Contents

0.1 Architecture et modèle de programmation pour les mémoires partagées

Architecture type

N'importe quelle mémoire peut accéder n'importe quel banc de mémoire. Mais il peut y avoir de la congestion si plusieurs processeurs veulent accéder à la même mémoire.

On peut avoir 2 versions:

- La version MIMD où les processeurs sont asynchrones ⇒ multicoeurs
- La version SIMD où les processeurs sont synchrones ⇒ GPU

Cette architecture s'appele un multiprocesseur.

0.1.1 Avantages

L'espace mémoire n'est plus fragmenté, il est unique et global, comme en séquentiel.

→ on se rapproche du monde séquentiel, plus familier.

La collaboration entre processeurs se fait par la modification de données partagées. Cela permet de paralléliser facilement et automatiquement certaines parties du code: des boucles sans dépendances.

0.1.2 Désavantages

Les choses sont plus compliquées qu'elles n'ont l'air.

Comme on a des mémoires cache, il faut pouvoir assurer la cohérence des mémoires caches. Une même variable pourrait être modifiée à plusieurs endroits. Et être modifiée de façon inconsistante.

⇒ Protocole de cohérence de cache Chaque donnée dans le cache a un status:

- copie unique (atomique)
- copie multible, mais valide
- invalide

- ...

Cette couche se fait au niveau matériel, de façon transparente.

On a aussi le problème dit de "**race condition**", qui indique qu'un résultat dépend de l'ordre dans lequel les processeurs modifient la mémoire

On a le principe dit de cohérence séquentielle qui oblige l'exécution parallèle à donner les mêmes résultats.

Donc le matériel est plus compliqué (cohérence du cache, primitives de synchronisation)

⇒ **Faible scalabilité: on ne peut pas avoir trop de processeurs dans une telle architecture...**

0.2 Modèle de programmation

0.2.1 Approche multithreads

(petit shéma)

En calcul parallèle, on souhaite avoir un thread par coeur. De fait, on a pas toujours un grand contrôle sur ce genre de choses, l'OS a souvent la main...

Le système Posix de threads peut être exploité pour cet objectif de parallélisation. OpenMP (Open Multi-Processing): permet une utilisation facilitée pour le programmeur.

C'est par des directives de compilation ajoutées au code qu'on précise le souhait de paralléliser une portion du code et d'en faire une région parallèle.

0.2.1.1 Exemple en C++

```
# pragma omp parallel
for (i = 0; i < n; i++) z[i] = a*x[i] + y[i];
# pragma end omp parallel
```

Il n'y a pas de dépendances, cette boucle peut facilement se paralléliser... Chaque thread va s'occuper d'une section du tableau. Mais pour du parallélisme plus compliqué, ce n'est pas toujours simple...

Dans openMP, il y a la notion de variables privées, associée à chaque thread, ou publiques accessibles partout.

0.2.2 Problème du contrôle d'accès en mémoire partagée

Deux threads ne doivent pas accéder à les mêmes données en même temps.

Il y a plusieurs primitives qui permettent de garantir cela, plus ou moins explicites selon l'environnement de programmation.

Exemple:

```
Lock(Global-sum)
// Global-sum: variable globale...
// Cela bloque l'accès à Global-sum si un thread s'en occupe déjà.
Global-sum += local_sum // local_sum: variable privée au thread
Unlock(Global-sum)
```

Ce mécanisme s'appelle **l'exclusion mutuelle** et les portions de code qui le nécessitent s'appellent des sections critiques ou régions critiques.

Utiliser

```
# pragma omp critical
list of critical instructions
# pragma end omp critical
```

0.2.3 Problème de contrôle de séquence

Il faut garantir que certains processeurs n'utilisent pas des données qui ne sont pas encore mises à jour.

Exemple:

```
repeat
  for i = 1 to n: b[i] = (a[i - 1] + a[i + 1])/2
# pragma omp barrier // À ajouter ici pour ne pas avoir de soucis...
  for i = 1 to n: a[i] = b[i]
```

Il faut ajouter une barrière de synchronisation pour pallier à ce problème....

Le contrôle d'accès et de séquence s'obtient grâce à des primitives de synchronisation de bas niveau souvent implémentées au niveau matériel.

Il faut avoir des **instructions dites atomiques ou indivisibles**, c'est à dire qui ne peuvent pas être interrompues et qui ne peuvent s'exécuter que par un seul thread à la fois.

On peut par exemple avoir une sérialisation de l'exécution des threads sur ces instructions.

⇒ perte de parallélisme... (C' est la sérialisation temporelle ?..).

Souvent ces primitives sont basées sur les systèmes d'exploitations multiprocesseurs

- Sémaphores, compare-and-swap (on prend la variable, on la modifie, et on vérifie que la variable n'a pas été modifiée avant d'écrire la modification sur le disque...)