Programmation des architectures Parallèles

Cours Master 1 informatique
Université de Bordeaux 1

27 avril 2006

Sommaire

Chapitre 1

Introduction

L'objectif de ce cours est de comprendre comment exploiter efficacement les machines parallèles, c'est à dire comprendre le fonctionnement des architectures, étudier différentes techniques permettant d'optimiser l'exploitation des ressources et comprendre les limitations des architectures et des applications.

1.1 Qu'est ce que le parallélisme ?

Le parallélisme est une technique consistant à utiliser plusieurs unités de calculs (processeurs) pour accélérer l'exécution d'une application. Pour cela, on utilise des architectures parallèles. Ces architectures sont de natures différentes. On peut avoir par exemple :

- plusieurs machines en réseaux
- machines multiprocesseurs, à mémoire commune ou distribuée
- processeurs contemporains faisant des calculs en parallèles

1.2 Pourquoi ?

On espère ainsi accélérer les applications. Certains domaines applicatifs ont des besoins qui croissent plus rapidement que la puissance des processeurs. On retiendra par exemple les domaines de la climatologie (Meteo-France), de l'aerospatiale, de l'automobile, et aussi de l'armée (applications militaires).

1.3 Comment s'y prend-t-on alors ?

Le principe de la parallélisation d'une application est expliqué dans le schéma ci-desous. Au départ, on a une application séquentielle que l'on souhaite paralléliser. Une reflexion permet de découper cette application en taches séquentielles. La mise en place d'outils de communication entre ces taches va permettre de les synchroniser, et donc, de les exécuter en parallèle.

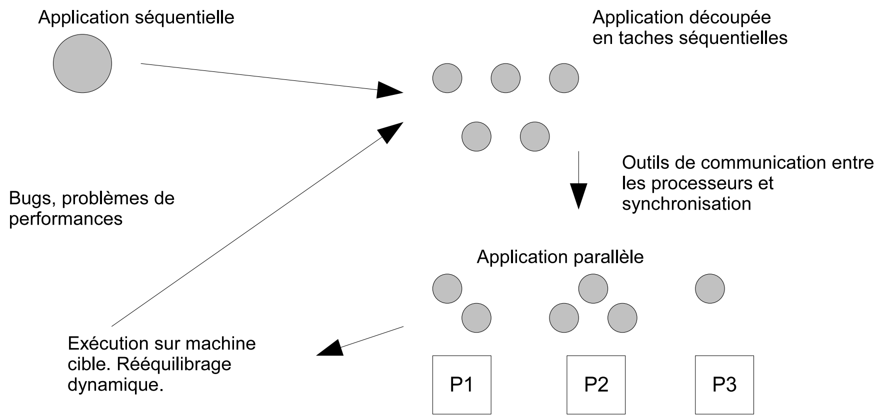

FIG. 1.1 – Parallélisation d'une application

Dans les années 90, le parallélisme est au sommet et on souhaite le rendre accessible aux non experts. On invente alors des langages « parallèles »qui permettent ou facilite l'expression du parallélisme dans les programmes (High Parallel Fortran HPF par exemple). On met de plus au point la parallélisation des boucles.

On remarque cependant une baisse de régime à la fin des années 90, mais le parallélisme revient au gout du jour au niveau de la recherche. Les machines de bureau deviennent parallèles, des grappes de calculs sont mises en place à l'échelle planétaire.

1.4 Aujourd'hui, quelles sont les tendances en matières d'architectures

Pendant longtemps, les supercalculateurs étaient ultra dominant (CRAY, SGI, DEC, IBM). Mais ces solutions étaient extrèmement chères et peu évolutives. Les machines ont donc évolué vers l'utilisation des « technologies répandues »au niveau du processeur.

1.5 Et les machines en réseaux

Les machines en réseaux ont deux avantages majeurs. Elles sont d'une part extensibles et ont un excellent rapport performance prix. Ce type d'architecture, que l'on appelle « clusters »ou « grappes »de PC a surtout été développé par Myricom (start up). Elle a proposé en 1998 une technologie d'interconnexion nommée Myrinet (envoi d'information entre deux PC en moins de $5\mu s$ à un débit de 1 jusqu'à 10 Gbits).

L'importance ici doit être mise sur la tolérance aux pannes. Comment un algorithme continue de fonctionner lorsque un ou plusieurs PC tombent en panne ? Comment réintégrer un PC réparé dans la grappe au cours d'un calcul ? Comment différencier les machines en pannes de celles qui sont en repos ?

Chapitre 2

Architectures parralèles contemporaines

2.1 processeurs

2.1.1 pipeline d'instructions

Un processeur fonctionne au travers d'un pipeline d'instructions. Ce pipeline peut contenir plusieurs instructions qui sont à un niveau différent d'exécution. Dans un premier temps, l'instruction doit être chargée dans le pipeline, puis décodée. Le pipeline doit charger les opérandes, exécuter l'instruction proprement dite et fournir les résultats. Ce principe de fonctionnement est résumé dans le schéma ci-dessous.

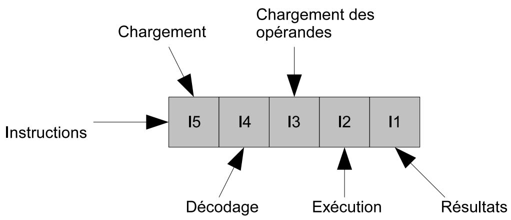

FIG. 2.1 – Pipeline d'instructions

2.1.2 plusieurs pipelines

On peut envisager les processeurs avec plusieurs pipelines d'instructions (les premiers Pentium fonctionnaient ainsi). Les instructions arrivent de manière continue et sont envoyées au premier pipeline d'instructions libre.

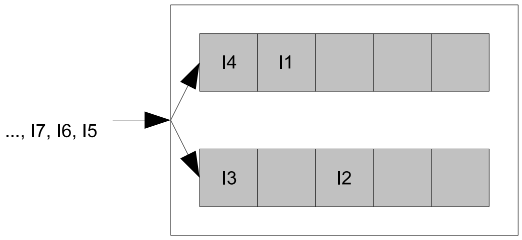

FIG. 2.2 – Processeur avec plusieurs pipelines

Dans le premier pentium, il y a en fait un appareillement des instructions effectué par le compilateur. Ensuite, le processeur réordonne lui même les instructions. Lorsqu'on augmente le nombre de pipelines, on s'écarte encore plus de la puissance théorique du processeur (il y a plus de « trous »dans les pipelines qu'avec un seul car les instructions peuvent attendre des fins d'exécutions d'autres instructions). On appelle ce type de d'architecture des « processeurs superscalaires ».

2.1.3 processeurs multithreads

Ici, l'idée est que ce n'est plus le processeur qui décide du parallélisme. L'application doit utiliser plusieurs flots d'exécutions explicitement (threads) pour exploiter efficacement le processeur. Ainsi, dans un pipeline, les instructions stockées sont associées à un thread.

Comme le processeur utilise des threads, il faut, dans celui ci, ajouter des circuits complémentaires pour pouvoir gérer le contexte de chaque thread (le nombre de threads géré par le matériel est fixé). Ces processeurs sont alors capables de basculer d'un thread à l'autre entre deux cycles.

Le processeur choisit les instructions d'un thread T1 pour les faire entrer dans le pipeline, puis, par exemple, il bascule sur le thread T2 et fait entrer à la suite des instructions de T2 dans le pipeline. Le processeur décide par exemple de changer de thread lorsque celui s'apprête à faire une opération couteuse (accès mémoire, thread bloqué) ou au bout d'un quantum de temps.

Nous citerons ici l'exemple des processeurs TERA, gérant 128 threads « matériels », sans cache. Sur ce processeur, il y a un changement de contexte à chaque instruction (i.e. on change de contexte à chaque insertion d'instruction dans le pipeline). L'absence de cache permet, dans cette architecture un peu particulière, de prédire l'arrivée des données dans le processeur.

En 1995, l'équipe de H. Levy propose la technique du « Simultaneous Multithreading ». Cette technique reprend à la fois l'idée du processeur avec plusieurs pipelines et du processeur multithreadé. Le principe de fonctionnement de ce processeur est résumé dans le schéma ci-dessous.

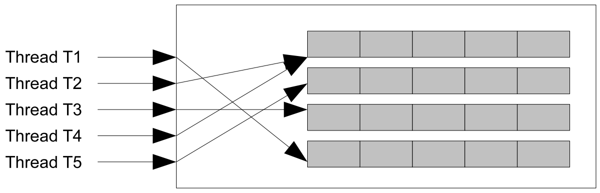

FIG. 2.3 – Processeurs avec plusieurs pipelines multithreadés

Cependant, il faut remarquer ici que favoriser le remplissage des pipelines est contradictoire avec les stratégies d'équités mises en place dans les systèmes d'exploitations. L'évolution de l'architecture précédente va monter au niveau processus. Ainsi, dans les processeurs actuels (processeur « Hyperthreadé »chez Intel) les pipelines permettent d'exécuter des processus qui ont donc accès à une MMU chacun.

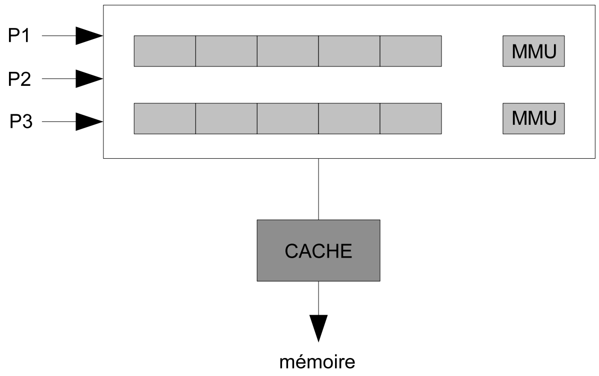

FIG. 2.4 – Processeur HyperThreading

Dans ce type de processeur, les deux pipelines sont donc vus comme deux processeurs. Ils partagent cependant la même mémoire cache. Cela peut poser des problèmes si elle est mal dimensionnée : des données du cache utiles aux processus P1 peuvent être détruites par un processus P2, ayant pour conséquence beaucoup d'accès mémoire. Un autre problème existe : la gène entre les processus.

2.1.4 Et les processeurs vectoriels ?

Dans ce type de processeurs, on est capable d'appliquer une même opération à plusieurs données à la fois (en particulier si les données sont organisées sous forme de tableaux : les données sont envoyées en rafale dans le processeur qui effectue la même opération et retourne les données en rafale.

2.2 Machines multiprocesseurs à mémoire commune

2.2.1 Machines multiprocesseurs symétriques SMP

Le schéma ci-dessous montre le principe d'architecture multiprocesseur symétrique. Nous allons aborder dans cette section quelques problèmes à prendre en compte avec ce type d'architecture.

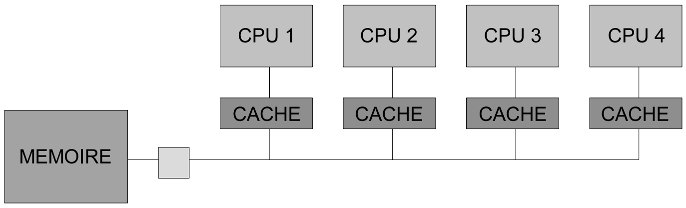

FIG. 2.5 – Machines multiprocesseurs symétriques SMP

Si le CPU 1 change la valeur x=0 en x :=2, la nouvelle valeur est à priori stockée dans le cache de CPU 1. Si CPU 4 fait print(x), il va aller le chercher dans la mémoire et afficher 0. Faudrait-il donc aussi, à chaque modification de la valeur x, mettre à jour la mémoire ?

Supposons que cela soit le cas. Si CPU 4 fait print(x), il récupère la valeur dans la mémoire et affiche correctement 2. De plus, il stocke x=2 dans son cache. Mais si un autre processeur modifie x :=4, le processeur CPU 4, en refaisant ensuite un print(x), va aller chercher la valeur dans son cache et afficher 2 ; ce qui est faux ! Faudrait-il donc de plus que le cache écoute sur le bus pour modifier les données concernées par des modifications effectuées par chaque processeur ?

Il existe en fait deux stratégies pour résoudre ce problème :
- La stratégie consistant à effectuer des écritures à la fois en mémoire et dans le cache s'appelle « write through »
- la stratégie consistant à ne mettre à jour que le cache s'appelle « write back ». En fait, cette stratégie permet d'invalider la zone de cache des autres caches en précisant uniquement que la valeur de x a été modifiée (i.e. il s'agit d'une invalidation des copies de x que peuvent posséder les autres caches).

Il faut toutefois ne pas oublier ici que les objectifs des protocoles de cohérence de cache utilisés doivent utiliser le bus mémoire le moins possible. La technique d'écoute sur le bus s'appelle « Bus Snooping ». Il faut alors se demander quelle vision de la mémoire le programme souhaite-t-il avoir ?

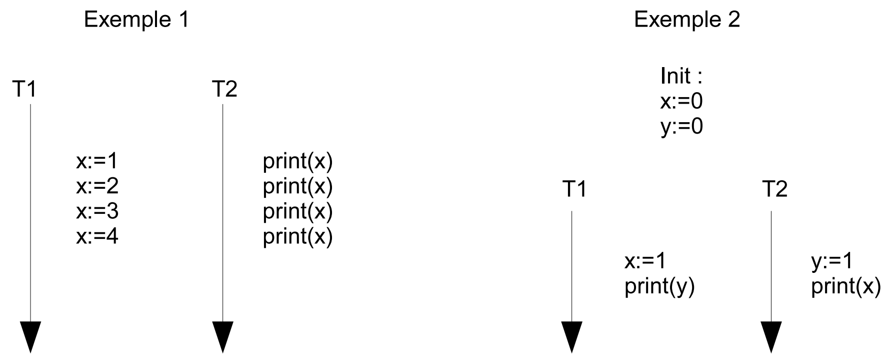

FIG. 2.6 – Quelle vision de la mémoire ?

Dans l'exemple 1 ci dessus, on peut rater des valeurs de x sur plusieurs processus, mais c'était déjà le cas sur une machine monoprocesseur. Dans l'exemple 2, sur une machine monoprocesseur, nous ne serions pas étonné d'obtenir les résultats :
- x=0 et y=1
- x=1 et y=0
- x=1 et y=1

Sur une machine multiprocesseurs, on peut aussi avoir le résultat « x=0 et y=0 »si l'application est mal programmée (problème de consistance du programme).

On aborde ainsi la notion de « consistance séquentielle » : à tous moment, l'état de la mémoire résulte d'un entrelacement des séquences d'instructions exécutées par chaque processeur qui préserve l'ordre de chacune de ces séquences. En particulier, tous les processeurs voient les écritures dans le même ordre (NB : les mutex font aussi de la cohérence mémoire).

2.2.2 Simultaneous Multithreading

Peut-on adapter l'architecture des SMP à des processeurs plus complexes ? On peut par exemple avoir des machines bi-corps (deux processeurs dans un seul processeur) et bi-processeurs.

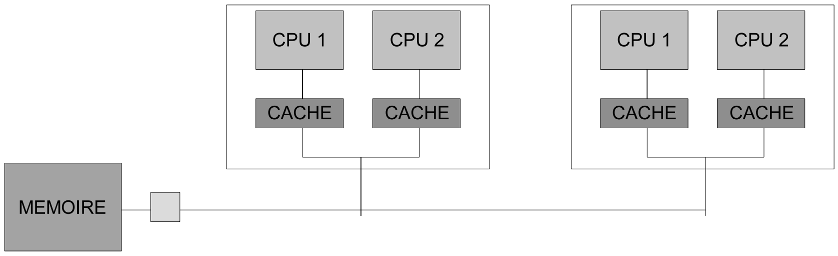

Fig. 2.7 – Machine bi-corps et bi-processeur

Cependant, sur les machines SMP, pour augmenter significativement le nombre de processeurs, il faut renoncer à l'utilisation d'un bus unique pour accéder à la mémoire. Il va y avoir en effet des conflits d'accès au niveau du bus. Par conséquent, l'architecture présentée par le schéma ci-dessous n'est pas idéale :

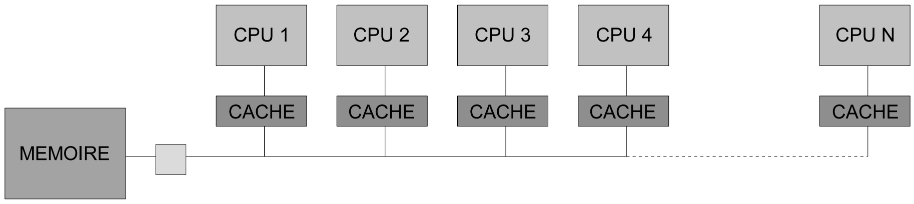

Fig. 2.8 – Architecture SMP non idéale

Il faut alors envisager de nouvelles architectures dérivées. Une première méthode consiste à utiliser un commutateur entre les processeurs et des zones mémoires. Le shéma ci-dessous résume le principe de fonctionnement d'une telle architecture :

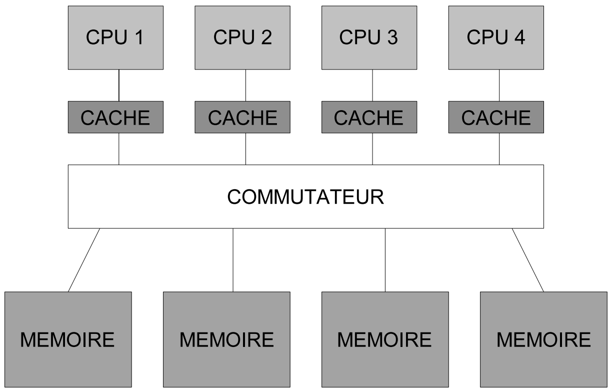

Fig. 2.9 – Commutateur entre processeurs et mémoires

Dans ce type d'architecture, la mémoire est entrelacée : les adresses sont donc réparties de manière cyclique. On se retrouve alors avec un problème de cache. On ne sait pas sur quelle mémoire il peut être associé. On pourrait aussi avoir le cache proche de la mémoire (entre la mémoire et le commutateur). Mais avec cette autre solution il y aurait un problème s'il y a beaucoup de processeurs. En effet, le temps de passage entre le processeur et le cache dans le commutateur est proportionnel au log(nbprocesseur). Si le temps d'accès au cache est trop long, cela contredit l'intérêt du cache d'être rapidement accessible depuis le processeur. Par conséquent, cette architecture est encore incorrecte.

La solution proposée est alors la suivante :

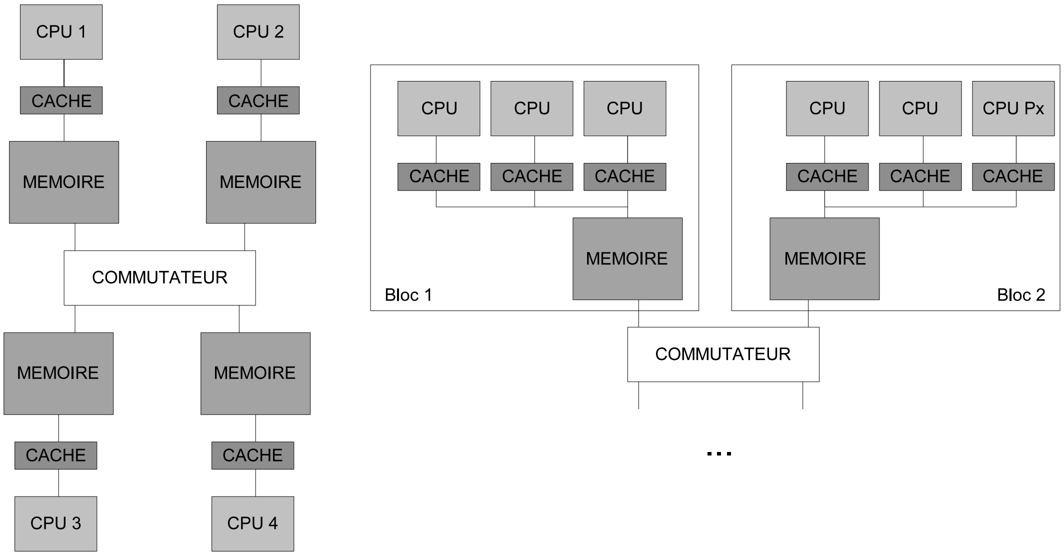

FIG. 2.10 – Machine à accès mémoire non uniforme

Dans les machines à mémoire non uniforme, chaque processeur accède à une mémoire locale rapide et l'accès aux mémoires distantes est plus couteuse. On peut bien sur avoir une architecture ou chaque bloc Processeur + cache + mémoire contient un unique CPU (à gauche sur le schéma ci-dessus) ou avec plusieurs processeurs (SMP, à droite). Dans ce cas, le nombre de processeurs par bloc ne doit pas être trop élevé (pour éviter, comme dans le SMP, des conflits sur le bus).

Dans ce cas, la cohérence des caches est effectuée par snooping comme précédemment. Voyons maintenant le fonctionnement de cette architecture. En supposant que le CPU Px fasse x=3 (x est mémorisé dans la mémoire du bloc 1), il faut prévenir de la modification. En fonction des constructeurs, on trouve différentes techniques :
- NC NUMA : Non coherent : pour avertir que la mémoire a été modifiée et pas le cache
- CC NUMA : les caches sont cohérents mais les accès coutent plus : comment invalider la valeur x d'un bloc 3 ? C'est à dire, comment savoir qui a une copie dans le cache ?

Certaines configurations ajoutent un « Directory »lié sur le bus qui permet de connaître les processeurs qui ont une copie de x.

2.3 Machines à mémoires distribuées

2.3.1 Exemple de machine à mémoire distribuée

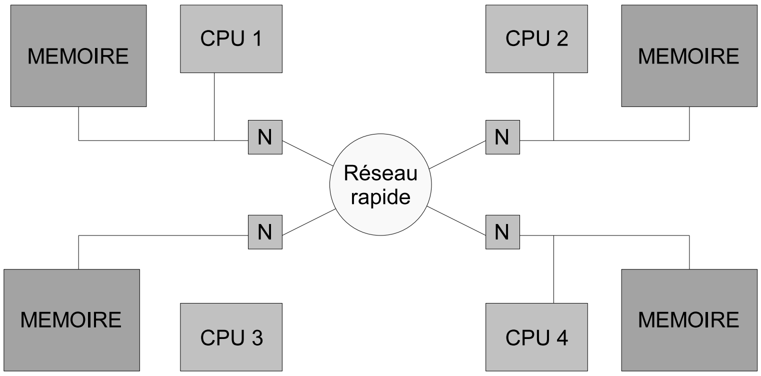

FIG. 2.11 – Machine à mémoires distribuées

Sur ce type de machine, seule la mémoire locale est accessible. Les communications entre processeurs s'effectuent au moyen de primitives spéciales. A l'origine, le Network était inclu au processeur. Mais cela posait un problème d'évolution : les composants réseaux doivent être cadencés à la même vitesse que le processeur. Si on change le CPU, il faut changer le Network. Technologiquement, il est difficile de suivre en parallèle l'évolution des processeurs et des Network. Les machines avec Network relié au processeur ont donc maintenant disparu. Les constructeurs préfèrent s'orienter vers un schéma proche de l'architecture ci-dessus ou les composants peuvent évoluer indépendamment les uns des autres.

2.3.2 Des réseaux internes propriétaires...

Une des contraintes forte des réseaux est qu'ils doivent être performant. Contrairement aux réseaux ouverts, le routage est très simple puisque la topologie est fixée. A ce propos, un mot sur le routage :

- Dans Ethernet : au niveau du routeur, il y a des tampons mémoires permettant de stocker plusieurs trames entrantes, regarder le contenu de la trame pour savoir ou le redistribuer (lien de sortie)
- Dans les machines parallèles : le message n'est pas découpé. Le message prévient les routeurs (commutateurs) de la direction qu'il prend. En fait, le routage est donc fait à la source (par l'émetteur). Le message contient donc un en-tête stockant respectivement les « numéros »de sortie de chaque routeur rencontré. Au passage dans le routeur i, l'octet représentant la sortie pour le ième routeur est consommé par celui ci. A l'arrivée, il ne reste donc que le message.

Regardons maintenant l'exemple du schéma ci dessous pour s'apercevoir d'un problème au niveau des communications.

2.3.3 Exemple de machine à mémoire distribuée

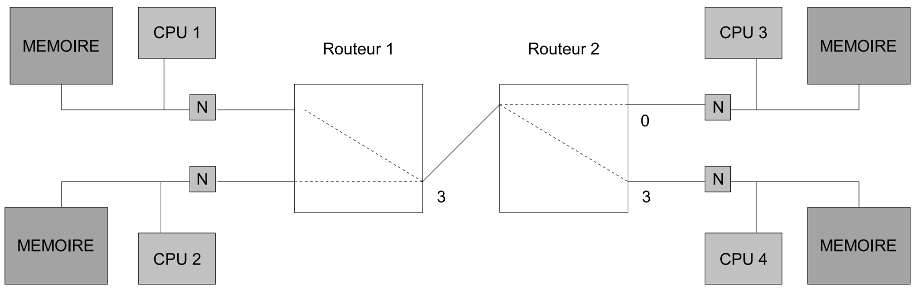

FIG. 2.12 – Problème de performance sur les réseaux

Supposons que les processeurs CPU 1 et CPU 2 veulent envoyer en même temps respectivement à CPU 3 et CPU 4. Si le message de CPU 2 à CPU 4 est long, le message de CPU 1 à CPU 3 est obligé d'attendre. Il y a donc un problème de conflit sur le partage de lien. Une solution (très onéreuse) est de n'utiliser qu'un seul commutateur (routeur) ayant autant d'entrées que de processeurs.

2.3.4 ... aux technologies des réseaux rapides pour PC

On distingue 3 technologies principales :
- Fast Ethernet (100 Mbits/s)
- Myrinet (Myricom) : carte programmable
- SCI : Scalable Coherent Interface

2.3.4.1 Technologie Fast Ethernet

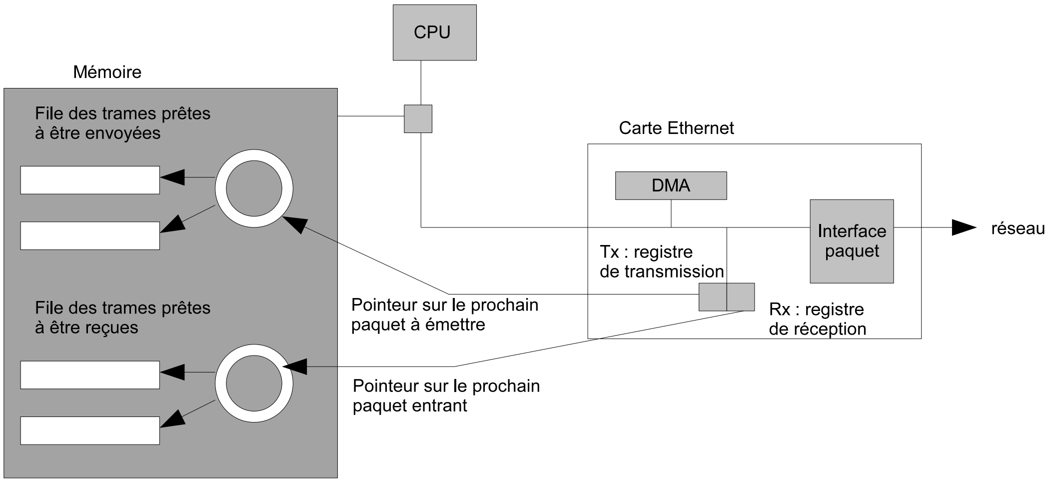

FIG. 2.13 – Fast Ethernet

Le CPU envoi une requête DMA. La carte Ethernet récupère les données dans la mémoire (le processeur n'est alors pas occupé par cette requête).

2.3.4.2 Réseau Myrinet

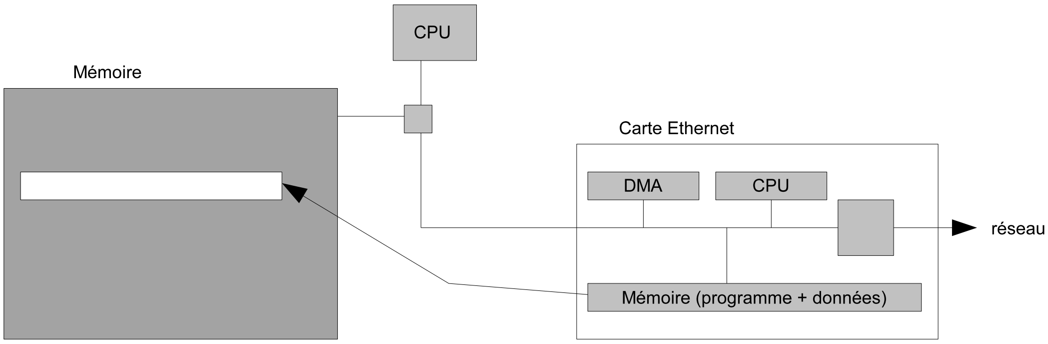

FIG. 2.14 – Réseau Myrinet

La carte d'un réseau Myrinet est programmable (langage C, compilation croisée). Un utilitaire permet de charger dynamiquement le programme (firmware) sur la carte. Le fait que la carte soit programmable a permis de voir naître une multitude de bibliothèques de communication pour Myrinet. La latence (envoi d'une carte à une autre) est de l'ordre de 3μs. Le débit de ces réseaux rapides = 1Go/s (en mémoire = 4Go/s).

2.3.4.3 Norme SCI : Scalable Coherent Interface

Les réseaux de norme SCI sont des réseaux à capacité d'adressage. Leur principe de fonctionnement global est résumé dans le schéma ci dessous.

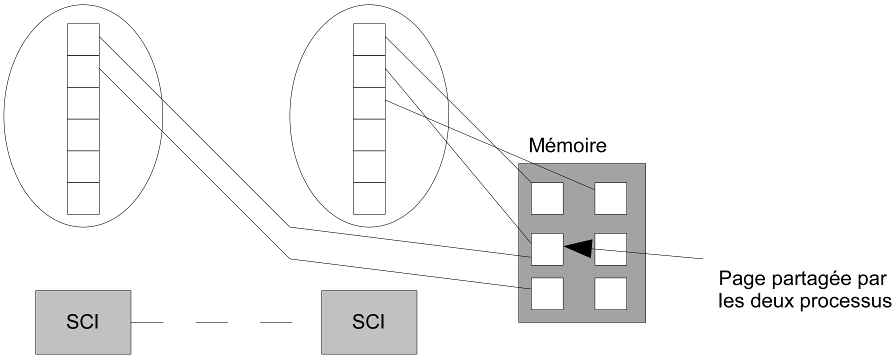

FIG. 2.15 – Norme SCI : fonctionnement global

Le deuxième schéma ci dessous explique dans le détail le principe de fonctionnement de la norme SCI. Nous rappelons ici que la mémoire comme les périphériques sont accessibles via le bus. Pour des adresses entre 0 et MAXRAM, on accède donc à la mémoire, puis entre MAXRAM et MAXBUS on peut accéder au périphériques. La carte SCI est donc accessible (dans l'exemple ci-dessous) entre les adresses 50000 et 60000.

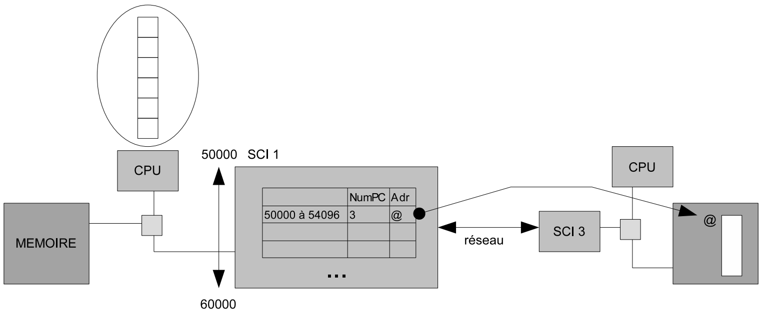

FIG. 2.16 – Norme SCI : détail de fonctionnement

La carte SCI 1 contient les adresses liées à la carte SCI 3. Ainsi, si elle veut accéder à la page 200 en mémoire du PC 3, elle précise qu'elle veut accéder à 3 et à la page 200. La conversion est faite automatiquement par la carte qui accède en fait à la mémoire à l'adresse @+200.

Sur ce type de réseau, le temps de lecture est d'environ 4μs et le temps d'écriture de 2μs. Dans la théorie, la carte SCI permet donc de construire des machines à mémoire partagée assez facilement. Malheureusement, la réalité est plus complexe. Pour des raisons d'efficacité (maximiser le débit), les cartes sont dotées de tampons. De plus, elles utilisent des caches en réception. Il y a alors un compromis entre la cohérence des données et les performances de transferts. Le schéma ci dessous est un exemple de ce problème.

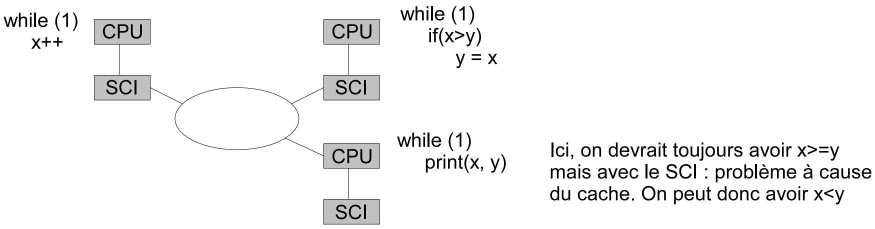

FIG. 2.17 – Exemple de dysfonctionnement avec SCI

Chapitre 3

Exploitation des machines multiprocesseurs

Pour programmer les machines multiprocesseurs, on utilise des processus légers.

3.1 Comment exécuter les taches parallèles d'un programme ?

L'accélération d'un programme se calcule par :

$$SpeedUp_p = \frac{T_{seq}}{T_{surPproc}} = \frac{T_1}{T_p}$$

L'espoir est que $S_p = P$ ou P est le nombre de processeurs. C'est à dire que si l'on a 4 processeurs, on va 4 fois plus vite. Amdahl a établi une loi sur le travail W à effectuer dans une application. On note α la portion de code séquentiel et $(1-\alpha)$ le pourcentage de code en parallèle. :

$$W = \alpha.W + (1-\alpha).W$$

Dans ce cas, on obtient :

$$S_p = \frac{T_1}{T_p} = \frac{W}{\alpha.W + \frac{(1-\alpha).W}{P}} = \frac{P}{1+(P-1)\alpha}$$

Par conséquent, lorsque le nombre de processeurs tend vers l'infini, $S_p = \frac{1}{\alpha}$. Ainsi, si l'on a $\alpha = 10\%$, on ira au plus 10 fois plus vite (donc 10 processeurs suffisent).

Certaines applications sont régulières, c'est à dire que l'on sait abstraire des parties parallèles pour lesquelles les différentes tâches sont de durées égales. D'autres applications en revanche, sont dites irrégulières.

3.2 Implémentation d'un programme multitaches à l'aide de processus légers (threads)

Chaque tache exécutable est prise en charge par un nouveau thread spécialement créé pour l'occasion. Cette méthode a l'avantage d'être facile à programmer. En revanche on peut se retrouver avec des anomalies d'accélération du programme. On peut par exemple assister à une superaccélération due à des effets de cache lorsque le nombre de thread n'est pas trop élevé. Lorsque le nombre de threads devient trop important, les couts de création, gestion et terminaison des threads ralentissent l'exécution en elle même. On observe alors une accélération en fonction du nombre de threads proche de celle du schéma ci-dessous.

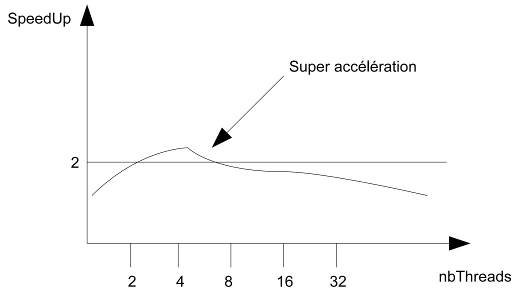

Fig. 3.1 – Speed Up en fonction du nombre de threads

Pour diminuer le coût des opérations de gestion des threads, on peut utiliser un système de threads hybrides. Nous rappelons (voir cours de système) qu'il existe principalement deux types de threads :

- thread noyau : le thread est exécuté au niveau du noyau
- thread utilisateur : le thread est exécuté au niveau utilisateur et les threads sont gérés par un ordonnanceur. D'une manière générale ce système est plus efficace pour la création et la destruction de threads. Mais le noyau n'a pas connaissance des threads et ne peut donc pas les séparer sur plusieurs processeurs

Un système de thread hybride permet (machine Solaris par exemple) de gérer plusieurs threads utilisateurs sur plusieurs processeurs. Des threads noyaux sont en relations avec un ordonnanceur au niveau utilisateur. L'ordonnanceur distribue les différents threads utilisateurs sur les threads noyaux et ceux ci exécutent alors les tâches sur le processeur associé.

Le programme peut tenter de maintenir un degré de parallélisme constant au nombre de processeurs. Quand une tâche disparait, on en recrée une. Un point difficile consiste ici à réagir rapidement lorsqu'un processeur devient inoccupé. Une méthode consiste à créer un pool de processus légers qui vont exécuter une à une les tâches de l'application. Cependant, il faut alors que les tâches aient une durée de vie suffisante pour rendre les opérations de synchronisations efficaces.

Dans le problème du TSP abordé en TD, il y a un compromis à trouver entre la borne inférieure et la borne supérieure pour la durée d'exécution des tâches. On aborde alors le problème de la bonne « granularité »du parallélisme à générer. S'il y a par exemple un grand nombre de processeurs, il peut alors y avoir un problème de synchronisation dans le pool de tâches (devient un goulot d'étranglement). Une solution pour résoudre ce problème consiste à considérer que chaque processeur à un pool de tâches privé. Dans ce cas, il faut faire attention à distribuer uniformément les tâches entre les différents pools lors de leurs générations. Cependant, les tâches peuvent être irrégulières. On ne peut donc pas garantir que tous les processeurs vont toujours travailler (i.e. des déséquilibres de charges vont apparaitre à l'exécution).

La solution consiste à utiliser la méthode de « vol de travail ». Quand le pool de tâches est vide, le processeur va regarder le pool d'un autre processeur pour lui prendre des tâches (ce qui implique que les verrous doivent être conservés). Le choix du pool « victime »peut se faire aléatoirement (pour éviter que cela soit toujours le même pool qui soit choisi, sinon il deviendrait un goulot d'étranglement). Dans le cas particulier ou les tâches d'un pool sont d'égales importances, on peut éviter un recours au verrou, « la plupart du temps »...

3.3 Quels sont les aspects qui peuvent avoir un impact négatif important sur les performances ?

3.3.1 Accès mémoire

Le cache du processeur gère la mémoire par ligne de 64 octets. Si deux valeurs x et y, utilisés par 2 threads différents, sont trop proches dans le cache (donc sur la même ligne), à chaque modification de l'une, autre sera aussi invalidée. C'est ce que l'on appelle « problème de faux partage ». On peut alors modifier le code pour obliger un entier à prendre 64 octets :

```
struct{
        int x;
        char padding[64 - sizeof(int)]
}
```

Il faut toutefois faire attention : sizeof(int) dépend de la machine. L'application peut alors devenir non portable.

3.3.2 Allocation dynamique

Sur les machines NUMA, l'allocation des zones mémoires du programme est particulièrment délicate. Souvent, sur les machines, les pages qui sont allouées dynamiquement (lors du premier accès à la page) près du thread qui en a fait la demande. Par conséquent, il faut souvent s'arranger pour que le premier accès à une donnée soit effectuée par le thread qui l'utilisera majoritairement.

3.4 OpenMP

En 1997, le standard OpenMP est proposé. OpenMP est une série d'annotations utilisées pour étendre les langages existants (Fortran, C, C++). On compile ainsi un code en C « annoté »par une compilation OpenMP qui génère automatiquement un code C utilisant une bibliothèque multithreads. On compile alors le code obtenu normalement. Nous allons maintenant voir quelques exemples de OpenMP

```
void main(){
        F();
#pragma omp parallel
        g();     //effectué en parallèle par plusieurs threads
        h();
}

void g(){
#ifdef _OPENMP //Si OpenMP est utilisé
        int myid = omp-get-thread-num(); //rang ou nom du groupe de threads
        int nb = omp-get-num-threads(); //nombre de threads
#else
        int myid = 0;
        int nb = 0;
#endif
}

void main(){
#pragma omp for schedule (static, 1) //répartit les calculs de boucle
        for (i=0; i < MAX; i++)
                A[i] = ... B[i] ...;
}
```

Dans le dernier exemple ci-dessus, le terme « schedule »permet de préciser la manière dont est réparti le calcul. Ici, le terme « static »permet de répartir les calculs cycliquement. Ainsi, s'il y a 4 threads notés de 0 à 3, le thread 0 gérera les valeurs de boucles i = {0, 4, 8, ...}, le thread 1 les valeurs i = {1, 5, 9, ... } et ainsi de suite.
Si on choisit l'option « schedule (static, 5) »les valeurs de boucles seront distribuées 5 par 5. Le thread 0 traitera les valeurs i de 0 à 4, le thread 1 de 5 à 9, et ainsi de suite cycliquement.
Une dernière option permet d'effectuer un traitement dynamique : « schedule (dynamic) ». Ici, le premier thread qui termine obtient le prochain calcul. Attention, l'option « dynamic »est coûteuse. A n'utiliser qu'avec un grain de parallélisme important.

```
void main(){
        int i;
        int A[]
#pragma omp shared(A) //A est partagée à tous les threads
#pragma omp private (i) //i est privée : chaque thread reçoit une copie
}
```

Pour finir, nous allons voir rapidement comment exécuter une opération séquentielle entre deux tâches parallèles. Par exemple, considérons le code ci-dessous :

```
#pragma omp for
        for (i = 0; i < MAX; i++)
                A()

        sum+= F() //on veut que cette instruction soit en séquentiel

#pragma omp for
        for (j = 0; j < MAX; j++)
                B()
```

Cette solution n'est cependant pas optimale. En effet, lors du premier « #pragma omp for », il va y avoir création de threads. Ces threads vont ensuite être détruits, et après l'exécution de la ligne séquentielle, de nouveaux threads vont être créés. Une meilleure méthode (voir ci-dessous) permet d'éviter de créer des threads (coût de création trop grand).

```
#pragma omp for
        for (i = 0; i < MAX; i++)
                A()
        #pragma omp single
        sum+= F()
        for (j = 0; j < MAX; j++)
                B()
```

Ici, un pool de threads est créé au démarrage. Les threads dorment en attendant la prochaine section parallèle. La ligne de code à exécuter en séquentiel ne sera en fait exécutée que par un seul thread.

3.5 Le langage Cilk (MIT)

voir dept-info.labri.fr/~namyst/pap/. Ce langage permet de programmer des Fork-Join récursifs.

Chapitre 4

Communication par envoi de messages sur réseaux rapides

4.1 Vocabulaire et définitions

Les communications par envoi de messages s'appuient typiquement sur les opérations :

- ```
  send(int numDestinataire, char* donnees, int taille)
  ```
- ```
  recv(char* emplacementDAccueil, int taille)
  ```

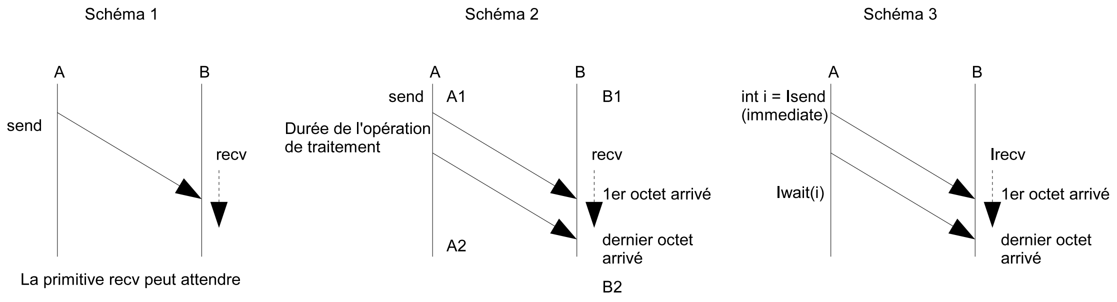

FIG. 4.1 – Primitives de communications

Il existe des primitives synchrones et des primitives asynchrones. Lorsque la communication est synchrone, l'opération « send »ne se termine que lorsque l'opération « recv »a commencée de l'autre coté. Ainsi, cela permet de faire une barrière. Sur le schéma 2 ci dessus, l'opération B2 ne commencera pas avant la fin de A1 et l'opération A2 ne commencera pas avant la fin de B1.

Par la suite, on se placera dans le cas asynchrone. Lors d'une communication non bloquante en émission, l'émetteur reprend son exécution même si le message n'est pas envoyé complètement à B. On utilise alors la primitive « Isend », qui libère immédiatement même si les données ne sont pas encore envoyées. On peut alors améliorer l'exécution, mais il faut s'assurer que l'on ne modifie pas les données à envoyer. Pour attendre l'envoi effectif du message (si l'on veut le modifier), on utilise alors la primitive bloquante « Iwait() ». On fait de même côté récepteur. L'objectif de ces primitives est donc de recouvrir les communications par du calcul.

4.2 Que cherche-t-on à faire ?

On souhaite les propriétés suivantes :
- transmissions fiables : messages non altérés - réseaux Myrinet par exemple, ni perdus. On peut éviter de perdre des messages par contrôle de flux. Cela évite de noyer une machine en envoyant des messages qui ne peuvent être stockés.
- performances : latence ? débit ? recouvrement maximal ?
- portabilité

La latence correspond au temps de transmission (aller-retour) d'un entier. On peut la calculer par exemple en effectuant à la suite plusieurs aller-retours entre A et B et en divisant le temps total par le nombre de communications (pour une meilleure approximation). Pour calculer le débit (nombre d'octets par seconde), on fait un ping pong permettant de tester le débit de manière bidirectionnelle (schéma 1 ci dessous)

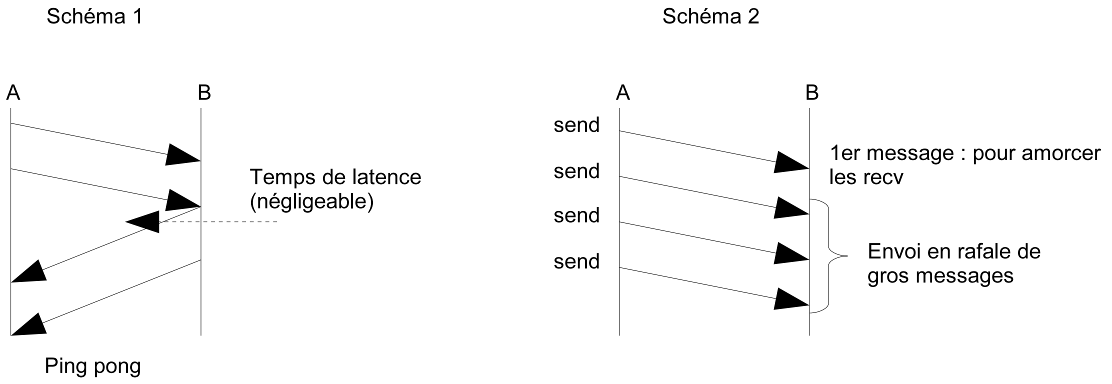

Fig. 4.2 – Déterminer le débit

Une autre méthode (schéma 2 ci dessus) consiste à envoyer en rafale un même gros message pour en déduire le débit en fonction de la taille du message (et du nombre fois ou le message est envoyé, sachant que le premier envoi n'est pas compté : il sert à amorcer la synchronisation de l'émetteur et du récepteur au niveau des primitives send et recv). La mesure peut se faire aussi bien sur l'émetteur que sur le récepteur.

4.2.1 Latence

Comment préserver une latence aussi petite que celle offerte par le matériel ? Le mode TCP/IP propose un contrôle d'erreurs et des entêtes et recopie de données. Ces traitements longs et superflus sont donc inutiles. De même, le contrôle de congestion proposé par TCP/IP peut être évité. Ici, on utilise alors uniquement des bibliothèques de communication (send et recv par exemple). Un autre problème de TCP est qu'il est situé au niveau du noyau. Depuis une application, il y a alors un appel système (1μs). Cela coûte donc plus en temps qu'un appel de fonction. Pour un réseau rapide, il faut donc peut être renoncer à faire des appels systèmes. L'application doit pouvoir parler directement à une carte réseau. Dans ce cas :
- les performances sont meilleures
- mais il faut s'assurer qu'il n'y a qu'un seul processus qui peut parler à la carte en même temps. Le système n'est plus là pour jouer le rôle d'arbitre. Ce rôle doit être effectué par l'application. De manière générale, ce n'est pas un gros problème.
- il y a un problème de sécurité : la question est de savoir si l'on peut accepter que les utilisateurs soient en root sur les machines. Dans le cadre de ce cours, on considère que cela

n'est pas génant.

On veut alors faire 2 choses :
- court-circuiter le noyau. Mais comment faire pour récupérer l'adresse physique du segment de message à envoyer ? On est obligé la première fois de faire un appel système pour récupérer les adresses physiques (qui seront stockées dans un cache en espace utilisateur).
- ne pas introduire de recopie.

Il faut de plus surveiller la mémoire. En supposant que la mémoire utilisée est très importante, il est possible que celle ci « swappe »pour transférer des données depuis la mémoire vers un disque dur et libérer de la place dans la RAM. En fait, il faut donc empécher la mémoire de swapper car la carte réseau n'a aucun moyen de voir que le contenu de la « zone mémoire »à envoyer a changé. Dans la mémoire, il faut alors ajouter et gérer un bit « don't swap ». Ainsi, lorsqu'on envoi un message, il faut mettre à jour (valider) le bit « don't swap » ; et pour cela, il faut faire un appel système...

On se rend donc compte que si l'on veut contourner la gestion noyau, et donc des appels systèmes qu'elle engendre, on est de toute façon obligé d'utiliser des appels systèmes... il y a donc un problème à ce niveau là !

4.2.2 Latence courte

La latence courte est obtenue par des communications en espace utilisateurs en court-circuitant le noyau. Le principe consiste en une utilisation judicieuse des modes PIO et DMA. On peut observer sur le schéma ci dessous le fonctionnement général en mode PIO et DMA et les temps d'émissions en fonction de la taille des messages.

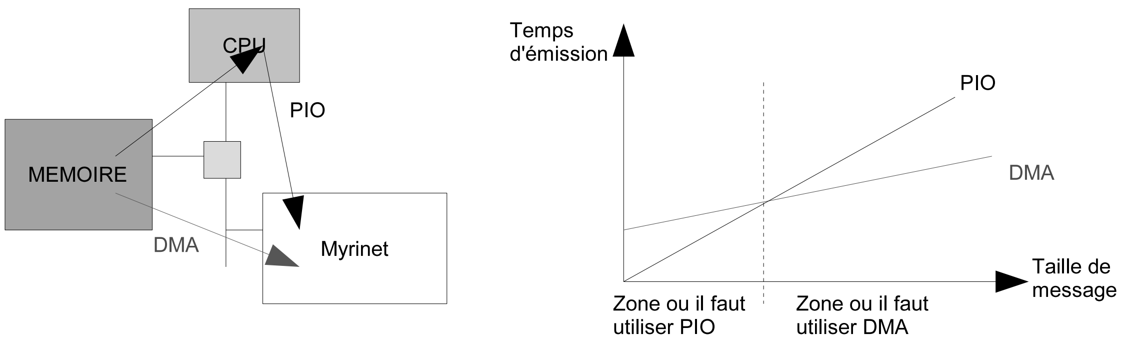

Fig. 4.3 – Mode PIO et DMA

4.2.3 Que ce passe-t-il en réception ?

En réception, que ce passe-t-il, dans le cas précédent, lorsque la primitive « recv »n'a pas été appelée ? On peut utiliser la technique de rendez-vous expliquée sur le schéma 1 ci dessous.

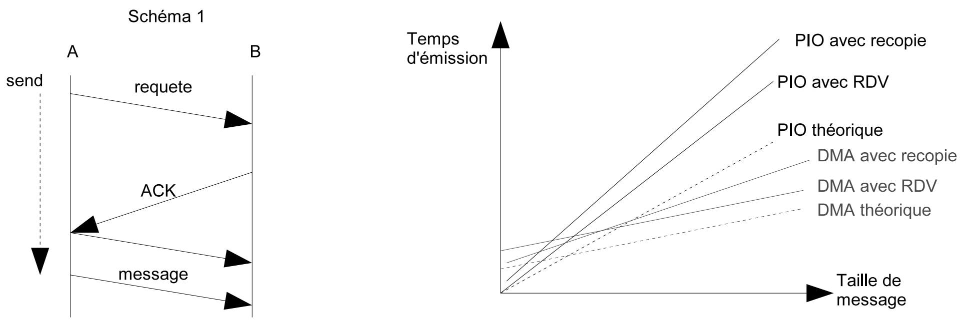

FIG. 4.4 – Mode PIO et DMA, technique RDV ou recopie

Il faut alors trouver un bon compromis entre un aller-retour supplémentaire (technique du rendez-vous) et une recopie. Le schéma 2 ci dessus montre que lorsque la taille du message devient importante, la recopie devient d'une manière générale plus coûteuse que la technique de rendez-vous.

4.2.4 Exemple de la bibliothèque BIP : Basic Interface for Parallelism

Cette bibliothèque comprend deux primitices principales :
- bip_send(int host, char* data, unsigned len, int tag)
- bip_receive(char* data, unsigned len, int tag)

Le tag sert à spécifier le mode de transfert (RDV par exemple). La bibliothèque BIP est toutefois très dégarnie. Les messages courts sont recopiés dans des files de taille fixe sans aucun contrôle de débordement. Les gros messages sont jetés à l'arrivée si le récepteur n'est pas prêt. Cette bibliothèque propose donc des primitives de bases que l'on peut encadrer selon le type de traitement souhaité. Par exemple :

```
void send (char* data, unsigned len){
        if (len > BIP_MAX_SMALL)
                //rendez-vous
        else{
                bip_send(...)
                ...
        }
}
```

Dans le cas d'un rendez vous, on pourra donc avoir, côté émetteur et côté récepteur les codes suivants :

```
emetteur (...){
        bip_send(REQ); //demande d'envoi
        bip_recv(ACK); //réception d'ACK
        bip_send(data);
}

recepteur (...){
        bip_recv(REQ); //réception demande d'envoi
        send(ACK); //envoi d'ACK pour démarrer envoi
        bip_irecv(data) //réception non bloquante des données
}
```

Pour le mode PIO (cas « else »précédent), il faut contrôler le nombre de places disponibles en réception. On peut pour cela effectuer un contrôle de flux à base de crédits. A chaque lecture, on crédite le nombre de messages lus. Mais il s'agit là d'une solution extrême. En fait, on peut prétendre que l'émetteur connait le nombre de messages MAX que le récepteur peut accepter. A chaque envoi, il débite le nombre de places disponibles. La difficulté consiste alors à ce que le récepteur envoi un acquittement de messages suffisamment tôt pour que l'émetteur le recoive au moment ou il envoie son dernier message.

4.2.5 Virtual Interface Architecture (VIA)

Problème : Ces interfaces sont spécifiques à des technologies réseaux. Pour des raisons de portabilité, les applications ne peuvent pas s'appuyer directement dessus.

Solution : Une solution consiste à utiliser la technologie MPI (Message Passing Interface - 1997, standard de MPI Forum).
MPI_send(char* message, str len(message) + 1, MPI_CHAR, des, tag, MPI_COMM_WORLD)

Avec :
- MPI_CHAR : type. Nécessaire car MPI fonctionne sur des machines hétérogènes
- MPI_COMM_WORLD : « communicateur » qui est un ensemble de processus qui disposent d'un canal privé

On dénotera plusieurs sémantiques possibles.
- MPI_Send : potentiellement bloquant
- MPI_bsend : appel qui utilise des tampons pour éviter un blocage
- MPI_nsend : ready send = « je promets que le récepteur est prêt ». Cela permet d'éviter un rendez vous interne

Lorsqu'on envoie un message via MPI_Isend le processeur peut continuer à effectuer des calculs. Il existe alors, comme vu précédemment, la méthode MPI_Itest pour attendre un acquittement du récepteur. Celui ci informe alors que le récepteur est effectivement près à recevoir un nouveau message. Le message est alors effectivement envoyé (jusqu'alors, il avait été conservé en attente sur l'émetteur). Cette solution implique toutefois d'ajouter un thread interne à MPI pour gérer les envois différés.

Une autre solution consiste alors à descendre une partie de MPI au niveau d'une carte réseau programmable. De cette façon, les acquittements de rendez-vous peuvent être pris en charge par le matériel.

Chapitre 5

Communications par Mémoire virtuellement partagée (MVP)

Une mémoire virtuellement partagée (MVP) (ou DSM : Distributed Shared Memory) est un système qui donne l'illusion d'une mémoire partagée au dessus d'une architecture à mémoire distribuée. L'implémentation d'une telle architecture peut être réalisée aussi bien au niveau noyau qu'au niveau utilisateur (dans une bibliothèque).

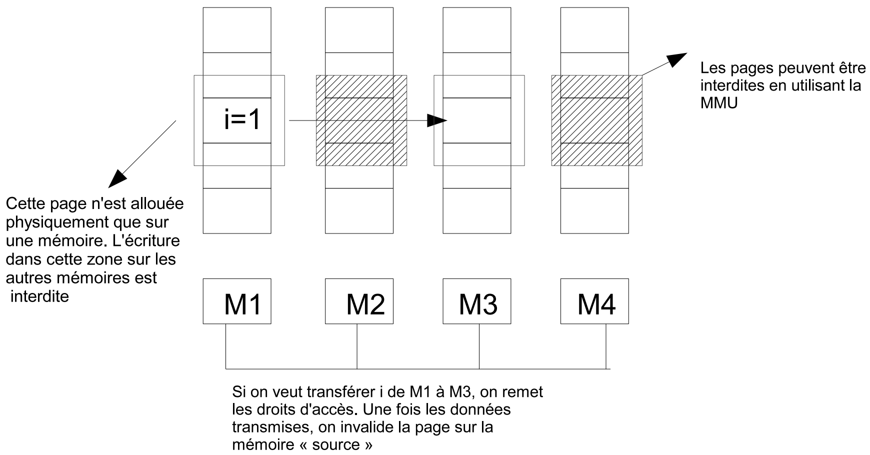

Fig. 5.1 – DSM à base de pages

A chaque accès à une page que le processus ne possède pas, on est obligé d'aller la chercher sur le réseau. Par peur des problèmes de cohérence, on n'authorise donc qu'un seul exemplaire de chaque page. Comment retrouver le propriétaire de la page cherchée ? Diffuser la question est trop couteux en nombre de messages. Une méthode consiste à centraliser l'information sur une machine connue de toute les autres. Chaque fois qu'une machine Mi cherche une page, elle demande à la machine M0. Celle-ci connait la machine cherchée Mj et demande à celle-ci de transférer la page à Mi. Mj transmet alors directement la page à Mi. M0 met à jour ses informations : maintenant, c'est Mi qui possède la bonne version de la page concernée.

Pour désengorger le serveur d'information M0. On peut utiliser une méthode alternative : chaque machine gère un sous ensemble des pages (calculée en fonction du nombre de machines par exemple, pour une distribution cyclique de la gestion des pages mémoire).

On peut de plus utiliser la notion de propriétaire probable : Mj peut stocker que le propriétaire probable de la page qu'il vient d'envoyer à Mi est Mi lui-même.

Pour augmenter les performances des lectures, on peut toutefois autoriser plusieurs copies d'une même page tant que personne ne la modifie. Lors d'une demande de pages, les deux processus doivent restreindre leurs droits en lecture seule.

5.1 Cours 9 : 04/04/2006 MANQUANT

5.2 Cours 9 : 04/04/2006 MANQUANT

5.3 Protocoles de cohérence des données

5.3.1 Protocole de consistance séquentielle

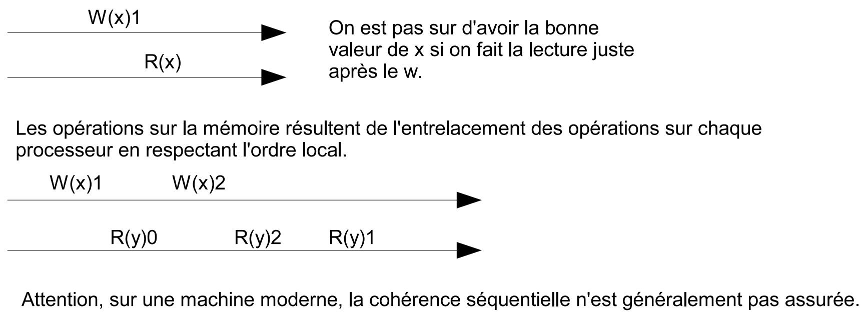

Fig. 5.2 – Protocole de consistance séquentielle

5.3.2 Protocole de consistance causale

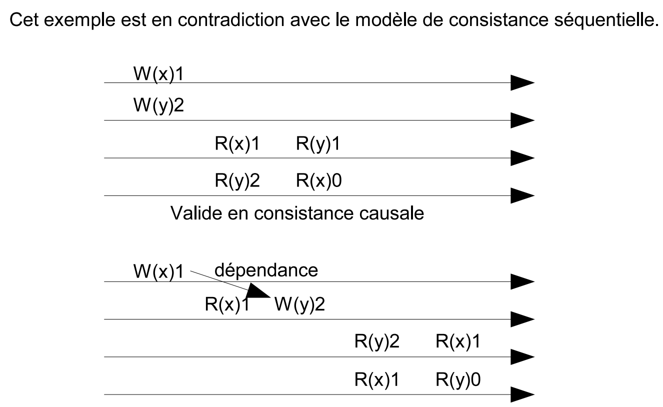

Fig. 5.3 – Protocole de consistance causale

5.3.3 Protocole de consistance relachée

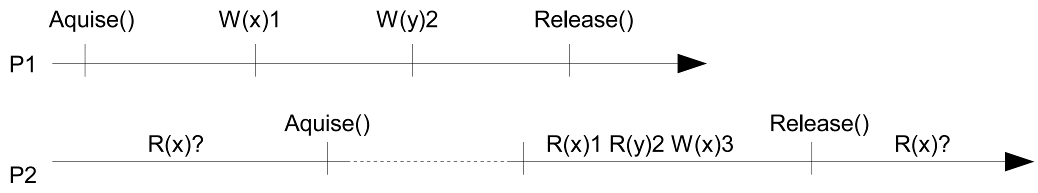

FIG. 5.4 – Protocole de consistance relachée

Lorsque P2 fait Aquise(), il faut attendre que P1 fasse Release() pour pouvoir consulter la bonne valeur de x, y et z. Même en dehors du Release et après ce dernier, on est pas sur de la valeur de x. Un autre processus peut l'avoir changé entre temps.

Ceci induit le problème des sections critiques. En effet, comment entrer en section critique sur une machine distribuée ? Une méthode consiste à utiliser un thread sur P0 par exemple. Ce thread permet de gérer une liste de priorité pour savoir qui doit entrer en section critique. Cette méthode apporte un autre problème : P0 devient un goulot d'étranglement (tous les autres processus doivent demander une autorisation à P0, et ce dernier doit acquitter la requête).

De plus, P0 est le seul a être au courant de ses modifications. Comment les autres processus peuvent détecter les modifications de P0 ? A quel moyen envoyer ces informations ? Une méthode consiste à tout passer en « read only ». A chaque écriture, dans le traitement du signal, on marque dans un tableau que la page a été modifiée et on remet les droits en écritures. Au moment du « release », il reste juste à parcourir le tableau. Pour cela, on distingue deux principales stratégies :

- Stratégie « Eager » Release Consistancy : lors du release, on diffuse les pages modiées. Cette stratégie impose un problème majeur : la diffusion à tous les processus implique que dans le pire des cas il peut y avoir plusieurs sections « Aquise() - Release » sur le même processeur.
- Stratégie « Lazy » Release Consistency : lors du Aquise() ; on va chercher les pages. Le défaut de cette stratégie est qu'il y a une attente induite par l'attente de la réponse.

Il faut de plus, avec la deuxième stratégie, faire attention au point suivant : Il faut compter les pages que l'on a reçues lors du Aquise(). Pour ne pas renvoyer des pages qui n'ont pas été modifiées, on stocke les informations dans un tableau, avec toutes les versions de chaque page, indicé par les pages. Cependant, on distingue un autre problème : envoyer l'intégralité des pages est coûteux. Une idée consiste alors à n'envoyer que les modifications au lieu des pages elles-mêmes.

Lors du traitement d'un signal (écriture lors d'un Aquise()), on effectue un double de la page (twin) que l'on stocke dans une mémoire privée. Pour calculer la différence entre deux pages, en général, on calcule sur la base de comparaisons sur les mots mémoires. La techniques des écrivains multiples (employée avec certains protocoles de consistance relachées) permet d'effectuer un ping pong de la page entre plusieurs mémoires virtuellement partagées.

5.4 Mémoire virtuellement partagées à base d'objets

Dans ce type de mémoire, les pages ont une granularité fixe et n'utilise aucune information du compilateur.

```
class A {
        champs
        m1();
        m2();
        m3();
};
```

Les méthodes sont des opérations atomiques sur l'obnjet du point de vue de la DSM sous jacente. Un exemple de ce type de mémoire : ORCA : trie les méthodes suivant qu'elles font une modification ou une lecture. Une méthode est considérée comme « lecture » si elle n'effectue aucune affectation ou ne fait pas appel à une méthode qui modifie l'objet.

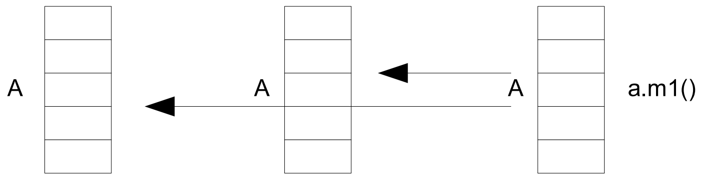

Fig. 5.5 – Exécution de m1()

Un premier avantage : la granularité peut être adaptée. Une deuxième avantage : il n'y a pas de surcoût système pour la gestion des lectures et écritures. Le troisième avantage est qu'on peut mettre à jour les copies en exécutant la même méthode sur toutes les copies. Attention toutefois : si A est un grand objet et m1 est une méthode très courte, cela devient couteux d'invalider les pages. De plus, la granularité adaptée de marche pas avec toutes les méthodes (problème notamment avec celles qui ont des effets de bords).

Chapitre 6

Algorithmique distribuée

L'algorithmique distribuée implique l'utilisation d'horloges logiques. En effet, sur une architecture distribuée, l'accès à une notion de temps absolu est impossible. Une horloge logique est propre à une machine et évolue presque indépendamment. De temps en temps, certains algorithmes utilisent des resynchronisations d'horloges.

6.1 Problème de la diffusion causale

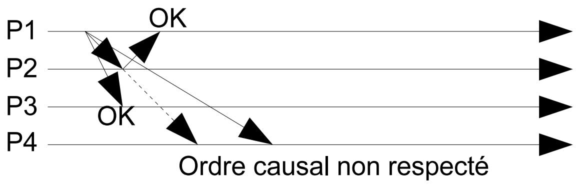

Fig. 6.1 – Problème de l'ordre causal

Sur le schéma ci-dessus, l'ordre causal n'a pas été respecté. Un message envoyé par P2 après réception du message de P1 arrive en P4 avant que ce dernier n'ait lui même reçu le message en provenance de P2. Le problème consiste donc ici à détecter que le message est arrivé trop tôt. Une méthode consiste à faire en sorte que chaque processus Pi possède une horloge logique hi initialisée à 0. A chaque émission de message, l'horloge est incrémentée. A chaque réception de message, l'horloge logique est également incrémentée.

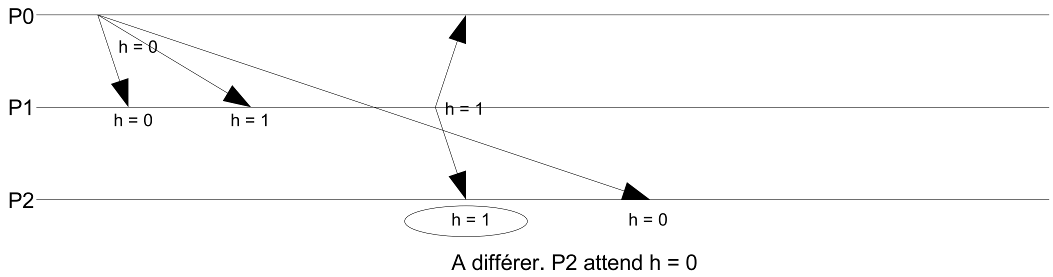

Fig. 6.2 – Détection de message par valeur d'horloge

Sur la figure ci-dessus, P0 envoie un message à P1 et P2 avec l'horloge h = 0. Il renvoie ensuite un message uniquement à P1 avec l'horloge à h = 1. P1 envoie un message à P0 et P2 avec le top d'horloge à 1. Le message h = 1 est reçu par P2 avant celui de h = 0 de P0. La valeur de h attendue par P2 est h = 0. Par conséquent, le message h = 1 est différé.

Une autre méthode consiste à conserver localement les informations. Chaque processus essaie de maintenir un tableau de ce qu'il connaît des autres horloges. Par exemple, le processus i contient un tableau V, tel que V[j] est la dernière valeur connue de l'horloge de Pj.

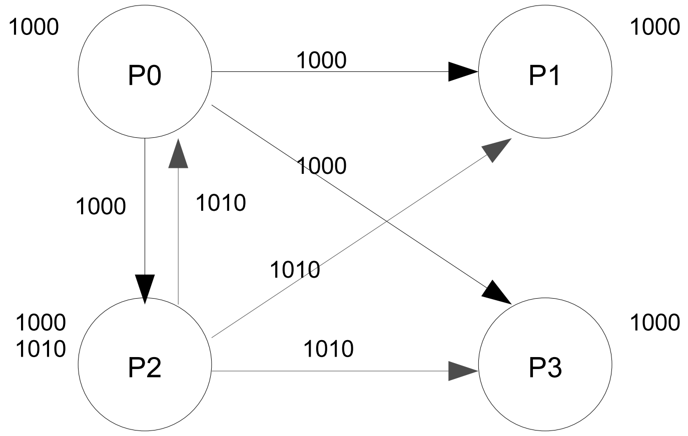

FIG. 6.3 – Utilisation d'un tableau de valeurs connues des horloges

Le schéma ci-dessus montre un exemple de fonctionnement d'une architecture distribuée avec utilisation de tableaux. Chaque processus possède donc un tableau qui conserve la dernière valeur connue des horloges des autres processus. Dans un premier temps, P0 envoie avec son message de diffusion le contenu de son tableau. Les autres processus peuvent alors mettre à jour leur valeur locale de P0. Ensuite, P2 fait lui aussi une diffusion. Le principe de fonctionnement est identique.

Imaginons que P3 reçoit en premier le message de P2 (il devrait recevoir en premier P0). Dans ce cas, il reçoit de P2 l'information que P0 est incrémenté. Il n'est donc pas renseigné par P0 lui-même de cette information. P3 déduit alors qu'il est en retard. Il n'a pas reçu la première diffusion. Il sait alors qu'il va recevoir un message de P0. Ainsi, le message 1010 est différé. On attend celui de P0. Une fois ce dernier reçu, on peut libérer le message 1010.

Pour résumer, lorsqu'un message m arrive sur le processus Pj (envoyé depuis Pi), le message peut être délivré immédiatement si :

- $V_m[i] = V_j[i] + 1$
- $V_m[k] =< V_j[k], \forall k \neq 1$

6.2 Problème de l'exclusion mutuelle

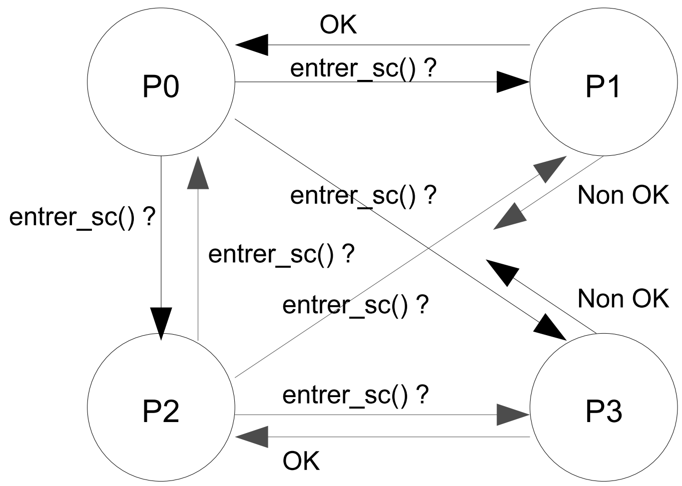

FIG. 6.4 – Problème de l'exclusion mutuelle

Considérons le schéma ci-desus. P0 demande à rentrer en section critique. Il attend l'affirmation des autres. Un problème se pose ici si deux processus (P0 et P2 dans notre exemple) demandent presque en même temps l'entrée en section critique. Il faut alors trouver un algorithme efficace et équitable pour distribuer les sections critiques. Sans cela, la situation du schéma ci-dessus est bloquante. En effet, P1 va autoriser P0 à entrer en section critique. P3 autorise P2. Mais P1 interdit P2 et P3 interdit P0. Ainsi, ni P0, ni P2 vont être autorisé à entrer en section critique.

Il faut alors des critères de sélection pour déterminer qui, de P0 ou P2 va effectivement entrer en section critique. Pour cela, on peut considérer que l'évènement (Pi, hi) est antérieur à Pj, hj) si :
- hi < hj
- hi = hj et i < j

Chaque processus conserve dans un tableau le dernier message reçu depuis chacun des autres processus. Ainsi, le processus Pi conserve dans un TABi les valeurs (j, hj, TypeDeMessage). L'entrée en section critique se demande via une requète REG. La sortie de section se fait par une libération (release) REL. Chaque processus utilise un thread de gestion des demandes d'entrées en section critique. Par conséquent, sur Pi, on a :

```
entrer_sc(){
        tab[i] = (REQ, hi, i)
        Diffusion (REQ, hi, i)
        hi++
        Attendre, pour tout j différent de i :
                estampille (tab[i]) < estampile (tab[j])
}

sortir_sc(){
        tab[i] = (REL, hi, i)
        Diffusion(REL, hi, i)
}
```

De plus, à la réception d'un message provenant de j :

```
Si REQ :
        hi = max(hi, hj) + 1
        tab[j] = (REQ, hj, j)
        envoyer (ACK, hi, i) à j
Si REL :
        hi = max(hi, hj) + 1
        tab[j] = (REL, hj, j)
Si ACK :
        hi = max(hi, hj) + 1
        Si Type(tab[j] \ne REQ):
                tab[i] = (ACK, hj, j)
```

La version actuelle de cet algorithme distribué implique qu'il n'y a pas plus de 3(n-1) messages par entrée en section critique (valeur dans le pire des cas).

Table des figures