Contents

0.1 Primitives de coordination 1
0.1.1 Primitive Fetch-and-add 1
0.1.2 Comment utiliser le fetch-and-add pour coordonner les threads ? 1
0.2 2.4 Modèle de programmation qui a priori s'adapte aux mémoires partagée ou distribuées. 3

0.1 Primitives de coordination

0.1.1 Primitive Fetch-and-add

D'un point de vue fonctionnel, cette primitive peut être représentée comme:

```
fetch-and-add(S, I) # S variable globale, partagée; I: incrément privé à chaque thread
  atomic{tmp = S; S = S + I}
  return tmp
```

atomic: un seul thread peut le faire à la fois, et non-interruptible

On peut proposer des implémentations matérielles de fetch-and-add qui garantissent la fonctionnalité, tout en permettant à plusieurs processeurs de l'exécuter en même temps. C'est une implémentation dite **non-bloquante**.

(Petit shéma du prof) Les incréments sont combinés sur leur chemin vers la mémoire et le résutlat du fetch-and-add multiple est:

- A reçoit 10 (valeur de S original)
- B reçoit 12 (valeur de S après incrément par A)
- C reçoit 17 (valeur de S après incrément de A et B)

Donc cela revient à une exécution séquentielle dans l'ordre A, B et C. Ici, c'est fait en parallèle, mais dans un "ordre virturel" imprévisible par le programmeur.

0.1.2 Comment utiliser le fetch-and-add pour coordonner les threads ?

0.1.2.1 Équilibrage de charge: s'assurer que tous les processeurs soient toujours occupés. version sans équilibrage

```
# pragma omp parallel
for i = 0 to n:
  compute(a[i])
endfor
# pragma end omp parallel
```

Ici, le partage des a[i] est fait à l'avance, sans se préoccuper du temps de calcul de chacun.

version avec équilibrage

```
# pragma omp parallel
i = 0
repeat
  j = fetch-and-add(i, 1) // Ici, chaque processeur obtient un autre indice: 0, 1, ...
  if j > n: exit
  compute(a[i])
end repeat
```

0.1.2.2 Exclusion mutuelle avec fetch-and-add

```
S = 1
while fetch-and-add(S, -1) < 1
  fetch-and-add(S, 1)  // Dans cette boucle, S va varier entre 0 et p - 1 avec p: # threa
endwhile
CRITICAL SECTION
fetch-and-add(S, 1) // S augmente de 1 et la partie Lock va permettre à S
// de revenir à 1 et laisser un autre thread dans la section critique
```

0.1.2.3 Barrières de synchronisation Une seule barrière:

```
x = 0
.
.
.
fetch-and-add(x, 1)
wait for x = N // N: # processeurs à synchroniser
```

Cas de plusieurs barrières avec la même variable de x

Mauvaise solution

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

Le problème est qu'un processeur pourrait quitter la barrière et atteindre la deuxième barrière, remettre x à 1 et ceci avant que tous les processeurs aient quittés la première barrière.

Cette dernière se ferme dès que $x \neq 0$

$\Rightarrow$ deadlock

Bonne solution

On introduit une variable locale, local-flag, avec des valeurs différentes pour chaque processeur.

```
local-flag = (x < N) // Vrai au début si x=0
if fetch-and-add(x, 1) == 2N - 1 then x = 0
wait for (x < N) != local-flag
```

Après l'arrivée des N processeurs, x < N sera faux et la condition d'ouverture sera vraie. La barrière s'ouvre.

La deuxième barrière s'ouvre que quand x sera remis à 0, soit quand tous les processeurs ont atteint la deuxième barrière. **Le point clé est qu'une barrière reste ouverte jusqu'à l'ouverture de la suivante.**

0.2 2.4 Modèle de programmation qui a priori s'adapte aux mémoires partagée ou distribuées.

- Chaque instruction agit en parallèle sur tous les éléments d'une structure de données.
- On a des instructions dites "parallèles" et le programme est une séquence de ces instructions.
- Exemple: C = A + B où A, B, C sont des matrices et le + agit simultanément sur tous les éléments.
- where (A == 0) A = 5: pour tous les éléments de A qui sont nuls, leur mettre la valeur 5.
- sort(A) où A est un vecteur.
- Cela rend le code assez lisible et diminue les risques d'erreurs. Par contre, cela s'adapte moins bien aux problèmes irréguliers, où chaque processeur a des tâches assez différentes.

  ⇒ C++17: Cette version du C++ permet une forme de parallélisme de données sur des architectures multicoeurs (mémoire partagée) et sur les GPU, avec le compilateur nvidia. *Ici, on va seulement illustrer cette possibilité.* **Ce C++ repose sur les concepts suivants:**

  - vecteurs
  - lamba functions
  - algorithms
  - execution policy (si on fait du séquenciel ou du parallèle...) (exemple: execution:par ou execution:seq ou execution:par_unseq etc...)
  - On a le même code qui fonctionne en séquentiel, en parallèle ou en GPU

vectoriser: on prend n valeurs et on applique la même opération en même temps sur ces n valeurs...