Chapitre 8

Modèle de Programmation à Mémoire Partagée

Dans un modèle de programmation à mémoire partagée, tous les processeurs ont un accès uniforme à la mémoire et peuvent lire ou écrire sur n'importe quelle variable commune ou globale. C'est ainsi que les processeurs communiquent, un peu comme si un groupe de personnes collaborait en modifiant des informations sur un tableau noir visible de tous.

Le fait d'avoir une mémoire unique allège de façon significative la programmation car il n'est pas nécessaire de se soucier de l'emplacement d'une donnée ni de gérer son mouvement entre des processeurs différents.

Par contre, l'accès simultané par plusieurs processeurs à des ressources partagées ne peut se faire sans règles strictes, sous peine de générer des incohérences. On cherche en particulier à garantir le principe de **cohérence séquentielle** qui signifie que, quelle que soit la manière dont l'exécution parallèle se fait, le résultat du programme est identique à celui d'une exécution séquentielle suivant l'ordre des instructions du programme. La coordination et synchronisation des processeurs est donc un point crucial de l'approche à mémoire partagée et nous les discuterons en détail dans le paragraphe 8.2.

8.1 Multithreading

D'un point de vue pratique la mise en oeuvre du parallélisme dans une architecture à mémoire partagée se fait avec le concept de **microtasking** plutôt que celui de **macrotasking**. Le macrotasking correspond à du parallélisme à gros grain et utilise des primitives comparables aux primitives de UNIX permettant de créer de nouveaux processus et de les terminer (spawn, fork et join).

Le microtasking est plus adapté à un parallélisme à grain fin, comme par exemple à celui qui peut exister au niveau des variables d'une boucle. Le microtasking utilise le concept de **multithreading** qui est une notion de base dans un

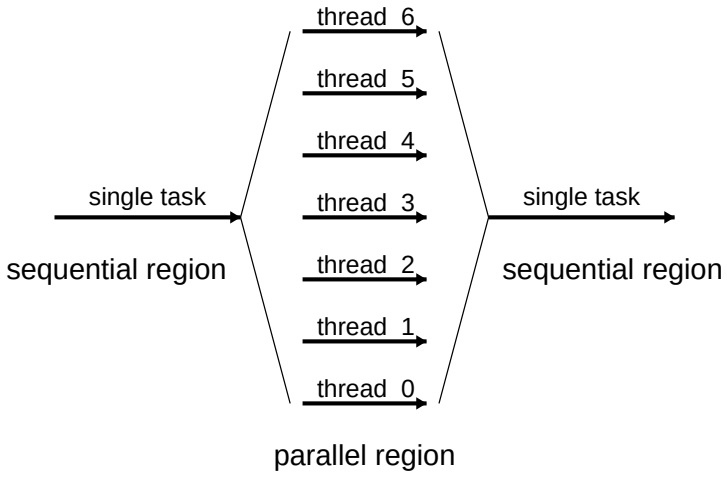

FIGURE 8.1 – *Exemple de multithreading d'une tâche dans une région parallèle.*

modèle de programmation à mémoire partagée. Il permet la création de plusieurs flots de contrôle séparés qui pourront être exécutés sur des processeurs différents.

La création de processus enfants comme le permet le système d'exploitation UNIX avec la primitive *fork* n'est pas adaptée au parallélisme à grain fin en raison de la lourdeur d'un tel processus. La structure de données nécessaire à la description d'un processus UNIX standard est considérable car elle contient toutes les informations relatives à l'état de la machine (mémoire virtuelle, privilèges d'accès, état de fichiers, etc).

Toute cette structure doit être créé ou détruite chaque fois qu'un processus apparaît ou disparaît. Cela implique un temps d'overhead important, inacceptable en parallélisme où un flot de contrôle doit pouvoir se créer rapidement. De plus, l'état complet d'un processus est souvent inutile dans le cas d'une exécution parallèle.

Pour cette raison, on introduit le concept de **threads** qui sont des processus légers qui *partagent* l'état du processus parent (souvent noté **tâche** dans ce contexte). Les threads ont été d'abord mises en oeuvre dans le système d'exploitation MACH puis la terminologie est restée.

Une tâche peut ainsi se subdiviser en plusieurs threads, chacun assigné à un autre processeur (en général, il n'y a qu'un thread par processeur). Un thread est beaucoup plus éphémère qu'une tâche. Il peut être créé et détruit avec très peu d'overhead. Le modèle d'exécution par thread est illustré sur la figure 8.1. On parle de **région séquentielle** là où il n'y a qu'une tâche et de **région parallèle** lorsqu'il y a plusieurs threads concurrents. Souvent le système d'exploitation se charge de la création des threads. Leur nombre n'est pas fixe d'une exécution à l'autre, ce qui permet d'exploiter le parallélisme à disposition et de donner lieu à des programmes portables : le même code peut s'exécuter indifféremment sur $1, 2, ..., p$ processeurs. Bien que les différents threads peuvent prendre un temps

plus ou moins long, la fin de la section parallèle implique que tous aient terminé avant de débuter la section séquentielle qui suit. Cela constitue donc un point de synchronisation.

8.1.1 OpenMP

Il est possible de créer des threads parallèles directement avec les primitives offertes par POSIX à partir d'un programme écrit dans un langage quelqconque. Cependant, les machines à mémoire partagée actuelles (par exemple les SMP) offrent une approche un peu plus simple pour le programmeur.

En utilisant le standard OpenMP on peut, à partir de directives de compilation ajoutées comme commentaires à des programmes écrits en C, C++ ou Fortran, spécifier la création de threads parallèles pour la partie du code concernée par la directive. Par exemple, avec la syntaxe Fortan, on aurait

```
!$omp parallel
  print *, omp_get_thread_num()
!$omp end parallel
```

Dans cet exemple, les lignes commentaires, qui commencent par `!$omp`, sont interprêtés par OpenMP qui, en l'occurence, crée une section parallèle où chaque thread ne fait rien d'autre qu'écrire son identificateur. En C ou C++, les directives s'appellent des *pragma* et la syntaxe est

```
#pragma omp parallel
```

Ces directives sont typiquement placées avant une boucle pour indiquer la possibilité de répartir les itérations sur plusieurs processeurs. Pour indiquer une telle situation en C++, on aurait par exemple

```
#pragma omp parallel
  for(i=0,i<n,i++){
    z[i]=a*x[i] + b
  }
#pragma end omp parallel
```

ce qui, dans ce cas, se parallélise très bien car chaque itération est indépendante des autres. On voit par ces exemples que la programmation multithread, ou plus spécifiquement OpenMP est du type SPMD : il n'y a qu'un seul programme qui décrit l'ensemble des threads.

Cepandant, pour bénéficier des performances que plusieurs coeurs peuvent offrir, il faut que chaque thread ait suffisament de travail entre deux accès mémoire. Sinon, les threads sont en compétition pour l'accès aux données et l'exécution parallèle peut être plus lente que l'exécution séquentielle.

Le code ci-dessous, qui ne fait pas grand chose d'utile, illustre néanmoins l'importance d'avoir une grande intensité arithmétique. Pour chaque donnée `v[i]`

on calcule un résultat `w[i]` avec une itération arbitraire nécessitant 1 million d'étapes pour chacune des $N = 10'000$ éléments du vecteur.

L'instruction `omp_set_num_threads(n)` permet de choisir le nombre de threads. Ici on pris $n = 4$ en vue d'une exécution sur les 4 coeurs d'un laptop de 2020. Avec $n = 1$, on a une exécution séquentielle. On observe alors un speedup de 3.3 sur 4 coeurs. Par contre, si on réduit la taille de la boucle sur `j`, le speedup chute jusqu'à des valeurs qui peuvent devenir inférieure à 1.

Le temps d'exécution est mesuré grâce à la fonction `omp_get_wtime()`. Certaines autres fonctions de mesure de temps peuvent donner des résultats éronés. Ce code utilise les threads posix et nécesssite la bibliothèque et les «header files» appropriés.

```
//  g++  iter.cc  -fopenmp -lpthread

#include <iostream>
#include <pthread.h>
#include <omp.h>
#include <vector>
#include <numeric>  // for acumulate

using namespace std;

int main(){
  int N=1e4;
  double s=0;
  double t0,t1;
  auto v=vector<double>(N);
  auto w=vector<double>(N);

#pragma omp parallel for
  for(int i=0;i<N;i++)v[i]=i;

  omp_set_num_threads(4);  // change it to 1 for a seq execution
  // 40 sec in sequential   12 sec with 4 threads

  t0=omp_get_wtime();

#pragma omp parallel for
  for(int i=0; i<N; i++){
    for(int j=1;j<=1000000;j++)w[i]+=v[i]/(j*j);}

  t1=omp_get_wtime();
  s=accumulate(w.begin(), w.end(), 0);
  cout<<"s="<<s<<"  execution time="<<t1-t0<<" [s]"<<endl;
}
```

Des directives semblable au `parallel for` permettent de spécifier plus finement comment le partage de la boucle devra être réalisé (par bloc d'indices consécutifs ou plutôt de façon modulo). Nous renvoyons le lecteur à la documentation OpenMp pour plus de détails.

Finalement, soulignons que bien que l'espace mémoire soit accessible par tous les processeurs, certaines directives OpenMP (par exemple `!$omp private ma_variable`) permettent de définir des variables locales dont l'accès n'est possible que par un seul processeur.

Finalement notons que la fonction

```
omp_set_num_threads(p)
```

qui permet d'imposer le nombre de threads souhaité ne garantit pas forcément une distribution équilibrée des threads sur le processeurs disponibles. S'il n'y a pas assez de processeurs, ou bien si certains sont trop chargés aux yeux du système d'exploitation, plusieurs threads pourrant être lancés sur le même processeur.

8.2 Coordination et Synchronisation des processeurs

Plusieurs processeurs qui travaillent ensemble dans le but de résoudre un même problème doivent **coordonner** leurs actions pour garantir l'exécution correcte d'un programme. La *coopération* entre les processeurs se double obligatoirement d'une certaine forme de *compétition* qui, si elle n'est pas strictement contrôlée, conduit à une exécution erronée. Par exemple, plusieurs processeurs peuvent vouloir écrire en même temps des valeurs différentes en une même position mémoire ou entrer en compétition pour l'accès à des ressources communes.

Le temps d'exécution d'un processus peut fluctuer de manière imprévisible d'un processeur à l'autre. Par conséquent, l'ordre dans lequel les opérations prennent place n'est pas répétitif d'une fois à l'autre, ce qui peut conduire à un comportement non déterministe si un schéma de synchronisation n'est pas clairement défini. Si par exemple le processeur P_1 contient l'instruction `A=A+B` et le processeur P_2 l'instruction `B=B+A`, la valeur finale de `A` et `B` dépend crucialement de l'ordre dans lequel ces deux instructions sont exécutées. En supposant qu'initialement `A=3` et `B=5`, le résultat final est `A=8`, `B=13` si P_1 passe avant P_2, et `A=11`, `B=8` dans le cas contraire.

Une situation, comme celle expliquée ci-dessus, où un résultat dépend du premier processeur qui atteint une instruction donnée s'appelle une *race condition* (condition de course).

La mise au point d'un programme et les raisonnements qui s'y rapportent peuvent ainsi s'avérer trompeusement faciles. Le contrôle de la coordination et de la compétition entre les processeurs se résume au contrôle de l'accès à la mémoire, et au contrôle de l'ordonnancement (synchronisation) des différentes tâches.

8.2.1 Contrôle d'accès

Un exemple simple permet d'illustrer le problème du contrôle de l'accès à une variable partagée par tous les processeurs. Supposons que l'on désire calculer la somme d'un grand nombre de données et que chaque processeur se charge

d'évaluer une partie de cette somme. Le résultat final est obtenu en récoltant les sommes locales contenues dans chaque processeur et en les additionnant dans la variable `Global_sum`, par exemple. Chaque processeur aura donc typiquement à exécuter l'instruction

```
Global_sum=Global_sum+local_sum
```

où `local_sum` a une valeur différente en chaque processeur. Comme tous les processeurs travaillent à leur propre rythme, il est impossible de prévoir comment l'accès à `Global_sum` va se faire. Il se pourrait très bien que le processeur P_1 lise la valeur de `Global_sum` et, avant d'avoir eu le temps de la modifier et de la récrire en mémoire, un autre processeur, disons P_2 lise de même `Global_sum`. Lorsque la valeur de `Global_sum` sera récrite en mémoire par P_2, mettons juste après P_1, la contribution de P_1 sera oubliée et le résultat de la somme totale incorrect. L'instruction

```
Global_sum=Global_sum+local_sum
```

doit donc être exécutée de manière séquentielle, un processeur après l'autre, ou, autrement dit, il ne faut permettre l'accès à certaines variables qu'à un processeur à la fois. C'est ce qu'on appelle l'**exclusion mutuelle**.

Un autre exemple classique illustrant l'importance d'une exclusion mutuelle est donné par l'accès simultané par plusieurs personnes à un même compte en banque. On peut imaginer que des titulaires communs se présentent au même moment à différents guichets automatiques et essayent tous de retirer la totalité du montant disponible. Si chaque guichet automatique ne bloque pas l'accès à toutes les personnes présentes, sauf une, chacun pourra effectuer ce retrait et tromper la machine en prenant en tout plus d'argent qu'il n'y en a sur le compte.

Pour assurer cette exclusion mutuelle, des mécanismes de contrôle d'accès sont disponibles dans les machines à mémoire partagée. Ils s'expriment typiquement par les primitives `LOCK` et `UNLOCK`, avec lesquelles l'exemple précédent s'écrirait

```
LOCK(Global_sum)
Global_sum=Global_sum+local_sum
UNLOCK(Global_sum)
```

Le premier processeur qui rencontre le `LOCK` pénètre dans la section de code en bloquant l'accès à `Global_sum`. Les autres processeurs qui rencontrent ensuite le `LOCK` doivent attendre que la variable soit débloquée par le `UNLOCK` avant de pouvoir continuer et exécuter l'instruction de somme.

Ce mécanisme présente en principe un risque d'*interblocage* ou *deadlock* qui apparaît lorsque deux processeurs attendent mutuellement que l'autre débloque la situation. Par exemple, P_1 bloque la variable `A` et essaye de lire la variable `B` déjà bloquée par P_2. Si P_2 a aussi besoin de `A` avant de débloquer `B`, le système part dans une boucle infinie dans l'attente d'un événement qui n'aura jamais lieu.

Pour éviter ce type de complication et dans le cas d'une programmation SPMD, il est préférable de définir **sections critiques** pour protéger les parties de code qui ne doivent être effectuées que par un seul processeur à la fois. Une section critique est donc, par sa nature même, une entité séquentielle, qui empêche le parallélisme de prendre place. Si on ne peut pas éviter les sections critiques dans certaines applications parallèles, il faut en revanche essayer de réduire au maximum l'attente qu'elles provoquent dans les autres processeurs.

Avec OpenMp, on déclare une section critique de la façon suivante :

```
!$omp critical
  list of critical instructions
!$omp end critical
```

8.2.2 Contrôle de séquence

Le but du contrôle de séquence est de définir l'ordonnancement des différentes tâches, afin de garantir la validité d'un programme s'exécutant sur un multiprocesseur. Comme on l'a déjà mentionné, on ne peut pas compter sur les vitesses d'exécution relatives d'un processeur à l'autre pour prédire l'ordre dans lequel les instructions d'un programme auront lieu.

Le besoin de synchronisation est particulièrement évident dans le cas de calculs itératifs sur un grand nombre de données. Si les données sont réparties sur plusieurs processeurs et que le calcul implique des communications entre les processeurs, il est indispensable de synchroniser les actions de chacun : avant de passer à l'itération suivante, il faut être sûr que la précédente est totalement achevée et que toutes les données sont correctement mises à jour par les processeurs qui en ont la charge.

En pratique, on réalise ce but en incluant dans le programme des **barrières** de synchronisation. Une barrière de synchronisation est un point du code qui ne peut être franchi avant que tous les processeurs l'aient atteint.

Par exemple, si on a le code multithread suivant qui calcule un tableau `b` à partir d'un tableaux `a` de même taille `n`, selon le code

```
for i=1 to n:  b[i]=0.5*(a[i-1] + a[i+1])
```

Puis le tableau `a` est mis à jours avec les nouvelles valeurs calculée dans `b`

```
for i=1 to n:  a[i]=b[i]
```

et le tout est itéré plusieurs fois. La figure 8.2 illustre la nécessité de mettre une barrière de synchronisation après chacune des boucles, afin d'éviter que certains threads utilisent des valeurs déjà modifiée par d'autre threads.

Le code ci-dessous propose une implémentation des barrières dans le cas d'un code Fortran utilisant openMP.

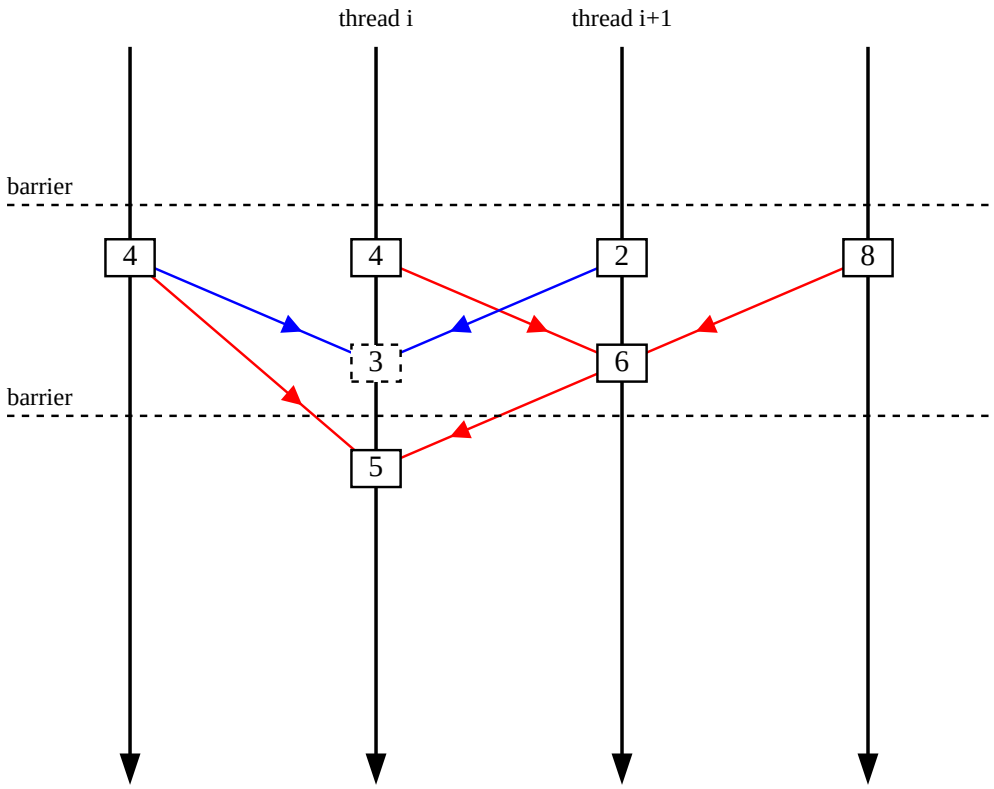

FIGURE 8.2 – Illustration d'un problème d'incohérence de résultat qui peut arriver en l'absence d'une barrière de synchronisation. Dans cet exemple le thread i est en retard sur le thread $i+1$ et utilise la valeur de $a[i+1]$ de l'itération suivante pour calculer $b[i]$.

```
!$omp parallel private me, p, i, i0, i1
  omp_set_num_thread(p)    ! p est le nombre the thread choisi
  me=omp_get_thread_num()
! compute the local limits of the loop
  i0=(n/p)*me
  i1=(n/p)*(me+1)

  do i=i0, i1
    b(i)=(1/2)*(a(i-1) + a(i+1) )
  enddo

!$omp barrier

  do i=i0, i1
    a(i)=b(i))
  enddo

!$omp end parallel
```

La directive `!$omp barrier` indique que la boucle sur `i` doit se terminer pour tous les processeurs avant de passer à la boucle suivante suivante. Sans quoi, certains processeurs pourraient déjà avoir modifié certaines valeurs de `a` avant que celles-ci ne soit utilisée par d'autres.

Une façon simple de construire une barrière de synchronisation est d'utiliser une variable partagée (appellons-la "`barrier`"), dont la valeur initiale est le nombre de tâches à synchroniser. Chaque fois qu'un processeur arrive à la barrière (parce qu'il a terminé), il décrémente de 1 cette variable. Lorsque la valeur atteint zéro, cela veut dire que tous les processeurs ont fini la tâche qui leur était assignée et que la suite du programme peut être exécutée.

A un tel point de synchronisation, les processeurs attendent habituellement que la barrière se lève en contrôlant sans cesse la valeur de la variable `barrier`. Les processeurs les plus rapides attendent les plus lents et il vaut mieux avoir des tâches de durée à peu près égales sur chaque processeur pour ne pas voir se dégrader le bénéfice du parallélisme (load balancing).

Que cela soit à une barrière de synchronisation ou à un `LOCK`, le fait de "boucler" continuellement sur l'instruction qui bloque les processeurs s'appelle **busy-waiting**. C'est la même chose qu'attendre au téléphone que son correspondant soit prêt à répondre. Pendant ce temps, on ne fait rien et on gaspille des cycles CPU.

Remarquons encore qu'un *busy-waiting* mis en oeuvre de manière trop naïve peut conduire à une situation de deadlock si le nombre de tâches est supérieur au nombre de processeurs. Par exemple, dans le cas d'une barrière de synchronisation, tous les processeurs pourraient se bloquer en attendant la fin de la dernière

tâche. Mais celle-ci n'aura jamais lieu car la dernière tâche doit être exécutée par un des processeurs déjà en attente.

8.3 Primitives de Synchronisation

Une grande partie des primitives de synchronisation utilisées dans les multiprocesseurs proviennent des systèmes d'exploitation multitâches développés pour les monoprocesseurs. Bien que la compétition pour des ressources communes soit de même nature, les solutions inspirées par ces techniques ne sont pas toujours adaptées au cas de systèmes comportant beaucoup de processeurs. Le parallélisme inhérent au système d'exploitation séquentiel multitâche est de grande granularité et intervient à un degré beaucoup plus faible que dans le cas d'une machine véritablement parallèle.

La clé des primitives de coordination les plus simples est l'utilisation d'instructions dites **indivisibles** ou **atomique** qui sont des instructions non interruptibles qui ne peuvent être exécutées que par un processeur à la fois. Cela signifie qu'il existe un dispositif hardware qui "sérialise" les appels provenant des différents processeurs pour exécuter ces instructions indivisibles. Dans ce sens là, ces primitives permettent de contrôler et gérer le parallélisme en ayant recours à une technique d'exécution séquentielle et ne sont ainsi pas vraiment satisfaisantes pour des systèmes avec beaucoup de processeurs. Comme on le verra, il existe tout de même une primitive de coordination véritablement parallèle (la primitive `fetch-and-add`), mais qui nécessite un réseau d'interconnexion actif.

8.3.1 La primitive test-and-set

Une des primitives de base est le **test-and-set** qui utilise une variable de blocage ou flag (`S`, dans l'exemple qui suit). La définition de `test-and-set` peut s'exprimer par les instructions suivantes

```
test-and-set(S)
   atomic{ tmp=S; S=0}
   return tmp
```

Les accolades { et } délimitent la partie indivisible de l'instruction `test-and-set`. Le fonctionnement est le suivant. La variable `S` est initialisée à 1. Lorsque le `test-and-set` est exécuté pour la première fois, en une étape indivisible, `S` est copié dans `tmp` et mis à zéro. Le processeur qui exécute le `test-and-set` en premier reçoit en retour la valeur 1, alors que les suivants reçoivent la valeur 0.

La primitive `test-and-set`, par le biais de la variable `S`, réalise la mise en oeuvre de **sémaphores** qui permettent d'assurer une exclusion mutuelle ou un `LOCK`. Lorsque `S` est trouvé à 0, l'accès est bloqué, alors que si `S=1`, le processeur

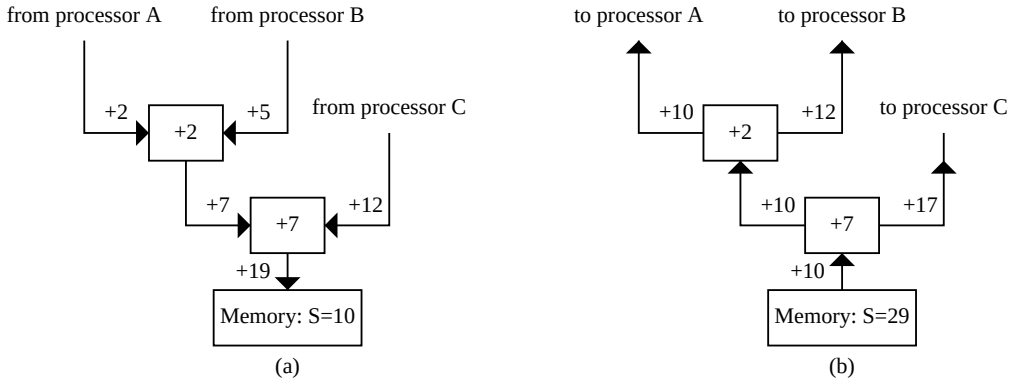

FIGURE 8.3 – *Mise en oeuvre du Fetch-and-Add sur un réseau d'interconnexion dont les switches permettent des opérations arithmétiques et le stockage d'information : (a) Les données provenant des processeurs sont respectivement 2, 5 et 12. Chaque switch conserve la valeur venant de gauche et renvoie vers le bas la somme des valeurs reçues. La valeur de S est incrémentée et passe à 29. (b) L'ancienne valeur de S (10) est renvoyée dans le réseau. Les switches renvoient à gauche la valeur reçue par le bas et à droite la somme de la valeur reçue et la valeur stockée. Le processeur A reçoit ainsi l'ancienne valeur de S.*

peut continuer le programme. A la fin de la section critique (ou région d'exclusion), le processeur "ayant la main" remet `S` à 1 (`UNLOCK`), et les autres peuvent à leur tour exécuter le code réservé.

Parfois, les sémaphores sont accessibles à l'utilisateur sous forme de primitives de haut niveau, généralement notée `P` et `V`, dont l'usage est le suivant :

```
P(S); section critique; V(S)
```

8.3.2 La primitive Fetch-and-add

Les primitives de synchronisation que nous venons de voir sont dites **bloquantes**, car un seul processeur à la fois peut les exécuter. Il peut arriver, si le nombre de processeurs est important, que chaque tâche passe un temps considérable simplement à attendre de pouvoir exécuter la primitive en question ou tester un sémaphore.

Il y a donc un besoin pour une primitive de coordination véritablement parallèle qui puisse être exécutée simultanément par plusieurs processeurs. Il se trouve que beaucoup d'opérations sur des variables communes sont indépendantes de l'ordre dans lequel elles sont exécutées. C'est le cas, notamment, de la somme de plusieurs données ou encore d'une file d'attente. Le `fetch-and-add` est aussi une opération de ce type.

La primitive `fetch-and-add` peut être exécutée par plusieurs processeurs en

même temps, sans créer de blocages ni nécessiter d'essais réitérés. Elle permet de protéger une section critique sans elle-même en constituer une. Cependant, elle demande un réseau d'interconnexion élaboré qui permet de combiner plusieurs données devant être stockées au même endroit et de conserver des résultats intermédiaires. Le `fetch-and-add` retourne en chaque processeur une valeur différente permettant de décider de la suite des opérations.

On verra plus loin comment on utilise `fetch-and-add` pour mettre en oeuvre des sémaphores et des barrières de synchronisation en parallèle. Mais, avant cela, étudions son fonctionnement. L'instruction `fetch-and-add` peut s'exprimer ainsi :

```
fetch-and-add(S,I)
  atomic{ tmp=S; S=S+I}
   return tmp
```

`S` est la variable partagée et `I` un incrément propre à chaque processeur. Lors de plusieurs appels simultanés, le résultat du `fetch-and-add` est identique à celui qui serait obtenu si ces appels étaient faits de manière séquentielle, mais dans un ordre quelconque non spécifié. De plus, la variable `S` n'est lue qu'une seule fois.

La mise en oeuvre du `fetch-and-add` sur une machine parallèle disposant d'un réseau capable d'assurer les opérations intermédiaires est illustrée dans la figure 8.3. Les incréments `I` sont respectivement 2, 5 et 12. La variable `S` (initialement égale à 10) en mémoire vaut 29 après le `fetch-and-add` et les valeurs retournées au processeurs sont 10, 12 et 17, ce qui correspond aux valeurs intermédiaires de `S`. Ces mêmes valeurs seraient retournées par 3 appels consécutifs de `fetch-and-add` avec incréments `I=2,5,12`, respectivement.

Le `fetch-and-add` est une construction très puissante qui limite les contentions de réseau et de mémoire. Par contre, le prix du réseau d'interconnexion est élevé par rapport à un bus.

Une utilisation naturelle du `fetch-and-add` est la mise en oeuvre d'une boucle partagée par plusieurs processeurs :

```
!$omp parallel do
  do i=1,n
    compute a(i)
  enddo
!$omp parallel do
```

Si l'indice de boucle commun `i` est incrémenté avec `fetch-and-add(i,1)`, il y aura une gestion efficace du partage du travail. On peut donc imaginer l'implémentation suivante

```
  i=1
  repeat
```

```
    j=fetch-and-add(i,1)  // j est une variable locale
    if(j>n)exit
    compute a(j)
  end repeat
```

8.3.3 Réalisation d'un sémaphore parallèle avec fetch-and-add

Un sémaphore pouvant être simultanément testé par plusieurs processeurs et protégeant une section critique peut être construit comme suit, avec une variable partagée `S` initialisée à 1.

```
while fetch-and-add(S,-1)<1
   do fetch-and-add(S,1)
end while
CRITICAL SECTION
fetch-and-add(S,1)
```

La boucle `while` fait office d'instruction `LOCK`. En effet le "premier" processeur qui exécute le `fetch-and-add` entre dans la section critique. Les autres $p-1$ processeurs incrémentent et décrémentent continuellement `S` mais, avec $p-1$ processeurs, `S` ne peut varier qu'entre $p-1$ et 0. Les $p-1$ processeurs seront en "busy waiting" dans le `while` tant que le premier processeur n'aura pas incrémenté à nouveau `S` avec le `fetch-and-add` qui suit la section critique (le `UNLOCK`). Dès lors, `S` pourra repasser à 1 et le processeur suivant sera admis dans la section critique.

8.3.4 Réalisation d'une barrière de synchronisation avec fetch-and-add

La manière la plus simple de construire une barrière avec un `fetch-and-add` est de considérer une variable globale `X` initialisée à zéro. Pour synchroniser N tâches, il suffit d'écrire

```
fetch-and-add(X,1)
wait for X=N
```

à la fin de chaque tâche. La variable `X` sera mise à N lorsque la Nème tâche sera terminée, débloquant ainsi la barrière. Remarquons, entre parenthèse, que tester la valeur de `X` peut se faire de façon non-blocante avec la valeur retournée par `fetch-and-add(X,0)`.

En principe, on pourrait se contenter de l'implémentation ci-dessus pour une barrière. Mais en pratique, un programme contient beaucoup de telles barrières et

on veut éviter d'avoir une nouvelle variable dans chaque cas. Il faut donc remettre `X` à zéro avant la prochaine barrière. La solution consistant à écrire

```
X=0
...
if fetch-and-add(X,1)=N-1 then X=0
wait for X=0
```

n'est pas satisfaisante non plus car les processeurs peuvent être à des phases imprévisibles dans l'exécution des instructions. On pourrait avoir le cas défavorable suivant : la barrière est levée par le processeur A. Avant que le processeur B n'exécute le test pour savoir si `X=0`, le processeur A pourrait avoir déjà rencontré une autre barrière et remis `X` à 1. A nouveau, on tombe sur une situation de deadlock car B ne quittera jamais la première barrière et pourra encore moins ouvrir la deuxième. La bonne solution est

```
local_flag=(X<N)
if fetch-and-add(X,1)=2N-1 then X=0
wait for (X<N)≠ local_flag
```

Avant la première barrière, `X` est mis à 0 et `local_flag` (qui est une variable différente pour chaque processeur) est initialisé à la valeur `true`. Lorsque tout le monde a atteint la première barrière, `X=N`, et la barrière est levée. On arrive à la deuxième barrière avec toujours `X=N` mais `local_flag=false`. Comme `X<N` est faux aussi, la barrière est fermée jusqu'à ce que `X=2N` (c'est à dire que tous les processeurs aient terminé et `fetch-and-add(X,1)` ait retourné la valeur `2N-1`). Alors, `X` passe à zéro et `(X<N)` devient vrai. On passe ainsi la deuxième barrière, avec `X=0` à nouveau. Avec cette technique, on alterne, d'une barrière à l'autre, les conditions d'ouverture `X=N` et `X=2N`.

La raison fondamentale qui fait que cette approche marche est que la première barrière reste ouverte jusqu'à ce que le dernier processeur atteigne la deuxième.

En conclusion, la primitive **fetch-and-add** offre un moyen élégant et flexible pour gérer la coordination entre processeurs dans une architecture à mémoire partagée. Du point de vue fonctionnelle, elle permet de réaliser des barrières de synchronisation, des contrôles d'accès (lock/unlock) et permet aussi d'obtenir un équilibrage de charge dynamique. Du point de vue hardware, elle peut en principe être réalisée de façon non-blocante.

8.4 Avantages et désavantage du modèle

Dans ce paragraphe, nous discutons brièvement les avantages et désavantages du modèle à mémoire partagée par rapport au modèle à mémoire distribuée.

8.4.1 Avantages

Un système à mémoire partagée est plus simple du point de vue de l'utilisateur qui est en présence d'un modèle de programmation proche de celui des machines séquentielles.

Les communications entre les différentes tâches se font en modifiant des données dans des locations mémoires communes, ce qui constitue une extension naturelle du modèle de von Neumann. Le réseau d'interconnexion est ainsi caché à l'utilisateur et il en résulte une facilité de programmation qui évite au programmeur le besoin de gérer des mouvements de données entre les processeurs.

Notons que les mémoires partagées permettent par ailleurs d'émuler facilement des modèles de programmation différents, comme les échanges de messages. En conséquence, il est en général plus facile d'adapter, sur une architecture à mémoire partagée, un code développé pour une machine séquentielle car les modifications à y apporter sont de nature moins profondes.

Finalement, il n'est pas nécessaire de dupliquer des données communes en mémoire, ce qui optimise la gestion de l'espace mémoire global unique. Par ailleurs, cette gestion est facilitée par les techniques sophistiquées inspirées de l'expérience acquise pendant de nombreuses années avec les machines séquentielles. Les systèmes multi-utilisateurs sont ainsi plus faciles à réaliser que dans le cas des machines à mémoire distribuée pour lesquelles cette gestion n'atteint pas le même degré de maturité.

8.4.2 Désavantages

La souplesse des architectures à mémoire partagée se solde cependant par plusieurs problèmes. Du point de vue conception, elles sont plus compliquées en raison de la complexité de la gestion de la coopération entre les processeurs, notamment des conflits qui peuvent survenir. En pratique les systèmes à mémoire partagée se caractérisent par un nombre modeste de processeurs (quelques dizaines au maximum). Ces architectures ne sont pas extensibles ou scalables (on ne peut pas augmenter le nombre de processeurs arbitrairement) en raison, soit de la saturation du bus d'interconnexion, soit de l'augmentation des conflits mémoire. L'accès simultané de plusieurs processeurs à une même mémoire est limité et implique un temps de latence important.

De plus, les architectures à mémoire partagée sont peu compatibles avec un principe de **localité de référence** qui s'applique au parallélisme de la manière suivante : statistiquement, les données ne sont pas utilisées de manière uniforme par tous les processeurs. Certains utilisent des données plus fréquemment que d'autres et il faudrait que ces dernières soient accessibles rapidement par les processeurs qui les consomment. Dans une architecture à mémoire partagée, il n'y a pas de concept de proximité puisque l'on veut la mémoire uniforme. Si les données sont effectivement réparties de façon homogène dans la mémoire,