Contents

0.1 Partitionnement statique et dynamique

Ce sont les méthodes qui permettent de répartir équitablement la charge de calcul entre tous les PE.

0.1.1 Cas statique

Avant l'exécution, on découpe le problèmes en tâches d'égale durée.

La difficulté vient dans le cas de problèmes irréguliers, où les temps CPU varient entre les différentes données.

Exemple: modèle de trafic routier (schéma cours).

Il faudrait répartir le réseau routier de façon équilibrée.

Cela peut aussi dépendre de où les voitures se trouvent.

(NB: si les voitures bougent, la charge va bouger aussi => partitionnement dynamique)

Cas statique: découpage **avant exécution**, avec comme nécessité de connaître les temps de calcul associés à chaque point du domaine.

Il y a plusieurs techniques pour faire ce découpage statique.

- Bisection récursive
- Courbes de Hilbert / space filling curves
- Partitionnement de graphes

Toutes ces techniques reviennent à faire un découpage équilibré d'un problème irrégulier.

0.1.1.1 Bisection récursive Découper le domaine en 2 moitiés équivalente en terme de travail. Puis récursivement, on redécoupe chaque moitié en 2. => nb de processeurs qui sera une puissance de 2.

Les découpages sont dans les axes du domaine. On change d'axe à chaque découpage...

(Beaucoup d'exemples)

0.1.2 Cas dynamique

Dans ce cas là, on ne connaît pas d'avance la charge du calcul associé à chaque point du domaine.

- Exemple: ensemble de Mandelbrot, trafic routier en fonction de l'heure, météo.

Donc souvent, il faut repartitionner le domaine pour réequilibrer les charges. Mais cela implique de migrer des données, et cela prend du temps.

A priori, c'est un problème difficile de décision quand il faut réequilibrer.

Pour mettre en place un équilibrage de charge dynamique, il faut:

- Monitorer l'activité des PE et détecter un déséquilibrage ($\Delta = m - \mu$)
- Décision: est-ce que cela vaut la peine de passer du temps à rééquilibrer plutôt que de laisser faire ? Ce déséquilibre pourrait se corriger lui-même après un certain temps.

  -> migrer ou non.

On peut avoir une stratégie globale ou locale:

- globale: 1 processeur prend une décision sur la base de la connaissance de tout le système
- locale: des groupes de processeurs voisins se mettent d'accord pour échanger du travail.

L'approche peut être très différente pour les problèmes dit itératifs (où on peut exploiter ce qui s'est passé aux itérations précédentes pour améliorer les suivantes)

Et les problèmes non-itératifs, où il n'y a qu'une seule tâche par PE sans point de synchronisation intermédiaire. (Mandelbrot)

0.1.3 Cas non-itératif

Les PE qui ont fini cherchent du travail chez ceux qui n'ont pas fini. On peut faire cela centralement ou localement, il faut savoir à qui demander, et être à l'écoute de ceux qui demandent.

Le "**work-stealing**" auprès d'un agent centralisé est ce qui est le plus simple, mais pas forcément le plus scalable.

0.1.4 Cas itératif

Modifier le partitionnement en fonction du déséquilibre précédent.

0.1.4.1 Cas d'étude Cet exemple va nous permettre de définir un critère de réequilibrage, à savoir à quelles itérations il est optimum de migrer les données pour les re-répartir équitablement.

$$T_{cpu} = \int_0^\gamma m(s)ds = \int_0^\gamma (m(s) - \mu(s))ds + \int_0^\gamma \mu(s)ds$$

Donc si on équilibre à chaque temps T_i:

$$T_{cpu} = \int_0^\gamma \mu(s)ds + \sum_{i=1}^{n} (\int_0^{T_i} u_i(t)dt + c_i)$$

avec $u_i(t)$: et c_i:

On suppose une certaine régularité dans le déséquilibrage.

$$T_i = T, u_i = u$$

$$T_{cpu} = \int_0^\gamma \mu(s)ds + \frac{\gamma}{T}(\int_0^T u(t)dt + c)$$

On cherche T tel que T_{cpu} est minimum

$$o = \frac{\delta T_{cpu}}{\delta T} = -\frac{\gamma}{T^2}(\int_0^T u(t)dt + c) + \frac{\gamma}{T}u(T)$$

D'où le T optimal obéit à

$$Tu(T) - \int_0^T u(t)dt = C$$

(schéma cours...) u(T) T est l'intégrale complète à laquelle on soustrait la surface ...