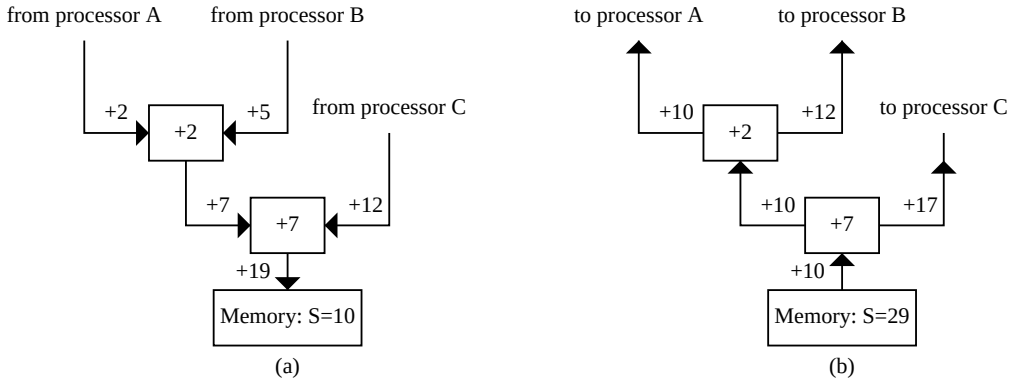

FIGURE 8.3 – *Mise en oeuvre du Fetch-and-Add sur un réseau d'interconnexion dont les switches permettent des opérations arithmétiques et le stockage d'information : (a) Les données provenant des processeurs sont respectivement 2, 5 et 12. Chaque switch conserve la valeur venant de gauche et renvoie 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.