Contents

1 Chapitre 6

1.1 6.1 Notion de tâche

Une tâche T_i est un ensemble d'instructions réalisées sur un seul PE, entre 2 points de communication ou synchronisation.

Une tâche est caractérisée par ses données d'entrée E_i et données de sortie S_i.

1.1.1 Granularité

La granularité est une mesure de la taille de la tâche (nombre d'instructions réalisées pendant cette tâche...), à savoir le nombre d'instructions réalisée.

On parle de granularité fine, moyenne ou grossière en fonction de sa taille.

1.1.2 Indépendance de deux tâches

T_i et T_j sont indépendants si

- $S_i \subset S_j =$ donc pas de données communes modifiées
- $S_i \subset E_j =$ donc pas de données lues par T_j et modifiée par T_i
- $S_i \subset E_i =$
- $E_i \subset E_j$ -> pas de contrainte dessus....

Deux tâches qui sont indépendantes peuvent être exécutées dans un arbre quelconque.

Deux tâches qui ne sont pas indépendantes doivent être exécutées dans un ordre spécifié. On notera par exemple:

$$T_i < T_j$$

pour indiquer que T_i doit être fini avant que T_j ne démarre.

Deux tâches qui sont indépendantes peuvent s'exécuter en parallèle !

Cela peut s'exprimer par une **relation de précédence** et un **graphe de précédence**.

Notre question sera de voir comment on peut paralléliser une application sur la base de son graphe de précédence.

1.2 6.2 Partitionnement, placement et ordonnancement

1.2.1 Partitionnement

C'est la division d'un problème en tâches

Souvent, le partitionnement concerne la division des données en sous-domaines de calcul. (On peut faire des partitionnements en tranches verticales ou horizontales...) (Partitionnement (décomposition) en blocs) (Ou encore 'modulo' ou cyclique'... Utile pour répartir équitablement toutes les zones du domaine sur tous les processeurs, mais peu recommandé s'il y a des communications car on page la latence plusieurs fois...)

Le partitionnement est au choix de l'utilisateur, et il y a plusieurs contraintes: - si on choisit une granularité fine, il y aura un meilleur potentiel de parallélisme. Mais cela risque d'augmenter l'overhead de la gestion de ce parallélisme: (regarder schéma cours)

En général, on s'attend à ce que la granularité optimale requière la solution d'un problème d'optimisation.

1.2.2 Placement

C'est le choix du processeur qui exécutera chacune des tâches résultant du partitionnement.

En général, cela peut aussi être un problème d'optimisation si on veut minimiser le temps d'exécution:

Exemple: On a des tâches indépendantes, mais de durée différente.

```
T1: --------------
T2: -----
T3: ----------
```

La répartition de ces tâches sur plusieurs processeurs avec la contrainte qu'ils soient tous également chargé est un problème qui n'est pas simple (optimisation combinatoire)

1.2.3 Ordonnancement

À quel moment chaque processeur peut commencer les tâches dont il a la charge, de sorte à ne pas violer des dépendances, tout en minimisant le temps d'exécution total.

La méthode des temps au plus tôt et au plus tard

C'est une technique de recherche opérationnelle qui s'applique à la résolution de ce type de problème.

On va considérer un exemple didactique, basé sur un graphe de précédence pour lequel on veut résoudre le problème du placement et de l'ordonnancement.

Soit le graphe de tâches suivant: (schéma du cours...) + (tableau du cours header: tache, earlest, latest)

Avec les marges obtenues dans le tableau, on va placer les tâches sur des processeurs. On voit que le processeur P2 est inactif pendant 3 unités de temps...

On constate que pour ce problème, 2 processeurs sont nécessaires et suffisants. On voit que P2 est inactif pendant 3 unités de temps.

$T_{par} = 15$ $T_{seq} = 27$ $S = \frac{27}{15} = 1.8$ Efficacité $= \frac{1.8}{2} = 0.9 = 90$

Ce qui correspond au 3 secondes inactives sur les 30 secondes au total de 2 processeurs: $\frac{3}{30} = 0.1 = 1 - 0.9$

Si on voulait avoir un meilleur speedup, il faudrait repartitionner le problème en tâches différentes. Par exemple $T_7 \rightarrow T'_7$ *et* T''_7

1.3 6.3 Équilibrage de charge

En anglais: **load balancing**

Le but est que tous les processeurs soient occupés pendant la même durée.

Le temps parallèle T_{par} est le temps du processeur le plus lent. On va donc essayer au mieux de répartir le travail de façon équilibrée à travers les processeurs. C'est ce qu'on appelle le load balancing.

On va proposer ci-dessous des approches qui favorisent cet équilibrage...

1.3.1 Métrique de déséquilibrage de charge

Soit T_i le temps cpu du processeur p_i mesuré entre 2 points de synchronisation (tous les processeurs doivent avoir fini leur travail pour passer le point de synchronisation...).

On définit le temps moyen $\mu = \frac{1}{p} \sum_{i=1}^{p} T_i$

Si tous les T_i sont égaux, on a un équilibre parfait. Le déséquilibre se mesurera sur la variation de ces T_i.

La déviation standard n'est pas forcément le meilleur critère....

Les exemples nous suggèrent que la déviation standard des T_i n'est pas la meilleure valeur pour caractériser le déséquilibre. Mieux vaut comparer le temps

du processeur le plus lent à la moyenne.

$$m = max_i T_i$$

$$\Delta = m - \mu$$

$$\Delta \geq 0$$

$\Delta = 0 \Rightarrow$ équilibre parfait et plus Δ augrmente, plus le déséquilibre augmente

On peut aussi le mesurer de façon relative

$$\Delta' = \frac{\Delta}{\mu} = \frac{m}{\mu} - 1$$

Dans la littérature, il y a de nombreuses façon de mesurer le déséquilibrage de charge.

Si le déséquilibre est trop grand, on va vouloir redistribuer les données à travers les processeurs, et on souhaite que suite à ce répartitionnement, on ait:

$$m = \mu$$

mais cela n'est pas une garantie.

On verra qu'il existe des techniques de rééquilibrage qui n'assurent pas ceci.