Contents

Organisation de la famille Dupont Les ingrédients sont en commun

C'est un cas de mémoire partagée (Mais ce n'est pas une obligation du MIMD. Mémoire distribuée est Ok)

La difficulté potentielle est d'avoir de la compétition pour des ressources communes. Ex: tout le monde veut le pain en même temps. => temps d'attente.

- Deadlock (interbolckage: 2 processeurs (PE) qui attendent mutuellement que l'autre ait fini l'action.

Par exemple: il n'y a qu'un seul couteau et une seule plaque de beurre. Celui qui a le couteau ne le rend pas tant qu'il n'a pas le beurre et celui qui a le beurre ne le rend pas tant qu'il n'a pas le couteau. . . .

Pour le problème des tartines, comment s'arrête-t-on ?

- Soit il faut continuellement négocier avant de refaire une tartine
- Soit on rique d'en faire trop
- Soit chacun sait à l'avance combien il doit en faire. Et il faut peut-être prendre en compte la vitesse de chacun.

0.0.1 Résultats

Qu'a-t-on appris avec cet exemple ?

1. Il y a plusieurs façons de faire travailler les gens avec avantages et défauts.
2. Mémoire distribuée ou partagée, avec implication au niveau performance, et la programmation. (Sort un peu de notre façon naturelle de penser. . .)
3. Granularité: taille des tâches individuelles entre interation
4. Scalabilité: est-ce qu'on peut ajouter autant de PE tout en augmentant proportionnellement les performances ?
5. 2 types de parallélisme:
    - le parallélisme de données (le fait qu'ils doivent faire beaucoup de tartines)
    - le parallélisme de contrôle ou de tâche (les tâches à faire pour faire une tartine)

Le premier type de parallélisme offre un potentiel de performance (degré de parallélisme) bien plus grand que le second: il y a beaucoup de problèmes où on traîte de la même façon un très grand nombre de données.

Le degré de parallélisme sur les données peut être de plusieurs millier, millions, etc...

Pour le parallélisme de tâche, c'est plutôt l'ordre de 10 max...

6. Plus le système est contrôlé, plus il est facile à gérer...
7. Equilibrage de charge: il faut donner du travail à tout le monde pour espérer aller plus vite. Sinon, 1 PE fait tout le travail et il n'y a pas de gain... Cela signifie qu'on est capable de diviser le travail de façon équitable parmi les PE.

Ex: Multiplication matrice vecteur

On prend un algo particulier, appelé algo systolique..... ON a des PE spécialisés, qui fonctionnent ainsi:

```
Ax = y
                  a44
              a43 a34
          a42 a33 a24
t3   a41 a32 a23 a14
t2   a31 a22 a13  0
t1   a21 a12  0   0
t0   a11  0   0   0
0 ->  ->  ->  ->
     x1   x2   x3   x4
```

Après 4 itérations, on obtient en sortie

$$y1 = a11x1 + a12x2 + a13x3 + a14x4$$

Mais pendant ce temps les autres composantes yi se calculent. Et ainsi, après ces 4 premières itérations, on obtient y2, y3, y4 à chaque nouvelle itération

0.0.1.1 Performance de cet algorithme: (cas $n \times n$ avec n PE)

$$T_{systolique} = n\tau + (n-1)\tau = (2n-1)\tau$$

avec $n\tau$: l'itération pour y_1 et $(n-1)\tau$: les composantes suivantes

Qu'en est-il de l'algo classique séquenciel ?

$$T_{seq} = (n\tau + (n-1)\tau) \times n = n \times (2n-1)\tau$$

avec n de $n\tau$: nombre de multiplications, n de $(n-1)\tau$ le nombre d'additions et n de $(\dots)$ n le nombre de lignes

Cela ne sert à rien de paralléliser et d'utiliser de gros moyens pour un petit problème !

1 Chapitre 2: architecture et modèles de programmation.

1.1 Introduction:

Il y a plusieurs types d'architecture matérielle qui ont été développés pour faire du calcul à haute performance (HPC). On veut en faire le tour et discuter des modèles de programmation relatifs à ces architectures (En gros, on va voir comment on programme ces machines...)...

- Architecture SIMD, MIMD, hybrides, ... , vectorielles,

- Mémoires partagées et distribuées, ou hybrides
- "Instruction driven" (ce qui se fait à base d'instructions ?) ou "data driven" (ce qui se fait à base de disponibilité de données ?)
- Architecture ILP (Instruction Level Parallelism)

Pour les modèles de programmation, il y en a plusieurs:

- Échanges de messages (MPI)
- Multithreading (thread posix, openMP, ...)
- Data-parallelism (C++17)

1.2 Échanges de messages et mémoire distribuée

1.2.1 SIMD

processeur frontal connecté à pleins de processeurs, chacun avec une mémoire propre, et chacun est connecté à un réseau d'interconnexion. Ceci permet les communications entre PE, à travers des messages. On a un flot d'instruction unique diffusé du processeur frontal à chaque PE, selon une horloge unique

Le chapitre 3 sera consacré au réseau d'interconnexions

1.2.2 MIMD à mémoire distribuée

réseau de processeurs avec leur mémoire propre, connectés à un réseau d'interconnexion, chacun avec son horloge.

Si on suppose qu'on a des liens entre la mémoire de chaque processeur avec le réseau d'interconnexions, on appelle cette architecture NUMA: accès dirrect aux mémoires autres...

Du point de vue programmation, ce modèle implique qu'on doit distribuer les données du problème dans chacun des PE: **décomposition** des données, ou le **partitionnement** du problème.

1.2.2.1 Exemple Illustration: on veut calculer une fonction sur un domaine de points, représenté par une grille. (Par exemple les pixels d'une image).

But: faire coopérer chacun des PE pour résoudre le problème.

On peut donc diviser le domaine en sous-domaines identiques que chacun des PE reçoit. Chaque PE va effectuer le calcul sur son sous-domaine.

Souvent dans les calculs qui nous intéressent, la mise à jour d'un point de la grille dépend de la valeur des points voisins. Certains de ces points voisins sont dans la mémoire d'un autre PE => échange de messages avec les PE concernés.