Ordonnancement et placement de processus

Master 2 Informatique - UFR S.A.T

Pr. Ousmane THIARE

ousmane.thiare@ugb.edu.sn
http://www.ousmanethiare.com/

16 avril 2020

Soient un ensemble de tâches (travaux) interdépendantes et un nombre fixé de processeurs. On veut déterminer l'ordre d'exécution des tâches de façon à ce que l'exécution de l'ensemble soit le plus rapide possible sachant que l'on peut exécuter certaines tâches en parallèle.
Hypothèses applicables à un ordonnancement :

- **H1 - exécution statique :** on connaît l'ensemble de tâches, leurs durées et la structure des graphes de dépendances (c'est-à-dire l'ensemble de couples (T_i, T_j) tel que Ti doit être terminée pour que T_j puisse commencer).

Introduction
Problématique et hypothèses

Soient un ensemble de tâches (travaux) interdépendantes et un nombre fixé de processeurs. On veut déterminer l'ordre d'exécution des tâches de façon à ce que l'exécution de l'ensemble soit le plus rapide possible sachant que l'on peut exécuter certaines tâches en parallèle.
Hypothèses applicables à un ordonnancement :

- **H1 - exécution statique :** on connaît l'ensemble de tâches, leurs durées et la structure des graphes de dépendances (c'est-à-dire l'ensemble de couples (T_i, T_j) tel que Ti doit être terminée pour que T_j puisse commencer).
- **H2 - durée invariante :** la durée d'une tâche est la même quelque soit le contexte dans lequel elle s'exécute.

- **H3 - indivisibilité :** les tâches ne sont pas pré-emptives (non morcelables).

Dans la suite, sauf indication contraire, les hypothèses H2, H3, H7 seront vérifiées.

- **H3 - indivisibilité :** les tâches ne sont pas pré-emptives (non morcelables).
- **H4 - communication immédiate :** il n'y a pas de délai dans les communications. T_j peut commencer dès que T_i a terminé.

Dans la suite, sauf indication contraire, les hypothèses H2, H3, H7 seront vérifiées.

Hypothèses applicables à un ordonnancement :

- **H5 - nombre de processeurs suffisant :** quelque soit l'ordonnancement proposé, on dispose d'assez de processeurs.

Dans la suite, sauf indication contraire, les hypothèses H2, H3, H7 seront vérifiées.

Hypothèses applicables à un ordonnancement :

- **H5 - nombre de processeurs suffisant :** quelque soit l'ordonnancement proposé, on dispose d'assez de processeurs.
- **H6 - absence de priorités :** on ne dispose pas a priori de moyen de fixer de priorités sur les tâches.

Dans la suite, sauf indication contraire, les hypothèses H2, H3, H7 seront vérifiées.

Hypothèses applicables à un ordonnancement :

- **H5 - nombre de processeurs suffisant :** quelque soit l'ordonnancement proposé, on dispose d'assez de processeurs.
- **H6 - absence de priorités :** on ne dispose pas a priori de moyen de fixer de priorités sur les tâches.
- **H7 - ressources suffisantes :** les tâches ne sont jamais bloquées par absence de ressources (disque, mémoire, ...) et les processeurs sont suffisamment puissants pour les supporter.

Dans la suite, sauf indication contraire, les hypothèses H2, H3, H7 seront vérifiées.

Hypothèses applicables à un ordonnancement :

- **H5 - nombre de processeurs suffisant :** quelque soit l'ordonnancement proposé, on dispose d'assez de processeurs.
- **H6 - absence de priorités :** on ne dispose pas a priori de moyen de fixer de priorités sur les tâches.
- **H7 - ressources suffisantes :** les tâches ne sont jamais bloquées par absence de ressources (disque, mémoire, ...) et les processeurs sont suffisamment puissants pour les supporter.
- **H8 -** ...

Dans la suite, sauf indication contraire, les hypothèses H2, H3, H7 seront vérifiées.

Soit :

- un ensemble de tâches $\{T_1, \cdots, T_n\}$ et un ensemble de durées d'exécution :$\{ex(T_1), \cdots, ex(T_n)\}$ sans communication

On appelle *graphe de précédence* un graphe dans lequel

Soit :

- un ensemble de tâches $\{T_1, \cdots, T_n\}$ et un ensemble de durées d'exécution :$\{ex(T_1), \cdots, ex(T_n)\}$ sans communication
- et une relation de précédence ? telle que $T_i \ll T_j$ si T_i doit être terminée pour que T_j puisse commencer

On appelle *graphe de précédence* un graphe dans lequel

Soit :

- un ensemble de tâches $\{T_1, \cdots, T_n\}$ et un ensemble de durées d'exécution :$\{ex(T_1), \cdots, ex(T_n)\}$ sans communication
- et une relation de précédence ? telle que $T_i \ll T_j$ si T_i doit être terminée pour que T_j puisse commencer

On appelle *graphe de précédence* un graphe dans lequel

- les noeuds représentent les tâches ;

Soit :

- un ensemble de tâches $\{T_1, \cdots, T_n\}$ et un ensemble de durées d'exécution :$\{ex(T_1), \cdots, ex(T_n)\}$ sans communication
- et une relation de précédence ? telle que $T_i \ll T_j$ si T_i doit être terminée pour que T_j puisse commencer

On appelle *graphe de précédence* un graphe dans lequel

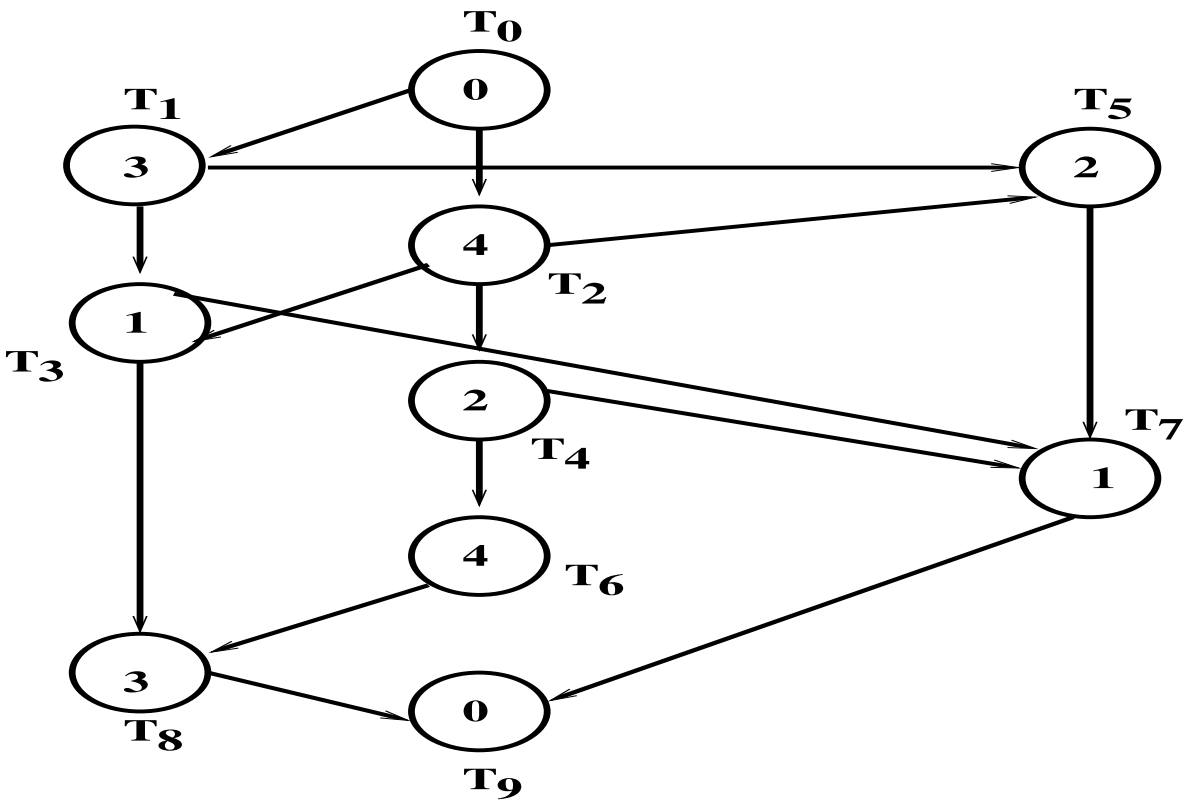
- les noeuds représentent les tâches ;
- deux tâches fictives T_0 dite tâche initiale, et T_{n+1} dite tâche finale, de durée nulle sont ajoutées ;

Soit :

- un ensemble de tâches $\{T_1, \cdots, T_n\}$ et un ensemble de durées d'exécution :$\{ex(T_1), \cdots, ex(T_n)\}$ sans communication
- et une relation de précédence ? telle que $T_i \ll T_j$ si T_i doit être terminée pour que T_j puisse commencer

On appelle *graphe de précédence* un graphe dans lequel

- les noeuds représentent les tâches ;
- deux tâches fictives T_0 dite tâche initiale, et T_{n+1} dite tâche finale, de durée nulle sont ajoutées ;
- les noeuds portent la durée de la tâche dont ils sont issus.



Un ordonnancement sur p processeurs est défini comme une application Ord de $\{T_1, \cdots, T_n\}$ vers $(N, [1, \cdots, p])$ associant à chaque T_i le couple $(debut(T_i), processeur(T_i))$ où $debut(T_i)$ est la date de début de T_i et $processeur(T_i)$ le processeur qui lui est affecté telle que :

- si $T_i \ll T_j$ alors $debut(T_j) - debut(T_i) \geq ex(T_i)$

La condition 2 assure que deux tâches ne peuvent se dérouler en même temps sur un même processeur.
On notera plus simplement t_i la date $debut(T_i)$, appelée aussi *potentiel*.

Un ordonnancement sur p processeurs est défini comme une application Ord de $\{T_1, \cdots, T_n\}$ vers $(N, [1, \cdots, p])$ associant à chaque T_i le couple $(debut(T_i), processeur(T_i))$ où $debut(T_i)$ est la date de début de T_i et $processeur(T_i)$ le processeur qui lui est affecté telle que :

- si $T_i \ll T_j$ alors $debut(T_j) - debut(T_i) \geq ex(T_i)$
- si $processeur(T_j) = processeur(T_i)$ alors $debut(T_i) + ex(T_i) \leq debut(T_j)$ ou $debut(T_j) + ex(T_j) \leq debut(T_i)$

La condition 2 assure que deux tâches ne peuvent se dérouler en même temps sur un même processeur.
On notera plus simplement t_i la date $debut(T_i)$, appelée aussi *potentiel*.

Sur l'exemple précédent, calculons la date minimale avant qu'une tache puisse s'exécuter

▷$(t_1 = 0, p_1)$ et $(t_2 = 0, p_2)$ car rien ne précède ces tâches

▷$t_3 =$? Or

$$\begin{cases} t_1 \ll t3 \quad \Leftrightarrow \quad t_1 + ex(T_1) \leq t_3 \\ \text{et} \\ t_2 \ll t_3 \quad \Leftrightarrow \quad t_2 + ex(T_2) \leq t_3 \end{cases} \tag{1}$$

d'où $t_3 = max(t_1 + ex(T_1), t_2 + ex(T_2))$

▷$t_8 = max(t_3 + ex(T_3), t_6 + ex(T_6)) =$

$max(max(t_1 + ex(T_1), t_2 + ex(T_2)), t_6 + ex(T_6)) = \cdots = 10$

L'**algorithme de Bellmann** met en œuvre ce calcul :
$t_0 = 0$; marquer T_0
Tant qu'il existe des sommets non marqués faire
soit T_j un sommet non marqué dont tous les prédécesseurs T_k sont marqués (il en existe au moins un sinon c'est qu'il y a un cycle dans le graphe) alors
$t_j = max\{t_k + ex(T_k)\}$
marquer T_j
finSoit
FinTantque
On remarque sur l'exemple, que T_1 peut commencer à t = 1 sans que cela modifie le temps d'exécution de l'application.
Par contre, dès que sa date de début est supérieure à 1s, l'ensemble de l'application prend du retard.
A noter que pour $T_{n+1=9}$ on obtient $t_9 = 13s$

Chemin critique La durée minimale de l'application est alors la valeur maximale des chemins menant de T_0 à T_{n+1}. On l'appelle *chemin critique*.
Dans l'exemple $\Rightarrow$ $T_0, T_2, T_4, T_6, T_8, T_9$ pour 13s (qui est la date de début de T_9).
Date au plus tôt / au plus tard On appellera :

- date au plus tôt t_i pour T_i, la valeur maximale de tous les chemins menant de T_0 à T_i (La date au plus tôt t_i pour T_i est calculée de façon évidente par l'algorithme de Bellmann).

Chemin critique La durée minimale de l'application est alors la valeur maximale des chemins menant de T_0 à T_{n+1}. On l'appelle *chemin critique*.
Dans l'exemple $\Rightarrow$ $T_0, T_2, T_4, T_6, T_8, T_9$ pour 13s (qui est la date de début de T_9).
Date au plus tôt / au plus tard On appellera :

- date au plus tôt t_i pour T_i, la valeur maximale de tous les chemins menant de T_0 à T_i (La date au plus tôt t_i pour T_i est calculée de façon évidente par l'algorithme de Bellmann).
- date au plus tard di pour T_i, t_{n+1} - la valeur maximale de tous les chemins menant de T_i à T_{n+1}

En effet, la valeur d'un chemin menant de T_i à T_{n+1} est le temps que mettra au minimum la branche correspondante à s'exécuter. Donc pour que toutes les branches aient le temps de s'exécuter avant la date au plus tôt de T_{n+1} (qui est la valeur minimale d'exécution : donc si T_{n+1} ne débute pas à cet instant, l'application ne se sera pas déroulée de façon optimale) il est évident que T_i doit commencer au plus tard à t_{n+1} - le maximum de ces temps.
Dans l'exemple

i	1	2	3	4	5	6	7	8	9
Dtard$_i$	1	0	9	4	10	6	12	10	13

On peut aisément montrer que pour les tâches T_i du chemin critique : $t_i = d_i$

Durée totale d'exécution d'un système de tâches La durée d'exécution totale d'exécution d'un système de taches est le temps écoulé entre le début de T_0 (tâche initiale) et la fin de T_{n+1}(tâche finale)

Durée moyenne d'exécution d'une tache dans un système de tâches La durée moyenne d'exécution d'une tache dans un système de tâches est la moyenne des temps d'exécution de chaque tâche.

Optimalité / minimalité d'un ordonnancement Un ordonnancement O est minimal si quelque soit le nombre de processeurs, il n'existe pas d'autre ordonnancement dont la durée totale d'exécution soit inférieure à celle de O. Tout algorithme qui propose un ordonnancement tel que la durée d'exécution du système soit égal à la durée d'exécution du chemin critique est minimal.

Un ordonnancement O est optimal si pour un nombre donné de processeurs, il n'existe pas d'autre ordonnancement dont la durée totale d'exécution soit inférieure à celle de O.
En effet, il peut ne pas exister de solution minimal. Par exemple, si la somme des durées des taches divisée par le nombre de processeurs est supérieure à la durée du chemin critique, alors il ne peut pas exister de solution minimale.
Le but d'un algorithme d'ordonnancement sera donc de trouver pour tout système de taches, un ordonnancement minimal si possible , optimal sinon.
Il devra aussi chercher à minimiser le temps d'exécution moyen des taches.

Cas d'une exécution statique (H1) sans communication (H4)

Cas H5 : on dispose d'assez de processeurs

Si l'on dispose d'un nombre suffisant de processeurs, tout ordonnancement tel que : $\forall\, i,\ t_i \leq debut(T_i) \leq d_i$ est minimal.
Ainsi, si l'on dispose d'assez de processeurs, il suffira d'en allouer un "au hasard" à la tâche T_i dès que la date au plus tôt t_i est atteinte mais au plus tard avant que la date au plus tard di soit atteinte, pour que l'ordonnancement soit minimal.

Cas d'une exécution statique (H1) sans communication (H4)

Cas NON-H5 : on ne dispose pas d'assez de processeurs

On supprime alors souvent l'hypothèse H6 en introduisant des priorités entre les tâches et en gérant une liste des tâches exécutables (c'est-à-dire celles dont la date au plus tôt est déjà passée sans que la tâche est débutée).

Première solution dans le cas général

On applique l'algorithme précédent, puis lorsque l'on doit affecter un processeur qui se libère, on l'affecte à une tâche exécutable suivant l'ordre de priorité.

Si il n'existe pas de priorités fixées à priori, on peut prendre dans l'ordre

- les tâches dont les dates au plus tard sont les plus dépassées (ce sont donc les tâches les plus en retard)

Cas d'une exécution statique (H1) sans communication (H4)

Cas NON-H5 : on ne dispose pas d'assez de processeurs

On supprime alors souvent l'hypothèse H6 en introduisant des priorités entre les tâches et en gérant une liste des tâches exécutables (c'est-à-dire celles dont la date au plus tôt est déjà passée sans que la tâche est débutée).

Première solution dans le cas général

On applique l'algorithme précédent, puis lorsque l'on doit affecter un processeur qui se libère, on l'affecte à une tâche exécutable suivant l'ordre de priorité.

Si il n'existe pas de priorités fixées à priori, on peut prendre dans l'ordre

- les tâches dont les dates au plus tard sont les plus dépassées (ce sont donc les tâches les plus en retard)
- celles dont les dates au plus tard sont les plus proches

Cas d'une exécution statique (H1) sans communication (H4)

Cas NON-H5 : on ne dispose pas d'assez de processeurs

On supprime alors souvent l'hypothèse H6 en introduisant des priorités entre les tâches et en gérant une liste des tâches exécutables (c'est-à-dire celles dont la date au plus tôt est déjà passée sans que la tâche est débutée).

Première solution dans le cas général

On applique l'algorithme précédent, puis lorsque l'on doit affecter un processeur qui se libère, on l'affecte à une tâche exécutable suivant l'ordre de priorité.

Si il n'existe pas de priorités fixées à priori, on peut prendre dans l'ordre

- les tâches dont les dates au plus tard sont les plus dépassées (ce sont donc les tâches les plus en retard)
- celles dont les dates au plus tard sont les plus proches
- et enfin celles dont les dates au plus tôt sont les plus dépassées

Cas d'une exécution statique (H1) sans communication (H4)

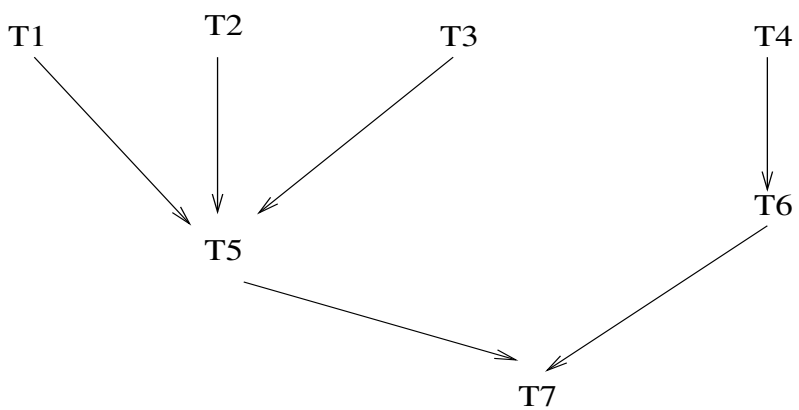
Cas NON-H5 : on ne dispose pas d'assez de processeurs

Malheureusement, cet algorithme ne fourni pas toujours l'ordonnancement optimal (en fait, on peut montrer que le calcul de l'ordonnancement optimal est NP-complet).
Existe-il des cas particuliers d'algorithme optimal ?

Cas d'une anti-arborescence

Le graphe est tel que chaque tâche n'a qu'un seul successeur.

Exemple :



Cas d'une exécution statique (H1) sans communication (H4)

Cas NON-H5 : on ne dispose pas d'assez de processeurs

Cas d'une anti-arborescence

On peut montrer que dans ce cas, quelque soit le nombre de processeurs, un ordonnancement suivant la date au plus tard est optimal.

Ainsi, dans l'exemple, si chaque tâche à la même durée : T1, T2, T3, T4, T5, T6, T7 est optimal. Mais aussi T2, T4, T3, T1, T6, T5, T7.

De plus, la construction de cette liste est en O(n).

Cas d'une exécution statique (H1) sans communication (H4)

Cas NON-H5 : on ne dispose pas d'assez de processeurs

Cas d'un graphe quelconque et 2 processeurs

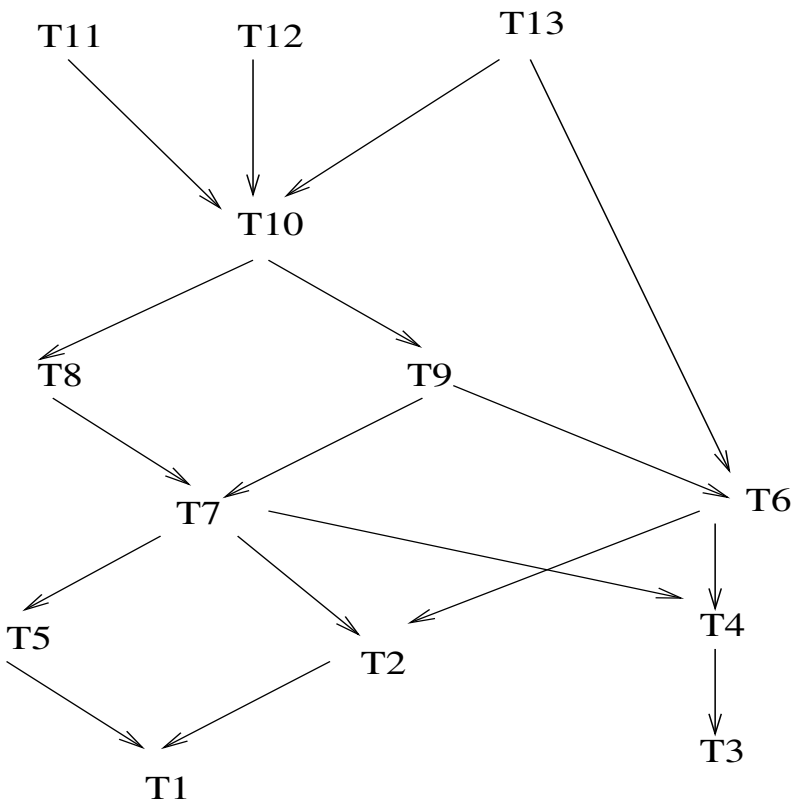
On classe les tâches suivant leurs dates au plus tard.
Puis pour deux tâches ayant la même date au plus tard, on applique la règle suivante :
Si l'ensemble des successeurs de T_i est strictement inclus dans l'ensemble des successeurs de T_j alors T_j doit être plus prioritaire que T_i pour qu'on puisse exécuter les successeurs de T_j qui ne sont pas successeurs de T_i

Cas d'une exécution statique (H1) sans communication (H4)

Cas NON-H5 : on ne dispose pas d'assez de processeurs

Cas d'un graphe quelconque et 2 processeurs

Exemple : On suppose les tâches de durée unitaire (uniquement pour simplifier la représentation)



Cas d'une exécution statique (H1) sans communication (H4)

Cas NON-H5 : on ne dispose pas d'assez de processeurs

Cas d'un graphe quelconque et 2 processeurs

Ainsi, par exemple, T6 et T7 sont de même "niveau". Comme $succ(T6) \subset succ(T7)$, T7 sera plus prioritaire. Intuitivement, en exécutant T7 avant T6, on a plus de chance de permettre la poursuite des tâches dépendantes de T7 qui ne dépendent pas de T6 : exemple T5.

Cas d'une exécution statique (H1) sans communication (H4)

Cas NON-H5 : on ne dispose pas d'assez de processeurs

Cas d'un graphe quelconque et 2 processeurs

L'**algorithme de Coffman et Graham**, dit aussi algorithme d'étiquetage fournit une liste de priorité qui respecte la règle précédente et qui de plus tient compte des priorités des successeurs. Les durées de tâches n'ont pas d'importance.

Choisir une tâche terminale Ti : ET[Ti] = 1

Pour k=2 à N faire soit $S = \{TE1, TE2, \cdots, TEp\}$ l'ensemble de tâches étiquetables /* c'est-à-dire celles qui n'ont pas de successeurs ou celles dont tous les successeurs sont étiquetés*/

Pour chaque TEi de S faire

Calculer la liste L(TEi) = liste ordonnée décroissante des étiquettes des successeurs de TEi

fait

Cas d'une exécution statique (H1) sans communication (H4)

Cas NON-H5 : on ne dispose pas d'assez de processeurs

Cas d'un graphe quelconque et 2 processeurs

L'**algorithme de Coffman et Graham**

Déterminer TEm telles que L(TEm) soit inférieure ou égale à tous les L(TEi) par ordre lexicographique

ET[TEm] = k

fait

Cet algorithme fournit une liste des tâches par priorité croissante.

Il suffit alors d'ordonnancer en utilisant cette priorité et alors l'ordonnancement est optimal.

Exécution statique (H1) avec communication (Non-H4)

Hypothèse

La communication entre tâches n'a lieu qu'à la fin de la tâche émettrice pour le début de la tâche réceptrice.
Ainsi, à la relation de précédence (voir "système de tâches") est associée une relation de communication : chaque arc sur le graphe de précédence sera remplacé par un arc de communication .
On parelra alors de *graphe de communication*.

Exécution statique (H1) avec communication (Non-H4)

Communication et ordonnancement

On introduit la fonction de communication $C(T_i, T_j)$ qui donne le temps de communication entre la tâche T_i et la tâche T_j :

$$C(T_i, T_j) = \begin{cases} c_{i,j} \text{ si } processeur(T_i) \neq processeur(Tj) \\ 0 \text{ si } processeur(T_i) = processeur(T_j) \end{cases} \quad (2)$$

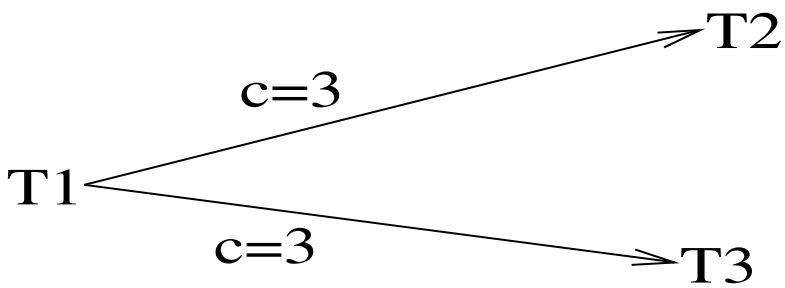
En fait, $C(T_i, T_j)$ vaut $c_{i,j}$ (donné par le réseau) si T_i et T_j ne sont pas sur le même processeur, 0 sinon (on considère qu'ils communiquent par mémoire partagée et que cela est "instantané").

La relation de précédence devient :

$$(T_i \rightarrow T_j) \Rightarrow \begin{cases} deb(T_j) \geq deb(T_i) + ex(T_i) \text{ si } proc(T_i) = proc \\ deb(T_j) \geq deb(T_i) + ex(T_i) + ci,j \text{ si } proc(T_i) \neq \end{cases}$$

En fait, $C(T_i, T_j)$ vaut $c_{i,j}$ (donné par le réseau) si T_i et T_j ne sont pas sur le même processeur, 0 sinon (on considère qu'ils communiquent par mémoire partagée et que cela est "instantane ?").

D'où, une solution consiste à essayer de mettre sur un même processeur les tâches qui communiquent entre elle Mais cela ne permet pas d'optimiser une situation comme par exemple, celle-ci :



En effet, a priori, on ne peut lancer T_2 et T_3 en même temps. En effet, c'est soit T_3 qui est sur le même processeur que T_1 et alors c'est T_2 doit attendre la communication 3 secondes, soit réciproquement c'est T_3 qui doit attendre.

Exécution statique (H1) avec communication (Non-H4)

Communication et ordonnancement

Une solution consiste à dupliquer T_1 : sur deux processeurs différents (par exemple p_1 et p_2) on lance T_1, puis dès qu'elle se termine, on peut lancer T_2 sur un de ces deux processeurs (p_1 par exemple) et T_3 sur l'autre (p_2 dans ce cas).
Comme il n'existe pas d'algorithme satisfaisant dans le cas où les tâches ne sont pas duplicables, nous supposerons sdans la suite qu'elles le sont.

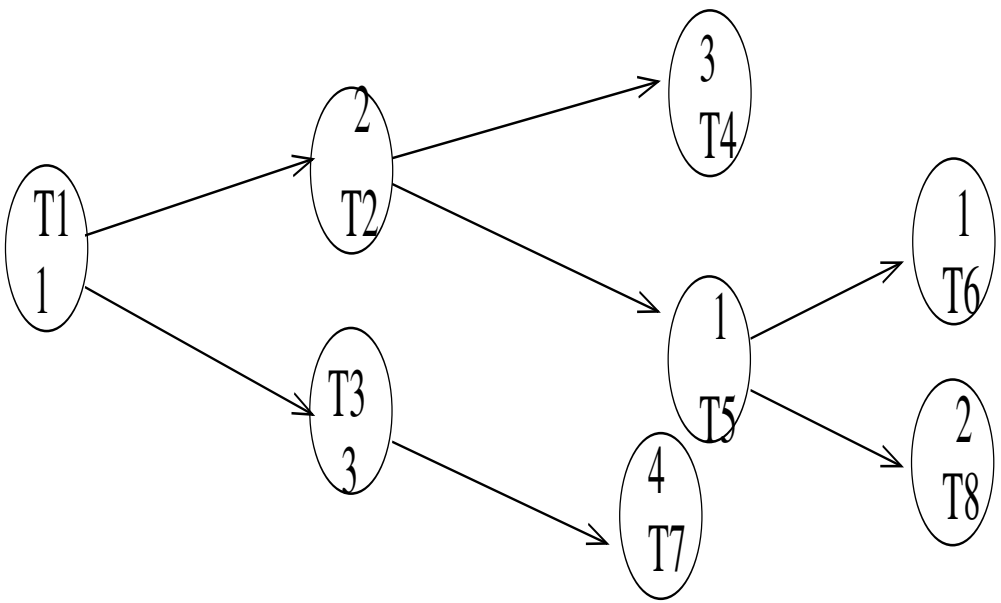
Exécution statique (H1) avec communication (Non-H4)

Cas d'un nombre de processeurs suffisant (H5)

Cas d'un arbre de précédence.

Il est possible, ainsi, comme chaque tâche ne possède qu'un prédécesseur, d'associer un processeur à chaque feuille et de faire exécuter sans délai, le chemin menant de la racine à la feuille.

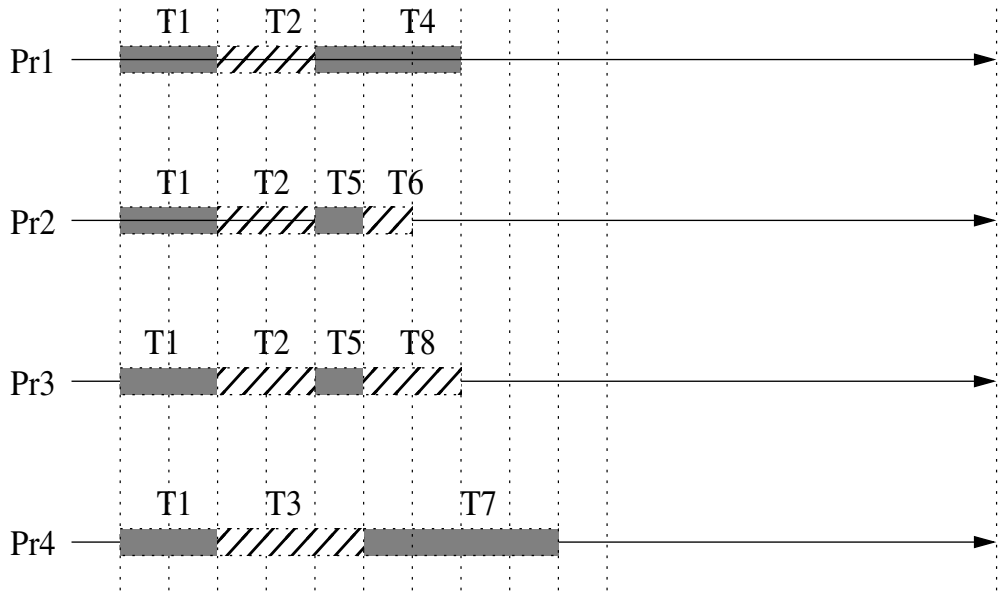
Exemple :



Exécution statique (H1) avec communication (Non-H4)

Cas d'un nombre de processeurs suffisant (H5)

Cas d'un arbre de précédence.
nous donne :



Exécution statique (H1) avec communication (Non-H4)

Cas d'un nombre de processeurs suffisant (H5)

Cas d'un graphe quelconque

Idée : transformer le graphe en un arbre Pour chaque tâche T_i , on détermine le chemin critique menant à celle-ci. Mais, une seule tâche T_j, prédécesseur de T_i peut être mise sur le même processeur que T_i.

Question : laquelle ? *Réponse :* Celle qui retarderait le plus T_i.

D'où l'algorithme :

Cas d'un graphe quelconque

Pour chaque tâche T_i dont les chemins critiques de tous les prédécesseurs sont déterminés

- Si T_i n'a aucun prédécesseur, on considère que T_i peut être mis sur n'importe lequel des processeurs, $s = \emptyset$

Exécution statique (H1) avec communication (Non-H4)

Cas d'un nombre de processeurs suffisant (H5)

Cas d'un graphe quelconque

Pour chaque tâche T_i dont les chemins critiques de tous les prédécesseurs sont déterminés

- Si T_i n'a aucun prédécesseur, on considère que T_i peut être mis sur n'importe lequel des processeurs, $s = \emptyset$
- Si T_i n'a qu'un prédécesseur T_k , on considère que T_i sera mis le même processeur que son prédécesseur et on calcule la date au plus tôt avec un temps de communication nul, s = k

Cas d'un graphe quelconque

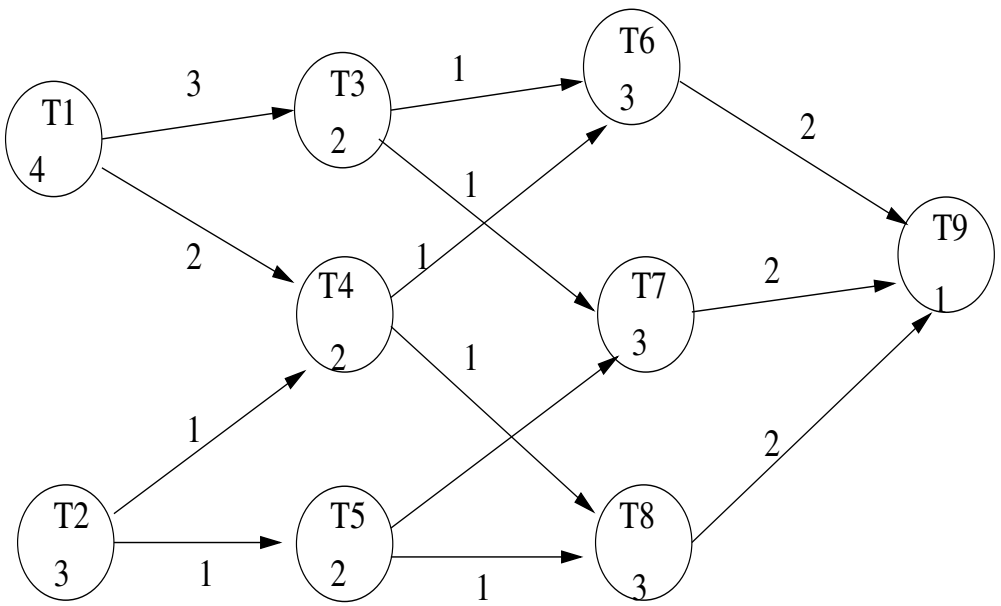
Pour chaque tâche T_i dont les chemins critiques de tous les prédécesseurs sont déterminés

- Si T_i n'a aucun prédécesseur, on considère que T_i peut être mis sur n'importe lequel des processeurs, $s = \emptyset$
- Si T_i n'a qu'un prédécesseur T_k , on considère que T_i sera mis le même processeur que son prédécesseur et on calcule la date au plus tôt avec un temps de communication nul, s = k
- Si T_i a plus d'un prédécesseur
  - 1. on calcule tous les chemins menant à T_i en considérant que toutes les tâches sont sur un processeur différent
  - 2. soit Ts_m , la tâche prédécesseur de T_i tel que $T_0 \rightarrow \cdots \rightarrow \cdots T_{s_m} \rightarrow T_i$ soit le chemin critique
  - 3. on choisira alors de mettre T_i et T_{s_m} sur le même processeur, $s = s_m$

Cas d'un graphe quelconque

- 4. on peut alors recalculer le chemin critique (qui peut donc avoir diminué) qui est alors la date au plus tôt de T_i.

Exemple :



Exécution statique (H1) avec communication (Non-H4)

Cas d'un nombre de processeurs suffisant (H5)

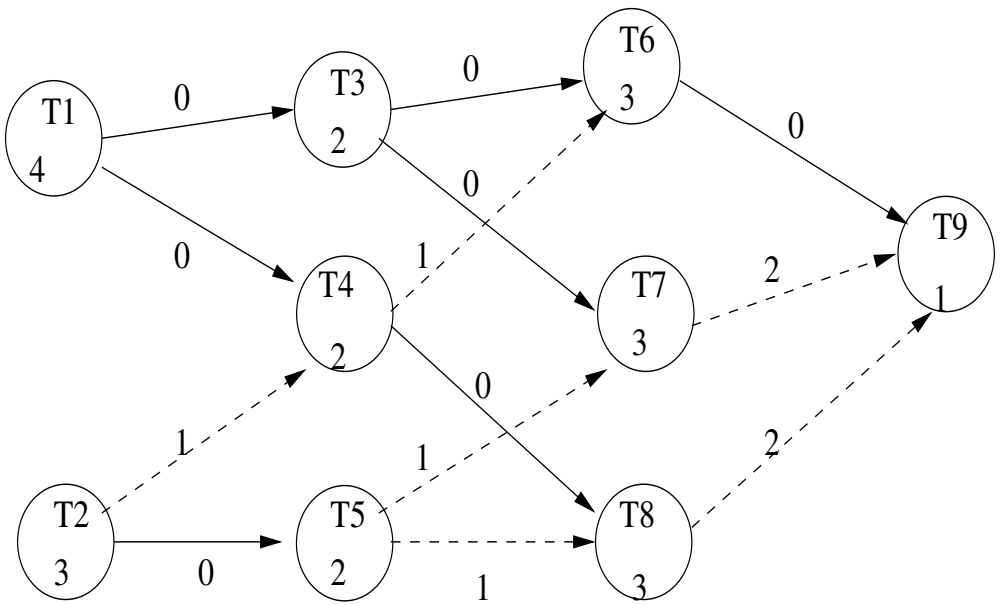
Cas d'un graphe quelconque

donne :

Tache	1	2	3	4	5	6	7	8	9
D_i	0	0	4	4	3	7	6	6	11
s	-	-	1	1	2	3	3	4	6

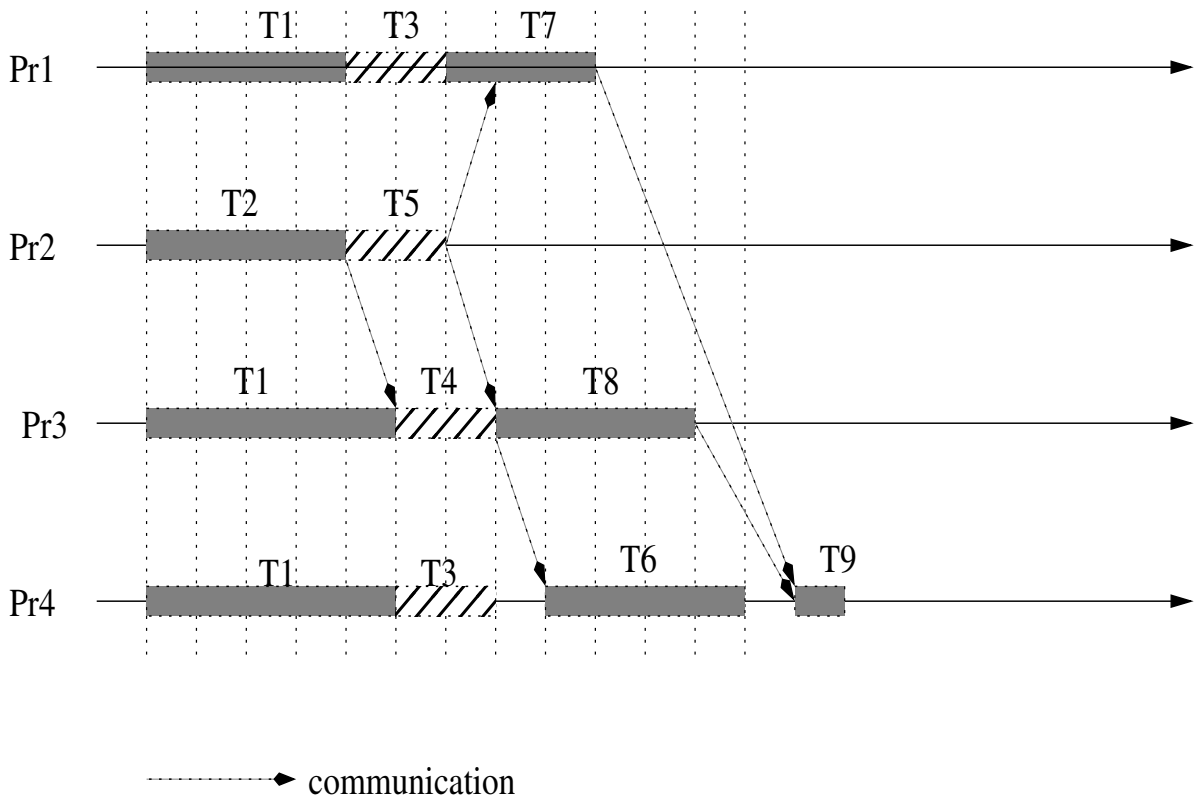
Cas d'un graphe quelconque

D'où l'arbre des chemins critiques (en gras)



Cas d'un graphe quelconque

D'où l'ordonnancement :



On peut noter que cet algorithme donne une allocation sur 4 processeurs alors que la largeur du graphe est de 3

Exécution statique (H1) avec communication (Non-H4)

Cas d'un nombre de processeurs insuffisant (Non-H5)

Pas d'algorithme satisfaisant :

- on fait l'algorithme précédent

Exécution statique (H1) avec communication (Non-H4)

Cas d'un nombre de processeurs insuffisant (Non-H5)

Pas d'algorithme satisfaisant :

- on fait l'algorithme précédent
- on éclate l'arbre obtenu en autant de sous arbres qu'il y a de processeurs

Exécution statique (H1) avec communication (Non-H4)

Cas d'un nombre de processeurs insuffisant (Non-H5)

Pas d'algorithme satisfaisant :

- on fait l'algorithme précédent
- on éclate l'arbre obtenu en autant de sous arbres qu'il y a de processeurs
- on ordonnance sur ces processeurs en gérant des listes de tâches exécutables

Cas d'une exécution sans information préalable (NON-H1)

Il n'existe à l'heure actuelle aucun algorithme d'ordonnancement a priori satisfaisant et du plus, très souvent les processus doivent être préemptifs pour être ordonnancés. En général, l'ordonnancement se fait de façon éclatée et dynamique : chaque site gère ses processus locaux par des algorithmes classiques mono-processeur.

Cas d'une exécution sans information préalable (NON-H1)

Néanmoins, un partage des processus entre les processeurs peut être faite soit lors de la création d'un processus (il faut lui trou- ver le meilleur processeur pour son exécution) soit dynamique (on essaie d'équilibrer dynamiquement la charge des processeurs : *load balancing*)
Dans ce deuxième cas, une hypothèse supplémentaire doit être faite : les processus doivent pouvoir migrer en cours d'exécution. C'est-à-dire, le système ou le processus lui-même peuvent changer le processeur affecté à un processus au cours de son exécution.

Cas d'une exécution sans information préalable (NON-H1)

Soient les hypothèses supplémentaires suivantes :

- **HS1** quelque soit le site où il s'exécute, un processus accède toujours aux ressources (au sens physique) dont il a besoin.

Cas d'une exécution sans information préalable (NON-H1)

Soient les hypothèses supplémentaires suivantes :

- **HS1** quelque soit le site où il s'exécute, un processus accède toujours aux ressources (au sens physique) dont il a besoin.
- **HS2** un site est capable d'estimer sa charge (en nombre de processus et/ou en quantité de ressources utilisées ...) ;

Cas d'une exécution sans information préalable (NON-H1)

Soient les hypothèses supplémentaires suivantes :

- **HS1** quelque soit le site où il s'exécute, un processus accède toujours aux ressources (au sens physique) dont il a besoin.
- **HS2** un site est capable d'estimer sa charge (en nombre de processus et/ou en quantité de ressources utilisées ...) ;
- **HS3** un site est capable d'évaluer les besoins d'un processus.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

Vaut-il mieux ordonnancer les processus sur l'ensemble des processeurs dans une seule file d'attente (approche centralisée) ou au contraire, pré-affecter les processus à des machines avec une file d'attente par machine (approche répartie) ?
Imaginons que sur une station de travail S

- les utilisateurs engendrent des demandes aléatoires de travail : soit λ le débit d'entrée de ces demandes ;

Il est évident que si $\mu < \lambda$, la file d'attente va augmenter à l'infini. Donc, on peut admettre que $\mu < \lambda$, mais uniquement sur un court laps de temps.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

Vaut-il mieux ordonnancer les processus sur l'ensemble des processeurs dans une seule file d'attente (approche centralisée) ou au contraire, pré-affecter les processus à des machines avec une file d'attente par machine (approche répartie) ?
Imaginons que sur une station de travail S

- les utilisateurs engendrent des demandes aléatoires de travail : soit λ le débit d'entrée de ces demandes ;
- la station est capable de traiter ces demandes avec un flux de sortie de μ.

Il est évident que si $\mu < \lambda$, la file d'attente va augmenter à l'infini. Donc, on peut admettre que $\mu < \lambda$, mais uniquement sur un court laps de temps.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

On peut alors montrer que le temps moyen d'attente entre une demande et sa réponse est : $T = \frac{1}{\mu - \lambda}$

Par exemple, un système de fichiers peut traiter $\mu = 50$ demandes/s, alors que la file de demandes est de $\lambda = 40$ demandes/s. Le temps moyen de traitement est alors $T_i = \frac{1}{10}s$

Remarque lorsque λ s'approche de 0 (charge nulle), le temps moyen ne devient pas nul : en effet, si le serveur peut traiter 50 demandes/s, c'est qu'une demande dure en moyenne $\frac{1}{50}s$ soit 20ms. Donc, même si $\lambda = 1, T = 20ms$.

On dispose de N stations de puissance S_i équivalentes de puissance $\mu_i = \mu$. Supposons que la demande globale λ_g soit égale $\lambda_g = N.\lambda$.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

▷ Supposons que λ_g soit répartie sur les N stations : chaque station a donc $\lambda_i = \lambda = \frac{1}{N}\lambda_g$ demandes. La durée moyenne de traitement d'une station est donc

$$T_i = \frac{1}{\mu_i - \lambda_i} = \frac{1}{\mu - \lambda}$$

La moyenne des temps d'attente est donc

$$T = \frac{1}{N}\sum_i T_i = \frac{1}{N}.(N.\frac{1}{\mu - \lambda}) = \frac{1}{\mu - \lambda}$$

▷ Que ce passe-t-il si pour la même demande $\lambda_g = N.\lambda$ on utilise un serveur de puissance $\mu_g = N.\mu$? Le temps d'attente est alors $T = \frac{1}{\mu_g - \lambda_g} = \frac{1}{N.\mu - N.\lambda} = \frac{1}{N}\frac{1}{\mu - \lambda}$.

D'où un temps moyen par rapport à N serveurs divisé par N.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

Mais :

- en général, le prix d'un serveur de puissance par exemple 10 000 mips est largement supérieur à celui de 10 stations de 1000 mips ;

D'où l'idée intermédiaire de "simuler" un serveur centralisé en regroupant les stations de travail : *grappe de processeurs*. Ainsi, la puissance théorique est $N.\mu$.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

Mais :

- en général, le prix d'un serveur de puissance par exemple 10 000 mips est largement supérieur à celui de 10 stations de 1000 mips ;
- la tolérance au pannes est meilleure sur 10 stations car la "perte" d'une station n'est pas un catastrophe. Or les coûts de maintenance d'un serveur centralisé peuvent être prohibitifs

D'où l'idée intermédiaire de "simuler" un serveur centralisé en regroupant les stations de travail : *grappe de processeurs*. Ainsi, la puissance théorique est $N.\mu$.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

Mais :

- en général, le prix d'un serveur de puissance par exemple 10 000 mips est largement supérieur à celui de 10 stations de 1000 mips ;
- la tolérance au pannes est meilleure sur 10 stations car la "perte" d'une station n'est pas un catastrophe. Or les coûts de maintenance d'un serveur centralisé peuvent être prohibitifs
- e temps calculé n'est qu'un temps moyen : si un gros processus P_1 prend 95% du temps CPU (par exemple 0.95s), il bloque les 5% restant (par exemple, P_2= 0.05s) : le propriétaire de P_2 va attendre 1s ! !

D'où l'idée intermédiaire de "simuler" un serveur centralisé en regroupant les stations de travail : *grappe de processeurs*. Ainsi, la puissance théorique est $N.\mu$.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

Malheureusement, cette idée n'est valable que si :

- le réseau ne pénalise pas trop les échanges (temps de chargement, algorithme d'ordonnancement, surveillance de la charge, ...)

En conclusion Le choix entre stations indépendantes, station en grappe et serveur centralisé dépend de la nature de la charge. Si tous les utilisateurs font des choses simples (mail, éditons de texte, projet étudiant, etc.) une station de travail individuelle de bas de gamme est suffisant. A l'opposé, s'ils utilisent principalement de gros programmes peu parallèlisés, la solution serveur centralisé s'impose.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

Malheureusement, cette idée n'est valable que si :

- le réseau ne pénalise pas trop les échanges (temps de chargement, algorithme d'ordonnancement, surveillance de la charge, ...)
- l'application est N-parallèlisable : en effet, si l'application ne peut être divisée qu'en par exemple 5 parties indépendantes, sur 10 processeurs, la moitié de ces processeurs ne fera rien.

En conclusion Le choix entre stations indépendantes, station en grappe et serveur centralisé dépend de la nature de la charge. Si tous les utilisateurs font des choses simples (mail, éditons de texte, projet étudiant, etc.) une station de travail individuelle de bas de gamme est suffisant. A l'opposé, s'ils utilisent principalement de gros programmes peu parallèlisés, la solution serveur centralisé s'impose.

Cas d'une exécution sans information préalable (NON-H1)

Grappe de processeurs

En fait, dans la réalité, très souvent, on se trouve dans une situation intermédiaire dans laquelle les utilisateurs font majoritairement des choses simples mais où ponctuellement, ils utilisent de gros logiciels (atelier de génie logiciel, calcul intensifs) de plus en plus souvent fortement parallélisé soit dans le code directement soit par une implantation multi-processus (clients-serveurs). L'utilisation de grappes de processeurs semble la plus adéquate dans cette situation.

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement sans migration

Algorithme centralisé

Principe Un coordinateur gère une table d'utilisation comportant une entrée par utilisateur (ou application).
Ce coordinateur cherche a donner à chaque utilisateur ou application une part équitable de la puissance de calcul.
On suppose que chaque application ou utilisateur se voit affecter une machine particulière.
Ainsi, quand un processus doit être créé et lorsque la machine hôte de l'utilisateur décide que ce processus ne peut lui être affecté, elle demande au coordinateur de lui trouver un autre processeur :

- il en existe un libre et non demandé : affectation de ce processeur et FIN

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement sans migration

Algorithme centralisé

Principe Un coordinateur gère une table d'utilisation comportant une entrée par utilisateur (ou application).
Ce coordinateur cherche a donner à chaque utilisateur ou application une part équitable de la puissance de calcul.
On suppose que chaque application ou utilisateur se voit affecter une machine particulière.
Ainsi, quand un processus doit être créé et lorsque la machine hôte de l'utilisateur décide que ce processus ne peut lui être affecté, elle demande au coordinateur de lui trouver un autre processeur :

- il en existe un libre et non demandé : affectation de ce processeur et FIN
- sinon : demande enregistrée mais rejetée

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement sans migration

Algorithme centralisé

Principe Quand un utilisateur exécute des processus sur d'autres machines que la sienne, il accumule des points de pénalité à chaque seconde. Par contre, lorsqu'une de ses demandes n'est pas satisfaite, on lui enlève des points de pénalité à chaque seconde. De même lorsqu'il n'utilise aucun processeur, mais avec une limite inférieure donnée.

Ainsi, lorsque sa note de pénalité est positive : l'utilisateur "sur-exploite" le système. Dans le cas contraire (<0) il a besoin de ressources.

D'où, lorsqu'un processeur se libère, il est affecté à l'utilisateur en attente ayant la note minimale.

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement sans migration

Algorithme distribué

Trois approches lors du choix du processeur pour une tâche :

- au hasard : la machine qui ne peut accepter la tâche, tire au hasard une autre machine et lui envoie le processus à créer. Si cette machine est elle-même saturée, elle fait la même chose.
  $\rightarrow$ l'algorithme se déroule jusqu'à ce qu'une machine accepte le processus ou que celui-ci ait fait un nombre de sauts maximum (dans ce cas, il s'exécute là où il se trouve)

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement sans migration

Algorithme distribué

- sondage par tirage au sort : la machine qui ne peut accepter la tâche, tire au hasard une autre machine et lui envoie un message de sondage en lui demandant si sa charge est inférieure à un seuil donné. Si oui, elle lui envoie le processus, sinon, elle envoie la sonde à une autre machine.

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement sans migration

Algorithme distribué

Si au bout de S sondages, aucune machine n'accepte le processus, alors celui-ci s'exécute sur la machine d'origine Cet algorithme est stable et satisfaisant quelque soit le seuil fixé, le coût de transfert et le nombre de processus et de machines.

- sondage précis : la machine qui ne peut accepter la tâche, envoie message de sondage a toutes les autres machines en leurs demandant si leurs charges. Elle choisit la "meilleure".
  Problème : le temps de sondage peut être tel que les valeurs reçues par la machine ne soit plus à jour $\rightarrow$ la machine choisie n'est peut-être plus la meilleure ou pire elle est elle-même saturée (elle recommencera le sondage !).

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement avec migration

En fait pour être efficace, cet algorithme nécessite une complexité importante.

Définition On appelle *migration* d'un processus le transfert de tout ou d'une partie du contexte (utilisateur : adressage interne ; d'état : horloge, CO ; et/ou système : droit d'accès, propriétaire, ...) d'un processus d'un site à un autre.

Algorithme centralisé

Il est évident que l'algorithme centralisé précédent s'applique sans grande difficulté. La principale différence est que la décision d'ordonnancement est prise après chaque événement d'ordonnancement (processeur demandé, processeur libéré, interruption, blocage d'un processus...)

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement avec migration

Algorithme réparti

Deux types d'algorithmes :

- par appel d'offre : un site surchargé émet vers les autres une demande précisant les besoins (mémoire, ressources, ...) des processus dont il veut se décharger (algo distribué précédent) ;

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement avec migration

Algorithme réparti

Deux types d'algorithmes :

- par appel d'offre : un site surchargé émet vers les autres une demande précisant les besoins (mémoire, ressources, ...) des processus dont il veut se décharger (algo distribué précédent) ;
- par recrutement : un site sous-chargé demande aux autres sites s'ils ont besoin d'aide

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement avec migration

Algorithme réparti

Exemple d'un **algorithme de recrutement** : l'algorithme de Ni, Xu et Gendreau (1985).

On définit un taux de besoin $B_{i,k}$ d'un processus P_k sur un site S_i en fonction :

- des besoins du processus

Chaque site maintient :

- son état à trois niveaux : LC (légèrement chargé), NC (normalement chargé), FC (fortement chargé)

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement avec migration

Algorithme réparti

Exemple d'un **algorithme de recrutement** : l'algorithme de Ni, Xu et Gendreau (1985).

On définit un taux de besoin $B_{i,k}$ d'un processus P_k sur un site S_i en fonction :

- des besoins du processus
- de sa priorité à migrer (dépendant par exemple de sa priorité dans l'application)

Chaque site maintient :

- son état à trois niveaux : LC (légèrement chargé), NC (normalement chargé), FC (fortement chargé)
- une table des charges qui contient le dernier état connu de chacun des autres sites

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement avec migration

Fonctionnement

Un site S_i dans l'état LC émet un message "prêt" à tous les sites F_i dans l'état FC (donnés par la table des charges) et attend leur réponse.

Un site F_j qui reçoit un message "prêt" de S_i, calcule les taux de besoin B_{j,k_j} de chacun de ses k_j processus susceptibles de migrer et les envoie à S_i.

Lorsque S_i a reçu tous les B_{j,k_j} de tous les sites, il calcule un taux de besoin standard B_s égal à la moyenne de tous les B_{j,k_j}

Ce taux B_s est alors transmis au site F_j qui a répondu le $B_{i,k}$ le plus grand.

Le site F_j détermine alors, parmi ses processus, le processus P_l tel que $B_{j,l}$ soit maximal sur F_j et $B_{j,l}$ soit supérieur à B_s.

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement avec migration

Fonctionnement

Si il existe un tel processus : il enclenche la migration sinon il répond "Trop tard" à S_i.

Quelques problèmes et limitations d'un tel algorithme :

- tables de états à maintenir : est-ce les sites qui diffusent leur nouvel état à chaque changement ou est-ce chaque site qui émet vers les autres sites une demande d'état ?

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement avec migration

Fonctionnement

Si il existe un tel processus : il enclenche la migration sinon il répond "Trop tard" à S_i.

Quelques problèmes et limitations d'un tel algorithme :

- tables de états à maintenir : est-ce les sites qui diffusent leur nouvel état à chaque changement ou est-ce chaque site qui émet vers les autres sites une demande d'état ?
- aucune tolérance face à la perte de messages ;

Cas d'une exécution sans information préalable (NON-H1)

Ordonnancement avec migration

Fonctionnement

Si il existe un tel processus : il enclenche la migration sinon il répond "Trop tard" à S_i.

Quelques problèmes et limitations d'un tel algorithme :

- tables de états à maintenir : est-ce les sites qui diffusent leur nouvel état à chaque changement ou est-ce chaque site qui émet vers les autres sites une demande d'état ?
- aucune tolérance face à la perte de messages ;
- un des problèmes principaux est la migration et l'augmentation simultanée de la charge de Si (par création de processus locaux ou par l'arrivée d'autres processus en migration) ce qui peut amener une oscillation des processus entre les différents sites