leur achèvement respectif avant de pouvoir se terminer.

Les critères qui définissent un bon placement et ordonnancement sont évidemment les performances obtenues. Il faut en particulier diminuer l'overhead du parallélisme et obtenir un bon **équilibre des charges** (ou **load balancing**) sur tous les processeurs.

Le choix d'un bon partitionnement, placement et ordonnancement pour une application donnée est en général un problème d'optimisation combinatoire NP-complet. En d'autres termes, il n'existe pas d'algorithme simple qui le résolve rapidement. Heureusement, la nature de l'application considérée offre souvent des solutions heuristiques acceptables.

6.6 La méthode des temps «au plus tôt» et «au plus tard»

Dans ce paragraphe, nous allons illustrer le problème du placement et de l'ordonnancement par un exemple simple pour lequel les temps de communications inter-processeurs sont négligeables (parallélisation à gros grain). De plus, nous supposerons connu le temps d'exécution de chaque tâche, ce qui correspond à une situation d'ordonnancement statique car ces temps ne changent pas durant l'exécution du programme.

La méthode dite des temps au plus tôt et au plus tard constitue une technique permettant d'analyser le graphe de précédence d'une application et de construire le placement et l'ordonnancement des tâches (qui n'est pas forcément l'optimum absolu).

Pour illustrer cette technique, considérons l'exemple décrit dans la figure 6.4. Les différentes tâches sont notées T_i et les indications à côté de chaque tâche représentent la durée d'exécution. On commence par déterminer le premier moment où une tâche pourra débuter. Cela s'obtient en parcourant le graphe de haut en bas. La tâche T_1 est supposée commencer au temps $t = 0$. Au plus tôt, les tâches T_2 et T_3 pourront débuter au temps $t = 1$, soit dès que T_1 est terminée.

De même, les tâches T_4 et T_5 ne peuvent pas commencer avant que T_2 soit achevée, c'est-à-dire au plus tôt au temps $t = 3$. En poursuivant ce raisonnement, la dernière tâche T_9 ne pourra commencer qu'au temps $t = 14$. Donc, l'exécution optimum de ce programme sera telle que T_9 débutera à $t = 14$. Si T_9 débute plus tard, les performances ne seront pas les meilleures et il faut essayer d'éviter cela. Dans ce but, on cherche maintenant le dernier moment où chaque tâche peut commencer afin de garantir que T_9 démarre effectivement au bon moment. On observe ainsi que T_7 doit commencer au plus tard 7 secondes avant le temps optimal de T_9, soit au plus tard au temps $t = 7$. De même, la tâche T_8, ne durant que 4 secondes, doit débuter au plus tard au temps $t = 14 - 4 = 10$. En remontant de la sorte jusqu'à T_1 on détermine la table des temps « au plus

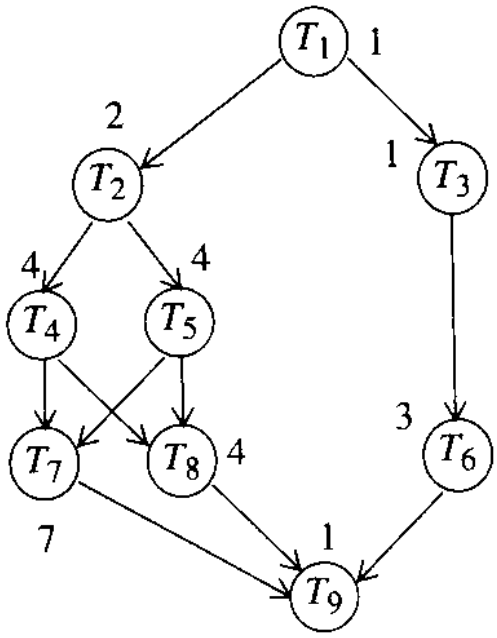

FIGURE 6.4 – *Graphe de précédence d'une application composée de plusieurs tâches (Moldovan, p. 385).*

tôt »et « au plus tard »indiquant le début de chaque tâche, comme le montre le tableau 6.1 (Moldovan, p. 385).

Comme on le voit, certaines de ces tâches ont ces deux temps identiques (T_4, T_7, par exemple), ce qui signifie qu'elles doivent impérativement débuter à un moment précis. Ce temps est d'ailleurs le même pour T_4 et T_5 ce qui indique qu'il faudra au moins deux processeurs si l'on ne désire pas introduire des délais supplémentaires et garder le rythme fixé.

Pour établir le placement final (et l'ordonnancement) correspondant à cette situation, on construit la table 6.2 dont chaque ligne correspond à une unité de temps (ici, on aurait 15 lignes, pour les temps $t = 0$ jusqu'à $t = 14$). Les colonnes correspondent aux processeurs à disposition. On place dans cette table les tâches dont le temps d'exécution est bien défini (earliest_ time=latest_ time), en partant de la première colonne. Si celle-ci est pleine, on remplit la colonne suivante, ce qui signifie qu'un autre processeur devient nécessaire. Une fois que toutes les tâches sont placées (on place en priorité les tâches pour lesquelles la différence latest_time-earliest_ time est minimum), le nombre de colonnes remplies indique le nombre de processeurs nécessaires. Ici, il en faut deux. On constate aussi que le premier processeur est utilisé à 100% alors que l'autre l'est 12 secondes sur 15 (soit 80%). Il y a donc un léger déséquilibre de charge.

Le speedup maximum qu'on peut espérer pour cette situation est facile à ob-

Task	Earliest Time	Latest Time
1	0	0
2	1	1
3	1	10
4	3	3
5	3	3
6	2	11
7	7	7
8	7	10
9	14	14

TABLE 6.1 – *Table des temps au plus tôt et au plus tard pour le graphe de la figure précédente.*

tenir de notre analyse : T_9 termine après 15 secondes avec 2 processeurs (ce qui est l'optimum pour cette décomposition en 9 tâches). Avec un seul processeur, le temps total serait la somme des temps de chaque tâche, soit 27 secondes. Le speedup maximum est donc 27/15=1.8, indépendamment du nombre de processeurs à disposition (cf loi d'Amdahl : le speedup est limité par la quantité de parallélisme à disposition).

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P_1	T_1	T_2	T_2	T_4	T_4	T_4	T_4	T_7	T_7	T_7	T_7	T_7	T_7	T_7	T_9
P_2	idle	T_3	idle	T_5	T_5	T_5	T_5	T_6	T_6	T_6	T_8	T_8	T_8	T_8	idle

TABLE 6.2 – *Placement et ordonnancement des tâches. Deux processeurs suffisent. Les cases marquées "idle" signifie que le processeur concerné est inactif.*

6.7 Equilibrage de charge

Comme on l'a vu déjà plusieurs fois, un déséquilibre de charge entre les processeurs est une cause importante de perte d'efficacité dans une application parallèle. On se souvient (paragraphe 4.2) que le speedup est donné par le **degré de parallélisme moyen**. S'il y a un fort déséquilibre de charge (load unbalance), certains processeurs seront inactifs et le degré de parallélisme diminuera. Une telle situation est illustrée dans la section 6.2, pour le calcul de l'ensemble de Mandelbrot. On constate dans cet exemple que seuls un petits nombre de processeurs travaillent pendant la durée complète de l'exécution. Le travail réalisé par de nombreux processeurs est immédiat et ne contribue pas à apporter un speedup intéressant. Dans une telle situation, il faut trouver des stratégies pour