Chapitre IV : Méthodologie de Parallélisation d'Algorithmes Séquentiels

ISSATSo

2023-2024

Sommaire

Principe de parallélisation

- Pour paralléliser un algo sur une machine // donnée, on procède comme suit :
  1. Partitionner l'algo en tâches (instructions ou groupes d'instructions) : ce partitionnement peut être fait automatiquement par programme ou bien au cas par cas.
  2. Elaborer le graphe de précédence qui permet de définir les contraintes temporelles pour l'exécution des tâches et de rechercher les tâches indépendantes, susceptibles d'être exécutées en parallèle.
  3. Ordonnancer ensuite les tâches sur les procs, en respectant les contraintes de dépendances, ainsi que les contraintes matérielles liées à l'architecture de la machine.

Remarque

- Un même algo peut conduire à plusieurs versions parallèles différentes suivant :
  - l'accès aux données,
  - le découpage en tâches,
  - l'affectation des tâches aux procs,
  - etc.
- Certaines versions seront mieux adaptées à une structure d'ordinateurs.
- Plusieurs facteurs peuvent influencer le choix de la meilleure (au sens performances) version parmi toutes les versions parallèles obtenues pour la machine utilisée (voir plus tard).

Sommaire

Notion de tâche

- Une tâche est une unité de traitement indivisible caractérisée uniquement par son comportement extérieur : entrées, sorties, instructions et temps d'exécution.
- L'étude des dépendances entre les tâches permet de construire un système de précédence qui traduit le parallélisme interne à l'algo.
- Dans un tel système, 2 tâches sont soit liées par une relation de précédence (dans ce cas leur exécution est séquentielle) soit indépendantes et leur exécution peut être effectuée en //.

Notion de granularité

- Plusieurs décompositions différentes sont possibles pour un même algo.
- Exemple : Produit matriciel C=A*B (les matrices sont carrées d'ordre n)

Algorithme

```
Pour i de 1 à n faire
        Pour j de 1 à n faire
                Pour k de 1 à n faire
                        C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

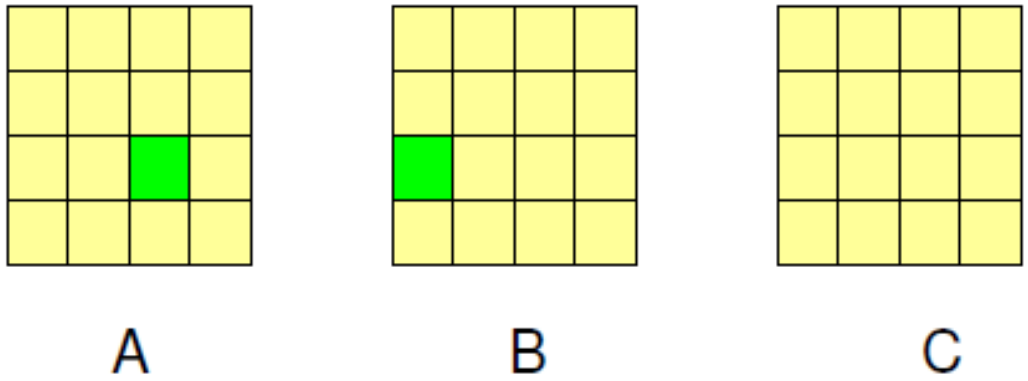
Notion de granularité

- Si nous prenons comme tâches les opérations élémentaires :

Tâche(i,j,k)

C(i,j) = C(i,j) + A(i,k) * B(k,j)

- la granularité de la décomposition est fine.



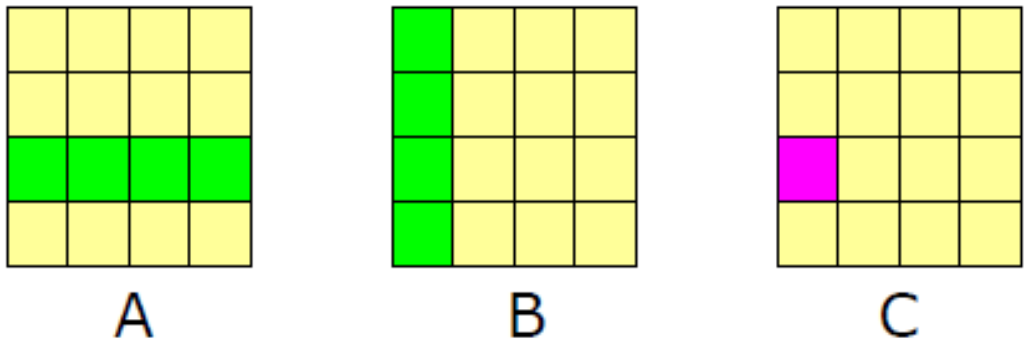
Notion de granularité

- Si chaque tâche est le calcul d'un élément de C :

Tâche(i,j)

Pour k de 1 à n faire
C(i,j) = C(i,j) + A(i,k) * B(k,j)

- la granularité de la décomposition est moyenne.



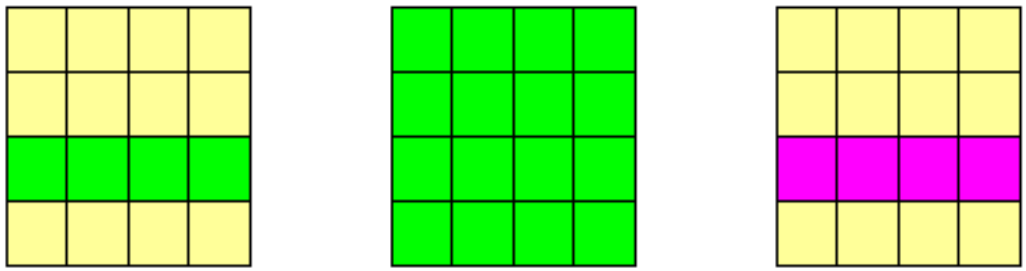
Notion de granularité

- Si chaque tâche correspond au calcul d'une ligne ou d'une colonne du produit :

Tâche(i)

```
Pour j de 1 à n faire
        Pour k de 1 à n faire
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

- la granularité de la décomposition est grossière.



Choix de la décomposition

- Le choix de la meilleure décomposition est un problème difficile lié aux temps d'exécution des tâches et à l'architecture de la machine
- Les principaux facteurs de choix sont :
  - Le nombre de processeurs.
  - Le rapport entre unité de temps de communication et unité de temps de calcul.
  - Les accès mémoire.
  - Etc.

Système de tâches : définitions

- Un système de tâches S=(T_1,. . . ,T_n,<<) est un ensemble de tâches muni d'une relation d'ordre partiel notée <<.
  T_i << T_k ($i \neq k$) signifie que l'exécution de la tâche T_i doit être terminée avant que l'exécution de T_k ne commence.
- 2 tâches T_i et T_k sont indépendantes si elles ne modifient aucune variable commune, sinon elles sont dépendantes.

système de tâches : définitions

- Les tâches T_i et T_k sont consécutives s'il n'existe aucune autre tâche T_j tel que $T_i << T_j << T_k$ ou $T_k << T_j << T_i$.
- Un système de tâches S=(T_1,...,T_n,<<) est un système de précédence si, pour tout couple de tâches (T_i, T_k), une et une seule des 3 conditions suivantes est vérifiée :
  - $T_i << T_k$
  - $T_k << T_i$
  - T_i et T_k sont indépendantes.

Construction du graphe de tâches

Exemple : Considérons un algo composé de 8 tâches $T_1, \ldots, T_8$ dont le système de précédence associé est le suivant :

$T_1 << T_3 ; T_1 << T_4 ; T_1 << T_5 ; T_1 << T_6 ; T_1 << T_7 ; T_1 << T_8$
$T_2 << T_3 ; T_2 << T_4 ; T_2 << T_5 ; T_2 << T_6 ; T_2 << T_7 ; T_2 << T_8$
$T_3 << T_7 ; T_3 << T_8$
$T_4 << T_6 ; T_4 << T_7 ; T_4 << T_8$
$T_5 << T_7$
$T_6 << T_8$

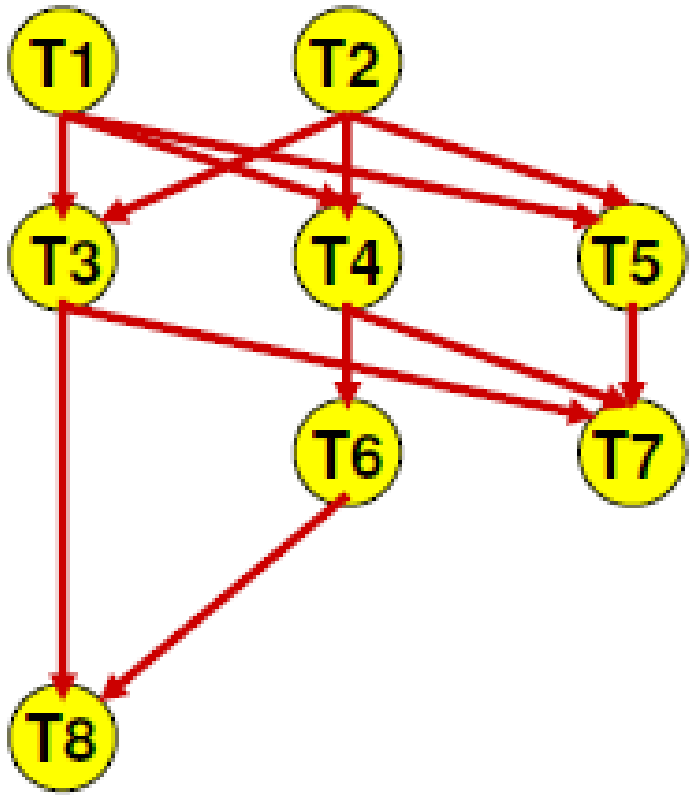
Construction du graphe de tâches

En éliminant les contraintes redondantes, on obtient l'ensemble de relations suivantes :

$T_1 << T_3 ; T_1 << T_4 ; T_1 << T_5$
$T_2 << T_3 ; T_2 << T_4 ; T_2 << T_5$
$T_3 << T_7 ; T_3 << T_8$
$T_4 << T_6 ; T_4 << T_7$
$T_5 << T_7$
$T_6 << T_8$

Construction du graphe de tâches

Le graphe de précédence associé est le suivant :



Sommaire

Problème

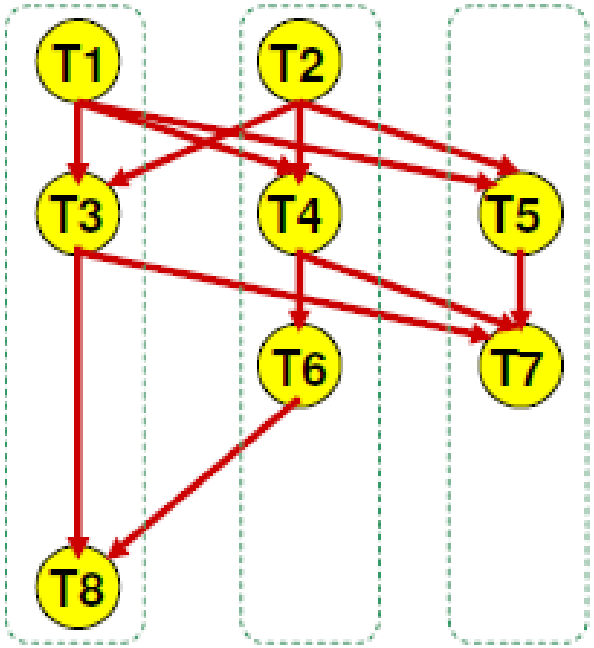
- Dans un contexte informatique, ordonnancer des tâches (sous programmes, processus, fils d'exécution, . . .) revient à affecter à chaque tâche :
  - Une date d'exécution
  - Un processeur(Sous contraintes de ressources ou non)
- Le problème de l'allocation des tâches d'un prog // à un ensemble de procs est très difficile.
- On suppose ici qu'on connaisse les tâches (en particulier leur temps d'exécution ou une estimation de coût) et les relations de précédence qui les lient entre elles.

Hypothèses et objectif

- Les tâches $T_j (j = 1..n)$ sont caractérisées par une durée d'exécution, et éventuellement par une date au plutôt ou une date au plus tard.
- On s'interdit la préemption : possibilité d'interrompre l'exécution d'une tâche commencée.
- Le plus courant objectif à réaliser est de minimiser la fonction qui représente le temps d'exécution total (le makespan).

Exemple d'ordonnancement

Exemple précédent avec p=3 et tâches identiques un premier ordonnancement peut être représenté par le diagramme de Gantt suivant :



Graphe de tâches

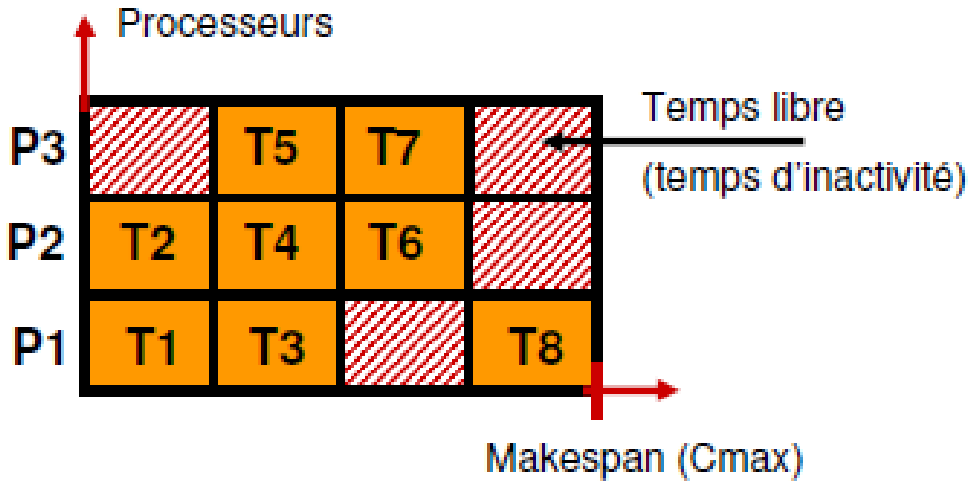


Diagramme de Gantt associé

Sommaire

Complexité des algorithmes parallèles

- La complexité a été connue à être la plus importante mesure de performances d'un algorithme.
- L'analyse de la complexité // permet de mesurer l'efficacité des algos //s et de les comparer.
- Le coût de l'algorithme peut être déterminé comme suit :Coût= temps d'exécution * nombre de processeurs
  - Le temps d'exécution est le temps du dernier processeur
  - Le temps d'exécution sur un processeur est composé de temps de calcul, de temps d'inactivité et de temps de communications

Définitions

- Le problème fondamental de la parallélisation d'une méthode de calcul est de trouver un ordonnancement optimal pour affecter les tâches aux procs en respectant le graphe de précédence, de telle façon que l'algo // s'exécute en temps minimal (que l'on note T_{opt}) sur un nombre infini de procs.
- Dans une phase ultérieure, déterminer le nombre minimum de procs (P_{opt}) nécessaires pour effectuer l'algo en T_{opt}.
- Enfin, étant donné un nombre p fixé de procs, on doit être en mesure de trouver un algo optimal.

Définitions

- Pour concevoir des algorithmes parallèles éfficaces, on doit considérer les règles générales suivantes :
  - Le nombre de processeurs doit être borné par la taille du problème.
  - Le temps d'exécution parallèle doit être considérablement inférieur au temps d'exécution du meilleur algo séquentiel.
  - Le coût de l'algorithme est optimal. (attention : utiliser plus de processeurs peut augmenter le coût de l'algo).

Sommaire

Décompositions du graphe de précédence

- Supposons pour simplifier que toutes les tâches ont le même temps d'exécution (on le prend égal à 1). On parle alors de tâches UET (Unit Execution Time).
- Le temps d'un chemin du graphe de précédence est la somme des temps d'exécution des tâches qui le composent.
- Le plus long chemin du graphe de précédence est celui dont le temps d'exécution est le plus grand.
- La hauteur H(G) du graphe des tâches est le nombre de tâches du plus long chemin.

Décompositions du graphe de précédence

- On appellera décomposition en niveaux du graphe des tâches, l'ensemble :
  $D(G) = \{N_1, \ldots, N_{H(G)}\}$
  qui constitue une partition en H(G) sous-ensembles (niveaux) des sommets de ce graphe vérifiant les conditions suivantes :
  - Le niveau 1 est constitué de tâches n'ayant aucun prédécesseur.
  - Le niveau k est constitué de tâches dont tous les prédécesseurs sont dans des niveaux inférieurs et dont tous les successeurs sont dans des niveaux supérieurs.
  - Le niveau H(G) est constitué de tâches n'ayant aucun successeur.
- Remarque : Un même graphe peut admettre plusieurs décompositions en niveaux.

Décompositions du graphe de précédence

- On appellera décomposition par prédécesseur Dp(G) la décomposition en niveaux suivante :
  - Le niveau 1 est constitué de toutes les tâches n'ayant aucun prédécesseur.
  - Pour k=2 à H(G), le niveau k est constitué des tâches dont tous les prédécesseurs sont dans des niveaux inférieurs et ayant au moins un prédécesseur dans le niveau k-1.
- Cette décomposition est aussi appelée décomposition au plus tôt.
- Exemple :
  Niveau(1) = {T1,T2}
  Niveau(2) = {T3,T4,T5}
  Niveau(3) = {T6,T7}
  Niveau(4) = {T8}

Décompositions du graphe de précédence

- De même, on appellera décomposition par successeur Ds(G) la décomposition en niveaux suivante :
  - Le niveau H(G) est constitué des tâches n'ayant aucun successeur.
  - Pour $k = H(G) - 1$ à 1, le niveau k est constitué des tâches dont tous les successeurs sont dans des niveaux supérieurs et ayant au moins un successeur dans le niveau $k + 1$.
- Cette décomposition est aussi appelée décomposition au plus tard.
- Exemple :
  Niveau(1) = {T1,T2}
  Niveau(2) = {T4}
  Niveau(3) = {T3,T5,T6}
  Niveau(4) = {T7,T8}

Décompositions du graphe de précédence

- Autres décompositions possibles :

Niveau(1) = {T1,T2}	Niveau(1) = {T1,T2}
Niveau(2) ={T3,T4}	Niveau(2) = {T4,T5}
Niveau(3) = {T5,T6}	Niveau(3) = {T3,T6}
Niveau(4) = {T7,T8}	Niveau(4) = {T7,T8}

Définitions

- On appellera largeur d'une décomposition D(G) le maximum des cardinaux des niveaux :
  L(D) = $max(\mid N_k \mid)$, pour k = 1..H(G)
- On appellera en fin largeur L(G) du graphe le min des largeurs des décompositions de G :
  L(G) = $min(L(D))$

T_{opt} et P_{opt} sur les graphes UET

- Théorèmes
  1. Avec un nombre illimité de procs, le temps topt d'un algo // optimal est égal au temps d'exécution du plus long chemin du graphe des tâches, c-à-d H(G) (sans communications).
  2. Le nombre minimal de procs popt permettant de réaliser un algo s'exécutant en temps topt est égal à la largeur L(G) du graphe des tâches.

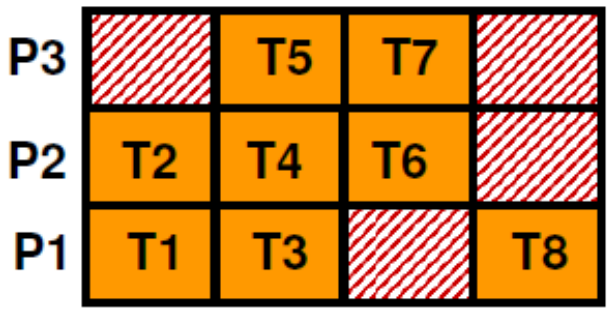
Exemple

- Considérons le graphe de tâches à 8 tâches et supposons que toutes les tâches sont UET.
- Le plus long chemin (T1-T4-T6-T8 ou T2-T4-T6-T8) a un temps d'exécution = 4 et par conséquent $T_{opt} = 4$.
- A chacune des 4 décompositions en niveaux correspond un algo //.

Ordonnancements relatifs aux décompositions

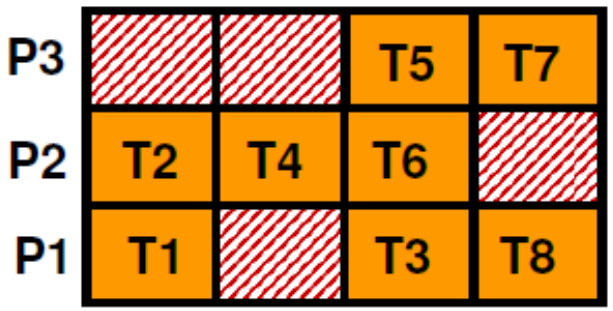
Décomposition par prédécesseur

sa largeur = 3



Décomposition par successeur

sa largeur = 3



Ordonnancements relatifs aux décompositions

Autres décompositions :

largeur = 2

P2	T2	T4	T6	T8
P1	T1	T3	T5	T7

largeur = 2

P2	T2	T5	T6	T8
P1	T1	T4	T3	T7

D'où, $p_{opt} = 2$