *Répartition en morceaux (blocs) de la structure de données de gauche sur quatre processeurs (à droite) interconnectés selon une grille* 2×2.

exemple, on divisera le tableau original par blocs 2×2 comme indiqué sur la figure 7.5.

On peut aussi envisager une distribution des données dite **cyclique** ou **modulo**, comme illustré sur la figure 7.6.

On remarque que cette répartition fait perdre la contiguité des sous-domaines, ce qui peut générer des communications supplémentaires, par exemple dans la situation d'échange avec les voisins nord, sud, est et ouest discutée au paragraphe 7.2. Ceci est illustré sur la figure 7.7 où l'on voit un partitionnement cyclique en bande avec 4 processeurs. Le processeur P_i reçoit les colonnes x telles que $x \bmod 4 = i$. On voit aussi dans cet exemple que, dans le cas d'un schéma de communication comme celui de la résolution de l'équation de Laplace, chaque PE doit communiquer l'ensemble de ses données à ses voisins, ce qui est très peu efficace quand on se souvient que l'overhead de la parallélisation va comme le rapport du volume de données communiquées sur le volume des données modifiées.

Par contre, un avantage de la distribution cyclique est qu'elle permet un bon équilibrage de charge dans certaines applications (par exemple le calcul de l'ensemble de Mandelbrot) où la quantité de calcul varie d'un point à l'autre de la structure de données. C'est donc un paritionnement qui peut être très intéressant

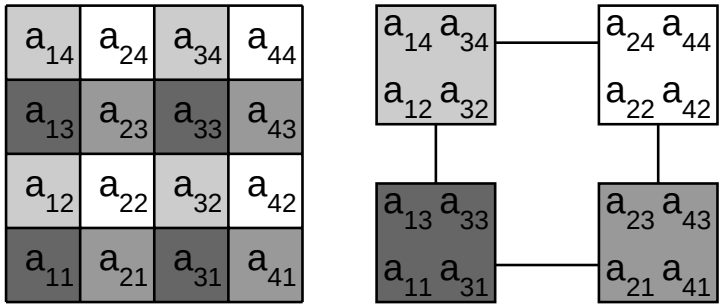

FIGURE 7.6 – *Répartition cyclique (ou modulo) de la structure de données de gauche sur quatre processeurs (à droite) interconnectés selon une grille* 2×2.

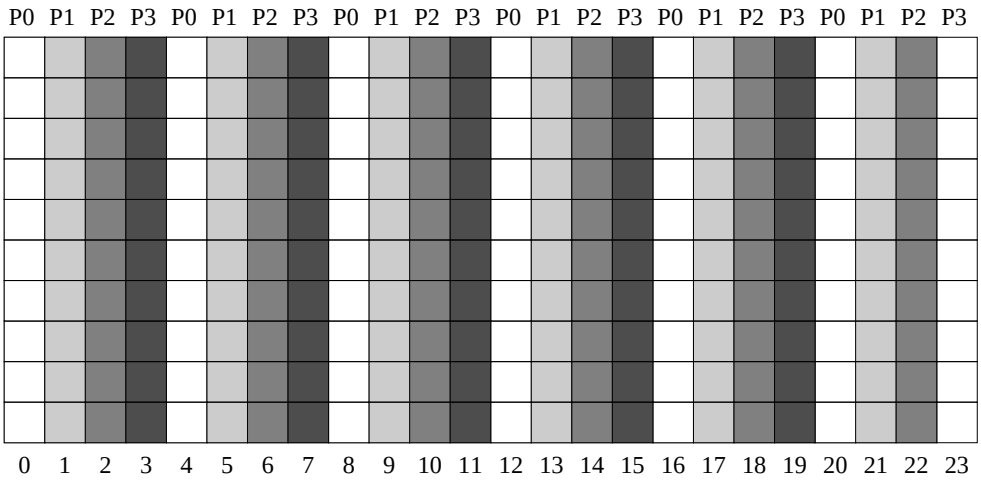

FIGURE 7.7 – Paritionnement cyclic (ou modulo) avec 4 PE, chacun recevant une colonne sur 4. On voit que les frontières cumulées de chaque processeur est très grande ce qui rend ce partitonnement peu adapté à des situations où les processeurs communiquent régulièrement.

pour l'équilibrage de charge dynamique, ainsi que discuté dans la section 6.9.

En fonction des communications souhaitées, la manière de répartir les données sur l'ensemble des processeurs a des conséquences sur l'efficacité du calcul. Il faut trouver un schéma optimum qui tienne compte du réseau à disposition et des communications qui seront requises pour résoudre le problème considéré. Cette tâche se révèle parfois difficile, comme dans le cas des transformées de Fourier rapides où l'interaction entre les données est compliquée. Le stockage de matrices creuses (contenant une majorité d'éléments nuls) est un autre exemple où la distribution des données s'avère difficile. C'est en général le cas de tous les problèmes qui utilisent des structures de données irrégulières.

Pour terminer ce paragraphe, il est important de remarquer que dans les architectures modernes (par exemple les réseaux dynamiques multiétages), la topologie des liens entre processeurs est de plus en plus transparente à l'utilisateur. En fait, du point de vue du temps d'accès à une donnée, ce qui est déterminant c'est de savoir si la donnée est *locale* (c'est à dire dans le processeur même) ou distante (*"remote"*) (c'est à dire dans la mémoire d'un autre processeur). Ceci est principalement dû au fait que le temps de propagation d'un long message est proportionnel à sa longueur plutôt qu'au nombre de noeuds qu'il a à traverser. On pourrait alors penser que la distribution des données est secondaire puisque, en première approximation, chaque processeur est voisin de tous les autres. Cependant, si toutes les données sont trop éparpillées à travers les processeurs, un trafic **non local** important en résultera et pourra produire des congestions du réseau néfastes aux performances de l'application.

7.4 Les primitives SEND et RECEIVE

La forme générale de l'instruction SEND est typiquement

```
SEND(message_out, msg_size, destination, type)
```

où `message_out` est la position mémoire qui contient l'ensemble des données à envoyer, `msg_size` est la taille en octets du message, `destination` est l'adresse du processeur qui recevra le message et `type` est un identificateur permettant de distinguer des messages de types différents (par exemple, on peut alternativement envoyer des entiers ou des flottants et, à la réception, il faut pouvoir retrouver la nature des données reçues).

La structure de la primitive RECEIVE est similaire :

```
RECEIVE(message_in, msg_size, source, type)
```

où `message_in` est cette fois la position mémoire où le message sera stocké après réception et `source` est l'adresse du processeur ayant expédié le message.

En pratique, après un SEND, le système d'exploitation copie le message dans une zone mémoire réservée, le **buffer de communication**, et y inclut une entête

avec des informations de routage. Il déclenche ensuite le départ du message qui, à l'arrivée, est stocké dans le buffer de communication du processeur destinataire. Une variable système est activée pour indiquer l'arrivée du message. La primitive RECEIVE prend le contenu du buffer de communication et le copie dans la zone mémoire choisie par l'utilisateur.

Il n'est pas nécessaire de connaître la source d'un message pour le recevoir (parfois, on ne sait pas d'avance de qui on va recevoir quelque chose). Il est possible d'utiliser des *wildcards* qui permettent que n'importe quel message présent soit reçu, quelle que soit son origine. De même, le type d'un message peut aussi être donné comme wildcard. Lorsque le message est reçu, les variables `source` et `type` sont en général mises à jour par le système et contiennent alors la source réelle du message lu et son type.

Il existe aussi des primitives qui permettent de tester l'arrivée d'un message et d'en connaître ses paramètres, sans pour autant le lire. Cela est très utiles lors d'applications avec des échanges de données non structurées et peu prévisibles.

Finalement, on distingue encore les communications bloquantes et non bloquantes. Les communications sont dites *bloquantes* si la tâche est interrompue jusqu'à ce que le transfert soit terminé. Elles sont *non-bloquantes* si le message est transmis dans le réseau et la tâche poursuivie. Ainsi, un *blocking receive* est une instruction qui arrête l'exécution jusqu'à la réception du message attendu. Le *nonblocking receive* permet de ne pas perdre de temps et d'accomplir une autre tâche en attendant l'arrivée du message. Il est utile lorsque plusieurs messages venant de processeurs différents sont attendus, mais dans un ordre imprévisible. Lorsqu'on utilise une telle primitive, il est nécessaire de pouvoir tester ultérieurement, avec des primitives adéquates, le succès et l'accomplissement de la communication.

Selon les systèmes, le *blocking send* interrompt l'exécution en cours jusqu'à ce que le buffer de communication soit rempli. Cela garantit que la variable dont le contenu est envoyé n'est pas modifiée entre le moment où la primitive SEND est appelée et le moment où le buffer de communication reçoit la valeur.

Les communications non bloquantes sont asynchrones. Par contre, une primitive telle que

```
bsendrecv(msg_out,msg_in,msg_size,source,destination)
```

est une communication bloquante synchrone où le send et le receive sont condensés en une seule instruction. Ce type de SEND-RECEIVE couplé est très utile dans les problèmes où les processeurs échangent des données de façon régulière et répétitive (par exemple, chaque fois qu'un processeur envoie un message à son voisin de droite, il attend aussi le message correspondant envoyé par son voisin de gauche). C'est aussi une opération qui optimise les temps d'exécution et la place mémoire dans les buffers de communication.

Les communications asynchrones (et en particulier le RECEIVE non-bloquant) sont aussi très utiles dans la mesure où elles permettent de faire se chevaucher

communications et calculs. On peut ainsi (si le problème le permet) profiter de faire certains calculs pendant que les données qu'on attend transitent encore dans le réseau.

Messages actifs : Les messages actifs (*active messages*) sont une autre catégorie de messages. Comme leur nom l'indique, ils ne véhiculent pas seulement des données passives mais peuvent aussi déclencher, dans le processeur destinataire, un appel à une fonction et lui en fournir les paramètres adéquats.

Le résultat d'un message actif est d'interrompre le processeur destinataire immédiatement dès réception, afin qu'il exécute la fonction dont il a reçu l'adresse. Lorsque cette dernière est exécutée, le processeur retourne à la tâche interrompue.

7.5 Implémentation

La mise en oeuvre d'un protocole d'échange de message peut différer notablement d'un constructeur à l'autre, ou d'une bibliothèque de primitives à l'autre. Il s'en suit que certaines opérations sont possibles dans un cas mais pas dans l'autre. Par exemple, certaines bibliothèques d'échange de messages permettent de superposer des communications collectives avec des communications point à point alors que d'autres pas. De même certains systèmes ne peuvent recevoir plusieurs messages venant d'un même expéditeur que dans l'ordre dans lequel ils ont été envoyés. La taille des buffers de communication peut aussi varier selon les implémentations et certaines stratégies sont parfois nécessaires pour garantir un transfert important de données.

Dans ce paragraphe, nous présentons le modèle de mise en oeuvre utilisé sur la Connection Machine CM-5. Les principes de bases sont génériques mais pas certains détails. La figure 7.8 illustre la façon dont les différents composants sont organisés. L'accès des processeurs au réseau se fait à travers des chips spécialisés, appelés le *Network Interface* (NI) dans la CM-5 (et adaptateurs sur l'IBM SP2). Leur but est d'offrir aux processeurs une vue unifiée du réseau et, au réseau, une vue unifiée des processeurs. L'avantage est de pouvoir découpler ces deux côtés complémentaires d'une architecture parallèle. Indépendamment, processeurs ou réseaux peuvent évoluer et être améliorés selon les nécessités spécifiques ou les progrès technologiques du moment.

Grâce aux NI, les communications se font sans recours au système d'exploitation et sont directement contrôlées par le programme utilisateur. Le principe de fonctionnement des NI est le suivant : les processeurs voient l'interface comme une collection d'adresses mémoire chacune représentant les destinations possibles du réseau. Les processeurs peuvent lire ou écrire dans ces positions mémoire (buffer de communication), ce qui revient, respectivement, à recevoir ou envoyer un message à travers le réseau. L'envoi proprement dit des messages est généré automatiquement par le NI après écriture, par un processeur, d'un message dans une

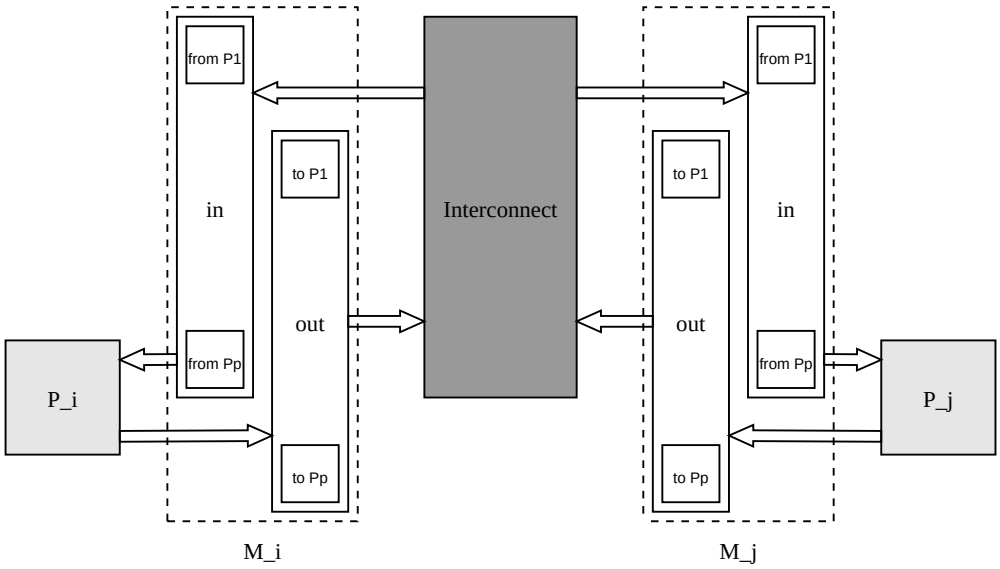

FIGURE 7.8 – *Implémentation schématique du processus d'échange de message entre deux processeurs P_i et P_j qui écrivent et lisent dans des buffers de communications IN ou OUT selon qu'ils envoient ou reçoivent un message. Le réseau d'interconnexion se charge de faire transiter les messages.*

position mémoire correspondante. De cette manière, l'accès au réseau est transparent tout en ne nécessitant pas d'appels système. Les communications sont générées par le code utilisateur, ce qui permet un parallélisme à grain fin avec peu d'overhead. Le NI permet en outre de protéger efficacement les utilisateurs les uns des autres car les adresses correspondant à des processeurs d'une autre partition peuvent être interdites par le système d'exploitation.

En pratique, le NI est constitué de piles FIFO qui sont entrantes (in) ou sortantes (out) selon qu'elles génèrent des messages dans le réseau ou qu'elles en reçoivent. Il existe une telle paire in/out de ces piles pour chaque fonctionnalité du réseau (une pile pour les communications point à point, une pour les opérations de synchronisation, une pour les opérations combinant des données (SCAN), etc). Ces piles sont gérées par le NI selon un jeu de priorités établies. Schématiquement, le mécanisme est le suivant :

— processeur → out-FIFO → réseau ;
— réseau → in-FIFO → processeur.

Ainsi, la vue que l'utilisateur a du réseau est indépendante de sa topologie réelle. La topologie physique du réseau n'est d'ailleurs pas garantie puisque, en cas de panne telles ou telles parties du réseau peuvent être désactivées et se trouver absentes. La topologie physique peut aussi, selon les machines, changer selon la partition qui est allouée à un utilisateur.

Dans la CM-5, un mode timesharing est possible. Pour se faire, le NI garde les données qu'il envoie jusqu'à la fin du transit (un signal de fin de routage est envoyé par le réseau de contrôle). En cas d'interruption de la tranche de temps d'un utilisateur, ces données font partie intégrante de l'état associé à chaque utilisateur et sont sauvegardées pour que le transfert puisse recommencer plus tard.

Contrôle du transfert : cela désigne la manière dont les échanges se font et les techniques permettant de confirmer la bonne réception des messages. En règle générale, les protocoles utilisés sont fiables et on peut compter sur le fait qu'un message envoyé arrive toujours à destination. Cela est assuré par des mécanismes **d'accusé de réception**, transparents à l'utilisateur, qui, si le message n'est pas arrivé, provoque la répétition de l'envoi.

En général, les messages sont asynchrones, ils sont envoyés dès que l'instruction `SEND` (ou son équivalent) est exécutée et que le réseau permet le transfert. Il n'y a pas de concertation préalable avec le destinataire chez lequel les messages sont stockés dans des buffers jusqu'à ce qu'ils soient lus. Si les buffers sont pleins, le message est renvoyé.

On peut aussi envisager des protocoles d'échange par **rendez-vous** où l'expéditeur et le destinataire se mettent d'accord sur un transfert synchrone d'un message. Cependant, cette technique est plus lourde à mettre en oeuvre (nécessitant plusieurs aller-retours de messages pour établir la communication) et moins

adaptée à une architecture naturellement asynchrone.

7.6 MPI et PVM

Parmi les nombreuses bibliothèques d'échange de messages, deux d'entre elles s'imposent actuellement comme des standards. Il s'agit de MPI (Message Passing Interface) et PVM (Parallel Virtual Machine). Toutes deux offrent des primitives similaires à celles décrites ci-dessus, avec bien sûr certaines variations de fonctionnalité, de paramètres d'appel et de noms.

Toutefois, certaines différences méritent d'être soulignées, bien que les avantages et désavantages d'un système par rapport à l'autre soient actuellement en train de se combler.

PVM a été initialement développé pour un ensemble hétérogène de machines UNIX (par exemple des stations de travail de différents types connectées avec des super-ordinateurs, etc) alors que MPI a d'abord été conçu pour une machine MPP homogène (par exemple une SP2 avec plusieurs processeurs identiques). MPI a tout de suite été pensé dans l'esprit d'applications parallèles fortement couplées et offre de nombreuses primitives de communications collectives qui manquaient dans les premières versions de PVM.

PVM est un peu plus lourd à utiliser que MPI, en raison de l'inhomogénéité possible des plate-formes. En effet, PVM doit d'abord coder les messages selon un format propre avant de les envoyer car la représentation interne des nombres n'est pas forcément la même sur toutes les architectures. De plus, différents protocoles sont utilisés selon les plate-formes (TCP/IP pour les stations de travail ou des protocoles natifs dans les MPP).

Par contre, PVM offre des possibilités de gestion de processus que MPI n'a pas. Par exemple, PVM permet de créer et arrêter un processus avec la primitive `pvm_spawn`. L'ensemble d'une application PVM est gérée par des démons qui tournent sur chacune des machines hôtes. PVM peut varier dynamiquement le nombre de machines hôtes utilisées alors que MPI travaille avec un nombre fixe de processeurs.

Notons encore que dans la version de MPI-2.0, des primitives d'écriture et de lecture directes dans la mémoire des autres processeurs seront disponibles. Cette fonctionnalité NUMA s'appelle les communications "one-sided" puisqu'elle n'implique pas un comportement symétrique des deux processeurs impliqués. De plus, MPI-2.0 prévoit aussi des primitives d'entrées-sorties parallèles.

7.7 Avantage et désavantage du modèle

Au contraire des architectures à mémoire partagée qui imposent des contraintes difficiles à réaliser, les architectures à mémoires distribuées sont plus simples de